Connaissez-vous les autres ouvrages PSI sur l'Apple IIGS et le langage C ?

– **Clefs pour Apple IIGS** - Nicole Bréaud-Pouliquen.

Ce mémento s'adresse aux programmeurs en assembleur, C et Basic de l'Apple IIGS : il offre une synthèse des spécificités du matériel et des logiciels de développement : architecture interne, brochages, jeu d'instructions du 65816, mémoires, ressources graphiques, entrées-sorties. Le système CPW est décrit en détail ; l'ensemble des outils du bureau est répertorié, fonction par fonction.

– **Clefs pour C** - François Piette.

Ce livre s'adresse à tous ceux qui veulent écrire des programmes simples et performants en C standard. Après présentation complète et détaillée du langage, l'auteur traite en profondeur de 2 versions très répandues de C :
Le Lattice C et le Digital Research C. De nombreux exemples de programmes illustrent les différentes notions abordées.

– **C et ses fichiers** - Jacques Boisgontier et Jean-Pierre Lagrange.

Ce livre destiné aux utilisateurs avertis de ce langage rappelle les principales notions et fonctions de C ainsi qu'un des aspects méconnus de ce langage : la gestion de fichiers. Les fichiers à accès direct et à accès séquentiel abordés, la dernière partie est consacrée aux méthodes pratiques : méthodes simples, accès indexé et base de données.

Jean-Pierre Curcio est ingénieur statisticien économiste dans une grande banque. Il travaille dans le service « Techniques Avancées / Télécom » qui s'occupe entre autres de toute la partie micro-informatique. Par ailleurs il est collaborateur régulier aux revues INFOMAG et DÉCISION INFORMATIQUE (série Passeport pour Mac).

SOMMAIRE

AVERTISSEMENT

Tous les exemples figurant dans cet ouvrage ont été testés avec la version alpha du compilateur croisé Megamax sur Macintosh.

Certains problèmes peuvent apparaître lors d'une compilation dans un autre environnement de travail :

• Les structures de taille variable *Cursor*, *AlertTemplate* et *DialogTemplate*, notamment, peuvent poser problème. Pour les curseurs, on utilisera la déclaration non structurée (vue dans plusieurs exemples). Pour les alertes et les dialogues, on pourra utiliser l'artifice suivant au moment de la déclaration :

```
#define maxDT 8                          /* 8 items maximum */

struct _DialogTemplate {
    Rect    BoundsRect;
    int     Visible;
    long    RefCon;
    Pointer Items[maxDT+1];
    };
#define DialogTemplate struct _DialogTemplate
```

La constante maxDT est définie avant la déclaration et donne le nombre maximal d'items de tous les dialogues de l'application.

• Quand une structure contient un pointeur sur chaîne de caractères, nous avons très souvent déclaré la chaîne au milieu de l'initialisation de la structure, laissant le compilateur séparer la chaîne et remplacer par le pointeur. Certains environnements n'acceptent pas cette écriture, d'autres génèrent un code faux ! On appliquera donc plutôt l'écriture de la page 125, où la chaîne est d'abord initialisée, puis un pointeur passé dans la structure.

DISQUETTE D'ACCOMPAGNEMENT

L'auteur verse dans le domaine public tous les exemples (et non les sources des programmes) présentés dans cet ouvrage.

Dans cette disquette (librement copiable) figureront les conditions d'obtention des sources sur support magnétique, compilables soit dans l'environnement APWC (disquette Apple IIGS), soit dans l'environnement compilateur croisé Megamax (disquette Macintosh).

Vous pouvez vous les procurer en envoyant une disquette et une enveloppe timbrée à votre nom et adresse :

Monsieur J.P. Curcio
11, avenue Albert-Camus
77400 Lagny

INTRODUCTION

L'Apple IIGS est un Apple II : il suffit d'essayer de compter les anciens programmes Apple II tournant sur la nouvelle machine pour s'en persuader. En conséquence, tout ce qui a déjà été écrit sur l'Apple II (ou du moins une grande partie) reste valide, et peut continuer à être lu et employé.

Mais l'Apple IIGS est plus qu'un simple Apple II. L'interface utilisateur Apple *(desktop user interface),* apparu avec le Lisa, extraordinaire machine disparue prématurément, sans doute morte d'être née trop tôt, glorifié par le Macintosh qui lui doit son phénoménal succès, plagié sans vergogne par les constructeurs concurrents qui en ont compris tout l'intérêt commercial, y fait son apparition.

L'interface utilisateur a deux facettes. La remarquable facilité qu'elle apporte à l'utilisateur final du fait même de son universalité est due à un ensemble de règles auxquelles doit impérativement se plier le programmeur. Plus de facilité pour l'utilisateur signifie en l'occurrence moins de liberté pour le développeur. C'est à ce prix que la portée générale de tels concepts est possible.

Le programmeur doit changer ses habitudes. L'époque où l'on réinventait la roue à chaque nouvelle application est révolue. Maintenant, Apple met à notre disposition plusieurs centaines de sous-programmes qu'il faut apprendre à utiliser, ou du moins dont il faut connaître l'existence au cas où on aurait à les utiliser. Un programme consistera en une suite d'appels à ces sous-programmes, constituant ce que nous appellerons *Toolbox* ou boîte à outils, entrecoupés de sous-programmes propres à l'application.

Une fois l'apprentissage effectué (c'est long pour certains, plus rapide pour d'autres, mais cela vaut vraiment la peine de ne pas abandonner en cours de route), le programmeur saura rapidement constituer un squelette d'application qu'il pourra utiliser « à toutes les sauces » : l'investissement sera devenu rentable. *Une fois les bases acquises, il est beaucoup plus rapide de développer une application respectant l'interface et utilisant sa boîte à outils qu'un programme classique.* L'apprentissage est

sans doute plus rapide sur l'Apple IIGS que sur le Macintosh : la boîte à outils du dernier-né a en effet profité de l'expérience de celles de ses aînés, et certaines facilités qui n'existaient pas sont apparues sur la nouvelle machine. En première approche, le programmeur pourra presque ignorer le fonctionnement de certains gestionnaires, grâce à la création de la procédure **TaskMaster,** qui rend subitement toute simple la programmation de la boucle d'événements. L'importance de cette procédure est telle, que nous lui consacrerons tout un chapitre, à la fin de l'étude des gestionnaires, avec un exemple complet d'application. Grâce à **TaskMaster** et quelques autres routines à connaître, on en vient rapidement à consacrer ses efforts de programmation sur les caractéristiques de son application, et non plus sur son environnement d'utilisation.

La *Toolbox* de l'Apple IIGS ressemble fortement à celle du Macintosh, avec deux différences notables : l'Apple II possède un affichage couleur alors que le Macintosh (des premières générations avant les ROM 256Ko) est noir et blanc en standard, le Macintosh possède une notion de « ressources » (gérées par le *resource manager*) que l'Apple IIGS ignore complètement. Les programmes écrits pour l'une ou l'autre des deux machines devront tenir compte (entre autres choses) de ces deux différences fondamentales.

Remarque La gestion de la couleur est totalement différente sur le GS et sur les nouveaux Macintoshs. On constatera rapidement que le choix du langage de programmation utilisé est beaucoup moins crucial qu'avant. Pour programmer en Apple IIGS comme pour programmer en Macintosh, on a besoin d'un environnement de programmation permettant l'appel aisé de ces programmes constituant la boîte à outils. Bien entendu, un assembleur restera plus rapide qu'un C lui-même plus rapide qu'un Pascal et plus rapide qu'un Basic. Mais l'important, c'est la possibilité d'appeler – et d'appeler efficacement – les programmes de la *Toolbox*. Le critère principal de choix sera donc l'environnement, et à environnement égal, la rapidité ou les goûts personnels.

Au moment où nous écrivons ces lignes, *aucun* environnement de programmation définitif n'est disponible. Celui d'Apple (appelé *programmer's workshop*) est en cours de développement (il inclura un assembleur, un C signé Megamax, un Pascal signé TML et différents autres langages, y compris des langages orientés objet) ; de même, l'environnement proprement C de Megamax. **Les exemples que nous donnons ont été écrits sur Macintosh, compilés et linkés sur Macintosh grâce à un compilateur croisé,** toujours signé Megamax. Gageons que comme sur Macintosh un grand choix d'environnements sera possible dans un avenir proche, et que tous respecteront les formats fixés par Apple en ce qui concerne les codes objets (résultant d'une compilation), ce qui permettra de linker ensemble des modules provenant de différents environnements et de différents langages pour obtenir une application Apple IIGS qui tourne encore !

Le présent ouvrage se veut une introduction aux concepts de la programmation d'applications tournant sur Apple IIGS, conformes à l'interface utilisateur Apple. La plupart des programmes liés à cet interface sont évoqués, certains plus longuement que d'autres. Le but n'est pas de se substituer à la documentation technique fournie par Apple aux développeurs, mais d'avoir une approche différente, peut-être plus illustrée, et surtout *en français*. Nous espérons qu'il aidera le programmeur en herbe à se familiariser avec les concepts (nouveaux s'il n'a jamais tâté de la programmation en Macintosh) qu'il faut obligatoirement maîtriser pour écrire un vrai programme Apple IIGS.

Le langage choisi pour assurer l'interface avec la boîte à outils est le langage C. Ce langage a la réputation d'être compliqué à utiliser et difficile à relire. On lui reproche parfois d'avoir les inconvénients de la programmation structurée sans en présenter les avantages, etc. A notre avis, C présente le meilleur rapport performance/facilité d'emploi du marché. Grâce à ses primitives permettant la programmation structurée (et donc l'absence de l'instruction GOTO), on peut faire des programmes extrêmement lisibles : au développeur de ne pas embrouiller à souhait le lecteur potentiel par

l'emploi de structures tarabiscotées ou de pointeurs à tort et à travers. Plus permissif que Pascal, il sera donc moins bavard. Certains opérateurs permettant la manipulation au niveau du bit, il évitera au maximum l'utilisation de l'assembleur. Enfin, la possibilité de compilation séparée de n'importe quelle fonction permettra la constitution de véritables bibliothèques de programmes.

Pour lire cet ouvrage, il est nécessaire de connaître les rudiments du C, et non d'être un champion de sa manipulation. Pour le novice, la lecture préalable de n'importe quel ouvrage d'initiation au C sera amplement suffisante.

L'ouvrage se veut le plus possible indépendant de l'environnement de programmation (et pour cause, aucun n'est vraiment disponible au moment où nous écrivons ces lignes, comme nous l'avons déjà dit). Cela présente l'avantage de l'ouvrir à tout le monde (et même à ceux qui programment dans un langage autre que le C), au prix quelquefois d'adaptations plus ou moins légères.

Par exemple, nous verrons dans cet ouvrage la définition de structures, telle que celle du rectangle :

```
struct _Rect {
    int  top;          /* coordonnée verticale du coin supérieur gauche */
    int  left;         /* coordonnée horizontale du coin supérieur gauche */
    int  bottom;       /* coordonnée verticale du coin supérieur droit */
    int  right;        /* coordonnée horizontale du coin supérieur droit */
    };
#define Rect struct _Rect
```

Dans cette définition, le rectangle est une structure comportant quatre entiers. Un rectangle sera manipulé par l'intermédiaire d'un pointeur, ainsi que le montrent les lignes suivantes :

```
Rect    r;              /* r est déclaré comme un rectangle */

FrameRect(&r);          /* un pointeur sur rectangle est utilisé */
```

Notons au passage l'emploi du style *italique* pour désigner les champs de structures et certains autres termes définis dans l'environnement de développement (et dont ils peuvent dépendre), et l'emploi du style **gras** pour désigner les sous-programmes appartenant à la *Toolbox*.

Une autre façon de définir le rectangle suit, sans utilisation de structure. On notera la différence dans la syntaxe, même si le résultat est parfaitement équivalent à celui de l'exemple précédent.

```
int  r[4];              /* chaîne de quatre entiers */
    /* r[0] = coordonnée verticale du coin supérieur gauche */
    /* r[1] = coordonnée horizontale du coin supérieur gauche */
    /* r[2] = coordonnée verticale du coin inférieur droit */
    /* r[3] = coordonnée horizontale du coin inférieur droit */

FrameRect(r);           /* un pointeur sur chaîne est utilisé */
```

Dans le premier cas, on utilisait l'opérateur & pour prendre l'adresse d'une structure. Cette opération est implicite dans le second cas.

Et si nous avions déclaré la structure de type rectangle de la façon suivante :

```
struct {
    int top;          /* coordonnée verticale du coin supérieur gauche */
    int left;         /* coordonnée horizontale du coin supérieur gauche */
    int bottom;       /* coordonnée verticale du coin supérieur droit */
    int right;        /* coordonnée horizontale du coin supérieur droit */
    } Rect;
```

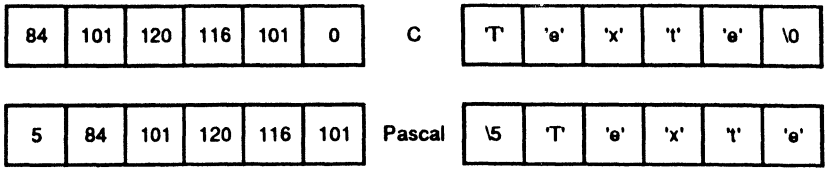
il aurait fallu répéter le mot *struct* dans chaque déclaration ultérieure :

```
struct Rect r;          /* r est déclaré comme un rectangle */

FrameRect(&r);          /* un pointeur sur rectangle est utilisé */
```

Cet exemple est donné simplement pour montrer qu'il est souvent facile d'adapter la syntaxe de certains appels à son propre environnement de travail.

Tous les appels *Toolbox* sont décrits en employant une nomenclature empruntée au Pascal : alors que le C standard ne connaît que la notion de fonction (tout sous-programme réserve de la place pour une valeur en retour), Pascal distingue les notions de fonction et de procédure : une *fonction* retourne une valeur, une *procédure* non, et ne réserve même pas de place sur la pile. Les sous-programmes de la boîte à outils agissent à la manière Pascal. De même, l'ordre dans lequel les arguments sont passés sur la pile et la façon dont une fonction retourne sur une valeur diffèrent entre C et Pascal, mais l'environnement de développement rend cette différence transparente au programmeur. Les curieux constateront que dans les fichiers *headers* (ceux qui se terminent par .h) du *Workshop* C d'Apple développé par Megamax, tous les appels *Toolbox* sont définis avec la directive *pascal,* et les environnements sérieux en C se devront de proposer cette directive, qui permet à une fonction C de se comporter comme un sous-programme (fonction ou procédure) Pascal. Le débutant n'utilisera pas les *glue routines* et autres fonctions filtres, autrement dit les fonctions qui modifient l'action d'un appel *Toolbox,* mais très vite on voudra adapter à ses propres besoins un dialogue modal ou la procédure d'action d'une barre de défilement, par exemple, et une fonction filtre sera indispensable. Cette routine devra obligatoirement agir comme si elle était écrite dans un environnement Pascal (en ce qui concerne toutes les valeurs qui transitent par la pile), et la directive *pascal* sera employée pour que la fonction C simule ce comportement.

Le Pascal est encore présent dans la *Toolbox* en ce qui concerne les chaînes de caractères. Sauf QuickDraw qui connaît à la fois les chaînes C et Pascal, tous les *managers* utilisent les chaînes de type Pascal. Une chaîne de caractères est constituée d'un nombre variable de caractères significatifs compris entre 0 et 255. Chaque caractère est codé (suivant la table ASCII) sur un octet, et un octet supplémentaire sert à déterminer la longueur de la chaîne. Ainsi, une chaîne de type C sera terminée par le caractère nul (un octet à zéro), qui signifiera fin de chaîne. Par contre, Pascal utilise en tête de chaîne un octet dont la valeur donne le nombre de caractères significatifs qu'elle contient. Par suite, seule la chaîne vide de caractères a la même définition en C et en Pascal : elle est constituée d'un seul octet, qui contient la valeur zéro. Dès qu'une chaîne n'est plus vide, la différence apparaît. La chaîne « Texte » sera ainsi codée, suivant le langage utilisé (chaque carré représente un octet) :



Un environnement C digne de ce nom fournira des fonctions permettant de passer de l'une à l'autre des représentations, puisque les managers ne connaissent que la forme Pascal : un titre de fenêtre, un article dans un menu déroulant seront des chaînes Pascal. On pourra toutefois les écrire directement, au niveau des déclarations de variables :

```
char    pstring[ ] = "\32Ceci est une chaîne Pascal";
```

Rappelons qu'on peut inclure en C un octet de valeur quelconque dans une chaîne de caractères, en utilisant la forme \xxx où xxx est la valeur de l'octet en base 8 (octal). L'instruction précédente créera une chaîne un peu particulière, puisque le premier octet contiendra la longueur de la chaîne (32 en base 8 égale 26 caractères), ce qui lui permettra d'être utilisée par les managers. Mais le compilateur C lui ajoutera tout de même l'octet nul de fin de chaîne C. Certains environnements, encore plus sympathiques, permettront la directive \P qui nous évitera de compter les caractères de la chaîne et de faire la conversion octale : le compilateur s'en chargera tout seul :

```
char    pstring[ ] = "\PCeci est une chaîne Pascal";
```

A vous de vérifier les possibilités de votre environnement de travail.

Il est même possible que certains environnements fassent des conversions Pascal/C de manière interne, par utilisation de *glue routines.* Par exemple, nous verrons une fonction, **GetWTitle,** qui retourne un pointeur sur le titre d'une fenêtre donnée. Ce pointeur pointe sur une chaîne Pascal, puisque c'est ainsi qu'un titre est géré par la *Toolbox.* Grâce à un tour de passe-passe, certains compilateurs peuvent vous en faire une chaîne C, sans que vous n'ayez rien demandé ! C'est intéressant si vous voulez manipuler ce titre avec des outils de la bibliothèque C (concat, par exemple), mais désastreux si vous destiniez ce pointeur à servir d'argument à une autre routine de la *Toolbox* ! (voir un tel exemple dans le chapitre consacré au Menu Manager).

Une autre subtilité liée au microprocesseur 65816 est à prendre en considération. Quand une valeur est codée sur deux octets (c'est le cas des entiers), nous l'écrirons en C sous la forme hexadécimale OxPPMM, où PP désigne l'octet le plus significatif et MM l'octet le moins significatif. Pour que le 65816 (à l'instar de son ancêtre le 6502) puisse comprendre cette valeur, elle sera codée MMPP en langage machine (les octets sont inversés). Dans cet ouvrage, les octets seront décrits dans l'ordre où on les écrit en C, et non tels qu'ils sont transcrits en mémoire (ceci est particulièrement important au niveau de la définition des couleurs, composante par composante).

Un entier long (sur 4 octets) sera codé OxAABBCCDD en langage C et traduit $DDCCBBAA en mémoire. Les deux groupes de 16 bits sont eux aussi inversés.

A cause de ce genre de subtilité, les deux lignes suivantes ne sont pas du tout équivalentes sur l'Apple IIGS, alors qu'elles le seraient sur Macintosh :

```
char    vect1[6] = {0x0F,0xAB,0x23,0xC7,0xD0,0x1E};
int     vect2[3] = {0x0FAB,0x23C7,0xD01E};
```

En effet, vect1 est représenté en mémoire par la séquence hexadécimale OFAB 23C7 DO1E, tandis que vect2 est codé ABOF C723 DO1E... ce qui fait une sacrée différence ! Attention donc à ceux qui définiront des curseurs de manière non structurée (voir le chapitre III sur QuickDraw).

La suite de cet ouvrage obéira aux conventions suivantes :

char	variable sur 1 octet	char*	pointeur sur caractère (occupe 4 octets)
int	variable sur 2 octets	int*	pointeur sur entier (occupe 4 octets)
long	variable sur 4 octets	long*	pointeur sur entier long (occupe 4 octets)

On définira les pointeurs et les handles (voir le chapitre Memory Manager) de la manière suivante :

```
typedef char    * Pointer;
typedef Pointer * Handle;
```

Quand des variables prendront des valeurs booléennes, elles seront codées sur 2 octets et prendront la valeur TRUE ou FALSE, ainsi définies :

```
#define   TRUE     (-1)
#define   FALSE    0
```

La valeur – 1 pour TRUE signifie que les 16 bits sont à 1, la valeur 0 pour FALSE que les 16 bits sont nuls. Quand une valeur booléenne sera retournée par une fonction, on ne testera jamais son égalité à TRUE, mais son inégalité à FALSE : on ne se posera jamais la question *xxx is true ?* mais *xxx is not false ?*

En C, on ne se pose même pas la question : toute valeur nulle est fausse, toute valeur non nulle est vraie. Si *valeur* est la valeur à tester, on n'écrira jamais :

```
if (valeur == TRUE) ...
```

mais :

```
if (valeur) ...
```

ou à la rigueur :

```
if (valeur != FALSE) ...
```

La première de ces trois expressions pourrait conduire au résultat inverse de celui qu'on espère, les deux autres ne présentent pas ce risque. Par contre, pour assigner une valeur booléenne à une variable, on écrira :

```
valeur1 = TRUE;
valeur2 = FALSE;
```

Autre valeur à définir, zéro-long. Nous emploierons souvent cette expression, qui désigne tout simplement la valeur nulle sur 4 octets. Suivant l'environnement de développement, elle sera peut-être prédéfinie sous l'appellation NIL ou NULL. L'important, c'est qu'il s'agisse de 32 bits tous nuls !

CHAPITRE I

APERÇU DES OUTILS

GÉNÉRALITÉS SUR LES OUTILS

Pour développer des applications dans le respect de l'interface utilisateur, mais aussi pour être sûr qu'elles resteront compatibles avec l'évolution future inévitable du système, Apple met à la disposition du développeur un ensemble de routines, groupées par thèmes, localisées soit en mémoire morte, soit dans des répertoires particuliers de la disquette système et chargés en mémoire vive au moment de leur utilisation.

Identification des outils

Chaque outil (appelons *outil* un groupe de routines formant un ensemble cohérent) possède un numéro d'identification qui lui est assigné de manière permanente et définitive. Les outils actuellement assignés sont les suivants :

Outils en ROM

1. Tool Locator
2. Memory Manager
3. Miscellaneous Tools
4. QuickDraw II
5. Desk Manager
6. Event Manager
7. Scheduler
8. Sound Tools
9. Apple Desktop Bus
10. SANE
11. Integer Math
12. Text Tools
13. *Utilisation interne*

Outils en RAM

14. Window Manager
15. Menu Manager
16. Control Manager
17. System Loader
18. QuickDraw auxilliary
19. Printer Driver
20. Line Edit
21. Dialog Manager
22. Scrap Manager
23. Standard File
24. Disk Utilities
25. Note Synthesizer
26. Note Sequencer
27. Font Manager

Ces outils se trouvent soit en mémoire morte, soit sur disquette et chargés en mémoire vive (sous la responsabilité de l'application) au moment de leur utilisation. Le premier outil, le Tool Locator, est l'outil des outils. C'est lui qui permet le chargement des routines, qu'elles soient en mémoire morte ou sur disquette, et qui

permettra les évolutions futures. Tel outil actuellement en mémoire morte pourrait subir des modifications *(patches)* et se retrouver en partie sur disquette dans une version ultérieure. Au contraire, tel outil actuellement sur disquette pourrait se retrouver un jour en mémoire morte, étant donné ses possibilités d'extension. Grâce au Tool Locator, une application écrite correctement n'aura pas à être modifiée lorsque ces modifications éventuelles interviendront (et elles interviendront sûrement : il n'est pas d'exemple de mémoire morte sans bug). Le Tool Locator sera étudié dans le paragraphe suivant.

Dans cet ouvrage, nous étudierons certains outils de la liste, liés à l'emploi de l'interface utilisateur (fenêtres, menus, contrôles, dialogues) et d'autres nécessaires à leur emploi (gestion mémoire, gestion graphisme). Ces outils, et tous les autres, sont décrits de manière exhaustive dans la documentation technique (en anglais) fournie par Apple. En voici une très rapide description (les outils précédés du signe ◇ sont étudiés dans cet ouvrage, ceux précédés du signe – sont évoqués dans le chapitre X).

◇ Tool Locator est l'outil qui permet aux outils et à l'application de communiquer. Il est possible d'en créer d'autres que ceux fournis par Apple, le Tool Locator saura les gérer.

◇ Memory Manager gère la mémoire de l'Apple IIGS quand celui-ci est utilisé en mode natif (c'est-à-dire quand il n'utilise pas le mode émulation, pour faire tourner les anciens programmes Apple IIe ou IIc).

– Miscellaneous Tools gère des routines très proches de la machine, pour gérer la mémoire permanente (par exemple tout ce qui est géré par l'accessoire de bureau Tableau de Bord), l'horloge interne (lire ou écrire l'heure), les interruptions et le vertical blanking, les erreurs fatales (plantage du système), l'interfaçage avec la souris, le compactage des images, certaines manipulations de chaînes de caractères, etc.

◇ QuickDraw II (dans la suite nous dirons QuickDraw) gère tout le graphisme super haute résolution (dans les deux modes possibles) : tout ce qui apparaît à l'écran dans ces modes dépend de QuickDraw.

– Desk Manager gère les deux types d'accessoires de bureau : accessoires de bureau classiques, obtenus par Pomme - Contrôle - Escape, et accessoires nouveau style, présents dans le menu .

◇ Event Manager gère les événements (clic souris, touche enfoncée, etc.).

• Scheduler s'occupe de gérer certains délais quand des accessoires de bureaux ou d'autres tâches essaient d'utiliser des ressources occupées.

• Sound Tools permet à une application d'accéder au hardware lié au son sans connaître la moindre adresse d'entrée-sortie, et d'alimenter les 64 Ko de mémoire dédiés à la gestion des sons. Deux configurations sont possibles, l'une pour une gestion compatible avec les anciens Apple II, l'autre pour prendre en compte les possibilités du Digital Oscillator Chip d'Ensoniq.

• ADB permet la gestion de l'Apple Desktop Bus (pour programmer des périphériques d'entrées sur la prise clavier, tels un joystick, un lecteur de codes à barres, une tablette graphique ou un clavier musical).

• SANE (Standard Apple Numerics Environment) est un ensemble de routines permettant les calculs en virgule flottante (sur 32 bits, 64 bits ou 80 bits). L'utilisation de ces routines est impérative pour bénéficier par exemple des vertus d'un futur coprocesseur arithmétique. Un environnement de développement rendra généralement transparente l'utilisation de SANE, s'il est prévu pour s'en servir.

• Integer Math Tools est un ensemble de routines permettant calculs et conversions sur entiers longs et nombres fractionnaires particuliers.

• Text Tools permet l'utilisation par une application respectant l'interface utilisateur des anciens modes texte (40 et 80 colonnes) avec quelques améliorations : le texte et le fond peuvent prendre une couleur parmi 16 possibles (ce sont les couleurs permises par le Tableau de Bord).

◇ Window Manager est le gestionnaire de fenêtres.

◇ Menu Manager est le gestionnaire de menus déroulants.

◇ Control Manager est le gestionnaire de contrôles.

◇ System Loader permet à une application d'être découpée en plusieurs segments. Une de ses fonctions sera de charger en mémoire vive les segments non présents nécessaires à la bonne marche de l'application. Fait partie intégrante du système d'exploitation de la machine.

• High level Printer Driver fournit des routines permettant à l'application d'imprimer, quelle que soit l'imprimante en ligne (pourvu que le driver propre à l'imprimante soit présent). Ces routines appellent celles du Low level Driver, plus proches de la machine.

◇ Line Edit fournit des routines d'édition de texte (insertion, copier-coller, etc.).

◇ Dialog Manager est le gestionnaire de dialogues et alertes.

– Scrap Manager est le gestionnaire de presse-papier, grâce auquel différentes applications peuvent s'échanger des informations par copier-coller.

– Standard File fournit les fenêtres standard pour répondre aux articles *Ouvrir...* et *Enregistrer...* du menu *Fichier* (ouverture des fichiers, sauvegarde des fichiers).

• *Disk Utilities* permettent notamment l'initialisation des disquettes et la duplication des fichiers.

• Note Synthesizer et Note Sequencer offrent des routines de haut niveau pour faire de la musique et de la synthèse vocale grâce au Digital Oscillator Chip d'Ensoniq. Le séquenceur est capable de jouer une partition avec les instruments définis par le synthétiseur.

– Font Manager permet l'utilisation de divers jeux de caractères et de divers styles à l'intérieur de chaque police.

Identification des routines d'un outil

Dans un outil donné, chaque routine (procédure ou fonction) possède un identifiant. Le couple identifiant de l'outil/identifiant de la routine détermine un sous-programme de la boîte à outils de manière unique. Les huit premiers identifiants dans chaque outil ont une signification particulière et similaire d'un outil à l'autre :

1. Routine d'initialisation au moment du boot (suffixe *BootInit*).
2. Routine d'initialisation au niveau de l'application (suffixe *StartUp*).
3. Routine de désallocation à la fin de l'application (suffixe *ShutDown*).
4. Fonction retournant le numéro de version de l'outil (suffixe *Version*).
5. Routine à exécuter en cas de *Reset* (suffixe *Reset*).
6. Fonction retournant le statut actif/non actif d'un outil (suffixe *Status*).
7. Réservé.
8. Réservé.

• Les routines de suffixe *BootInit* sont exécutées lors de la phase de démarrage du système, les outils en mémoire morte par le *ROM startup code* et les outils sur disquette au moment de leur installation dans le système. Ces routines ne devront jamais être appelées par une application.

• Les routines de suffixe *StartUp* sont appelées dans un ordre rigoureux par l'application à son démarrage. Elles initialisent les outils, réservent de la mémoire, etc. Tout appel à une routine de la *Toolbox* sans avoir préalablement initialisé l'outil auquel elle appartient se soldera invariablement par des résultats non maîtrisés ou par un plantage du système. Les outils non utilisés par une application et par les autres outils n'auront pas besoin d'être initialisés. Si une application tente d'initialiser une deuxième fois un outil, elle recevra un message d'erreur mais il ne se passera rien de grave. Voir le chapitre XII pour une séquence type d'initialisation des outils.

• Les routines de suffixe *ShutDown* sont appelées par l'application au moment où elle s'achève, dans l'ordre inverse des initialisations. Toute la mémoire occupée par les outils est ainsi libérée, ce qui permet à l'application qui prend la main (généralement le Finder), de travailler dans des conditions correctes. Voir le chapitre XII pour une manière type de quitter une application.

• Chaque outil possède un numéro de version (sur deux octets) qui lui est propre, et qui obéit aux règles suivantes :
- bit 15 : s'il est à 1, l'outil est encore à l'état de prototype, dans une version non définitive ;
- bits 14 à 8 : numéro majeur de la version (au moins 1 pour une version non prototype) ;
- bits 7 à 0 : numéro mineur de la version (démarre à 0).

Les fonctions de suffixe *Version* retournent le numéro de version de l'outil. Ainsi, une valeur telle que $0103 signifie qu'on utilise la version 1 après trois modifications mineures, une valeur telle que $800A qu'on a affaire à la dixième version d'un outil en cours de développement. Ce numéro de version est très utile pour la compatibilité future : certaines applications ne pouvant tourner avec des versions trop anciennes des outils pourront refuser de démarrer en donnant un message d'erreur explicite, plutôt que de se planter lamentablement (voir plus loin le Tool Locator, et dans le chapitre XII un exemple d'utilisation de ces fonctions).

• Les routines de suffixe *Reset* sont appelées de manière interne dès que l'utilisateur force un *Reset*, en appuyant simultanément sur les touches Contrôle et Reset.

Note La combinaison Pomme - Contrôle - Reset est plus draconienne et force la réinitialisation de tous les outils.

• Les fonctions de suffixe *Status* retournent la valeur TRUE si l'outil dont elles dépendent est initialisé, FALSE sinon. C'est le seul exemple où on peut appeler une routine avant d'avoir initialisé son outil d'appartenance, ou après l'avoir désalloué (par *ShutDown*).

Sauf en ce qui concerne les routines d'initialisation (suffixe *StartUp*), nous ne détaillerons pas dans les différents chapitres ces routines obligatoirement présentes dans les outils. Dans les exemples du chapitre XII, nous verrons comment écrire les fonctions C assurant le début et la fin d'une application valables dans la plupart des cas.

Nom des routines

Toutes les routines de la *ToolBox* portent un nom unique. Suivant les environnements de développement, il faudra ou non respecter les majuscules et les minuscules dans ces noms. Dans les environnements Megamax C, les noms des routines sont déclarés dans des fichiers .h, suivant l'exemple suivant :

```
#define dispatcher 0xE10000
...
extern pascal void DialogStartUp( ) inline(0x0215, dispatcher);
```


Cette déclaration (notons l'instruction spéciale *inline*) crée la relation entre le couple numéro de l'outil/numéro de routine et le nom qui sera employé pour appeler cette routine. **DialogStartUp** est la routine numéro 2 de l'outil $15 (Dialog Manager). Attention à l'orthographe ! Peu importe tout ce que vous lirez dans cet ouvrage ou ailleurs, si votre environnement déclare **DialogStartup** au lieu de **DialogStartUp** et qu'il fait la distinction entre les majuscules et les minuscules *(case sensitive)*, vous serez obligé d'employer son orthographe... ou de corriger son fichier d'interfaçage. Il est toujours bon d'aller y jeter un œil !

Codes d'erreur

Chaque outil est susceptible de retourner des codes d'erreur. Un code d'erreur possède un format parfaitement défini sur l'Apple IIGS : c'est un entier sur deux octets, l'octet le plus significatif contenant le numéro de l'outil dans lequel l'erreur s'est produite, et le moins significatif donnant l'erreur proprement dite.

Suivant les environnements, on pourra récupérer certains codes d'erreur (ceux des erreurs non fatales en tout cas) dans le registre A (cas de l'assembleur) ou dans une variable (dans les environnements Megamax C, cette variable s'appelle *_errno* et doit être déclarée en début de programme).

Nous signalerons les procédures susceptibles de retourner un code erreur, en précisant simplement que c'est dans *_errno* qu'il faut récupérer ce code.

TOOL LOCATOR

Survol du fonctionnement

On accède aux routines de la boîte à outils par l'intermédiaire du Tool Locator, l'outil des outils. Cet outil fonctionne grâce à quelques tables, qui permettent ensuite une grande souplesse pour les modifications futures.

Étant donné un numéro d'outil, le Tool Locator peut trouver une entrée dans la *Tool Pointer Table*. Cette table contient l'adresse des *Function Pointer Tables* (chaque outil possède une telle table). Ces tables contiennent à leur tour l'adresse réelle des fonctions, l'entrée étant repérée par le numéro de la routine.

Chaque outil en ROM possède sa table d'adresses de routines en ROM. Il existe aussi en ROM une table d'adresses d'outils (qui ne référence évidemment que les outils en ROM). Une place est fixée en mémoire vive pour recevoir l'adresse de la table des outils en ROM, initialisée au démarrage du système. De sorte que, si du jour au lendemain Apple décide de modifier ses mémoires mortes, le système n'aura qu'à donner la nouvelle adresse de la table pour que tout continue à fonctionner.

Les outils ayant besoin d'un peu de mémoire pour fonctionner, le Tool Locator gère une dernière table d'adresses, la *Work Area Pointer Table*, qui désigne pour chaque outil l'adresse de l'espace mémoire qu'il s'est réservé.

Nous n'entrerons pas dans les détails de ce qui se passe durant le démarrage du système, mais il faut savoir que les outils en mémoire morte sont chargés automatiquement, pas les outils résidant sur la disquette système : il est de la responsabilité de l'application de les charger.

Routines à utiliser

Nous ne verrons dans ce paragraphe que les routines dont l'utilisation est obligatoire pour faire fonctionner une application.

Comme tout autre outil, le Tool Locator nécessite une initialisation avant de pouvoir être utilisé. Cette tâche est assurée par la procédure **TLStartUp,** sans argument.

Le chargement des outils qui ne sont pas en mémoire morte peut s'effectuer de plusieurs manières : la procédure **LoadTools** charge plusieurs outils à la fois, alors que la procédure **LoadOneTool,** comme son nom l'indique, n'en charge qu'un à la fois et peut être contrebalancée par un appel à **UnloadOneTool,** qui supprime l'outil de la mémoire.

LoadOneTool admet deux arguments : le numéro de l'outil et la version minimale de l'outil à utiliser. Si, par exemple, pour une raison ou pour une autre, une application ne peut marcher qu'avec une version 1.03 ou postérieure du Window Manager, on pourra trouver l'instruction suivante dans les premières lignes de programme :

LoadOneTool (14, 0×0 103);

Si le fichier *Tool014* sur le disque de démarrage correspond à une version 1.03 ou postérieure (1.04, 2.00, ...), l'outil sera chargé. Si le fichier correspond à une version antérieure (1.02, $8101, ...), le fichier ne sera pas chargé et le code erreur $0110 *(version error)* sera retourné suivant la méthode générale (dans la variable *_errno*). Notons que toute erreur détectée par le System Loader ou ProDOS durant cet appel sera également répercutée.

Mais comment prévenir l'utilisateur qu'une erreur est survenue ? La méthode normale de l'interface utilisateur est d'afficher une fenêtre d'alerte (voir le chapitre IX consacré au Dialog Manager). Ici c'est impossible puisque c'est le gestionnaire de fenêtres lui-même qui ne peut être chargé !

Le Tool Locator offre deux fonctions pour répondre à ce problème : **TLMountVolume** (qui simule une fenêtre d'alerte) et **TLTextMountVolume** (qui utilise le mode texte classique 40 colonnes). Deux lignes de message peuvent être affichées (se limiter à 36 caractères environ), et les fonctions retourneront la valeur 1 si le bouton 1 ou Retour est choisi par l'utilisateur, la valeur 2 si le bouton 2 ou Escape est choisi.

La fonction **TLMountVolume** admet six arguments : l'abscisse et l'ordonnée du coin supérieur gauche de la pseudo-fenêtre d'alerte (en coordonnées écran, voir le chapitre sur QuickDraw), un pointeur sur la première ligne de texte, un pointeur sur la deuxième ligne de texte, un pointeur sur le texte du premier bouton (équivalant à *OK*), et un pointeur sur le texte du deuxième bouton (équivalant à *Annuler*). Elle utilise le mode super hi-res, il est donc impératif que QuickDraw ait été initialisé ; elle utilise le clic souris, il est donc impératif que l'Event Manager ait été initialisé, et la procédure **InitCursor** exécutée. Voir le chapitre XII pour un exemple complet.

La fonction **TLTextMountVolume** admet quatre arguments, identiques aux quatre derniers arguments de **TLMountVolume.** Du fait qu'elle utilise le mode texte 40 colonnes, aucune initialisation autre que celle du Tool Locator n'est requise. Cette fonction pourra être appelée quand on n'est pas sûr de pouvoir charger QuickDraw et l'Event Manager, parce qu'on veut forcer pour leur utilisation un numéro de version particulier.

Pour charger plusieurs outils en un seul appel, on utilisera la procédure **LoadTools,** avec pour seul argument un pointeur sur une table ayant le format suivant :

- un entier donnant le nombre d'outils à charger ;
- pour chaque outil, deux entiers : son identifiant et la version minimale autorisée.

LoadTools retourne des codes d'erreur identiques à **LoadOneTool** (*Version error,* erreurs de chargement et erreurs ProDOS).

Pour charger en une seule instruction Window Manager, Menu Manager, Control Manager, Line Edit, Dialog Manager, Scrap Manager et Standard File, on procédera de la manière suivante :

```
int tools[ ] = {
    7,                          /* sept outils à charger */
    14, 0x0100,                 /* Window Manager à partir de la version 1.00 */
    15, 0x0100,                 /* Menu Manager à partir de la version 1.00 */
    16, 0x0100,                 /* Control Manager à partir de la version 1.00 */
    20, 0x0100,                 /* Line Edit à partir de la version 1.00 */
    21, 0x0100,                 /* Dialog Manager à partir de la version 1.00 */
    22, 0x0100,                 /* Scrap Manager à partir de la version 1.00 */
    23, 0x0100                  /* Standard File à partir de la version 1.00 */
    };

LoadTools(tools);               /* charge les 7 outils */
```

Si une application est divisée en plusieurs segments et qu'ils ne sont pas tous statiques (c'est-à-dire qu'ils ne sont pas obligés de résider continuellement en mémoire vive), si un segment non statique est seul à utiliser un outil particulier, on peut avoir envie de supprimer cet outil de la mémoire en même temps que le segment. Ce qui sera fait avec la procédure **UnloadOneTool,** qui recevra en argument l'identifiant de l'outil à effacer. Aucune erreur ne sera générée, même si l'outil n'a pas été chargé ou a déjà été effacé.

CHAPITRE II

MEMORY MANAGER

MÉMOIRE DE L'APPLE IIGS

Notre propos n'est pas dans ce chapitre d'entrer dans des détails trop techniques sur l'organisation de la mémoire de l'Apple IIGS. Cependant, quelques notions fondamentales sont à connaître, et nous allons nous y arrêter quelques instants.

Le microprocesseur 65816 au cœur de la machine est capable de gérer les adresses sur 24 bits, c'est-à-dire qu'il est capable d'adresser jusqu'à 16 Mo. La mémoire est formée de blocs de 64 Ko appelés *banques*. Chacune des 256 banques possibles est numérotée de 0 à $FF. La machine possède en termes de mémoire accessible les caractéristiques suivantes :

- banques 0 et 1 : 128 Ko de mémoire vive. La banque 0 joue un rôle très important, puisqu'elle contient le *stack* (pile par où transitent les arguments que se passent les programmes et sous-programmes, fonctions et autres procédures) et les diverses pages zéro, qui intéressent particulièrement un mode d'adressage privilégié du microprocesseur.
- banques 2 à $7F : extension de la mémoire vive (jusqu'à 8 Mo).
- banques $E0 et $E1 : 128 Ko de mémoire vive. C'est notamment sur ces banques que se trouve la mémoire écran. Tout ce qui apparaît sur un écran d'Apple IIGS, aussi bien en mode natif qu'en mode émulation 6502, transite par ces banques.
- banques $FO à $FD : extension de la mémoire morte (jusqu'à 1 Mo).
- banques $FE et $FF : 128 Ko de mémoire morte standard.

C'est dans toute cette étendue de mémoire par blocs que les applications vont pouvoir stocker le contenu de leurs variations et des objets qu'elles manipulent. Le Memory Manager a pour mission de prendre en charge toute la gestion de cette mémoire, et d'en optimiser si possible l'utilisation. Le programmeur devra faire très attention à la façon dont il sollicitera le Memory Manager, et avoir continuellement à l'esprit qu'il n'a pas le droit de tout utiliser pour lui : l'Apple IIGS est conçu pour faire fonctionner plusieurs applications en même temps (il n'est pas multitâches, mais il est possible de charger plusieurs applications simultanément en mémoire (du moins en théorie), et de les utiliser à tour de rôle, en switchant presque instantanément de l'une à l'autre).

Notons que 64 Ko de mémoire non évoqués ci-dessus sont la propriété exclusive du coprocesseur sonore Ensoniq DOC, et qu'ils sont accessibles en utilisant les routines de Sound Tools, non traitées dans cet ouvrage.

UTILISATION DU MEMORY MANAGER

Blocs fixes et pointeurs, blocs relogeables et handles

Dans la mémoire de l'Apple IIGS vont cohabiter deux types d'objets : les objets fixes et les objets relogeables. Par la suite, nous parlerons d'objets, mais aussi de blocs mémoire représentant ces objets.

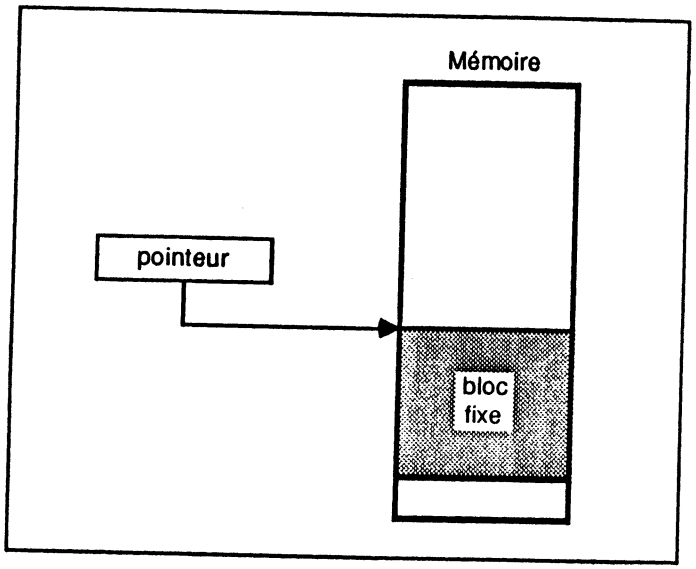


Figure II.1. Pointeur et bloc fixe

- Les blocs fixes représentent des objets dont l'adresse n'a pas le droit de changer : le programme qui les manipule attend ces objets à une adresse précise, et se plantera à coup sûr s'il ne les trouve pas à leur place. Ces blocs seront repérés par des *pointeurs* (Figure II.1).

Qu'est-ce qu'un pointeur ? Tout simplement un objet qui contient l'adresse de début d'un bloc fixe. De tels objets seront représentés sur 4 octets (même si 3 auraient suffi, puisqu'on a déjà dit que le bus d'adresses du 65816 contenait 24 bits).

- Les blocs relogeables représentent des objets dont l'adresse n'est pas fixe dans la mémoire. Dès qu'il le jugera nécessaire, le Memory Manager déplacera ces blocs pour essayer d'optimiser l'occupation de la mémoire, en comblant tant que faire se peut les trous laissés entre les blocs fixes. De tels blocs sont repérés par des *handles*, des pointeurs sur des pointeurs maîtres (Figure II.2).

Qu'est-ce qu'un pointeur maître ? C'est un bloc fixe contenant l'adresse actuelle d'un bloc relogeable. Il faut bien comprendre le schéma suivant : un handle connaît l'adresse d'un bloc fixe connaissant l'adresse d'un bloc relogeable. Quand le bloc change de localisation, il est de la responsabilité du Memory Manager de stocker sa

nouvelle adresse dans le pointeur maître qui le repère. Le handle n'est pas modifié dans l'opération. C'est donc un pointeur particulier : il sera lui aussi stocké sur 4 octets.

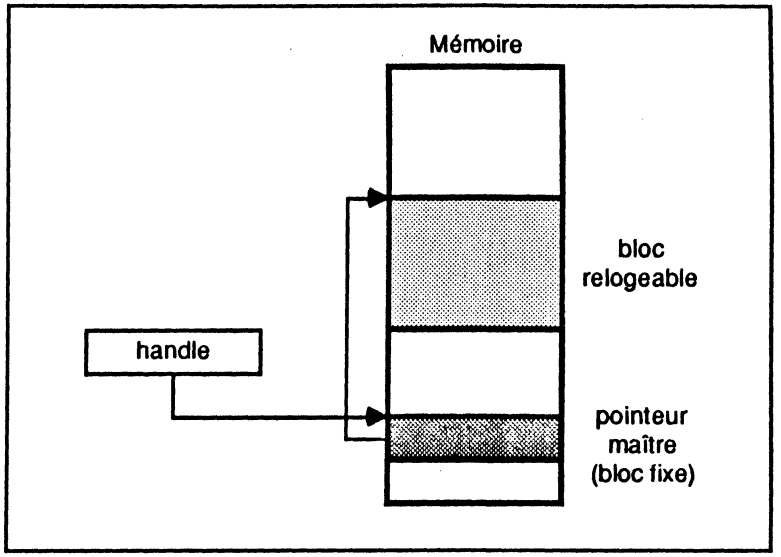


Figure II.2. Handle et bloc relogeable.

Remarque Contrairement à celui du Macintosh, le Memory Manager de l'Apple IIGS ne connaît pas véritablement les pointeurs : un bloc fixe est repéré par un handle, tout comme un bloc relogeable. C'est l'application qui gérera le pointeur, en déréférençant le handle. (Voir la section « Initialisation, allocation de blocs mémoire » de ce chapitre).

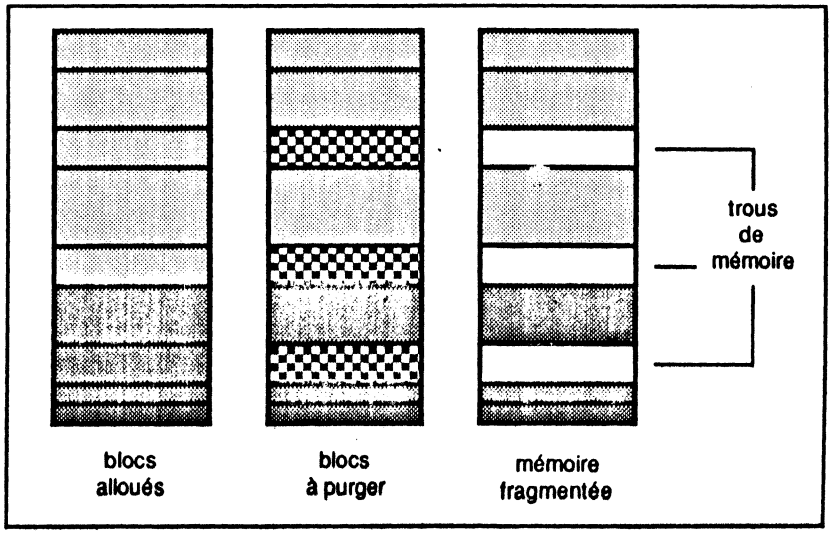


Figure II.3. Fragmentation de la mémoire

Fragmentation et compactage de mémoire

Une application va allouer des blocs de mémoire au fur et à mesure de ses besoins, et va les désallouer dès qu'elle en aura terminé avec leur utilisation. Ces opérations pouvant intervenir dans n'importe quel ordre, on finira par avoir au bout d'un certain temps une mémoire constituée de blocs libres perdus au milieu de blocs alloués (en cours d'utilisation). Beaucoup de place perdue, puisque l'allocation d'un bloc ne peut se faire que d'un seul tenant. On peut arriver au paradoxe de ne pouvoir allouer un bloc même s'il y a assez de mémoire disponible, tout simplement parce que cette place disponible est trop morcelée (Figure II.3).

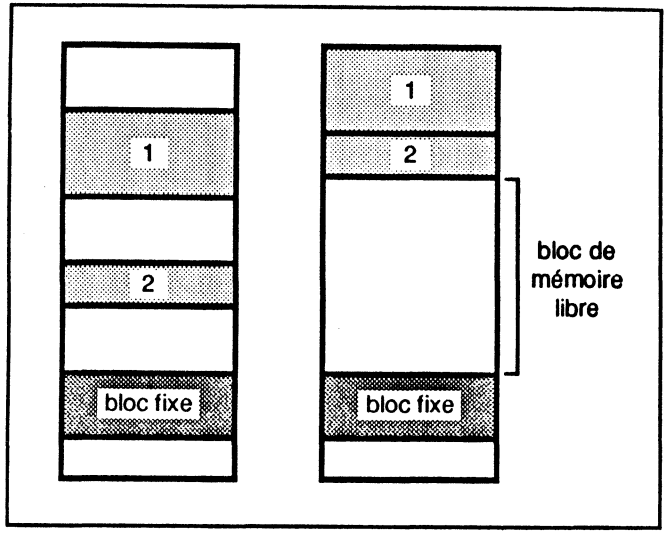


Figure II.4. Compactage de la mémoire.

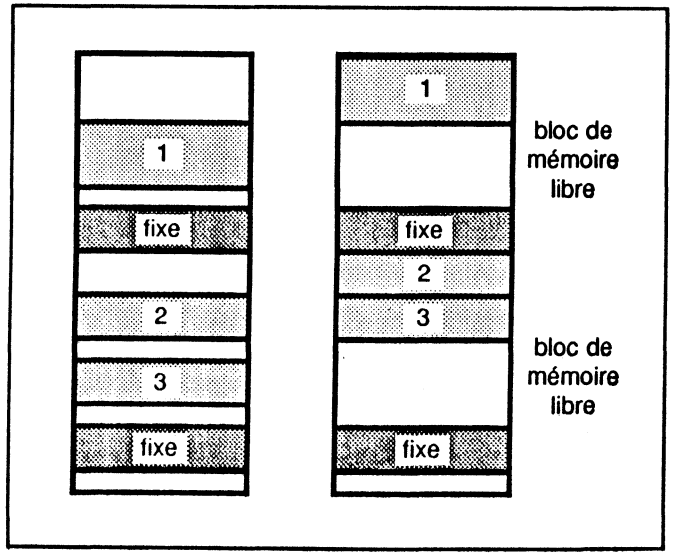


Figure II.5. Les blocs fixes peuvent gêner le compactage.

C'est pourquoi le Memory Manager essaie de temps à autre de compacter la mémoire. En quoi faisant ? Tout simplement en déplaçant les blocs relogeables de telle sorte qu'ils comblent le maximum de vide, libérant par la même occasion un maximum d'espace en continu. Les blocs fixes ne sont évidemment pas déplacés, et constituent des îlots inamovibles dans la mémoire (Figure II.4). Pire : si un bloc relogeable est coincé entre deux blocs fixes, même un compactage ne pourra l'en faire sortir (Figure II.5). Pour éviter ce genre de situation, le Memory Manager essaie d'allouer les blocs fixes vers le bas de la mémoire, et les blocs relogeables vers le haut.

Dans la mesure du possible, les objets que manipule une application devraient être traités comme des blocs relogeables, afin de faciliter le compactage et donc gâcher moins de mémoire.

Blocs verrouillés et blocs purgeables

Quand on est en train d'utiliser un bloc de mémoire relogeable (par exemple, quand on transfère la représentation mémoire d'une image vers la mémoire écran), il peut s'avérer inopportun de voir subitement son bloc se déplacer lors d'un compactage ! Les résultats les plus imprévisibles pourraient en découler...

Le Memory Manager offre la possibilité de verrouiller provisoirement un bloc relogeable. Quand un bloc est verrouillé, c'est comme si on avait affaire à un bloc fixe : un compactage ne le fait pas changer d'adresse. Dès qu'on a fini de traiter le bloc, il faut le déverrouiller (sinon, on perd tout le bénéfice d'un bloc relogeable par rapport à un bloc fixe).

Un bloc dont on n'a plus besoin peut être rendu purgeable. Un bloc purgeable ne disparaît pas immédiatement de la mémoire, mais seulement en cas de besoin, quand après compactage le Memory Manager est toujours incapable d'allouer un nouveau bloc. Il y a trois niveaux de priorité dans la purgeabilité d'un bloc : les blocs purgeables de niveau III seront purgés avant les blocs de niveau 2 et 1 (le niveau 0 signifiant non purgeable).
Attention Un bloc verrouillé ne peut être purgé.

Remarque Le niveau 3 de purge est utilisé par le System Loader et ne devrait pas être utilisé par une application. Quand une application est terminée, les segments de programmes qu'elle utilise ne sont pas effacés de la mémoire, mais seulement rendus purgeables au niveau 3. De la sorte, si l'application devait de nouveau être appelée, il n'y aurait pas à la recharger en mémoire. Si le système a besoin de mémoire entre temps, il effacera ces blocs inutiles : dès qu'un bloc de niveau 3 est purgé, tous les autres le sont aussi.

Quand un bloc relogeable est purgé, son pointeur maître reste alloué et prend la valeur zéro-long (NIL). Dans ce cas précis, le handle est appelé *handle vide*. Les données que contenait le bloc sont définitivement perdues (elles doivent être recréées par le programme pour une nouvelle utilisation).

Attributs de blocs mémoire

Chaque bloc de mémoire possède un certain nombre d'attributs, stockés sur un mot de 16 bits :

– bit 0 : banque fixe. Si ce bit est positionné, le bloc doit impérativement commencer dans une banque spécifiée. Par exemple, l'allocation d'un bloc à utiliser comme page zéro doit impérativement se situer dans la banque 0 ;

– bit 1 : adresse fixe. Si ce bit est positionné, le bloc doit impérativement être alloué à une adresse précise. Par exemple, le système alloue la mémoire écran à une adresse immuable ;

– bit 2 : alignement de la page. Si le bloc doit être aligné sur une page, ce bit doit être positionné ;

– bit 3 : non-utilisation de mémoire spéciale. Certaines parties de la mémoire sont qualifiées de spéciales, car des restrictions sont à apporter à leur utilisation (ce sont les parties utilisées en émulation Apple IIe). Si ce bit est positionné, le bloc ne pourra pas se trouver dans ces parties de mémoire spéciales ;

– bit 4 : non-chevauchement. Si ce bit est positionné, le bloc ne pourra pas s'étaler sur deux banques distinctes. C'est le cas notamment des segments de programmes ;

– bits 8 et 9 : niveau de priorité pour la purge. Ces bits seront à zéro à l'allocation du bloc, et modifiés par la suite ;

– bit 14 : bloc fixe. Si le bit est positionné, le bloc est fixe ; s'il est à zéro, le bloc est relogeable. En règle générale, un segment de programme constituera un bloc fixe, et un bloc de données devrait être relogeable ;

– bit 15 : bloc verrouillé. Si le bit est positionné, le bloc est verrouillé, donc provisoirement fixe et non purgeable. A l'allocation d'un bloc relogeable, ce bit devrait être à zéro.

EXEMPLES D'UTILISATION

Initialisation, allocation de blocs mémoire

```
int      myId;                                  /* identifiant de l'application */
Handle   theHdl;                                /* un handle */
Pointer  zeropg;                                /* un pointeur */

...
myId = MMStartUp( );                            /* initialisation du Memory Manager */
...
theHdl = NewHandle(0x800L, myId, 0xC001, 0L);   /* allocation d'un nouveau bloc (fixe) */
zeropg = * theHdl;                              /* pointeur sur bloc fixe */
...
```

• Comme tous les gestionnaires que nous allons étudier, le Memory Manager doit être initialisé et c'est seulement ensuite que nous pouvons utiliser les procédures et fonctions qui le composent. C'est la fonction **MMStartUp** qui assure l'initialisation du Memory Manager. Elle retourne dans un entier un identifiant dont l'importance est capitale, puisqu'il servira de manière interne à identifier l'application : tout bloc de mémoire alloué par une application gardera trace de cet identifiant, ce qui permettra notamment au Memory Manager d'évacuer tous ces blocs quand elle sera terminée. Pour avoir une vision d'ensemble du début et de la fin d'une application, consulter le chapitre XII.

• Pour allouer un nouveau bloc de mémoire, on utilisera la fonction **NewHandle**, qui réclame quatre arguments :

– le premier argument donne la taille en octets du bloc à allouer. Cette taille est codée sur quatre octets (entier long). Dans l'exemple, on demande au Memory Manager d'allouer un bloc de huit pages (rappelons que dans le jargon Apple II, une page fait 256 octets, soit $100 octets).

– le deuxième argument est l'identifiant de l'application. Puisque chaque bloc gardera mémoire de l'application qui l'a créé, cet argument est nécessaire.

– le troisième argument décrit les attributs du bloc à allouer, tels qu'ils ont déjà été définis. Puisque C001 hexa = 1100 0000 0000 0001 binaire, nous demandons un bloc verrouillé, fixe, dans une banque déterminée.

– le quatrième argument est un entier long qui sert de complément à l'argument précédent. Il contiendra une adresse permettant de fixer la valeur de certains attributs (banque fixe, adresse fixe). Dans l'exemple, l'adresse zéro-long signifie que la banque dans laquelle le bloc doit être alloué est la banque zéro.

Cet exemple permet donc d'allouer huit pages zéro consécutives. La fonction **NewHandle** retourne un handle sur le bloc alloué, ou zéro-long si l'allocation a échoué (pas assez de mémoire libre consécutive, même après compactage et purge des blocs purgeables).

Remarquons la façon dont on obtient un pointeur à partir d'un handle en langage C : on affecte à une variable de type pointeur le contenu de la zone pointée par le handle qui n'est autre que l'adresse du bloc. Puisque le bloc est déclaré fixe, il n'y a aucune précaution particulière à prendre pour déréférencer le handle. Pour pouvoir parler de *Pointer* et de *Handle* en C, il est nécessaire de définir ces types :

```
typedef char *Pointer
typedef char **Handle
```

Si l'application ne doit jamais utiliser le handle quand elle cherche à gérer un pointeur sur bloc fixe, il est plus rapide d'écrire l'allocation ainsi :

```
zeropg = * NewHandle(0x800L, myId, 0xC001, 0L);      /* pointeur sur bloc fixe */
```

Inutile de préciser quelles catastrophes pourraient survenir si on faisait la même chose à un bloc relogeable ! Remarquons que cette façon d'écrire interdit ensuite de purger le bloc individuellement, puisque le Memory Manager ne sait y accéder que par l'intermédiaire d'un handle. C'est pourquoi il existe une fonction, **FindHandle**, qui permet de savoir à quel bloc appartient une adresse donnée : on passe en argument l'adresse en question, et elle retourne le handle sur le bloc, s'il existe, ou zéro-long.

NewHandle permet de créer des blocs de toutes tailles, y compris un bloc vide. Dans ce cas, le bloc doit être impérativement relogeable et non verrouillé, et le pointeur maître associé contiendra zéro-long.

Réallocation et désallocation de blocs mémoire

• L'application peut purger un bloc dont elle n'a plus besoin, grâce à la procédure **PurgeHandle**. Un seul argument, le handle sur le bloc à purger. Quand le bloc est purgé, il n'est plus accessible par l'application, mais le handle reste disponible et le pointeur maître associé contient zéro-long.

Remarque Le bloc est purgé si les deux conditions suivantes sont réalisées : il doit avoir été déclaré purgeable et il ne doit pas être verrouillé. Une autre procédure peut être appelée, **PurgeAll**, pour purger tous les blocs purgeables (et non verrouillés) associés à une application. Un seul argument : l'identifiant de l'application.

• Un handle disponible après une purge peut être réalloué à un autre bloc, grâce à la procédure **ReallocHandle** (Figure II.6). Cinq arguments, le dernier étant le handle à utiliser, les quatre premiers étant similaires à ceux de la fonction **NewHandle**.

• Un handle disponible après une purge peut être restauré grâce à la procédure **RestoreHandle**, qui admet comme seul argument le handle à réallouer. Le bloc aura la même taille et les mêmes attributs que celui déjà purgé, mais pas forcément la même localisation en mémoire. C'est pourquoi cette procédure ne marche pas pour les blocs situés à une adresse fixe ou dans une banque fixe.

• Pour purger un bloc de mémoire et désallouer le handle qui lui est associé, on utilisera la procédure **DisposeHandle**. Un seul argument, le handle à désallouer. La procédure agit même si le bloc n'a pas été déclaré purgeable, même s'il est verrouillé. La procédure **DisposAll** désallouera tous les handles associés aux blocs que l'application a créés (il faut rappeler en argument l'identifiant de l'application.

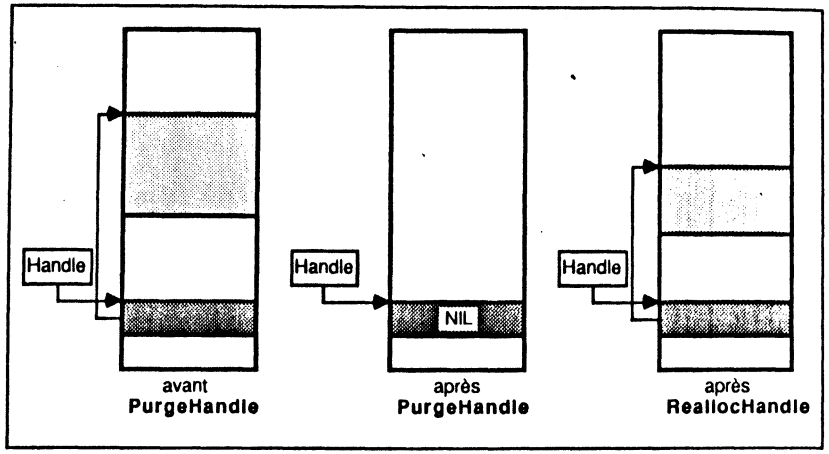


Figure II.6. Fonctionnement de PurgeHandle et ReallocHandle.

Modification des attributs d'un bloc

• Pour verrouiller un bloc, on doit utiliser la procédure **HLock**. Pour déverrouiller un bloc, on fait appel à la procédure **HUnlock**. Un seul argument dans chaque cas, le handle sur le bloc à traiter. Tant que le bloc est verrouillé, il possède une adresse fixe en mémoire et ne peut être purgé.

```
Handle thePict;                 /* thePict est un handle sur une image */
Rect   r;                       /* un rectangle */

HLock(thePict);          /* on verrouille le bloc avant de commencer à dessiner */
DrawPicture(thePict,&r); /* on dessine l'image (procédure QuickDraw) */
HUnlock(thePict);        /* c'est fini, on peut déverrouiller le bloc */
```

On peut verrouiller d'un coup tous les blocs appartenant à une application grâce à la procédure **HLockAll**, et les déverrouiller grâce à la procédure **HUnlockAll**. Un seul argument dans les deux cas, l'identifiant de l'application propriétaire des blocs.

• Pour rendre un bloc purgeable ou non purgeable, la procédure **SetPurge** permet de fixer le niveau de purge du bloc : la valeur 0 signifie non purgeable, les valeurs 1 à 3 signifient purgeable (3 étant un niveau de purgeabilité plus prioritaire que 1). Deux arguments : le niveau de purge (entier compris entre 0 et 3, l'application utilisant plutôt une valeur comprise entre 0 et 2) et le handle désignant le bloc.

En règle générale, une application préférera appeler **DisposeHandle** quand elle n'aura plus besoin d'un bloc, plutôt que **SetPurge**. Si par contre elle doit manipuler de nombreux blocs, elle peut fixer des niveaux de purge sur certains blocs qu'elle n'est pas en train d'utiliser. Si le Memory Manager a besoin de mémoire, il pourra les purger et l'application devra les réallouer (ou les restaurer) pour les utiliser de nouveau.

Existe également la procédure **SetPurgeAll**, qui rend purgeables tous les blocs alloués par l'application. Deux paramètres : le niveau de purge et l'identifiant de l'application... Nous voyons mal quelle application pourra se servir d'une telle routine.

Taille de blocs, espace mémoire

• Pour connaître la taille d'un bloc, on peut appeler la fonction **GetHandleSize**. On passe en argument le handle repérant le bloc, et elle retourne dans un entier long la taille du bloc en octets.

• Réciproquement, on peut changer la taille d'un bloc, grâce à la procédure **SetHandleSize**. Deux arguments : la nouvelle taille (entier long) et le handle concerné. La taille d'un bloc peut être réduite ou agrandie. En cas d'agrandissement, le Memory Manager fera la place nécessaire si besoin est (compactage de mémoire, purge de blocs) et déplacera éventuellement le bloc dans l'opération (si celui-ci n'est pas verrouillé, bien entendu). Il est impossible de changer la taille d'un handle vide.

• Pour forcer le compactage de la mémoire, la procédure **CompactMem** peut être utilisée. Aucun argument à passer. Compactage signifie déplacement de blocs relogeables vers le haut de la mémoire, de manière à créer le plus grand bloc libre possible. En aucun cas les blocs purgeables ne sont purgés.

• Pour connaître la taille mémoire disponible à un moment donné, on peut appeler la fonction **FreeMem**, sans argument. Elle commence par effectuer un compactage, puis retourne dans un entier long le nombre d'octets disponibles. Ce nombre ne tient pas compte des blocs purgeables, puisqu'ils ne sont pas encore purgés. A cause des problèmes de fragmentation, il ne sera sans doute pas possible d'allouer un bloc de cette taille. Par contre, la fonction **MaxBlock** retourne dans un entier long la taille en octets du plus grand bloc libre en mémoire, avant tout compactage ou toute purge.

• Pour connaître la taille mémoire totale utilisée par la machine, on appelle **TotalMem**. Cette fonction tiendra compte des 256 Ko présents sur la carte mère et de la mémoire additionnelle sur la carte d'extension.

Copie d'un bloc

Il existe quatre procédures permettant de copier un bloc de mémoire, qui diffèrent essentiellement par la manière dont est passée l'adresse du bloc à copier. Chacune des quatre procédures admet trois arguments (tous sur 32 bits) : un repère pour le bloc source, un repère pour le bloc destination et un nombre d'octets (taille du bloc à copier). Les procédures liront le contenu des n octets désignés à partir de l'adresse source, et les réécriront à partir de l'adresse destination, sans aucun contrôle de vraisemblance, en écrasant ce qui pourrait précédemment se trouver là. Le travail sera effectué même si la source et la destination se chevauchent, même si ces zones traversent des frontières de banques.

• **PtrToHand** : la source est repérée par une adresse, la destination par un handle (Figure II-7). Le handle doit exister au moment de l'appel, il est de la responsabilité de l'application de le créer. La logique voudrait que le bloc alloué possède une taille supérieure ou égale au nombre d'octets à transférer. La source n'est pas nécessairement un bloc.

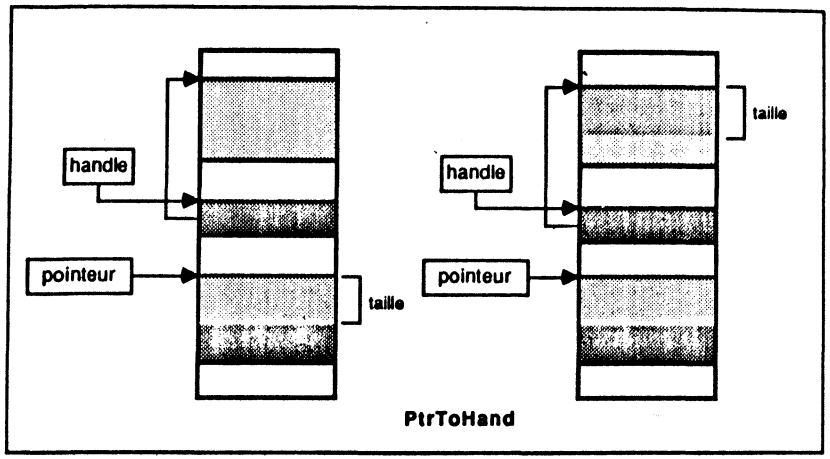


Figure II.7. État mémoire avant et après PtrToHand.

• **HandToPtr** : la source est repérée par un handle, la destination par une adresse (Figure II.8). Le handle doit évidemment exister. Tout ce qui existait à partir de l'adresse de destination est écrasé. La destination n'est pas nécessairement un bloc.

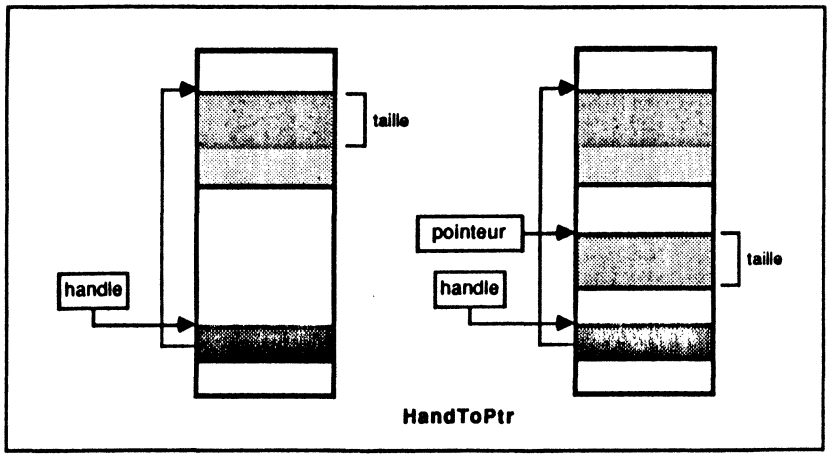


Figure II.8. État mémoire avant et après HandToPtr.

• **HandToHand** : la source est repérée par un handle, la destination par un handle (Figure II.9). Les deux handles doivent exister au moment de l'appel. La logique voudrait que le bloc destination alloué possède une taille supérieure ou égale au nombre d'octets à transférer.

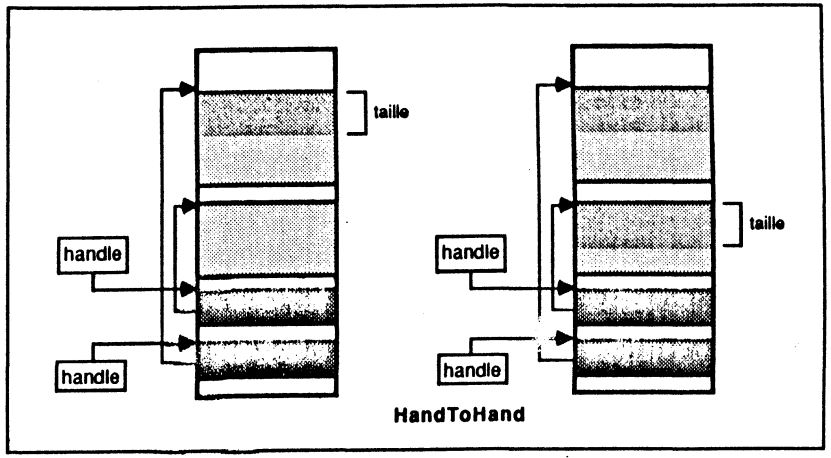


Figure II.9. État mémoire avant et après HandToHand.

• **BlockMove** : la source est repérée par un pointeur, la destination par un pointeur (Figure II.10). Cette procédure assure la copie « sauvage » d'un nombre d'octets déterminés d'un endroit à un autre de la mémoire. Ni la source ni la destination ne sont nécessairement des blocs.

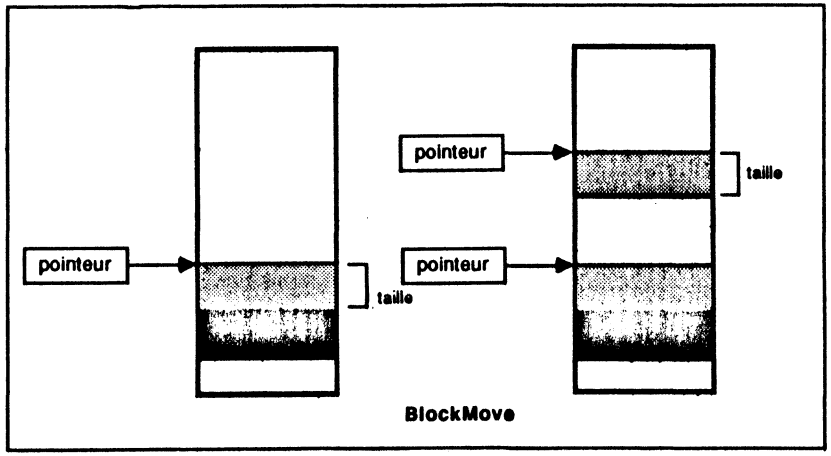


Figure II.10. État mémoire avant et après BlockMove.

Exemple : obtention de handles

Dans le chapitre consacré à QuickDraw, nous allons souvent parler de handle pointant sur une police de caractères ou sur une image. Cet exemple partiel a pour but de montrer comment obtenir un handle sur un objet stocké sur disque. Nous n'insisterons pas sur les fonctions permettant la manipulation de fichiers, car elles sont propres à l'environnement de travail utilisé. Consultez plutôt votre documentation.

On commence par créer le handle sur l'objet à repérer, qui sera relogeable. La taille du bloc repéré par le handle doit correspondre au nombre d'octets à lire (donc à la taille du fichier), les attributs de ce bloc seront fixés de telle sorte qu'ils ne

chevauchent pas deux banques de mémoire (ceci impose une taille inférieure à 64 Ko). Éventuellement, le bloc sera créé de taille quelconque, et celle-ci sera fixée dès qu'elle sera connue.

Ensuite, le handle est verrouillé et déréférencé. Nous obtenons donc un pointeur qui donne une adresse à partir de laquelle nous devons stocker les données lues sur le disque. Une fois la lecture terminée, le handle est déverrouillé et repère la police de caractères ou l'image chargée en mémoire.

Exemple d'une image écran. Une telle image fait forcément 32 Ko, ce pourrait être le résultat d'une sauvegarde particulière issue de GS Paint. Notons que cet exemple est repris de manière concrète à la fin du chapitre V consacré au Window Manager.

```
Handle hdl;                                   /* le handle à créer */
int    myID;                                  /* identifiant de l'application */
int    valid;                                 /* indicateur de validité du chargement */

    main( )
{
...                                           /* début de l'application */
hdl = NewHandle(0x8000L, myID, 0x0010, 0L); /* allocation d'un bloc de 32K */
valid = Load(hdl, "/monDisque/monSousRep/monDessin");      /* chargement des données */
if (valid) ...                          /* on agit en fonction de la validité du chargement */
...                                           /* fin de l'application */
}

    Load(PicDest, path)
Handle  PicDest;                              /* handle (déjà créé) sur les données */
Pointer path;                                 /* chemin d'accès aux données sur disque */
{
int    id;                                    /* identifiant fichier */
int    count;                                 /* nombre d'octets lus */
int    good = TRUE;                           /* indicateur d'erreur */

    HLock(PicDest);                           /* le bloc est verrouillé */
    id = open(path, 0);                       /* le fichier est ouvert... */
    count = read(id, *PicDest, 0x8000);       /* ...et lu (remarquer le déréférencement) */
    if(count != 0x8000) good = FALSE;         /* a-t-on lu le bon nombre d'octets? */
    close(id);                                /* le fichier est fermé */
    HUnlock(PicDest);                         /* le bloc est déverrouillé */
    return good;
}
```

L'application devra faire toutes les vérifications nécessaires pour s'assurer que le fichier a été lu correctement, sous peine de planter plus tard, à l'utilisation du handle, sans raison apparente !

Note Nous verrons dans le chapitre X une section consacrée au Font Manager, qui montre qu'on n'a vraiment pas besoin de se fatiguer à aller chercher des handles sur polices de caractères !

Codes d'erreur

La plupart des routines du Memory Manager sont susceptibles de retourner un code erreur, qu'on pourra récupérer dans la variable *_errno* (au moins dans les environnements Megamax). Le tableau suivant donne le code erreur, un libellé pouvant être défini pour le désigner et une explication. Comme d'habitude, 0 signifie pas d'erreur.

$201 *MemErr*	Impossible d'allouer le bloc : erreur typique pour **NewHandle** s'il n'y a pas assez de mémoire utile, malgré tous les efforts du Memory Manager.

$202 *EmptyErr*	Opération illégale sur un handle vide : tentative d'utilisation d'un handle qui a été purgé auparavant.
$203 *NotEmptyErr*	Un handle vide était attendu pour cette opération : on a essayé de réallouer ou de restaurer un handle non vide.
$204 *LockErr*	Opération illégale sur un bloc verrouillé ou non relogeable : certaines opérations du Memory Manager ne sont valides que sur les blocs pouvant changer d'adresse.
$205 *PurgeErr*	Tentative de purge d'un bloc non purgeable.
$206 *HandleErr*	Un handle invalide a été passé en argument : pour une raison quelconque, le Memory Manager ne reconnaît pas un de ses blocs.
$207 *IDErr*	Un identifiant d'application invalide a été donné.
$208 *AttrErr*	L'opération sur le bloc est incompatible avec ses attributs : on ne peut pas, par exemple, restaurer un handle sur bloc fixe (on peut par contre le réallouer).

SEGMENT LOADER

Pour offrir le maximum de liberté aux programmes tournant sur l'Apple IIGS, le système d'exploitation de cette machine a été doté d'un outil capable de gérer les segments de programmes et de données. Un programme n'a plus à être chargé complètement en mémoire pour pouvoir tourner. Un grand programme pourra être fractionné en segments, ces segments pourront être chargés n'importe où en mémoire, et ce de manière automatique par le Segment Loader. Additionnellement, l'application pourra elle-même gérer ses segments, les charger en mémoire ou les purger de la mémoire à son instigation.

Il existe deux sortes de segments : les segments statiques, chargés en mémoire au début du programme et qui doivent y rester jusqu'à la fin, et les segments dynamiques, chargés au fur et à mesure des besoins, purgés par le Memory Manager quand ils ne sont plus référencés.

Le Segment Loader et le Memory Manager travaillent en étroite collaboration. C'est le Segment Loader qui détecte les segments à charger ou à décharger, et le Memory Manager qui exécute le travail.

Nous n'entrerons pas plus dans les détails : la majorité des petites applications laisseront faire le compilateur, qui se chargera de créer des segments, et le linker, qui générera les informations nécessaires au Segment Loader pour que celui-ci puisse s'y retrouver.

CHAPITRE III

QUICKDRAW

L'interface utilisateur Apple telle qu'elle a été popularisée par le Macintosh repose en grande partie sur un écran tout graphique de très haute définition (fini les modes texte, basse résolution, mixte, etc.). L'Apple IIGS reprenant tous les principes de l'interface Macintosh, il est normal de retrouver le puissant gestionnaire qui en est l'un des piliers : QuickDraw.

En mode émulation, l'Apple IIGS connaît les anciens modes de la famille Apple (texte 40 et 80 colonnes, basse résolution, haute résolution et double haute résolution). Ces modes-là, nous les oublions définitivement pour utiliser exclusivement le tout-graphique géré par QuickDraw, matérialisé par deux nouveaux modes appelés super haute résolution (ou super hi-res) :

- 200 lignes de 640 points en 4 couleurs par ligne ;
- 200 lignes de 320 points en 16 couleurs par ligne.

Dans ces deux modes super haute résolution, tout ce qui apparaît à l'écran est géré par QuickDraw : un dessin, du texte, un menu déroulé, une fenêtre... Tous les gestionnaires devant afficher quelque chose à l'écran (Menu Manager, Window Manager, Control Manager, Line Edit, etc.) appellent QuickDraw de manière interne. Une application appellera explicitement QuickDraw pour définir le contenu d'une fenêtre qu'elle gère, à moins qu'elle n'utilise un gestionnaire qui le fera à sa place. Cas typique : QuickDraw est appelé par le Window Manager pour dessiner une fenêtre, et par l'application directement pour dessiner le contenu de cette fenêtre.

Gardons à l'esprit que tout est dessin en mode super hi-res : quand du texte est affiché, ce texte est dessiné et peut subir des déformations. C'est grâce à cette propriété que l'utilisateur pourra visualiser des jeux de caractères différents, ou styles différents (gras, italique, souligné, ...), à l'intérieur d'un jeu de caractères.

Ce chapitre est divisé en trois grandes parties : la première décrit quelques éléments matériels, la deuxième traite des concepts QuickDraw, la troisième des outils qui permettent réellement de dessiner.

ÉLÉMENTS MATÉRIELS LIÉS AU GRAPHISME

SCB : scan line control byte

Sur l'Apple IIGS, la mémoire écran occupe les 32 Ko allant de $2000 à $9FFF sur la banque $E1. Cette mémoire écran est composée de deux parties : 32 000 octets servant à représenter les 200 lignes de pixels que peut afficher l'écran (une ligne occupe toujours 160 octets), et 768 octets d'informations diverses (dont 512 octets pour stocker 16 tables de couleurs, à partir de l'adresse $9E00). Chaque ligne est caractérisée par une information appelée SCB (scan line control byte), stockée sur un octet (ce qui prend donc 200 octets).

Structure du SCB :

- bits 0 à 3 : numéro d'une palette de couleurs (de 0 à 15) ;
- bit 4 : réservé ;
- bit 5 : mode remplissage (0 = OFF, 1 = ON) ;
- bit 6 : interruption (0 = OFF, 1 = ON) ;
- bit 7 : mode (0 = 320 pixels par ligne, 1 = 640 pixels par ligne).

• Une palette de couleurs est comme son nom l'indique une table définissant les couleurs disponibles sur une ligne. Chaque table contient 16 entrées. Chaque couleur étant codée sur deux octets, une palette de couleurs occupe donc 32 octets.
Remarque Les 16 tables de couleurs différentes peuvent être utilisées simultanément, puisque chaque ligne peut se référer à un SCB différent. On peut donc atteindre la visualisation simultanée de 256 couleurs.

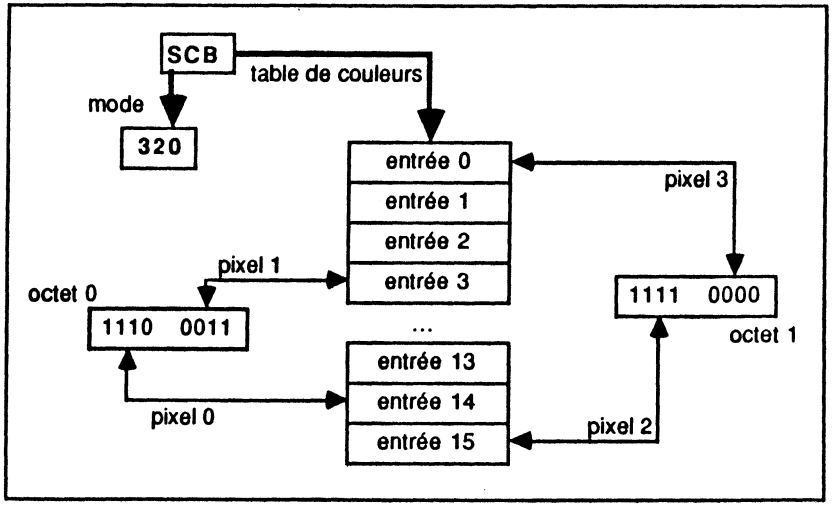


Figure III.1. Le mode 320.

Sur une ligne en mode 320, un pixel occupe 4 bits et leur représentation numérique correspond à une couleur dans la table couleurs. On peut donc avoir jusqu'à 16 couleurs par ligne dans ce mode, chaque pixel pouvant accéder à chaque couleur. Dans un octet, le premier pixel occupe la partie haute et le deuxième pixel la partie basse, ce qui est un ordre naturel : premier pixel à gauche, deuxième pixel à droite.

Sur une ligne en mode 640, un pixel occupe 2 bits. Les quatre pixels d'un octet suivent une disposition naturelle : premier pixel dans les bits 7 et 6, deuxième dans les bits 5 et 4, troisième dans les bits 3 et 2, quatrième dans les bits 1 et 0. Leur

représentation numérique correspond à une couleur dans un sous-ensemble de la palette utilisée : le premier pixel accède aux couleurs 8 à 11, le deuxième aux couleurs 12 à 15, le troisième aux couleurs 0 à 3, le quatrième aux couleurs 4 à 7. De sorte que l'on peut toujours avoir 16 couleurs par ligne, mais les couleurs ne sont plus librement accessibles : il y a une contrainte entre la couleur et la position du pixel dans la ligne.

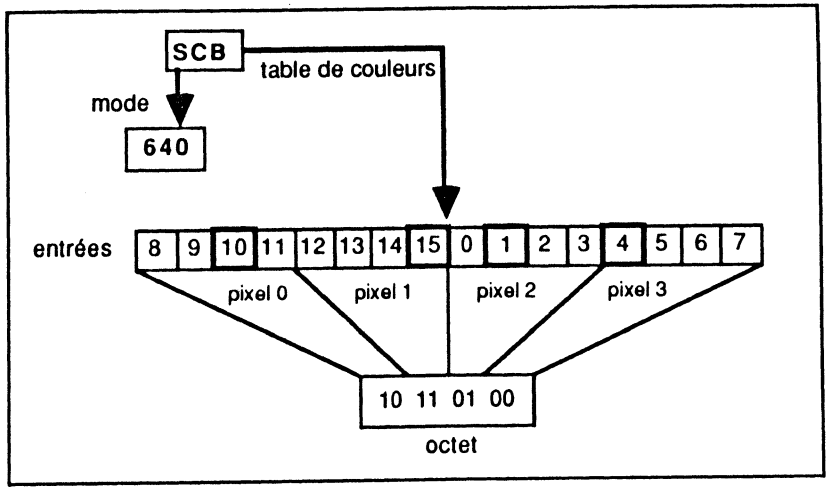


Figure III.2. Le mode 640.

Codage des couleurs : deux octets. Description :

- bits 0 à 3 : niveau de bleu (0 à 15) ;
- bits 4 à 7 : niveau de vert (0 à 15) ;
- bits 8 à 11 : niveau de rouge (0 à 15) ;
- bits 12 à 15 : réservé.

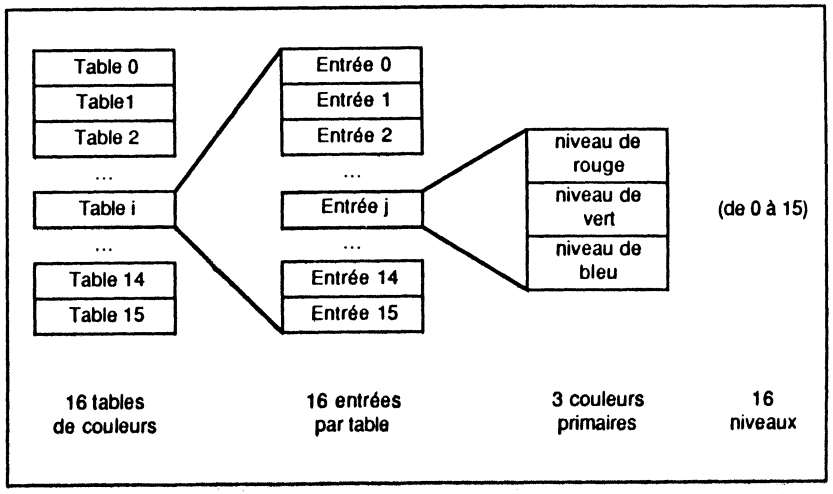


Figure III.3. Tables de couleurs.

Les 16 niveaux de rouge, de vert et de bleu définissent 4 096 couleurs au maximum. Notons que l'absence des trois couleurs primaires se traduit par le noir (couleur $000) et que le mélange maximal des trois couleurs primaires se traduit par le blanc (couleur $FFF). Entre ces deux extrêmes, les différents gris sont obtenus par des niveaux égaux des trois couleurs élémentaires ($111, $222, ..., $EEE).

• Le mode remplissage autorise une alternative au codage des pixels vu précédemment. Si ce mode est actif, la couleur 0 de la table devient inaccessible, et un pixel dont la valeur est à zéro sera affiché avec la même couleur que le pixel précédemment affiché.

Exemple Si les pixels d'une ligne prennent les valeurs :

1 0 0 0 2 0 0 0 0 1 0 0

et si 1 signifie noir, 2 signifie blanc, la ligne sera vue avec les couleurs :

N N N N B B B B B N N N

Note Cette technique n'ayant aucun sens dans un environnement multifenêtres (elle sert au remplissage des lignes), QuickDraw ne propose aucun outil pour l'utiliser. Elle est limitée au mode 320.

• Les interruptions peuvent être utilisées pour synchroniser l'action de dessiner avec le rafraîchissement de l'écran (chaque pixel est redessiné tous les soixantièmes de seconde quand le système tourne à 60 hertz), ou encore pour changer les tables de couleurs avant qu'un écran soit entièrement dessiné, ce qui permet de montrer plus de 256 couleurs à la fois... à condition de gérer lesdites interruptions.

• La résolution (320 ou 640 points par ligne) peut être fixée indépendamment pour chaque ligne. Cependant, on se gardera bien d'utiliser cette possibilité de résolution mixte en environnement multifenêtres : les résultats pourraient s'avérer surprenants.

ROUTINES DE CONTRÔLE

Initialisation de QuickDraw

Comme chaque gestionnaire, QuickDraw doit être initialisé avant l'utilisation d'une quelconque de ses routines. C'est la procédure **QDStartUp** qui se charge du travail :

```
int  zeropg;                /* première page zéro utilisée par QuickDraw */
int  masterSCB;             /* scan line control byte maître */
int  maxWidth;              /* largeur de la plus grande pixel map utilisée */
int  myID;                  /* identifiant de l'application */

QDStartUp(zeropg, masterSCB, maxWidth, myID); /* initialisation de QuickDraw */
...
```

QuickDraw utilise trois pages consécutives dans la banque zéro pour y stocker les renseignements qu'il gère de manière interne : *zeropg* sera l'adresse de la première.

masterSCB définit les caractéristiques de l'environnement, entre autres la table de couleurs à utiliser et le mode 320 ou 640. Cette valeur est notamment utilisée pour l'initialisation de ports graphiques. Deux valeurs seront généralement utilisées : $0000 pour désigner un SCB en mode 320 et $0080 pour désigner un SCB en mode 640 (dans les deux cas, la table de couleurs numéro 0 est retenue, le mode remplissage est OFF, ainsi que les interruptions).

maxWidth est un nombre qui indique la largeur en octets de la plus large des pixels maps qui seront dessinées par l'application (ou zéro pour désigner la largeur de l'écran). Ceci permet à QuickDraw d'optimiser l'allocation de certains buffers internes.

myID est le paramètre retourné par la fonction **MMStartUp** du Memory Manager (voir ce chapitre).

Routines gérant les SCB

Ces routines peuvent être d'une utilisation périlleuse, et méritent d'être testées et approfondies avant d'être employées à tort et à travers.

• La machine possède un SCB standard : il est retourné par la fonction **GetStandardSCB**. Il a les caractéristiques suivantes : la table de couleurs est la table zéro, le mode remplissage est inactif, les interruptions sont inhibées et le mode retenu est le mode 320. En d'autres termes, tous les bits composant le SCB (sauf peut-être le bit 4, sans signification) sont à zéro.

• Le SCB maître peut être retrouvé (il est retourné par la fonction **GetMasterSCB**) ou modifié (par la procédure **SetMasterSCB**). Ces appels seront rarement utilisés, mais à supposer qu'une application doive jongler avec un écran en mode 320 et un écran en mode 640, par exemple, il faut bien pouvoir initialiser les ports correspondants !

• Nous avons dit que chaque ligne pouvait avoir son propre SCB. Au moment de l'initialisation du port, le SCB maître est affecté à toutes les lignes. Ensuite, l'application peut le modifier pour chaque ligne, grâce à la procédure **SetSCB**, qui admet deux arguments : le numéro de la ligne écran concernée (0 à 199) et le SCB à lui fixer. L'effet à l'écran est immédiat. Sauf à vouloir créer des effets non maîtrisés, on se gardera d'utiliser cette possibilité en environnement multifenêtres. Par contre, c'est le seul moyen pour présenter plus de seize couleurs à la fois sur l'écran, dans des utilisations graphiques particulières.

• En complément de la routine précédente, la fonction **GetSCB** retournera le SCB de la ligne écran dont le numéro est passé en argument.

• Pour remettre d'un coup un SCB identique à toutes les lignes écran, on utilisera **SetAllSCBs** (on passe en argument la valeur du SCB à fixer). Là encore, l'effet est immédiat.

Routines gérant les tables de couleurs

• Tout comme il existe un SCB standard, QuickDraw conserve en mémoire une table des couleurs standard, qui est la table utilisée par défaut à l'initialisation. Pour récupérer les valeurs stockées dans cette table, on se réserve un peu de place en mémoire (32 octets exactement) et on passe l'adresse de cette mémoire réservée à la procédure **InitColorTable** qui se charge de la remplir. Voici ce que l'on obtient, en fonction du SCB maître :

MODE 320			MODE 640		
Entrée	Couleur	Code	Entrée	Couleur	Code
0	Noir	$000	0	Noir	$000
1	Gris foncé	$777	1	Rouge	$F00
2	Brun	$841	2	Vert	$0F0
3	Pourpre	$72C	3	Blanc	$FFF
4	Bleu	$00F	4	Noir	$000
5	Vert foncé	$080	5	Bleu	$00F
6	Orange	$F70	6	Jaune	$FF0
7	Rouge	$D00	7	Blanc	$FFF
8	Chair	$FA9	8	Noir	$000
9	Jaune	$FF0	9	Rouge	$F00
10	Vert	$0E0	10	Vert	$0F0
11	Bleu clair	$4DF	11	Blanc	$FFF

12	Lilas	$DAF	12	Noir	$000
13	Bleu pervenche	$78F	13	Bleu	$00F
14	Gris clair	$CCC	14	Jaune	$FF0
15	Blanc	$FFF	15	Blanc	$FFF

Note Ces palettes de couleurs n'ont pas été choisies au hasard par Apple. Pour le mode 640, on constate que le noir et le blanc figurent dans les quatre quarts de la palette, ils seront donc accessibles quelle que soit la position du pixel. Par contre, on ne pourra avoir deux pixels consécutifs de même couleur parmi le rouge, le vert, le bleu et le jaune. Mais si l'on considère deux pixels consécutifs comme « associés », on retrouve 16 pseudo-couleurs : noir-rouge, noir-vert, noir-bleu, noir-jaune, blanc-rouge, blanc-vert, blanc-bleu, blanc-jaune, rouge-bleu, rouge-jaune, vert-bleu, vert-jaune, noir-noir, blanc-blanc, noir-blanc, blanc-noir. Ce qui permet de donner l'illusion qu'on est en mode 640 pour tout ce qui est noir ou blanc (les textes sont donc impeccables), et en mode 320 pour tout ce qui est couleur (avec 16 pseudo-couleurs).

• Puisque nous pouvons gérer jusqu'à seize tables de couleurs, QuickDraw nous offre deux procédures, **GetColorTable** pour connaître le contenu d'une table, **SetColorTable** pour remplir une table. Dans les deux cas, nous utilisons en argument : d'abord le numéro de la table (compris entre 0 et 15), ensuite l'adresse de la zone de stockage du contenu (32 octets).

Exemple Mise en place de trois tables de couleurs (mode 320).

```
int  table0[16];
int  table1[16] = {0, 0x888, 0x952, 0x83D, 0x11F, 0x191, 0xF81, 0xE11,
                   0xFBA, 0xFF1, 0x1F1, 0x5EF, 0xEBF, 0x89F, 0xDDD, 0xFFF};
int  table2[16] = {0, 0x666, 0x730, 0x61B, 0xE, 0x70, 0xE60, 0xC00,
                   0xE98, 0xEE0, 0xD0, 0x3CE, 0xC9E, 0x67E, 0xBBB, 0xFFF};

InitColorTable(table0);          /* retourne les couleurs standard */
SetColorTable(0, table0);        /* mise en place de la table standard */
SetColorTable(1, table1);        /* mise en place de la table numéro 1 */
SetColorTable(2, table2);        /* mise en place de la table numéro 2 */
```

Remarque Les deux tables de l'exemple présentent la particularité suivante :

– chaque couleur de la table 2 se déduit de la couleur correspondante de la table 0 par la baisse d'une unité (quand c'est possible) de l'intensité de chaque couleur élémentaire, le blanc étant épargné. Ainsi $777 dans la table 0 devient $666 dans la table 2. Cette astuce permet d'avoir deux palettes de couleurs dont l'une est l'assombrissement de l'autre.

– chaque couleur de la table 1 se déduit de la couleur correspondante de la table 0 par la hausse d'une unité (quand c'est possible) de l'intensité de chaque couleur élémentaire, le noir étant épargné. Ainsi $777 dans la table 0 devient $888 dans la table 1. Cette astuce permet d'avoir deux palettes de couleurs dont l'une est l'éclaircissement de l'autre.

Nous allons voir tout de suite (point suivant) une manière plus élégante (et plus automatique) de faire la même chose. Pourquoi préserver les couleurs blanches et noires ? Tout simplement pour ne pas affecter la couleur des objets « système » (texte des menus, encadrement des fenêtres, boutons, etc.) quand la nouvelle table sera utilisée.

Mais auparavant, constatons qu'il est très facile de copier le contenu d'une table dans une autre table.

Exemple Recopie de la table 6 dans la table 7.

```
char    tampon[32];              /* réserve 32 octets de mémoire tampon */

GetColorTable(6, tampon);        /* remplit la zone tampon */
SetColorTable(7, tampon);        /* utilise la zone tampon */
```

Remarque Dans l'exemple précédent, nous avions déclaré la table comme étant composée de 16 éléments et de 2 octets (int table0[16]) et ici comme étant composée de 32 éléments d'un octet (char tampon[32]). Ces deux déclarations sont évidemment équivalentes, la première étant absolument obligatoire si on doit accéder à une couleur directement.

• QuickDraw nous permet enfin d'accéder directement à une couleur dans une table déterminée. La fonction **GetColorEntry** permet de connaître une couleur particulière d'une table, la procédure **SetColorEntry** de fixer une couleur dans une table. Nous allons utiliser ces routines pour assombrir ou éclaircir de manière automatique la table des couleurs standard. C'est plus long à écrire, mais c'est plus élégant, et d'utilisation générale.

```
int  couleur;                    /* sera la couleur manipulée */
int  i;

for (i=1; i<15; ++i)             /* i varie de 1 à 14: le noir et le blanc sont préservés */
    {
    couleur = GetColorEntry(0, i);  /* on va chercher la couleur i de la table 0 */
    if ((couleur & 0x0F00) != 0x0F00) couleur += 0x0100;   /* niveau de rouge augmenté */
    if ((couleur & 0x00F0) != 0x00F0) couleur += 0x0010;   /* niveau de vert augmenté */
    if ((couleur & 0x000F) != 0x000F) couleur += 0x0001;   /* niveau de bleu augmenté */
    SetColorEntry(1, i, couleur);   /* stocke la couleur modifiée dans la table 1, entrée i */
    }

for (i=1; i<15; ++i)             /* i varie de 1 à 14: le noir et le blanc sont préservés */
    {
    couleur = GetColorEntry(0, i);  /* on va chercher la couleur i de la table 0 */
    if ((couleur | 0xF0FF) != 0xF0FF) couleur -= 0x0100;   /* niveau de rouge diminué */
    if ((couleur | 0xFF0F) != 0xFF0F) couleur -= 0x0010;   /* niveau de vert diminué */
    if ((couleur | 0xFFF0) != 0xFFF0) couleur -= 0x0001;   /* niveau de bleu diminué */
    SetColorEntry(2, i, couleur);   /* stocke la couleur modifiée dans la table 2, entrée i */
    }
```

Il ne reste plus qu'à utiliser ces tables, pour créer des effets spéciaux. Le dessin original sera éclairci ou assombri uniquement par le jeu des tables de couleurs.

```
int  scb;                        /* sera le code du SCB courant */

scb = 0;                         /* SCB standard, mode 320, palette de couleurs n° 0 */
SetAllSCBs(scb);                 /* toutes les lignes sont au standard */
...                              /* dessin dans ce mode */
SetAllSCBs(scb+1);               /* passage instantané à la palette 1: éclaircissement */
...                              /* temporisation (voir chapitre IV) */
SetAllSCBs(scb);                 /* retour instantané à la palette 0: assombrissement */
...                              /* temporisation */
SetAllSCBs(scb+2);               /* passage instantané à la palette 2: assombrissement */
...                              /* temporisation */
SetAllSCBs(scb);                 /* retour instantané à la palette 0: éclaircissement */
```

Les tables de couleurs recèlent des propriétés qui seront découvertes au fur et à mesure de l'utilisation de QuickDraw. Un ouvrage entier pourrait être consacré à ce gestionnaire aux richesses insoupçonnables, ce qui n'est pas notre propos... Il sera par exemple très facile de faire apparaître ou disparaître un objet, rien qu'en jouant avec les tables de couleurs.

CONCEPTS QUICKDRAW

QuickDraw est constitué d'un ensemble de routines permettant de manipuler dans un environnement graphique qui lui est propre un certain nombre d'objets prédéfinis. Le but de cette partie consiste à découvrir quel est cet environnement graphique et quels sont ces objets.

FONDATIONS MATHÉMATIQUES

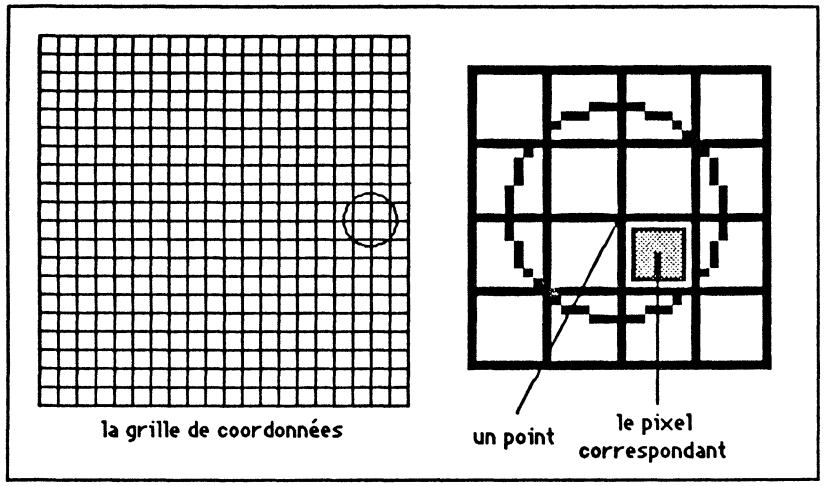


Figure III.4. Concepts de base de QuickDraw.

Plan de coordonnées

On peut l'assimiler à une grille bi-dimensionnelle formée d'un nombre *fini* de lignes *infiniment fines*. La grille est de dimension finie, les coordonnées étant définies comme des nombres entiers compris entre − 16384 et + 16383. L'axe des abscisses est classiquement orienté de la gauche vers la droite, l'axe des ordonnées moins classiquement du haut vers le bas. Cette orientation non mathématique suit pourtant une ligne logique implacable : nous écrivons de gauche à droite et de haut en bas. Les lignes composant la grille sont infiniment fines, ce qui signifie qu'elles n'ont aucune existence physique : elles ne servent qu'à définir un système de coordonnées.

Point

Chaque intersection des lignes de la grille définit un point. Puisque les lignes sont infiniment fines, le point est infiniment petit, également sans existence physique. Le plan de coordonnées contient ainsi plus d'un milliard de points, l'origine (0,0) se trouvant en plein milieu de la grille. Le coin supérieur gauche de l'écran correspond à cette origine.

Ligne

Deux points quelconques définissent une ligne (en réalité un segment de droite). La ligne est constituée de points, et du fait que le nombre des points est fini, ceux-ci ne sont pas forcément rigoureusement alignés.

Rectangle

Deux points quelconques peuvent définir un rectangle : l'un étant le haut-gauche du rectangle, l'autre le bas-droit (condition : les côtés du rectangle doivent être horizontaux et verticaux). Le rectangle, comme la ligne, n'est qu'un concept mathématique, puisque les côtés qui le délimitent sont infiniment fins. Un rectangle de taille n × m englobe exactement (n−1) × (m−1) pixels.

Le rectangle est un objet essentiel de la philosophie QuickDraw, comme nous le verrons tout au long de ce chapitre : il servira notamment à délimiter des régions complexes.

Rectangle arrondi

Pour définir un rectangle arrondi, il suffit d'ajouter à la définition d'un rectangle normal deux rayons de courbure : un rayon de courbure horizontal et un rayon de courbure vertical. QuickDraw saura gérer de tels objets.

Ellipse (ou ovale)

Une ellipse est parfaitement définie par le rectangle circonscrit (on détermine immédiatement le demi petit axe et le demi grand axe). Un rectangle lui étant donné, QuickDraw se débrouillera tout seul.
Remarque Pour obtenir un cercle parfait, il suffit que le rectangle circonscrit soit un carré.

Arc

Si pour une ellipse donnée nous déterminons un angle d'origine et un angle d'arc, nous obtenons une part de camembert chère aux graphiques de gestion. QuickDraw sait manipuler les parts du camembert.

Région

Le concept de région est sans doute le concept le plus puissant de QuickDraw. Une région, c'est un ensemble quelconque de points définissant une structure cohérente. Elle peut être concave ou convexe, connexe ou disjointe, pleine ou évidée... QuickDraw offre les outils pour créer et manipuler les régions, et ce avec des temps de réponse suffisamment performants pour que le concept présente un quelconque intérêt. Nous verrons ces outils et leur utilisation concrète. Une façon d'imaginer une région, c'est de prendre le plus petit rectangle qui l'englobe, et de définir à l'intérieur deux sortes de points : ceux qui appartiennent à la région, et ceux qui lui sont étrangers. Nous avons parlé de points et non de pixels : la région est un concept mathématique.

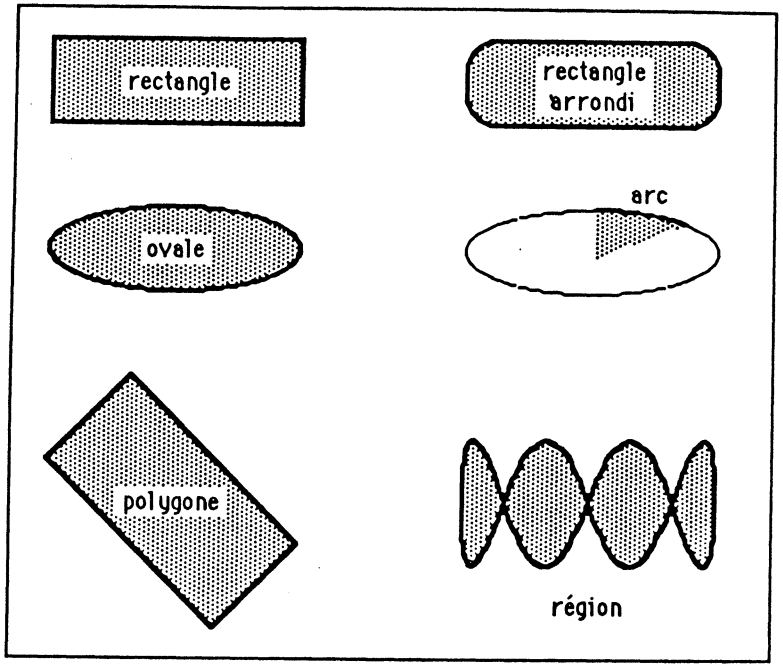


Figure III.5. Concepts géométriques.

Polygone

Un polygone est défini par un ensemble de points dont l'ordre est primordial (le dernier point coïncidant avec le premier) : les lignes qui rejoignent deux points consécutifs séparent le plan en deux : les points qui sont intérieurs au polygone et ceux qui lui sont extérieurs. Les lignes pouvant se croiser, la distinction n'est pas toujours évidente à faire. QuickDraw, lui, ne se trompera pas.

ENTITÉS GRAPHIQUES

Pixel

Chaque point (infiniment petit) définit un pixel qui, lui, a une réalité physique. Un pixel est constitué de la surface comprise entre deux lignes horizontale et verticale consécutives, ses coordonnées étant fixées par le point situé à son angle supérieur gauche. Un pixel possède une couleur. La technologie *bitmap* voulant que chaque point soit adressable individuellement, à un pixel correspond de manière très précise un certain nombre de bits en mémoire. 2 bits permettent de définir 4 couleurs, donc un pixel sera constitué de 2 bits dans le mode 640 × 200. De même, 4 bits permettent de définir 16 couleurs, donc un pixel sera constitué de 4 bits dans le mode 320 × 200. Dans les deux cas, une ligne d'écran aura la même taille : 160 octets (sur Macintosh, l'écran monochrome ne permet que 2 couleurs, le noir et le blanc : dans ce cas très particulier, un pixel équivaut à un bit).

Pixel map

Une pixel map est une portion de mémoire contenant une image graphique composée de pixels, chaque pixel ayant la même taille (2 ou 4 bits). QuickDraw sait dessiner n'importe où, et pas seulement dans la mémoire écran. Il suffit de lui dire où dessiner. Le fait de dessiner ne consiste réellement qu'en la modification de pixels au sein d'une pixel map, et la mémoire écran n'est qu'une pixel map particulière, située à une adresse fixe en mémoire.

Une étendue de mémoire représente quelque chose de linéaire : tous les pixels sont mis bout à bout. Dans la réalité, un dessin est un objet à deux dimensions. Le passage entre une pixel map et sa représentation plane s'effectue par l'apport d'éléments supplémentaires, stockés dans une structure dédiée appelée *LocInfo*.

Définition de *LocInfo* :

```
struct _LocInfo {
    int       PortSCB ;          /* SCB pour la pixel map (dans l'octet bas) */
    Pointer   baseAddr ;         /* pointeur sur la pixel map */
    int       rowBytes ;         /* largeur de l'image, en octets */
    Rect      BoundsRect ;       /* rectangle pour la pixel map */
    };
#define LocInfo struct _LocInfo
```

PortSCB désigne pour la pixel map à la fois la palette de couleurs utilisée et la taille de chaque pixel (en fonction de la résolution annoncée). *baseAddr* désigne l'adresse où débute la pixel map en mémoire. *rowBytes* désigne le nombre d'octets contenus dans chaque ligne de pixels (ce nombre doit obligatoirement être un multiple de 8). Enfin, *BoundsRect* est un rectangle qui détermine l'étendue exacte de la pixel map et qui lui impose un système de coordonnées, dites globales.

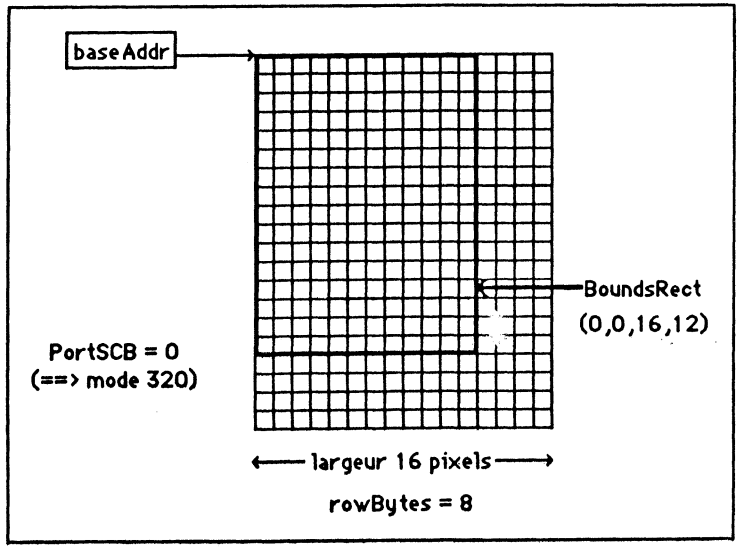


Figure III.6. La structure LocInfo.

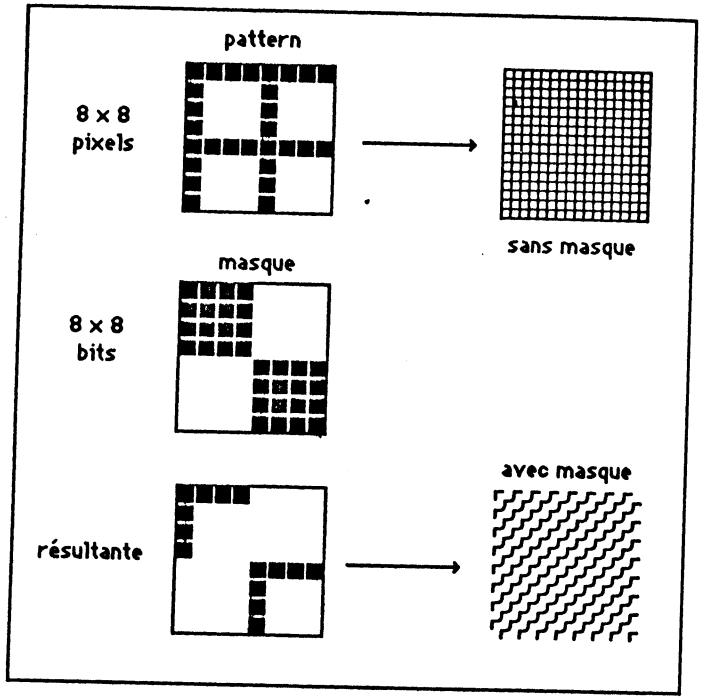


Figure III.7. Pattern et masque.

Patterns et masques

Un pattern est une pixel map carrée composée de 8 × 8 pixels, utilisée comme motif de remplissage (« couleur » des objets, d'un fond écran) ou comme encre (« couleur » du crayon). Les motifs répétés sont dessinés de telle manière qu'ils se raccordent parfaitement, formant ainsi une trame parfaite. Les couleurs sont des patterns un peu particuliers, dits patterns solides (tous les pixels du pattern sont de la même couleur).

Un masque de dessin est un carré de 8 × 8 bits utilisé pour masquer le pattern sélectionné : chaque 1 du masque laissera passer le pixel associé du pattern, chaque 0 empêchera le pixel de passer (c'est-à-dire que seuls les pixels du pattern associés à un 1 dans le masque seront dessinés, les autres restant de la couleur du fond).

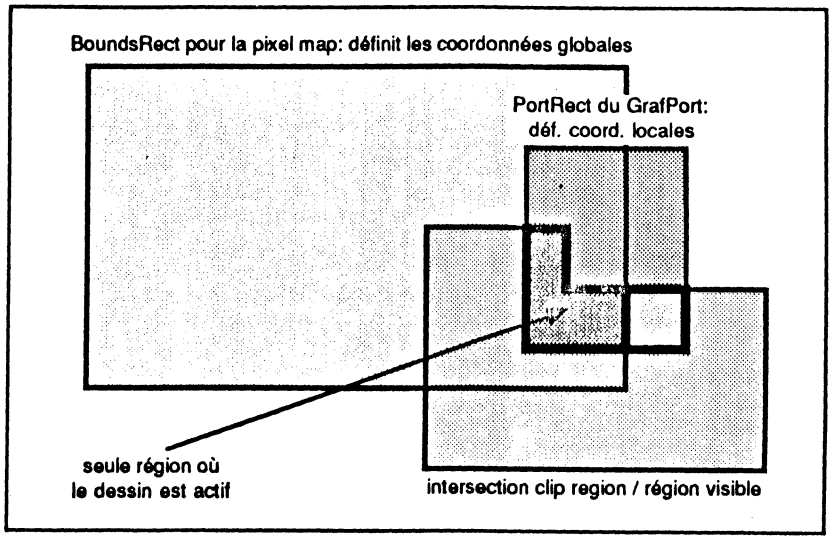


Figure III.8. Les régions du grafport.

Grafport

Pour dessiner, il est nécessaire de définir un environnement graphique. Le grafport est une structure qui définit complètement un tel environnement.

Le grafport est composé d'un grand nombre d'éléments, dont les plus importants sont :

• Une pixel map avec ses données additionnelles permettant la représentation du dessin dans un plan (autrement dit une structure *LocInfo* vue plus haut, appelons-la *PortInfo*. Nous avons dit qu'un système de coordonnées était attaché à une telle structure : pour le grafport, il s'agit du *système de coordonnées globales.*

• Un rectangle appelé *PortRect* qui désigne une partie de la pixel map : c'est dans ce rectangle exclusivement que s'effectueront les opérations graphiques. Ce rectangle impose un second système de coordonnées, appelé *système de coordonnées locales.* Quand nous étudierons le Window Manager, nous verrons que la partie « contenu » d'une fenêtre n'est autre que la visualisation du contenu du *PortRect* associé au grafport de la fenêtre en question. (Voir le chapitre XI.)

• Deux régions : la *clip region* et la région visible, qui limitent le champ d'application de certains outils de dessin.

La clip region est une région (au sens QuickDraw) délimitant le lieu où les outils de dessin agissent : il est impossible de dessiner un pixel en dehors de la clip region. Toute tentative de dessin en dehors de la clip region se soldera par une absence de résultat. L'exemple classique d'utilisation de cette région peut être ici rappelé : pour dessiner un demi-cercle, il suffit de dessiner un cercle entier, la moitié du carré d'accueil étant clippée, l'autre pas.

La région visible est également une région au sens QuickDraw, délimitant le lieu où le dessin est visible de celui où il ne l'est pas. Pour revenir au concept de fenêtres, il est possible qu'une d'entre elles cache partiellement le contenu d'une autre. On a là un exemple de restriction de la région visible.

Ces deux concepts ne doivent pas être confondus : quand une application voudra restreindre le champ de son dessin, elle modifiera la clip region, et non pas la région visible.

D'ores et déjà, nous pouvons faire une constatation : il n'est pas possible de dessiner n'importe où (voir figure III.8). On ne peut dessiner que dans un grafport, et dans l'intersection de deux rectangles et de deux régions, il faut être dans la région visible et dans la clip region, il faut être dans le rectangle *PortRect* du grafport, mais aussi dans le rectangle frontière qui délimite la pixel map (le *BoundsRect* du *PortInfo*, pour parler en charabia).

- Un pattern de fond, dit *bkPat* (remplissage du *PortRect* à l'initialisation, effacement de formes).

- Un crayon possédant un certain nombre de caractéristiques : une localisation, une taille, un mode de transfert, un pattern et un masque. De plus, il peut être visible ou invisible.

- Une police de caractères utilisée pour dessiner du texte, avec également un certain nombre de caractéristiques : une taille, un style, un mode de transfert, une couleur pour le corps des lettres *(ForeColor)*, une couleur pour le fond des lettres *(BackColor)*.

- Une zone d'utilisation libre (4 octets) pour l'application, appelée *UserField*.

- Plusieurs zones d'utilisation interne au système.

Il est inutile d'attacher une importance particulière à la définition exacte du grafport en terme de structure : QuickDraw propose tout un jeu de sous-programmes permettant de modifier les caractéristiques du grafport courant, sans qu'il y ait à connaître le moindre nom de champ de cette structure. Il est même fortement déconseillé de vouloir modifier directement le contenu de l'un des champs : passer par les routines adéquates permet de s'affranchir des problèmes de compatibilité avec les versions ultérieures de QuickDraw. Les caractéristiques intéressantes du grafport et les routines permettant de les modifier sont décrites dans le présent chapitre.

Retenons simplement que dans l'environnement de travail du programmeur, sera sans doute défini un type nommé *GrafPort* et que, quand on déclarera une variable de type *GrafPort*, l'espace nécessaire pour stocker les caractéristiques de ce grafport (170 octets) sera automatiquement réservé.

Curseur

Le curseur est un objet graphique de dimension quelconque qui sert principalement à indiquer le lieu de l'écran où pointe la souris, et qui donc suit fidèlement tous ses déplacements. De manière plus précise, c'est l'un des pixels du curseur qui localise la position de la souris. Le point associé est appelé *hotspot* ou point chaud.

Attention Le curseur est souvent appelé pointeur, car sa fonction est de désigner l'endroit où un clic souris peut intervenir. Ce terme de pointeur n'a rien à voir évidemment avec celui que nous employons sans arrêt en programmation.

La structure de type *Cursor* est une structure à taille variable, dont la définition pourrait être la suivante :

```
struct _Cursor {
        int     CursorHeight;       /* hauteur du curseur, en pixels */
        int     CursorWidth;        /* largeur du curseur, en nombre de mots */
        char    CursorImage[][];    /* image du curseur */
        char    CursorMask[][];     /* masque du curseur */
        int     HotSpotY;           /* ordonnée du point chaud */
        int     HotSpotX;           /* abscisse du point chaud */
        };
#define Cursor struct _Cursor
```

La hauteur du curseur est définie en nombre de pixels. Pas de problème ici, c'est le nombre de lignes de l'image du curseur. Plus compliquée est la définition de la largeur : l'image est constituée d'un certain nombre de pixels en largeur, dont la taille dépend du mode de résolution. De plus, chaque ligne doit être terminée par un mot de deux octets à zéro. La « largeur » du curseur est le nombre de mots nécessaires pour définir une telle ligne.

L'image du curseur est une pixel map classique, avec le dernier mot de chacune de ses lignes à zéro. Le masque servira à combiner l'image du curseur et celle sur laquelle il passe, ce qui permettra de fabriquer des curseurs partiellement transparents, par exemple (le dernier mot de chacune de ses lignes doit aussi être à zéro).

Les coordonnées du point chaud sont données par rapport au coin supérieur gauche de l'image du curseur (cette origine a évidemment (0,0) pour coordonnées).

Exemple de déclaration : un curseur tout noir en forme de flèche en mode 320 (voir Figure III.9). Il s'agit très exactement du curseur système qu'on obtient par défaut.

```
Cursor arrow = {
    11,                         /* hauteur de l'image (nombre de lignes) */
    4,                          /* largeur de l'image (nombre de mots, donc 8 octets) */
                                                    /* image du curseur */
    {
    { 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00 },
    { 0x0F,0x00,0x00,0x00,0x00,0x00,0x00,0x00 },
    { 0x0F,0xF0,0x00,0x00,0x00,0x00,0x00,0x00 },
    { 0x0F,0xFF,0x00,0x00,0x00,0x00,0x00,0x00 },
    { 0x0F,0xFF,0xF0,0x00,0x00,0x00,0x00,0x00 },
    { 0x0F,0xFF,0xFF,0x00,0x00,0x00,0x00,0x00 },
    { 0x0F,0xFF,0xFF,0xF0,0x00,0x00,0x00,0x00 },
    { 0x0F,0xFF,0xFF,0xFF,0x00,0x00,0x00,0x00 },
    { 0x0F,0xF0,0xFF,0x00,0x00,0x00,0x00,0x00 },
    { 0x00,0x00,0x0F,0xF0,0x00,0x00,0x00,0x00 },
    { 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00 }
        },
                                                    /* image du masque */
        {
    { 0xFF,0x00,0x00,0x00,0x00,0x00,0x00,0x00 },
    { 0xFF,0xF0,0x00,0x00,0x00,0x00,0x00,0x00 },
    { 0xFF,0xFF,0x00,0x00,0x00,0x00,0x00,0x00 },
    { 0xFF,0xFF,0xF00,0x0,0x00,0x00,0x00,0x00 },
    { 0xFF,0xFF,0xFF,0x00,0x00,0x00,0x00,0x00 },
    { 0xFF,0xFF,0xFF,0xF0,0x00,0x00,0x00,0x00 },
    { 0xFF,0xFF,0xFF,0xFF,0x00,0x00,0x00,0x00 },
    { 0xFF,0xFF,0xFF,0xFF,0xF0,0x00,0x00,0x00 },
    { 0xFF,0xFF,0xFF,0xFF,0x00,0x00,0x00,0x00 },
    { 0xFF,0xF0,0xFF,0xFF,0x00,0x00,0x00,0x00 },
    { 0x00,0x00,0x0F,0xFF,0x00,0x00,0x00,0x00 }
        },
    1, 1                        /* coordonnées du point chaud */
    };
```

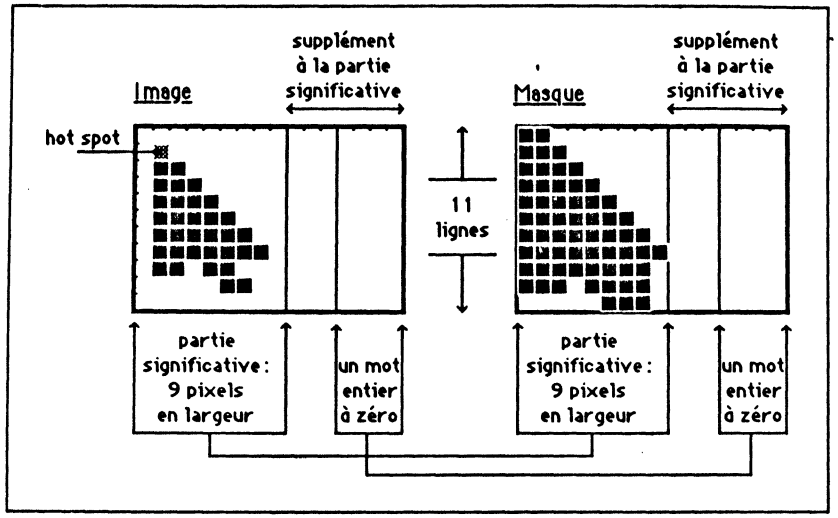


Figure III.9. Le curseur.

Le malheur, quand on utilise ce type de structure, c'est que dès qu'un élément est déclaré, il en fixe la taille. Ainsi, après la déclaration du curseur de l'exemple précédent, tous les autres curseurs sont condamnés à avoir 11 lignes de quatre mots dans leur définition, ce qui peut être préjudiciable pour une application qui voudrait gérer plusieurs curseurs de tailles différentes.

Dans ce cas, trois solutions : utiliser en C des types différents pour chaque taille de curseur, utiliser des modules en assembleur pour définir les curseurs, ou faire de la déclaration sauvage en C, sans aucune structuration. Le curseur précédent peut très bien être déclaré comme une chaîne de caractères (en fait des entiers sur 8 bits), sans se soucier d'une quelconque définition de structure, sous réserve de quelques précautions : les entiers sur 16 bits que comporte la structure doivent chacun être définis comme 2 caractères de 8 bits, la moitié la moins significative avant la moitié la plus significatve (puisque c'est ainsi que sont représentés les mots de 16 bits en mémoire).

```
char arrow[ ] = { 11, 0,  4, 0,            /* 11 lignes de 4 mots */
    0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x0F,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
    0x0F,0xF0,0x00,0x00,0x00,0x00,0x00,0x00,0x0F,0xFF,0x00,0x00,0x00,0x00,0x00,0x00,
    0x0F,0xFF,0xF0,0x00,0x00,0x00,0x00,0x00,0x0F,0xFF,0xFF,0x00,0x00,0x00,0x00,0x00,
    0x0F,0xFF,0xFF,0xF0,0x00,0x00,0x00,0x00,0x0F,0xFF,0xFF,0xFF,0x00,0x00,0x00,0x00,
    0x0F,0xF0,0xFF,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x0F,0xF0,0x00,0x00,0x00,0x00,
    0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
    0xFF,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0xFF,0xF0,0x00,0x00,0x00,0x00,0x00,0x00,
    0xFF,0xFF,0x00,0x00,0x00,0x00,0x00,0x00,0xFF,0xFF,0xF00,0x0,0x00,0x00,0x00,0x00,
    0xFF,0xFF,0xFF,0x00,0x00,0x00,0x00,0x00,0xFF,0xFF,0xFF,0xF0,0x00,0x00,0x00,0x00,
    0xFF,0xFF,0xFF,0xFF,0x00,0x00,0x00,0x00,0xFF,0xFF,0xFF,0xFF,0xF0,0x00,0x00,0x00,
    0xFF,0xFF,0xFF,0xFF,0x00,0x00,0x00,0x00,0xFF,0xF0,0xFF,0xFF,0x00,0x00,0x00,0x00,
    0x00,0x00,0x0F,0xFF,0x00,0x00,0x00,0x00,
      1, 0, 1, 0 };                        /* point chaud en (1,1) */
```

C'est évidemment un peu plus touffu, mais ça donne le même résultat ! Attention toutefois à l'emploi : dans le premier cas, on passera l'adresse de la définition du curseur en utilisant &arrow, tandis que dans le second cas, arrow suffira (plus besoin de l'opérateur &).

Picture

Il y a au moins deux manières de mémoriser une image. On peut soit conserver le résultat final sous forme de pixel map, soit stocker les différentes actions qui s'enchaînent dans la constitution d'un dessin. Une *picture* (au sens QuickDraw du terme), c'est un dessin mémorisé suivant la deuxième méthode.

La mémorisation des actions constitutives d'une image présente plusieurs avantages sur celle de l'image finie :

– elle est généralement plus concise (dans le sens où elle tient moins de place en mémoire). Cela est vrai surtout pour les dessins simples ou de petite taille, et se vérifiera aisément en comparant la taille des fichiers résultant de GS-Paint et GS-Draw (pour des images comparables) ;

– elle est apte à subir des déformations de taille sans que cela altère la qualité finale du dessin : agrandir ou rétrécir une image modifie la forme des objets qu'elle contient, mais n'altère ni l'épaisseur de leurs traits, ni l'uniformité de leurs motifs de remplissage ;

– chaque instruction QuickDraw mémorisée possède un équivalent PostScript (le langage de composition de l'imprimante LaserWriter), ce qui permet l'impression du dessin à la résolution de l'imprimante (malheureusement en noir et blanc pour l'instant), et non à celle de l'écran.

Il n'est pas toujours évident de faire un dessin complet en utilisant des ordres QuickDraw simples (tels qu'ils seront décrits dans la troisième partie de ce chapitre). Aussi une image peut-elle contenir à son tour des dessins finis sous forme de pixel map, et un ordre simple de constitution de l'image finale sera de dessiner ces parties déjà mémorisées.

Attention à ne pas confondre région et picture. La région est un concept mathématique qui permettra de manipuler des objets compliqués (la région mémorise les éléments constitutifs d'une forme qui sera utilisée dans des dessins), la picture est un élément graphique achevé (elle mémorise les éléments constitutifs d'un dessin, par exemple le dessin d'une région). Seule la manière dont une picture est mémorisée peut entretenir cette possibilité de confusion.

Modes de transfert

Quand un stylo doit dessiner sur une feuille de papier, généralement son encre fait complètement disparaître la couleur originale de la feuille. En informatique, on a plusieurs possibilités pour composer la couleur du crayon et celle du fond sur lequel il vient dessiner. Ces possibilités sont déterminées par les modes de transfert. Sur l'Apple IIGS, on distinguera deux catégories de modes de transfert : ceux qui s'appliquent au dessin, y compris le dessin du texte (transfert d'une couleur sur une autre couleur), et ceux qui s'appliquent au texte exclusivement (passage d'un caractère à un bit par pixel à un caractère dessiné, sans se soucier du fond sur lequel il est dessiné).

Modes de transfert crayon

Ils sont au nombre de huit. Dans les tables et la figure qui suivent, nous donnons les transformations binaires propres à chaque mode. Ces transformations affectent individuellement chaque bit du pixel, et non une couleur en entier, ce qui peut conduire à des résultats inattendus en fonction des palettes de couleurs utilisées. Pour ce qui est du texte, à la fois les pixels du premier plan et les pixels du fond sont affectés.

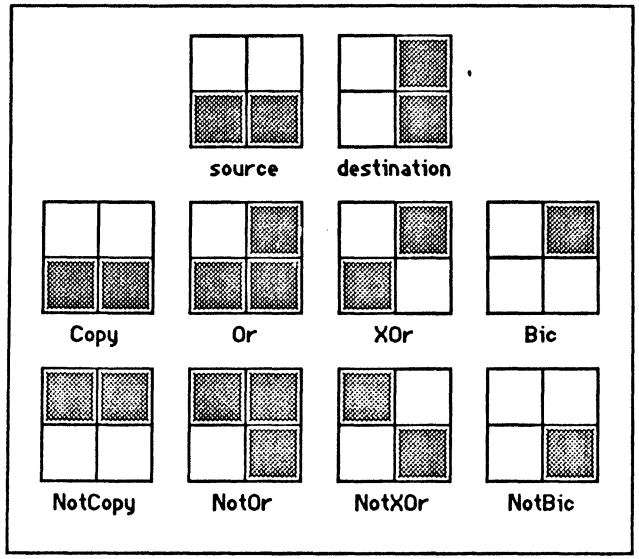


Figure III.10. Les modes de transferts normaux (blanc = 0, gris = 1).

• Modes *Copy* et *NotCopy :* copie la couleur source (ou sa négation) vers la destination. *Copy* est le mode typique du dessin, puisqu'il agit comme un stylo sur une feuille de papier.

• Modes *Or* et *NotOr :* superposition de la source (ou sa négation) sur la destination. On peut utiliser ces modes pour une superposition non destructive d'images (normales ou inversées) sur d'autres images.

• Modes *XOr* et *NotXOr :* « ou » exclusif entre la source (ou sa négation) et la destination. Ces deux modes sont idéaux pour le dessin du curseur ou des silhouettes d'objets à déplacer, puisqu'il suffit d'appliquer une deuxième fois la source sur le résultat pour rétablir le dessin original.

• Modes *Bic* et *NotBic :* « et » logique entre la négation de la source (ou la source) et la destination. Le mode *Bic* sert à effacer des pixels avant de superposer la source à la destination, le mode *NotBic* peut être utilisé pour représenter l'intersection de deux images.

	source	dest	Copy 0000	NotCopy 8000	Or 0001	NotOr 8001	XOr 0002	NotXOr 8002	Bic 0003	NotBic 8003
Code hexa										
	0	0	0	1	0	1	0	1	0	0
	0	1	0	1	1	1	1	0	1	0
	1	0	1	0	1	0	1	0	0	0
	1	1	1	0	1	1	0	1	0	1
décimal	3	5	3	12	7	13	6	9	4	1
décimal	12	10	12	3	14	11	6	9	2	8

Dans le tableau récapitulatif, nous trouvons le code hexadécimal qui sert à désigner chaque mode de transfert, le résultat bit à bit de chaque mode de transfert, et enfin ce que cela donne sur deux couples arbitraires de couleurs. Par exemple, copier la couleur 3 sur la couleur 5 en mode *XOr* donnera la couleur 6, ce qui signifie avec la palette standard : pourpre sur vert foncé donne orange.

Modes de transfert spéciaux au texte

Ils sont également au nombre de huit. Ils s'appliquent quand on dessine des caractères (2 ou 4 bits par pixel) à partir de leur représentation « noir et blanc » (1 bit par pixel), telle qu'elle est stockée dans la définition de la police de caractères. En aucun cas les pixels du fond ne sont affectés.

• Modes *ForeCopy* ($0004) et *NotForeCopy* ($8004) : copie des pixels du premier plan (éventuellement inversés) dans la destination.

• Modes *ForeOr* ($0005) et *NotForeOr* ($8005) : effectue un « ou » logique entre les pixels du premier plan (éventuellement inversés) et la destination.

• Modes *ForeXOr* ($0006) et *NotForeXOr)* ($8006) : effectue un « ou » exclusif entre les pixels du premier plan (éventuellement inversés) et la destination.

• Modes *ForeBic* ($0007) et *NotForeBic* ($8007) : effectue un « et » logique entre les pixels du premier plan (inversés ou non) et la destination.

Dans tout ce qui précède, le terme « pixels du premier plan » désigne les pixels constituant le corps du caractère, dans la couleur de premier plan *(ForeColor)* affectée au grafport (voir plus bas, le paragraphe consacré aux caractéristiques du texte).

L'environnement de développement proposera vraisemblablement les définitions suivantes, pour éviter l'emploi de constantes numériques :

```
#define Copy          0x0000
#define NotCopy       0x8000
#define Or            0x0001
#define NotOr         0x8001
#define XOr           0x0002
#define NotXOr        0x8002
#define Bic           0x0003
#define NotBic        0x8003
#define ForeCopy      0x0004
#define NotForeCopy   0x8004
#define ForeOr        0x0005
#define NotForeOr     0x8005
#define ForeXOr       0x0006
#define NotForeXOr    0x8006
#define ForeBic       0x0007
#define NotForeBic    0x8007
```

CARACTÉRISTIQUES DU GRAFPORT

Créer un grafport

• On peut ouvrir plusieurs ports simultanément, sinon il ne serait pas possible d'avoir plusieurs fenêtres à l'écran en même temps. Seul l'un des ports est actif, il est désigné sous le nom de port courant. La plupart des routines QuickDraw affectent le port courant. Pour rendre un port courant, on utilise la procédure **SetPort,** dont le seul argument est un pointeur sur le port à activer. Pour connaître le port courant, la fonction **GetPort** (sans argument) retourne un pointeur sur ce port.

• Dans le chapitre consacré au Window Manager, nous verrons comment créer une fenêtre. C'est généralement à la création d'une fenêtre qu'est créé le grafport associé, et c'est le Window Manager qui appelle la procédure **OpenPort.** Retenons simplement en première approche qu'un grafport est repéré par un pointeur, et que ce pointeur nous est retourné par la fonction **NewWindow** du Window Manager.

Dans certains cas particuliers, on peut vouloir créer directement un grafport, sans le concours du Window Manager. Cas typique, une application peut vouloir dessiner

dans la même pixel map avec deux jeux d'outils différents : on crée alors un deuxième grafport référençant la même structure *LocInfo,* et on change ses autres caractéristiques par les procédures que nous verrons plus loin. L'intérêt ? Au lieu d'appeler à tout bout de champ ces procédures de modification d'outils graphiques pour changer puis rétablir une valeur, on appelle **SetPort** en travaillant alternativement avec l'un ou l'autre des grafports sur la même image !

La procédure **OpenPort** réclame un seul argument : l'adresse de la structure grafport qui sera utilisée pour stocker les caractéristiques du port. On vérifiera que la structure *GrafPort* est bien définie dans l'environnement de travail utilisé, sinon on réservera la place pour un objet de 170 octets.

```
GrafPort    port;                   /* un grafport */
GrafPort    *gp;                    /* un pointeur sur grafport */

gp = GetPort( );                    /* pointeur sur le grafport courant */
OpenPort(&port);                    /* création du grafport */
port = *gp;                         /* recopie du grafport courant dans le nouveau */
...                                 /* modification des caractéristiques du nouveau grafport */
SetPort(gp);                        /* on active l'ancien grafport */
...                                 /* on dessine avec ses caractéristiques (crayon, texte, etc) */
SetPort(&port);                     /* on active le nouveau grafport */
...                                 /* on dessine avec ses caractéristiques (crayon, texte, etc) */
```

Dans l'exemple, en déclarant *port* de type *GrafPort,* on réserve la place nécessaire pour y stocker ses caractéristiques, la place mémoire nécessaire au grafport pointé par *gp* ayant été réservée par la fonction qui l'a créée (éventuellement par le Window Manager). Ensuite, la procédure **OpenPort** est appelée. Elle initialise le nouveau port avec les valeurs standard et alloue l'espace nécessaire à la manipulation de la clip region et de la région visible. Ce port devient le port courant. Il sera utilisable jusqu'à l'appel de la procédure **ClosePort**, qui désallouera l'espace occupé et écartera les handles sur les régions associées. L'instruction suivante est très puissante, mais tous les compilateurs C ne l'acceptent pas forcément. Il s'agit d'une assignation de structure. A droite, **gp* représente une variable de type *GrafPort.* A gauche, *port* également. Le signe = (assignation) fait que la variable de gauche va prendre la valeur de la variable de droite, il s'agit donc bien là d'une recopie du contenu de tous les champs constituant la structure. Les environnements Megamax (dont l'APW-C) acceptent une telle instruction C. La recopie fait notamment que les deux grafports ont alors la même *LocInfo* associée, donc que les outils de dessin vont bien toucher la même image !

Remarque Un grafport existant peut être réinitialisé grâce à la procédure **InitPort. OpenPort** appelle d'ailleurs **InitPort** avant de créer la clip region et la région visible. Les deux procédures ont la même syntaxe.

• Quand c'est le Window Manager qui ouvre un nouveau port, la structure *LocInfo* associée désigne l'écran tout entier, et on n'a toujours pas de question à se poser. Quand par contre on veut dessiner en dehors de l'écran, il faut définir une pixel map distincte de la mémoire écran et ses données additionnelles. Pour inclure ces nouvelles données dans la définition du grafport courant, on utilisera la procédure **SetPortLoc**, dont le seul argument est un pointeur sur la nouvelle structure *LocInfo.*

```
LocInfo     infos;                  /* une structure LocInfo */
long        taille;                 /* taille en octets de la pixel map */

taille = 1280L;                     /* 40 lignes de 32 octets */
infos.PortSCB = 0;                  /* SCB standard en mode 320 */
infos.baseAddr = *NewHandle(taille, myID, 0xC000, 0L); /* allocation mémoire, bloc fixe */
infos.rowBytes = 32;                /* nombre d'octets par ligne */
SetRect(&infos.BoundsRect, 0, 0, 50, 40); /* 50 pixels de large sur 40 de haut */
SetPortLoc(&infos);                 /* mise en place de la structure */
```


Le fait de s'allouer de la mémoire par la fonction **NewHandle** nous assure que nous ne travaillerons pas dans la mémoire écran, donc que nous dessinerons bien hors écran !

Notons quelques chausse-trapes dans lesquelles il ne faudra pas tomber. Le but final est d'obtenir une image de 50 pixels de large sur 40 pixels de haut, ainsi qu'en témoigne le champ *BoundsRect* de la structure. Puisque nous sommes en mode 320, il faut 25 octets pour coder les 50 pixels. *rowBytes* sera donc le plus petit multiple de 8 supérieur ou égal à 25, soit 32, et la pixel map s'étendra sur 32 × 40 octets. Pour éviter de se tromper, rien ne vaut une bonne formule de calcul. Si r représente le rectangle frontière de la *LocInfo,* on a :

```
Rect    r;                          /* un rectangle */

infos.rowBytes = (r.right <= r.left) ? 0 : ((r.right - r.left - 1) / 16 ) + 1) * 8;
                    /* valable en mode 320 uniquement, remplacer 16 par 32 en mode 640) */
taille = (long) (r.bottom - r.top) * infos.rowBytes;    /*mémoire à allouer */
```

Ajoutons que la procédure **GetPortLoc** permet de connaître (par son adresse) la structure *LocInfo* du port courant, ce qui est pratique pour jongler par exemple de la mémoire écran à un dessin hors écran :

```
LocInfo     oldloc;                 /* une structure LocInfo */

GetPortLoc(&oldloc);                /* on récupère la structure courante */
...                                 /* on prépare les caractéristiques de la nouvelle structure */
SetPortLoc(&infos);                 /* mise en place de la nouvelle structure */
...                                 /* on dessine avec les caractéristiques du grafport courant hors écran */
SetPortLoc(&oldloc);                /* rétablissement de l'ancienne structure */
...                                 /* on dessine de nouveau dans la mémoire écran */
```

Remarquons qu'avec cet exemple, les caractéristiques d'un grafport servent pour dessiner à deux endroits différents, ce qui est exactement le contraire de l'exemple du point précédent, où on utilisait les caractéristiques de deux grafports distincts pour dessiner à un endroit unique. QuickDraw autorise sans sourciller ce genre de libertés.

• Terminons par le rectangle *PortRect,* qui délimite la surface dans laquelle on pourra dessiner. Quand le Window Manager crée une fenêtre, ce rectangle est l'un des paramètres à renseigner. Dans le cas où nous voudrions dessiner en dehors de l'écran (suite de l'exemple précédent), la procédure **SetPortRect** nous permet de fixer ce rectangle :

```
Rect    r;                          /* un rectangle */

SetRect(&r, 0, 0, 50, 40);          /* un rectangle est défini */
SetPortRect(&r);                    /* PortRect est fixé */
```

Notre *PortRect* coïncide avec le rectangle frontière de la pixel map. C'est souvent le cas quand on dessine en dehors de l'écran, puisque le défilement permis par les fenêtres n'a aucune raison d'être.

Notons que **GetPortRect** est une procédure qui permet de connaître le *PortRect* du grafport actif. On passe en argument l'adresse du rectangle qui recevra le *PortRect.*

Nous laisserons de côté l'étude de routines telles que **SetPortSize** (sert à modifier la taille du port courant) et **MovePortTo** (sert à déplacer le port courant), qui sont principalement appelées de manière interne par le Window Manager quand une fenêtre doit être redimensionnée ou déplacée.

Deux régions associées au grafport

• La clip region peut être modifiée au gré de l'application, grâce à deux procédures, **SetClip** et **ClipRect**. La première admet comme unique argument un handle sur une région qui va devenir la nouvelle clip region. La seconde permet de donner à la nouvelle clip region la forme d'un rectangle (dont un pointeur est passé en argument).

Pour connaître la clip region courante, une procédure, **GetClip**, dont l'unique argument est un handle qui pointe de manière indirecte sur une région qui va recevoir la clip region.

Lors de ces trois appels, il y a copie de région. Le grafport conserve dans la structure qui lui est associée un handle sur la clip region. **SetClip** ne modifie pas ce handle, mais copie la définition de la région associée au handle passé en argument dans celle du handle associé au grafport. De même, **GetClip** ne récupère pas un handle sur la clip region, mais une copie de la clip region dans l'espace mémoire désigné par le handle passé en argument.

Pour changer le handle sur la clip region associée à un grafport, on utilisera la fonction **SetClipHandle**. Pour connaître le handle sur la clip region associée à un grafport, on utilisera la fonction **GetClipHandle**. Les appels auront cette forme :

```
Handle oldclip, newclip;                    /* deux handles sur région */

Oldclip = GetClipthandle ();                /* récupère le handle associé au grafport */
SetClipHandle(newclip);                     /* change le handle associé au grafport */
```

Rien ne vaut un exemple pour mettre les choses au clair. Supposons que nous voulions dessiner le fameux demi-cercle, mais que nous ne sachions pas quelle est l'actuelle clip region. Nous allons la récupérer et la garder au chaud, la réduire à notre demi-carré pour exécuter le dessin (par intersection, tant pis si tout n'est pas dessiné), puis la rétablir.

```
Handle oldclip, newclip, demicadre;         /* trois handles sur région */
Rect   cadre;                               /* un rectangle */

oldclip = NewRgn( );                        /* on récupère un handle sur région */
newclip = NewRgn( );                        /* et un deuxième */
demicadre = NewRgn( );                      /* et un troisième */
GetClip(oldclip);                  /* on récupère la clip region actuelle (par copie)*/
SetRectRgn(demicadre, 50, 50, 100, 150); /* ce sera notre demi-carré */
SectRgn(oldclip, demicadre, newclip);       /* intersection des deux régions */
SetClip(newclip);                           /* on fixe la nouvelle clip region (par copie) */
SetRect(&cadre, 50, 50, 150, 150);          /* ce rectangle est un carré */
PaintOval(&cadre); /* on dessine le cercle entier, seule une moitié est prise en compte */
SetClip(oldclip);                           /* on rétablit l'ancienne clip region (par copie) */
DisposeRgn(oldclip);                        /* on fait le ménage */
DisposeRgn(newclip);                        /* on fait le ménage */
DisposeRgn(demicadre);                      /* on fait le ménage */
```

On remarquera que les trois handles manipulés dans cet exemple (on aurait pu faire l'économie de l'un d'entre eux) ont été obtenus par la fonction **NewRgn**. Il faudra en prendre l'habitude : aucune des routines de manipulation des régions n'a été conçue pour s'allouer elle-même l'espace nécessaire.

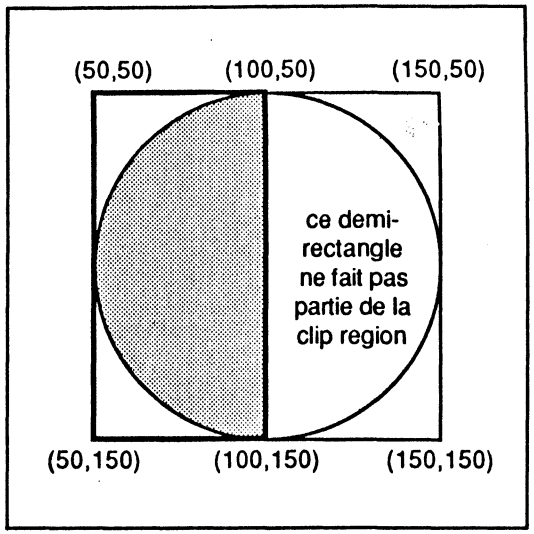


Figure III.11. Dessin d'un demi-cercle.

En faisant l'impasse sur la forme et la localisation de la clip region en cours, on pouvait écrire la même chose un peu plus simplement :

```
Handle oldclip;                             /* handle sur région */
Rect   cadre, demicadre;                    /* deux rectangles */

oldclip = NewRgn( );                        /* on récupère un handle sur région */
GetClip(oldclip);                           /* on récupère la clip region actuelle */
SetRect(&demicadre, 50, 50, 100, 150);      /* ce sera notre demi-carré */
ClipRect(&demicadre);                       /* on fixe la nouvelle clip region */
SetRect(&cadre, 50, 50, 150, 150);          /* ce rectangle est un carré */
PaintOval(&cadre); /* on dessine le cercle entier, seule une moitié est prise en compte */
SetClip(oldclip);                           /* on rétablit l'ancienne clip region */
DisposeRgn(oldclip);                        /* on fait le ménage */
```

• La région visible possède deux outils identiques, **SetVisRgn** et **GetVisRgn**, qui permettent de la fixer et de la connaître. Ces fonctions sont d'utilisation absolument identique à **SetClip** et **GetClip**, nous ne nous y attarderons donc pas.

Notons toutefois que c'est le Window Manager qui utilise le plus ces procédures, notamment au moment de mettre à jour une fenêtre : la procédure **BeginUpdate** modifie l'actuelle région visible de la fenêtre avant mise à jour et la procédure **EndUpdate** la rétablit après.

Idem pour les routines **SetVisHandle** et **GetVisHandle**, qui permettent de modifier ou connaître le handle sur la région visible associée au grafport.

Pattern d'arrière-plan

Quand un port est créé, il est uniformément rempli par un pattern (le défaut pattern est le pattern solide de couleur blanche). On vient dessiner par-dessus ce pattern. Pour effacer un dessin, il suffira de remplir la surface correspondante avec ce pattern d'arrière-plan.

Les routines **SetBackPat** et **GetBackPat** permettent de fixer ou de connaître le pattern d'arrière-plan. La procédure **SetSolidBackPat** permet de fixer un pattern solide (c'est-à-dire une véritable couleur, tous les pixels de couleur identique).

Le fonctionnement de ces routines est identique à celui des routines gérant le pattern du crayon, étudiées dans le paragraphe suivant. Aucune de ces routines n'a d'effet immédiat à l'écran.

Caractéristiques du crayon

A chaque port est associé un crayon, dont on peut modifier les caractéristiques. La procédure **PenNormal** (sans argument) rétablit les caractéristiques par défaut du crayon du port courant, telles qu'elles sont utilisées au moment de son initialisation (la localisation n'est pas affectée).

• Localisation du crayon : c'est le point (en coordonnées locales) où se trouve présentement le crayon. Si une procédure de dessin de ligne ou d'écriture de texte est invoquée, elle utilisera cette localisation comme point d'origine. Les procédures **Move** et **MoveTo** permettent de changer la localisation du crayon.

```
MoveTo(50, 30);          /* positionne le crayon sur le point (50,30) */
Move(10, 20);  /* déplace le crayon de 10 points horizontalement et 20 verticalement */
```

Après ces deux instructions, le crayon est positionné sur le point (60,50). Évidemment, des valeurs positives dans la procédure **Move** indiquent un déplacement vers la droite ou vers le bas, et des valeurs négatives un déplacement vers la gauche ou vers le haut.

Pour connaître la localisation courante du crayon, on utilisera la procédure **GetPen** : on passe en argument l'adresse d'un point (voir plus bas la définition de la structure *Point*) qui recevra le résultat :

```
Point   loc;                    /* loc a une structure de point */
int     x, y;

GetPen(&loc);                   /* le point courant est renseigné dans loc */
x = loc.H;                      /* abscisse du point */
y = loc.V;                      /* ordonnée du point */
```

• Taille du crayon : le trait qu'est capable de tracer un crayon n'est pas limité à un pixel d'épaisseur, mais peut être quelconque. Il suffit de définir ce que nous pouvons appeler une « unité de traçage », largeur et hauteur de chaque élément du trait. Le « point chaud » de cette unité de traçage en sera le coin supérieur gauche (quand on tracera une ligne, les coordonnées s'appliqueront à ce point ; pour des objets plus complexes, ce ne sera pas toujours vrai, comme nous le verrons plus loin). Les routines **SetPenSize** et **GetPenSize** permettent de fixer ou de connaître la taille du crayon. Par défaut, la taille est 1 × 1.

```
Point   pt;

GetPenSize(&pt);                /* on récupère la taille de l'unité de traçage */
SetPenSize(3,4);                /* l'unité de traçage sera  large de 3 pixels, haute de 4 */
...                             /* on dessine avec cette unité */
SetPenSize(pt.H, pt.V);         /* on rétablit l'ancienne unité */
```

• Mode du crayon : c'est le mode de transfert tel que nous l'avons évoqué plus haut. Les routines **SetPenMode** et **GetPenMode** permettent de fixer ou de connaître le mode de transfert du crayon. Le mode par défaut est *Copy*.


```
int  oldmode;

oldmode = GetPenMode( );        /* sauvegarde du mode courant */
SetPenMode(NotOr);              /* utilisation du mode NotOr */
...                             /* dessins avec le nouveau mode */
SetPenMode(oldmode);            /* on rétablit l'ancien mode */
```

Dans cet exemple, on commence par mettre en mémoire le mode courant, puis on fixe un nouveau mode, on dessine en l'utilisant, et enfin on rétablit l'ancien mode.

• Pattern du crayon : puisque l'unité de traçage n'est pas limitée à un pixel, on peut définir pour le crayon un pattern avec lequel il tracera ses traits. Ce pattern de remplissage de traits fonctionne de manière identique au pattern de remplissage de formes, vu plus haut. Les routines **SetPenPat** et **GetPenPat** permettent de fixer ou de connaître le pattern du crayon (véritable pattern, pixel par pixel). La procédure **SetSolidPenPat** permet de fixer un pattern solide au crayon (c'est-à-dire une véritable couleur, tous les pixels de couleur identique). Le pattern par défaut est le noir, couleur solide.

Un pattern est un objet assez délicat à définir : puisqu'il représente huit fois huit pixels, il sera de taille 256 bits en mode 320 (où un pixel est codé sur 4 bits) et de taille 128 bits en mode 640 (où un pixel vaut 2 bits). On accédera toujours à un pattern par l'intermédiaire d'un pointeur. Pour définir le pattern, il faudra jouer sur les bits !

Création et utilisation d'un pattern en mode 320 :

```
int     i;
Pointer oldpat;
char    newpat[32];             /* chaque caractère représente 2 pixels */

oldpat = GetPenPat( );          /* on récupère l'adresse du pattern courant */
for (i=0; i<8; ++i)
    {
    newpat[4*i] = 0x33;     /* couleur 3 pour les deux premiers pixels de chaque ligne */
    newpat[4*i+1] = 0x35;   /* couleur 3 pour le 3ème pixel, couleur 5 pour le 4ème */
    newpat[4*i+2] = 0x53;   /* couleur 5 pour le 5ème pixel, couleur 3 pour le 6ème */
    newpat[4*i+3] = 0x33;   /* couleur 3 pour les deux derniers pixels de chaque ligne */
    }
SetPenPat(newpat);              /* on fixe un nouveau pattern */
...                             /* dessins avec le nouveau pattern */
SetPenPat(oldpat);              /* on rétablit l'ancien pattern */
```

Le pattern créé est constitué de deux bandes verticales, l'une très large de la couleur correspondant à l'entrée n° 3 de la table de couleurs active, l'autre très mince ayant la couleur n° 5. Après avoir sauvegardé le pattern courant, on fixe ce nouveau pattern, on dessine avec, et enfin on rétablit l'ancien.

Ainsi qu'il a été dit dans l'introduction, il ne sera généralement pas équivalent de déclarer un pattern char[32], int[16] ou long[8] (mode 320), à cause de la position inversée des octets haut et bas en mémoire. Nous verrons dans d'autres chapitres (VI et XI, notamment) des exemples de définition de patterns en mode 640.

Création et utilisation d'un pattern solide :

```
Pointer oldpat;                 /* un pointeur */

oldpat = GetPenPat( );          /* on récupère l'adresse du pattern courant */
SetSolidPenPat(8);              /* utilisation de la couleur n°8 de la table */
...                             /* dessins avec le nouveau pattern */
SetPenPat(oldpat);              /* on rétablit l'ancien pattern */
```

On constate que l'utilisation d'un pattern solide, c'est-à-dire une couleur uniforme, est nettement plus commode ! Il suffit de préciser le numéro de la couleur dans la table active, et QuickDraw se débrouille, quel que soit le mode d'affichage utilisé. Peu importe si l'ancien pattern était solide ou pas, on le mémorise comme un pattern quelconque, par son adresse.

Notons une procédure intéressante, qui permet de faire d'un pattern un pattern solide, **SolidPattern,** ce qui offre une alternative à l'utilisation de **SetSolidBackPat** ou **SetSolidPenPat**. L'exemple suivant est équivalent au précédent, sauf que le pattern solide défini peut être utilisé ailleurs.

```
char    colPat[32];                 /* réserve de la place pour le nouveau pattern */
Pointer oldpat;

SolidPattern(colPat, 8);/* le nouveau pattern est solide (couleur 8 de la palette active) */
oldpat = GetPenPat( );              /* on récupère l'adresse du pattern courant */
SetPenPat(colPat);                  /* on fixe un nouveau pattern (solide) */
...                                 /* dessins avec le nouveau pattern */
SetPenPat(oldpat);                  /* on rétablit l'ancien pattern */
```

• Masque du crayon : associé au pattern, a les fonctions vues plus haut. Les routines **SetPenMask** et **GetPenMask** permettent de fixer ou de connaître le masque du crayon. Par défaut, tous les bits du masque sont à un. Un masque étant une collection de huit fois huit bis, on peut le mémoriser sous forme de char[8] et y accéder par l'intermédiaire d'un pointeur :

```
Pointer oldmask;
char    newmask[8];

oldmask = GetPenMask( );                    /* on récupère l'adresse du masque courant */
newmask[0] = newmask[2] = newmask[4] = newmask[6] = 0xAA;
newmask[1] = newmask[3] = newmask[5] = newmask[7] = 0x55;
SetPenMask(newmask);                        /* on fixe un nouveau masque */
...                                         /* dessins avec le nouveau masque */
SetPenMask(oldmask);                        /* on rétablit l'ancien masque */
```

Dans cet exemple, l'adresse de l'ancien masque (qu'il ait été fixé ailleurs par l'application, ou que ce soit le masque par défaut utilisé par QuickDraw) est sauvegardée, puis un nouveau masque est fixé pour exécuter certains dessins, et enfin le masque précédent est rétabli. Les valeurs $AA et $55 utilisées ici sont remarquables, puisqu'elles s'écrivent en binaire 1010 1010 et 0101 0101 respectivement. On crée donc un masque qui ne laisse passer qu'un pixel sur deux, en quinconce.

• Toutes les caractéristiques précédentes du crayon (localisation, taille, mode, pattern et masque) sont résumées dans une structure appelée *PenState* dont nous ne donnerons pas la définition en C car elle fait intervenir la notion de pattern (dont la taille dépend du mode utilisé). Le plus simple est d'utiliser un bloc de 50 octets (4 pour la localisation, 4 pour la taille, 2 pour le mode, 32 pour le pattern et 8 pour le masque). Par l'intermédiaire d'un pointeur sur un tel bloc, la procédure **GetPenState** permettra de mémoriser toutes les caractéristiques du crayon en cours d'utilisation, et la procédure **SetPenState** de les rétablir.

```
char    state[50];                  /* réserve 50 octets */

GetPenState(state);                 /* on mémorise les caractéristiques du crayon */
...                                 /* modifications diverses, utilisation d'un nouveau crayon */
SetPenState(state);                 /* on rétablit l'ancien crayon */
```

• Niveau d'invisibilité du crayon : le crayon peut être visible ou invisible, et deux procédures gèrent le niveau d'invisibilité du crayon. **HidePen** le décrémente tandis que **ShowPen** l'incrémente. Ce niveau est initialement à zéro, signifiant que le crayon est

visible, donc que ce qu'il dessine apparaît à l'écran. Quand ce niveau devient négatif, le crayon devient invisible : les dessins sont exécutés, mais n'apparaissent plus. L'intérêt de niveaux multiples dans l'invisibilité est évident : supposons qu'un sous-programme veuille faire quelque chose de non visible (voir par exemple la définition d'une région, plus bas). Il commence par appeler **HidePen**, fait ce qu'il a à faire, et termine par **ShowPen**. Globalement, il a rétabli le niveau d'invisibilité du départ. Si avant l'appel du sous-programme, le crayon était déjà invisible, il l'est toujours après, malgré l'appel de **ShowPen**. Ces deux procédures devraient toujours être appelées conjointement, comme une parenthèse ouvrante et une parenthèse fermante.

Caractéristiques du texte

Tout ce que nous allons dire dans ce paragraphe ne présentera plus énormément d'intérêt quand le Font Manager sera vraiment opérationnel. Il suffit de comparer une partie des appels que nous allons voir, et la seule procédure **InstallFont** que nous verrons en fin de chapitre X pour en convenir.

• Tout texte sera écrit en utilisant une police de caractères déterminée. Les routines **SetFont** et **GetFont** permettent de fixer ou de connaître la police de caractères utilisée (par l'intermédiaire d'un handle). Idem sur la police système avec **SetSysFont** et **GetSysFont.**

• Les polices de caractères peuvent être de largeur fixe ou proportionnelle. Largeur fixe signifie que tous les caractères ont la même largeur (la lettre m occupera la même place que la lettre i dans un texte, mais il y aura beaucoup plus de vide autour du i qu'autour du m). Largeur proportionnelle signifie au contraire que chaque caractère a sa propre largeur, rendant la lecture plus harmonieuse.

Dans les 16 bits du champ *FontFlags* qui donne certaines précisions sur les opérations affectant le texte dessiné dans le grafport, le bit 0 précise si la police de caractères doit être considérée comme fixe (1) ou proportionnelle (0). Les routines **SetFontFlags** et **GetFontFlags** permettent de fixer ou de connaître le champ *FontFlags* lié au port courant. On peut notamment rendre fixe à l'affichage une police proportionnelle (la largeur du plus grand des caractères fixant la largeur de tous les caractères).

Dans les deux cas, le dessin d'un caractère est défini par un rectangle dont les pixels sont noirs ou blancs, le noir représentant le corps du caractère et le blanc le vide autour. La hauteur du rectangle est caractéristique de la taille de la police de caractères, la largeur est variable (polices proportionnelles) ou fixe (polices fixes).

• Quand QuickDraw dessine un caractère, il dessine en fait un petit rectangle. Si le mode de transfert du texte est un mode normal, la couleur *ForeColor* du grafport est utilisée pour le corps du texte, et la couleur *BackColor* est utilisée pour remplir les vides. On peut par exemple obtenir un rectangle représentant une lettre en bleu sur fond jaune. C'est à ce rectangle que va s'appliquer le mode de transfert, pour recouvrir ce qu'il y a déjà sur l'écran.

Si le mode de transfert du texte est un mode spécial texte, la couleur *ForeColor* est utilisée de manière identique, mais le fond reste vide, et le mode de transfert ne s'applique qu'au corps du texte.

Les procédures **SetForeColor** et **SetBackColor** permettent de fixer les couleurs affectant le texte (corps et fond), les fonctions **GetForeColor** et **GetBackColor** permettent de connaître ces couleurs. Les routines **SetTextMode** et **GetTextMode** permettent de fixer ou de connaître le mode de transfert appliqué aux textes.

Exemples d'utilisation (assume que la palette standard en mode 320 est utilisée) :

```
int forecol, backcol, txtmode;

forecol = GetForeColor( );          /* mémorise la couleur du corps du texte */
backcol = GetBackColor( );          /* idem couleur fond de texte */
txtmode = GetTextMode( );           /* idem mode de transfert du texte */
SetForeColor(9);                    /* corps du texte en jaune (entrée n°9) */
SetBackColor(4);                    /* fond bleu (entrée n°4) */
SetTextMode(Copy);                  /* mode Copy */
...          /* on écrit avec ces caractéristiques, on peut ensuite rétablir les précédentes */
```

Avec les trois instructions précédentes, tout texte ultérieur sera dessiné en jaune sur fond bleu, écrasant tout ce qui pourrait se trouver déjà sur l'écran (plus exactement dans la pixel map).

• On peut donner un style à un texte, par déformation calculée du dessin des caractères. Cinq styles sont prédéfinis, ils sont sélectionnés en mettant à un le bit correspondant du champ *TxFace*.

bit 0 : **gras**
bit 1 : *italique*
bit 2 : souligné
bit 3 : **relief**
bit 4 : **ombré**

Par combinaison, on obtiendra des styles composés. Par exemple, la valeur 5 signifiera gras souligné, puisque 5 s'écrit en binaire 0000 0000 0000 0101. La valeur 0 signifie absence de style, il s'agit donc de texte normal (*plain text*, en anglais). Les routines **SetTextFace** et **GetTextFace** permettent de fixer ou de connaître le style employé pour dessiner du texte.

```
int face;

face = GetTextFace( );                  /* quel est le style courant? */
if (face & 4) SetTextFace(face-4);      /* on en supprime le souligné */
```

Dans l'exemple précédent, si le style courant contient l'indicateur « souligné » positionné (c'est-à-dire le bit 2 à un, c'est pourquoi on compare avec la valeur 4), on le supprime, sans affecter les autres bits. On passe ainsi du gras italique souligné au gras italique, ou du souligné au style standard. On pourra vérifier que l'écriture suivante est équivalente (puisque FFFB en hexa s'écrit 1111 1111 1111 1011 en binaire, on force le bit 2 à zéro) :

```
SetTextFace(GetTextFace( ) & 0xFFFB); /* on supprime le souligné du style courant */
```

Note Seuls les styles gras et souligné sont définis en mémoire morte. La version 1.02 de QuickDraw ne connaît pas encore les autres styles.

• On peut changer la taille des caractères dessinés grâce à la procédure **SetTextSize**, ou connaître la taille actuelle avec la fonction **GetTextSize**. La taille est exprimée en nombre de points, mais il ne faut pas toujours accorder une signification exacte à ce nombre : en fonction de la définition d'un jeu de caractères, une taille 14 peut paraître plus petite qu'une taille 12.

```
int taille;

taille = GetTextSize( );            /* on mémorise la taille en cours */
SetTextSize(12);                    /* on fixe la taille 12 */
...                                 /* on dessine du texte dans cette taille */
SetTextSize(taille);                /* on rétablit la taille précédente */
```


• Un des champs du grafport s'appelle *FontID*. C'est un entier long, dont le mot haut contient l'identifiant d'une famille de caractères, et le mot bas la taille et le style de la police désirée. Le fait de modifier ce champ n'affecte en rien le dessin du texte par QuickDraw, mais les programmes (et notamment les programmes utilisant l'impression laser) viendront chercher dans ce champ les caractéristiques désirées par l'application ou par l'utilisateur. On passera par l'intermédiaire du Font Manager, plutôt que par la procédure **SetFontID** pour modifier ce champ. La fonction **GetFontID** (sans argument) en retourne le contenu (entier long).

Champ utilisateur

Pas grand chose à dire à son propos. Dans sa grande générosité, QuickDraw nous offre quatre octets de stockage associés à chaque grafport, que l'application peut utiliser comme elle l'entend. On pourra par exemple y stocker l'adresse d'une procédure de dessin, ou n'importe quoi, ou rien du tout. La procédure **SetUserField** permet de stocker dans le champ la valeur passée en argument (entier long), la fonction **GetUserField** retourne (dans un entier long) le contenu du champ.

```
long    val;

val = GetUserField( );              /* on récupère la valeur */
SetUserField(val + 1);              /* on la change */
```

STRUCTURE ET MANIPULATION

Attention Avant d'utiliser les routines présentées dans cette section, il faut avoir initialisé QuickDraw avec la procédure **QDStartUp.**

Point

La manipulation d'un point peut se faire de plusieurs manières, ce qui peut provoquer une certaine confusion. Une structure de type *Point* est définie, et un point est soit repéré par un pointeur sur ce type de structure, soit utilise directement ses coordonnées (abscisse et ordonnée dans un système de coordonnées donné). On pourra également voir un point comme un entier long, abscisse dans le mot haut et ordonnée dans le mot bas (nous ne nous en priverons pas dans les chapitres suivants).

La structure de type Point peut être définie de la manière suivante :

```
struct _Point {
    int  V;                         /* coordonnée verticale (ordonnée) */
    int  H;                         /* coordonnée horizontale (abscisse) */
    };
#define Point struct _Point;
```

La procédure **SetPt** permet de remplir une structure de type point à partir de ses coordonnées.

Exemple

```
Point   Pt;                         /* un point */

SetPt(&Pt, 50, 100);                /* Pt représente le point (50,100) */
    /* équivaut à:
    Pt.H = 50;
    Pt.V = 100;
    */
```

On notera l'ordre différent entre les arguments de **SetPt** et les éléments constitutifs de la structure *Point*. La déclaration suivante est absolument équivalente à la précédente :

```
long    pt;                     /* un entier long */

SetPt(&pt, 50, 100);            /* pt représente le point (50,100) */
```

Calculs sur les points

• QuickDraw met à notre disposition plusieurs routines pour faire des calculs sur les points. **AddPt** permet d'obtenir la somme des coordonnées de deux points (addition vectorielle) et **SubPt** leur différence. On désigne les points par l'intermédiaire de pointeurs, le premier opérande reste inchangé tandis que le second reçoit le résultat. Dans le cas de la soustraction, on fait second opérande moins premier opérande.

```
Point   pt1, pt2;               /* deux points */

SetPt(&pt1, 50, 100);           /* on fixe le premier point */
SetPt(&pt2, 20, 30);            /* on fixe le second point */
AddPt(&pt1, &pt2);              /* le point pt1 reste inchangé, c'est (50,100) */
                                /* le point pt2 devient (70,130) */
SubPt(&pt2, &pt1);              /* le point pt2 reste inchangé, c'est (70,130) */
                                /* le point pt1 devient (-20,-30) */
```

• La fonction **EqualPt** nous permet de savoir si deux points ont mêmes coordonnées ou pas. Elle retourne la valeur TRUE en cas d'égalité, FALSE si les points sont distincts. La fonction n'affecte pas la valeur des points :

```
int     result;
Point   pt1, pt2;               /* deux points */

result = EqualPt(&pt1, &pt2); /* result est non nul si pt1 et pt2 représentent le même point */
```

• QuickDraw nous offre deux procédures pour convertir les composantes d'un point du système de coordonnées locales (défini par le *PortRect* du grafport courant) à celui de coordonnées globales (défini par le coin supérieur gauche du *BoundsRect* de la pixel map où on dessine) et réciproquement. Les deux procédures affectent le point dont un pointeur est passé en argument.

```
Point   pt;                     /* un point */

LocalToGlobal(&pt);             /* pt est traduit en coordonnées globales */
GlobalToLocal(&pt);             /* pt est traduit en coordonnées locales */
```

Après ces deux appels, le point a retrouvé ses composantes d'origine : les deux procédures sont réciproques l'une de l'autre. Pour passer des coordonnées locales d'un grafport à celles d'un autre, on changera de port courant entre ces deux appels (grâce à la procédure **SetPort**).

Rectangle

Le rectangle est une structure mathématique primordiale dans les concepts QuickDraw. On se référera toujours à un rectangle par l'intermédiaire d'un pointeur.

La structure de type Rect peut être définie de la manière suivante :

```
struct _Rect {
    int  top;       /* coordonnée verticale du coin supérieur gauche */
    int  left;      /* coordonnée horizontale du coin supérieur gauche */
    int  bottom;    /* coordonnée verticale du coin supérieur droit */
    int  right;     /* coordonnée horizontale du coin supérieur droit */
    };
#define Rect struct _Rect;
```

Pour définir un rectangle, QuickDraw met à notre disposition deux procédures, **SetRect** à partir de ses quatre coordonnées, et **Pt2Rect** à partir de deux points : le coin supérieur gauche et le point inférieur droit. Cette limitation est déplorable : sur Macintosh, cette procédure accepte deux points diagonalement opposés, sans restriction sur la position relative de chaque coordonnée.

```
Rect    r1, r2;                 /* deux rectangles */
Point   pt1, pt2;               /* deux points */

SetRect(&r1, 40, 30, 150, 180); /* on fixe le rectangle */
    /* équivaut à:
    r1.left = 40;
    r1.top = 30;
    r1.right = 150;
    r1.bottom = 180;
    */

SetPt(&pt1, 40, 30);            /* coin supérieur gauche du rectangle */
SetPt(&pt2, 150, 180);          /* coin inférieur droit du rectangle */
Pt2Rect(&pt1, &pt2, &r2);       /* on fixe le rectangle */
```

Après ces instructions, les deux rectangles définis sont identiques. On notera l'ordre différent entre les composantes de la structure *Rect* et les arguments de la procédure **SetRect.**

Avec **SetRect**, le rectangle résultant sera vide si *right* ≤ *left* ou si *bottom* ≤ *top*. Avec **Pt2Rect**, il sera vide si le deuxième point n'est pas plus bas et plus à droite que le premier.

Calculs sur un rectangle

• La procédure **OffsetRect** permet de déplacer un rectangle sans affecter sa taille. Le déplacement horizontal s'effectue de dH pixels (vers la droite si dH est positif, vers la gauche sinon), le déplacement vertical de dV pixels (vers le bas si dV est positif, vers le haut sinon).

```
Rect    *rect;                  /* pointeur sur rectangle */
int     dH, dV;

dH = 5;                         /* déplacement horizontal */
dV = 10;                        /* déplacement vertical */
OffsetRect(rect, dH, dV);       /* les coordonnées du rectangle sont modifiées */
```

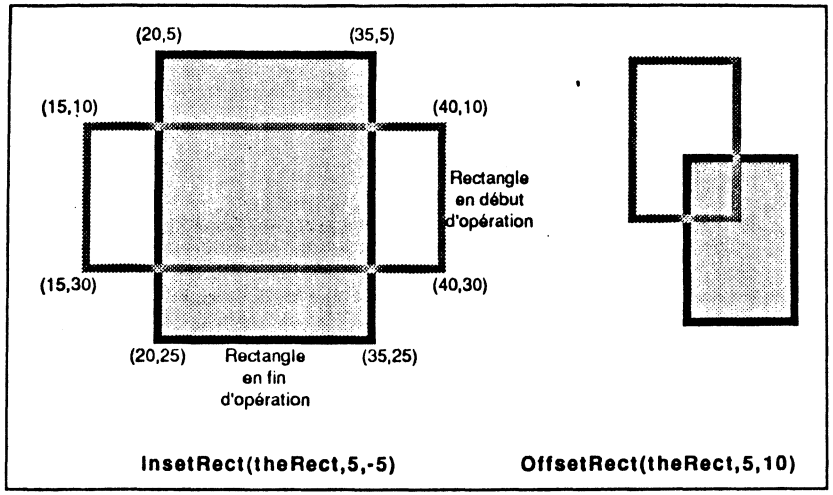


Figure III.12. Déformation et déplacement d'un rectangle.

• La procédure **InsetRect** agrandit ou rétrécit un rectangle sans déplacer son centre. Si dH est positif, les deux bords verticaux se rapprochent, chacun se déplaçant de dH pixels (sinon ils s'éloignent). Si dV est positif, les deux bords horizontaux se rapprochent, chacun se déplaçant de dV pixels (sinon ils s'éloignent). Quand dH ou dV est trop grand, on peut obtenir un rectangle vide (les quatre coordonnées sont alors forcées à zéro).

```
Rect    *rect;                  /* pointeur sur rectangle */
int     dH, dV;

dH = 5;                         /* demi-rétrécissement horizontal */
dV = -5;                        /* demi-élargissement vertical */
InsetRect(rect, dH, dV);        /* les coordonnées du rectangle sont modifiées */
```

Attention Aucune de ces opérations n'entraîne un effet visible à l'écran. QuickDraw ne fait que modifier les coordonnées d'un rectangle.

• La fonction **EmptyRect** teste si le rectangle pointé par son argument est vide, et retourne (bizarrement) FALSE dans ce cas (voir l'un des exemples du chapitre V).

```
Rect    r1, r2;                 /* deux rectangles */
int     x, y;                   /* pour recevoir une valeur booléenne */

SetRect(&r1, 10, 20, 30, 40);   /*      (10,20) est le coin supérieur gauche,
                                     (30,40) est le coin inférieur droit   */
r2 = r1;                /* assignation de structures, copie d'un rectangle dans l'autre */
OffsetRect(&r1, 10, -10);       /*      le coin supérieur gauche devient (20,10),
                                     le coin inférieur droit (40,30)   */
InsetRect(&r2, -5, 5);          /*      le coin supérieur gauche devient (5,25),
                                     le coin inférieur droit (35,35)   */
InsetRect(&r1, 15, -15);        /* les quatre coordonnées sont mises à zéro */
x = EmptyRect(&r2);             /* x prend la valeur TRUE (rectangle non vide) */
y = EmptyRect(&r1);             /* y prend la valeur FALSE (rectangle vide) */
```

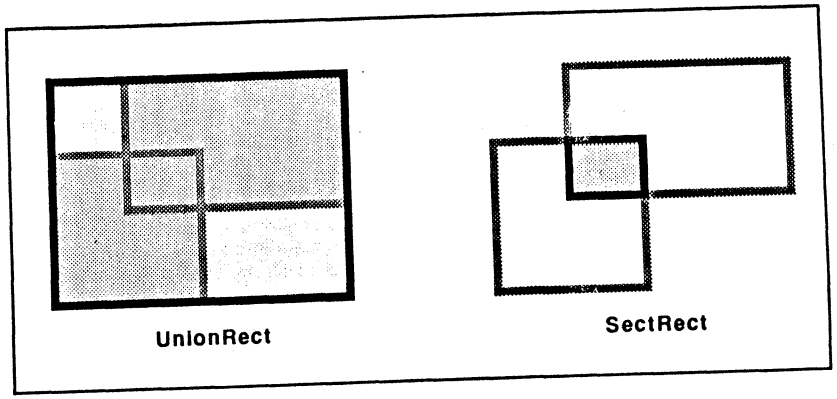



Figure III.13. Union et intersection de deux rectangles.

Calculs sur deux rectangles

• La procédure **UnionRect** calcule le plus petit rectangle qui contient chacun des deux rectangles dont les pointeurs sont passés en argument.

• La fonction **SectRect** calcule le rectangle intersection des deux rectangles dont les pointeurs sont passés en argument, et retourne la valeur TRUE si cette intersection est non vide. Si l'intersection est vide, la fonction retourne FALSE et les quatre coordonnées du rectangle résultant sont forcées à zéro.

• La fonction **EqualRect** teste si les deux rectangles dont les pointeurs sont passés en argument sont égaux, c'est-à-dire s'ils ont leur quatre coordonnées égales. Deux rectangles de taille identique ne sont pas égaux s'ils sont décalés l'un par rapport à l'autre. La valeur TRUE est retournée en cas d'égalité, FALSE sinon.

```
Rect    r1, r2, union, inter;           /* quatre rectangles */
int     x, y;

SetRect(&r1, 0, 20, 60, 70);
SetRect(&r2, 10, 0, 120, 30);
UnionRect(&r1, &r2, &union);        /* le rectangle union aura:
                                       (0,0) comme coin supérieur gauche et
                                       (120,70) comme coin inférieur droit */
x = SectRect(&r1, &r2, &inter);     /* le rectangle inter aura:
                                       (10,20) comme coin supérieur gauche et
                                       (60,30) comme coin inférieur droit...
                                       x est donc non nul (TRUE) */
y = EqualRect(&union, &inter);      /* y est nul (FALSE) */
```

Attention Aucune de ces opérations n'entraîne d'effet visible à l'écran.

Polygone

On fera toujours référence à un polygone par l'intermédiaire d'un handle sur une structure à taille variable, dont nous ne donnerons pas la définition exacte. Disons simplement que son premier champ *(PolySize)* est un entier qui contient la taille de la structure, en octets, et que le champ suivant *(PolyBBox)* est un rectangle, le plus petit rectangle englobant le polygone. Ensuite sont mémorisés tous les points constituant les sommets du polygone, points sur lesquels une application n'a pas à intervenir directement.

Un polygone est donc une structure de longueur variable. Sa forme sera définie par les opérations de dessin de lignes (**Line** et **LineTo**). Si le dernier sommet ne coïncide pas avec le premier dans la définition, un côté supplémentaire sera généré automatiquement.

La gestion d'un polygone se fait par l'intermédiaire de trois routines : deux pour la constitution du polygone, et une pour sa destruction.

La fonction **OpenPoly** crée un nouvel enregistrement de type polygone, l'ouvre pour mémoriser une définition de polygone et retourne un handle sur cet enregistrement. Tous les appels suivants de type **Line** et **LineTo** (procédures de dessin des lignes) seront gardés en mémoire pour définir la forme du polygone (ils sont rendus invisibles par un appel implicite à **HidePen**). Les côtés du polygone seront infiniment fins, quelles que soient les caractéristiques du crayon employé pour le décrire.

Attention Un seul polygone peut être ouvert à la fois.

La procédure **ClosePoly** ferme le polygone en cours de définition. Le champ *PolyBBox* est alors recalculé, de telle sorte que le rectangle contienne tous les sommets du polygone. Un appel implicite à **ShowPen** est effectué, pour rétablir le précédent niveau d'invisibilité du crayon (voir plus haut).

Quand on n'a plus besoin d'un polygone, la procédure **KillPoly** permet de le détruire et rend disponible la mémoire qu'il occupait. Le polygone est inutilisable après cet appel.

Exemple Définition et dessin d'un triangle.

```
Handle triPoly;                 /* triPoly est un handle sur polygone */

triPoly = OpenPoly( );          /* début de la définition, crayon rendu invisible */
   MoveTo(30,10);               /* on se positionne sur le premier sommet */
   LineTo(40,20);               /* tracé du premier côté, non visible à l'écran */
   LineTo(20,20);               /* tracé du deuxième côté, non visible à l'écran */
   LineTo(30,10);               /* tracé du troisième côté, non visible à l'écran */
ClosePoly( );                   /* fin de la définition, crayon de nouveau visible */
...
PaintPoly(triPoly);             /* triangle plein dessiné à l'écran */
...
KillPoly(triPoly);              /* on n'a plus besoin du polygone */
```

Calculs sur polygones

QuickDraw est pauvre en routines de calcul sur polygone. Une seule est en effet disponible : **OffsetPoly** qui déplace un polygone sans modifier sa forme ou sa taille. Le déplacement horizontal s'effectue de dH pixels (vers la droite si dH est positif, vers la gauche sinon), le déplacement vertical de dV pixels (vers le bas si dV est positif, vers le haut sinon).

```
Handle poly;                    /* handle sur polygone */
int    dH, dV;

dH = 5;                         /* déplacement horizontal */
dV = 10;                        /* déplacement vertical */
OffsetPoly(poly, dH, dV);       /* les composantes du polygone sont modifiées */
```

En pratique, c'est cette pauvreté dans les utilitaires qui fera souvent préférer l'utilisation d'une région à celle d'un polygone.

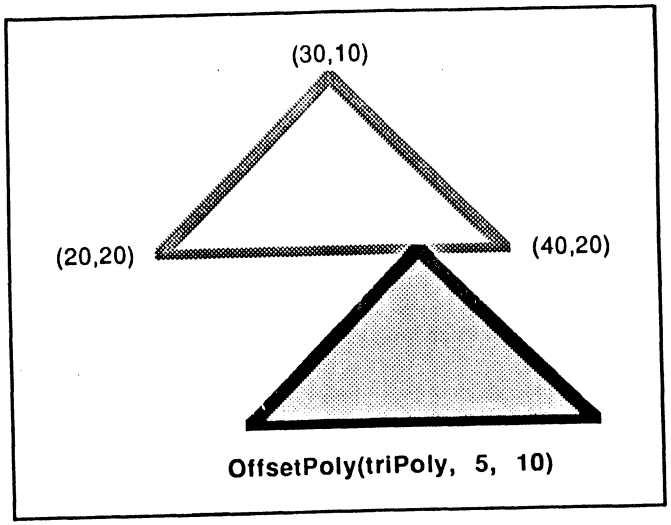


Figure III.14. Déplacement d'un polygone.

Région

On fera toujours référence à une région par l'intermédiaire d'un handle sur une structure à taille variable, dont nous ne donnerons pas la définition. Disons simplement que le premier champ *(rgnSize)* contient la taille exacte de la structure, en octets, et que le champ suivant *(rgnBBox)* est un rectangle, le plus petit rectangle englobant complètement la région. La suite est constituée de données définissant la forme de la région, sur lesquelles une application n'a pas à intervenir directement.

Comme pour le polygone, la région est une structure de longueur variable. Sa forme sera définie par une série de lignes et de formes obtenues en utilisant les opérations de dessin de lignes (**Line** et **LineTo**) et de formes (**FrameRect, FrameRRect, FrameOval, FramePoly** et **FrameRgn**).
Attention Les arcs sont ignorés dans la définition d'une région.

La région la plus simple est le rectangle. Dans ce cas, la structure a une taille de 10 octets seulement (pas de données additionnelles, seuls les deux premiers champs sont présents).

La gestion d'une région se fait par l'intermédiaire de quatre routines : une pour s'allouer un handle, deux pour la constitution de la région, et une pour sa destruction. On notera que la manière retenue pour créer une région diffère notablement du polygone, même si les principes de base sont identiques.

La fonction **NewRgn** crée un nouvel enregistrement de type région et retourne un handle. La procédure **OpenRgn** démarre la mémorisation de la définition de la région (aucun argument), et c'est à la fin de la définition, quand on ferme par **CloseRgn**, qu'on précise sur quelle région existante elle va s'appliquer. Comme pour les polygones, on n'a pas le droit d'ouvrir plus d'une région à la fois, et le crayon est rendu invisible pendant la définition. Le rectangle *rgnBBox* sera automatiquement recalculé.

Quand on n'a plus besoin d'une région, on appelle **DisposeRgn** pour la détruire et libérer la mémoire qu'elle occupait. La région n'est plus utilisable après cette opération.

La création d'une région obéit à des règles strictes : à chaque nouvelle instruction, on ajoute à l'ancienne région la nouvelle, et on retranche leur intersection. Cette propriété permet la création de « trous » dans les régions.

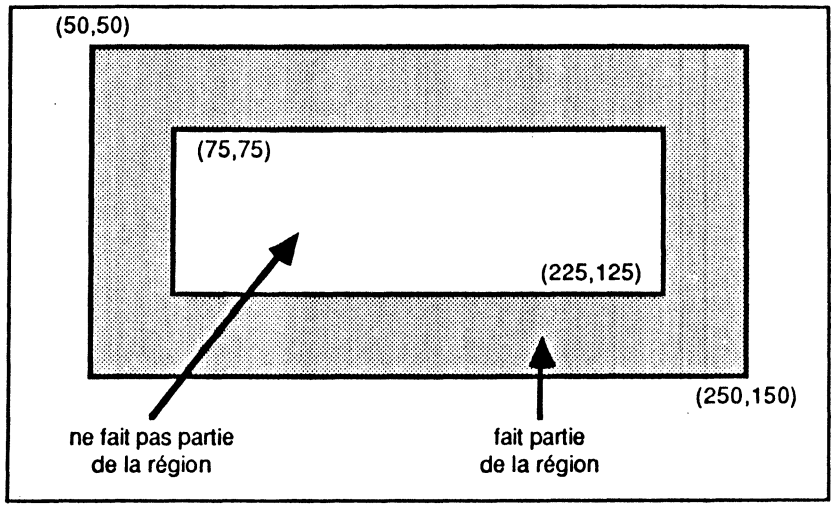


Figure III.15. La région de l'exemple.

Exemple Définition et dessin d'un rectangle évidé.

```
Handle rgn;                     /* handle sur région */
Rect   r;                       /* un rectangle */

rgn = NewRgn( );                /* obtention d'un handle sur région */
OpenRgn( );                     /* début de la définition, crayon rendu invisible */
    SetRect(&r,50, 50, 250, 150); /* le grand rectangle */
    FrameRect(&r);              /* on le trace */
    InsetRect(&r, 25, 25);      /* le petit rectangle */
    FrameRect(&r);              /* on fait le trou */
CloseRgn(rgn);                  /* fin de la définition, crayon de nouveau visible */
...
PaintRgn(rgn);                  /* rectangle évidé dessiné à l'écran */
...
DisposeRgn(rgn);                    /* on n'a plus besoin de la région */
```

Mise en place de régions particulières

• Pour rendre une région vide (c'est-à-dire remettre sa définition à blanc), on peut utiliser la procédure **SetEmptyRgn**. Le handle sur la région reste utilisable.

• Les deux procédures **RectRgn** et **SetRectRgn** font l'une et l'autre d'une région existante une région en forme de rectangle (si le rectangle est vide, alors la région devient vide). La définition de la région précédente est évidemment perdue.

• La procédure **CopyRgn** donne à une région existante la même forme qu'une autre région. Il y a copie effective de la définition de la région : les handles sont différents.

Attention Dans tous les cas, la région affectée doit exister (donc avoir été créée par **NewRgn**).

Exemple

```
Handle rgn1, rgn2;              /* handles sur région */
Rect   r;                       /* un rectangle */

rgn1 = NewRgn( );
SetRect(&r, 20, 50, 120, 150);      /* création d'un rectangle */
RectRgn(rgn1, &r);                  /* la région aura la forme du rectangle */
SetEmptyRgn(rgn1);                  /* la région est maintenant vide */
SetRectRgn(rgn1, 20, 50, 120, 150); /* on refait différemment la même région */
rgn2 = NewRgn( );
CopyRgn(rgn1, rgn2);                /* rgn2 est maintenant identique à rgn1 */
```

Calculs sur une région

• La procédure **OffsetRgn** déplace la région sans affecter sa forme ni sa taille. Le déplacement horizontal s'effectue de dH pixels (vers la droite si dH est positif, vers la gauche sinon), le déplacement vertical de dV pixels (vers le bas si dV est positif, vers le haut sinon).

```
Handle rgn;                     /* handle sur région */
int    dH, dV;

dH = 50;                        /* déplacement horizontal */
dV = 62;                        /* déplacement vertical */
OffsetRgn(rgn, dH, dV);         /* déplacement de la région */
```

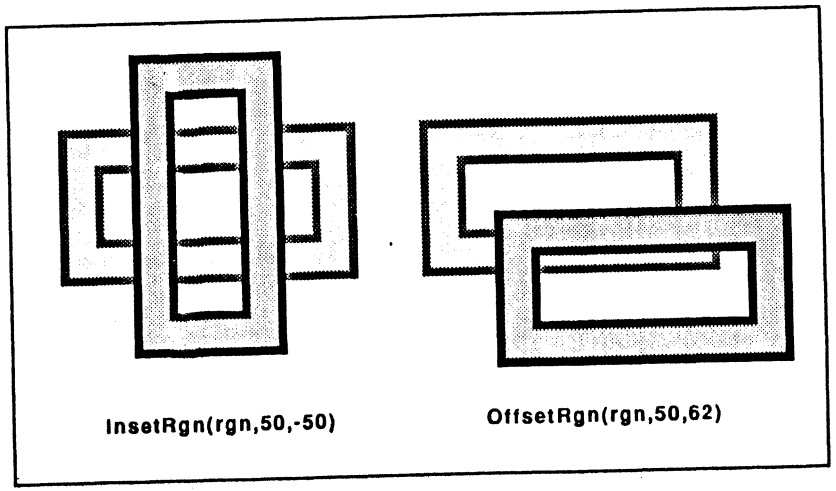


Figure III.16. Déformation et déplacement d'une région

• La procédure **InsetRgn** agrandit ou rétrécit une région sans déplacer son centre. Toutes les coordonnées définissant la région sont modifiées de telle sorte qu'elles se rapprochent du centre pour des arguments positifs, s'en éloignent pour des arguments négatifs.

```
Handle rgn;                 /* handle sur région */
int    dH, dV;

dH = 50;                    /* déformation horizontale */
dV = -50;                   /* déformation verticale */
InsetRgn(rgn, dH, dV);      /* déformation de la région */
```

• La fonction **EmptyRgn** teste si la région passée en argument est vide, et retourne TRUE dans ce cas.

Attention Aucune de ces opérations n'entraîne d'effet visible à l'écran, le concept étant purement mathématique. On remarque que la syntaxe est identique à celle affectant les rectangles, pourvu qu'on remplace les pointeurs sur rectangle par des handles sur région.

Calculs sur deux régions

• La procédure **UnionRgn** calcule l'union des deux régions (la plus petite région contenant les deux régions).

• La procédure **SectRgn** calcule l'intersection des deux régions (la plus grande région incluse dans les deux régions à la fois).

• La procédure **DiffRgn** calcule la différence des deux régions (partie de la première région qui n'appartient pas à la seconde).

• La procédure **XorRgn** calcule le « ou exclusif » de deux régions (différence entre l'union et l'intersection).

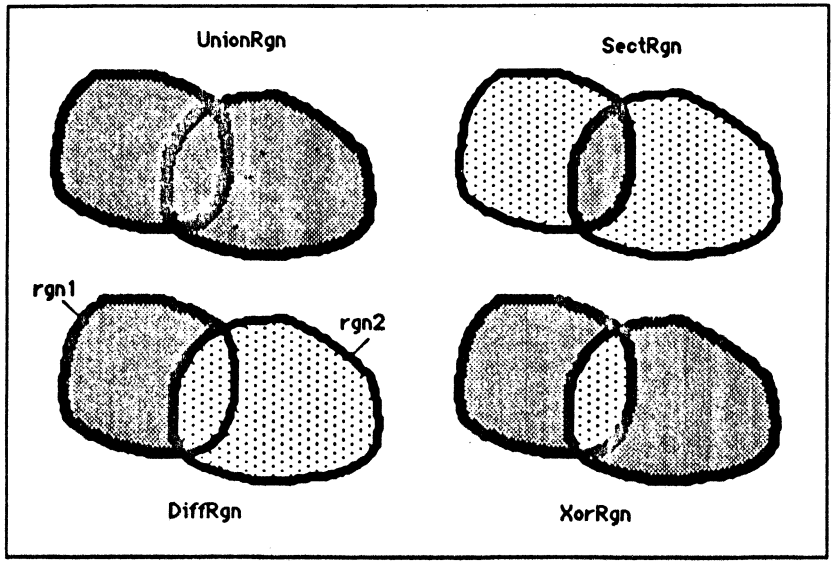


Figure III.17. Union, intersection, différence et « ou » exclusif de deux régions.

Attention Dans tous les cas, la région résultante doit exister (donc avoir été créée par **NewRgn**). Ce pourra être éventuellement la région vide. Chacune des trois procédures admet trois arguments : le handle sur la première région, le handle sur la seconde région, et le handle sur la région résultante.


```
Handle rgn1, rgn2;          /* handles sur région */
Handle union, inter, diff, xor;   /* handles sur région */

            /*on suppose que rgn1 et rgn2 existent déjà, on crée les autres */
union = NewRgn( );
inter = NewRgn( );
diff = NewRgn( );
xor = NewRgn( );
UnionRgn(rgn1, rgn2, union);   /* l'union de rgn1 et rgn2 est mise dans union */
SectRgn(rgn1, rgn2, inter);    /* l'intersection de rgn1 et rgn2 est mise dans inter */
DiffRgn(rgn1, rgn2, diff);     /* la différence rgn1 - rgn2 est mise dans diff */
XorRgn(rgn1, rgn2, xor);       /* le ou-exclusif entre rgn1 et rgn2 est mis dans xor */
...
            /* on n'a plus besoin des régions résultantes, on libère de la mémoire */
DisposeRgn(union);
DisposeRgn(inter);
DisposeRgn(diff);
DisposeRgn(xor);
```

Notons que de ces quatre procédures de calcul, seule **DiffRgn** n'admet pas la commutativité de ses deux premiers arguments.

• La fonction **EqualRgn** teste si les deux régions désignées par leur handle sont égales (même forme, même taille, même localisation) et retourne TRUE dans ce cas, FALSE sinon. Notons que deux régions vides sont égales, puisqu'elles apparaissent comme un rectangle dont les quatre coordonnées sont forcées à zéro.

Attention Aucune de ces cinq opérations n'entraîne d'effet visible à l'écran. Ce sont uniquement des calculs mathématiques.

Tests d'intersection

• La fonction **PtInRect** teste si le pixel associé au point passé en premier argument appartient au rectangle donné en deuxième argument, et retourne TRUE si oui, FALSE si non.

• La fonction **PtInRgn** teste si le pixel associé au point en premier argument appartient à la région donnée en deuxième argument, et retourne TRUE si oui, FALSE si non.

• La fonction **RectInRgn** teste l'intersection du rectangle et de la région et retourne TRUE si au moins un pixel appartient à la fois à l'un et à l'autre, FALSE sinon.

Exemple

```
Point  pt1, pt2;            /* deux points */
Rect   r;                   /* un rectangle */
Handle rgn;                 /* un handle sur région */
int    x, y, z;

SetRect(&r, 10, 10, 20, 20);
SetPt(&pt1, 10, 10);
SetPt(&pt2, 20, 20);
x = PtInRect(&pt1, &r);     /* x prendra la valeur TRUE (non nul) */
y = PtInRect(&pt2, &r);     /* y prendra la valeur FALSE (nul) */
rgn = NewRgn( );
RectRgn(rgn, &r);
x = PtInRgn(&pt1, rgn);     /* x prendra la valeur TRUE (non nul) */
y = PtInRgn(&pt2, rgn);     /* y prendra la valeur FALSE (nul) */
z = RectInRgn(&r, rgn);     /* z prendra la valeur TRUE (non nul) */
```

Utilitaires de mise à l'échelle

Il est souvent pratique de pouvoir changer la taille de certains objets en respectant une déformation proportionnelle à la taille de deux rectangles, dits rectangle source et rectangle destination. Les cinq procédures suivantes permettent ces changements d'échelle.

• La procédure **ScalePt** ajuste les coordonnées du point en fonction du ratio des dimensions des deux rectangles :

(nle abscisse) = (anc abscisse) × (largeur destination) / (largeur source)
(nle ordonnée) = (anc ordonnée) × (hauteur destination) / (hauteur source)

Cette opération ne tient aucun compte de la localisation de rectangles, au contraire des suivantes.

```
Point   pt;                          /* un point */
Rect    source, dest;                /* deux rectangles */

SetRect(&source, 60, 60, 240, 160);
SetRect(&dest, 150, 100, 270, 300);
SetPt(&pt, 150, 85);
ScalePt(&pt, &source, &dest);        /* pt représente maintenant le point (100,170) */
```

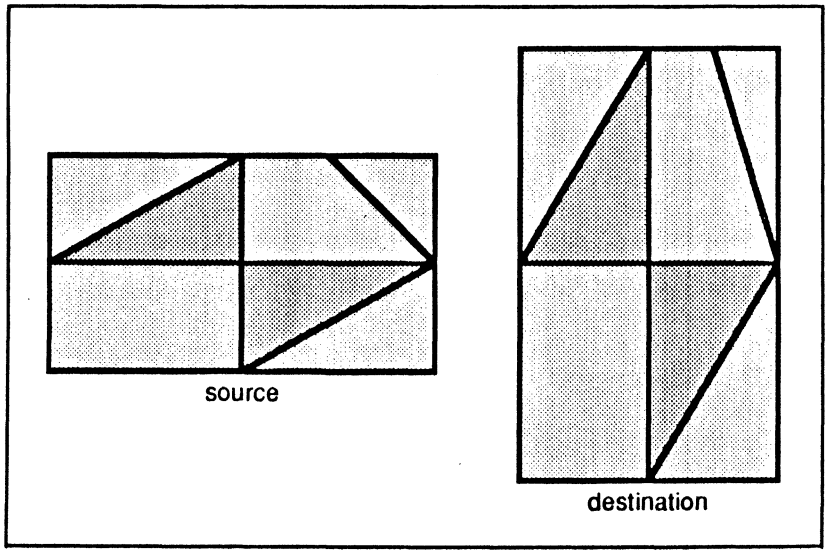


Figure III.18. La mise à l'échelle (procédures Map).

• La procédure **MapPt** ajuste la position d'un point du rectangle source au rectangle destination. **MapRect**, **MapPoly** et **MapRgn** transforment tous les points d'un rectangle, d'un polygone et d'une région suivant les règles de **MapPt**. Dans l'exemple suivant, l'abscisse du point origine est exactement au milieu du rectangle source et l'ordonnée au premier quart. Ces proportions sont identiques à l'arrivée, entre le point recalculé et le rectangle destination.


```
Point   pt;                          /* un point */
Rect    source, dest;                /* deux rectangles */

SetRect(&source, 60, 60, 240, 160);
SetRect(&dest, 150, 100, 270, 300);
SetPt(&pt, 150, 85);
MapPt(&pt, &source, &dest);          /* pt représente maintenant le point (210,150) */
```

Encore une fois, il s'agit là de concepts mathématiques, et aucun effet visible n'est à attendre de ces procédures de calcul.

Calculs sur texte

Les polices de caractères pouvant être à espacement proportionnel, il est intéressant de savoir calculer la largeur d'un caractère ou d'un ensemble de caractères. Ces calculs sont effectués en tenant compte bien évidemment de la police courante (un caractère Chicago est plus large qu'un caractère Times), de la taille courante (la taille désigne la hauteur des caractères, mais en affecte également la largeur) et du style courant (un caractère gras est plus large qu'un caractère normal).

QuickDraw propose deux sortes de routines pour mener à bien ces calculs : quatre fonctions permettent de calculer la largeur d'un caractère ou d'une chaîne de caractères, quatre procédures permettent de connaître le rectangle englobant exactement le caractère ou la chaîne de caractères désignés.

• **CharWidth** admet en argument un caractère, **StringWidth** un pointeur sur une chaîne de type Pascal, **CStringWidth** un pointeur sur une chaîne de type C et **TextWidth** un pointeur sur un ensemble de caractères ni Pascal ni C suivi du nombre de caractères. Ces quatre fonctions retournent la largeur de l'ensemble des caractères en nombre de pixels.

```
int     L1, L2, L3, L4;              /* les quatre résultats */
char    car = 'A';                   /* car est un caractère */
char    pas[ ] = "\6Pascal";         /* pas pointe sur une chaîne de type Pascal */
char    Cstr[ ] = "Cstring";         /* Cstr pointe sur une chaîne de type C */
char    text[5] = {'T','e','x','t','e'};  /* text n'est ni de type Pascal, ni de type C */

L1 = CharWidth((int) car);           /* largeur du caractère */
L2 = StringWidth(pas);               /* largeur de la chaîne de type Pascal */
L3 = CStringWidth(Cstr);             /* largeur de la chaîne de type C */
L4 = TextWidth(text, 5);             /* largeur du texte */
```

Notons l'intérêt de la dernière fonction : pour calculer la largeur occupée par les vingt premiers caractères d'une chaîne de type C, on procédera ainsi :

```
int     L20;                         /* recevra le résultat */
char    texte[ ] = "Ceci est une chaîne de type C qui dépasse vingt caractères";

L20 = TextWidth(texte, 20);          /* largeur des 20 premiers caractères */
```

Une chaîne de type Pascal se prêtera moins facilement à ce genre de calculs, puisque suivant sa signification ASCII, son premier élément peut créer une largeur artificielle qu'il faut éliminer.

• Les quatre procédures suivantes sont le pendant des quatre fonctions précédentes. **CharBounds**, **StringBounds**, **CStringBounds** et **TextBounds** admettent chacune un argument de plus que leur équivalent : un pointeur sur rectangle. Dans ce rectangle sera retournée la place exacte occupée par le ou les caractères désignés, aussi bien en largeur qu'en hauteur. La position du rectangle dépend de la position courante du crayon.

```
Rect   r1, r2, r3, r4;                 /* quatre rectangles */

CharBounds((int) car, &r1);            /* r1 recevra le résultat */
StringBounds(pas, &r2);                /* r2 recevra le résultat */
CStringBounds(Cstr, &r3);              /* r3 recevra le résultat */
TextBounds(text, 5, &r4);              /* r4 recevra le résultat */
```

Ces huit routines seront d'un grand intérêt, par exemple pour centrer des textes dans des fenêtres de visualisation, aussi bien horizontalement que verticalement : connaissant le rectangle nécessaire à la représentation d'un texte, il sera facile d'ajuster la localisation du crayon pour centrer ledit rectangle, ainsi qu'on peut le voir sur la figure suivante :

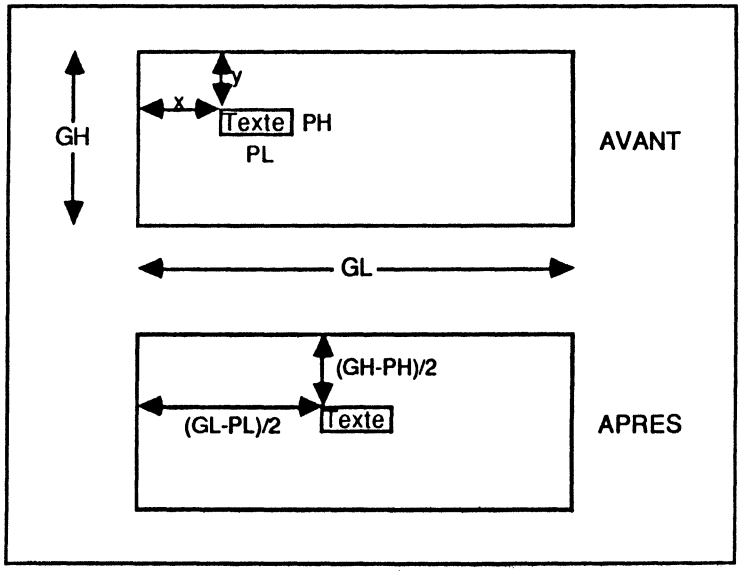


Figure III.19. Centrer un texte.

Dans cette figure, GL et GH désignent la largeur et la hauteur du rectangle dans lequel le texte doit être centré. Grâce à une procédure de type **TextBounds**, on peut connaître le rectangle englobant le texte, donc sa largeur (PL) et sa hauteur (PH), et par un simple calcul de translation, la position relative (x,y) de son coin supérieur gauche par rapport à celui du grand rectangle.

Il suffit alors de déplacer la localisation du crayon de la valeur [(GL−PL)/2 − x] horizontalement, et de [(GH−PH2) − y] verticalement, et de dessiner le texte : il est centré !

Tout ceci s'écrit :

```
Rect   GR = {50, 60, 280, 160};    /* le grand rectangle, défini arbitrairement */
char   Cstr[ ] = "Texte";          /* chaîne de type C */
Rect   r;                          /* rectangle contenant le texte */
int    x, y, PL, PH, GL, GH;       /* les variables de calcul */

MoveTo(GR.left, GR.top);           /* on positionne le crayon
                                      en haut à gauche du grand rectangle */
GL = GR.H2 - GR.H1 ;               /* grande largeur */
GH = GR.V2 - GR.V1 ;               /* grande hauteur */
CStringBounds(Cstr, &r);           /* et le petit rectangle */
x = r.H1 - GR.H1 ;                 /* abscisse relative du coin supérieur gauche */
y = r.V1 - GR.V1 ;                 /* ordonnée relative du coin supérieur gauche */
PL = r.H2 - r.H1 ;                 /* petite largeur */
PH = r.V2 - r.V1 ;                 /* petite hauteur */
Move( (GL - PL) / 2 - x, (GH - PH) / 2 - y); /* on change la localisation du crayon */
DrawCString(Cstr);                 /* on dessine, c'est centré! */
```

A aucun moment on n'a besoin de connaître la localisation exacte du crayon, ni même la position du petit rectangle par rapport à cette localisation. On calcule des positions relatives et des déplacements relatifs : ces calculs sont donc de portée générale.

• Mentionnons pour être plus complet deux procédures qui ont à voir avec le calcul sur les textes. Quand un traitement de texte doit effectuer ce qu'on appelle une justification totale (les lignes de texte sont alignées à la fois sur le bord gauche et le bord droit de la feuille), il a deux manières de faire : soit il augmente l'espace entre chaque mot, soit il augmente l'espace entre chaque lettre. QuickDraw propose une procédure pour chacune des deux solutions : **SetSpaceExtra** permet d'augmenter l'espace entre les mots et **SetCharExtra** celui entre les lettres, ainsi que deux fonctions pour connaître la valeur courante de chacun de ces champs : **GetSpaceExtra** et **GetCharExtra**. Notre but n'étant pas d'écrire un traitement de texte, nous ne développerons pas davantage la question.

DESSINER AVEC QUICKDRAW

Après avoir passé en revue tous les concepts mathématiques et graphiques de QuickDraw, il est sans doute temps de voir comment on peut dessiner ! Rappelons une dernière fois quelques éléments de base : on dessine exclusivement dans le grafport courant, à l'intersection de deux rectangles (le rectangle frontière de la pixel map et le *PortRect* du grafport) et de deux régions (la clip region et la région visible).

Il faudra toujours garder à l'esprit quand on dessine qu'un pixel ne représente pas quelque chose de carré à l'écran (contrairement au Macintosh) : le ratio hauteur d'un pixel/largeur d'un pixel est de 6 sur 5 en mode 320, et de 12 sur 5 en mode 640. Il faudra en tenir compte (par exemple dans la taille du crayon) pour dessiner les traits verticaux de la même épaisseur que les traits horizontaux.

DESSIN DE FORMES

Dessin de lignes

Deux procédures permettent de dessiner une ligne, plus précisément, un segment de droite. L'origine du segment est la localisation courante du crayon. Avec **LineTo**, l'autre extrémité du segment est définie en absolu (coordonnées locales), avec **Line**, en relatif. Si nous supposons que le crayon se trouve sur le point de coordonnées (20,30), les deux instructions suivantes sont équivalentes :

```
LineTo(60,80);          /* on va en absolu jusqu'au point (60,80) */
  /* ou */
Line(40,50);            /* on va en relatif jusqu'au point (20+40,30+50) */
```

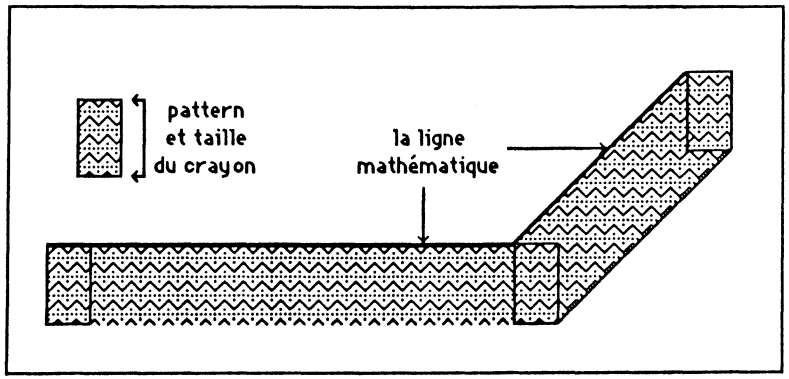


Figure III.20. Dessin de deux lignes.

Dans les deux cas, la localisation du crayon est modifiée, l'autre extrémité du segment devenant la nouvelle localisation (revoir comment a été défini le triangle dans le paragraphe consacré à la définition des polygones, plus haut).

La ligne est dessinée avec les caractéristiques du crayon : taille, pattern, masque, mode de transfert. Pour un point donné, c'est un rectangle de pixels situé vers le bas et vers la droite de ce point qui est dessiné.

Dessin de rectangles

Cinq procédures permettent de dessiner graphiquement des rectangles. Nous retrouverons à l'identique ces cinq procédures pour dessiner les autres formes étudiées dans les parties précédentes.

FrameRect dessine uniquement le contour du rectangle, en utilisant les caractéristiques du crayon (taille, pattern, masque et mode). Tous les pixels dessinés sont *intérieurs* au rectangle, même si la taille du crayon est supérieure à (1,1).

PaintRect dessine le contenu de la forme rectangulaire avec le pattern du crayon (limité à son masque) comme motif de remplissage et le mode du crayon comme mode de transfert, ce qui signifie que l'uniformité du remplissage peut être affectée par le fond sur lequel on dessine.

EraseRect « efface » le contenu de la forme rectangulaire, c'est-à-dire la remplit avec le pattern de fond (que nous avons appelé *bkPat*) sans masque et en mode *Copy*.

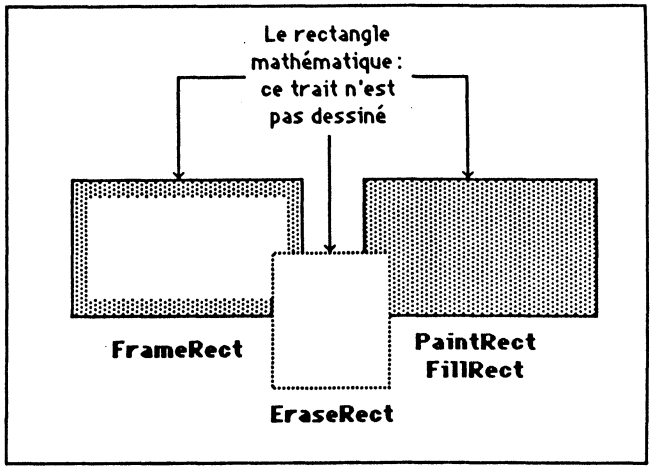


Figure III.21. Dessins de rectangles.

InvertRect « inverse » le contenu de la forme rectangulaire. Cette inversion se fait au niveau du bit, ce qui signifie que la couleur 0 devient 15, la couleur 1 devient 14, la couleur 2 devient 13, etc. Avec la palette standard en mode 320 (voir plus haut), le noir devient blanc, le gris foncé devient gris clair, le brun devient bleu pervenche, le pourpre devient lilas, le bleu devient bleu clair, le vert foncé devient vert, l'orange devient jaune et le rouge devient chair. Et réciproquement. Ce qui confirme le fait que cette palette de couleurs n'a pas été choisie au hasard.

FillRect remplit le contenu de la forme rectangulaire par un pattern désigné en argument. Le mode utilisé est *Copy*, ce qui signifie que l'uniformité du remplissage ne sera pas affectée par le fond sur lequel on dessine. Fonctionnellement, **FillRect** agit comme **EraseRect**, avec un pattern différent.

L'emploi de ces procédures est ultra-simple. Les quatre premières n'utilisent qu'un argument : un pointeur sur le rectangle à représenter. **FillRect** utilise un deuxième

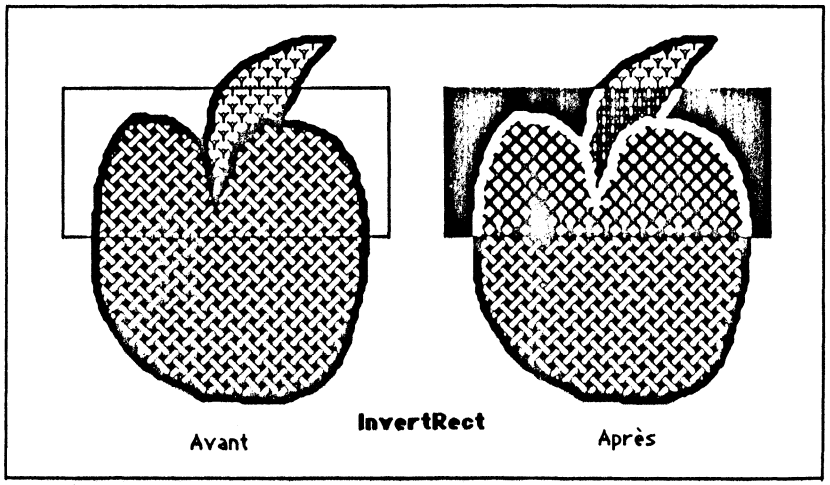


Figure III.22. Inversion du rectangle.

argument : un pointeur sur le pattern qui servira de motif de remplissage. Ces cinq procédures ont un effet immédiat à l'écran.

Notons qu'aucune de ces procédures n'affecte la localisation du crayon.

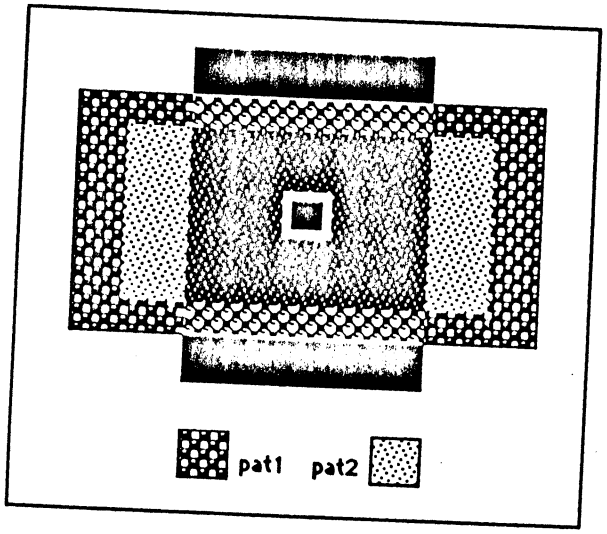


Figure III.23. Dessins avec rectangles.

Exemple

```
Rect    r;                          /* un rectangle */
char    pat1[32], pat2[32];         /* et deux patterns */

SetRect(&r, 50, 50, 150, 100);      /* définition du rectangle */
...                                 /* définition des patterns, voir plus haut */
SetPenPat(pat1);                    /* on fixe le pattern du crayon */
SetPenSize(10, 6);                  /* et sa taille */
                        /* les autres caractéristiques sont celles du grafport courant */
FrameRect(&r);                      /* dessin du cadre */
InsetRect(&r, 5, 3);                /* rétrécissement du rectangle */
FillRect(&r, pat2);                 /* on remplit le nouveau rectangle */
SetRect(&r, 75, 40, 125, 110);      /* nouveau rectangle */
InvertRect(&r);                     /* inversion de ce rectangle */
InsetRect(&r, 20, 30);              /* rétrécissement du rectangle */
EraseRect(&r);                      /* effacement de son contenu */
SetSolidPenPat(0);                  /* on change le pattern du crayon: couleur noire */
InsetRect(2, 2);                    /* on rétrécit encore le rectangle */
PaintRect(&r);                      /* on remplit son contenu */
        /* et voilà le résultat (figure III.23) */
```

Au risque de se répéter, précisons qu'aucune de ces procédures ne dessine en dehors du rectangle désigné. Si le rectangle a pour taille m × n points, il englobe (m−1) × (n−1) pixels et les procédures n'en affectent pas un de plus.

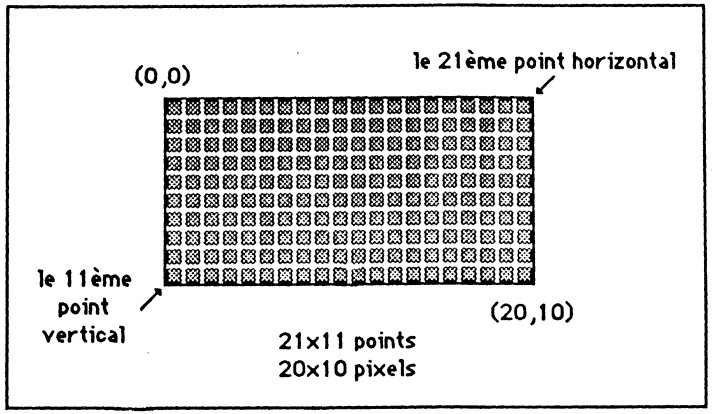


Figure III.24. Les pixels d'un rectangle.

Dessin de rectangles arrondis

Les cinq procédures vues pour le rectangle s'appliquent à l'identique pour le rectangle arrondi. Elles s'appellent **FrameRRect**, **PaintRRect**, **EraseRRect**, **InvertRRect** et **FillRRect**. Elles admettent trois arguments à la place du simple pointeur sur rectangle des procédures vues précédemment : un pointeur sur le rectangle qui englobe le rectangle arrondi, un diamètre de courbure horizontal et un diamètre de courbure vertical qui définissent les arrondis. **FillRRect** admet bien sûr en quatrième argument un pointeur sur pattern.

FrameRRect (&r, dH, dV) où r est un rectangle et dH et dV des entiers dessinera le contour du rectangle arrondi de la figure III.25. Il faut bien remarquer qu'aucune des cinq procédures ne dessine un pixel à l'extérieur du rectangle arrondi ni ne modifie la localisation du crayon.

Voir un exemple d'utilisation en fin de chapitre V, ainsi que pour les autres types de formes élémentaires.

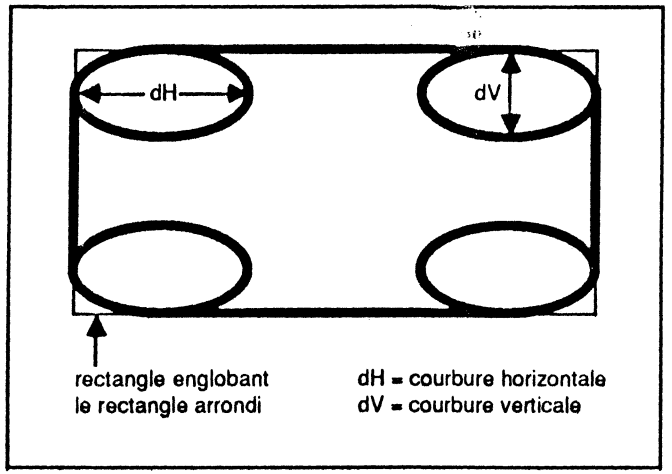


Figure III.25. Les éléments constituant le rectangle arrondi.

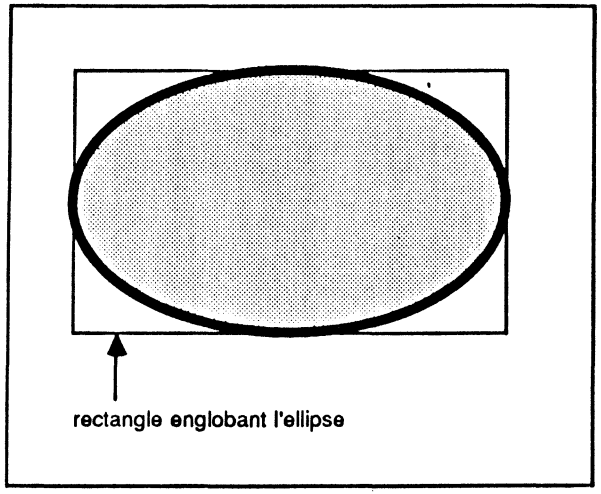


Figure III.26. L'ellipse est parfaitement définie par le rectangle circonscrit.

Dessin d'ellipses

Les cinq procédures vues pour le rectangle s'appliquent pour l'ellipse, et emploient les mêmes arguments. Le rectangle circonscrit définit parfaitement l'ellipse. Les procédures s'appellent **FrameOval**, **PaintOval**, **EraseOval**, **InvertOval** et **FillOval**. Là encore, aucun pixel n'est dessiné en dehors de l'ellipse, et la localisation du crayon n'est pas affectée.

Dessin d'arcs

Un arc est une portion d'ellipse. Pour définir cette portion, il faut l'ellipse, donc le rectangle circonscrit, et deux angles délimitant la portion. On notera la manière dont QuickDraw calcule ces angles, par référence au rectangle qui englobe le tout. L'origine de ces angles est l'axe vertical dirigé vers le haut (midi) et les angles positifs se calculent dans le sens des aiguilles d'une montre. Tous les angles sont donnés en degrés et traités modulo 360.

Là encore, les cinq procédures existent. Elles s'appellent **FrameArc**, **PaintArc**, **EraseArc**, **InvertArc** et **FillArc**. Trois arguments : pointeur sur rectangle, angle de début et angle de l'arc au lieu du seul pointeur sur rectangle des procédures précédentes. Ces procédures n'affectent pas la localisation du crayon.

Rappelons une petite particularité : la procédure **FrameArc** n'a aucune influence sur la constitution d'une région. En d'autres termes, il est impossible d'inclure un arc dans la définition des régions.

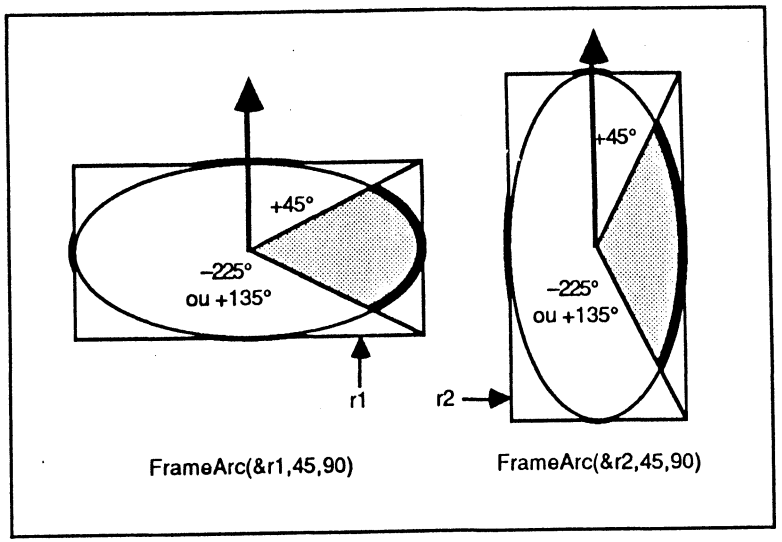


Figure III.27. Les angles sont calculés par référence au rectangle circonscrit.

Dessin de régions

Une région une fois définie, elle peut être dessinée comme toutes les formes élémentaires, et on retrouve également ici les cinq procédures de dessin. L'argument principal n'est plus un pointeur sur rectangle mais un handle sur région. L'utilisation est parfaitement identique. Ces procédures se nomment **FrameRgn**, **PaintRgn**, **EraseRgn**, **InvertRgn** et **FillRgn.**

Ici encore, ces procédures n'affectent que les pixels *intérieurs* à la région désignée, et notamment **FrameRgn**, quelle que soit la taille du crayon. Elles n'affectent pas sa localisation.

Dessin de polygones

Pas de surprise en ce qui concerne les polygones, les cinq procédures sont là : **FramePoly**, **PaintPoly**, **ErasePoly**, **InvertPoly** et **FillPoly** admettent comme argument essentiel un handle sur polygone.

Il faut cependant noter une différence fondamentale avec les autres procédures de ce type, liée à la définition intrinsèque des polygones. Le polygone est constitué de lignes, et **FramePoly** ne fait que dessiner des lignes, de la façon vue plus haut. Les pixels sont donc tracés pour partie *extérieurement* au polygone. La localisation du crayon change à chaque ligne dessinée, mais pas globalement, puisque le dernier point coïncide avec le premier.

DESSIN DE TEXTES

Les routines de calculs sur texte allaient quatre par quatre, il en est de même des procédures de dessin de texte. Pour dessiner un caractère, on utilisera **DrawChar**. Pour dessiner une chaîne de type Pascal, on utilisera **DrawString**. Pour dessiner une

chaîne de type C, on utilisera **DrawCString**. Enfin pour dessiner une chaîne de caractères ni Pascal ni C, on utilisera **DrawText** (en précisant le nombre de caractères qu'elle contient).

Toutes ces procédures utilisent évidemment les caractéristiques du grafport courant attachées au dessin de texte. Elles affectent toutes la localisation du crayon, de telle sorte qu'il se positionne pour dessiner la chaîne suivante. En aucun cas l'une de ces procédures n'est capable d'écrire sur plusieurs lignes. C'est à l'application de gérer la mise en page, par des **Move** ou des **MoveTo** appropriés.

```
char    car = 'A';                  /* car est un caractère */
char    pas[ ] = "\6Pascal";        /* pas pointe sur une chaîne de type Pascal */
char    texte[ ] = "Ceci est une chaîne de type C qui dépasse vingt caractères";

MoveTo(10,10);                      /* on positionne le crayon */
DrawChar((int) car);                /* on dessine le caractère */
Move(5,0);                          /* on laisse un peu d'espace */
DrawString(pas);                    /* on dessine la chaîne de type Pascal */
MoveTo(10,30);                      /* on positionne le crayon plus bas */
DrawCString(texte);                 /* on dessine la chaîne de type C */
MoveTo(10,50);                      /* on positionne le crayon plus bas */
DrawText(texte, 20);                /* on dessine les 20 premiers caractères seulement */
```

A l'issue de cet exemple, on aura trois lignes de texte cadrées à gauche dans le grafport courant. Voir les chapitres X et XII pour d'autres exemples.

PICTURE

Constitution de pictures

Nous avons vu plus haut sur quels principes repose la picture. Nous allons voir, maintenant que nous savons dessiner des formes élémentaires et du texte, comment créer une picture. Attention, la version 1.02 de QuickDraw ne connaissant pas encore les pictures, tout ce qui est dit ici est sujet à modifications. C'est toutefois ainsi que cela fonctionne sur Macintosh, et il serait étonnant que cela change !

Une picture sera mémorisée dans une structure à taille variable, et toujours désignée par l'intermédiaire d'un handle sur cette structure, dont nous ne donnerons pas la définition exacte. Le premier champ *(picSize)* contient la taille exacte de la structure des octets. Le deuxième champ *(picFrame)* est un rectangle, le plus petit rectangle englobant la picture. On trouvera ensuite les données constitutives de la picture proprement dite, suivant une codification QuickDraw.

La gestion d'une picture se fait par l'intermédiaire de trois routines : une pour s'allouer un handle et ouvrir le mode définition, une pour fermer le mode définition et une pour sa destruction. On notera que la manière retenue pour gérer une picture est identique à la gestion du polygone.

La fonction **OpenPicture** retourne un handle sur la nouvelle picture. En argument, un pointeur sur rectangle qui représentera le cadre de la picture. La fonction appelle **HidePen**, donc plus rien ne sera dessiné à l'écran à partir de ce moment. A la place, tout appel à une procédure de dessin (dessin de forme ou dessin de texte) sera mémorisé suivant un code propre à QuickDraw et entrera dans la définition de la picture.

L'enregistrement de la picture se termine par l'appel de la procédure **ClosePicture**. Cette procédure rétablit le niveau d'invisibilité du crayon en appelant **ShowPen**. Comme pour les polygones et les régions, on n'a pas le droit d'ouvrir plus d'une picture à la fois.

Quand on n'a plus besoin d'une picture, on appelle **KillPicture** pour la détruire et libérer la mémoire qu'elle occupait. La picture n'est plus utilisable après cette opération.

Nous passons volontairement sous silence la possibilité qu'offre QuickDraw de glisser des commentaires au milieu de la définition des pictures, puisque dans son mode de fonctionnement standard, il n'en tient pas compte. Cet aspect est à réserver aux programmeurs de niveau avancé.

A titre d'exemple, créons une picture qui dessinera le drapeau de la Croix-Rouge en mode 320.

```
Handle  croix;                  /* handle sur picture */
Rect    r1, r2;                 /* deux rectangles */
char    rouge[32];              /* place pour un pattern en mode 320 */

SolidPattern(rouge, 7);     /* l'entrée 7 est la couleur rouge dans la palette standard */
SetRect(&r1, 0, 0, 100, 100);
croix = OpenPicture(&r1);       /* début de définition */
    PenNormal( );
    FrameRect(&r1);
    SetRect(&r2, 30, 10, 70, 90);
    FillRect(&r2, rouge);
    SetRect(&r2, 10, 30, 90, 70);
    FillRect(&r2, rouge);
ClosePicture( );                /* fin de définition */
OffsetRect(&r1, 30, 30);
DrawPicture(croix, &r1);        /* dessin sans mise à l'échelle */
InsetRect(&r1, 30, 30);
OffsetRect(&r1, 0, 70);
DrawPicture(croix, &r1);        /* dessin avec mise à l'échelle */
KillPicture(croix);             /* on n'a plus besoin de la picture */
```

Représentation de pictures

Nous avons vu comment enregistrer les éléments constitutifs d'une picture. Il ne reste plus qu'à les faire apparaître à l'écran. Pour ce faire, une procédure unique : **DrawPicture**, qui permet non seulement la représentation d'une picture, mais aussi sa mise à l'échelle.

DrawPicture admet deux arguments. Le premier est un handle sur la picture à représenter. Le second est un pointeur sur le rectangle de destination dans lequel la picture va être représentée. Nous avons vu que la définition de la picture retient la notion de rectangle englobant celle-ci. Si le rectangle de destination a même taille que le rectangle circonscrit, pas de problème : la picture est dessinée sans déformation. Sinon, il y a mise à l'échelle, ce qui peut conduire à des résultats fort décevants, QuickDraw faisant du mieux qu'il peut. Le résultat sera excellent sur les éléments constitutifs du type appels de dessin de formes vus ci-dessus (voir exemple), mais beaucoup moins bon sur les mémorisations de type pixel map. Dans ce dernier cas, la mise à l'échelle est d'autant meilleure que les dimensions des deux rectangles sont des multiples exacts les unes des autres.

TRANSFERT DE PIXELS

Trois procédures gèrent ce que nous appellerons un transfert de pixels, c'est-à-dire le déplacement d'un groupe désigné de pixels d'un endroit à l'autre.

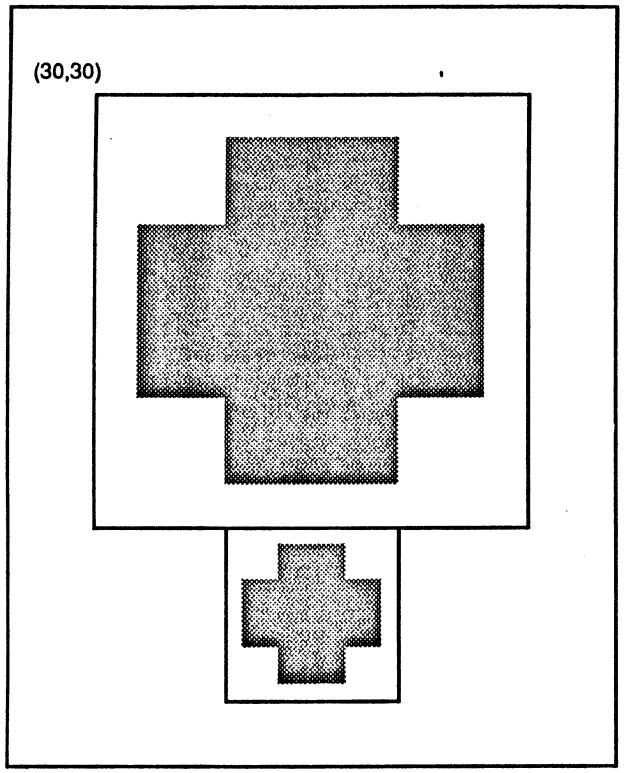


Figure III.28. La Croix-Rouge (imaginez le rouge !).

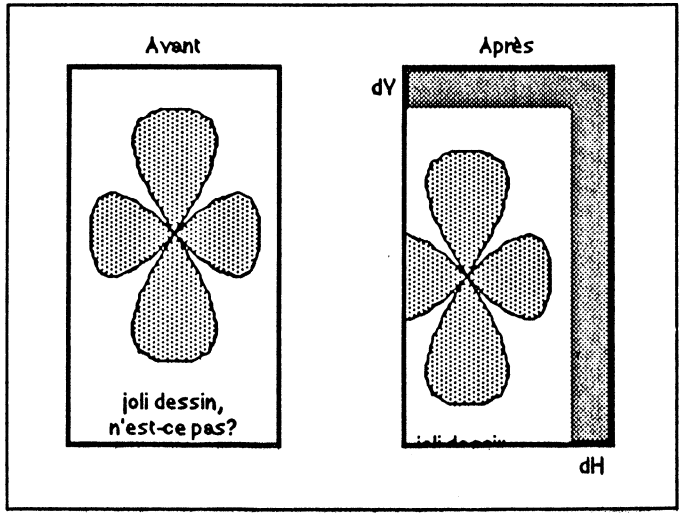


Figure III.29. La procédure ScrollRect.

Transfert dans le même grafport

La procédure **ScrollRect** permet de translater tous les pixels d'un rectangle d'une distance horizontale et verticale à définir. Exemple d'utilisation :

```
Rect    r;                          /* un rectangle */
int     dH, dV;                     /* deux entiers */
Handle  upd;                        /* handle sur région */

upd = NewRgn( );                    /* allocation d'un handle sur région */
GetPortRect(&r);                    /* le rectangle sera le PortRect du grafport */
dH = -10;                           /* déplacement de 10 vers la gauche */
dV = 10;                            /* déplacement de 10 vers le bas */
ScrollRect(&r, dH, dV, upd);        /* on effectue le déplacement */
DisposeRgn(upd);                    /* on n'a plus besoin de la région */
```

La région désignée par le handle upd est celle qui va apparaître après le déplacement des pixels. Cette région, gérée de manière interne, sera remplie par le pattern de fond du grafport *(bkPat)*. Les déplacements positifs s'effectuent vers la droite et vers le bas. Les pixels qui sortent du rectangle sont perdus. Notons que le transfert est limité à la région habituelle de dessin (intersection des deux rectangles et des deux régions qui limitent l'action de dessiner).

On verra en fin de chapitre XI un exemple concret d'utilisation de cette procédure.

Transfert sans référence à un grafport

La procédure **PaintPixels** transfère une pixel map d'une structure *LocInfo* à une autre structure *LocInfo*, sans référence aucune au grafport courant. Le mode de transfert est choisi dans l'appel de la procédure (les huit modes « normaux » sont permis). Deux rectangles sont également à définir : un rectangle dans la structure d'origine, qui délimite la partie de la pixel image à transférer, et un rectangle dans la structure réceptrice : le rectangle d'arrivée n'est pas de la même taille que le rectangle d'origine, il y aura mise à l'échelle de l'image transférée (avec des résultats plus ou moins bons visuellement). Le transfert peut être limité à une partie seulement de la pixel map par définition d'une région masque (une clip region particulière) dans la structure réceptrice.

PaintPixels n'admet qu'un argument : un pointeur sur une structure qui contient en fait les véritables arguments de la procédure. Cette structure contient les six éléments suivants :

- un pointeur sur la structure *LocInfo* d'origine ;
- un pointeur sur la structure *LocInfo* réceptrice ;
- un pointeur sur le rectangle source ;
- un pointeur sur le rectangle de destination ;
- le mode de transfert (dans un entier sur 16 bits) ;
- un handle sur la région masque.

Rappelons que le rectangle frontière d'une structure *LocInfo* définit un système de coordonnées. Le rectangle source sera défini dans le système de la *LocInfo* d'origine, le rectangle de destination et la région masque seront définis dans le système de la *LocInfo* réceptrice.

Pour ne pas utiliser de région masque, mettre zéro-long en guise de handle.

Transfert vers le grafport courant

Dans certaines applications, on n'a pas forcément envie d'exécuter ses dessins directement à l'écran : on peut préférer dessiner en dehors de l'écran, donc de manière invisible, et faire apparaître la totalité du dessin d'un coup. Dans les animations graphiques, par exemple, on prépare l'image suivante hors écran, et elle vient remplacer la précédente en une copie instantanée vers la mémoire écran.

Pour copier une pixel map dans le grafport courant, on utilisera la procédure **PPToPort**. L'avantage de cette procédure sur **PaintPixels** est qu'elle respecte la clip region et la région visible du grafport courant, autrement dit qu'elle ne risque pas de dessiner dans la partie invisible d'une fenêtre, par exemple.

PPToPort utilise cinq arguments :

- un pointeur sur la structure *LocInfo* d'origine ;
- un pointeur sur le rectangle source ;
- l'abscisse du coin supérieur gauche du rectangle destination ;
- l'ordonnée du coin supérieur gauche du rectangle destination ;
- le mode de transfert (entier sur 16 bits).

On constate que le rectangle de destination n'est pas précis : seul son coin supérieur gauche doit être fourni. Cela signifie que contrairement à **PaintPixels**, **PPToPort** ne permet pas la mise à l'échelle des pixels transférés (le rectangle destination a la même taille que le rectangle origine). De même, pas de région masque, puisque nous savons qu'implicitement, l'intersection de la clip region et de la région visible servira de masque.

A ces différences près, **PPToPort** et **PaintPixels** fonctionnent de manière identique.

COULEUR D'UN PIXEL

Il est possible de connaître (mais indirectement) la couleur du pixel associé à un point déterminé, grâce à la fonction **GetPixel**. On passe en argument l'abscisse et l'ordonnée du point (coordonnées locales), et la fonction retourne le contenu définissant le pixel (dans les deux ou quatre bits bas de l'entier résultant). L'exemple suivant utilise **GetMouse**, une procédure de l'Event Manager qui permet de récupérer, justement en coordonnées locales, le point où se trouve la souris.

```
int     num, scb, col;
Point   pt;

GetMouse(&pt);                          /* voir l'Event Manager */
num = GetPixel(pt.H, pt.V);             /* entrée dans la table de couleurs utilisée */
LocalToGlobal(&pt);                     /* le point est traduit en coordonnées globales */
scb = GetSCB(pt.V);                     /* le SCB dont il dépend */
col = GetColorEntry(scb & 0x000F, num & 0x000F); /* la couleur exacte */
```

Dans l'exemple précédent, un petit exercice de style : nous sommes en mode 320 et nous souhaitons connaître la couleur exacte d'un pixel (et pas seulement son numéro d'entrée dans la table des couleurs dont il dépend). Pour ce faire, on traduit ses coordonnées locales en coordonnées globales, ce qui nous permet de connaître la ligne d'écran à laquelle il appartient, donc le SCB dont il dépend, donc la table de couleurs associée. Connaissant l'entrée dans la table de couleurs, on en déduit facilement la couleur exacte (en niveaux de rouge, de vert et de bleu).

GESTION DU CURSEUR

Nous avons vu plus haut comment une application pouvait définir un curseur. Le curseur sera géré par l'intermédiaire d'un pointeur sur la structure Cursor (ou par une chaîne d'entiers sur 8 bits non structurée).

- Au début de l'application, on commence par initialiser le curseur avec la procédure **InitCursor**. Un curseur en forme de flèche (toute noire) devient visible et suit les mouvements de la souris.
- Pour modifier le dessin du curseur utilisé, on emploiera la procédure **SetCursor**, l'argument désignant le nouveau curseur (par son adresse).
- Pour connaître l'adresse de la description du curseur en cours d'utilisation, on se servira de la fonction **GetCursorAdr**.

Dans l'exemple qui suit, on change momentanément de curseur, puis on rétablit celui d'origine.

```
Pointer arrow;                          /* pointeur sur curseur */
Cursor  croix = {
    5,                                  /* hauteur de l'image (nombre de lignes) */
    3,                                  /* largeur de l'image (nombre de mots, donc 6 octets) */
    {                                               /* image du curseur */
    { 0x00,0xF0,0x00,0x00,0x00,0x00 },
    { 0x00,0xF0,0x00,0x00,0x00,0x00 },
    { 0xFF,0xFF,0xF0,0x00,0x00,0x00 },
    { 0x00,0xF0,0x00,0x00,0x00,0x00 },
    { 0x00,0xF0,0x00,0x00,0x00,0x00 }
    },                                                  /* image du masque */
    {
    { 0, 0, 0, 0, 0, 0 },
    { 0, 0, 0, 0, 0, 0 },
    { 0, 0, 0, 0, 0, 0 },
    { 0, 0, 0, 0, 0, 0 },
    { 0, 0, 0, 0, 0, 0 }
    },
    2, 2                                /* coordonnées du point chaud */
    };

InitCursor( );                          /* curseur en forme de flèche */
arrow = GetCursorAdr( );                /* on garde son adresse */
SetCursor(&croix);                      /* curseur en forme de croix */
...
SetCursor(arrow);                       /* on rétablit la flèche */
```

- Tout comme le crayon, le curseur possède un niveau d'invisibilité. Le niveau zéro signifie que le curseur est visible (**InitCursor** initialise le niveau à zéro). La procédure **HideCursor** décrémente le niveau d'une unité, tandis que **ShowCursor** l'incrémente. Ces deux procédures devraient toujours être utilisées conjointement, comme une accolade ouvrante et une accolade fermante.

```
HideCursor( );                          /* normalement, le curseur devient invisible */
...
ShowCursor( );                          /* normalement, le curseur redevient visible */
```

- La procédure **ObscureCursor** (sans argument) rend le curseur invisible jusqu'au prochain mouvement de la souris. Cette procédure est très pratique pour les applications à l'intérieur desquelles l'utilisateur doit entrer du texte par l'intermédiaire du clavier (éditeurs de textes, traitements de textes, etc.) : dès qu'il commence à taper, le curseur disparaît, et réapparaît dès qu'il touche à la souris.

CHAPITRE IV

EVENT MANAGER

PRINCIPES GÉNÉRAUX

Ceux qui n'ont jamais programmé sur Macintosh seront peut-être surpris de la logique de programmation sur l'Apple IIGS, qui est absolument identique : une application est pilotée par des événements (*event driven,* comme disent les américains), ce qui se traduit par une structure de programme très particulière. Le cœur d'une application est une boucle qui attend sans cesse le prochain événement, et qui appelle des procédures adaptées au traitement de chacun de ces événements.

Fini les programmes linéaires qui s'organisent en menus successifs et hiérarchiques : la boucle d'événements n'impose aucune hiérarchie dans les demandes de l'utilisateur, l'application doit être prête à tout instant à répondre à n'importe quelle sollicitation extérieure.

Mais qu'entend-on au juste par événement ? Quand l'utilisateur enfonce le bouton de la souris ou une touche du clavier, il déclenche un mécanisme qui permet à l'Event Manager, le gestionnaire des événements, de détecter une telle action et d'en déterminer les caractéristiques, qu'il pourra transmettre à l'application pour que celle-ci puisse y répondre.

Les événements sont nombreux et variés, ils ne sont pas tous le fait de l'utilisateur. Dans ce chapitre, nous étudierons les principaux types d'événements, et la manière dont l'application devrait en tenir compte.

UTILISATION DE L'EVENT MANAGER

Divers types d'événements

- Événements de type souris

Il y a deux types d'événements liés à la souris. Quand le bouton de la souris est enfoncé, un événement de type *MouseDown* est généré ; quand le bouton de la souris est relâché, on a un événement de type *MouseUp*. La simple action de faire glisser la souris pour déplacer le pointeur à l'écran ne constitue pas un événement. Seule

l'action sur le bouton en génère, et les génère par paires, sauf cas particuliers (le Window Manager par exemple empêche l'événement *MouseUp* d'être rapporté à l'application quand le bouton de la souris est relâché après avoir été enfoncé dans un contrôle de fenêtre).

Plusieurs informations sont liées à un événement de type souris :

- le lieu où se trouvait le pointeur quand l'événement s'est produit ;
- la date relative à laquelle l'événement s'est produit ;
- les touches de modification enfoncées à ce moment précis ;
- le numéro du bouton incriminé (il n'y a qu'un bouton sur la souris, mais d'autres matériels sont utilisables, et l'Event Manager sait gérer deux boutons au maximum).

• Événements de type clavier

Il y a deux types d'événements liés au clavier. Quand une touche normale est enfoncée, un événement de type *KeyDown* est généré. Par touche normale, nous entendons toutes les touches du clavier (normal ou numérique) à l'exception des cinq touches qui modifient son comportement : Majuscule, Blocage-Majuscule, Contrôle, Option et Pomme (Commande ou Pomme ouverte, c'est la même chose).

Attention L'enfoncement d'une combinaison quelconque des touches de modification sans l'enfoncement simultané d'une touche normale ne génère aucun événement.

Quand une touche normale est gardée enfoncée, des événements de type *AutoKey* sont générés périodiquement. Le délai de répétition et la vitesse de répétition sont des paramètres que l'utilisateur peut régler grâce à l'accessoire de bureau Tableau de Bord.

Notons qu'aucun événement n'est généré quand la touche est relâchée.

Plusieurs informations sont liées à un événement de type clavier :

- la date relative à laquelle l'événement s'est produit ;
- les touches de modification enfoncées à ce moment précis ;
- le code ASCII du caractère correspondant à la touche enfoncée (les touches de modification ayant une influence sur cette information sont prises en compte).

• Événements de type fenêtre

Deux types d'événements sont générés par le Window Manager : les événements d'activation et de désactivation des fenêtres *(activate event)*, et les événements de mise à jour *(update event)*. Ces types d'événements sont étudiés en détail dans le chapitre V

• Autres types d'événements

Plusieurs types d'événements complètent la panoplie de l'Event Manager. Certains pourront être ignorés dans une première approche.

- *device driver event :* certains contrôleurs de périphériques peuvent générer des événements ;
- *application events :* une application peut définir pour ses besoins propres jusqu'à quatre types d'événements. A elle de les gérer complètement ;
- *switch event :* le bouton de la souris a été enfoncé dans le contrôle de commutation d'applications. Cet événement est généré par le Control Manager : l'utilisateur souhaite changer d'application courante.

Remarque Au moment où nous écrivons, nous ne savons pas si cet événement sera un jour généré par le Control Manager. Il faut donc considérer tout ce qui est dit par la suite concernant ce type d'événement comme théorique.

- *desk accessory event :* une combinaison particulière de touches (Contrôle - Pomme - Escape) a été enfoncée, invoquant l'utilisation des accessoires de bureau classiques (voir le chapitre X).

• Événement nul

Quand l'Event Manager n'a pas d'autre événement à rapporter, il génère un événement factice, dit *null event*. Cet événement nul apporte plusieurs informations, qui peuvent servir dans certains cas précis :

- le lieu où se trouvait le pointeur quand l'événement s'est produit ;
- la date relative à laquelle l'événement s'est produit ;
- les touches de modification enfoncées à ce moment précis.

Priorité d'événements

Au fur et à mesure de leur génération, la plupart des événements sont stockés dans une file d'événements *(event queue)*, dans un ordre chronologique. C'est notamment dans cette file que l'application ira chercher les événements qu'elle aura à traiter, éventuellement en sélectionnant certains d'entre eux.

Tous les types d'événements n'ont pas le même niveau de priorité. L'ordre est le suivant :

1. *activate event* (désactivation puis activation) ;
2. *switch event ;*
3. événements de type souris, de type clavier, *device driver event, application events, desk accessory event* (dans l'ordre chronologique) ;
4. *update event.*

Les événements d'activation de fenêtres sont tellement prioritaires qu'ils n'entrent même pas dans la file d'événements. Ils sont détectés par un mécanisme spécial du Window Manager et pris en compte immédiatement.

De même, le *switch* n'est pas posté dans la file et est pris en compte dès qu'il n'y a plus d'événement d'activation en cours. Il rend alors prioritaire les éventuels événements de mise à jour de fenêtre, tous exécutés avant que la commutation d'application n'intervienne.

Les événements de la catégorie 3 sont les seuls à entrer réellement dans la file. Ils sont pris en compte dans l'ordre où ils y sont entrés (premier entré, premier sorti), donc dans l'ordre chronologique de leur génération.

Les événements de mise à jour sont traités en dernier, quand aucun événement plus prioritaire n'est en attente. Comme les événements d'activation, ils sont détectés par un mécanisme spécial du Window Manager et n'entrent pas dans la file.

Quand il n'y a plus aucun événement en suspens, l'Event Manager retourne un événement nul, signifiant à l'application qu'elle n'a rien de spécial à faire.

Remarque La file d'événements a une taille limitée, déterminée au moment de l'initialisation de l'Event Manager. Que se passe-t-il quand cette file est pleine, s'il rentre un événement supplémentaire ? Dans ce cas, c'est l'événement le plus ancien qui est perdu corps et bien. Il n'aura jamais été traité.

Structure d' événements et définition de constantes

Les événements sont manipulés au travers d'une structure particulière appelée *Event record*, dont voici les composantes :

```
int     what;          /* code de l'événement */
long    message;       /* contenu variable en fonction du code événement */
long    when;          /* date relative de l'événement */
long    where;         /* endroit où la souris pointait lors de l'événement */
int     modifiers;     /* quelles touches de modification étaient enfoncées */
```

A cette structure, suffisante pour la seule utilisation de l'Event Manager, nous préférerons immédiatement celle de *Task record,* indispensable dès que l'application manipulera des menus déroulants ou battra au rythme cardiaque de la fonction **TaskMaster**. En voici la définition :

```
struct _TaskRec {
    int     what;          /* code de l'événement */
    long    message;       /* contenu variable en fonction du code événement */
    long    when;          /* date relative de l'événement */
    long    where;         /* endroit où la souris pointait lors de l'événement */
    int     modifiers;     /* quelles touches de modification étaient enfoncées */
    long    TaskData;      /* données additionnelles */
    long    TaskMask;      /* masque pour l'utilisation de TaskMaster */
    };
#define TaskRec struct _TaskRec
```

Remarquons dans cette structure que nous avons défini le champ *where* comme un entier long plutôt que comme un point. Il faudra en tenir compte dans les manipulations futures.

• Le code de l'événement (champ *what*) est prédéfini par les constantes suivantes :

```
#define NullEvent       0    /* événement nul */
#define MouseDown       1    /* bouton de la souris enfoncé */
#define MouseUp         2    /* bouton de la souris relâché */
#define KeyDown         3    /* touche du clavier enfoncée */
#define AutoKey         5    /* répétition de touche */
#define UpdateEvt       6    /* mise à jour de fenêtre */
#define ActivateEvt     8    /* activation ou désactivation de fenêtre */
#define SwitchEvt       9    /* commutation d'application */
#define DeskAccEvt      10   /* accessoire de bureau classique */
#define DriverEvt       11   /* périphérique */
#define app1Evt         12   /* défini par l'application */
#define app2Evt         13   /* défini par l'application */
#define app3Evt         14   /* défini par l'application */
#define app4Evt         15   /* défini par l'application */
```

• Le champ *message* contient de l'information additionnelle sur l'événement :

– numéro du bouton (0 à 1) dans le mot bas pour les événements de type souris ;

– code ASCII du caractère dans l'octet bas pour les événements de type clavier (le bit le plus significatif de cet octet étant toujours à zéro, c'est l'application qui devra le forcer à un pour utiliser le jeu de caractères optionnels, voir l'exemple en fin de chapitre) ;

– pointeur sur la fenêtre pour les événements de type fenêtre ;

– défini par le contrôleur pour les événements de périphériques ;

– défini par l'application pour les événements propres à l'application ;

– non défini dans les autres cas.

• Le champ *when* contient une notion de temps que nous avons appelée date relative. Il s'agit très exactement du nombre de ticks (cinquantièmes de secondes quand le système tourne à 50 hertz) écoulés depuis le démarrage du système au moment où l'événement est survenu. Cette notion suffit pour calculer un intervalle de temps entre deux événements, par exemple pour déterminer un double-clic.

• Le champ *where* contient les coordonnées globales du point où se trouvait le pointeur quand l'événement a eu lieu. Nous l'avons déclaré comme un entier long, nous trouverons donc l'abscisse dans le mot haut et l'ordonnée dans le mot bas. Pour isoler chacune des composantes, nous utiliserons les opérateurs de décalage qu'offre le langage C (par exemple au travers de la fonction getbits, dont la définition est donnée à la fin de ce chapitre).

• Le champ *modifiers* est à manipuler au niveau du bit : chaque bit a une signification précise suivant sa valeur.

– bit 0 : si l'événement fenêtre est une activation, ce bit est à 1, et à 0 s'il s'agit d'une désactivation ;

– bit 1 : si la fenêtre active change de type (application <–> système), ce bit est à 1, sinon à 0 ;

– bit 6 : si le bouton n° 1 est enfoncé, ce bit est à 0, sinon à 1 ;

– bit 7 : si le bouton n° 0 est enfoncé, ce bit est à 0, sinon à 1 ;

– bit 8 : si la touche Pomme est enfoncée, ce bit est à 1, sinon à 0 ;

– bit 9 : si la touche Majuscule est enfoncée, ce bit est à 1, sinon à 0 ;

– bit 10 : si la touche Blocage-Majuscule est enfoncée, ce bit est à 1, sinon à 0 ;

– bit 11 : si la touche Option est enfoncée, ce bit est à 1, sinon à 0 ;

– bit 12 : si la touche Contrôle est enfoncée, ce bit est à 1, sinon à 0 ;

– bit 13 : si la touche normale enfoncée appartient au clavier numérique, ce bit est à 1, sinon à 0.

Notons que le bouton unique de la souris porte le numéro 0.

Nous définirons un masque pour chaque bit, de façon à rendre plus lisibles les programmes qui iront les tester :

```
#define ActiveFlag      0x0001
#define ChangeFlag      0x0002
#define Btn1State       0x0040
#define Btn0State       0x0080
#define AppleKey        0x0100
#define ShiftKey        0x0200
#define CapsLock        0x0400
#define OptionKey       0x0800
#define ControlKey      0x1000
#define KeyPad          0x2000
```

Ainsi, pour tester la valeur du bit indiquant si la touche Pomme est enfoncée, on testera la valeur

```
tache.modifiers & AppleKey
```

Si cette valeur est vraie (non nulle), le bit 8 est à 1, donc la touche Pomme était enfoncée. De même, on pourra tester plusieurs bits à la fois :

```
tache.modifiers & (CapsLock + ShiftKey)
```

Si cette expression est vraie, c'est qu'au moins l'une des deux touches Blocage-Majuscule ou Majuscule était enfoncée quand l'Event Manager a rapporté l'événement.

Masques d'événements

On peut demander à certaines routines de l'Event Manager d'opérer sur un ensemble restreint de types d'événements, au lieu d'opérer sur la totalité. Pour ce faire, on définira un masque d'événements qui servira d'argument à ces routines.

Le masque est un entier sur 16 bits, chaque bit étant en correspondance avec le code événement vu plus haut. Si le bit est à 1, l'événement correspondant est traité, et non retenu si le bit est à 0. Si tous les bits sont à 1, le masque ne réduit pas l'ensemble des types d'événements. Un tel masque non sélectif sera codé − 1, en jouant sur la représentation des nombres entiers en mémoire.

Note L'événement nul ne peut être masqué.

Pour rendre plus lisible une application qui utilise des masques d'événements, nous pouvons définir les constantes suivantes :

```
#define mDownMask      0x0002
#define mUpMask        0x0004
#define KeyDownMask    0x0008
#define AutoKeyMask    0x0020
#define UpdateMask     0x0040
#define ActivMask      0x0100
#define SwitchMask     0x0200
#define DeskAccMask    0x0400
#define DriverMask     0x0800
#define app1Mask       0x1000
#define app2Mask       0x2000
#define app3Mask       0x4000
#define app4Mask       0x8000
#define EveryEvent     0xFFFF      /* équivaut à -1 */
```

De la sorte, pour se restreindre aux événements de type souris, le masque s'écrira :

```
mDownMask + mUpMask
```

De même, pour utiliser tous les types d'événements sauf les événements de type clavier, le masque s'écrira :

```
EveryEvent - (KeyDownMask + AutoKeyMask)
```

Absolument n'importe qui est capable de lire ces expressions, alors que leur équivalent hexadécimal serait plutôt délicat à interpréter.

Généralités sur les routines

Quand une application utilise à la fois l'Event Manager et le Window Manager (ce qui devrait toujours être le cas), elle doit initialiser l'Event Manager avant le Window Manager. Les deux gestionnaires se partagent la même page zéro en mémoire, puisqu'ils doivent se partager des données (les événements de type fenêtre). Si le Window Manager n'est pas initialisé après l'Event Manager, celui-ci assumera qu'aucune fenêtre n'est utilisée, et ne générera aucun événement de type fenêtre ! Le chapitre XII donnera une vision d'ensemble de l'initialisation des outils.

Une fois l'initialisation effectuée grâce à **EMStartUp**, l'application gérera une boucle au cœur de laquelle la fonction **GetNextEvent** ira retrouver l'événement suivant à prendre en compte (en tenant compte des diverses priorités vues précédemment). L'application réagira en fonction du type d'événement retourné :

- Bouton de la souris enfoncé : l'application appellera la fonction **FindWindow** du Window Manager pour apprendre dans quelle partie de l'écran le pointeur se trouvait lors de l'événement, et en fonction de cette réponse, utilisera les outils appropriés du Menu Manager, du Desk Manager, du Window Manager, du Control Manager ou de Line Edit pour satisfaire l'utilisateur.

Remarquons que jusqu'à présent nous n'avons pas parlé de double-clic. Le double-clic n'est en effet pas un événement en tant que tel, mais doit être détecté comme la succession de deux événements particuliers : un bouton relâché suivi très vite d'un bouton enfoncé. Nous verrons sur un exemple comment détecter un double-clic.

Si les combinaisons de type Majuscule-clic doivent être gérées par l'application (par exemple pour opérer une sélection multiple disjointe comme dans le finder), il faudra prendre en compte le contenu du champ *modifiers* de l'Event record.

- Événement de type clavier : il faut immédiatement vérifier dans le champ *modifiers* si la touche Pomme était enfoncée, auquel cas l'utilisateur a invoqué un article de menu par l'intermédiaire d'un de ses deux équivalents-clavier.

Si ce n'est pas une commande de menu, l'application gère le caractère reçu : insertion dans le texte de la fenêtre active s'il y a lieu, ou ignorance pure et simple de l'événement.

- Événement de type fenêtre : appel immédiat au Window Manager pour utilisation des procédures adéquates.
- *desk accessory event* : l'application n'aura jamais à traiter ce type d'événement, car il est intercepté et pris en compte automatiquement par le Desk Manager.
- *switch event* : quand l'application reçoit un événement de commutation d'application, elle doit appeler une routine (inconnue à ce jour) pour sauvegarder son propre état courant et passer la main à l'autre application.

Note Au lieu d'utiliser la fonction **GetNextEvent** pour gérer la boucle d'événements, il sera souvent préférable d'utiliser la fonction **TaskMaster** qui évite pas mal de lignes de programmation quand l'application s'y prête. Nous verrons comment utiliser cette fonction dans le chapitre XI, qui lui est consacré, une fois que la plupart des concepts qu'elle gère auront été introduits.

EXEMPLES D'UTILISATION

Boucle d'événements

```
int         indic = TRUE;                  /* indicateur de fin d'application */
TaskRec     tache;                         /* événement courant */

...                                        /* début de l'application: initialisations diverses */
EMStartUp(pgzero, 20, 0, maxX, 0, 200, myID);   /* initialisation de l'Event Manager */
...                                        /* suite des initialisations */
...
FlushEvents(EveryEvent, 0);                /* ménage dans la file d'événements */
...
do                                         /* début de la boucle d'événements */
{
SystemTask( );                             /* nécessaire aux accessoires de bureau */
if (!GetNextEvent(EveryEvent, &tache)) continue;  /* aucun événement à gérer */
switch (tache.what)                                /* quel événement à gérer? */
    {
    case MouseDown :                       /* bouton de la souris enfoncé */
    ...                                    /* appel de la fonction FindWindow du Window Manager */
    break;

    case KeyDown :                         /* touche du clavier enfoncée */
    case AutoKey :                         /* répétition touche du clavier */
    if (tache.modifiers & AppleKey)        /* la touche Pomme était-elle enfoncée? */
```

```
    ...                         /* oui, on appelle la fonction MenuKey du Menu Manager */
    else
    ...                         /* l'application traite le caractère ASCII reçu */
    break;

    case UpdateEvt:                         /* mise à jour d'une fenêtre */
    ...         /* appel de fonctions du Window Manager pour en redessiner le contenu */
    break;

    case ActivateEvt:                       /* événement d'activation */
    if (myEvent.modifiers & ActiveFlag)     /* teste l'indicateur activation/désactivation */
    ...                         /* il est à 1: la fenêtre doit être activée */
    else
    ...                         /* il est à 0: la fenêtre doit être désactivée */
    break;

    case SwitchEvt:                         /* commutation d'application */
    ...                         /* mystère et boule de gomme */
    break;
    }
}
while (indic);                          /* fin de la boucle d'événement */
...
...             /* fin de l'application: fermeture des managers et retour au finder */
```

L'application débute par la déclaration des variables globales, puis le programme principal commence par l'initialisation des divers gestionnaires. L'Event Manager est initialisé par la procédure **EMStartUp**, qui ne réclame pas moins de sept arguments.

Le premier désigne une frontière de page dans la banque 0, l'Event Manager ayant besoin d'une page complète dans la banque 0 pour pouvoir fonctionner, page qu'il partage d'ailleurs avec le Window Manager.

Le deuxième argument désigne le nombre maximal d'événements que la file d'événements peut gérer : quand on donne la valeur 0, une taille par défaut de 20 est utilisée.

Les arguments trois à six précisent quelle est l'aire d'action de la souris. Seul le quatrième argument est sujet à variation : maxX doit prendre la valeur 320 pour une utilisation du mode super hi-res 320 pixels par ligne, et 640 pour une utilisation du mode super hi-res 640 pixels par ligne.

Le septième est le numéro identifiant l'application (tel qu'il est retourné par la fonction **MMStartUp** du Memory Manager).

On aura dans le chapitre XII une vision d'ensemble de l'initialisation des outils.

Avant de commencer à scruter les événements, il est toujours bon de faire du ménage pour éliminer tout événement parasite qui aurait pu survenir pendant les initialisations. La fonction **FlushEvents** est là pour cela. Ses deux arguments sont des masques d'événements. Le premier indique quels types d'événements vont être retirés de la file (− 1 pour tous les événements), le second définit un point d'arrêt : on détruit tous les événements précédemment définis jusqu'à ce qu'on en rencontre un correspondant au second masque (0 signifie pas de masque : on s'arrête quand il n'y a plus rien). Dans l'exemple ci-dessus, on supprime tous les événements qui pourraient se trouver dans la file avant le début réel de l'application.

Pour supprimer tous les événements de type clavier jusqu'au premier clic souris, on écrirait :

```
type = FlushEvents(KeyDownMask + AutoKeyMask, mDownMask);
```


et la fonction renverrait la valeur 1 *(MouseDown)*, signifiant qu'un événement de code 1 (bouton enfoncé) a été rencontré et a stoppé le processus, ou 0 signifiant que tous les événements ont été supprimés.

La file d'événements étant débarrassée de ses parasites, on peut démarrer la boucle d'événements. La fonction **GetNextEvent** est appelée périodiquement. Elle va nous retourner l'événement suivant, en respectant les priorités vues plus haut. Si l'événement se trouvait dans la file, il en est retiré. La fonction admet deux arguments : le premier est un masque précisant quels types d'événements l'application désire recevoir (les autres types resteront dans la file). Le second donne l'adresse d'une zone de stockage de type *TaskRec* (plutôt qu'un *Event record*, comme nous l'avons déjà dit), où la fonction va écrire les caractéristiques de l'événement à traiter.

GetNextEvent commence par appeler la fonction **SystemEvent** du Desk Manager, pour voir si le système ne veut pas intercepter et prendre en charge l'événement suivant. Dans ce cas, ou bien si l'événement renvoyé est l'événement nul, la fonction renvoie la valeur booléenne FALSE (l'application n'a rien à faire, sinon à aller chercher l'événement suivant). Si au contraire la valeur TRUE est retournée, l'application doit répondre à l'événement.

Remarque Le Desk Manager intercepte les événements suivants :

- *desk accessory events* ;
- *activate events* et *update events* concernant une fenêtre d'accessoire de bureau ;
- événements clavier et bouton relâché si la fenêtre active appartient à un accessoire de bureau.

Ce qui signifie qu'une application gérant les accessoires de bureau nouvelle norme aura très peu de travail à assurer (tout au plus l'appel aux menus déroulants et les actions périodiques), et que toute application saura gérer les accessoires de bureau classiques, sans rien avoir à faire. Voir le chapitre X pour plus de détails sur les accessoires de bureau.

Notons dans la boucle d'événements la présence de la procédure **SystemTask** du Desk Manager, destinée à gérer les accessoires de bureau nouvelle norme pour lesquels une action périodique est à entreprendre, sans que cette action soit liée à un événement. Par exemple, l'horloge a besoin d'être remise à l'heure toutes les secondes, même si sa fenêtre n'est pas la fenêtre active (fenêtre au premier plan).

Dans certains cas très précis, on peut vouloir savoir si tel type d'événement est disponible dans la file d'attente. La fonction **EventAvail** est faite pour cela. Elle utilise les mêmes arguments que **GetNextEvent**, elle retourne la même valeur que **GetNextEvent**, la seule différence est qu'elle ne supprime pas l'événement de la file.

Supposons qu'on veuille savoir si un événement de mise à jour est en attente. On emploiera l'instruction suivante :

```
flag = EventAvail(UpdateMask, &tache);
```

Si flag prend la valeur TRUE, tache désigne le premier événement de mise à jour en attente. Si flag est nul, il n'y a pas de tel événement en suspens. Voir dans le chapitre XI une application pratique de cette routine.

Une application peut forcer un événement en le plaçant elle-même dans la file d'événements, grâce à la fonction **PostEvent**. Deux arguments : le type de l'événement à forcer (valeur comprise entre 0 et 15, suivant les constantes prédéfinies vues plus haut), et l'entier long qui remplira le champ *message*. Les autres champs seront déterminés automatiquement (position de la souris, date relative, touches de modification...) La fonction retournera zéro si l'événement est accepté (pas d'erreur). Une exception toutefois : pour un événement de type clavier ou souris, l'entier long doit contenir dans le mot haut la valeur exacte du champ *modifiers*, qui sera recopiée

au bon endroit. On veillera à ne pas faire n'importe quoi avec cette fonction, notamment à ne pas introduire dans la file des événements de type fenêtre, qui n'ont rien à y faire, et pour lesquels le Window Manager propose d'autres routines. Dans la plupart des cas, cette fonction ne servira qu'aux applications qui gèrent leurs propres types d'événements.

Ajuster le dessin du curseur

Quand on a besoin de connaître l'endroit où se trouve le pointeur, soit on récupère le contenu du champ *where* des événements, y compris l'événement nul, soit on appelle la procédure **GetMouse**. On lui donne en argument l'adresse d'une zone de quatre octets, où elle renverra la position de la souris au moment de l'appel. Différence fondamentale entre les deux méthodes : le point dans le champ *where* de l'événement est exprimé en coordonnées globales, tandis que le point renvoyé par **GetMouse** est exprimé dans le système de coordonnées locales du grafport courant (c'est généralement, mais pas obligatoirement, celui associé à la fenêtre active).

Une application intéressante de cette routine (qui évite d'avoir à gérer les événements nuls) : ajuster le dessin du pointeur en fonction de la zone d'écran où il se trouve. Nous verrons un exemple d'utilisation de **GetMouse** dans le chapitre consacré au Window Manager ainsi que deux exemples de fonctions ajustant le dessin en fonction de certaines conditions, à la fin du présent chapitre et à la fin du chapitre V.

Dans les lignes suivantes, les calculs des points pt1 et pt2 (stockés sous forme d'entiers longs) donnent un résultat équivalent.

```
long    pt1, pt2;                   /* deux points */

pt1 = tache.where ;                 /* pt1 est déjà en coordonnées globales */
GetMouse(&pt2);                     /* pt2 est récupéré en coordonnées locales... */
LocalToGlobal(&pt2);                /* ...et traduit en coordonnées globales */
```

Bouton de la souris

• Pour savoir si le bouton de la souris est à un moment précis enfoncé ou non, on appelle la fonction **Button**. Un seul argument, le numéro du bouton (0 ou 1). La fonction renvoie TRUE si le bouton est enfoncé au moment de l'appel, FALSE sinon.

On pourrait imaginer ainsi un bout de programme simulant une partie de flipper (à condition qu'il y ait autre chose que la souris Apple branchée sur le *Front Desk Bus*, un joystick à deux boutons, par exemple) :

```
int  nonfini = TRUE;            /* indicateur de fin */

do
{
if (Button(0))
GaucheHaut( );                  /* bouton 0 enfoncé, donc dessin du flipper gauche en l'air */
else
GaucheBas( );                   /* bouton 0 relâché, donc dessin du flipper gauche au repos */
if (Button(1))
DroiteHaut( );                  /* bouton 1 enfoncé, donc dessin du flipper droit en l'air */
else
DroiteBas( );                   /* bouton 1 relâché, donc dessin du flipper droit au repos */
...     /* modifie la valeur de nonfini si la partie est terminée */
}
while (nonfini);
```


Notons une utilisation particulière de **Button**, pour attendre le prochain clic souris. L'instruction suivante peut être la condition de sortie d'une boucle do, ou tout simplement une boucle vide. Il n'y aura aucune attente si le bouton de la souris est déjà enfoncé quand cette instruction s'exécute.

```
while(!Button(0));
```

• Pour savoir si le bouton de la souris est toujours enfoncé après un événement de type *MouseDown*, on appelle la fonction **StillDown** ou la fonction **WaitMouseUp**. Un seul argument : le numéro du bouton. Ces deux fonctions retournent TRUE si le bouton est enfoncé au moment de l'appel, et si de plus il n'y a aucun événement de type souris pour ce bouton en attente dans la file. Cette condition assure bien que le bouton n'a pas été relâché (puis enfoncé) depuis le dernier événement *MouseDown*. La différence entre les deux ? Si la condition n'est pas remplie, avant de répondre FALSE, **WaitMouseUp** supprimera de la file l'événement de type *MouseUp* qui s'est forcément produit, alors que **StillDown** le conservera. La distinction peut se révéler intéressante en cas de gestion simultanée par l'application d'événements de type *MouseUp* et de double-clics.

C'est l'une de ces fonctions qu'il faut employer pour gérer un événement qui se répète tant que le bouton de la souris est gardé enfoncé, par exemple quand l'application doit tenir compte du glissement de la souris.

L'exemple suivant non seulement illustre, outre la fonction **StillDown**, certaines particularités de QuickDraw, notamment le mode de transfert XOr et le fonctionnement malheureusement limité de **Pt2Rect**, mais encore use largement des conversions entre coordonnées locales et globales, et enfin anticipe certaines fonctions du Window Manager que nous étudierons au chapitre V.

```
/* un clic souris a été enregistré dans la région contenu de la fenêtre active,
dans laquelle on veut dessiner des rectangles, par exemple.
    Tant que l'utilisateur promène la souris à l'intérieur de la fenêtre, on souhaite
voir la silhouette du rectangle se dessiner et se déformer, comme dans GSPaint */

Pointer wind;                   /* pointeur sur fenêtre */
Handle cont;                    /* handle sur sa région contenu */
Rect   r;                       /* un rectangle */
long   pt1, pt2;                /* les deux sommets opposés du rectangle */
long   pt3;                     /* le précédent point courant */

SetPort(wind);                  /* on va dessiner dans la fenêtre wind */
cont = GetContRgn(wind);        /* on récupère un handle sur sa région contenu */
pt1 = tache.where ;             /* lieu où la souris a été enfoncée... */
GlobalToLocal(&pt1);            /* ...traduit en coordonnées locales */
SetRect(&r,0,0,0,0);            /* initialisation d'un rectangle vide, au cas où */
pt3 = pt1;                      /* initialisation du point précédent */
SetPenMode(2);                  /* mode XOr pour le crayon */
SetSolidPenPat(1);              /* pour faire des rectangles gris et blanc */

do {                            /* on fait... */
    GetMouse(&pt2);                 /* recherche du point courant */
    if (!EqualPt(&pt2,&pt3))        /* le point courant a-t-il changé? */
        {
        LocalToGlobal(&pt2);        /* le point courant en coordonnées globales */
        if (PtInRgn(&pt2, cont))    /* est-il encore dans la région contenu de la fenêtre? */
            {
            GlobalToLocal(&pt2);    /* on rétablit les coordonnées locales */
            Pt2Rect(&pt1, &pt3, &r); /* le rectangle précédent */
            FrameRect(&r);          /* on efface son contour, grâce au mode XOr */
            Pt2Rect(&pt1, &pt2, &r); /* le rectangle actuel */
            FrameRect(&r);          /* on dessine son contour */
            pt3 = pt2;              /* le point courant devient l'ancien point */
```

```
        }
    }
}
while (StillDown(0));          /* ...tant que le bouton est gardé enfoncé */
PaintRect(&r);                 /* et on dessine définitivement le rectangle */
```

Gestion du temps, le double-clic

• Pour connaître la durée de certaines actions ou l'ordre dans lequel les événements se produisent, l'Event Manager gère une notion de temps qui n'a rien à voir avec l'horloge intégrée à la machine et qui est alimentée par batterie. Tous les cinquantièmes de secondes, une action se produit dans le système de la machine : l'écran est redessiné, c'est-à-dire qu'un faisceau d'électrons parcourt tout l'écran, de gauche à droite et de haut en bas, traduisant en pixels colorés les groupes de bits constituant la mémoire écran. Quand le faisceau arrive en bas à droite, une interruption se produit (dite *vertical blanking* ou *vertical retrace interrupt*), pour lui permettre de se repositionner en haut à gauche de l'écran. Durant cette interruption, toute action réclamant une périodicité régulière très courte peut avoir lieu. Par exemple la mise à jour de la position de la souris, ou l'incrémentation d'un compteur. La fonction **TickCount** retourne précisément le contenu de ce compteur, qui représente par construction le nombre de cinquantièmes de secondes (encore appelés ticks) écoulés depuis le redémarrage du système (moment où le compteur a été initialisé à zéro).

Attention Le résultat est un entier long. Du fait qu'il existe la possibilité de désactiver momentanément l'interruption VBL, et que cette désactivation peut se prolonger sur plusieurs ticks, ce résultat ne correspond pas forcément à la réalité, puisqu'une incrémentation du compteur peut être sautée de temps en temps. Pour connaître l'heure, il faut procéder autrement (en appelant **ReadAsciiTime**, procédure appartenant à l'ensemble nommé Miscellaneous Tools, voir chapitre X).

Exemple d'application : les résultats d'un benchmark.

```
long    deb, fin;
int     temps;
char    msg[25];

deb = TickCount( );            /* date relative du début du test */
...                            /* le benchmark, par exemple le crible d'Erathostène */
fin = TickCount( );            /* date relative du fin du test */
temps = (int) fin–deb;         /* temps écoulé,
                               exact si l'interruption VBL n'a pas été désactivée */
sprintf(msg, "Résultat: %d ticks", temps);      /* sprintf est une fonction
                                                   de la bibliothèque C standard */
MoveTo(10,30); DrawCString(msg); /* écriture du résultat à l'écran */
```

Autre exemple Création artificielle d'un délai durant lequel l'application ne fait rien.

Rappel Un tick est un soixantième de seconde sur un sytème tournant à 60 hertz, mais un cinquantième de secondes sur un système tournant à 50 hertz.

```
    Delai(ticks)
long ticks;
{
long fin;
    fin = TickCount( ) + ticks;    /* date actuelle plus le nombre de ticks à attendre */
    while (TickCount( ) <= fin);   /* on ne fait rien tant que la date de fin n'est pas atteinte */
}
```


• Quand une application veut faire clignoter quelque chose (par exemple une barre verticale pour localiser un point d'insertion dans du texte) et qu'elle est obligée de gérer ce curseur (Line Edit le gérerait pour elle), elle peut faire appel à la fonction **GetCaretTime**, qui retournera dans un entier long le nombre de ticks correspondant à la période du clignotement, telle qu'elle a été ajustée par l'utilisateur grâce à l'accessoire de bureau Tableau de Bord.

• Quand une application veut gérer les double-clics, un des moyens de faire (ce n'est pas le seul, loin de là !) est de contrôler si un événement de type *MouseDown* a suivi très vite un événement de type *MouseUp*. La notion de « très vite » est déterminée par la fonction **GetDblTime**, qui retourne dans un entier long le nombre de ticks qui doit être considéré comme le délai maximal entre les deux événements pour qu'ils forment un double-clic. Cette valeur est modifiable par l'utilisateur par l'intermédiaire de l'accessoire Tableau de Bord.

```
TaskRec   tache;               /* événement courant */
TaskRec   tachePrec;           /* événement précédent */
int       indic;               /* indicateur de fin de boucle */

FlushEvent(EveryEvent, 0);     /* plus aucun événement en attente */
tachePrec.what = 0;            /* initialisation partielle mais suffisante... */
tachePrec.when = 0;            /* ...de la variable tachePrec */
...
do
{
if (!GetNextEvent(EveryEvent, &tache)) continue;  /* aucun événement à gérer */
switch (myEvent.what)                             /* quel événement à gérer? */
    {
    case MouseDown :                      /* bouton de la souris enfoncé */
    if (tachePrec.what == MouseUp && (tache.when – tachePrec.when)<GetDblTime( ))
                          /* réponse à un double-clic */
    ...
    else
    ...                   /* réponse à un simple-clic */
    break;

                          /* suite des instructions case */
    ...
    }
tachePrec = tache;                        /* mise en mémoire du dernier événement */
}
while (indic);
...
```

Pour gérer le double-clic, il faut comparer deux événements : celui que l'application doit traiter, et celui qu'elle vient juste de traiter. On utilise donc une variable de type événement pour stocker l'événement précédent, en particulier son type et sa date relative. Comme toujours en pareil cas, il est préférable d'initialiser cette variable pour que la première comparaison puisse avoir lieu et donner un résultat faux (pas de double-clic), mais la probabilité d'erreur est t ement infime que ce n'est pas indispensable. On obtiendra un double-clic quand l'événement précédent sera de type *MouseUp*, l'événement en cours de type *MouseDow* et l'intervalle de temps entre les deux inférieur à la valeur retournée par **GetDblTime.**

Les puristes pourront ajouter une condition supplémentaire portant sur le lieu des événements : la souris ne doit pas avoir trop bougé entre le *MouseUp* et le *MouseDown* (voire entre le *MouseDown* actuel et le *MouseDown* précédent) pour qu'il y ait effectivement double-clic. Le « pas trop bougé » peut être vu comme une borne supérieure à la somme des valeurs absolues des différences de coordonnées entre le point de l'événement actuel (x_2,y_2) et le point de l'événement précédent (x_1,y_1) :

$$|x_2 - x_1| + |y_2 - y_1| < 5 \quad \text{par exemple.}$$

Nous n'avons pas tenu compte d'une telle condition dans l'exemple qui suit, voyez à l'usage à quelles aberrations cela peut conduire.

Exemple complet

L'exemple suivant propose une visualisation à l'écran de divers types d'événements. Est affiché en permanence l'état des touches de modification et des boutons (traduction du champ *modifiers*), la position de la souris en coordonnées globales (traduction du champ *where*) et le temps qui s'écoule en secondes (traduction du champ *when*). L'application détecte les événements de type *MouseDown, KeyDown* et *AutoKey*, de même que les double-clics. Pour les événements de type clavier, le caractère invoqué est affiché, ainsi que son code ASCII (en hexadécimal). Si la touche Option était enfoncée, le caractère optionnel (obtenu en forçant à 1 le bit le plus significatif du code ASCII du caractère) est également affiché.

On profitera de cet exemple pour constater que les touches Retour et Entrée renvoient le même code ASCII (13 décimal). Elles sont donc d'un usage parfaitement équivalent, notamment dans l'utilisation des boutons par défaut (voir le Control Manager, chapitre VII, et le Dialog Manager, chapitre IX).

Pour quitter, on appuiera sur la touche Escape (aucune des touches Pomme, Contrôle, Majuscule ou Option ne doit être enfoncée à ce moment-là).

Nous avons également pris en compte les événements de type *DeskAccEvt,* juste pour montrer que l'application les reçoit, alors qu'elle n'a pas à en tenir compte. Essayez pour voir la combinaison de touches Pomme - Contrôle - Escape !

Nous profiterons de cet exemple pour jouer avec deux pointeurs : un pointeur en forme de point d'interrogation quand la touche Blocage-Majuscule est enfoncée, et la flèche traditionnelle dans le cas contraire. Cet exemple n'est pas gratuit. Il est au contraire très élégant pour une application devant gérer des aides permanentes. Si l'utilisateur bloque les majuscules, il se place en mode Assistance : il va cliquer quelque part grâce au point d'interrogation, et l'application affichera un message fonction de la position du pointeur au moment du clic.

On notera deux versions de la fonction AjusteCurs, en fin de listing. La version non retenue était certes extrêmement simple, mais provoquait un scintillement insupportable du curseur. Au contraire, le fait de mémoriser l'état précédent dans une variable statique et de ne provoquer le changement de curseur qu'en cas de nécessité donne un résultat impeccable.

On notera également la présence de la fonction getbits, qui est d'intérêt général en C. Elle permet d'extraire de 1 à 16 bits d'un entier long, et en fait un entier court. C'est souvent plus pratique à utiliser que les bitfields (structures C particulières au niveau du bit), et cela nous a permis de programmer une boucle plutôt que huit instructions quasiment identiques, pour tester le champ *modifiers* de l'événement.

Pour compiler ce programme, il est nécessaire que tous les termes en italique soient définis dans un ou plusieurs fichiers *headers* (ce sont des notions que nous avons préalablement définies), à inclure. De même, les termes en gras représentent les fonctions de la *ToolBox* et doivent également figurer dans un fichier *header*. Ces fichiers sont généralement fournis avec votre environnement de travail. Ils existent évidemment sur la disquette d'accompagnement de cet ouvrage.

La fonction **debut_appl** (initialisation des outils) et la fonction **quitter** (fin de l'application) sont communes à tous les exemples de l'ouvrage. Elles seront listées en exemple dans le chapitre XII, quand tous les concepts auxquels elles font appel seront introduits.

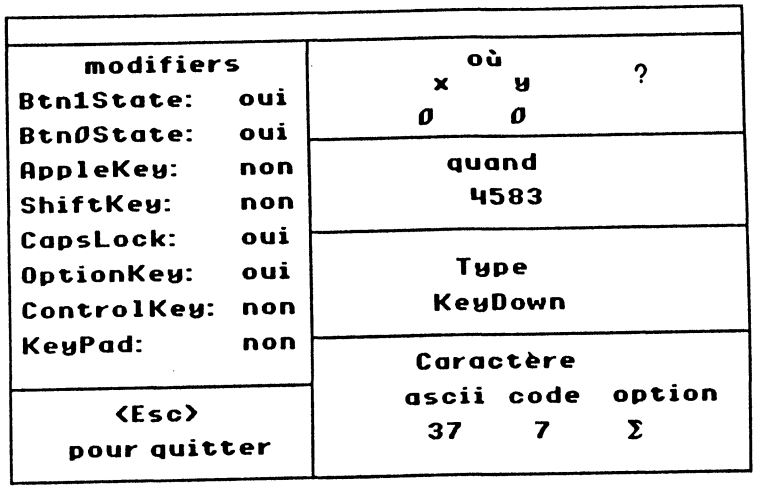


Figure IV. 1. L'écran de l'exemple.

```
#include <tools.h>                  /* contient la définition des termes en gras */
#include <entete.h>                 /* contient la définition des termes en italique */

Cursor intCurs = {                  /* curseur en forme de point d'interrogation */
    9, 3,                           /* 9 lignes, 6 octets par ligne */
    {                               /* image du pointeur */
    {0x00,0xFF,0xF0,0x00,0x00,0x00},
    {0x0F,0x00,0x0F,0x00,0x00,0x00},
    {0x0F,0x00,0x0F,0x00,0x00,0x00},
    {0x00,0x00,0x0F,0x00,0x00,0x00},
    {0x00,0x00,0xF0,0x00,0x00,0x00},
    {0x00,0x0F,0x00,0x00,0x00,0x00},
    {0x00,0x00,0x00,0x00,0x00,0x00},
    {0x00,0x0F,0x00,0x00,0x00,0x00},
    {0x00,0x00,0x00,0x00,0x00,0x00},
    },
    {                               /* masque du pointeur */
    {0x00,0xFF,0xF0,0x00,0x00,0x00},
    {0x0F,0x00,0x0F,0x00,0x00,0x00},
    {0x0F,0x00,0x0F,0x00,0x00,0x00},
    {0x00,0x00,0x0F,0x00,0x00,0x00},
    {0x00,0x00,0xF0,0x00,0x00,0x00},
    {0x00,0x0F,0x00,0x00,0x00,0x00},
    {0x00,0x00,0x00,0x00,0x00,0x00},
    {0x00,0x0F,0x00,0x00,0x00,0x00},
    {0x00,0x00,0x00,0x00,0x00,0x00},
```

```
    },
    7,3                         /* point chaud */
};

TaskRec   tache;                /* ce que manipule GetNextEvent */
Pointer   arrow;                /* l'adresse du curseur système */

            /***** PROGRAMME PRINCIPAL *****/

    main( )
{
TaskRec   tachePrec;            /* l'événement précédent */
int       indic = TRUE;         /* indicateur de fin de boucle */
int       myID;                 /* identifiant de l'application */
char      msg[15];              /* ce qui sera affiché */
int       i;                    /* un compteur */
char      car;                  /* le caractère affiché */

    myID = debut_appl(0);       /* initialisations en mode 320 */
    Desktop(5,0x400000FF);  /* le bureau prend un fond blanc (routine du Window Manager) */
    FlushEvents(EveryEvent,0);  /* la file d'événements est vidée */
    arrow = GetCursorAdr( );    /* adresse du curseur (initialisé dans debut_appl) */
                                /* dessin des éléments fixes à l'écran */

MoveTo(130,13); LineTo(130,200);
MoveTo(35,25); DrawCString("modifiers");
MoveTo(5,40); DrawCString("Btn1State:");
MoveTo(5,55); DrawCString("Btn0State:");
MoveTo(5,70); DrawCString("AppleKey:");
MoveTo(5,85); DrawCString("ShiftKey:");
MoveTo(5,100); DrawCString("CapsLock:");
MoveTo(5,115); DrawCString("OptionKey:");
MoveTo(5,130); DrawCString("ControlKey:");
MoveTo(5,145); DrawCString("KeyPad:");
MoveTo(130,55); LineTo(320,55);
MoveTo(200,25); DrawCString("où");
MoveTo(185,35); DrawCString("x");
MoveTo(220,35); DrawCString("y");
MoveTo(130,95); LineTo(320,95);
MoveTo(190,70); DrawCString("quand");
MoveTo(130,140); LineTo(320,140);
MoveTo(193,115); DrawCString("Type");
MoveTo(175,155); DrawCString("Caractère");
MoveTo(170,170); DrawCString("ascii");
MoveTo(215,170); DrawCString("code");
MoveTo(260,170); DrawCString("option");
MoveTo(0,160); LineTo(130,160);
MoveTo(45,175); DrawCString("<Esc>");
MoveTo(25,190); DrawCString("pour quitter");
                                /* initialisation de l'événement précédent */
tachePrec.what = 0;
tachePrec.when = 0;

do {
    AjusteCurs( );              /* ajuste le dessin du curseur */
    GetNextEvent(EveryEvent, &tache)    /* tous les événements sont acceptés */
    for(i=0; i<8; ++i)          /* traitement des 8 modifiers */
        {
        MoveTo(100, 40+15*i);
        if (getbits((long)tache.modifiers, i+6,1)) DrawCString("oui ");
        else DrawCString("non");
        }
    sprintf(msg,"%d   ",getbits(tache.where, 31,16));       /* abscisse du pointeur */
```



```
MoveTo(178,50); DrawCString(msg);
sprintf(msg,"%d   ",getbits(tache.where, 15,16));       /* ordonnée du pointeur */
MoveTo(218,50); DrawCString(msg);
sprintf(msg,"%ld",tache.when /50);      /* date de l'événement, traduite en secondes */
MoveTo(200,85); DrawCString(msg);

switch(tache.what)              /* quel événement à traiter? */
    {
    case MouseDown :            /* bouton de la souris enfoncé */
        MoveTo(175,130);
        if (tachePrec.what == MouseUp &&
            (tache.when – tachePrec.when) < GetDblTime( ) )
        DrawCString("DoubleClic");          /* double clic */
        else
        DrawCString("MouseDown ");          /* simple clic */
        break;

    case KeyDown :              /* touche clavier enfoncée */
        car = tache.message & 0xFF;         /* code du caractère */
        MoveTo(175,130); DrawCString(" KeyDown  ");
        sprintf(msg,"%x  ",car);
        MoveTo(180,185); DrawCString(msg);
        sprintf(msg," %c  ",car);
        MoveTo(218,185); DrawCString(msg);
        if(tache.modifiers & OptionKey)     /* si la touche est enfoncée... */
            {
            car |= 0x80;            /* ...on force à 1 le bit significatif du code ASCII... */
            sprintf(msg," %c  ",car);       /* ...et on écrit le caractère obtenu */
            MoveTo(258,185); DrawCString(msg);
            }
        if ((car == 0x1B) &         /* $1B = code ASCII du caractère Escape */
        !(tache.modifiers & AppleKey+OptionKey+ShiftKey+ControlKey))
        indic = FALSE;
        break;

    case AutoKey :              /* touche clavier maintenue enfoncée */
        MoveTo(175,130); DrawCString(" AutoKey  ");
        break;

    case DeskAccEvt :           /* retour d'un accessoire de bureau classique */
        MoveTo(175,130); DrawCString(" DeskAcc  ");
        break;
    }
tachePrec = tache;              /* assignation de structures de type TaskRec */

    }
    while(indic);               /* fin de la boucle d'événement */

    quitter(myID);              /* gère la fin de l'application */
}

            /***** FONCTION GETBITS *****/

    getbits(x,p,n)              /* prend n bits à partir de la position p dans un entier long */

unsigned long  x;
unsigned int   p,n;             /* p peut prendre les valeurs 31,30,...,1,0 */
                                /* n doit être compris entre 1 et 16 */
{
    return( (x>>(p+1-n)) & ~(~0<<n) );      /* retourne un entier sur 16 bits */
}
```

```
        /***** FONCTION AJUSTECURS *****/

    AjusteCurs( )               /* ajuste le dessin du pointeur en fonction de l'état de
                                                    la touche Blocage-Majuscule */
{
/* ce qu'aurait pu être cette fonction:

    if(tache.modifiers & CapsLock) SetCursor(&intCurs);
    else SetCursor(arrow);
*/

static  modif;
int     ind;

    ind = tache.modifiers & CapsLock ;      /* état de la touche Blocage-Majuscule */
    if(ind == modif) return;                /* le même qu'avant? On sort! */
    if(ind) SetCursor(&intCurs);        /* touche bloquée —> point d'interrogation */
    else SetCursor(arrow);                  /* touche débloquée —> curseur flèche */
    modif = ind;                            /* nouvel état mémorisé */
}
```

CHAPITRE V

WINDOW MANAGER

PRINCIPES GÉNÉRAUX

Si vous ne vous en étiez pas rendu compte, l'écran de l'Apple IIGS est censé représenter le dessus de votre table de travail *(desktop)*. Au niveau du finder, les icônes représentent des documents rangés dans des dossiers, ce qui est conforme à la réalité des choses. De la même manière que vous n'écrivez ni ne dessinez sur votre bureau (vous utilisez plutôt une feuille de papier), vous n'écrirez ni de dessinerez directement sur l'écran, vous utiliserez des fenêtres.

La fenêtre est le moyen privilégié d'échanger de l'information avec l'utilisateur : toutes ses actions autres que des commandes (dessin, texte, précisions à apporter sur une action, etc.) interviendront dans des fenêtres, tous les messages qu'il recevra (aide, alerte, demande de précisions, etc.) également.

Ces fenêtres seront de type différent suivant leur origine : fenêtres gérées par le système (alertes générales, accessoires de bureau), fenêtres gérées par l'application.

Une fenêtre possède un certain nombre de caractéristiques qui assurent leur aspect universel dans le monde de l'interface utilisateur Apple : une *barre de titre* (facultative) qui contient le titre de la fenêtre et qui sert à la déplacer (en faisant glisser la souris) ; dans cette barre de titre, deux contrôles optionnels : une *case de fermeture* située à gauche qui sert à fermer la fenêtre (par simple clic) et une *case zoom* située à droite qui permet d'agrandir la fenêtre au maximum autorisé ou de revenir à la taille initiale (par simple clic également). Immédiatement sous la barre de titre, peut se trouver une *zone d'information* (facultative) qui présente la particularité d'être fixe même quand l'utilisateur fait défiler le contenu de la fenêtre. Ce défilement s'obtient grâce à des *barres de défilement* (une barre horizontale en bas et une barre verticale à droite, toutes deux optionnelles), constituées chacune d'une *bande de défilement* dans laquelle se déplace un *curseur de défilement,* et de deux *flèches de défilement.* Enfin, en bas à droite, la *case de contrôle de taille* (optionnelle elle aussi) permet de modifier la taille de la fenêtre dans des limites imposées (en faisant glisser la souris).

Toutes les caractéristiques étant optionnelles, la fenêtre la moins compliquée à gérer sera constituée d'un simple rectangle, que l'utilisateur ne pourra ni déplacer, ni agrandir, ni faire défiler. La fenêtre la plus complète ressemblera à celle de la figure V.1. Entre les deux, toutes les combinaisons seront possibles, en se souvenant des contraintes suivantes :

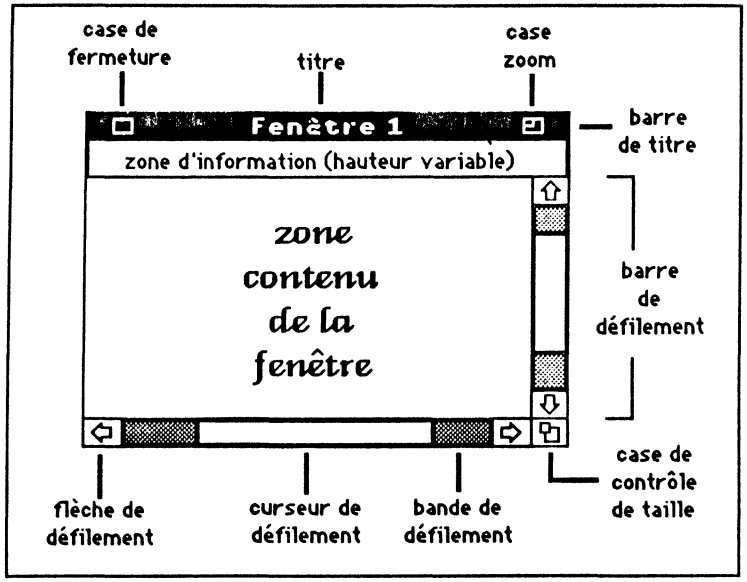


Figure V.1. La fenêtre document standard.

– on ne peut pas avoir de case de fermeture ou de case zoom sans barre de titre ;

– on ne peut pas avoir de case de contrôle de taille sans avoir au moins une barre de défilement ;

– il est généralement ridicule d'avoir une case zoom et pas de case de contrôle de taille.

Il existe une alternative à ce type de fenêtre : la fenêtre d'alerte. Elle est caractérisée par un contour fait d'un double trait, qui permet de déplacer la fenêtre à l'instar de la barre de titre de la fenêtre classique. Une telle fenêtre ne peut posséder aucun autre contrôle : elle sera donc de taille fixe, sans possibilité de défilement du contenu.

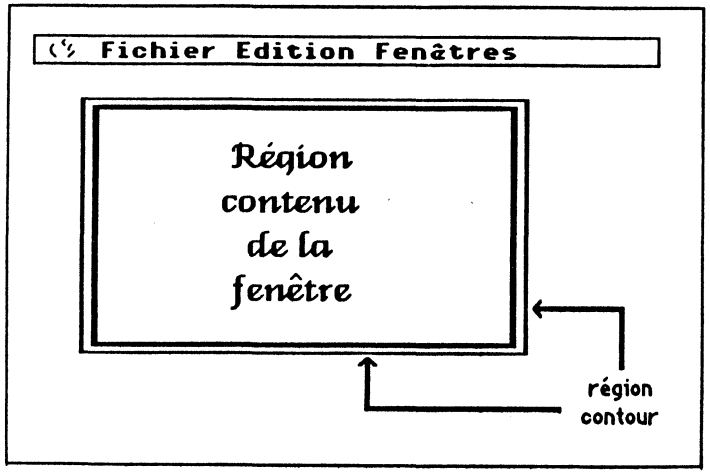


Figure V.2. L'autre type de fenêtre.

Le Window Manager est le gestionnaire des fenêtres. Grâce à lui, nous allons pouvoir créer lesdites fenêtres, gérer leurs différentes caractéristiques et prendre en compte le délicat problème du multifenêtrage, qui assure aux applications une puissance exceptionnelle et à l'utilisateur une liberté accrue.

Le multifenêtrage obéit à des règles générales, l'application devra les suivre scrupuleusement : il ne peut y avoir qu'une fenêtre active à la fois, et il suffit à l'utilisateur de cliquer dans une fenêtre inactive pour la rendre active. Les contrôles dans une fenêtre inactive sont désactivés, de telle sorte qu'il est impossible par exemple de fermer une fenêtre inactive par sa case de fermeture. On pourra toutefois déplacer une fenêtre inactive sans la rendre active en faisant glisser la souris si la touche Pomme est tenue enfoncée durant l'opération.

Pour savoir comment s'effectue le défilement dans la région contenu d'une fenêtre, on se reportera au chapitre XI.

UTILISATION DU WINDOW MANAGER

Qu'est-ce qu'une fenêtre ?

Pour une application, une fenêtre est avant tout un grafport, tel qu'il a été défini dans QuickDraw, avec toutes ses caractéristiques. Ecrire ou dessiner dans une fenêtre relève donc de l'utilisation de QuickDraw.

Une fenêtre, c'est toutefois un peu plus qu'un grafport, puisqu'un certain nombre de contrôles lui sont associés. Ces contrôles, le Window Manager les gère et a la responsabilité de les dessiner. Pour ce faire, il manipule lui-même un grafport à la taille de l'écran, appelé *Window Manager port*. L'écran est donc lui-même une fenêtre, entièrement gérée par le Window Manager. Cette fenêtre ne possède aucun contrôle particulier, mais d'un aspect purement visuel, la barre de menus système peut s'apparenter à une zone d'informations. Voir le chapitre VII sur le Control Manager pour obtenir plus de détails sur les contrôles gérés par le Window Manager et les contrôles gérés par l'application.

On gardera toujours présente à l'esprit la règle suivante : c'est le Window Manager qui dessine les contours d'une fenêtre, c'est l'application qui en dessine le contenu.

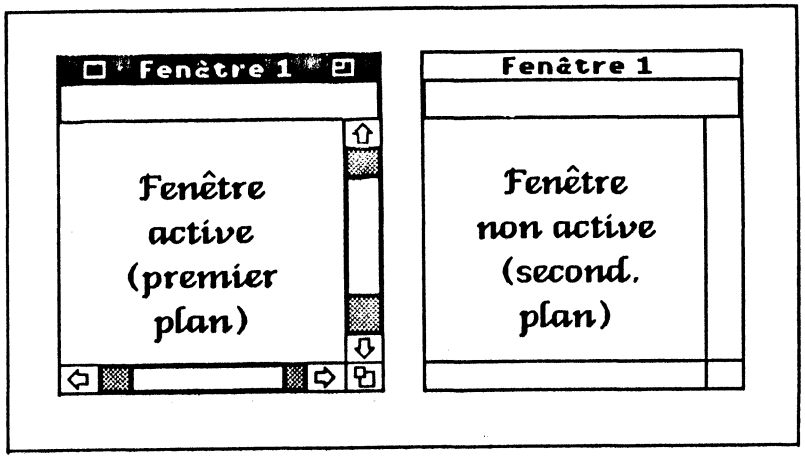


Figure V.3. Fenêtre active, fenêtre non active.

Le Window Manager sait gérer plusieurs fenêtres à la fois, sous la forme d'une liste de fenêtres. Quand plusieurs fenêtres sont présentes à l'écran, on considère que chacune est située dans un plan différent. La fenêtre du premier plan est la fenêtre active, c'est-à-dire la fenêtre dans laquelle l'utilisateur écrit ou dessine. Cette fenêtre devrait toujours être mise en lumière (contrôles activés). Les fenêtres dans les plans suivants sont dites inactives (et leurs contrôles désactivés). Elles sont partiellement ou totalement (ou pas du tout) cachées par les fenêtres situées devant elles.

On peut avoir des fenêtres invisibles : elles font partie de la liste des fenêtres, mais ne sont pas dessinées par le Window Manager. A ne pas confondre avec les fenêtres cachées ! La fenêtre active est très exactement la première fenêtre visible de la liste gérée par le Window Manager.

Différentes régions d'une fenêtre

Quelle que soit la complexité de la fenêtre, celle-ci possède toujours, qu'elle soit active ou inactive, deux régions : son *contenu* et son *contour.*

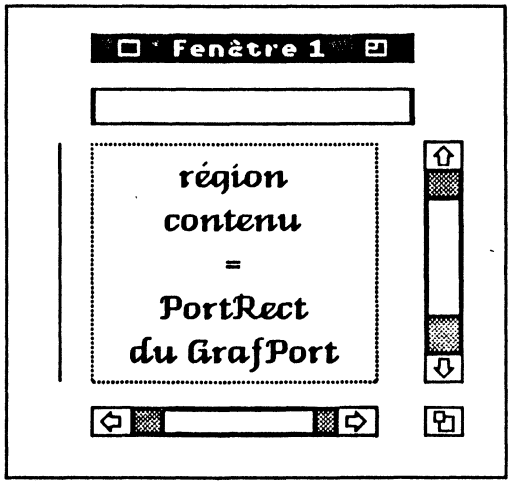


Figure V.4. La fenêtre : un contenu et un contour

La région contenu est définie par un rectangle à spécifier au moment de la création de la fenêtre. Ce rectangle est le *PortRect* du grafport associé à la fenêtre. C'est dans cette région que l'utilisateur écrit ou dessine, ou qu'il reçoit des informations de l'application.

La région contour borde la région contenu. Ce peut être un simple trait (partie gauche de la fenêtre) ou quelque chose de très complexe : barre de titre, zone d'informations, barres de défilement. Plusieurs régions sont contenues dans la région contour : la région définie par la case de fermeture, la région définie par la case de zoom, la région permettant le déplacement de la fenêtre (barre de titre moins les deux régions précédentes), la région définie par la zone d'informations, la région définie par la case de contrôle de taille, ou encore les régions définies par chacune des barres de défilement (flèches, bandes, curseur).

Quand la fenêtre est inactive, toutes ces régions (sauf les cases de fermeture et de zoom) sont encore distinguées par le Window Manager, même si tous les contrôles correspondants sont neutres (voir figure V.3). Un clic dans une fenêtre inactive doit l'activer, rendant son identité à chacune des régions citées ci-dessus par réactivation des contrôles associés (y compris ceux des cases de fermeture et de zoom), sauf peut-être si le clic intervient dans la barre de titre quand la touche Pomme est enfoncée.

Comment une fenêtre est dessinée : l'événement de mise à jour

Quand une fenêtre est déplacée, qu'elle change de taille ou de plan, elle a besoin d'être redessinée, de même que certaines des autres fenêtres visibles présentées à l'écran. Le dessin d'une fenêtre s'effectue généralement en deux étapes : le dessin de la région contour, puis le dessin du contenu.

La première étape est entièrement gérée par le Window Manager, qui redessine dans son propre grafport toutes les régions qui doivent l'être. Il utilise pour cela une fonction de définition de fenêtre par défaut, à moins qu'une autre définition soit fournie par le programmeur pour que son application gère des fenêtres particulières (pourquoi pas des fenêtres en forme de pomme ?). Il utilise aussi une procédure déclarée par l'application pour dessiner le contenu de la zone d'informations, quand elle existe.

Pour réaliser la deuxième étape, le Window Manager génère un événement de mise à jour, un *UpdateEvt* (voir le chapitre IV consacré à l'Event Manager). Il réalise cela en accumulant dans une région particulière qu'il gère au niveau de chaque fenêtre, l'*update region*, les parties de la région contenu qui nécessitent d'être redessinées. Périodiquement, le Window Manager vérifie le contenu de l'*update region*. S'il est vide, pas d'événement de mise à jour. Sinon, il fait dire à l'application, par l'intermédiaire de **GetNextEvent** ou **TaskMaster**, qu'un *UpdateEvt* est survenu, précisant dans le champ *message* de l'*EventRecord* de quelle fenêtre il s'agit (ces événements sont générés dans l'ordre des fenêtres, du premier plan au dernier plan). C'est l'application qui se chargera du reste, en procédant ainsi :

- appel de la procédure **BeginUpdate**, qui remplace temporairement la région visible du grafport de la fenêtre par son intersection avec la région à mettre à jour, et qui remet à vide cette dernière ;
- dessin du contenu de la fenêtre ;
- appel de la procédure **EndUpdate**, qui restaure la région visible correcte.

La figure V.5 montre en trois phases deux événements de mise à jour. La fenêtre X est au deuxième plan (V.5.a), elle va être déplacée verticalement sans être activée (touche Pomme enfoncée durant l'opération). En V.5.b, nous voyons le résultat après déplacement : les régions contour des fenêtres ont été redessinées par le Window Manager, et deux fenêtres ont un contenu nécessitant une mise à jour. Le Window Manager va donc générer deux *update events*, un pour la fenêtre au deuxième plan, puis un pour la fenêtre au troisième plan. L'application prend en compte ces événements, et nous obtenons la figure V.5.c.

Notons un point intéressant : c'est au moment de redessiner la région contour que le Window Manager appelle la procédure de dessin (gérée par l'application) de la zone d'informations. Celle-ci est donc mise à jour avant le contenu de la fenêtre.

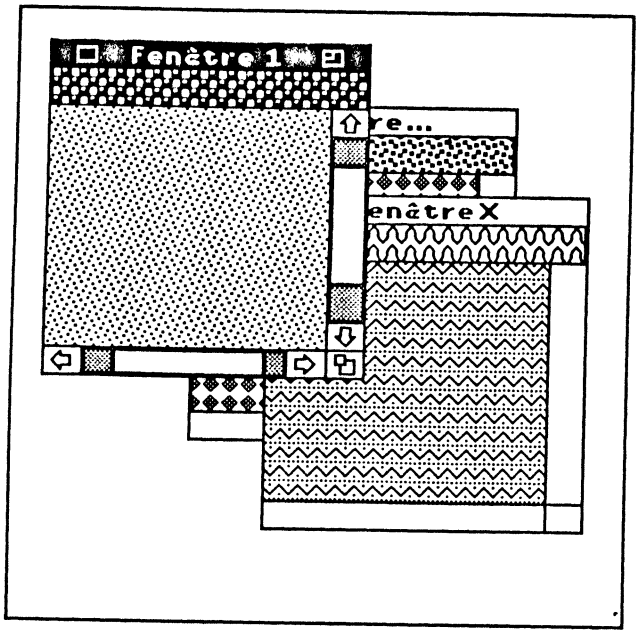


Figure V.5.a. Situation initiale.

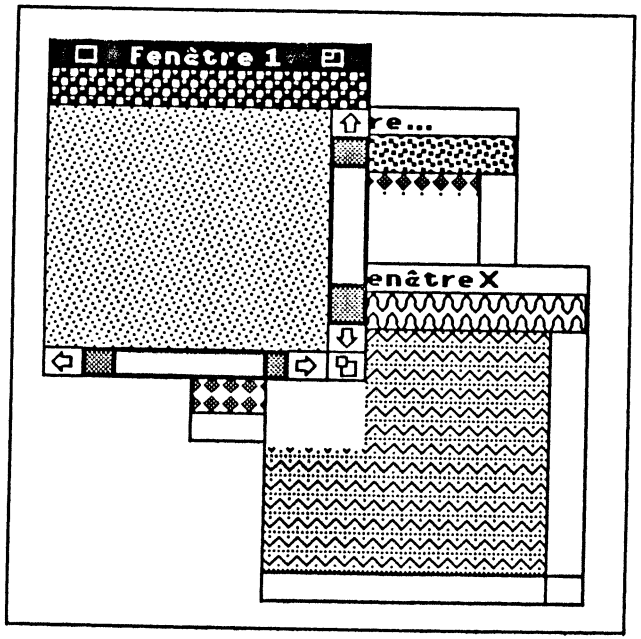


Figure V.5.b. La fenêtre X est déplacée, touche Pomme enfoncée.

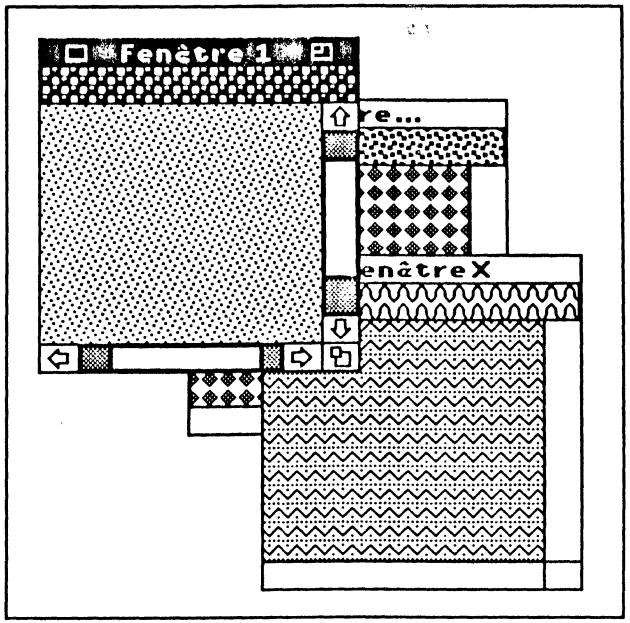


Figure V.5.c. Après les événements de mise à jour.

Comment une fenêtre est activée : l'événement d'activation

Plusieurs routines du Window Manager font passer les fenêtres de l'état actif à l'état inactif, et inversement. Pour de tels changements, le Window Manager génère un événement d'activation/désactivation. Le champ *message* de l'*EventRecord* précise de quelle fenêtre il s'agit, le bit 0 *(ActiveFlag)* du champ *modifiers* précise s'il s'agit d'une activation ou d'une désactivation, et le bit 1 *(ChangeFlag)* du même champ signale en cas d'activation si le type de la fenêtre active a changé (entre fenêtre système et fenêtre de l'application). Remarque importante : dans les versions 1.00 de l'Event Manager et 1.03 du Window Manager, ce bit ne fonctionne pas du tout ainsi. Est-ce un bogue qui sera réparé dans une version future des outils, est-ce la fonctionnalité du bit *ChangeFlag* qui a changé ? A l'heure où nous écrivons ces lignes, nous sommes dans l'incapacité de répondre à ces questions, et nos différents exemples éviteront l'utilisation de ce bit.

Dès que l'Event Manager détecte la présence d'un événement de ce type, il le fait savoir à l'application par l'intermédiaire de **GetNextEvent** ou **TaskMaster**, et de manière immédiate, puisque cet événement est prioritaire sur tous les autres.

Généralement, quand une fenêtre devient active, une autre devient inactive. Aussi les événements de type *ActivateEvt* sont-ils la plupart du temps générés par paires : d'abord l'événement de désactivation, ensuite l'événement d'activation. Parfois, un seul événement intervient, par exemple quand il n'y a qu'une fenêtre dans la liste des fenêtres existantes, ou quand une fenêtre est définitivement détruite.

De manière plus précise, notons que le Window Manager ne génère aucun événement d'activation ou de désactivation à propos d'une fenêtre système (accessoire de bureau, alerte ou modal dialog). Un événement d'activation est généré à l'apparition d'une fenêtre application quand elle passe au premier plan (que ce soit

suite à sa création, ou parce qu'elle a été sélectionnée par l'utilisateur). Un événement de désactivation est généré quand une fenêtre application quitte le premier plan. Aucun événement de désactivation n'intervient si une fenêtre devient invisible ou est fermée.

En réponse à de tels événements, l'application doit procéder de la manière suivante :

- rendre courant le grafport de la nouvelle fenêtre active si l'utilisateur ou l'application doivent directement dessiner dedans, sans passer par un événement de mise à jour ;
- désactiver les contrôles de la fenêtre inactive, activer les contrôles de la fenêtre active (seuls sont concernés les contrôles que l'application a créés elle-même) ;
- dans une fenêtre destinée à manipuler du texte, rendre son aspect normal à la zone sélectionnée ou cacher le point d'insertion clignotant quand la fenêtre devient inactive, restaurer la zone sélectionnée ou le point d'insertion quand elle redevient active ;
- activer ou désactiver certains menus ou articles de menus en fonction des commandes qu'il est possible de lancer à partir de la fenêtre active.

Notons que toutes ces actions sont optionnelles, et on peut très bien imaginer une application ignorant purement et simplement les événements d'activation. Il ne faut pas confondre événement d'activation et événement de mise à jour, même si le passage au premier plan d'une fenêtre nécessite presque toujours de redessiner partiellement ou complètement son contenu.

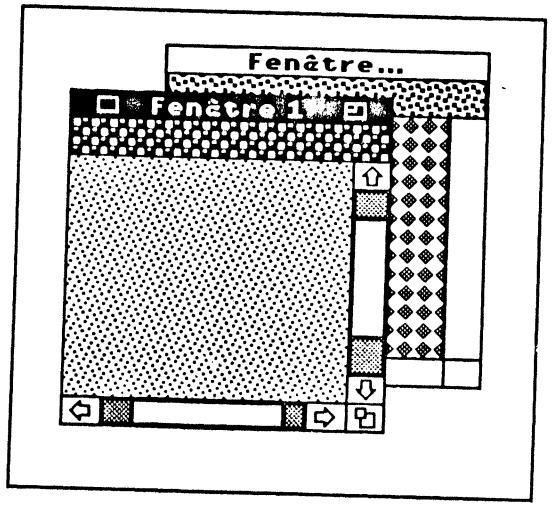


Figure V.6.a. Situation initiale.

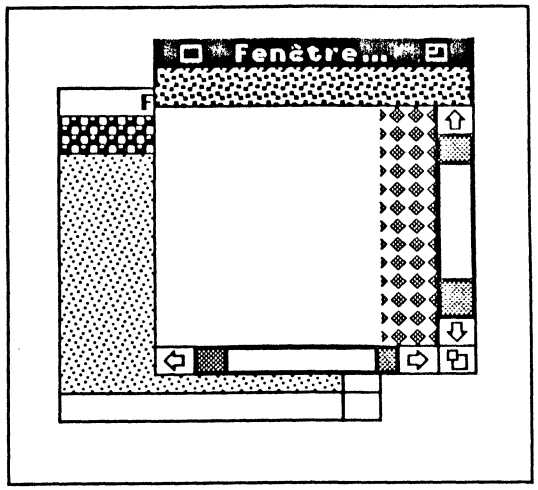


Figure V.6.b. Après les événements d'activation/désactivation.

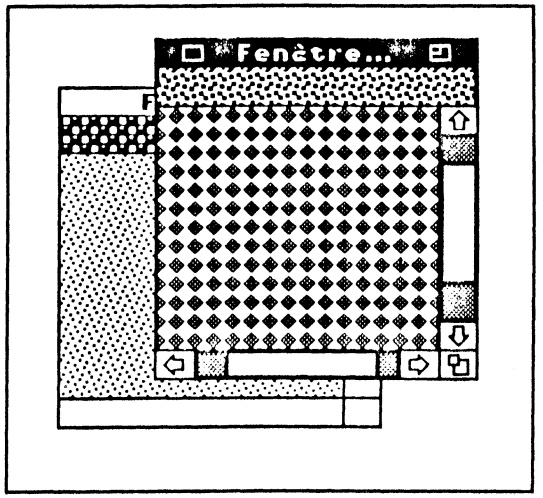


Figure V.6.c. Après l'événement de mise à jour.

La figure V.6 montre l'enchaînement activation – mise à jour pour une fenêtre passant au premier plan. La fenêtre... est au second plan quand l'utilisateur clique dedans, peu importe où (V.6.a). L'application envoie l'ordre de rendre cette fenêtre active, et le Window Manager l'exécute (V.6.b). La fenêtre... passe au premier plan et est contrastée tandis que la fenêtre 1 passe au second plan et est rendue inactive. Le Window Manager a redessiné les contours et appelé la procédure de dessin de la zone d'informations. Il ne lui reste plus qu'à générer l'événement de mise à jour sur la fenêtre de premier plan. L'application le prend en compte, et nous obtenons l'écran de la figure V.6.c.

Structure manipulée par le Window Manager

Pour créer une fenêtre, le Window Manager réclame un très grand nombre de renseignements. Au lieu de les passer les uns après les autres par la pile (en arguments de fonction), on les rassemblera dans une structure particulière, appelée *ParamList*, qui de plus pourra être gérée dans un fichier source réservé aux données, et ne pas être mélangée au code de l'application, afin d'en faciliter la modification.

Voici la définition de cette structure :

```
struct _ParamList {
    int      param_length;
    int      wFrame;
    Pointer  wTitle;
    long     wRefCon;
    Rect     wZoom;
    Pointer  wColor;
    int      wYOrigin;
    int      wXOrigin;
    int      wDataH;
    int      wDataW;
    int      wMaxH;
    int      wMaxW;
    int      wScrollVer;
    int      wScrollHor;
    int      wPageVer;
    int      wPageHor;
    long     wInfoRefCon;
    int      wInfoHeight;
    long     (*wFrameDefProc) ();
    void     (*wInfoDefProc) ();
    void     (*wContDefProc) ();
    Rect     wPosition;
    long     wPlane;
    long     wStorage;
    };
#define ParamList struct _ParamList
```

Explication du contenu des différents champs de cet enregistrement :

• *param_length* donne en octets la taille de la liste complète d'arguments. Bien qu'elle soit de 78 octets, il est préférable de coder ce champ avec l'instruction C *sizeof(ParamList)*.

• *wFrame* est un masque qui définit les caractéristiques de la fenêtre. Chacun des 16 bits de cet entier a une signification précise :

bit 0 : *F_HILITED*. Géré de manière interne, précise si la fenêtre est contrastée (1) ou pas (0).

bit 1 : *F_ZOOMED*. Précise si la fenêtre est en état zoomé (1) ou pas (0). Mettre 0 à la création, sauf si la fenêtre est réellement ouverte dans son état zoomé.

bit 2 : *F_ALLOCATED*. Précise si le *Window record* a été alloué par la fonction **NewWindow** (1) ou directement par l'application (0). Dans le second cas, c'est l'application qui devra libérer l'espace mémoire réservé pour ce *Window record*.

bit 3 : *F_CTRL_TIE*. Précise si les contrôles associés à la définition de la fenêtre restent actifs même quand la fenêtre est inactive (1), ou sont considérés comme inactifs dès que la fenêtre est désactivée (0).

bit 4 : *F_INFO*. Précise si la fenêtre possède une zone d'informations (1) ou pas (0).

bit 5 : *F_VIS*. Précise si la fenêtre est visible (1) ou pas (0). On peut ainsi créer des fenêtres invisibles, dont l'apparition est momentanément différée.

bit 6 : *F_QCONTENT*. Ce paramètre est subordonné à l'utilisation de **TaskMaster**. Si le bit est à 0, un événement de type *MouseDown* dans une fenêtre inactive aura pour effet d'activer la fenêtre, sans plus. Si le bit est à 1, non seulement la fenêtre sera activée, mais l'événement sera encore utilisable, comme s'il avait eu lieu dans la fenêtre active.

bit 7 : *F_MOVE*. Précise si la fenêtre peut être déplacée (1) ou non (0) en faisant glisser la souris à partir de la barre de titre (ou du cadre dans le cas d'une fenêtre type alerte).

bit 8 : *F_ZOOM*. Précise si la barre de titre de la fenêtre contient une case zoom (1) ou pas (0).

bit 9 : *F_FLEX*. Précise si l'aire des données est flexible (1) ou pas (0). Si ce bit est à 1, l'origine ne sera pas modifiée par **GrowWindow** ou **ZoomWindow**.

bit 10 : *F_GROW*. Précise si la fenêtre possède une case de contrôle de taille (1) ou pas (0).

bit 11 : *F_BSCRL*. Précise si la fenêtre possède une barre de défilement horizontale (1) ou pas (0).

bit 12 : *F_RSCRL*. Précise si la fenêtre possède une barre de défilement verticale (1) ou pas (0).

bit 13 : *F_ALERT*. La région contour de la fenêtre est un double trait si ce bit est à 1, aucun autre contrôle ne doit alors être présent. Ce contour agira comme la barre de titre : la fenêtre pourra être déplacée à partir de ses quatre côtés (si *F_MOVE* est à 1, bien sûr).

bit 14 : *F_CLOSE*. Précise si la barre de titre de la fenêtre contient une case de fermeture (1) ou pas (0).

bit 15 : *F_TITLE*. Précise si la fenêtre possède une barre de titre (1) ou pas (0).

Dans l'exemple donné plus loin, *wFrame* prend la valeur 0×DDA0, ce qui signifie, puisque :

DDA0 hexa = 1101 1101 1010 0000 binaire,

présence d'une barre de titre, d'une case de fermeture, des deux barres de défilement, de la case de contrôle de taille et de la case de zoom. Par contre, absence de zone d'informations. De plus, la fenêtre sera visible au moment de sa création et elle pourra être déplacée.

Une autre valeur typique est 0×20A0, désignant une fenêtre visible, pouvant être déplacée et de type alerte (le contour est un trait double qui sert de zone de déplacement).

• *wTitle* est un pointeur sur le titre de la fenêtre (chaîne de caractères de type Pascal). Si la fenêtre n'a pas de barre de titre, on peut mettre zéro-long, ou pointer sur une chaîne vide.

• *wRefCon* est une valeur quelconque (sur quatre octets) associée à la fenêtre, que l'application peut utiliser comme bon lui semble. Nous ferons grand usage de ce champ dans plusieurs exemples de cet ouvrage.

• *wZoom* est un rectangle qui détermine la région contenu quand la fenêtre est zoomée. En mettant à zéro la coordonnée du bas du rectangle *(bottom)*, on utilisera un rectangle par défaut tel que la fenêtre utilise l'écran entier (moins la barre de menus, évidemment).

• *wColor* est un pointeur sur une table qui décrit les couleurs utilisées pour le dessin de la région contour. La valeur zéro-long désignera les couleurs par défaut. Cette table est composée de cinq entiers, soit dix octets. Chaque entier a sa propre codification :

- premier entier : couleur des lignes du contour.
  bits 0 à 3 : inutilisés ;
  bits 4 à 7 : numéro de couleur dans la palette en cours d'utilisation ;
  bits 8 à 15 : à zéro.

- deuxième entier : couleur du titre.
  bits 0 à 3 : numéro de couleur pour le titre et l'intérieur des cases de fermeture et de zoom ;
  bits 4 à 7 : numéro de couleur pour le titre quand la fenêtre est inactive ;
  bits 8 à 11 : numéro de couleur pour la barre de titre quand la fenêtre est inactive ;
  bits 12 à 15 : à zéro.

- troisième entier : barre de titre, couleur et motif.
  bits 0 à 3 : numéro de couleur du fond de la barre de titre (fenêtre active) ;
  bits 4 à 7 : numéro de couleur du motif de remplissage de la barre de titre (fenêtre active) ;
  bits 8 à 15 : style du motif de remplissage (0 = solide, 1 = points, 2 = lignes).

- quatrième entier : couleur de la case de contrôle de taille.
  bits 0 à 3 : numéro de couleur de l'intérieur de la case quand elle est sélectionnée ;
  bits 4 à 7 : numéro de couleur de l'intérieur de la case quand elle n'est pas sélectionnée ;
  bits 8 à 15 : à zéro.

- cinquième entier : couleur de la zone d'informations.
  bits 0 à 3 : inutilisés ;
  bits 4 à 7 : numéro de couleur pour l'intérieur de la zone d'informations ;
  bits 8 à 15 : à zéro.

• *wYOrigin* désigne l'offset vertical de la région contenu, autrement dit l'ordonnée du point de l'aire des données coïncidant avec le coin supérieur gauche du *PortRect* de la fenêtre. Forcer zéro si l'application n'utilise pas de barre verticale de défilement. Cette valeur est aussi utilisée dans le calcul des régions composant cette barre (taille et position du curseur de défilement).

• *wXOrigin* désigne l'offset horizontal de la région contenu, autrement dit l'abscisse du point de l'aire des données coïncidant avec le coin supérieur gauche du *PortRect* de la fenêtre. Forcer zéro si l'application n'utilise pas de barre horizontale de défilement. Cette valeur est aussi utilisée dans le calcul des régions composant cette barre (taille et position du curseur de défilement).

• *wDataH* désigne la hauteur totale de l'aire des données. Forcer zéro si l'application n'utilise pas de barre verticale de défilement. Cette valeur est aussi utilisée dans le calcul des régions composant cette barre (taille et position du curseur de défilement).

• *wDataW* désigne la largeur totale de l'aire des données. Forcer zéro si l'application n'utilise pas de barre horizontale de défilement. Cette valeur est aussi utilisée dans le calcul des régions composant cette barre (taille et position du curseur de défilement).

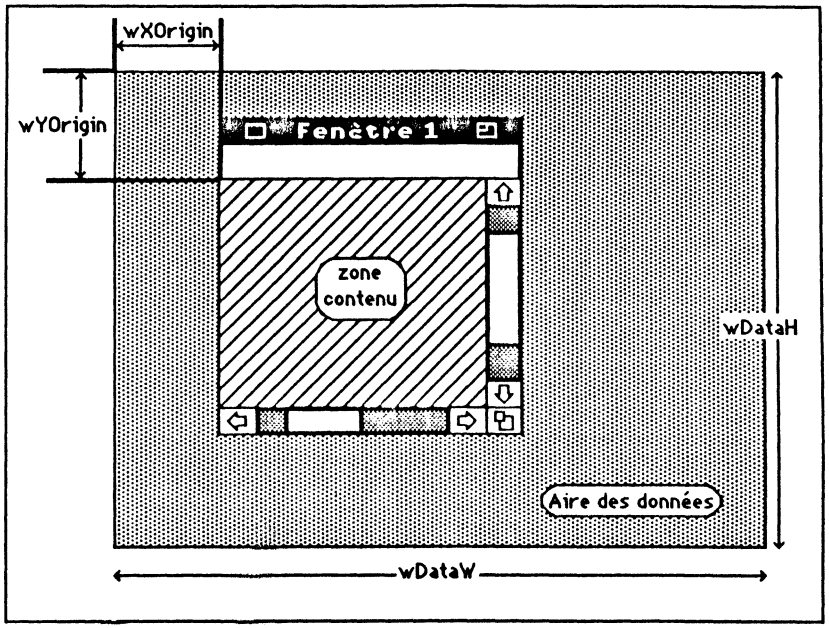


Figure V.7. Contenu de la fenêtre et aire des données.

• *wMaxH* désigne la hauteur maximale autorisée de la région contenu quand la fenêtre est agrandie. Mettre zéro pour autoriser l'utilisation de toute la hauteur de l'écran (barre de menus exclue), ou quand la fenêtre ne possède pas de case de contrôle de taille.

• *wMaxW* désigne la largeur maximale autorisée de la région contenu quand la fenêtre est agrandie. Mettre zéro pour autoriser l'utilisation de toute la largeur de l'écran, ou quand la fenêtre ne possède pas de case de contrôle de taille.

• *wScrollVer* désigne le nombre de pixels duquel **TaskMaster** doit faire défiler le contenu de la fenêtre quand l'une des flèches est sélectionnée dans la barre de défilement verticale. Mettre zéro si la fenêtre ne possède pas de barre verticale, ou si l'application n'utilise pas **TaskMaster.**

• *wScrollHor* désigne le nombre de pixels duquel **TaskMaster** doit faire défiler le contenu de la fenêtre quand l'une des flèches est sélectionnée dans la barre de défilement horizontale. Mettre zéro si la fenêtre ne possède pas de barre horizontale, ou si l'application n'utilise pas **TaskMaster.**

• *wPageVer* désigne le nombre de pixels duquel **TaskMaster** doit faire défiler le contenu de la fenêtre quand la bande de défilement (zones *PageUp* ou *PageDown*) est sélectionnée dans la barre de défilement verticale. Mettre zéro si la fenêtre ne possède pas de barre verticale, ou si l'application n'utilise pas **TaskMaster.** La valeur zéro signifie dans les autres cas hauteur de la région contenu moins dix unités.

• *wPageHor* désigne le nombre de pixels duquel **TaskMaster** doit faire défiler le contenu de la fenêtre quand la bande de défilement (zones *PageUp* ou *PageDown*) est sélectionnée dans la barre de défilement horizontale. Mettre zéro si la fenêtre ne possède pas de barre horizontale, ou si l'application n'utilise pas **TaskMaster.** La valeur zéro signifie dans les autres cas largeur de la région contenu moins dix unités.

• *wInfoRefCon* désigne une valeur (quelconque) qui sera passée à la procédure de dessin de la barre d'information. Mettre zéro-long si la fenêtre n'utilise pas de barre d'information.

• *wInfoHeight* désigne la hauteur en nombre de pixels de la barre d'information. Mettre zéro si la fenêtre n'utilise pas de barre d'information.

• *wFrameDefProc* est un pointeur sur la procédure de définition de la fenêtre, ou zéro-long pour utiliser la procédure standard définie dans le Window Manager.

• *wInfoDefProc* est un pointeur sur la procédure qui doit être appelé pour dessiner le contenu de la zone d'informations. Mettre zéro-long si la fenêtre n'utilise pas de barre d'information.

• *wContDefProc* est un pointeur sur la procédure qui doit être appelé pour dessiner la région contenu de la fenêtre. Cette valeur peut être zéro-long, à condition que la fenêtre ne possède pas de barre de défilement. Si la fenêtre possède des barres de défilement, cette procédure doit exister. Si l'application utilise **TaskMaster**, cette procédure doit exister pour laisser **TaskMaster** gérer toute seule les événements de mise à jour concernant cette fenêtre, qu'elle possède ou non des barres de défilement. **Attention** La procédure de dessin ne doit posséder aucun argument et ne doit retourner aucune valeur, elle ne doit ni changer le grafport courant ni modifier les valeurs *wYOrigin* et *wXOrigin* vues plus haut.

• *wPosition* est un rectangle, donné en coordonnées globales, qui détermine la taille et la localisation de la fenêtre. Ce sera le *PortRect* du grafport de la fenêtre et son coin supérieur gauche sera l'origine du système de coordonnées locales. Pour toute fenêtre dessinée par la procédure standard, ce rectangle définit la région contenu de la fenêtre.

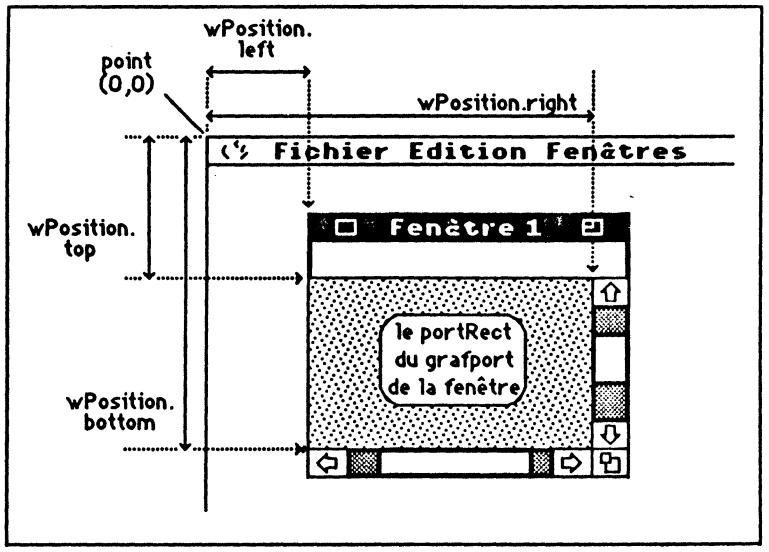


Figure V.8. Position de la fenêtre à l'écran (coordonnées globales).

• *wPlane* est un pointeur désignant la fenêtre derrière laquelle la fenêtre à créer doit apparaître. Mettre zéro-long pour qu'elle soit créée au dernier plan, moins-un-long pour qu'elle apparaisse au premier plan.

• *wStorage* donne l'adresse où sera stocké le *Window record* associé à la fenêtre. Mettre zéro-long pour laisser le système allouer lui-même l'espace nécessaire à cette opération. Cette dernière solution est préférable, mais dans certains cas, il est nécessaire de pouvoir allouer soi-même de la place, par exemple pour définir une fenêtre qui sera utilisée en alerte pour dire qu'il n'y a plus de place pour allouer d'autres fenêtres !

Un grand nombre de ces paramètres ne sont nécessaires qu'à l'utilisation de la fonction **TaskMaster** qui évite pas mal de lignes de programmation quand l'application s'y prête. Bien que cette fonction fasse partie intégrante du Window Manager, nous lui consacrerons spécialement un chapitre, le chapitre XI.

EXEMPLES D'UTILISATION

Création et suppression d'une fenêtre

```
#define mode 0                  /* 0 pour mode 320, 1 pour mode 640 */

void        Paint( );           /* déclaration d'une fonction */
ParamList   params = {
    sizeof(ParamList),          /* taille de cette structure */
    0xDDA0,                     /* contrôles associés à la fenêtre */
    "\22 Nouvelle fenêtre ",    /* pointeur sur le titre de la fenêtre */
    0L,                         /* entier long d'utilisation libre */
    {30, 0, 190, 300+320*mode},     /* rectangle contenu quand la fenêtre est zoomée */
    0L,                         /* pointeur sur la table des couleurs de la fenêtre */
    0, 0,                       /* offset vertical et horizontal de la région contenu */
    300, 800,                   /* hauteur et largeur  de l'aire des données */
    170, 300+320*mode,          /* hauteur et largeur maximum pour la région contenu */
    4, 16,                      /* nombre de pixels à faire défiler (flèches de défilement) */
    40, 160,                    /* nombre de pixels à faire défiler (bandes de défilement) */
    0L,         /* valeur passée à la routine de dessin de la zone d'information */
    0,                          /* hauteur de la zone d'information */
    0L,                         /* adresse de la procédure de définition de la fenêtre */
    0L,                         /* adresse de la routine qui dessine la zone d'information */
    Paint,      /* adresse de la routine qui dessine la région contenu */
    {40, 20, 100, 260 +320*mode }, /* rectangle contenu à la création */
    -1L,                        /* plan de la fenêtre */
    0L          /* adresse où les caractéristiques de la fenêtre sont stockées */
    };
Pointer     theWindow;                  /* pointeur sur la nouvelle fenêtre */

...                                     /* début des initialisations */
WindStartUp(myID);                      /* initialisation du Window Manager */
Refresh(0L);                            /* redessine l'écran */
...                                     /* suite des initialisations */

theWindow = NewWindow(&params);         /* nouvelle fenêtre */
...                                     /* suite de l'application */


void Paint( )                           /* fonction qui dessine le contenu de la fenêtre */
{
...                                     /* instructions pour dessiner le contenu de la fenêtre */
}
```

Le Window Manager est initialisé grâce à la procédure **WindStartUp**. Un seul paramètre, le numéro d'application tel qu'il est retourné par la fonction **MMStartUp** du Memory Manager. La page zéro utilisée étant celle de l'Event Manager, pas besoin de cette notion ici. L'écran est ensuite redessiné par la procédure **Refresh**, qui rétablit un environnement « bureau » à l'écran (menus, fenêtres...).
Remarque La région contour pouvant contenir des contrôles, on aura sans doute à initialiser également le Control Manager. Consulter le chapitre XII pour une vision d'ensemble de l'initialisation des outils.

Pour créer la nouvelle fenêtre, on utilise la fonction **NewWindow**. On précise en argument l'adresse de la structure *ParamList* qui contient toutes les caractéristiques de la nouvelle fenêtre, et la fonction retourne un pointeur qui servira à identifier la fenêtre dans les opérations futures où elle devra être identifiée. Ce pointeur désigne également l'adresse du grafport associé à la fenêtre.

NewWindow alloue l'espace nécessaire pour la bonne utilisation de la fenêtre, ajoute cette fenêtre dans la liste des fenêtres, retient qu'il s'agit d'une fenêtre gérée par l'application et non par le système, etc. Par appel à la procédure QuickDraw **OpenPort, NewWindow** fixe les caractéristiques habituelles d'un grafport par défaut (crayon, motifs de remplissage, jeu de caractères, etc.).

Note NewWindow peut renvoyer des codes d'erreur dans la variable *_errno :*

– $0E01 si la taille de la structure *ParamList* dont l'adresse est passée en argument est incorrecte ;

– $0E02 si elle est incapable d'allouer le *Window record* géré par le Window Manager.

Cette fenêtre va vivre jusqu'au moment où l'application n'en aura plus besoin, soit parce que l'utilisateur a demandé sa fermeture, soit parce qu'elle n'a plus de raison d'être. On utilisera la procédure **CloseWindow** pour la supprimer de la liste des fenêtres :

```
... /* opérations à effectuer avant la suppression (sauvegarde, libération de mémoire) */
CloseWindow(theWindow);                    /* la fenêtre n'existe plus */
```

Un seul argument, le pointeur désignant la fenêtre à supprimer. Cette procédure libère la mémoire occupée par tous les contrôles associés à la fenêtre, qu'ils soient gérés par le Window Manager ou par l'application. Si d'autres structures gérées par l'application étaient associées à cette fenêtre (en liaison par exemple avec le champ d'utilisation libre *wRefCon*), il est de la responsabilité de l'application de les désallouer, sous peine d'encombrer inutilement la mémoire avec les conséquences néfastes que cela pourrait entraîner.

CloseWindow générera si nécessaire les événements d'activation et de mise à jour adéquats pour les fenêtres restant présentes à l'écran.

Note Dans la *ParamList* de l'exemple, nous avons défini une procédure Paint pour dessiner le contenu de la fenêtre automatiquement. Cette procédure ne présente aucun intérêt si on n'utilise pas **TaskMaster**, et on passe alors zéro-long à la place. Voir le chapitre XI pour plus de détails sur l'emploi de cette procédure.

Attention Si votre environnement de développement n'accepte pas le titre de la fenêtre dans la forme que nous venons de voir, il faudra le définir à l'extérieur de la structure *ParamList,* et passer explicitement son adresse, de la manière suivante :

```
char        titrefen[ ] = "\22 Nouvelle fenêtre ";
ParamList   params = {
    sizeof(ParamList),
    0xDDA0,
    titrefen,                    /* pointeur sur le titre de la fenêtre */
    ...
    };
```

Modification des paramètres d'une fenêtre

Dans cette section, nous allons voir comment la plupart des composantes d'un *Window record* peuvent être changées individuellement. Evidemment, ces composantes font pratiquement toutes partie de la structure *ParamList* de création d'une fenêtre, et nous donnerons la liste des routines de modification dans l'ordre des champs de cette structure. Dans tous les exemples qui suivent, l'argument *theWindow* désigne un pointeur sur fenêtre tel qu'il a été retourné par la fonction **NewWindow**.

Nombre de ces routines vont par deux : l'une pour récupérer la valeur courante, l'autre pour fixer une nouvelle valeur. Il est donc facile de modifier provisoirement une valeur, puis de rétablir l'ancienne valeur courante.

• On peut modifier la liste des contrôles associés à une fenêtre, en utilisant la procédure **SetWFrame**. Deux arguments, un masque ayant la structure exacte du champ *wFrame* de la structure *ParamList,* et un pointeur désignant la fenêtre à modifier. Attention, même si la fenêtre est visible, son contour n'est pas redessiné à la suite de cet appel. Une façon d'opérer est de rendre la fenêtre invisible, de faire les modifications et de rendre la fenêtre visible de nouveau, inutile de dire qu'il faut avoir de sérieuses motivations pour utiliser cette procédure ! En cas de besoin, la fonction **GetWFrame** renvoie le masque actuellement utilisé par la fenêtre désignée en argument. L'exemple suivant montre comment supprimer du contour d'une fenêtre la barre de défilement horizontale, grâce à un « et » logique (il ne se passe rien si elle n'existait pas).

```
int wFlag;

wFlag = GetWFrame(theWindow);              /* le contour actuel */
HideWindow(theWindow);                     /* fenêtre rendue invisible */
SetWFrame(wFlag&0xF7FF, theWindow);        /* plus de barre de défilement horizontale! */
SelectWindow(theWindow);                   /* si la fenêtre doit revenir au premier plan */
ShowWindow(theWindow);                     /* fenêtre rendue visible */
```

Remarque au passage : tel qu'il est écrit, l'exemple précédent ne semble rien modifier à l'écran. En effet, pour toutes les modifications qui touchent aux barres de défilement, il faut provoquer leur effacement en changeant la taille de la fenêtre (par **SizeWindow**) ne serait-ce que d'un pixel. Tant que la fenêtre n'est pas redimensionnée (et donc que le contour n'est pas redessiné), la barre reste visible ! Aucune subtilité de ce genre, par contre, pour faire apparaître ou disparaître les cases de fermeture ou de zoom. Voir l'exemple en fin de chapitre XI pour une utilisation concrète de cette procédure.

On notera qu'il ne faut pas faire n'importe quoi avec cette routine. Par exemple, il ne faut pas jouer avec le bit *F_VIS,* mais utiliser les routines qui modifieront l'invisibilité des fenêtres :

– pour rendre une fenêtre invisible, on utilisera la procédure **HideWindow**. Pour rendre une fenêtre visible, on utilisera la procédure **ShowWindow**. Un seul argument dans les deux cas : le pointeur sur la fenêtre considérée.
Attention Ces procédures modifient l'ordre des plans de fenêtres, et des événements de type fenêtre sont générés en conséquence.

– pour ne pas modifier l'ordre des plans, utiliser **ShowHide**. Deux arguments : le premier est un booléen qui prend la valeur TRUE pour rendre la fenêtre visible et FALSE pour la rendre invisible, le second désigne la fenêtre. Aucun événement d'activation n'est généré, aussi cette procédure doit-elle être utilisée avec beaucoup de précautions : si la fenêtre de premier plan est rendue invisible puis visible par cette fonction, elle restera au premier plan, mais contrôles non actifs ! Situation indésirable.

Seule la fonction **GetWFrame** nous permettra de savoir si une fenêtre est visible ou pas. Après s'être assuré qu'elle existe bien, on fera le test suivant :

```
int wFlag;

wFlag = GetWFrame(theWindow);              /* les caractéristiques de la fenêtre */
if (wFlag & 0x0020) ...                    /* fenêtre visible */
else ...                                   /* fenêtre invisible */
```

On ne doit pas non plus jouer avec le bit *F_HILITED.* La procédure **SelectWindow** sert à passer une fenêtre au pemier plan et à l'activer, donc la contraster. Voici une suite d'instructions classique quand on manipule des fenêtres et qu'on est amené à les rendre alternativement invisibles et visibles :

```
HideWindow(theWindow);  /* fenêtre rendue invisible, elle n'est plus au premier plan */
...                     /* suite de l'application */
SelectWindow(theWindow);        /* la fenêtre est sélectionnée... */
ShowWindow(theWindow);          /* ...et rendue visible au premier plan */
```

Remarques

– après l'appel à **HideWindow** dans l'exemple précédent, non seulement la fenêtre est devenue invisible, mais l'ordre des plans a été permuté avec celle qui se trouvait juste derrière elle. Notre fenêtre est invisible au deuxième plan, celle qui suivait est devenue active (visible et au premier plan) ;

– si nous avions fait un **HideWindow** d'une fenêtre différente de la fenêtre active, son plan n'aurait pas été modifié ;

– après l'appel à **SelectWindow** dans l'exemple précédent, la fenêtre est revenue au premier plan, mais elle est toujours invisible. Elle n'est donc pas la fenêtre active. C'est l'un des rares cas où fenêtre de premier plan et fenêtre active ne coïncident pas : la fenêtre active est la première fenêtre visible dans l'ordre des plans ;

– pour mieux comprendre ce qui se passe, vous n'avez qu'à jouer avec l'exemple de fin de chapitre, où on peut rendre visible ou invisible chaque fenêtre, où on peut faire passer au premier plan des fenêtres invisibles, etc. C'est très instructif !

• On peut modifier le titre d'une fenêtre grâce à la procédure **SetWTitle**. Deux arguments : un pointeur sur une chaîne de caractères de type Pascal contenant le nouveau titre et un pointeur désignant la fenêtre à modifier. Le contour est redessiné automatiquement par la procédure, que la fenêtre soit ou non au premier plan. En cas de besoin, la fonction **GetWTitle** renvoie un pointeur sur le titre actuellement utilisé.

```
char    newTitle[ ] = "\32 Nouveau titre de fenêtre ";
Pointer oldTitle;

oldTitle = GetWTitle(theWindow);        /* on récupère l'adresse du titre courant */
SetWTitle(newTitle, theWindow);         /* on fixe un nouveau titre */
```

Remarque Si le titre est généré de manière dynamique par l'application (voir l'exemple complet en fin de chapitre VI), on n'oubliera pas de réserver un espace mémoire suffisant pour l'accueillir, puisque la fenêtre ne le connaît que par l'intermédiaire d'un pointeur. Voir également l'exemple complet en fin de chapitre XI.

Note Pour des raisons d'esthétique, il est préférable de laisser un blanc au début et à la fin du titre. Un accessoire de bureau est annoncé, qui permettra à l'utilisateur de changer lui-même la couleur des fenêtres. Quand des barres de titre style Macintosh sont utilisées (voir plus loin), ces blancs sont les bienvenus !

• On accède au champ *wRefCon* par la procédure **SetWRefCon** (pour fixer sa valeur, un entier long) et la fonction **GetWRefCon** (pour connaître sa valeur) :

```
long    oldRefCon, newRefCon;

oldRefCon = GetWRefCon(theWindow);      /* on récupère ce que contient le champ */
SetWRefCon(newRefCon, theWindow);       /* on fixe la valeur du champ */
```

• On accède au rectangle *wZoom* (taille du contenu de la fenêtre quand celle-ci est zoomée) grâce à la procédure **SetFullRect** (pour fixer sa valeur) et la fonction **GetFullRect** (pour connaître sa valeur) :


```
Pointer oldRect;                /* pointeur sur rectangle */
Rect    newRect;                /* un rectangle */

oldRect = GetFullRect(theWindow); /* on récupère l'adresse du rectangle */
SetRect(&newRect, 0, 30, 620, 180);     /* nouvelles coordonnées pour le rectangle */
SetFullRect(&newRect, theWindow);       /* on fixe la valeur du champ */
```

Remarque La nature du contenu du rectangle *wZoom* change en fonction de la valeur prise par le bit *F_ZOOMED* du champ *wFrame*. Quand ce bit est à 1 (état zoomé), *wZoom* contient la valeur précédente du rectangle contenu de la fenêtre, afin de pouvoir y revenir. Quand ce bit est à 0, *wZoom* contient réellement le rectangle de zoom. On fera donc attention à la valeur du bit avant d'employer **SetFullRect**, sous peine (comme au moment de la création) d'obtenir des résultats surprenants !

• On peut modifier la table des couleurs utilisée pour le dessin des contours, avec la procédure **SetFrameColor**, qui utilise deux arguments, un pointeur sur la définition des couleurs et un pointeur sur la fenêtre visée. Cette procédure sert à changer non seulement la couleur des contours d'une fenêtre, mais également le contenu de la table de couleurs par défaut. Cette table par défaut est une table système noir et blanc gérée par le Window Manager, il suffit pour la modifier de passer zéro-long en guise de pointeur sur fenêtre. C'est cette table par défaut qui sera utilisée si la valeur zéro-long est passée en premier argument. Cette procédure ne redessine pas la fenêtre, on utilisera donc **HideWindow** et **ShowWindow** si nécessaire.

Voyez dans l'illustration à quel point un blanc en tête et un en fin de titre seraient nécessaires !

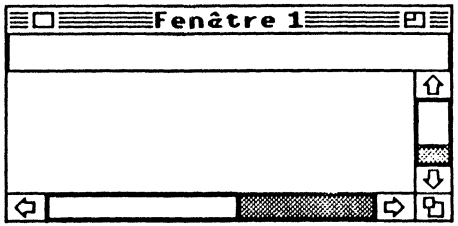


Figure V.9. Barre de titre dans le style Macintosh.

```
int  newColors[5];                      /* les 5 entiers de la table de couleurs */

newColors[0] = 0;                       /* traits noirs */
newColors[1] = 0x0F00;                  /* titre en noir */
newColors[2] = 0x020F;                  /* barre de titre comme sur Macintosh */
newColors[3] = 0xF0F0;                  /* case de contrôle de taille */
newColors[4] = 0x00F0;                  /* zone d'information */

   /* modification de la couleur des contours d'une fenêtre particulière */
HideWindow(theWindow);                  /* fenêtre rendue invisible */
SetFrameColor(newColors, theWindow);    /* on fixe la valeur du champ */
SelectWindow(theWindow);        /* si nécessaire */
ShowWindow(theWindow);          /* fenêtre rendue visible */

   /* modification de la table de couleurs par défaut */
SetFrameColor(newColors, 0L);   /* ne s'applique pas à une fenêtre en particulier */

HideWindow(theWindow);                  /* fenêtre rendue invisible */
SetFrameColor(0L, theWindow);           /* utilisation de la table par défaut */
SelectWindow(theWindow);        /* si nécessaire */
ShowWindow(theWindow);          /* fenêtre rendue visible */
```

En cas de besoin, la procédure **GetFrameColor** permet de récupérer à l'adresse indiquée en premier argument une copie de la table des couleurs actuellement utilisée par la fenêtre désignée en second argument (ou la table par défaut si le second argument est zéro-long).

```
int  oldColors[5];                    /* les 5 entiers de la table de couleurs */

GetFrameColor(&oldColors, theWindow);  /* on récupère les données dans oldColors */
```

• On peut modifier les offsets horizontal et vertical de la région contenu, grâce à la procédure **SetCOrigin**. Le premier argument représente l'offset horizontal, le second l'offset vertical, le troisième désigne la fenêtre. La fonction **GetCOrigin** retourne la valeur des offsets horizontal et vertical, dans un entier long (horizontal dans le mot haut, vertical dans le mot bas).

Ces routines n'auront à être employées que si l'application utilise des barres de défilement, et veut forcer un défilement sans intervention de l'utilisateur sur ces barres. Les barres de défilement étant gérées par **TaskMaster**, nous vous renvoyons donc au chapitre XI pour un développement plus complet à ce sujet.

• On peut modifier la taille (largeur et hauteur) de l'aire des données, grâce à la procédure **SetDataSize**. Le premier paramètre représente la largeur, le second la hauteur, le troisième désigne la fenêtre. La fonction **GetDataSize** retourne la valeur de la largeur et de la hauteur de l'aire des données, dans un entier long (largeur dans le mot haut, hauteur dans le mot bas).

Ces deux routines présentent un intérêt en cas d'utilisation de barres de défilement, donc de **TaskMaster** (voir chapitre XI).

Quand la procédure **SetDataSize** est utilisée, le Window Manager modifie automatiquement les barres de défilement, pour tenir compte de la nouvelle taille relative de la partie visible par rapport à l'aire des données.

```
int     larg,haut;
long    taille;

larg = 1000;                          /* 1000 décimal = 3E8 hexa */
haut = 500;                           /*  500 décimal = 1F4 hexa */
SetDataSize(larg, haut, theWindow);   /* on fixe la nouvelle valeur */
taille = GetDataSize(theWindow);      /* taille contient 03E8 01F4 hexa */
```

Remarque Le langage C tolérera une instruction du style :

```
SetDataSize(taille,theWindow);        /* taille est un entier long */
```

Le mot haut sera considéré comme la largeur, le mot bas comme la hauteur de l'aire des données, puisque **SetDataSize** est une routine type Pascal.

• On peut modifier la taille (largeur et hauteur) d'agrandissement maximal du contenu de la fenêtre, grâce à la procédure **SetMaxGrow**. Le premier paramètre représente la largeur, le second la hauteur, le troisième désigne la fenêtre. La fonction **GetMaxGrow** retourne les valeurs d'agrandissement maximal, dans un entier long (largeur dans le mot haut, hauteur dans le mot bas).

Ces deux routines présentent généralement un intérêt en cas d'utilisation de barres de défilement, donc de **TaskMaster** (voir chapitre XI).


```
int     larg,haut;
long    taille;

larg = 300;                           /* 300 décimal = 12C hexa */
haut = 150;                           /* 150 décimal =  96 hexa */
SetMaxGrow(larg, haut, theWindow);    /* on fixe la nouvelle valeur */
taille = GetMaxGrow(theWindow);       /* taille contient 012C 0096 hexa */
```

Remarque Le langage C tolérera une instruction du style :

```
SetMaxGrow(taille,theWindow);         /* taille est un entier long */
```

Le mot haut sera considéré comme la largeur, le mot bas comme la hauteur de l'aire d'agrandissement *maximal,* puisque **GetMaxGrow** est une routine type Pascal.

• On peut modifier la valeur du nombre de pixels duquel on doit faire défiler le contenu de la fenêtre quand l'une des flèches est sélectionnée dans une barre de défilement, grâce à la procédure **SetScroll**. Le premier paramètre représente le nombre de pixels horizontaux, le second le nombre de pixels verticaux, le troisième désigne la fenêtre. La fonction **GetScroll** retourne ces valeurs dans un entier long (nombre horizontal dans le mot haut, nombre vertical dans le mot bas).

Ces deux routines présentent un intérêt en cas d'utilisation de barres de défilement, donc de **TaskMaster** (voir chapitre XI). L'utilisation de ces routines est absolument identique aux précédentes, nous ne répéterons donc pas les exemples d'utilisation et la remarque concernant le raccourci d'utilisation.

• On peut modifier la valeur du nombre de pixels duquel on doit faire défiler le contenu de la fenêtre quand la bande de défilement est sélectionnée dans une barre de défilement, grâce à la procédure **SetPage**. Le premier paramètre représente le nombre de pixels horizontaux, le second le nombre de pixels verticaux, le troisième désigne la fenêtre. La fonction **GetPage** retourne ces valeurs dans un entier long (nombre horizontal dans le mot haut, nombre vertical dans le mot bas).

Ces deux routines présentent un intérêt en cas d'utilisation de barres de défilement, donc de **TaskMaster** (voir chapitre XI). L'utilisation de ces routines est absolument identique aux précédentes, nous ne répéterons donc pas les exemples d'utilisation et la remarque concernant le raccourci d'utilisation.

• On accède au champ *wInfoRefCon* par la procédure **SetInfoRefCon** (pour fixer sa valeur, un entier long) et la fonction **GetInfoRefCon** (pour connaître sa valeur) :

```
long    oldInfo, newInfo;

oldInfo = GetInfoRefCon(theWindow);   /* on récupère ce que contient le champ */
SetInfoRefCon(newInfo, theWindow);    /* on fixe la valeur du champ */
```

• La procédure de définition de la fenêtre peut être fixée par **SetDefProc**, à qui on passe son adresse et l'adresse de la fenêtre. La fonction **GetDefProc** retourne cette adresse.

• La procédure du dessin de la zone d'information de la fenêtre peut être fixée par **SetInfoDraw**, à qui on passe son adresse et l'adresse de la fenêtre. La fonction **GetInfoDraw** retourne cette adresse.

• La procédure de dessin du contenu de la fenêtre peut être fixée par **SetCDraw**, à qui on passe son adresse et l'adresse de la fenêtre. La fonction **GetCDraw** retourne cette adresse. Ainsi, pour changer la procédure de dessin du contenu d'une fenêtre en cours d'application, on écrira :

```
void Paint( );                        /* Paint doit évidemment être défini ailleurs */

SetCDraw(Paint, theWindow);
```

Et il y aura intérêt après cela à générer un événement de mise à jour pour la totalité du contenu de la fenêtre ! (on utilisera la procédure **InvalRect**, voir plus loin).

• A chaque fenêtre sont associées trois régions : la région contenu (là où dessine l'application), la région structure (c'est-à-dire le contenu plus le contour) et la région à mettre à jour (à partir de laquelle sont générés les *UpdateEvt*). Pour récupérer un handle sur ces régions (définies dans le système de coordonnées globales), on utilisera les fonctions **GetContRgn, GetStructRgn** et **GetUpdateRgn**. On verra dans plusieurs exemples l'intérêt de récupérer ces handles, du moins les deux premiers, quand on gère plusieurs curseurs différents.

```
Handle cont, struc, upd;

cont = GetContRgn(theWindow);
struc = GetStructRgn(theWindow);
upd = GetUpdateRgn(theWindow);
```

• A chaque fenêtre est associé un indicateur permettant de savoir si la fenêtre a été créée par l'application ou générée par le système (c'est le cas notamment des fenêtres d'accessoires de bureau). Deux fonctions (quelle richesse !) permettent de connaître cet indicateur pour la fenêtre dont le pointeur est passé en argument : **GetWKind** et **GetSysWFlag**. L'une et l'autre retourneront FALSE (zéro) pour une fenêtre de l'application et TRUE (valeur non nulle) pour une fenêtre système.

```
int flag;

flag = GetWKInd(theWindow);          /* nul si la fenêtre appartient à l'application... */
flag = GetSysWFlag(theWindow);       /* ...non nul si elle appartient à un accessoire */
```

Ces procédures permettront par exemple de déterminer si la fenêtre de premier plan appartient à l'application ou à un accessoire de bureau, ce qui permettra de pallier en partie la déficience du bit *ChangeFlag* du champ *modifiers* d'un événement.

• Les auteurs d'accessoires de bureau pourront marquer leurs fenêtres du sceau Système en utilisant la procédure **SetSysWindow**, avec le pointeur sur la fenêtre comme argument. Interdiction d'utiliser cette routine dans une application normale !

Interaction avec l'utilisateur

• Réponse à un événement de type *MouseDown* (bouton souris enfoncé). Revoir la boucle de gestion des événements dans le chapitre consacré à l'Event Manager.

```
int      loc;                        /* où le bouton de la souris a été enfoncé */
Pointer theWindow;                   /* dans quelle fenêtre */

...
case MouseDown :
loc = FindWindow(&theWindow, tache.where);
if (loc<0) SystemClick(&tache, theWindow, loc);        /* accessoire de bureau */
else switch(loc)                     /* où le bouton de la souris a-t-il été enfoncé? */
   {
   case wInDesk :                    /* dans une partie libre du bureau */
   ...               /* généralement, on ne fait rien dans ce cas */
   break;

   case wInMenuBar :                 /* dans la barre de menus système */
   ...               /* appel de MenuSelect dans le Menu Manager */
   break;
```



```
   case wInContent :                 /* dans la région contenu d'une fenêtre */
   ...               /* on répond comme l'application doit réagir en invoquant */
   ...               /* le Control Manager ou Line Edit ou QuickDraw ou autre chose */
   break;

   case wInDrag :                    /* dans la barre de titre (hors cases de contrôle) */
   ...               /* appel de DragWindow, voir ci-après */
   break;

   case wInGrow :                    /* dans la case de contrôle de taille */
   ...               /* appel de GrowWindow, voir ci-après */
   break;

   case wInGoAway :                  /* dans la case de fermeture */
   ...               /* appel de TrackGoAway, voir ci-après */
   break;

   case wInZoom :                    /* dans la case de zoom */
   ...               /* appel de TrackZoom, voir ci-après */
   break;

   case wInInfo :                    /* dans la zone d'information */
   ...               /* on répond comme l'application doit réagir */
   break;

   case wInFrame :                   /* dans ce qui reste des contours */
   ...               /* pas grand chose à faire, malheureusement */
   break;
   }
break;
...
```

Voici donc la structure de la réponse à apporter à un événement de type *MouseDown*. On commence par appeler la fonction **FindWindow**. Cette fonction attend trois arguments. Le premier représente l'adresse où elle va retourner un pointeur sur la fenêtre dans laquelle l'utilisateur a cliqué (ou zéro-long si l'utilisateur n'a pas cliqué dans une fenêtre). Le second contient l'abscisse et le troisième l'ordonnée (en coordonnées globales) du point où le bouton de la souris a été enfoncé, tel qu'il est stocké dans le champ *where* de l'événement. Par un tour de passe-passe dont nous ne tarderons pas à être familier et que nous avons déjà évoqué plusieurs fois, les deux entiers définissant l'abscisse et l'ordonnée du point peuvent être remplacés par l'entier long définissant le point lui-même, l'ordre des composantes ayant été choisi de telle sorte que le raccourci fonctionne sans anicroche.

FindWindow retourne un résultat explicite (sur deux octets), un code qui représente la zone dans laquelle le bouton a été enfoncé. Pour être plus lisible, le code peut prendre les valeurs prédéfinies suivantes :

```
#define wInDesk        16
#define wInMenuBar     17
#define wInContent     19
#define wInDrag        20
#define wInGrow        21
#define wInGoAway      22
#define wInZoom        23
#define wInInfo        24
#define wInFrame       27
```

Ces valeurs sont celles qui interviennent quand une fenêtre appartenant à l'application est invoquée. Si l'utilisateur a cliqué dans une partie d'une fenêtre système (appartenant donc à un accessoire de bureau), la valeur retournée par **FindWindow** est négative : valeur identique, mais le bit 15 est forcé à 1, ce qui revient

à ajouter la valeur hexadécimale 8 000 aux codes précédemment définis. Les applications gérant les accessoires de bureau devront appeler la procédure **SystemClick** du Desk Manager si la valeur retournée par **FindWindow** est négative.

Si l'utilisateur a cliqué dans une fenêtre d'application active, toutes les valeurs sont susceptibles d'être retournées. Si la fenêtre n'est pas active, **FindWindow** perd trace des cases de fermeture et de zoom (ces cases ne sont d'ailleurs pas dessinées dans la barre de titre d'une fenêtre inactive), mais distingue encore les autres régions (zone d'informations, contour, case de contrôle de taille, barre de titre et contenu). Dans tous les cas, sauf si la souris est pressée dans la barre de titre et que la touche Pomme est enfoncée, la fenêtre doit être activée.

Remarque Si la fenêtre a été créée avec les bits *F_ALERT* et *F_MOVE* positionnés, tout le cadre est considéré *wInDrag*. La fenêtre peut donc être déplacée par n'importe quelle partie de son contour, et plus seulement par le haut (une telle fenêtre ne possède pas de barre de titre).

C'est la fonction **FrontWindow** qui nous permet de savoir si la fenêtre est active ou non, puisque son rôle est de renvoyer l'adresse de l'actuelle fenêtre active (première fenêtre visible dans l'ordre des plans). Si cette adresse coïncide avec celle renvoyée par **FindWindow**, l'utilisateur a cliqué dans la fenêtre active et l'application doit répondre de manière appropriée à cette sollicitation. Sinon, pas de question à se poser : il faut activer la fenêtre dans laquelle l'utilisateur a cliqué, en appelant la procédure **SelectWindow** avec pour argument le pointeur désignant la fenêtre.

SelectWindow fait les choses proprement : elle amène la nouvelle fenêtre active au premier plan, la « met en lumière », « éteint » la fenêtre active précédente, et génère les événements d'activation adéquats.

• L'utilisateur a cliqué dans la région contenu d'une fenêtre. Si celle-ci n'est pas active, on l'active et on ne fait rien de plus. Si par contre elle est active, l'application répond comme elle doit réagir : en appelant les routines du Control Manager si la fenêtre est susceptible de contenir des contrôles créés par l'application (voir le chapitre VII), les routines de Line Edit si la fenêtre contient du texte éditable (voir le chapitre VIII), les routines de QuickDraw si un dessin doit intervenir dans cette fenêtre...

```
case wInContent:
if (theWindow != FrontWindow())          /* cette fenêtre est-elle active? */
SelectWindow(theWindow); /* non, alors on l'active et on ne fait rien de plus */
else
...                      /* suivant le rôle de l'application */
break;
```

• L'utilisateur a cliqué dans la barre de titre (ou dans le cadre d'une fenêtre type alerte) et la fenêtre a été déclarée déplaçable. La fonction **FindWindow** a renvoyé *wInDrag*, il va donc falloir vérifier si l'utilisateur veut déplacer la fenêtre, en faisant glisser la souris. On respectera les règles de l'interface utilisateur : si la touche Pomme est enfoncée au moment du clic souris, la fenêtre sera déplacée mais non activée (au cas où elle serait inactive). Si la touche Pomme n'est pas enfoncée, la fenêtre sera activée (si nécessaire) et déplacée.

Une seule procédure à appeler, **DragWindow**, qui va faire tout le travail : tant que l'utilisateur n'aura pas relâché son bouton, dessin du rectangle représentant la fenêtre en train de se déplacer ; dès qu'il l'a relâché, déplacement effectif de la fenêtre vers sa nouvelle localisation et génération des *update events* pour elle et éventuellement pour les autres. La seule chose que l'application doit fournir, c'est l'ensemble des arguments nécessaires à **DragWindow** :

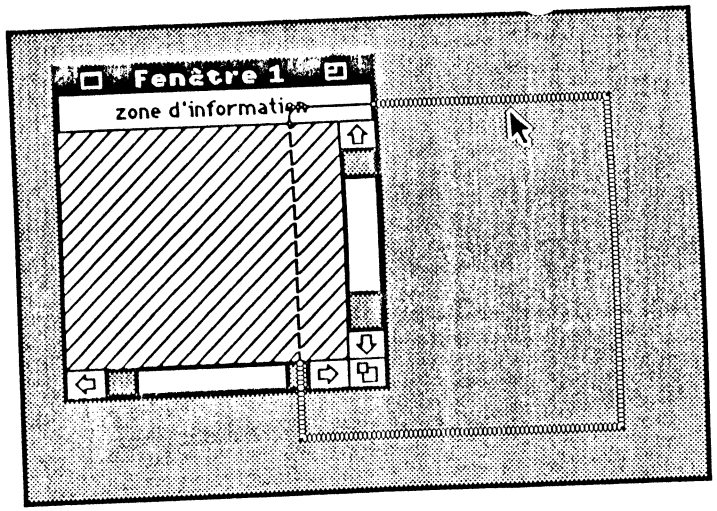


Figure V.10. Fenêtre en cours de déplacement.

– premier argument : un entier précisant la résolution du déplacement horizontal. Par défaut, la fenêtre sera déplacée de quatre points en quatre points en mode 320 et de huit points en huit points en mode 640, comme sur une grille invisible. Donner la valeur zéro, c'est accepter ces valeurs par défaut. La valeur 1 permet de placer la fenêtre n'importe où. Les autres valeurs autorisées sont les puissances de deux exclusivement ;

– deuxième et troisième arguments : l'abscisse et l'ordonnée (coordonnées globales) du point où débute l'action, c'est-à-dire le point stocké dans le champ *where* de l'événement. C'est le déplacement du pointeur par rapport à ce point qui déterminera le déplacement global de la fenêtre. Comme nous l'avons déjà dit, C permet de passer un entier long à la place des deux composantes sur 16 bits ;

– quatrième argument : un entier dont la signification est donnée ci-après ;

– cinquième argument : un pointeur sur un rectangle (coordonnées globales) qui précise les limites dans lesquelles le point qui suit la souris peut se déplacer. Si la souris sort de ce rectangle mais ne s'en éloigne pas plus que de la valeur contenue dans le quatrième argument, le point de prise de la fenêtre reste figé à la frontière du rectangle (la silhouette de la fenêtre s'immobilise par voie de conséquence). Au-delà, la silhouette disparaît (plus exactement elle revient à son point de départ), et si l'utilisateur relâche le bouton de la souris à ce moment-là, la fenêtre ne sera pas déplacée. Par défaut (si on passe zéro-long comme pointeur), le Window Manager utilisera comme rectangle frontière l'ensemble du bureau (barre de menus exclue, évidemment) moins quatre pixels de chaque côté, la « distance de grâce » (quatrième argument) étant de huit pixels. Ce rectangle frontière par défaut est choisi de telle manière que toute fenêtre visible présente à l'écran au moins seize pixels (quand elle est tirée au maximum en dehors de l'écran) ;

– sixième argument : un pointeur qui désigne la fenêtre à déplacer.

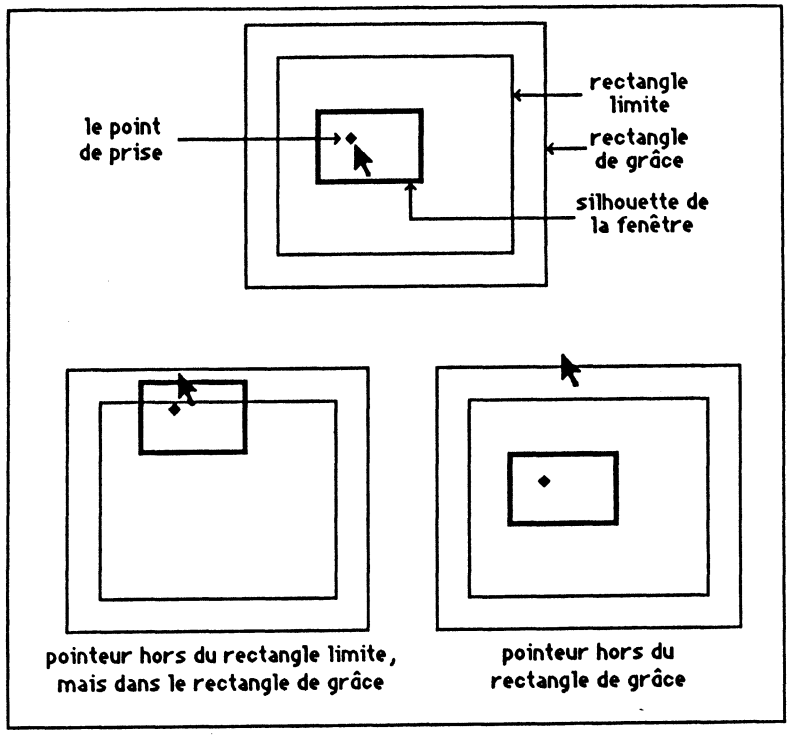


Figure V.11. Rectangle limite et rectangle de grâce.

```
Rect    limitRect;
int     left, top, right, bottom;

case wInDrag:
...             /* calcul éventuel des coordonnées du rectangle frontière */
SetRect(&limitRect, left, top, right, bottom);      /* on fixe le rectangle frontière */
if (theWindow != FrontWindow() && !(tache.modifiers & AppleKey))
    SelectWindow(theWindow);
DragWindow(1, tache.where, 8, &limitRect, theWindow);
break;
```

Attention au calcul du rectangle frontière : il ne faudrait pas que la totalité de la barre de titre d'une fenêtre classique puisse disparaître sous la barre de menus, car alors, l'utilisateur ne pourrait plus la déplacer ! Le plus simple, c'est encore d'utiliser les valeurs par défaut, comme le fera **TaskMaster** :

```
case wInDrag:
if (theWindow != FrontWindow() && !(tache.modifiers & AppleKey))
    SelectWindow(theWindow);
DragWindow(0, theEvent.where, 8, 0L, theWindow);
break;
```

Remarque Pour dessiner la fenêtre au moment où l'utilisateur lâche le bouton de la souris, **DragWindow** appelle la procédure **MoveWindow**, qui efface une fenêtre d'un endroit de l'écran pour la redessiner ailleurs, sans en modifier la taille. Il suffit de donner à **MoveWindow** la nouvelle abscisse et la nouvelle ordonnée (en coordonnées globales, évidemment) du coin supérieur gauche de la région contenu de la fenêtre, ainsi que le pointeur de la fenêtre à déplacer en troisième argument. On fera attention dans l'opération à ce que la barre de titre ne disparaisse pas complètement au-dessous de la barre des menus.

On verra dans l'un des exemples de fin du chapitre VI comment **MoveWindow** est utilisée pour créer des fenêtres décalées les unes par rapport aux autres.

- L'utilisateur a cliqué dans la case de contrôle de taille. La fonction **FindWindow** a renvoyé *wInGrow*, il va donc falloir vérifier tout d'abord si la fenêtre incriminée est active ou pas, et dans ce dernier cas, si l'utilisateur veut agrandir ou réduire la fenêtre, en faisant glisser la souris. (Si la fenêtre n'est pas active, on la sélectionne, c'est tout).

Deux routines à appeler : la fonction **GrowWindow**, qui va faire la première partie du travail (tant que l'utilisateur n'aura pas relâché son bouton, dessin des rectangles représentant la fenêtre et ses barres de défilement et train de changer de taille), et la procédure **SizeWindow**, qui termine le travail (affectation d'une nouvelle taille pour le rectangle contenu de la fenêtre, dessin du nouveau contour, avec ses contrôles modifiés en conséquence, et génération des événements de mise à jour pour les fenêtres qui en ont besoin).

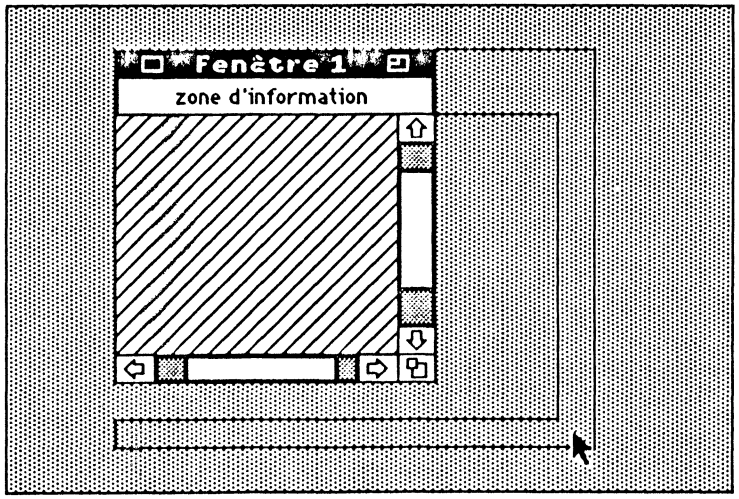


Figure V.12. Fenêtre en cours d'agrandissement.

La fonction **GrowWindow** réclame cinq arguments. Les deux premiers indiquent la taille minimale que le rectangle contenu de la fenêtre pourra prendre (largeur puis hauteur). Les deux suivants précisent le point où commence l'action (abscisse puis ordonnée), c'est-à-dire le point stocké dans le champ *where* de l'événement. Le cinquième désigne la fenêtre incriminée. La fonction retourne dans un entier long la nouvelle taille de la fenêtre (largeur dans le mot haut, hauteur dans le mot bas), ou la valeur zéro-long si la fenêtre n'a pas changé de taille.

La procédure **SizeWindow** admet trois arguments. Les deux premiers indiquent la nouvelle taille de la fenêtre (largeur plus hauteur). C'est exactement l'entier long retourné par **GrowWindow**, en utilisant le raccourci dont nous sommes maintenant coutumiers. Si les deux entiers sont nuls, **SizeWindow** ne fait rien. Le troisième argument spécifie la fenêtre à redessiner.

```
case wInGrow:
if (theWindow != FrontWindow()) SelectWindow(theWindow);
else
SizeWindow(GrowWindow(50,50,tache.where,theWindow), theWindow);
break;
```

• L'utilisateur a cliqué dans la case de contrôle de fermeture de la fenêtre. La fonction **FindWindow** a renvoyé *WinGoAway*, il faut vérifier si l'utilisateur a relâché le bouton de la souris à l'intérieur ou à l'extérieur de cette case avant d'entreprendre quelque action que ce soit. C'est la fonction **TrackGoAway** qui s'en charge. Trois arguments : l'abscisse et l'ordonnée du point où débute l'action, c'est-à-dire le point stocké dans le champ *where* de l'événement ; le pointeur qui désigne la fenêtre. La fonction retourne une valeur booléenne : FALSE signifie que l'utilisateur a relâché le bouton à l'extérieur de la case de fermeture, TRUE signifie que la fenêtre doit bien être fermée. A l'application de faire le nécessaire pour fermer la fenêtre.

```
case wInGoAway :
if(TrackGoAway(tache.where, theWindow)  /* bouton relâché dans la case? */
{               /* oui... */
...             /* proposer à l'utilisateur d'enregistrer ses modifications */
...             /* fermer les fichiers associés à la fenêtre */
...             /* désallouer les différentes structures associées à la fenêtre */
CloseWindow(theWindow);
}
break;
```

En fonction de l'application, un **HideWindow** pourra être appelé à la place de **CloseWindow** (pour autoriser une réouverture de la fenêtre à partir d'un menu déroulant, par exemple, sans avoir à réallouer toutes les structures associées).

• L'utilisateur a cliqué dans la case de zoom. La fonction **FindWindow** a renvoyé *wInZoom*, il faut vérifier si l'utilisateur a relâché le bouton de la souris à l'intérieur ou à l'extérieur de cette case avant d'entreprendre quelque action que ce soit. C'est la fonction **TrackZoom** qui s'en charge. Trois arguments : l'abscisse et l'ordonnée du point où débute l'action, c'est-à-dire le point stocké dans le champ *where* de l'événement ; le pointeur qui désigne la fenêtre. La fonction retourne une valeur booléenne : FALSE signifie que l'utilisateur a relâché le bouton à l'extérieur de la case de fermeture, TRUE signifie que la fenêtre doit bien être zoomée. A l'application de faire le nécessaire, en appelant la procédure **ZoomWindow**.

```
case wInZoom :
if(TrackZoom(tache.where, theWindow)        /* bouton relâché dans la case? */
   ZoomWindow(theWindow);                   /* oui, on zoome */
break;
```

ZoomWindow ne réclame qu'un argument : le pointeur sur la fenêtre à zoomer. Cette procédure ne fait qu'appeler **SizeWindow**, en lui passant les bons arguments : si la fenêtre est déjà à sa taille maximale de zoom, elle reviendra à sa taille précédente, sinon elle y passera. Par taille maximale de zoom, comprenons le rectangle *wZoom* passé en argument au moment de la création de la fenêtre (ou modifié par la procédure **SetFullRect**), et non *wMaxH* et *wMaxW* qui ne sont utilisés qu'avec le déplacement de la case de contrôle de taille.

• L'utilisateur a cliqué dans la zone d'informations d'une fenêtre. Si celle-ci n'est pas active, on l'active et on ne fait rien de plus. Si par contre elle est active, l'application répond comme elle doit réagir : en appelant les routines du Control Manager si la zone d'informations est susceptible de contenir des contrôles créés par l'application (nous ne verrons pas cette possibilité dans cet ouvrage), les routines du Menu Manager si elle contient des menus déroulants (possibilité que nous ne verrons pas non plus), les routines de QuickDraw si un dessin doit intervenir dans cette zone... ou en ne faisant rien : après tout, une zone d'informations peut aussi servir à ne donner que des informations !

```
case wInInfo :
if (theWindow != FrontWindow())          /* cette fenêtre est-elle active? */
   SelectWindow(theWindow);     /* non, alors on l'active et on ne fait rien de plus */
else
...                             /* suivant le rôle de l'application */
break;
```


• L'utilisateur a cliqué dans la région contour d'une fenêtre, hors de la barre de titre ou de la zone d'information (ou encore dans la barre de titre d'une fenêtre qui ne peut être déplacée). Si celle-ci n'est pas active, on l'active et on ne fait rien de plus. Si la fenêtre est active, le clic peut avoir eu lieu dans une barre de défilement. On pourrait alors penser pouvoir gérer soi-même ce contrôle, et faire défiler le contenu de la fenêtre, en faisant appel aux routines du Control Manager (c'est comme cela que ça se passe sur Macintosh). Malheureusement, les barres de défilement sont des contrôles créés et gérés par le Window Manager, et sauf à faire des acrobaties peu avouables sur certains pointeurs (du style permuter l'adresse du premier contrôle géré par l'application et celle du premier contrôle géré par le Window Manager en allant toucher directement à la structure définissant la fenêtre, ce qui est en principe interdit), la fonction **FindControl** ne saura les repérer.
Conclusion De deux choses l'une : soit l'application veut gérer elle-même son défilement, et elle crée elle-même les barres de défilement et la case de contrôle de taille dans la région contenu de la fenêtre (comme sur Macintosh), soit elle utilise **TaskMaster**, qui fera tout cela parfaitement en utilisant les contrôles créés par le Window Manager dans la région contour de la fenêtre.

Nous prenons personnellement le parti d'utiliser **TaskMaster**, ce qui est plus dans l'esprit de l'Apple IIGS. C'est pourquoi dans l'exemple qui est donné à la fin de ce chapitre et qui utilise **GetNextEvent**, nous créerons des fenêtres avec barres de défilement... incapables de défiler. Pour voir une barre défiler, rendez-vous dans le chapitre VII consacré au Control Manager, et bien sûr au chapitre XI consacré à **TaskMaster**.

Événements de type fenêtre

• Réponse à un événement de mise à jour. Puisque la fenêtre à mettre à jour n'est pas forcément la fenêtre active (si par exemple une partie d'une fenêtre d'arrière-plan a été découverte à la suite d'un déplacement d'une fenêtre d'avant-plan), certaines précautions doivent être prises pour y dessiner, notamment faire du grafport qui lui est associé le grafport courant, après avoir gardé trace du grafport en cours pour pourvoir le rétablir à la fin de l'opération de mise à jour.

```
Pointer savePort;               /* pointeur sur le précédent grafport */
Pointer port;                   /* pointeur sur la fenêtre à mettre à jour */

case UpdateEvt :
savePort = GetPort( );          /* sauvegarde du grafport courant */
port = tache.message ;          /* pointeur sur la fenêtre à traiter */
SetPort(port);          /* on va pouvoir dessiner dans la bonne fenêtre */
BeginUpdate(port);              /* restriction de la région visible à l'update region */
...                     /* ce que l'application a à dessiner */
EndUpdate(port);                /* rétablissement de la région visible originale */
SetPort(savePort);              /* rétablissement du grafport original */
break;
```

Comme il a été dit, le dessin du contenu de la fenêtre à mettre à jour s'effectue entre l'appel de deux procédures, **BeginUpdate** et **EndUpdate**. Seul argument pour chacune, un pointeur sur la fenêtre à traiter. La première restreint la région visible à l'*update region* et vide cette dernière, la seconde rétablit la région visible originale. Ce qu'il y a entre ces deux appels est de la responsabilité de l'application : ce sera généralement l'appel à la procédure de dessin (si elle existe) définie dans le champ *wContDefProc* de la *ParamList* de création de la fenêtre, ou bien aux routines de dessins du Control Manager si l'application manipule ses propres contrôles, ou encore aux routines de mise à jour de Line Edit si l'application manipule du texte éditable.

• On peut modifier directement le contenu de l'*update region* du grafport courant grâce à quatre procédures. **InvalRect** ajoute à l'*update region* le rectangle dont un

pointeur est passé en argument. **InvalRgn** ajoute à l'*update region* la région dont un handle est passé en argument. A l'inverse, **ValidRect** retire de l'*update region* le rectangle dont un pointeur est passé en argument et **ValidRgn** retire de l'*update region* la région dont un handle est passé en argument. But principal de ces procédures : provoquer ou empêcher un *update event.*

Dans tous les cas, le rectangle ou la région doivent être exprimés en coordonnées locales. Ils seront automatiquement limités *(clipped)* au rectangle contenu de la fenêtre.

Au lieu de dessiner directement dans une fenêtre, on générera artificiellement un événement de mise à jour en déclarant un rectangle ou une région invalides, et c'est en réponse à cet événement que ce rectangle ou cette région seront redessinés. (voir les exemples du chapitre VI ou du chapitre VIII pour une illustration de ce principe). Réciproquement, pour empêcher l'événement de mise à jour, on déclarera tel rectangle ou telle région valides, et l'événement de mise à jour (s'il existe encore) ne les touchera pas.

Pour redessiner la totalité du contenu d'une fenêtre (par exemple parce qu'on vient de changer sa procédure de dessin automatique), on pourra écrire :

```
Rect    r;
Pointer theWindow;

SetRect(&r,0,0,1000,1000);        /* rectangle arbitrairement grand */
InvalRect(&r);                    /* ajoute le rectangle à la région à mettre à jour */
```

Le rectangle étant plus grand que l'écran de l'Apple IIGS et possédant pour coin supérieur gauche le point (0,0), nous sommes sûr qu'il couvre la totalité du contenu visible de la fenêtre dans le système de coordonnées locales. Son intersection avec la région contenu ne posera donc aucun problème !

• Réponse à un événement d'activation/désactivation. Comme nous l'avons déjà dit, la réponse à ce type d'événement est facultative et dépend des besoins de l'application, qui pourra appeler les routines du Menu Manager, du Control Manager ou de Line Edit. Il pourrait être intéressant par exemple de désactiver tous les menus propres à l'application quand une fenêtre d'accessoires de bureau devient active, et de ne garder que le menu *Édition* et l'article *Fermer* du menu *Fichier*, l'application devant alors gérer cet article pour ses propres fenêtres mais aussi pour les fenêtres système.

```
case ActivateEvt :                    /* événement d'activation */
if (myEvent.modifiers & ActiveFlag)   /* teste l'indicateur activation/désactivation */
...                                   /* il est à 1: la fenêtre doit être activée */
else
...                                   /* il est à 0: la fenêtre doit être désactivée */
break;
```

Nous verrons plusieurs exemples où les événements d'activation sont gérés, plusieurs autres où ils ne le sont pas.

Dessin de la zone d'informations

L'opération qui consiste à dessiner dans la zone d'informations peut être très simple ou très complexe en fonction d'une situation donnée.

Il est très simple de dessiner une zone qui ne changera pas au cours du temps, par exemple un texte assimilable à un sous-titre pour la fenêtre. On explique au Window Manager comment le dessiner une fois pour toutes (c'est le rôle de la procédure *wIngoDefProc* définie au niveau de la *ParamList*), et ensuite, c'est lui qui se débrouille !

Plus compliquée est la gestion quand le contenu de la zone doit varier en fonction d'événements extérieurs (dans l'exemple complet donné plus loin, un message est affiché en fonction de la position de la souris). Dans ce cas, il faut à la fois faire le travail soi-même, et renseigner le Window Manager de ce qui a été fait pour qu'il le refasse lui-même en cas de besoin.

Plus compliquée également la gestion quand le contenu de la zone est en interaction avec l'utilisateur (si par exemple elle contient des menus déroulants ou des contrôles). Nous avons pris le parti de ne pas parler de ce point dans cet ouvrage, mais il n'y a plus beaucoup d'efforts à faire quand on a compris l'étape précédente !

• La procédure gérée par le Window Manager

Dès que le Window Manager ressent la nécessité de dessiner la zone d'informations d'une fenêtre, il appelle la procédure dont l'adresse a été passée dans la *ParamList* à la création de la fenêtre. C'est ainsi que le Window Manager assurera la mise à jour de cette zone (fenêtre rendue visible, portion de la zone d'informations masquée par une autre fenêtre et réapparaissant de nouveau, fenêtre redimensionnée, etc.). **NewWindow** appelle cette procédure si la fenêtre est créée visible, et ne l'appelle pas sinon. Donc, quand on ne sait pas ce que contiendra la zone d'informations, avant la création, on ouvrira une fenêtre invisible et on fixera alors, soit la procédure de dessin, soit la valeur qu'elle doit recevoir de l'application, puis on rendra la fenêtre visible.

La procédure de dessin doit être déclarée ainsi :

```
pascal void infoRect(bar, data, wnd)

Rect    *bar;
long    data;
Pointer wnd;

{
    ...        /* instructions de dessin de la zone */
}
```

Quand le Window Manager appellera cette procédure, l'environnement sera un peu spécial : le grafport courant sera le *Window Manager port*, avec un système de coordonnées locales dont l'origine se trouve exactement au coin supérieur gauche de la fenêtre (système de coordonnées permettant de dessiner les contours d'une fenêtre). Le Window Manager donne la valeur des arguments de la procédure. Le premier représentera un pointeur sur le rectangle définissant la zone d'informations, cadre exclu, dans ce système de coordonnées. C'est dans ce rectangle que l'application doit dessiner. Le deuxième argument sera la valeur contenue dans le champ *wInfoRefCon*, qui a été fixé soit dans la *ParamList*, soit au moyen de **SetInfoRefCon**. Le troisième argument désignera la fenêtre à laquelle appartient la zone d'informations.

Grâce au troisième argument, plusieurs fenêtres peuvent se partager la même procédure. Grâce au deuxième, fixé par l'application, n'importe quoi peut être dessiné (une chaîne de caractères, repérée par un pointeur, une barre de menus, repérée par un handle, une image tout en largeur, etc.). Grâce au premier argument, l'application sait où dessiner : dans ce rectangle, et nulle part ailleurs ! Pour éviter des surprises, on peut même fixer une clip region équivalente à ce rectangle, avant de commencer le dessin.

On peut changer les caractéristiques du grapfort avant de commencer à dessiner, mais il est impératif de les rétablir avant de quitter. Dans l'exemple que nous donnons plus loin, du texte est dessiné en style gras. Si nous ne rétablissons pas le style normal, surprise ! Les titres de fenêtres sont à leur tour dessinés en gras, et même les différents contrôles (nous ne l'avons pas dit, mais les cases de fermeture et de zoom, notamment, sont vues par le Window Manager comme des caractères, et non comme des images : il appelle **DrawChar** pour les dessiner !).

Remarque très importante Si cette procédure doit appeler une fonction C normale définie par l'application et qui comporte des arguments, il est impératif de passer les arguments par leur adresse et non par leur valeur, la page directe utilisée par la procédure étant différente de celle utilisée par l'application. Voir comment nous avons passé nos arguments dans l'exemple complet.

• Dessiner directement dans la zone d'informations

La procédure précédente est appelée uniquement quand le Window Manager décide qu'il y a utilité à l'appeler. Il serait donc illusoire de croire que parce qu'on va changer le contenu de *wInfoRefCon,* la zone va être mise à jour en tenant compte de la nouvelle valeur. Et comme il n'y a aucun moyen évident de provoquer son appel, malheureusement, il va falloir « transpirer ».

Pour se mettre dans le bon grafport avec les bonnes coordonnées, le Window Manager nous offre la procédure **StartInfoDrawing**. Deux arguments : l'adresse d'un rectangle dans lequel on va nous retourner le rectangle définissant la zone d'informations (dans le bon système de coordonnées) et un pointeur désignant la fenêtre où on va dessiner.

Pour quitter cet environnement et revenir dans quelque chose de plus sain, on appellera la procédure **EndInfoDrawing**, sans argument.

Entre ces deux appels, l'application dessinera le contenu de sa zone d'informations, comme si elle était dans la procédure appelée par le Window Manager, avec interdiction d'appeler la moindre routine du Window Manager dans cet environnement, sous peine de plantage assuré !

Facile, nous direz-vous. Certes. Mais dessiner directement dans une zone d'informations, c'est comme dessiner directement dans le contenu d'une fenêtre. En cas d'événement de mise à jour, il n'y a plus personne, et on reste devant une portion de fenêtre désespérément blanche. Puisque c'est la procédure appelée par le Window Manager et elle seule qui gère la mise à jour de la zone d'informations, il faut lui faire savoir ce qu'elle aura à redessiner pour être en phase avec l'application. D'où l'idée d'écrire une fonction de dessin unique, appelée à la fois par la procédure automatique et par l'application entre **StartInfoDrawing** et **EndInfoDrawing**. C'est ce que nous avons fait dans l'exemple complet : notre procédure type Pascal infoRect et notre fonction C AjusteInfo appellent toutes les deux la fonction C dessInfo, les arguments étant passés par des adresses, et non par des valeurs. Et le champ *wInfoRefCon* est ajusté en permanence.

• En dehors de toutes ces routines, le Window Manager nous offre un dernier moyen de connaître le rectangle définissant la zone d'informations (toujours dans les coordonnées locales de la fenêtre avec origine en haut à gauche de la région contour) : la procédure **GetRectInfo**. Mêmes arguments que pour **StartInfoDrawing**.

Remarque Dans tout ce que nous venons de dire, si la fenêtre ne possède pas de zone d'informations, le rectangle retourné est vide (ses quatre coordonnées sont à zéro). Il est évident qu'il vaut mieux ne pas essayer de dessiner dans une zone d'informations inexistante !

Quelques routines supplémentaires

• Nous avons dit que le Window Manager dessinait ses fenêtres dans un grafport à lui, le *Window Manager port.* Pour écrire à notre tour dans ce grafport, nous pouvons récupérer un pointeur, grâce à la fonction **GetWMgrPort**, sans argument. Dans ce port, coordonnées locales et coordonnées globales sont égales.

• Il existe une fonction, **Desktop**, que le Window Manager utilise de manière interne pour faire toutes sortes d'opérations concernant le bureau (c'est-à-dire le fond d'écran, là où **FindWindow** retourne *wInDesk*) :

- lui soustraire une région (par exemple la barre de menus système) ;
- lui ajouter une région (après déplacement d'une fenêtre, par exemple) ;
- lui donner un nouveau motif de dessin ; etc.

Nous n'entrerons pas dans les détails, mais il faut bien en toucher un mot, car plusieurs exemples dans cet ouvrage utilisent une forme particulière de cette fonction, justement pour redessiner le desktop. Nous avons employé quatre variantes dans ces exemples :

```
void Paint( );

Desktop(5, Paint);          /* utilise une procédure de dessin */
Desktop(5,0x40000088);      /* utilise l'entrée 8 de la table de couleurs utilisée */
Desktop(5,0x40000134);      /* des points: 3 est la couleur de devant, 4 celle de derrière */
Desktop(5,0x40000234);      /* des lignes: 3 est la couleur de devant, 4 celle de derrière */
```

La première dessinera un fond qui peut représenter n'importe quoi, par exemple une œuvre d'art ou la photo digitalisée de Jean-Louis Gassée, sur lequel viendront s'installer menus et fenêtres. Plus modestement, la deuxième se contentera de donner un fond de couleur uniforme, couleur à choisir parmi les seize possibles (en mode 320) de la palette en cours d'utilisation. La troisième dessinera des points régulièrement disposés (couleur de « devant ») sur un fond uniforme (couleur de « derrière »). La quatrième idem mais avec des lignes horizontales.

• Quand il y a plusieurs fenêtres à l'écran, le Window Manager en garde trace dans une liste. On a vu que la fenêtre du premier plan était récupérable par **FrontWindow**. Pour récupérer les autres, dans l'ordre des plans, on pourra utiliser la fonction **GetNextWindow**. On lui passe un pointeur sur une fenêtre, et elle renvoie un pointeur sur la fenêtre située dans le plan suivant, ou zéro-long s'il n'y a plus de fenêtre derrière.

On peut envoyer une fenêtre derrière une autre fenêtre, si cela se révèle nécessaire, il suffit pour cela de passer deux pointeurs à la procédure **SendBehind** : le second désigne la fenêtre qu'on veut envoyer derrière, le premier désigne la fenêtre derrière laquelle l'autre sera envoyée (la valeur moins-deux-long signifiant dans ce cas derrière toutes les autres fenêtres, au dernier plan). Si on a touché à la fenêtre du premier plan durant cette opération, les événements d'activation adéquats seront générés.

Pour amener une fenêtre au premier plan, on utilisera exclusivement **SelectWindow**. Cette fonction générera correctement les événements d'activation et de désactivation, de telle sorte qu'il y aura toujours une fenêtre (et une seule) « mise en lumière » à l'écran.

• La fonction **PinRect** permet de déterminer quel point appartenant à un rectangle est le plus proche d'un point quelconque. On passe en argument l'abscisse et l'ordonnée du point quelconque, ainsi que le pointeur sur rectangle, et la fonction retourne le point recherché (dans un entier long). Si le point quelconque appartient au rectangle, c'est ce point qui est retourné. Sinon, les règles suivantes sont appliquées (en appelant h et v les coordonnées du point argument et left, top, right et bottom les coordonnées du rectangle) :

- si h < left, elle retourne la coordonnée horizontale left ;
- si h > right, elle retourne la coordonnée horizontale right−1 ;
- si v < top, elle retourne la coordonnée verticale top ;
- si v > bottom, elle retourne la coordonnée verticale bottom−1.

Pour que cela fonctionne, il faut évidemment que le point et le rectangle soient donnés dans le même système de coordonnées.

Cette fonction est idéale pour contraindre un objet quelconque à rester dans un rectangle donné, quelle que soit la position du curseur, à l'intérieur ou à l'extérieur de ce rectangle.

Exemple complet de manipulation multifenêtres

L'exemple suivant est destiné à illustrer un certain nombre de routines du Window Manager. Il fait grandement appel à la gestion des menus déroulants, que nous étudierons dans le chapitre VI.

L'application gère quatre fenêtres qui lui sont propres, et peut ouvrir les fenêtres d'accessoires de bureau (sans gestion du copier – coller).

La fenêtre 0 est la plus complète : elle possède une barre de titre, avec case de fermeture et case de zoom, une zone d'informations, deux barres de défilement et une case de contrôle de taille. Tous ces contrôles sont gérés, à l'exception des barres de défilement, puisque nous n'utilisons pas **TaskMaster**. Le contenu de cette fenêtre est constitué de rectangles multicolores imbriqués. La zone d'informations reçoit un message qui change en fonction de la position de la souris : si la fenêtre est active, la zone d'informations signale si le pointeur se trouve dans sa région contenu, dans sa région contour, dans la barre des menus ou ailleurs hors de la fenêtre ; si la fenêtre est inactive, le message dit simplement que la fenêtre est non active. Dans son principe, la gestion de ces messages s'apparente tout à fait à celle des pointeurs changeant de forme : on mémorise la région précédente, on calcule la région actuelle d'appartenance et si la région a changé, on affiche le nouveau message. La complication provient du fait que l'affichage s'effectue dans la zone d'informations.

La fenêtre 1 possède une barre de titre avec cases de fermeture et de zoom, et une barre de défilement horizontale avec case de contrôle de taille. On remarquera que cliquer dans cette case n'a pas le même effet si la fenêtre est à sa taille maximale ou non ! Dans un cas, elle est considérée vraiment comme une case de contrôle de taille (**FindWindow** retourne *wInGrow*), dans l'autre cas, elle est considérée appartenir au contour, comme la barre de défilement (**FindWindow** retourne *wInFrame*). Est-ce un bogue de la version 1.03 du Window Manager, auquel cas il est possible que cette fenêtre fonctionne parfaitement avec une version future, ou est-ce normal ? Dans tous les cas, le problème est réglé par l'utilisation de **TaskMaster**. Le contenu de cette fenêtre est constitué d'ellipses multicolores imbriquées.

La fenêtre 2 est la fenêtre la plus simple qu'on puisse imaginer : c'est un simple rectangle, sans aucun contrôle. Son contenu est constitué de rectangles arrondis multicolores imbriqués.

La fenêtre 3 illustre le concept de fenêtre d'alerte, qui peut être déplacée n'importe où à partir de sa région contour. Son contenu est constitué d'arcs multicolores imbriqués.

Le champ d'utilisation libre de chaque fenêtre *(wRefCon)* contient le numéro de la fenêtre tel que nous venons de le définir. C'est grâce à ce champ que la fonction qui dessine les contenus saura faire la différence.

Le menu Fichier permet d'ouvrir chacune de ces fenêtres. Dès qu'une fenêtre est ouverte, l'article correspondant dans le menu est estompé, de telle sorte qu'il est impossible de l'ouvrir de nouveau. La commande Fermer de ce menu agit sur la fenêtre du premier plan, indifféremment qu'il s'agisse d'un accessoire ou d'une des fenêtres précédentes (dans ce dernier cas, l'article permettant de l'ouvrir est de nouveau activé).

Le menu Visibles gère la visibilité des fenêtres : il suffit de cocher un article pour rendre visible la fenêtre correspondante, de retirer la marque pour la rendre invisible. Les fenêtres non ouvertes ont leur article estompé de telle sorte qu'on ne peut agir sur leur état visible ou non !

Le menu Active permet de rendre active la fenêtre de son choix, y compris les fenêtres invisibles (mais pas les fenêtres fermées). Jouer avec ces deux menus en apprend plus que cinquante lignes de cet ouvrage !

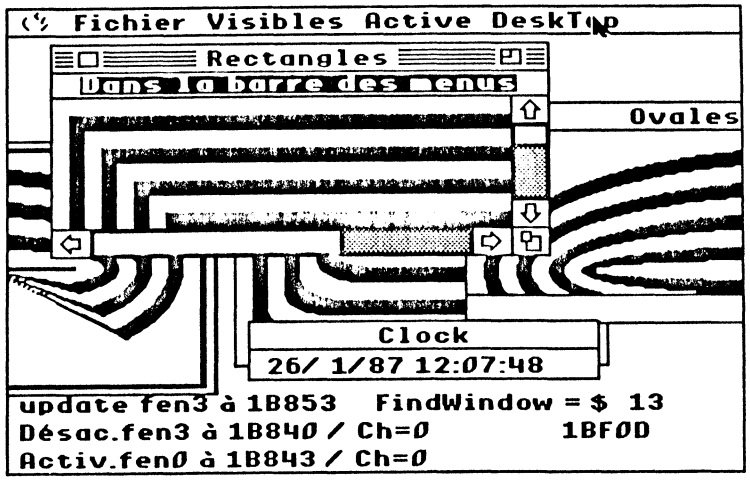


Figure V.13. Ecran tiré de l'exemple.

Enfin le menu DeskTop agit sur la représentation du fond d'écran. On peut l'effacer (il devient tout blanc), jouer avec les différentes options (couleurs solides, points, lignes) et enfin en faire un magnifique paysage, dont les couleurs sont plus réelles en mode 640 qu'en mode 320. Notons que le fichier sur disquette qui contient cette image s'appelle Fondecran et doit se trouver dans le même dossier que l'application qui l'utilise (notion de préfixe 1 gérée par ProDOS). N'importe quelle autre image fera l'affaire, pourvu qu'elle porte ce même nom. En particulier en mode 320, toute image d'écran créée par GS-Paint.

Les événements de type fenêtre sont affichés directement sur le desktop. Un compteur de temps défile en permanence. Le message précise le type d'événement, la fenêtre touchée et la date relative à laquelle il intervient. Pour les événements d'activation et de désactivation, l'état du bit *ChangeFlag* est donné (0 s'il est à zéro, 2 s'il est positionné). On verra ainsi qu'il ne fonctionne pas comme annoncé dans l'Event Manager !

```
#include <tools.h>              /* définition des termes en gras */
#include <entete.h>             /* définition des termes en italique */

#define mode 1                  /* 0 si mode 320, 1 si mode 640 */
                                        /* déclaration des menus déroulants */
char Menu1[ ] = "> @\\XN1";
char Menu11[ ] = "- A propos de fenêtres...\\N257VD";
char Menu19[ ] = "- ";
char Menu2[ ] = "> Fichier \\N2";
char Menu21[ ] = "- Ouvrir Rect\\N260*0 ";
char Menu22[ ] = "- Ouvrir Oval\\N261*1 ";
char Menu23[ ] = "- Ouvrir RRect\\N262*2 ";
char Menu24[ ] = "- Ouvrir Arc\\N263V*3 ";
char Menu25[ ] = "- Fermer fenêtre\\N265V*Ff";
char Menu26[ ] = "- Quitter\\N266*Qq";
char Menu29[ ] = "- ";
char Menu3[ ] = "> Visibles \\N3";
char Menu31[ ] = "- Rectangles\\N300D";
char Menu32[ ] = "- Ovales\\N301D";
char Menu33[ ] = "- Rect. arr.\\N302D";
char Menu34[ ] = "- Arcs\\N303D";
char Menu39[ ] = "- ";
char Menu4[ ] = "> Active \\N4";
```

```
char Menu41[ ] = "- Rectangles\\N310D";
char Menu42[ ] = "- Ovales\\N311D";
char Menu43[ ] = "- Rect. arr.\\N312D";
char Menu44[ ] = "- Arcs\\N313D";
char Menu49[ ] = ". ";
char Menu5[ ] = "> DeskTop \\N5";
char Menu51[ ] = "- Effacer\\N321V";
char Menu52[ ] = "- Solide\\N322";
char Menu53[ ] = "- Points\\N323";
char Menu54[ ] = "- Lignes\\N324";
char Menu55[ ] = "- Dessin\\N325";
char Menu59[ ] = ". ";
                                        /* définition de quelques constantes */
#define iOuvrir     260
#define iFermer     265
#define iQuitter    266
#define iVisibles   300
#define iActive     310
#define iEffacer    321
#define iSolide     322
#define iPoints     323
#define iLignes     324
#define iDessin     325
                                        /* les messages à afficher dans la zone d'information */
char    Mess0[ ] = "Fenêtre non active";
char    Mess1[ ] = "Dans la région contenu";
char    Mess2[ ] = "Dans la région contour";
char    Mess3[ ] = "Dans la barre des menus";
char    Mess4[ ] = "Hors de la fenêtre";

int         colfen[ ] = {0,0x0F00,0x020F,0xF0F0,0x00F0};    /* couleurs des fenêtres */
void        infoRect( );                                    /* déclaration d'une fonction */
ParamList   maFen0 =                    /* définition de la fenêtre contenant les rectangles */
        { sizeof(ParamList), 0xDDB0, "\14 Rectangles ", 0L,
        {40,2,150,302+300*mode}, colfen, 0, 0,
        200, 300+500*mode, 0, 0,
        4, 10, 40, 100,
        0L, 12, 0L, infoRect, 0L,
        {40,20,130, 196+250*mode}, -1L, 0L };

ParamList   maFen1 =                    /* définition de la fenêtre contenant les ovales */
        { sizeof(ParamList), 0xCDA0, "\10 Ovales ", 1L,
        {30, 10,150, 310+250*mode}, colfen, 0, 0,
        120, 300+250*mode, 120, 300+250*mode,
        4, 10, 40, 100,
        0L, 0, 0L, 0L, 0L,
        {50,40,120, 230+250*mode}, -1L, 0L };

ParamList   maFen2 =            /* définition de la fenêtre contenant les rectangles arrondis */
        { sizeof(ParamList), 0x0020, "", 2L,
        {0,0,0,0}, 0L, 0, 0,
        100,160+250*mode,0,0,
        0,0,0,0,
        0L, 0, 0L, 0L, 0L,
        {55,100,155, 260+250*mode}, -1L, 0L };

ParamList   maFen3 =                    /* définition de la fenêtre contenant les arcs */
        { sizeof(ParamList), 0x20A0, "", 3L,
        {0,0,0,0}, 0L, 0, 0,
        100,180+250*mode,0,0,
        0,0,0,0,
        0L, 0, 0L, 0L, 0L,
        {60,120,160, 300+250*mode}, -1L, 0L };
```



```
void        Paint( );                   /* dessinera le desktop */
Handle      hdl;                        /* handle sur l'image du desktop */
Handle      rgncont;                    /* région contenu de la fenêtre 0 */
Handle      rgnstruc;                   /* région structure de la fenêtre 0 */
Rect        rectMenu;                   /* rectangle contenant la barre des menus */
TaskRec     tache;                      /* ce que manipule GetNextEvent */
Pointer     fen[4] = {0L,0L,0L,0L};     /* pointeurs sur fenêtre */
Pointer     wind;                       /* la fenêtre courante */
Pointer     defPort;                    /* le grafport par défaut */
int         hMBar;                      /* hauteur de la barre des menus */
int         col, col1, col2;            /* couleurs du desktop */
int         indic = TRUE;               /* indicateur de fin de boucle */

            /***** PROGRAMME PRINCIPAL *****/

    main( )
{
int     myID;                           /* identifiant de l'application */
char    msg[10];

    myID = debut_appl(mode);
    hdl = NewHandle(0x8000L, myID, 0x0010, 0L);     /* allocation d'un handle de 32K */
    Load(hdl,"1/Fondecran");            /* chargement de l'image fond d'écran */
    PlaceMenus( );                      /* installe la barre des menus */
    FlushEvents(EveryEvent, 0);         /* fait le vide dans la file d'événements */
    defPort = GetPort( );               /* sauvegarde le grafport par défaut */

    do {
        AjusteInfo( );                  /* ajuste le texte dans la zone d'information */
        SystemTask( );                  /* pour les accessoires de bureau périodiques */
        SetPort(defPort);               /* on va écrire directement sur le desktop */
        sprintf(msg,"%lx",TickCount( ));    /* on écrit le compteur de temps */
        MoveTo(240,187); DrawCString(msg);
        if(!GetNextEvent(EveryEvent, &tache)) continue;

        switch(tache.what)              /* quel événement à traiter? */
            {
            case MouseDown :            /* un clic souris */
                sourisDans(FindWindow(&wind, tache.where));
                break;

            case KeyDown:               /* une touche enfoncée */
                if (tache.modifiers & AppleKey)     /* touche Pomme enfoncée */
                    {
                    MenuKey(&tache, 0L);            /* on invoque un menu déroulant... */
                    indic = ExecMenu(tache.TaskData);   /* ...voir chapitre VI */
                    }
                break;

            case UpdateEvt :            /* une fenêtre à mettre à jour */
                majFen(tache.message);

                break;

            case ActivateEvt :          /* une fenêtre à activer ou à désactiver */
                actFen(tache.message);
                break;
            }
        }
    while(indic);

    quitter(myID);                      /* on quitte l'application de manière standard */
}
```

```
        /***** FONCTION PLACEMENUS: installe la barre des menus *****/

    PlaceMenus( )

{
    InsertMenu(NewMenu(Menu5), 0);          /* installation des divers menus */
    InsertMenu(NewMenu(Menu4), 0);
    InsertMenu(NewMenu(Menu3), 0);
    InsertMenu(NewMenu(Menu2), 0);
    InsertMenu(NewMenu(Menu1), 0);
    FixAppleMenu(1);                        /* ajout des accessoires de bureau */
    hMBar = FixMenuBar( );                  /* calcul des dimensions de la barre */
    SetRect(&rectMenu, 0, 0, 1000, hMBar);  /* ce rectangle contient la barre des menus */
    DrawMenuBar( );                         /* dessin de la barre */
}

        /***** FONCTION EXECMENU: répond au choix d'un article de menu *****/

    int ExecMenu(art, menu)             /* retourne FALSE si quitter est choisi */

int  art;           /* article choisi */
int  menu;          /* dans ce menu */

{
int     mark;
char    msg[30];
int     ind;
                                        /* voir le chapitre VI pour comprendre cette fonction */
    if (art>255) switch (art)
        {
        case iOuvrir:
        case iOuvrir+1:
        case iOuvrir+2:
        case iOuvrir+3:
            ouvre(art-iOuvrir);             /* ouverture d'une fenêtre */
            break;

        case iFermer:
            ferme(FrontWindow( ));          /* fermeture de la fenêtre de premier plan */
            break;

        case iQuitter:
            return FALSE;                   /* signale la fin de l'application */
            break;

        case iVisibles:                     /* gestion des articles du menu Visibles */
        case iVisibles+1:
        case iVisibles+2:
        case iVisibles+3:
            ind = art - iVisibles;
            mark = GetItemMark(art);            /* l'article porte-t-il une marque? */
            if (mark) HideWindow(fen[ind]);     /* oui, on rend la fenêtre invisible */
            else ShowWindow(fen[ind]);          /* non, on rend la fenêtre visible */
            CheckMItem(!mark, art);             /* on met ou on enlève la marque */
            CheckMItem(FALSE, iActive+ind); /* on enlève la marque dans le menu Active */
            break;


        case iActive:                       /* gestion des articles du menu Active */
        case iActive+1:
        case iActive+2:
        case iActive+3:
            ind = art - iActive;
            if (fen[ind] != FrontWindow( ))     /* si la fenêtre n'est pas déjà active... */
```



```
                SelectWindow(fen[ind]);         /* ...on la rend active */
            break;

        case iEffacer:
            Desktop(5,0x400000FF);              /* le fond d'écran devient tout blanc */
            break;

        case iSolide:
            col = (col+1) %16;
            Desktop(5,0x40000000+col*0x11); /* le fond d'écran prend une couleur solide */
            break;

        case iPoints:
            col1 = (col1+1) %16;
            col2 = (col2-1) %16;
            Desktop(5,0x40000100+col1*0x10+col2); /* fond d'écran: un nuage de points */
            break;

        case iLignes:
            col1 = (col1+1) %16;
            col2 = (col2-1) %16;
            Desktop(5,0x40000200+col1*0x10+col2); /* fond d'écran: rayé horizontalement */
            break;

        case iDessin:
            Desktop(5,Paint);                   /* le fond d'écran est un véritable dessin */
            break;
        }

    else if (art>0) OpenNDA(art);       /* ouverture accessoire de bureau */

    if (art) HiliteMenu(FALSE, menu);
    return TRUE;
}


        /***** FONCTION OUVRE: ouverture d'une fenêtre appartenant à l'application *****/

    ouvre(ind)

int     ind;

{
    if (fen[ind] != 0L) return;     /* la fenêtre est déjà ouverte (test normalement inutile) */
    else
        {
        if (ind == 0)                       /* fenêtre contenant les rectangles */
            {
            fen[ind] = NewWindow(&maFen0);          /* on l'ouvre et on mémorise... */
            rgncont = GetContRgn(fen[ind]);         /* ...sa région contenu */
            rgnstruc = GetStructRgn(fen[ind]);      /* ...et sa région contour */
            }
        else if (ind == 1) fen[ind] = NewWindow(&maFen1);       /* ouverture fenêtre 1 */
        else if (ind == 2) fen[ind] = NewWindow(&maFen2);       /* ouverture fenêtre 2 */
        else if (ind == 3) fen[ind] = NewWindow(&maFen3);       /* ouverture fenêtre 3 */
        EnableMItem(iActive + ind); /* l'article correspondant (menu Active) rendu actif */
        EnableMItem(iVisibles + ind); /* l'article correspondant (menu Visibles) rendu actif */
        CheckMItem(TRUE, iVisibles + ind);      /* la fenêtre est marquée visible dans le menu */
        DisableMItem(iOuvrir + ind); /* l'article permettant l'ouverture de la fenêtre estompé */
        }
}

/***** FONCTION FERME: fermeture d'une fenêtre (application ou accessoire de bureau) *****/
```

```
    ferme(port)

Pointer     port;               /* pointeur sur la fenêtre à fermer */

{
int     ind;

    if (port == 0L) return;             /* rien à fermer! */
    else if (GetWKind(port)) CloseNDAbyWinPtr(port);  /* fermeture fenêtre système */
    else
        {
        ind = (int) GetWRefCon(port);           /* de quelle fenêtre s'agit-il? */
        CheckMItem(FALSE, iVisibles + ind); /* on retire la marque disant qu'elle est visible... *
        DisableMItem(iVisibles + ind);          /* ...et on estompe l'article */
        CheckMItem(FALSE, iActive + ind);  /* on retire la marque disant qu'elle est active... */
        DisableMItem(iActive + ind);            /* ...et on estompe l'article */
        EnableMItem(iOuvrir + ind);             /* on rétablit la possibilité d'ouvrir la fenêtre... *
        fen[ind] = 0L;                          /* ...on perd sa trace... */
        CloseWindow(port);                      /* ...et on la ferme */
        }
}

        /***** FONCTION MAJFEN: mise à jour du contenu d'une fenêtre *****/

    majFen(port)

Pointer     port;               /* pointeur sur la fenêtre à rafraîchir */

{
int     num;
char    msg[30];

    SetPort(defPort);                       /* on va écrire sur le desktop */
    num = (int) GetWRefCon(port);           /* on repère le numéro de la fenêtre... */
    sprintf(msg,"update fen%d à %lx",num,tache.when);
    MoveTo(5,175); DrawCString(msg); /* ...et on affiche le message de mise à jour */
    BeginUpdate(port);                      /* début de la mise à jour */
    SetPort(port);                          /* on va dessiner dans la bonne fenêtre */
    dessine(port, num);
    EndUpdate(port);                        /* fin de la mise à jour */
}

        /***** FONCTION ACTFEN: activation ou désactivation d'une fenêtre *****/

    actFen(port)

Pointer     port;               /* pointeur sur la fenêtre à activer ou désactiver */

{
int     ind;
char    msg[50];

    SetPort(defPort);                       /* on va écrire sur le desktop */
    ind = (int) GetWRefCon(port);           /* on repère le numéro de la fenêtre */
    if (tache.modifiers & ActiveFlag)               /* si activation... */
        {
        CheckMItem(TRUE, iActive + ind);    /* ...fenêtre cochée (elle devient active!) */
        sprintf(msg, "Activ.fen%d à %lx / Ch=%d",
                                ind, tache.when, tache.modifiers & ChangeFlag);
        MoveTo(5,199); DrawCString(msg);            /* message d'activation */
        }
    else                                            /* si désactivation... */
        {
        CheckMItem(FALSE, iActive + ind); /* ...fenêtre non cochée (elle devient inactive!) */
        sprintf(msg, "Désac.fen%d à %lx / Ch=%d",
                                ind, tache.when, tache.modifiers & ChangeFlag);
        MoveTo(5,187); DrawCString(msg);            /* message de désactivation */
        }
}

        /***** FONCTION DESSINE: dessine le contenu des fenêtres *****/

    dessine(por, n)

Pointer     por;                /* pointeur sur la fenêtre à dessiner */
int         n;                  /* n est le type du dessin */

{
Rect    r;                      /* rectangle */
long    dataSize;
int     i = 1;

    dataSize = GetDataSize(por);            /* la taille des données */
    SetRect(&r, 0, 0, dataSize);            /* attention au raccourci! */
    while(EmptyRect(&r))            /* on dessine tant que le rectangle n'est pas vide */
        {
        SetSolidPenPat(i);                      /* on change la couleur du crayon */
        if (n == 0) PaintRect(&r);              /* on paint un rectangle... */
        else if (n == 1) PaintOval(&r);         /* ...ou un ovale... */
        else if (n == 2) PaintRRect(&r,40,20);  /* ...ou un rectangle arrondi... */
        else PaintArc(&r, -135, 270);           /* ...ou un arc */
        i = (i+1)%16;                   /* la couleur sera différente à l'étape suivante */
        InsetRect(&r, 7*(mode+1), 7);  /* le rectangle sera plus petit à l'étape suivante */
        }
}

        /***** FONCTION SOURISDANS: réponse à un clic souris *****/

    sourisDans(code)

int     code;               /* code retourné par FindWindow */

{
char    msg[20];

    SetPort(defPort);                       /* on va écrire sur le desktop... */
    sprintf(msg, "FindWindow = $%4x  ", code);
    MoveTo(160,175); DrawCString(msg);      /* ...le code retourné par FindWindow */

    if (code<0) SystemClick(&tache, wind, code);    /* gestion d'un accessoire de bureau */
    else switch (code)

        {
        case wInMenuBar :                           /* gestion des menus déroulants */
            MenuSelect(&tache, 0L);
            indic = ExecMenu(tache.TaskData);
            break;

        case wInContent :                           /* clic dans le contenu d'une fenêtre */
            if (wind != FrontWindow( )) SelectWindow(wind);
            break;                                  /* on la sélectionne, si nécessaire */

        case wInDrag :                              /* la fenêtre va changer de position */
            if (wind != FrontWindow( ) && !(tache.modifiers & AppleKey))
                SelectWindow(wind);  /* elle n'est pas sélectionnée si Pomme est enfoncée */
            DragWindow(0, tache.where, 0, 0L, wind);
            break;
```

```
        case wInGrow :                          /* la fenêtre va changer de taille */
            if (wind != FrontWindow( )) SelectWindow(wind);
            else
            SizeWindow(GrowWindow(50*(mode+1), 50, tache.where, wind), wind);
            break;

        case wInGoAway :                        /* la fenêtre va être fermée */
            if (TrackGoAway(tache.where, wind)) ferme(wind);
            break;

        case wInZoom :                          /* la fenêtre va être zoomée */
            if (TrackZoom(tache.where, wind)) ZoomWindow(wind);
            break;

        case wInInfo :          /* clic dans la zone d'information, rien de bien fantastique! */
            if (wind != FrontWindow( )) SelectWindow(wind);
            break;

        case wInFrame :
            if (wind != FrontWindow( )) SelectWindow(wind);
            else ;                      /* frustration suprême: on ne gère pas le défilement! */
            break;
        }
}

/***** FONCTION AJUSTINFO: ajuste la zone d'information
                                        en fonction de la position du curseur *****/

    AjusteInfo( )

{
static Pointer  AncMess;                /* le précédent message géré */
Pointer     Mess;                       /* le message à gérer */
long        var1,var2;                  /* deux réservations de place */
long        pt;                         /* un point déguisé en entier long */
Rect        r;                          /* un rectangle */

    if (FrontWindow( ) == 0L || fen[0] == 0L)
        return;                         /* la fenêtre 0 n'est pas ouverte, rien à faire! */
    if (!(GetWFrame(fen[0]) & 0x0020)) return;  /* idem si la fenêtre est invisible! */
    if (FrontWindow( ) != fen[0]) Mess = Mess0; /* la fenêtre 0 n'est pas active */
    else
        {
        SetPort(fen[0]);                /* on fixe le bon grafport */
        GetMouse(&pt);                  /* position du pointeur, coordonnées locales... */
        LocalToGlobal(&pt);             /* ...traduites en coordonnées globales */
        if (PtInRgn(&pt, rgncont)) Mess = Mess1;        /* pointeur dans la région contenu */
        else if (PtInRgn(&pt, rgnstruc)) Mess = Mess2; /* pointeur dans la région structure */
        else if (PtInRect(&pt, &rectMenu))
            Mess = Mess3;               /* pointeur dans la barre des menus */
        else Mess = Mess4;                      /* pointeur ailleurs */
        }
    if (Mess == AncMess) return;        /* le pointeur n'a pas changé de région, on ne fait rien */
    SetInfoRefCon(Mess, fen[0]); /* on fixe la valeur à passer à la procédure gérée par le WM */
    StartInfoDrawing(&r, fen[0]); /* on va dessiner directement dans la zone d'information */
    var1 = (long) &r; var2 = (long) Mess;
    dessInfo(&var1, &var2);     /* on passe des adresses, et non directement des valeurs */
    EndInfoDrawing( );          /* on a fini de dessiner dans la zone d'information */
    AncMess = Mess;
}

/***** PROCEDURE INFORECT: appelée par le Window Manager
                                        pour dessiner la zone d'information *****/
```



```
    pascal void infoRect(bar, data, wnd)

long    bar, data, wnd;
{
    dessInfo(&bar, &data);      /* on passe des adresses, pas des valeurs! */
}

/***** FONCTION DESSINFO: dessine la zone d'information *****/

    dessInfo(pr, mess)

Rect    ** pr;                  /* adresse d'un pointeur sur rectangle */
long    *mess;                  /* adresse d'un pointeur sur chaîne de caractères */

{
Rect    r;
int     x,y,PL,PH,GL,GH;        /* intermédiaires de calcul pour le centrage du texte */
char    msg[40];

    EraseRect(*pr);                         /* on efface la zone d'information */
    if (*mess == 0L) return;                /* s'il n'y a rien à écrire, on ne va pas plus loin */
    SetForeColor(mode ? 15 : 9);            /* couleur des caractères */
    SetBackColor(mode ? 0 : 4);             /* couleur du fond des caractères */
    SetTextFace(1);                         /* on écrira en gras */
    MoveTo((*pr)->left, (*pr)->top);    /* le crayon est placé en haut à gauche de la zone */
    CStringBounds(*mess,&r);
    GL = (*pr)->right - (*pr)->left;
    GH = (*pr)->bottom - (*pr)->top;                /* voir le chapitre III... */
    x = r.left - (*pr)->left;               /* ...où ces calculs sont expliqués */
    y = r.top - (*pr)->top;
    PL = r.right - r.left;
    PH = r.bottom - r.top;
    Move((GL-PL)/2-x, (GH-PH)/2-y);     /* le crayon est placé de telle sorte
                                            que le texte est centré */
    DrawCString(*mess);                 /* le message est dessiné */
    SetForeColor(0);                    /* les caractères seront de nouveau noirs... */
    SetBackColor(15);                   /* ...sur fond blanc... */
    SetTextFace(0);                     /* ...et de style normal */
}

        /***** FONCTION LOAD: lit un fichier et charge le bloc repéré par un handle *****/

    Load(PicDest, path)

Handle      PicDest;                    /* handle (déjà alloué) sur les données */
Pointer     path;                       /* chemin d'accès aux données sur disque */

{
int         id;                         /* identifiant fichier */

    HLock(PicDest);                     /* le bloc est verrouillé */
    id = open(path,4);                  /* open est la fonction C d'ouverture de fichier */
    read(id, *PicDest, 0x8000);         /* read est la fonction C de lecture de fichier */
    close(id);                          /* close est la fonction C de fermeture de fichier */
    HUnlock(PicDest);                   /* le bloc est déverrouillé */
}

    /***** PROCEDURE PAINT: sera appelée par le Window Manager
                                                    pour dessiner le desktop *****/

    pascal void Paint( )

{
```

```
    PaintDesktop(&hdl, &hMBar);        /* on passe des adresses, pas des valeurs */
}
        /***** FONCTION PAINTDESKTOP: dessine le desktop *****/

    PaintDesktop(adhdl, hMB)

Handle      *adhdl;             /* adhdl pointe sur un handle */
int         *hMB;  /* adresse de la variable contenant la hauteur de la barre des menus */

{
LocInfo     source;             /* une pixel image et son environnement */
Rect        r;                  /* le rectangle de destination */

    HLock(*adhdl);              /* le bloc est verrouillé pendant le transfert */
    source.PortSCB = mode << 7;             /* le mode de résolution (0 ou $80) */
    source.baseAddr = **adhdl;              /* l'adresse de la pixel image */
    source.rowBytes = 160;          /* 160 octets par ligne = écran complet en largeur */
    SetRect(source.BoundsRect, 0, 0, 320*(mode+1), 200);    /* écran complet */
    SetRect(&r, 0, -(*hMB), 320*(mode+1), 187);         /* décalage pour tenir compte
                                                           de la barre des menus */
    PPToPort(&source, &r, 0, 0, 0);     /* transfert de l'image dans le grafport courant */

    HUnlock(*adhdl);            /* maintenant, le bloc peut être déverrouillé */
}
```

Exemple complet des coordonnées globales et locales

Cet exemple ouvre deux fenêtres à l'écran, dans lesquelles on ne peut rien faire. Quand le pointeur est dans la zone contenu de la fenêtre active, il prend la forme d'une petite croix. Sinon il garde sa forme habituelle de flèche. Dans la barre de menus (vide de menus), on affiche en permanence la position de la souris, à gauche dans le système de coordonnées globales (origine en haut à gauche de l'écran), à droite dans le système de coordonnées locales (origine en haut à gauche du rectangle contenu de la fenêtre active, puisque nous veillons à ce que le grafport courant soit celui associé à la fenêtre active au moment où nous récupérons les coordonnées du pointeur).

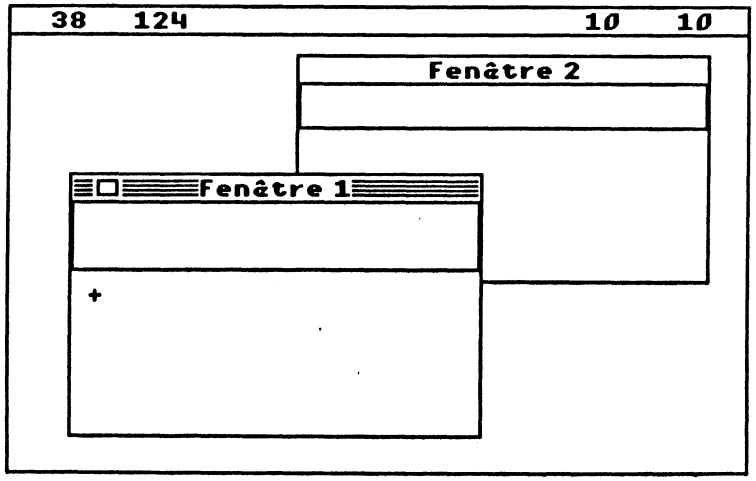


Figure V.14. Ecran tiré de l'exemple.

Amusez-vous à activer une fenêtre puis l'autre : les coordonnées globales ne sont pas affectées, les coordonnées locales oui. On peut déplacer les fenêtres. Faites sortir la fenêtre active sur la gauche du bureau, afin de voir l'abscisse des coordonnées locales supérieure à celle des coordonnées globales... On peut fermer les fenêtres. Quand il n'y en a plus, les coordonnées locales sont prises à partir du *Window Manager port*, et on peut constater qu'elles sont égales aux coordonnées globales : l'origine des deux systèmes coïncide.

A titre de curiosité, le bureau est multicolore. Au lieu de notre procédure de dessin simpliste, n'importe quelle image ferait l'affaire ! Notez également la façon d'écrire dans la barre des menus, nous referons cet exercice de style dans le chapitre consacré au Dialog Manager !

Voir les remarques données dans l'exemple complet du chapitre consacré à l'Event Manager pour compiler cet exemple. La fonction AjusteCurs a changé, mais sa philosophie est restée la même. La fonction getbits par contre est identique.

```
#include <tools.h>                          /* contient la définition des termes en gras */
#include <entete.h>                         /* contient la définition des termes en italique */
#define mode 0                              /* 0 pour mode 320, 1 pour mode 640 */

void Info( );                               /* déclaration procédure de dessin zone info */
void Paint( );                              /* déclaration procédure de dessin du bureau */
                                                    /* fenêtre 1 */
ParamList maFen1 = {
    sizeof(ParamList), 0xC0B0, "\11Fenêtre 1", 1L,
    {56, 2,187, 302+320*mode}, 0L, 0, 0,
    161, 300+320*mode, 161, 300+320*mode,
    4, 16, 40, 160,
    0L, 30, 0L, Info, 0L,
    {60,20,130, 196+320*mode}, -1L, 0L };
                                                    /* fenêtre 2 */
ParamList maFen2 = {
    sizeof(ParamList), 0xC0B0, "\11Fenêtre 2", 2L,
    {46, 2,187, 302+320*mode}, 0L, 0, 0,
    161, 300+320*mode, 161, 300+320*mode,
    4, 16, 40, 160,
    0L, 20, 0L, Info, 0L,
    {80,40,145, 216+320*mode}, -1L, 0L };
                                    /* curseur en forme de croix  (définition non structurée) */
char croix[ ] = { 5,0,3,0,                  /* 5 lignes de 3 mots */
    0,0xF0,0,0,0,0,                         /* image */
    0,0xF0,0,0,0,0,
    0xFF,0xFF,0xF0,0,0,0,
    0,0xF0,0,0,0,0,
    0,0xF0,0,0,0,0,
    0,0,0,0,0,0,                            /* masque */
    0,0,0,0,0,0,
    0,0,0,0,0,0,
    0,0,0,0,0,0,
    0,0,0,0,0,0,
    2,0,2,0 } ;                             /* point chaud */

int     colfen[ ] = {0,0x0F00,0x020F,0xF0F0,0x00F0}; /* table de couleurs pour fenêtres */
TaskRec tache;                                  /* ce que manipule GetNextEvent */
Pointer fen1, fen2, wind;                       /* pointeurs sur fenêtre */
int     indic = TRUE;                           /* indicateur de fin de boucle */
Pointer menuPort;                       /* adresse du Menu Manager port */
Handle  cont;                           /* handle sur la région contenu de la fenêtre active */
Pointer arrow;                          /* pointeur sur le curseur en forme de flèche */

        /***** PROGRAMME PRINCIPAL *****/
```

```
    main()
{
int     x;                          /* code retourné par GetNextEvent */
int     myID;                       /* identifiant de l'application */

    myID = debut_appl(mode);            /* initialisations diverses */
    SetFrameColor(colfen,0L); /* couleurs par défaut pour toutes les fenêtres */
    Desktop(5,Paint);                   /* une procédure de dessin pour le bureau */
    FlushEvents(EveryEvent, 0);         /* plus d'événement dans la file */
    fen1 = NewWindow(&maFen1);          /* création de la première fenêtre */
    fen2 = NewWindow(&maFen2);          /* création de la seconde fenêtre */
    menuPort = GetMenuMgrPort();    /* le port dans lequel les menus sont dessinés... */
    SetPort(menuPort);                  /* ...devient le port par défaut... */
    SetTextMode(0);                     /* ...avec mode de transfert Copy */
    arrow = GetCursorAdr( );            /* on garde l'adresse du curseur système */

    do {
        x = GetNextEvent(EveryEvent, &tache);       /* l'événement suivant */
        AffichCoord( );             /* affichage des coordonnées */
        AjusteCurs( );              /* dessin du curseur en fonction de sa position */
        if(!x) continue;            /* pas d'événement significatif à traiter */

        switch(tache.what)
            {
            case MouseDown :
                sourisDans(FindWindow(&wind, tache.where)); /* réponse à un clic souris */
                break;

            case KeyDown :
                indic = FALSE ;     /* une touche enfoncée, on sort! */
                break;

            case UpdateEvt :        /* on ne traite pas les événements de mise à jour */
                break;

            case ActivateEvt :              /* activation ou désactivation de fenêtre */
                if(tache.modifiers & ActiveFlag)    /* si c'est une activation... */
                cont = GetContRgn(tache.message); /* ...on calcule la région contenu... */
                break;                      /* ...de la nouvelle fenêtre active */
            }
        }
    while(indic);                   /* fin de la boucle d'événement */

    quitter(myID);                  /* on quitte l'application */
}

            /***** FONCTION SOURISDANS: réponse à un clic souris *****/

    sourisDans(code)

int     code;                       /* code retourné par FindWindow */

{
    switch(code)
        {
        case wInContent :               /* dans le contenu d'une fenêtre... */
            if (wind != FrontWindow( ))
                SelectWindow(wind);     /* ...on la sélectionne si nécessaire */
            break;

        case wInDrag :                  /* dans la barre de titre */
            if (wind != FrontWindow( ) && !(tache.modifiers & AppleKey))
                SelectWindow(wind);     /* on sélectionne si la touche
                                           Pomme n'est pas enfoncée... */
```



```
            DragWindow(0, tache.where, 0, 0L, wind);  /* ...et on déplace la fenêtre */
            break;

        case wInGoAway :                /* dans la case de fermeture */
            CloseWindow(wind);          /* on ferme la fenêtre courante */
            break;

        case wInInfo :              /* dans zone d'information: comme dans contenu */
            if (wind != FrontWindow( )) SelectWindow(wind);
            break;
        }
}

/***** FONCTION AFFICHCOORD: affichage des coordonnées dans la barre de menus *****/

    AffichCoord( )
{
long    pt;                         /* point déclaré comme un entier long */
char    msg[10];

    if (FrontWindow( ) != 0L)       /* s'il reste une fenêtre ouverte... */
    SetPort(FrontWindow( ));    /* ...on rend actif le port de la fenêtre du premier plan... */
    else SetPort(GetWMgrPort( ));   /* ...sinon le port du Window Manager */
    GetMouse(&pt);              /* on récupère les coordonnées du pointeur dans ce port */
    SetPort(menuPort);                  /* on rétablit le port du Menu Manager */
    sprintf(msg,"%4d ",getbits(pt,31,16));      /* on écrit l'abscisse... */
    MoveTo(240,10); DrawCString(msg);
    sprintf(msg,"%4d ",getbits(pt,15,16));      /* ...et l'ordonnée des coordonnées locales */
    MoveTo(280,10); DrawCString(msg);

    sprintf(msg,"%4d ",getbits(tache.where,31,16));     /* on écrit l'abscisse... */
    MoveTo(10,10); DrawCString(msg);
    sprintf(msg,"%4d ",getbits(tache.where,15,16));     /* ...et l'ordonnée
                                                           des coordonnées globales */
    MoveTo(50,10); DrawCString(msg);
}

/***** FONCTION AJUSTCURS: change la forme du pointeur en fonction de sa position *****/

    AjusteCurs( )
{
static  modif;                          /* l'état précédent */
int     ind;                            /* l'état courant */

    ind = PtInRgn(&tache.where,cont); /* TRUE si le pointeur est dans la région contenu */
    if (ind == modif) return;       /* pas eu de changement? On quitte directement */
    if (ind) SetCursor(croix);      /* sinon on change le curseur... */
    else SetCursor(arrow);
    modif = ind;                    /* ...et on mémorise le nouvel état */
}


        /***** PROCEDURE PAINT: définit le dessin du bureau *****/

    pascal void Paint( )
{
int     i;
Rect    r;

    SetRect(&r,-20,0,0,200);
    for(i=0; i<16*(mode+1); ++i)
        {
        OffsetRect(&r,20,0);
        SetSolidPenPat(i);
```

```
        PaintRect(&r);
        }
}

        /***** PROCEDURE INFO: définit le dessin de la zone d'information *****/

    pascal void Info(bar,data,wnd)
long    bar,data,wnd;

{
/* on ne dessine rien dans la zone d'information */
}

            /***** FONCTION GETBITS *****/

    getbits(x,p,n)              /* prend n bits à partir de la position p dans un entier long */

unsigned long  x;
unsigned int   p,n;                     /* p peut prendre les valeurs 31,30,...,1,0 */
                                        /* n doit être compris entre 1 et 16 */
{
    return( (x>>(p+1-n)) & ~(~0<<n) );          /* retourne un entier sur 16 bits */
}
```

CHAPITRE VI

MENU MANAGER

PRINCIPES GÉNÉRAUX

Les menus déroulants constituent l'une des fondations de l'interface utilisateur Apple. Ils sont caractérisés par une *barre des menus* qui contient le titre de chacun des menus. Quand le bouton de la souris est pressé sur l'un de ces titres, le titre s'inverse et le menu déroulé apparaît avec toutes ses options : les *articles*. Il restera visible tant que le bouton ne sera pas lâché. Il suffit alors de faire glisser la souris sur les différents articles pour opérer une sélection, et de lâcher le bouton pour lancer la commande : l'article sélectionné clignote et le menu disparaît. Le titre du menu reste inversé jusqu'à ce que l'exécution de la commande soit terminée. Si aucun article n'était sélectionné au moment du lâcher du bouton (pointeur en dehors du rectangle du menu déroulé ou au-dessus d'un article estompé), le menu disparaît et il ne se passe rien. De la sorte, l'utilisateur peut à tout instant consulter les menus, sans conséquence aucune sur le déroulement de l'application.

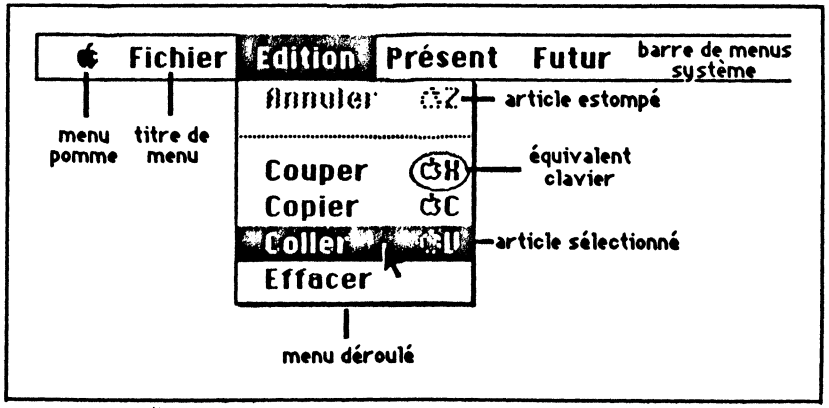


Figure VI.1. Les principaux concepts.

C'est le *Menu Manager* qui se charge de toute la gestion des menus déroulants. Grâce à lui, le programmeur va pouvoir créer facilement une barre de menus

conforme à l'interface utilisateur, modifier l'apparence de ces menus en cours d'application et détecter quel article l'utilisateur a sélectionné afin de déclencher la commande adéquate.

Les titres contenus dans une barre de menus sont généralement actifs, mais peuvent être temporairement désactivés. Dans ce cas, le menu peut toujours être déroulé, mais son titre et tous ses articles sont estompés, donc impossibles à sélectionner.

Une barre de menus peut apparaître n'importe où à l'écran. Toutefois, on évitera soigneusement d'abuser des barres de menus susceptibles de désorienter l'utilisateur. Sauf nécessité absolue, une application ne devrait utiliser qu'une seule barre de menus : la *barre système,* qui apparaît en haut de l'écran sur toute sa largeur, et que rien ne peut venir recouvrir, hormis le pointeur. Pour respecter l'interface, la barre de menus système devrait contenir au moins trois menus : le menu (accessoires de bureau), le menu *Fichier* (accès disquette) et le menu *Edition* (couper-copier-coller). Les menus propres à l'application ne viennent qu'ensuite. En mode 320, cela laisse peu de place !

Un article standard dans un menu peut être soit le texte d'une commande, soit une ligne divisant des groupes d'articles. Un article peut être coché (désignant ainsi la sélection permanente d'une option) ou non coché, peut posséder un équivalent clavier (l'article peut alors être sélectionné par Pomme-caractère) ou non. Il existe la possibilité de créer des menus non standard (par exemple pour offrir un choix de couleurs ou de trames, dans les applications graphiques). C'est alors le programmeur qui prend une partie de la gestion de ces menus personnalisés à son compte.

Note Les premières versions du Menu Manager n'offrent pas le défilement automatique des articles quand leur nombre dans un menu excède le maximum autorisé. Les versions futures le géreront comme il est géré sur Macintosh, sans qu'aucune modification n'ait à être apportée à ce qui est dit dans ce chapitre.

UTILISATION DU MENU MANAGER

Généralités

Condition préalable : avoir initialisé QuickDraw (dessin des menus) et l'Event Manager (interaction avec l'utilisateur). Généralement, le Window Manager aura également déjà été initialisé. On peut alors initialiser le Menu Manager par la procédure **MenuStartUp**.

Le programme commence par définir chaque menu (son titre et ses articles) avec **NewMenu**, et l'insère dans la barre de menus avec **InsertMenu**. L'affichage de la barre de menus se fait grâce à **DrawMenuBar**.

Quand un événement causé par l'utilisateur interviendra dans un menu, **MenuSelect** et **MenuKey** sont les sous-programmes qu'il faudra utiliser pour y répondre. En fin de commande, **HiliteMenu** sera appelé pour rendre à la barre des menus son aspect normal.

Il existe toute une panoplie d'appels qu'on peut utiliser occasionnellement : insertion et destruction d'articles dans un menu, modification des couleurs par défaut utilisées pour dessiner les menus, modification de l'intitulé d'un article, désactivation d'un article, cochage d'un article, style non standard dans l'écriture d'un article, etc.

Définition d'un menu

Un menu est défini par une chaîne de caractères obéissant à des règles strictes. Voici un premier exemple de définition de menu (il contient trois articles) :


```
> Titre 1\N2          > est un caractère spécial (suivi d'un blanc)
- Article 1.1\N301    - est aussi un caractère spécial (suivi d'un blanc)
- Article 1.2\N302
- Article 1.3\N303
.                     . est également un caractère spécial
```

Dans la chaîne de caractères qui définit un menu, chaque ligne, terminée par un Retour-ligne (code ASCII 13 décimal) ou par un caractère nul (code ASCII 0), représente un titre de menu ou un article. Le premier élément de la chaîne de caractères servira pour chaque titre (ici le caractère >). Le premier élément de la ligne suivante (c'est-à-dire le premier caractère après le code ASCII 0 ou le Retour) servira pour chaque article (ici le caractère –). Enfin, un élément différant des caractères de titre et d'article démarrera la dernière ligne, signifiant la fin de la définition du menu (ici le caractère .). Ainsi, trois caractères particuliers sont définis, en fonction de leur position dans la chaîne. Les caractères >, – et . ne sont donc absolument pas imposés. Juste après le caractère de titre ou d'article, une position doit être réservée (ici nous avons laissé un blanc, mais le caractère est indifférent). Le Menu Manager utilisera cette position pour stocker la longueur du titre ou de l'article, de manière interne, ce qui permettra la modification ultérieure de ces libellés. La définition de menu suivante est strictement équivalente à l'exemple précédent :

```
$$Titre 1\N2        $ est un caractère spécial (suivi d'un caractère bidon)
==Article 1.1\N301  = est aussi un caractère spécial (suivi d'un caractère bidon)
==Article 1.2\N302
==Article 1.3\N303
$                   $ est également un caractère spécial
```

Le caractère spécial marquant la fin de définition d'un menu peut être le même que celui marquant le titre. L'important en réalité est que le caractère marquant les articles soit différent des deux autres caractères de tête.

Certains caractères spéciaux peuvent ou doivent être ajoutés à un titre ou à un article pour changer leur apparence ou introduire des éléments particuliers. Ces caractères commenceront par le caractère \.

Liste des caractères spéciaux :

- \ début des caractères spéciaux ;
- * suivi de deux caractères, l'article possédera un équivalent clavier avec ces caractères ;
- C suivi d'un caractère, l'article sera coché avec ce caractère ;
- B l'article apparaîtra en caractères gras ;
- I l'article apparaîtra en caractères italiques ;
- U l'article apparaîtra en caractères soulignés ;
- V placera une ligne de séparation sous l'article (évite l'utilisation d'un article séparé) ;
- D estompera l'article ou le titre, le rendant ainsi inactif ;
- X utilisera une couleur de remplacement durant la sélection, et non l'inversion standard (par X0R) ;
- N suivi d'un nombre décimal, numéro du titre ou de l'article ;
- H suivi d'un nombre hexadécimal, numéro du titre ou de l'article.

Tous ces caractères spéciaux peuvent affecter la définition d'un article, seuls \, D, X, H et N peuvent affecter celle d'un titre. Leur ordre après \ n'a aucune importance. La numérotation du titre ou de l'article est obligatoire.

Remarque Le caractère \ ne pourra jamais faire partie du texte d'un titre ou d'un article : il marque toujours le début des caractères spéciaux.

Pour créer le titre du menu (accessoires de bureau), on utilise le caractère @, précédé du caractère de titre et du caractère réservé de longueur, et suivi du caractère \. Cette séquence de caractères doit être rigoureusement respectée : aucun

blanc supplémentaire ne doit être inséré avant et après @. Ce menu devrait toujours posséder la directive X (couleur de remplacement en cas de sélection), car la pomme apparaît toujours en couleur dans la barre de menus.

Voici un deuxième exemple de définition de menu (le menu) :

```
>>@\N1X                 menu , portant le numéro 1
--A propos de...\VN256  article 256, sous lequel est placé une barre de séparation
```

Reste à traduire en langage C ces types de chaînes de caractères. Si le caractère nul n'avait pas été admis pour séparer les différentes « lignes », on aurait été obligé de coder les définitions de menus en assembleur ! Heureusement, ce ne sera pas nécessaire.

Voici comment on pourrait écrire les deux menus donnés en exemple précédemment :

```
char menu1[ ] = ">>@\\N1X";
char menu11[ ] = "--A propos de...\\VN256";
char menu1x[ ] = ".";
char menu2[ ] = ">>Titre 1\\N2";
char menu21[ ] = "--Article 1.1\\N301";
char menu22[ ] = "--Article 1.2\\N302";
char menu23[ ] = "--Article 1.3\\N303";
char menu2x[ ] = ".";
```

On notera la double barre oblique pour autoriser le caractère \. Dans ces définitions, chaque ligne est terminée par le caractère nul, puisque nous avons affaire à des chaînes de type C. Les pointeurs *menu1* et *menu2* serviront à repérer l'ensemble du menu et de ses articles, les autres pointeurs ne seront presque jamais utilisés.

Il faut également savoir que le caractère standard qui sert au cochage des articles porte le code ASCII 18 décimal (soit 22 en base 8). Pour définir un menu dont l'un des articles est coché à la création, on écrira donc quelque chose comme :

```
char menu31[ ] = "--Article coché\\N421C\22";
char menu32[ ] = "--Article non coché\\N422";
```

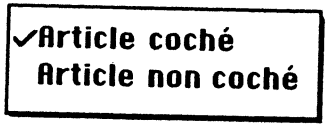


Figure VI.2. Le cochage standard.

Nous verrons d'autres exemples de définition de menus en fin de chapitre et tout au long de cet ouvrage, ainsi que leur manipulation.

Identification de titres et d'articles

Les titres et les articles sont numérotés dans la définition de chaque menu. Aucune limitation ni règle n'est imposée dans la numérotation des titres (pourvu qu'elle tienne sur deux octets). On utilisera généralement les numéros 1 à n (n étant le nombre de menus), sachant que deux menus différents dans une même barre ne peuvent porter le même numéro. Par contre, le numéro des articles utilisés par une application doit être compris entre 256 et 65534, les identifiants compris entre 1 et 249 étant réservés aux accessoires de bureau, et les numéros 250 à 255 devant théoriquement servir aux articles suivants :

250 Annuler	(premier article du menu Edition)
251 Couper	(deuxième article du menu Edition, équivalents clavier Xx)
252 Copier	(troisième article du menu Edition, équivalents clavier Cc)
253 Coller	(quatrième article du menu Edition, équivalents clavier Vv)
254 Effacer	(cinquième article du menu Edition)
255 Fermer	(article nécessaire du menu Fichier)

Ces articles peuvent être qualifiés de spéciaux, car ils devraient toujours appartenir à une application qui gère les accessoires de bureau, et être actifs quand la fenêtre de premier plan est une fenêtre système. Le fait de leur imposer un identifiant permettra à la fonction **TaskMaster** de prendre complètement en charge la réponse à leur sélection par l'utilisateur dès lors qu'elle concernera un accessoire de bureau.

Deux articles ne peuvent porter le même numéro, à l'intérieur d'une barre, même s'ils sont dans des menus différents. Ce sont ces numéros qui permettront d'identifier l'article ou le menu sélectionné par l'utilisateur. Aucune obligation n'est faite dans un menu d'ordonner les articles dans l'ordre croissant de leur identifiant, même si la logique nous invite à procéder de la sorte, ainsi que nous l'avons fait dans les exemples précédents.

Lignes de séparation

Il est souvent intéressant de séparer les articles d'un menu en groupes homogènes, afin de faciliter à l'utilisateur le repérage de certaines fonctionnalités. Le Menu Manager nous offre deux manières d'agir : l'utilisation d'un article qui sera une ligne de division (cet article devra être estompé et porter son propre identifiant), ou l'utilisation du caractère spécial V dans la définition de l'article, provoquant une ligne de soulignage de la largeur du menu. Choisir entre les deux sera souvent une affaire de goût personnel... à moins que le nombre élevé d'articles dans un menu ne contraigne à l'utilisation de la deuxième solution.

On utilisera les deux types d'instructions suivants :

```
    char menu53[ ] = "- Article 3\\N503V";
    char menu54[ ] = "- Article 4\\N504";
/* ou */
    char menu53[ ] = "- Article 3\\N503";
    char menu54[ ] = "- -\\N504D";
    char menu55[ ] = "- Article 5\\N505";
```

Notons dans le deuxième cas que le nom de l'article est limité à un simple caractère (le tiret), ce qui suffira à créer une ligne de séparation complète. L'option D fera de cette ligne un article non sélectionnable. Dans ce cas, et dans ce cas uniquement, l'article pourrait porter le même identifiant que celui qui le précède, mais inutile de jouer avec le feu !

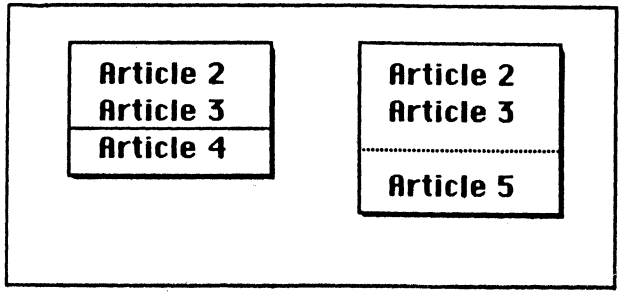


Figure VI.3. Les lignes de séparation.

Équivalents-clavier

Dans la définition d'un menu, à chaque article peuvent être associés deux équivalents-clavier. La définition :

```
char menu35[ ] = "--Poursuivre la recherche\\"RrN281";
```

signifie que l'article numéro 281 et qui porte le titre « Poursuivre la recherche » possède deux équivalents-clavier : Pomme-R et Pomme-r. Le premier équivalent s'appelle équivalent primaire, le second s'appelle équivalent alternatif. Quand on ne désire pas d'équivalent alternatif, on laisse obligatoirement un blanc à la place : le caractère spécial * doit toujours être suivi de deux caractères équivalents-clavier, le premier étant significatif.

De manière évidente, on ne devrait pas trouver deux équivalents-clavier identiques dans deux articles différents. Aucun contrôle n'est fait par le système pour empêcher cette anomalie.

Pourquoi deux équivalents-clavier ? Le Menu Manager ne convertit pas les minuscules en majuscules quand l'utilisateur invoque une commande à partir du clavier par une combinaison Pomme-touche. Les deux équivalents-clavier seront donc en règle générale (quand il s'agira d'une lettre) le caractère majuscule et minuscule correspondant.

Notion de barre de menus courante

Généralement, une application ne possède qu'une barre de menus : la barre système, qui prend place tout en haut de l'écran, où elle occupe généralement les treize premières lignes. Dans certains cas particuliers, une application peut gérer d'autres barres de menus en plus de la barre système, par exemple une barre de menus dans la zone d'informations de chaque fenêtre. Certaines routines du Menu Manager s'appliquent à la totalité d'une barre de menus (par exemple **InsertMenu** ou **DrawMenuBar**) et agissent sur la barre de menus courante. Quand il n'y a que la barre système, cette barre est toujours la barre courante. Quand par contre il y a plusieurs barres de menus, **SetMenuBar** permet d'en sélectionner une avant d'appeler certaines routines. Cette barre sélectionnée devient alors la barre courante.

Définir ses propres menus

Signalons pour être complet que si la manière utilisée par le Menu Manager pour dessiner les menus ne vous convient pas, par exemple parce que vous désirez créer un menu de couleurs ou de patterns, chose impossible par la procédure normale, vous pouvez définir vos propres procédures pour dessiner un menu et sélectionner ses articles... Nous vous renvoyons à la documentation technique Apple pour de plus amples renseignements sur ce sujet.

La possibilité de définir ses propres menus ne doit pas être confondue avec la possibilité de jouer sur les couleurs utilisées par les procédures standard. Comme pour tout ce qui est système, le Menu Manager utilise exclusivement le noir et le blanc pour dessiner ses menus (texte noir sur fond blanc, avec une exception, la pomme !), ce qui limite les surprises en cas de manipulation des tables de couleurs par l'application. Nous verrons par l'exemple qu'il est facile d'utiliser autre chose que le noir et blanc.

EXEMPLES D'UTILISATION

Mise en place d'une barre de menus système

```
extern char menu1[ ];          /* définition du menu  */
extern char menu2[ ];          /* définition du menu Fichier */
extern char menu3[ ];          /* définition du menu Edition */

...                            /* initialisations précédentes */
MenuStartUp(myID, zeropg);     /* initialisation du Menu Manager */
...                            /* suite des initialisations */

InsertMenu(NewMenu(menu3), 0);      /* troisième menu dans la barre */
InsertMenu(NewMenu(menu2), 0);      /* deuxième menu dans la barre */
InsertMenu(NewMenu(menu1), 0);      /* premier menu dans la barre */

FixAppleMenu(1);               /* mise en place des accessoires de bureau */
FixMenuBar( );                 /* taille par défaut pour la barre de menus */
DrawMenuBar( );                /* dessin de la barre de menus */
```

Pour pouvoir être utilisé, le Menu Manager doit être initialisé. La procédure **MenuStartUp** assure ce travail : elle crée une barre de menus système vide, en fait la barre de menus courante, et dessine cette barre vide. Si le Window Manager est initialisé, la place sur l'écran pour la barre de menus sera réservée, et aucune fenêtre ne pourra venir la recouvrir. **MenuStartUp** possède deux arguments : le premier est le numéro identifiant l'application (tel qu'il est retourné par la fonction **MMStartUp** du Memory Manager). Le second désigne une frontière de page dans la banque 0. Le Menu Manager a besoin d'une page complète dans la banque 0 pour pouvoir fonctionner. Consulter le chapitre XII pour une vision d'ensemble de l'initialisation des outils.

La fonction **NewMenu** réserve de la place pour un menu et ses articles, décrits parfaitement par la chaîne de caractères sur laquelle pointe son argument. Elle retourne un handle sur cette liste, et c'est ce paramètre qui servira ensuite à localiser ce menu. Dans l'exemple, les handles sur chaque menu sont passés en argument de la procédure **InsertMenu**, mais ne sont pas mémorisés.

La procédure **InsertMenu** permet d'ajouter un menu à la barre de menus courante (par défaut la barre système). Le premier argument est un handle retourné par la fonction **NewMenu**, le second un numéro d'ordre. Dans l'exemple, 0 signifie que le menu viendra à gauche de tous les menus déjà présents dans la barre. On aurait pu procéder différemment : 2 par exemple aurait signifié que le nouveau menu devait s'insérer à droite du menu portant l'identifiant 2. En procédant de droite à gauche comme dans l'exemple, on n'a pas à se poser de questions sur le deuxième argument de **InsertMenu** : il est systématiquement nul.

FixAppleMenu n'est pas une routine du Menu Manager, mais du Desk Manager. Sa fonction est d'ajouter au menu dont l'identifiant est passé en argument (de préférence le menu , merci !) les accessoires de bureau disponibles sur la disquette. Si l'application décide de ne pas supporter les accessoires de bureau, cette routine ne doit pas être utilisée. L'argument qu'utilise cette procédure désigne l'identifiant du menu . Les accessoires auront des numéros consécutifs à partir de 1. (Voir le chapitre X).

La fonction **FixMenuBar** calcule les dimensions que va utiliser la barre de menus courante ainsi constituée (elle retourne la hauteur de la barre de menus, valeur dont une application a rarement besoin, voir l'exemple de fin de chapitre), et **DrawMenuBar** la dessine à l'écran.

Réponse à une sélection d'article par l'utilisateur utilisant la souris

```
TaskRec    tache;                          /* tache est un Task record */
...        /* GetNextEvent a retourné un événement de type MouseDown */
...        /* FindWindow nous apprend qu'il a eu lieu dans la barre système */
...
MenuSelect(&tache, 0L);
ExecMenu(tache.TaskData); /* la fonction ExecMenu gère la sélection de l'utilisateur */
...
...          /* suite de l'application */


    ExecMenu(art, menu)

int art, menu;

{
switch(art)
    {
    case 256:  /* un case pour la commande correspondant à l'article  numéro 256*/
    ...        /* exécution de la commande */
    break;

    case 257:  /* un case pour la commande correspondant à l'article  numéro 257 */
    ...        /* exécution de la commande */
    break;

    ... /* etc, un case pour chaque article de la barre de menus */
    }
if (art) HiliteMenu(FALSE, menu);
}
```

L'Event Manager a retourné un événement de type *MouseDown* (le bouton de la souris a été enfoncé) par l'intermédiaire de **GetNextEvent**, et le Window Manager nous a appris que cette action a eu lieu dans la barre de menus système (**FindWindow** a retourné la valeur *wInMenuBar*). L'application doit alors appeler **MenuSelect**, procédure qui va prendre en charge la gestion complète des manipulations de menus : inversion du titre sélectionné, dessin du menu déroulé, inversion des articles en fonction de la position du pointeur, etc. Cette procédure nécessite deux arguments : le premier est un pointeur sur *TaskRec*, le second un handle désignant la barre de menus (zéro-long signifie barre système). Le *TaskRec* sert à la fois d'input et d'output à la procédure : l'input sera le résultat du dernier **GetNextEvent**, l'output désignera le menu et l'article sélectionnés par l'utilisateur (partie *TaskData* du *TaskRec*). A la suite de cet appel, la barre de menus désignée par le second argument devient la barre courante.

Si pour n'importe quelle raison il n'y a pas eu de sélection, le numéro d'article retourné est nul. Sinon, l'application doit répondre à l'intervention de l'utilisateur en exécutant la commande sélectionnée, puis rendre au titre de menu son aspect normal (il reste inversé pendant toute la durée de la commande) grâce à la procédure **HiliteMenu**. Cette procédure admet deux arguments : le premier indique si on doit contraster le titre (TRUE) ou si on doit le rendre normal (FALSE), le second est le numéro du menu dont le titre est concerné.

L'exemple utilise une astuce propre au langage C pour retrouver le numéro d'article et de menu à partir du champ *TaskData* du *Task record*. Ce type de raccourci est impossible en Pascal ! Le champ *TaskData* a pour longueur 32 bits. Le Menu

Manager retourne dans les 16 bits les plus significatifs l'identifiant du menu sélectionné, et dans les 16 bits bas l'identifiant de l'article sélectionné (ou zéro si rien n'est sélectionné). La fonction ExecMenu que nous avons définie reçoit en argument cet entier long, mais le traite en réalité comme deux arguments sur 16 bits : d'abord l'article, ensuite le menu (ne pas oublier qu'une fonction C traite les arguments dans l'ordre inverse de leur apparition dans les parenthèses).

Si le numéro d'article est zéro, aucun *case* ne sera sélectionnée dans notre fonction ExecMenu, et donc aucune commande ne sera lancée. **HiliteMenu** ne sera même pas exécutée dans ce cas, le Menu Manager rétablissant de lui-même l'aspect normal d'un menu quand aucun article n'est retenu.

Remarque importante Si une application n'utilise pas d'autre barre de menus que la barre système (ou les barres systèmes), elle pourra laisser **TaskMaster** gérer toute seule les sélections dans les menus. Nous verrons comment utiliser cette fonction dans le chapitre XI, qui lui est consacré. Par contre, l'appel à **MenuSelect** est indispensable pour désigner autre chose que la barre système.

Réponse à une sélection d'article par l'utilisateur utilisant le clavier

```
TaskRec    tache;                          /* tache est un Task record */

...        /* GetNextEvent a retourné un événement de type KeyDown */
...        /* l'analyse a montré que la touche Pomme était également enfoncée */

MenuKey(&tache, 0L);
ExecMenu(tache.TaskData); /* la fonction ExecMenu gère la sélection de l'utilisateur */
                          /* elle est commune à MenuSelect et MenuKey */
```

L'Event Manager a retourné un événement de type *KeyDown* (une des touches du clavier a été enfoncée) par l'intermédiaire de **GetNextEvent**, et l'analyse du *Task record* nous a appris que simultanément la touche Pomme était enfoncée : l'utilisateur veut donc sélectionner un article d'un menu par son équivalent-clavier. L'application doit alors appeler **MenuKey**, procédure qui va rechercher à quel article correspond la commande invoquée. Cette procédure nécessite deux arguments, identiques à ceux de **MenuSelect**, et retourne les mêmes informations que cette procédure (dans le champ *TaskData* du *Task record*).

La recherche de l'article se fait en deux temps. La routine commence par rechercher dans tous les menus (de la gauche vers la droite) si un article possède l'équivalent-clavier primaire correspondant à la touche enfoncée et s'arrête dès qu'elle en trouve un. S'il n'y en pas, elle recherche ensuite parmi les équivalents-clavier alternatifs. S'il n'y en a toujours pas, aucun article ne sera sélectionné.

Rappel Menukey fait la distinction entre majuscules et minuscules, l'équivalent alternatif a pour but principal de donner à une minuscule la même fonction commande que la majuscule.

A la suite de cet appel, la barre de menus désignée par le second argument devient la barre courante.

La fonction ExecMenu appelée dans cet exemple est commune avec l'exemple précédent. Notamment, il est toujours nécessaire d'appeler la procédure **HiliteMenu**.

Modifications sur un menu

- On peut changer les caractéristiques d'un menu de la barre courante grâce à la procédure **SetMenuFlag**. Cette procédure réclame deux arguments : une valeur

prédéfinie désignant le nouvel état et le numéro du menu. Voici les valeurs les plus intéressantes (celles qui concernent les caractères spéciaux \D et \X de la définition) :

```
SetMenuFlag(EnableMenu, n);     /* rend actif le menu n (annule \D) */
SetMenuFlag(DisableMenu, n);    /* rend inactif (estompe) le menu n (force \D) */
SetMenuFlag(XORhilite, n);/* sélection par inversion (annule \X) */
SetMenuFlag(ColorReplace, n);   /* sélection par couleur de remplacement (force \X) */
```

Les valeurs prédéfinies sont les suivantes :

```
#define EnableMenu      0xFF7F
#define DisableMenu     0x0080
#define ColorReplace    0xFFDF
#define XORhilite       0x0020
```

Il faut appeler **DrawMenuBar** pour rendre visible à l'écran le résultat de **SetMenuFlag**.

Par exemple, une application peut ne pas avoir besoin de gérer le copier-coller, mais posséder néanmoins un menu *Edition* à l'usage des accessoires de bureau. Ce menu sera estompé si l'une des fenêtres de l'application se trouve au premier plan, actif s'il s'agit d'une fenêtre système. Voir l'exemple complet en fin de chapitre.

Note L'action d'estomper un menu se répercute sur tous les articles de ce menu, qui apparaissent dès lors également estompés. Cependant, l'état réel de l'article n'est pas modifié, si bien que lorsque le menu est réactivé, seuls les articles déjà actifs auparavant le redeviendront.

• On peut changer le titre du menu, quand l'application le justifie, dans des cas très particuliers, grâce à la procédure **SetMenuTitle**.
Attention La chaîne de caractères constituant le nouveau titre doit être de type Pascal. On peut aussi récupérer le titre d'un menu, grâce à la fonction **GetMenuTitle** qui retourne un pointeur sur le titre dont l'identifiant est passé en argument.

```
Pointer oldTitle;
char    newTitle[ ] = "\16 Nouveau titre";    /* chaîne de caractères type Pascal */

oldTitle = GetMenuTitle(n);
SetMenuTitle(newTitle, n);                    /* le menu n porte un nouveau titre */
```

Il faut appeler **DrawMenuBar** pour rendre visible à l'écran le résultat de **SetMenuTitle**.

• On peut également vouloir changer le numéro d'un menu. Rien de plus simple, avec la procédure **SetMenuID** :

```
SetMenuID(m, n);                    /* le menu n porte désormais le numéro m */
```

Modifications sur un article

Contrairement aux modifications sur les menus, il n'est pas nécessaire de redessiner la barre de menus pour que les modifications sur articles soient prises en compte : ils seront dessinés correctement dès que le menu auquel ils appartiennent sera déroulé.

• Pour rendre actif ou inactif un article, deux procédures sont disponibles : **EnableMItem** et **DisableMItem**. Un seul argument : le numéro de l'article considéré. Au moment de la définition, ceci se traduit par la présence ou non de \D.


```
DisableMItem(n);        /* l'article n devient estompé, donc non sélectionnable */
EnableMItem(n);         /* l'article n redevient normal, sélectionnable */
```

• Pour manipuler la marque devant un article (article coché ou non), une seule procédure est nécessaire : **CheckMItem**. Deux arguments lui sont nécessaires. Le second est le numéro de l'article ; le premier a la valeur TRUE si on veut cocher l'article ou la valeur FALSE pour retirer la marque. Si on essaie de cocher un article déjà coché, il ne se passe rien, et réciproquement. Au moment de la définition, ceci se traduit par la présence ou non de la directive \C (suivie du caractère ASCII 18 décimal).

```
int indic = TRUE;                   /* indicateur booléen: TRUE = article coché */

/* l'article n est sélectionné, réponse à la commande */
case n:
    indic = !indic;                 /* on change la valeur de l'indicateur */
    CheckMItem(indic, n); /* suivant la valeur de l'indicateur, on coche ou on démarque */
    if (indic)                      /* y a-t-il encore une marque? */
    actionSiVrai( );                /* ...oui */
    else actionSiFaux( );           /* ...non */
    break;
```

Au lieu de gérer un indicateur précisant l'état coché ou non coché d'un article, on peut utiliser la fonction **GetItemMark**, qui retourne pour l'article donné en argument le code ASCII du caractère de cochage présent, ou la valeur zéro si l'article n'est pas coché. Si on ne se préoccupe pas du code du caractère de cochage, on peut considérer que la fonction retourne FALSE si l'article n'est pas coché, et !FALSE (au sens C du terme, pas vraiment TRUE puisque tous les bits ne sont pas à 1) si l'article est coché.

La procédure **CheckMItem** utilise exclusivement le code ASCII 18 décimal pour cocher un article. La procédure **SetItemMark** étend ses possibilités : le premier argument n'est plus un booléen, mais le code ASCII (quelconque) du caractère de cochage, la valeur zéro signifiant (comme dans **CheckMItem**) pas de cochage.

Voici un exemple, équivalant au précédent, utilisant ces deux routines. Seule différence : on utilise le caractère > comme caractère de cochage.

```
case n:
    if (GetItemMark(n))             /* s'il y a une marque... */
        {
        SetItemMark(0,n);           /* on la retire */
        actionSiFaux( );            /* et on fait ce qu'il faut */
        }
    else                            /* sinon... */
        {
        SetItemMark('>',n);         /* on met la marque */
        actionSiVrai( );            /* et on fait ce qu'il faut */
        }
    break;
```

On peut écrire le nom de l'article avec des styles différents (standard, gras, italique, souligné). Au moment de la définition, ceci se traduit par la présence, l'absence ou une combinaison de \B ou \U ou \I. On peut changer de style en cours de route avec la procédure **SetItemStyle**. Deux arguments : le second est le numéro de l'article, le premier le code du style à employer :

```
SetItemStyle(0, n1);    /* l'article n1 est écrit dans le style standard */
SetItemStyle(1, n2);    /* l'article n2 est écrit en gras, comme avec \B */
SetItemStyle(2, n3);    /* l'article n3 est écrit en italique, comme avec \I */
SetItemStyle(4, n4);    /* l'article n4 est écrit en souligné, comme avec \U */
SetItemStyle(7, n5); /* l'article n5 est écrit en gras italique souligné, comme avec \BIU */
```

On constate dans le dernier exemple qu'il est possible de combiner les styles, comme dans QuickDraw. Il est à noter que certaines polices de caractères ne tiennent pas compte du souligné, et notamment la police utilisée par le système, aussi bien en mode 320 qu'en mode 640 ! Pour retrouver le style utilisé par un article donné, on peut appeler la fonction **GetItemStyle** :

```
int x, y;

x = GetItemStyle(n2);      /* x reçoit la valeur 1 (style gras) */
y = GetItemStyle(n5);      /* y reçoit la valeur 7 (style gras italique souligné) */
```

• On peut changer le nom de l'article, quand l'application le justifie, grâce à la procédure **SetItem**. Deux arguments : un pointeur sur le nouveau nom et le numéro de l'article incriminé.
Attention La chaîne de caractères constituant le nouveau nom doit être de type Pascal. De plus, un octet en début de chaîne doit être réservé, le Menu Manager s'en servira pour stocker la longueur de l'article.
Exemple Quand une commande peut prendre deux états, on changera son nom à chaque sélection.

```
char    nom1[ ] = "\22-Afficher la règle";
char    nom2[ ] = "\21-Masquer la règle";
int     flag = FALSE;           /* TRUE si la règle est affichée, FALSE sinon */
int     menuID;                 /* l'identifiant du menu modifié */

/* l'article n est sélectionné, réponse à la commande */
case n:
    if (flag)     /* la règle est-elle affichée? */
        {
        SetItem(nom1, n);       /* oui: on change le nom de l'article... */
        masque( );              /* ...et on masque la règle */
        }
    else
        {
        SetItem(nom2, n);       /* non: on change le nom de l'article... */
        affiche( );             /* ...et on affiche la règle */
        }
    CalcMenuSize(0, 0, menuID); /* recalcule la largeur du menu */
    flag = !flag;               /* on change la valeur de l'indicateur */
    break;
```

Si les libellés sont de longueur nettement différente (en pixels), il sera obligatoire d'appeler **CalcMenuSize** (voir plus loin).

Notons l'existence de la fonction **GetItem**, qui retourne un pointeur sur le nom de l'article dont l'identifiant est passé en argument.

• On peut changer les autres caractéristiques d'un article (\V et \X dans la définition) grâce à la procédure **SetItemFlag**. Deux arguments : une valeur prédéfinie et le numéro de l'article. Voici les divers cas de figure :

```
SetItemFlag(UnderItem, n);      /* équivaut à forcer \V */
SetItemFlag(NoUnderItem, n);    /* équivaut à annuler \V) */
SetItemFlag(XORhilite, n);      /* équivaut à annuler \X) */
SetItemFlag(ColorReplace, n);   /* équivaut à forcer \X) */
```

Les deux dernières valeurs utilisées ont été vues plus haut. Les deux premières sont définies ainsi :

```
#define UnderItem       0x0040
#define NoUnderItem     0xFFBF
```

• Enfin, pour changer le numéro d'un article, il suffit d'appeler **SetItemID** :

```
SetItemID(m, n);        /* l'article n porte désormais le numéro m */
```

Accès à une barre de menus

• Nous avons déjà parlé de la procédure **InsertMenu** au moment de la création d'une barre de menus. C'est évidemment cette procédure qu'une application ou un accessoire de bureau doit appeler pour ajouter un menu supplémentaire dans la barre courante, à n'importe quel moment. Pour être sûr que le menu ajouté viendra à droite de tous les menus présents, on pourra écrire :

```
InsertMenu(NewMenu(menuX),65535);   /* menuX est un pointeur
                                       sur le menu d'identifiant X */
```

La valeur 65535 (équivalente à − 1 en représentation interne) assurera la place la plus à droite, puisqu'aucun menu ne peut avoir un identifiant plus grand. La fonction **NewMenu**, rappelons-le, réserve de la place en mémoire vive pour mémoriser les renseignements concernant le menu et ses articles. Si cette allocation a déjà été faite, il ne faut pas la recommencer, sous peine de voir sa mémoire saturer très rapidement ! Logiquement, quand l'application doit jongler avec ses menus, elle conserve trace des handles les localisant en mémoire. Si tel n'était pas le cas, il y aurait encore une possibilité, retrouver le handle sur un menu grâce à son identifiant par la fonction **GetMHandle**. Si nous supposons que MenuX a déjà été alloué par **NewMenu**, que nous n'avons pas gardé la valeur du handle qui lui est attaché et que son identifiant porte la valeur X, nous écrirons pour insérer ce menu :

```
InsertMenu(GetMHandle(X),65535);
```

La fonction **GetMHandle** retourne un handle sur le menu d'identifiant X, ou zéro-long en cas d'erreur (si par exemple la fonction **NewMenu** n'avait jamais été appelée pour ce menu).

Maintenant que nous savons ajouter des menus dans une barre, il peut être intéressant de savoir les enlever. C'est la procédure **DeleteMenu** qui retire de la barre courante le menu dont l'identifiant est passé en argument.
Attention Elle retire le menu, mais elle ne libère pas l'espace mémoire qui lui est consacré, donc le handle sur menu est toujours valide après cet appel. Pour détruire définitivement un menu et libérer la place qu'il occupe en mémoire, on utilisera la procédure **DisposeMenu** en donnant comme argument le handle du menu à évacuer. Le menu n'étant plus accessible, **GetMHandle** retournerait zéro-long en cas d'appel pour ce menu.

Après avoir inséré ou retiré un menu, il est obligatoire d'appeler **DrawMenuBar** pour répercuter ces modifications à l'écran.

• Ce qu'on peut faire pour les menus à l'intérieur de la barre courante, on peut également la faire pour les articles à l'intérieur d'un menu. Les procédures sont **InsertItem** et **DeleteItem**.

La procédure **InsertItem** réclame trois arguments :

– un pointeur sur une ligne de définition d'article (vous vous souvenez ? caractère spécial, suivi d'un caractère réservant de la place pour la longueur, suivi du titre de l'article, suivi des options, et terminé par le caractère nul) ;

– l'identifiant de l'article après lequel le nouvel article viendra s'insérer (donner la valeur zéro pour que l'article soit le premier du menu, et la valeur 65535 ou − 1 pour qu'il soit le dernier, tout en bas du menu déroulé) ;

– l'identifiant du menu dans lequel l'article doit venir s'insérer.

La procédure **DeleteItem** ne réclame qu'un seul argument (il est toujours plus facile de détruire que de construire), l'identifiant de l'article à retirer de la barre de menus.

On appellera **CalcMenuSize** pour laisser le Menu Manager recalculer la taille du menu incriminé. Trois arguments lui sont nécessaires : la nouvelle largeur du menu (mettre 0 pour que le Menu Manager se débrouille tout seul), la nouvelle hauteur du menu quand il est déroulé (mettre ici aussi la valeur 0) et l'identifiant du menu.

Notons que **FixMenuBar** appelle **CalcMenuSize** pour chaque menu de la barre dont elle doit calculer les dimensions, avec des nombres négatifs pour les deux premiers arguments, ce qui ne permet pas de recalculer une largeur de menu préalablement non nulle : cet appel est parfait au moment d'une création, mais pratiquement inopérant par la suite ! C'est bien **CalcMenuSize** qu'il faut appeler ici, et non **FixMenuBar**.

Exemple d'utilisation de ces possibilités : une application multifenêtres peut vouloir gérer un menu dont les articles seraient le titre de chacune des fenêtres ouvertes. Dès que l'utilisateur ouvre une nouvelle fenêtre, un article est ajouté dans le menu, et dès qu'il ferme une fenêtre, l'article correspondant est effacé.

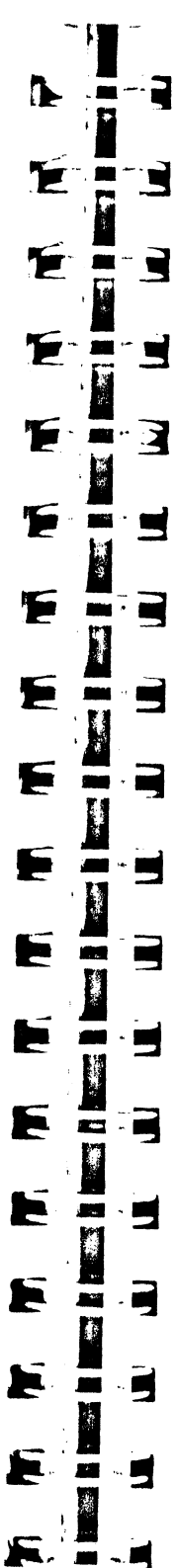
```
char    menu5[ ] = ">>Fenêtres\\N5";
char    menu51[ ] = "– \\N500";
char    menu5x[ ] = ".";
Pointer fen[9];              /* on autorise 9 pointeurs sur fenêtre */

/* au début du programme */
InsertMenu(NewMenu(menu5), -1);
DeleteItem(500);                        /* quand ça marchera! */
CalcMenuSize(0, 0, 5);                  /* recalcule la hauteur du menu 5 */

/* l'utilisateur ouvre la fenêtre N (1≤N≤9) */
InsertItem(menu51, -1, 5);
SetItemID(500+N, 500);                  /* identifiant de l'article */
SetItem(GetWTitle(fen[N–1]), 500+N);    /* ce qui aurait pu être le titre de l'article */
CalcMenuSize(0, 0, 5);

/* l'utilisateur ferme la fenêtre M (1≤M≤9)*/
DeleteItem(500+M);
CalcMenuSize(0, 0, 5);
```

Le principe de cet exemple est clair : on crée dans la définition du menu un article bidon, qu'on supprime juste après l'allocation du menu. Cet article servira à chaque insertion, mais on modifiera avant affichage son identifiant et son titre (en allant chercher un pointeur sur le titre de la fenêtre). Pour le retrait d'un article, il est de la responsabilité de l'application de faire le lien entre la fenêtre et l'identifiant de l'article qui la concerne.

Malheureusement, tel qu'il est écrit, cet exemple ne fonctionne pas. Deux causes à cela :

– dans sa version actuelle (1.03), le Menu Manager perd complètement les pédales quand on supprime tous les articles d'un menu ! On ne pourra donc pas supprimer l'article bidon d'emblée, et il faudra travailler un peu plus : le supprimer après insertion du premier article valable, le rétablir avant retrait du dernier article valable.

– le passage d'argument entre **GetWTitle** et **SetItem** est impossible, à cause de l'octet supplémentaire dont a besoin le Menu Manager pour gérer le libellé de l'article, et que ne possède évidemment pas le titre de la fenêtre. On peut penser ruser en laissant un blanc devant le titre de la fenêtre. Malheureusement, le Menu Manager va mettre une valeur à la place de ce blanc, générant un caractère parasite qui apparaîtra dans le titre de la fenêtre dès que celle-ci sera redessinée ! On verra dans l'exemple en fin de chapitre une façon de tourner la difficulté, en gérant deux listes de noms. C'est nettement plus pénible, plus gourmand en place mémoire, mais cela fonctionne sans anicroche !



Remarque Si le passage d'argument entre **GetWTitle** et **SetItem** avait été correct, il aurait tout de même fallu faire attention. Un compilateur qui convertirait de lui-même les chaînes Pascal en chaînes C n'apprécierait pas du tout une telle construction, qui conduirait au plantage assuré ! (voir dans l'introduction le paragraphe consacré aux chaînes de caractères).

• Une application peut à tout instant savoir combien d'articles contient un menu, ce qui lui évite une comptabilité parallèle en cas d'appels à **InsertItem** et **DeleteItem** dus aux manipulations de l'utilisateur. La fonction **CountMItems** retourne le nombre actuel d'articles présents dans le menu dont l'identifiant est passé en argument :

```
int nbArt;

nbArt = CountMItems(5);          /* nombre d'articles du menu 5 */
```

En conjonction avec l'exemple précédent, l'entier *nbArt* contiendrait le nombre de fenêtres actuellement ouvertes par l'application. Notons que dans la version 1.03 du Menu Manager, cette fonction renvoie n'importe quoi !

• Même si ce n'est pas sa vocation, une barre de menus peut servir à passer des messages. Par exemple, une application qui n'utiliserait pas les menus déroulants pourrait se servir de la barre système pour laisser à l'écran une sorte de signature. Un tel message pourrait être centré. La procédure **SetTitleStart** permet de laisser jusqu'à 127 pixels à blanc à gauche du premier titre de la barre. C'est une façon élégante d'opérer. Réciproquement, la fonction **GetTitleStart** renvoie le nombre de pixels laissés à blanc.

• La largeur d'un titre de menu peut être fixée indépendamment des caractères qui le composent (c'est pratique pour élargir un menu dont le titre serait vraiment trop étroit) grâce à la procédure **SetTitleWidth**. Deux arguments : la nouvelle largeur (un entier contenant le nombre de pixels) et l'identifiant du menu visé. La fonction **GetTitleWidth** retourne la largeur du titre dont l'identifiant est passé en argument.

Exemple On élargit le titre du menu 3 de 10 pixels, pour que l'utilisateur ait une surface plus grande pour sélectionner ce menu.

```
SetTitleWidth(GetTitleWidth(3), 3);
```

• Pour attirer l'attention sans faire de bruit, on peut utiliser la procédure **FlashMenuBar**, sans argument. Son rôle est conforme à son nom : elle va dessiner la barre courante avec sa couleur de sélection, puis la redessiner immédiatement avec sa couleur normale, créant ainsi une sorte d'éclair en haut de l'écran.

• Le Menu Manager dessine ses barres de menus par l'intermédiaire d'un grafport, à l'instar de tout ce qui apparaît à l'écran. Il peut être intéressant d'intervenir dans ce grafport, par exemple pour changer la police de caractères utilisée. Un pointeur sur ce grafport nous est retourné par la fonction **GetMenuMgrPort**, et grâce à ce pointeur nous pourrons utiliser les routines QuickDraw adéquates. Le point suivant va nous montrer qu'il est inutile de faire appel à cette fonction pour modifier les couleurs utilisées par le Menu Manager.
Note L'origine du *Menu Manager port* est en (0,0), ce qui signifie que les coordonnées locales et globales coïncident dans ce grafport, tout comme dans le *Window Manager port*.

• Et si nous changions les couleurs par défaut de la barre de menus ? Rien ne nous empêche d'écrire nos menus en bleu sur fond jaune, le texte devenant rouge quand il est sélectionné. Il suffit pour cela de passer les bons numéros de couleurs à la procédure **SetBarColors**, eu égard à la palette de couleurs active. Et si l'application change de palette en cours de route, les menus changeront de couleur corrélativement, avec le risque de ne plus être lisibles... La fonction **GetBarColors** nous permet de connaître le numéro des couleurs utilisées dans le dessin de la barre active.

La procédure **SetBarColors** admet trois arguments :

– un entier désignant les couleurs « normales » : numéro de couleur pour dessiner un texte non sélectionné (bits 0 à 3), couleur du fond quand non sélectionné (bits 4 à 7). Les bits 8 à 15 sont à zéro ;

– un entier désignant les couleurs « en sélection \X » : numéro de couleur pour dessiner un texte sélectionné (bits 0 à 3), couleur du fond si sélectionné (bits 4 à 7). Les bits 8 à 15 sont à zéro ;

– un entier désignant la couleur des traits utilisés dans le dessin des menus (cadres et barres de séparation). Les bits 4 à 7 contiennent ce numéro de couleur, les autres bits sont à zéro.

La fonction **GetBarColors** retourne ces mêmes informations, mais condensées dans un entier long :

- bits 0 à 3 : couleur du texte normal ;
- bits 4 à 7 : couleur du fond non sélectionné ;
- bits 8 à 11 : couleur du texte sélectionné (cas de la directive \X) ;
- bits 12 à 15 : couleur du fond sélectionné (cas de la directive \X) ;
- bits 16 à 19 : zéro ;
- bits 20 à 23 : couleur des lignes ;
- bits 24 à 31 : zéro.

Quand la barre de menus standard est utilisée, **GetBarColors** retourne la valeur hexadécimale 0000 0FF0, ce qui signifie que les textes sont en noir (couleur 0) sur fond blanc (couleur 15) en représentation normale, et en blanc sur fond noir en sélection avec la directive \X. Cela implique que dans ce cas la directive \X n'a aucune influence sur la représentation des menus, puisque l'inversion standard des couleurs donnera également blanc sur noir. Les lignes sont tracées en noir.

Pour écrire en bleu sur fond jaune en « normal » et en rouge sur fond jaune en « sélection \X », toutes les lignes étant en marron, on pourra coder (en mode 320 avec la palette standard) :

```
SetBarColors(0x009D, 0x0097, 0x0020);
```

Dans ces conditions, **GetBarColors** retournerait la valeur hexadécimale : 0020 979D. Quand un titre est sélectionné, le bleu sur fond jaune devient marron sur fond orange (couleurs « inverses ») si la directive \X n'est pas présente ou qu'on a utilisé les instructions **SetMenuFlag** (*XORhilite,* n) ou **SetItemFlag** (*XORhilite,* n) ; le bleu sur fond jaune devient rouge sur fond jaune si la directive \X est présente ou qu'on a utilisé les instructions **SetMenuFlag** (*ColorReplace,* n) ou **SetItemFlag** (*ColorReplace,* n).

Notons que **SetBarColors** admet des arguments négatifs, signifiant que la valeur courante ne doit pas être modifiée. Par exemple, si nous voulons changer seulement la couleur des lignes (orange au lieu de marron), nous écrirons :

```
SetBarColors(-1, -1, 0x0060);
```

Il faut appeler **DrawMenuBar** pour rendre effective la prise en compte des nouvelles couleurs. Notons que dans sa version 1.03, le Menu Manager ne rétablit pas les couleurs par défaut quand la procédure **MenuShutDown** est appelée. Il sera donc préférable d'ajouter l'instruction suivante en fin d'application (pour penser à celles qui suivront, notamment si elles utilisent une résolution différente !) :

```
SetBarColors(0x00F0, 0x000F, 0);          /* rétablit les couleurs initiales */
```


Utilisation de plusieurs barres de menus

Quand le Menu Manager est initialisé, une barre système est créée, et toutes les actions concernant les menus viennent se rapporter à cette barre, qu'on référence par la valeur zéro-long dans certains appels. Pour créer cette barre, **MenuStartUp** fait appel à la fonction **NewMenuBar**. Pour créer elle-même d'autres barres de menus, l'application va elle aussi faire appel à cette fonction.

Deux types de barres de menus sont possibles : les barres système et les barres fenêtre. Chaque barre appartient à un grafport : les barres système sont dessinées dans le *Window Manager port,* les barres fenêtre dans le grafport associé à la fenêtre d'appartenance. Les barres sont dessinées par défaut à partir de l'origine du grafport et ont généralement une hauteur de 13 pixels, donc une barre système se situe en haut de l'écran, une barre fenêtre en haut de la région contenu de la fenêtre. On peut également définir une barre de menus dans la zone d'informations d'une fenêtre, auquel cas elle n'appartient plus à son contenu, mais à son cadre... et elle est donc dessinée dans le *Window Manager port.*

• Une application peut manipuler plusieurs barres système, et jongler de l'une à l'autre en fonction des nécessités du moment. Une seule de ces barres est évidemment visible à un moment donné. Pour créer une barre système, on appelle **NewMenuBar** avec zéro-long comme argument. La fonction retourne un handle qui va servir à repérer la nouvelle barre. Pour dire qu'une barre devient la barre système, on appelle la procédure **SetSysBar** (le handle sur la barre est passé en argument, elle devient barre courante) et on la redessine. Pour connaître le handle qui repère l'actuelle barre système, on appelle la fonction **GetSysBar.**

Une séquence d'instructions typique pour une application gérant deux barres système pourrait être la suivante :

```
Handle barS1, barS2;                    /* handles sur barres système */

...                                     /* début des initialisations */
MenuStartUp(myID, zeropg);
...                                     /* suite des initialisations */
barS1 = GetSysBar( );                   /* handle sur la barre créée à l'initialisation */
                                        /* insertion de menus dans cette barre, par InsertMenu */
...
FixMenuBar( );                          /* calcul des dimensions de la barre */
barS2 = NewMenuBar(0L);                 /* nouvelle barre système... */
SetSysBar(barS2);                       /* ...sur laquelle on va travailler */
                                        /* insertion de menus dans cette barre, par InsertMenu */
...
FixMenuBar( );                          /* calcul des dimensions de la barre */
```

A ce stade, les deux barres système sont créées, elles contiennent des menus, mais aucune d'elles n'est dessinée. On passera alternativement de l'une à l'autre par des appels de ce type :

```
SetSysBar(barS1);                       /* change la barre par défaut */
DrawMenuBar( );                         /* dessine la barre par défaut */
```

Tout appel à **MenuSelect** ou à **MenuKey** avec zéro-long en deuxième argument s'adressera à la barre système par défaut, fixée par le dernier appel à **SetSysBar**.

• En plus d'une ou plusieurs barres système, une application (ou un accessoire de bureau) peut vouloir gérer des barres fenêtres. Le mécanisme de création est assez identique : on appelle **NewMenuBar** en donnant en argument le pointeur repérant le grafport de la fenêtre dans laquelle on veut placer la barre de menus, on appelle la procédure **SetMenuBar** avec en argument le handle sur la barre qui devient barre courante, on appelle éventuellement la fonction **GetMenuBar** pour connaître par son handle la barre actuellement courante.
Attention Il n'y a qu'une barre courante, cette fonction peut donc retourner un handle sur une barre système.

Dans l'exemple complet qui suit, l'application utilise une barre système et une barre dans chaque fenêtre. Notons que tel que l'exemple a été conçu, la barre dans une fenêtre fait partie de sa région contenu, et non de la zone d'informations. Cela impose quelques contraintes : en règle générale, il faudra éviter de venir dessiner par dessus la barre de menus, en modifiant la clip region (sauf si l'utilisateur n'a pas la possibilité de l'écraser, comme c'est le cas ici) ; la fenêtre ne devra pas posséder de barres de défilement, et si elle est redimensionnable, il sera préférable de ne pas autoriser une largeur trop faible qui masque une partie des menus !

La bonne méthode est évidemment de créer la barre fenêtre dans sa zone d'informations, où l'utilisateur n'aura pas accès pour dessiner dessus, et qui reste fixe même quand le contenu de la fenêtre défile. Malheureusement, la mise en place et surtout la gestion d'une telle barre est relativement compliquée (Apple a publié une note technique de huit pages pour en expliquer la mise en œuvre). Faute de place, nous n'en parlerons donc pas ici.

Les menus déroulants sont les mêmes dans chaque fenêtre, la définition leur est donc commune. Il n'empêche qu'ils doivent être gérés de manière distincte, donc la fonction **NewMenu** est appelée deux fois pour chaque menu, une fois par fenêtre. C'est grâce à cela que nous pouvons cocher les articles d'un menu d'une fenêtre sans pour autant interférer sur le menu de l'autre fenêtre.

Pour vérifier si l'utilisateur a bien cliqué dans la barre, on commence par récupérer sa hauteur (valeur retournée par **FixMenuBar**) et on garde trace du rectangle qu'elle définit (en coordonnées locales, la valeur pouvant être arbitrairement grande). Quand **FindWindow** retournera *wInContent*, on appellera la fonction **PtInRect** pour savoir si le bouton a été ou non enfoncé dans le rectangle (donc dans la barre) pour appeler ou non **MenuSelect**.

A titre d'expérience, supprimons ce test et regardons en bas à gauche comment le Menu Manager réagit quand on appelle **MenuSelect** alors que l'utilisateur n'a pas cliqué dans une barre de menus : il renvoie zéro pour l'article sélectionné, et la coordonnée horizontale (locale) du clic souris en numéro de menu ! Aucune gêne cependant dans cet exemple, puisque le cas de l'article nul est prévu dans notre fonction ExecMenu.

Le but de l'exemple est de dessiner dans chaque fenêtre une forme d'une certaine couleur, la forme et la couleur étant choisies dans leurs menus respectifs. Les choix seront stockés dans le champ *wRefCon* de chaque fenêtre. Notons que le code de cet exemple n'est absolument pas optimisé. On aurait pu utiliser des tableaux de fonctions pour faire riche et propre, mais la lisibilité s'en serait nettement ressentie ! On notera la manière retenue pour dessiner les formes et le texte d'accompagnement, **fondamentale** : on génère un événement de mise à jour en invalidant deux rectangles, plutôt que de dessiner directement dans la fenêtre. Ainsi, on est sûr que quelles que soient les manipulations de l'utilisateur (activation d'une autre fenêtre, masquage partiel par déplacement, etc.), la fenêtre aura toujours un contenu rigoureusement exact. Une fenêtre dans laquelle l'utilisateur ne dessine pas, doit toujours être remplie par l'intermédiaire de la fonction de mise à jour.

L'exemple est présenté en mode 640 (même si l'illustration correspond au mode 320), ce qui nous permet d'employer des définitions de pattern dans ce mode (pseudo rouge, vert, bleu, jaune : un pixel sur deux est coloré, l'autre est blanc, ce qui explique la pâleur des couleurs). Élément remarquable : il suffit de décaler les fenêtres d'un pixel horizontalement pour que les couleurs ne soient plus conforme à ce qu'on attend, puisqu'en mode 640 la couleur d'un pixel dépend de sa position à l'écran. On comprend mieux maintenant pourquoi **DragWindow** ne permet par défaut dans ce mode qu'un déplacement de huit pixels d'un coup : c'est exactement la largeur de la définition du pattern, donc pas de sautes de couleurs lors d'une promenade de fenêtre ! Vous pouvez toujours essayer de changer la définition du contenu d'une fenêtre d'un pixel, pour voir !

Pour faire tourner l'exemple en mode 320, il suffit de changer la valeur de la constante mode au début du programme : 0 signifie mode 320, 1 signifie mode 640. Toutes les dimensions (fenêtres, rectangles) tiennent compte de ce paramètre, de même évidemment que les couleurs utilisées.

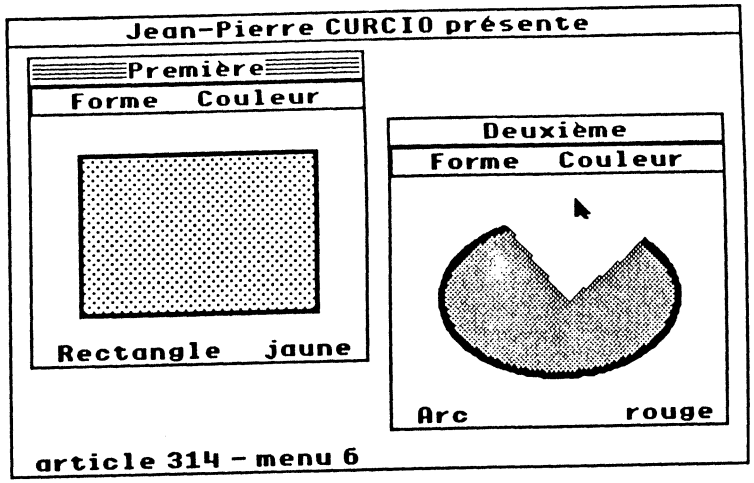


Figure VI.4. Fenêtres avec une barre de menus (en mode 320)

```
#include <tools.h>                          /* définition des termes en gras */
#include <entete.h>                         /* définition des termes en italique */

#define mode 1                              /* 0 si mode 320, 1 si mode 640 */

char Menu2[ ] = ">>Jean-Pierre CURCIO présente\\N2";        /* menu système */
char Menu21[ ] = "--les fenêtres avec menus...\\N271DV";
char Menu23[ ] = "--Quitter\\N273*Qq";
char Menu29[ ] = ". ";

char Menu5[ ] = ">> Forme \\N5X";                           /* premier menu fenêtre */
char Menu51[ ] = "--Rectangle\\N301X";
char Menu52[ ] = "--Ovale\\N302X";
char Menu53[ ] = "--Rect. arr.\\N303X";
char Menu54[ ] = "--Arc\\N304X";
char Menu59[ ] = ". ";
char Menu6[ ] = ">> Couleur\\N6X";                          /* second menu fenêtre */
char Menu61[ ] = "--Rouge\\N311X";
char Menu62[ ] = "--Vert\\N312X";
char Menu63[ ] = "--Bleu\\N313X";
char Menu64[ ] = "--Jaune\\N314X";
char Menu69[ ] = ". ";
                                            /* définition de quelques constantes */
#define mFichier    2
#define mForme      5
#define mCouleur    6
#define iQuitter    273
#define iRect       301
#define iOval       302
#define iRRect      303
#define iArc        304
#define iRouge      311
#define iVert       312
#define iBleu       313
#define iJaune      314

int     colfen[ ] = {0,0x0F00,0x020F,0xF0F0,0x00F0};    /* couleur des fenêtres */

ParamList maFen1 = {                                    /* la première fenêtre */
```

```
        sizeof(ParamList), 0x80A0, "\10Première", 0L,
        {0, 0, 0, 0}, colfen, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0,
        0L, 0, 0L, 0L, 0L,
        {30,10,150,155+160*mode}, -1L, 0L };

ParamList maFen2 = {                                    /* la seconde fenêtre */
        sizeof(ParamList), 0x80A0, "\10Deuxième", 0L,
        {0, 0, 0, 0}, colfen, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0,
        0L, 0, 0L, 0L, 0L,
        {60,166+160*mode,180,310+320*mode}, -1L, 0L };

Rect    theRect1 = {30,20,100,125+160*mode};    /* rectangle où on va dessiner */
Rect    theRect2 = {100,0,120,145+160*mode};    /* rectangle où on va écrire */

char    pat[4][16] = {                          /* pattern rouge/blanc (mode 640) */
        {0x77,0x77,0x77,0x77,0x77,0x77,0x77,0x77,
        0x77,0x77,0x77,0x77,0x77,0x77,0x77,0x77},
                                                /* pattern vert/blanc (mode 640) */
        {0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,
        0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB},
                                                /* pattern blanc/bleu (mode 640) */
        {0xDD,0xDD,0xDD,0xDD,0xDD,0xDD,0xDD,0xDD,
        0xDD,0xDD,0xDD,0xDD,0xDD,0xDD,0xDD,0xDD},
                                                /* pattern blanc/jaune (mode 640) */
        {0xEE,0xEE,0xEE,0xEE,0xEE,0xEE,0xEE,0xEE,
        0xEE,0xEE,0xEE,0xEE,0xEE,0xEE,0xEE,0xEE}
        };

int         indic = TRUE ;              /* indicateur de fin de boucle */
TaskRec     tache;                      /* ce que manipule GetNextEvent */
Pointer     wind, fen1, fen2;           /* pointeurs sur fenêtres */
Handle      barFen1, barFen2;           /* handles sur barres de menus */
Rect        menuRect;                   /* rectangle contenant chaque barre fenêtre */

int         forCh1 = iRect;             /* article forme coché fenêtre 1 */
int         colCh1 = iRouge;            /* article couleur coché fenêtre 1 */
int         forCh2 = iOval;             /* article forme coché fenêtre 2 */
int         colCh2 = iVert;             /* article couleur coché fenêtre 2 */

                /***** PROGRAMME PRINCIPAL *****/

    main( )
{
int     myID;                                   /* identifiant de l'application */

    myID = debut_appl(mode);                    /* initialisations standard */
    PlaceMenus( );                              /* installe la barre de menus système */
    OuvreFen( );                                /* ouverture des deux fenêtres */
    FlushEvents(EveryEvent, 0);                 /* ménage dans la file d'événements */

    do {
        if(!GetNextEvent(EveryEvent, &tache)) continue;
        switch(tache.what)
            {
            case MouseDown :                    /* souris */
                sourisDans(FindWindow(&wind, tache.where));
                break;

            case KeyDown :                      /* clavier */
                if (tache.modifiers & AppleKey)     /* touche Pomme enfoncée */
                    {
                    MenuKey(&tache, 0L);            /* appel menu système */
```



```
                    indic = ExecMenu(tache.TaskData);
                    }
                break;

            case UpdateEvt :                    /* fenêtre à mettre à jour */
                Ajour(tache.message);
                break;
            }
        }
    while(indic);

    SetBarColors(0x00F0,0x000F,0);              /* on rétablit les couleurs par défaut */
    quitter(myID);                              /* fin standard */
}

            /***** FONCTION SOURISDANS: réponse à un clic souris *****/

    sourisDans(code)

int     code;                                   /* code retourné par FindWindow */

{
long    pt;         /* un point, pour ne pas altérer le champ where de l'événement */

    switch(code)
        {
        case inMenuBar :
            MenuSelect(&tache, 0L);             /* appel menu système */
            indic = ExecMenu(tache.TaskData);
            break;

        case inContent :
            if (wind != FrontWindow( )) SelectWindow(wind);
            else
                {                               /* appel des différents menus fenêtre */
                pt = tache.where ;      /* lieu où la souris a été enfoncée (coord. globales) */
                SetPort(wind);          /* indispensable pour la conversion qui suit */
                GlobalToLocal(&pt);     /* passage en coordonnées locales */
                if (PtInRect(&pt, &menuRect)) /* le point est-il situé dans la barre fenêtre? */
                    {
                    if (wind == fen1) MenuSelect(&tache, barFen1);
                    else if (wind == fen2) MenuSelect(&tache, barFen2);
                    ExecMenu(tache.TaskData);
                    }
                }
            break;

        case inDrag :                           /* déplacement de fenêtre habituel */
            if (wind != FrontWindow( ) && !(tache.modifiers & AppleKey))
                SelectWindow(wind);
            DragWindow(0, tache.where, 0, 0L, wind); /* déplacement minimal par défaut */
            break;
        }
}

            /***** FONCTION OUVREFEN: ouvre 2 fenêtres avec barre de menus *****/

    OuvreFen( )

{
int     hMBar;                              /* hauteur des barres fenêtre */

    fen2 = NewWindow(&maFen2);              /* ouverture de la fenêtre 2... */
    barFen2 = NewMenuBar(fen2);             /* ...à laquelle on associe une barre de menus... */
```

```
    SetMenuBar(barFen2);                /* ...sur laquelle on va travailler immédiatement */
    SetTitleStart(16);                  /* on laisse 16 pixels à gauche de la barre */
    InsertMenu(NewMenu(Menu6), 0);      /* on inclut le menu Couleur */
    InsertMenu(NewMenu(Menu5), 0);      /* on inclut le menu Forme, à sa gauche */
    FixMenuBar( );                      /* on calcule la taille de la barre */
    CheckMItem(TRUE, iOval);            /* on coche la forme par défaut */
    CheckMItem(TRUE, iVert);            /* on coche la couleur par défaut */
        /* Note: on ne dessine pas la barre, c'est l'événement de mise à jour qui s'en charge! */

    fen1 = NewWindow(&maFen1);          /* idem fenêtre 1 */
    barFen1 = NewMenuBar(fen1);
    SetMenuBar(barFen1);
    SetTitleStart(16);
    InsertMenu(NewMenu(Menu6), 0);
    InsertMenu(NewMenu(Menu5), 0);
    hMBar = FixMenuBar( );              /* on mémorise la hauteur de la barre */
    CheckMItem(TRUE, iRect);
    CheckMItem(TRUE, iRouge);
    SetRect(&menuRect, 0, 0, 1000, hMBar);  /* le rectangle contenant la barre fenêtre */
}

        /***** FONCTION PLACEMENUS: installe la barre de menus système *****/

    PlaceMenus( )

{
    SetMenuBar(0L);                     /* on travaille sur la barre système */
    SetTitleStart(50);                  /* on commence à 50 pixels du bord */
    InsertMenu(NewMenu(Menu2), 0);      /* insertion du menu JP Curcio présente */
    FixMenuBar( );                      /* calcul de la taille de la barre système */
    if (!mode) SetBarColors(0x009D, 0x0097, 0x0020);
    DrawMenuBar( );             /* dessin immédiat (en couleurs si mode 320 uniquement) */
}

        /***** FONCTION EXECMENU: répond au choix d'un article de menu *****/


    int ExecMenu(art, menu)             /* retourne FALSE si quitter est choisi */

int  art;           /* article choisi */
int  menu;          /* dans ce menu */

{
char    msg[30];

    SetPort(GetWMgrPort( ));            /* on écrit dans le Window Manager port... */
    sprintf(msg,"article %d - menu %d   ",art,menu);    /* ...le menu et l'article choisis... */
    MoveTo(10,195); DrawCString(msg);                   /* ...en bas à gauche de l'écran */

    switch(art)
        {                               /* les valeurs parlent d'elles-mêmes! */
        case iQuitter:
            return FALSE;               /* l'application se termine bientôt! */
            break;

        case iRect:                     /* on va mémoriser la nouvelle forme... */
            SetWRefCon((long) (GetWRefCon(wind) & 0xF0) | 0x01, wind);
            ForMarque(art);             /* ...et faire les cochages nécessaires */
            break;

        case iOval:
            SetWRefCon((long) (GetWRefCon(wind) & 0xF0) | 0x02, wind);
            ForMarque(art);
            break;
```



```
        case iRRect:
            SetWRefCon((long) (GetWRefCon(wind) & 0xF0) | 0x03, wind);
            ForMarque(art);
            break;

        case iArc:
            SetWRefCon((long) (GetWRefCon(wind) & 0xF0) | 0x04, wind);
            ForMarque(art);

            break;

        case iRouge:                    /* on va mémoriser la nouvelle couleur... */
            SetWRefCon((long) (GetWRefCon(wind) & 0x0F) | 0x10, wind);
            ColMarque(art);             /* ...et faire les cochages nécessaires */
            break;

        case iVert:
            SetWRefCon((long) (GetWRefCon(wind) & 0x0F) | 0x20, wind);
            ColMarque(art);
            break;

        case iBleu:
            SetWRefCon((long) (GetWRefCon(wind) & 0x0F) | 0x30, wind);
            ColMarque(art);
            break;

        case iJaune:
            SetWRefCon((long) (GetWRefCon(wind) & 0x0F) | 0x40, wind);
            ColMarque(art);
            break;
        }

    if (art)                        /* s'il y a réellement choix d'un article */
        {
        SetPort(wind);              /* on sélectionne le grafport de la fenêtre active */
        InvalRect(&theRect1);       /* on rend invalide le rectangle contenant le dessin */
        InvalRect(&theRect2);       /* on rend invalide le rectangle contenant le texte */
        HiliteMenu(0, menu);        /* on rétablit le menu à son état normal */
        }
    return TRUE;                    /* l'application continue! */
}

        /***** FONCTION AJOUR: mise à jour d'une fenêtre *****/

    Ajour(port)

Pointer     port;                   /* pointeur sur la fenêtre à mettre à jour */

{
int     refcon;
char    msg[10];

    refcon = (int) GetWRefCon(port); /* récupère les caractéristiques du dessin à exécuter */
    SetPort(port);              /* on va dessiner dans le grafport de la fenêtre à rafraîchir */
    BeginUpdate(port);                          /* début de la mise à jour */
    if (port == fen1) SetMenuBar(barFen1);
    else if (port == fen2) SetMenuBar(barFen2);
    DrawMenuBar( );             /* on commence par redessiner la barre fenêtre */
    EraseRect(&theRect1);       /* on efface complètement le rectangle du dessin */
    EraseRect(&theRect2);       /* on efface complètement le rectangle du texte */
    switch (refcon & 0xF0)
        {
        case 0x10:                          /* rouge */
            if (mode) SetPenPat(pat[0]);    /* on fixe un pattern en mode 640 */
```

```
            else SetSolidPenPat(7);          /* ou la couleur rouge en mode 320 */
            sprintf(msg,"rouge");
            break;

        case 0x20:                           /* vert */
            if (mode) SetPenPat(pat[1]);
            else SetSolidPenPat(10);
            sprintf(msg,"vert");
            break;

        case 0x30:                           /* bleu */
            if (mode) SetPenPat(pat[2]);
            else SetSolidPenPat(4);
            sprintf(msg,"bleu");
            break;

        case 0x40:                           /* jaune */
            if (mode) SetPenPat(pat[3]);
            else SetSolidPenPat(9);
            sprintf(msg,"jaune");
            break;
        }
    MoveTo(100+160*mode,118); DrawCString(msg);          /* on écrit la couleur */

    switch (refcon & 0x0F)
        {
        case 0x01:                           /* rectangle */
            PaintRect(&theRect1);    /* on le dessine avec la couleur fixée plus haut */
            SetSolidPenPat(0); SetPenSize(2*(mode+1), 2);
            FrameRect(&theRect1);            /* et on souligne ses contours en noir */
            sprintf(msg,"Rectangle");
            break;

        case 0x02:                           /* ellipse */
            PaintOval(&theRect1);
            SetSolidPenPat(0); SetPenSize(2*(mode+1), 2);
            FrameOval(&theRect1);
            sprintf(msg,"Ovale");
            break;

        case 0x03:                           /* rect. arr. */
            PaintRRect(&theRect1, 40*(mode+1), 30);
            SetSolidPenPat(0); SetPenSize(2*(mode+1), 2);
            FrameRRect(&theRect1, 40*(mode+1), 30);
            sprintf(msg,"Rect. arr.");
            break;

        case 0x04:                           /* arc */
            PaintArc(&theRect1, 45, 290);
            SetSolidPenPat(0); SetPenSize(2*(mode+1), 2);
            FrameArc(&theRect1, 45, 290);
            sprintf(msg,"Arc");
            break;
        }
    MoveTo(10,118); DrawCString(msg);                    /* on écrit la forme */
    EndUpdate(port);                                     /* fin de la mise à jour */
}


    /***** FONCTION COLMARQUE: cochage des articles des menus Couleur *****/

    ColMarque(article)

int  article;

{
    if (wind == fen1)
        {
        CheckMItem(FALSE, colCh1); /* on retire la marque de la couleur précédente */
        CheckMItem(TRUE, article);     /* on coche la couleur choisie */
        colCh1 = article;              /* on mémorise la nouvelle couleur */
        }
    else if (wind == fen2)
        {
        CheckMItem(FALSE, colCh2);             /* idem */
        CheckMItem(TRUE, article);
        colCh2 = article;
        }
}

        /***** FONCTION FORMARQUE: cochage des articles des menus Forme *****/

    ForMarque(article)

int  article;

{
    if (wind == fen1)
        {
        CheckMItem(FALSE, forCh1);  /* on retire la marque de la forme précédente */
        CheckMItem(TRUE, article);  /* on coche la forme choisie */
        forCh1 = article;           /* on mémorise la nouvelle forme */
        }
    else if (wind == fen2)
        {
        CheckMItem(FALSE, forCh2);             /* idem */
        CheckMItem(TRUE, article);
        forCh2 = article;
        }
}
```


Exemple complet : menus et fenêtres

Dans cet exemple, nous allons voir l'utilisation d'une barre de menus système en conjonction avec l'activité des fenêtres (système et application). Le menu contient les accessoires de bureau. Quatre autres menus sont présents :

– le menu Fichier, toujours actif. Il permet d'ouvrir une nouvelle fenêtre (jusqu'à neuf possibles), de fermer une fenêtre ou un accessoire de bureau, et de quitter l'application ;

– le menu Edition, actif uniquement si une fenêtre système est au premier plan (l'application ne gère pas le copier-coller pour ses propres fenêtres, mais le permet entre accessoires de bureau) ;

– le menu Fenêtres, actif uniquement si une fenêtre de l'application est active. Chaque fois qu'une fenêtre est ouverte, son titre est ajouté dans ce menu. Chaque fois qu'une fenêtre est fermée, son titre est retiré de ce menu. De plus, sélectionner un article dans ce menu entraîne la mise au premier plan de la fenêtre correspondante ;

– le menu Spéciaux, actif uniquement si une fenêtre de l'application est active. Ce menu n'a aucune utilité. Il est là simplement pour illustrer quelques fonctions du Menu Manager : la directive \X (couleurs de remplacement), les styles que peuvent prendre les articles (remarquer le souligné, inutilisable avec la police système, et l'italique, que la version 1.02 de QuickDraw ne sait pas générer), le cochage et le changement de libellé pour un article.

Chaque fois que la barre des menus est sollicitée, le numéro de l'article et le numéro du menu sélectionnés sont affichés en bas de l'écran.

Le titre des fenêtres est généré automatiquement : un compteur incrémente le nombre de fenêtres ouvertes depuis le démarrage de l'application. Remarquer la double réservation d'espace mémoire, pour les titres de fenêtres et les libellés des articles correspondants. Les fenêtres sont décalées les unes par rapport aux autres à leur création. Elles affichent le contenu de la valeur d'utilisation libre, dont nous nous sommes servi comme lien avec le tableau de pointeurs sur fenêtres.

La barre de menus est bleue sur fond jaune en mode 320 (et rétablie en fin d'application), elle restera noire sur fond blanc en mode 640.

On notera la manière dont sont repérées les fenêtres : les éléments d'un tableau de dimension 9 contiennent soit zéro-long soit la valeur du pointeur repérant une fenêtre. L'indice de l'élément plus un est la valeur stockée dans le champ *wRefCon* de la fenêtre pointée. Cette valeur est également l'identifiant de l'article du menu Fenêtres correspondant (à une translation près).

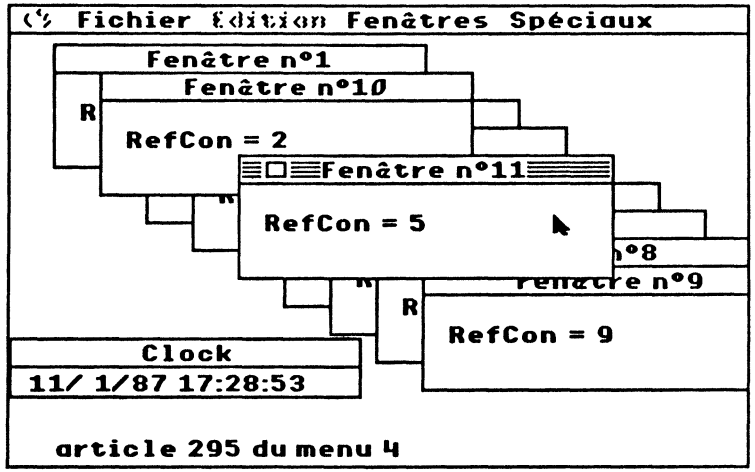


Figure VI.5 Neuf fenêtres et un accessoire de bureau ouverts.

```
#include <tools.h>                      /* définition des termes en gras */
#include <entete.h>                     /* définition des termes en italique */

#define mode 0                          /* 0 si mode 320, 1 si mode 640 */

char Menu1[ ] = ">>@\\XN1";                     /* menu  */
char Menu11[ ] = "--A propos de...\\N261VD";
char Menu19[ ] = ". ";
char Menu2[ ] = ">> Fichier \\N2";              /* menu Fichier */
char Menu21[ ] = "--Ouvrir\\N271*Oo";
char Menu22[ ] = "--Fermer\\N255D*Ff";
char Menu23[ ] = "--Quitter\\N273*Qq";
char Menu29[ ] = ". ";
char Menu3[ ] = ">> Edition \\N3D";             /* menu Edition */
char Menu31[ ] = "--Annuler\\N250V*Zz";
char Menu32[ ] = "--Couper\\N251*Xx";
char Menu33[ ] = "--Copier\\N252*Cc";
char Menu34[ ] = "--Coller\\N253*Vv";
char Menu35[ ] = "--Effacer\\N254";
char Menu39[ ] = ". ";
```



```
char Menu4[ ] = ">> Fenêtres \\N4D";            /* menu Fenêtres */
char Menu41[ ] = "-- \\N290";
char Menu49[ ] = ". ";
char Menu5[ ] = ">> Spéciaux \\N5XD";           /* menu Spéciaux */
char Menu51[ ] = "--Normal\\N301";
char Menu52[ ] = "--Gras\\N302B";
char Menu53[ ] = "--Italique\\N303I";
char Menu54[ ] = "--Souligné\\N304U";
char Menu55[ ] = "---\\N305D";
char Menu56[ ] = "--Coché\\N306C\22*Mm";
char Menu57[ ] = "--Estompé\\N307D";
char Menu58[ ] = "--Non standard\\N308X";
char Menu59[ ] = ". ";

char cochoui[ ] = "\6 Coché";                   /* un blanc obligatoire! */
char cochnon[ ] = "\12 Non coché";              /* idem */

#define mFichier    2                           /* définition de quelques constantes */
#define mEdition    3
#define mFenetres   4
#define mSpeciaux   5
#define iOuvrir     271
#define iFermer     255
#define iQuitter    273
#define iAnnuler    250
#define iCouper     251
#define iCopier     252
#define iColler     253
#define iEffacer    254
#define iFen        290
#define iCocher     306

int     colfen[ ] = {0,0x0F00,0x020F,0xF0F0,0x00F0};    /* couleur des fenêtres */

ParamList maFen = {                             /* la fenêtre de base (invisible) */
        sizeof(ParamList), 0xC080, "", 0L,
        {0, 0, 0, 0}, colfen, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0,
        0L, 0, 0L, 0L, 0L,
        {30,20,70,180+320*mode}, -1L, 0L };

char    titreF[9][14];                          /* 9 titres de 14 caractères maxi (fenêtres) */
char    titreM[9][15];                          /* 9 titres de 15 caractères maxi (menus) */
Pointer fen[9] = {0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L};  /* trace des différentes fenêtres */

int     indic = TRUE;                           /* indicateur de fin de boucle */
TaskRec tache;                                  /* ce que manipule GetNextEvent */
Pointer wind;                                   /* pointeur sur fenêtre */

                /***** PROGRAMME PRINCIPAL *****/

    main( )
{
int     myID;                                   /* identifiant de l'application */
Pointer savePort;                               /* pointeur sur grafport */
char    msg[15];

    myID = debut_appl(mode);                    /* initialisations standard */
    PlaceMenus( );                              /* installe la barre de menus */
    FlushEvents(EveryEvent, 0);                 /* le ménage dans la file d'événements */

    do {
        SystemTask( );                          /* pour les accessoires de bureau périodiques */
```

```
        if(!GetNextEvent(EveryEvent, &tache)) continue;      /* pas d'événement notable */
        switch(tache.what)                                   /* quel événement? */
            {
            case MouseDown :                                 /* bouton souris enfoncé */
                sourisDans(FindWindow(&wind, tache.where));
                break;

            case KeyDown :                                   /* clavier */
                if (tache.modifiers & AppleKey)              /* touche Pomme enfoncée */
                    {
                    MenuKey(&tache, 0L);                     /* quel article, quel menu? */
                    indic = ExecMenu(tache.TaskData);        /* réponse à la commande */
                    }
                break;

            case UpdateEvt :                                 /* mise à jour */
                savePort = GetPort( );
                SetPort(tache.message);                      /* on va écrire dans le bon grafport */
                BeginUpdate(tache.message);
                sprintf(msg,"RefCon = %d",(int) GetWRefCon(tache.message));
                MoveTo(10,20); DrawCString(msg);
                EndUpdate(tache.message);
                SetPort(savePort);                           /* on rétablit le grafport précédent */
                break;

            case ActivateEvt :                               /* activation ou désactivation */
                activer(tache.message);
                break;
            }
        }
    while(indic);

    SetBarColors(0x00F0,0x000F,0);                           /* couleurs standard pour les menus */
    quitter(myID);                                           /* fin standard */
}
        /***** FONCTION SOURISDANS: réponse à un clic souris *****/

    sourisDans(code)

int     code;                       /* code retourné par FindWindow */

{
    if (code<0)                                              /* dans un accessoire de bureau */
        {
        SystemClick(&tache, wind, code);                     /* manipulé par le Desk Manager */
        if ((code & 0xFF) == inGoAway) EtatMenu( );          /* on change la barre des menus */
        }
    else switch(code)
        {
        case inMenuBar :                                     /* dans la barre des menus */
            MenuSelect(&tache, 0L);                          /* quel article, quel menu? */
            indic = ExecMenu(tache.TaskData);                /* réponse à la commande */
            break;

        case inContent :                                     /* dans le contenu d'une fenêtre */
            if (wind != FrontWindow( )) SelectWindow(wind);
            break;

        case inDrag :                                        /* dans la barre de titre */
            if (wind != FrontWindow( ) && !(tache.modifiers & AppleKey))
                SelectWindow(wind);                          /* manipulations standard! */
            DragWindow(0, tache.where, 0, 0L, wind);
            break;
```



```
        case inGoAway :                                      /* dans la case de fermeture */
            fermer(wind);                                    /* on ferme! */
            break;
        }
}
        /***** FONCTION PLACEMENUS: installe la barre des menus *****/

    PlaceMenus( )

{
    InsertMenu(NewMenu(Menu5), 0);                           /* le menu Spéciaux */
    InsertMenu(NewMenu(Menu4), 0);                           /* à sa gauche, le menu Fenêtres */
    InsertMenu(NewMenu(Menu3), 0);                           /* à sa gauche, le menu Edition */
    InsertMenu(NewMenu(Menu2), 0);                           /* à sa gauche, le menu Fichier */
    InsertMenu(NewMenu(Menu1), 0);                           /* à sa gauche, le menu  */
    FixAppleMenu(1);                                         /* le nom des accessoires dans le menu  */
    FixMenuBar( );                                           /* calcul des dimensions de la barre des menus */
    if (!mode) SetBarColors(0x009D, 0x0097, 0x0020); /* en couleurs si mode 320 */
/*  DeleteItem(iFen);                                        quand cela marchera!
    CalcMenuSize(0, 0, mFenetres);                           */
    DrawMenuBar( );                                          /* on dessine la barre des menus */
}
        /***** FONCTION EXECMENU: répond au choix d'un article de menu *****/

    int ExecMenu(art, menu)                  /* retourne FALSE si quitter est choisi */

int  art;                /* article choisi */
int  menu;               /* dans ce menu */

{
int     flag = TRUE;
char    msg[30];
                                             /* le port par défaut est le Window Manager port */
    sprintf(msg,"article %d du menu %d  ", art, menu);
    MoveTo(20,195); DrawCString(msg);        /* on écrit en bas de l'écran */

    if (art>249)                             /* article créé par l'application */
    switch (art)
        {
        case iOuvrir:
            ouvrir( );                       /* ouverture d'une fenêtre de l'application */
            break;

        case iFermer:
            fermer(FrontWindow( ));          /* fermeture de la fenêtre active */
            break;

        case iQuitter:
            flag = FALSE;                    /* on quittera l'application */
            break;

        case iAnnuler:
            SystemEdit(Undo);                /* géré par le Desk Manager */
            break;

        case iCouper:
            SystemEdit(Cut);                 /* géré par le Desk Manager */
            break;

        case iCopier:
            SystemEdit(Copy);                /* géré par le Desk Manager */
            break;
```

```
    case iColler:
        SystemEdit(Paste);          /* géré par le Desk Manager */
        break;

    case iEffacer:
        SystemEdit(Clear);          /* géré par le Desk Manager */
        break;

    case iFen+1:
    case iFen+2:
    case iFen+3:
    case iFen+4:
    case iFen+5:
    case iFen+6:
    case iFen+7:
    case iFen+8:
    case iFen+9:
        SelectWindow(fen[art-iFen-1]);          /* activation de la fenêtre choisie */
        break;

    case iCocher:
        if(GetItemMark(iCocher))                /* l'article est-il coché? */
            {                                   /* oui... */
            CheckMItem(FALSE, iCocher); /* ...on retire la marque... */
            SetItem(cochnon, iCocher);          /* ...et on change le libellé */
            }
        else
            {                                   /* non... */
            CheckMItem(TRUE, iCocher);  /* ...on met la marque... */
            SetItem(cochoui, iCocher);          /* ...et on change le libellé */
            }
        CalcMenuSize(0, 0, mSpeciaux);  /* recalcul du menu touché */
        break;
    }
else if (art>0)
    {
    OpenNDA(art);                       /* ouverture accessoire de bureau */
    EnableMItem(iFermer);           /* article Fermer autorisé */
    EtatMenu( );
    }

if (art) HiliteMenu(0, menu);           /* on rétablit l'état normal du menu */

    return flag;
}

            /***** FONCTION OUVRIR: ouverture d'une nouvelle fenêtre *****/

    ouvrir( )

{
static  compteur;                       /* s'incrémente à chaque ouverture de fenêtre */
int     i;
Rect    r;                              /* un rectangle */

    for (i=0; i<9; ++i)
        {
        if (fen[i] == 0L)
            {
            fen[i] = NewWindow(&maFen);             /* nouvelle fenêtre (invisible)... */
                                                    /* ...dont on change le titre */
            sprintf(titreF[i],"Fenêtre n°%d",++compteur);
            ctopstr(titreF[i]);                     /* conversion C -> Pascal */
            SetWTitle(titreF[i], fen[i]);
```



```
                                                    /* ... dont on change la localisation */
            MoveWindow(20+20*(mode+1)*i, 30+12*i, fen[i]);
                                                    /* ...dont on change le champ wRefCon */
            SetWRefCon((long) i+1, fen[i]);
                                                    /* on fait apparaître la fenêtre */
            SelectWindow(fen[i]);
            ShowWindow(fen[i]);
                                                    /* manip sur le menu  Fenêtres */
            InsertItem(Menu41, -1, mFenetres);
            SetItemID(iFen+i+1, iFen);
            sprintf(titreM[i]," Fenêtre n°%d",compteur); /* un blanc de différence! */
            ctopstr(titreM[i]);                     /* conversion C -> Pascal */
            SetItem(titreM[i], iFen+i+1);           /* on change le libellé de l'article */
            CalcMenuSize(0, 0, mFenetres);  /* recalcul de la taille du menu Fenêtres */
            EnableMItem(iFermer);                   /* article Fermer autorisé */
            break;
            }
        }

    for (i=0; i<9; ++i) if (fen[i] == 0L) return;       /* si 9 fenêtres sont ouvertes... */
    DisableMItem(iOuvrir);                              /* ...l'article Ouvrir est estompé */
    /* ces deux dernières lignes pourront être évitées quand CountMItems remplira son rôle...
    On écrira:
            if (CountMItems(mFenetres) == 1) DisableMItem(iOuvrir);
    L'égalité à 1 se transformant en égalité à 0 si les menus vides d'articles sont tolérés */
}

            /***** FONCTION FERMER: fermeture d'une fenêtre *****/

    fermer(port)

Pointer port;                               /* pointeur sur la fenêtre à fermer */

{
int     ind;

    if (port == 0L) return;                 /* aucune fenêtre */
    else if (GetWKind(port)) CloseNDAbyWinPtr(port);  /* accessoire de bureau */
    else
        {                                   /* fenêtre de l'application */
        ind = (int) GetWRefCon(port);       /* laquelle? */
        fen[ind-1] = 0L;                    /* on annule son pointeur */
        CloseWindow(port);                  /* on ferme la fenêtre */
        EnableMItem(iOuvrir);               /* article Ouvrir autorisé */
        DeleteItem(iFen + ind);         /* article correspondant à la fenêtre supprimé */
        CalcMenuSize(0, 0, mFenetres);  /* recalcul de la taille du menu Fenêtres */
        }

    if (FrontWindow( ) == 0L)
        DisableMItem(iFermer);          /* s'il n'y a plus rien à fermer, on estompe! */
    EtatMenu( );                            /* on change la barre des menus */
}

            /***** FONCTION ACTIVER: activation ou désactivation d'une fenêtre *****/

    activer(port)

Pointer port;       /* pointeur sur la fenêtre à activer ou désactiver */
                    /* seules les fenêtres de l'application sont touchées */

{
int     ind;

ind = (int) GetWRefCon(port);
if (tache.modifiers & ActiveFlag)           /* si activation... */
```

```
        CheckMItem(TRUE, iFen + ind);          /* ...fenêtre cochée */
    else                                       /* si désactivation... */
        CheckMItem(FALSE, iFen + ind);         /* ...fenêtre non cochée */

    EtatMenu( );                               /* on change la barre des menus */
}

/***** FONCTION ETATMENU: transforme la barre des menus
                                               en fonction de la fenêtre active *****/

    EtatMenu( )

{
static  etatprec;              /* le type de la fenêtre active précédente */
int     etatact;               /* le type de la fenêtre active actuelle */

    etatact = (FrontWindow( ) == 0L) ? 0 : (GetWKind(FrontWindow( )) ? -1 : 1);
    if (etatact == etatprec) return;       /* on ne fait rien si le type n'a pas changé */
                                           /* sinon... */
    if (etatact > 0)                       /* si la fenêtre active appartient à l'application */
        {
        SetMenuFlag(DisableMenu, mEdition); /* menu Edition désactivé */
        SetMenuFlag(EnableMenu, mFenetres);        /* menu Fenêtres activé */
        SetMenuFlag(EnableMenu, mSpeciaux);        /* menu Spéciaux activé */
        }
    else if (etatact < 0)                  /* si la fenêtre active est un accessoire de bureau */
        {
        SetMenuFlag(EnableMenu, mEdition);  /* menu Edition activé */
        SetMenuFlag(DisableMenu, mFenetres);       /* menu Fenêtres désactivé */
        SetMenuFlag(DisableMenu, mSpeciaux);       /* menu Spéciaux désactivé */
        }
    else                                   /* s'il n'y a plus de fenêtre ouverte */
        {
        SetMenuFlag(DisableMenu, mEdition); /* menu Edition activé */
        SetMenuFlag(DisableMenu, mFenetres);       /* menu Fenêtres désactivé */
        SetMenuFlag(DisableMenu, mSpeciaux);       /* menu Spéciaux désactivé */
        }

    etatprec = etatact;                    /* on mémorise l'état actuel... */
    DrawMenuBar( );                        /* ...et on redessine la barre des menus */
}
```

CHAPITRE VII

CONTROL MANAGER

PRINCIPES GÉNÉRAUX

Les contrôles sont omni-présents dans les applications respectant l'interface utilisateur Apple, mais le plus souvent, c'est un gestionnaire particulier qui les gère plutôt que l'application elle-même. Qu'est-ce qu'un contrôle ? C'est un objet qui apparaît à l'écran avec lequel l'utilisateur, grâce à la souris, peut soit provoquer une action instantanée, soit changer des paramètres pour modifier une action future.

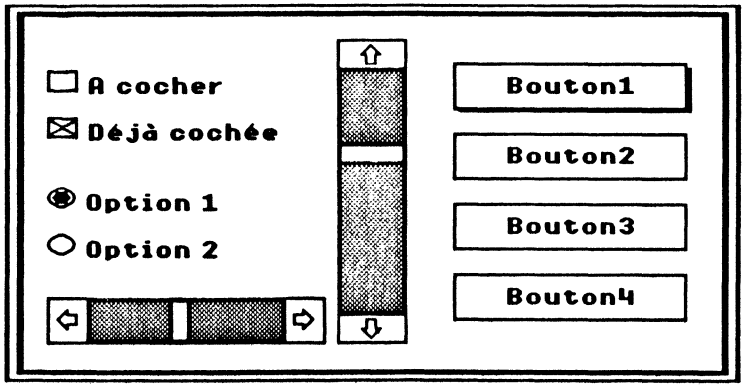


Figure VII.1. Les contrôles standard.

Le Control Manager gère les contrôles : il en permet l'affichage ou le masquage, s'occupe de l'aspect visuel des actions de l'utilisateur, garde en mémoire la valeur qu'ils prennent...

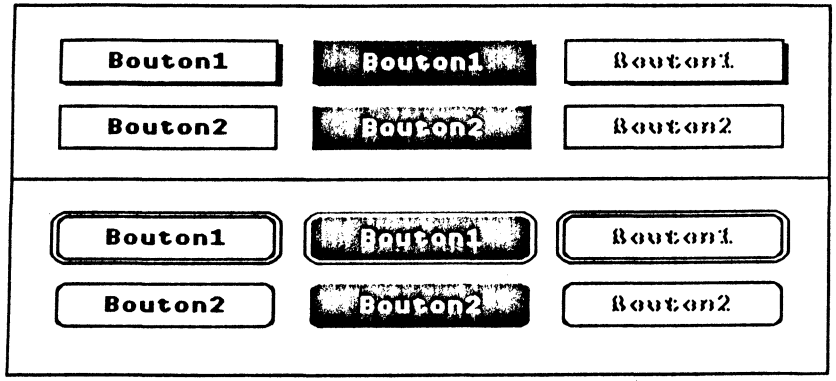


Figure VII.2. Les différents aspects des boutons simples des deux types (normal, en lumière, estompé).

Il existe deux grandes catégories de contrôles : ceux qui peuvent prendre au plus deux valeurs (catégorie des boutons) et ceux qui peuvent prendre un nombre fini (supérieur à deux) de valeurs (catégorie des cadrans).

Dans la première catégorie, trois types de contrôles sont prédéfinis :

– le bouton simple (ou case) peut avoir deux aspects : rectangle normal ou rectangle à bords arrondis, contenant un titre (centré). Le fait de cliquer dans un bouton cause la mise en lumière de l'intérieur du bouton, et si le bouton de la souris est relâché à l'intérieur, une action immédiate ou continue doit s'ensuivre. Optionnellement, un trait fort peut entourer le bouton arrondi ou une ombre accompagner le bouton non arrondi, signifiant que l'appui sur la touche Retour ou Entrée équivaut à un clic souris dans ce bouton (bouton par défaut) ;

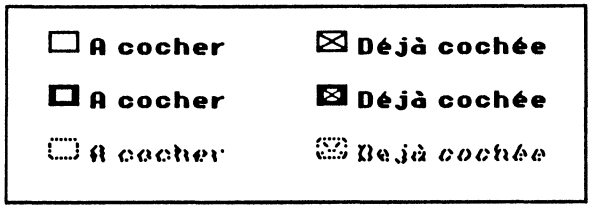


Figure VII.3. Les différents aspects des cases à cocher (normal, en lumière, estompé).

– la case à cocher est un petit carré avec un titre à sa droite. Le fait de cliquer (et surtout de relâcher le bouton) dans une telle case fait alternativement apparaître ou disparaître une croix, précisant une option sélectionnée ou pas. Typiquement, une telle action est rarement suivie d'un effet immédiat, mais permet la modification d'une action future ;

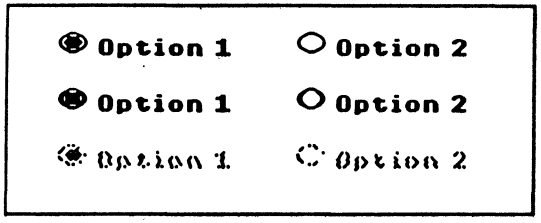


Figure VII.4. Les différents aspects des boutons radio (normal, en lumière, estompé).

– le bouton radio est un petit cercle avec un titre à sa droite. Il n'est jamais seul mais fait partie d'une famille de boutons radio. Le fait de cliquer dans l'un des boutons fait apparaître à l'intérieur du cercle un gros point, et disparaître le point du précédent bouton sélectionné. En effet, dans une famille de boutons radio, seule une option peut être sélectionnée à la fois : les options sont exclusives. Comme pour la case cochée, la famille de boutons radio permettra d'agir sur une action future, plutôt que de déclencher une action immédiate.

Pour chacun des trois types, le contrôle peut être momentanément désactivé, à l'instar des articles des menus déroulants. Le titre d'un contrôle inactif est estompé, et aucune action de la souris ne peut être suivie d'effet dans un tel bouton de contrôle.

Un seul type de contrôle est prédéfini dans la catégorie des cadrans : la barre de défilement. Une barre de défilement est un objet complexe, composé de plusieurs parties : flèches de défilement, bande de défilement, curseur de défilement. Le curseur se déplace dans la bande, délimitant avec les flèches deux régions (parfois vides) de défilement de page. Chacune des parties de la barre de défilement réagit à sa manière.

Une barre de défilement est généralement associée à une fenêtre, mais peut servir à représenter n'importe quelle quantité finie : la longueur de la bande de défilement représente la quantité totale, la taille du curseur représente la quantité visible ou affichée (en proportion de la taille totale), la position du curseur représente la position de la partie visible ou affichée vis-à-vis de l'ensemble.

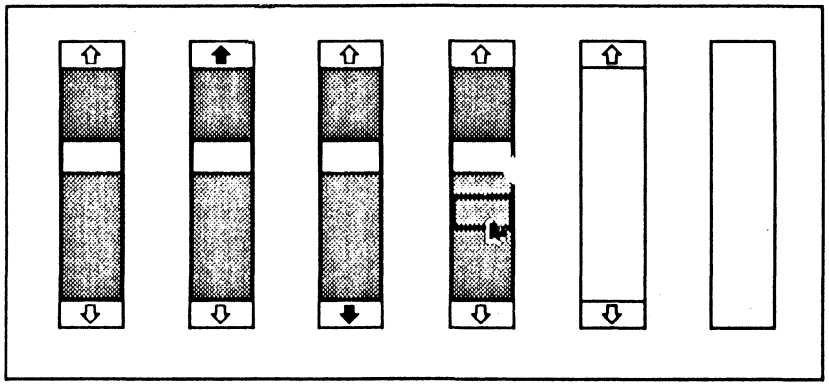


Figure VII.5. Les différents aspects des barres de défilement (normal, sélectionné de trois manières, inactif de deux manières).

Pour l'action de défilement, il faut définir une unité (par exemple une ligne pour une fenêtre contenant du texte, ou la valeur 1 pour une barre représentant un état variant de 0 à 15). Quand l'utilisateur cliquera dans une flèche de défilement, il déplacera son document d'une unité (défilement d'une ligne pour la fenêtre de texte, incrémentation ou décrémentation de 1 pour l'état de 0 à 15). La position du curseur sera modifiée en conséquence.
Attention Si l'utilisateur garde la souris enfoncée dans la flèche, l'action de défilement doit se poursuivre périodiquement, jusqu'à ce qu'il relâche le bouton de la souris.

Quand l'utilisateur cliquera dans la bande de défilement entre le curseur et l'une des flèches, il déplacera son document en une seule fois d'un nombre déterminé d'unités (par exemple défilement d'une page pour la fenêtre de texte, incrémentation ou décrémentation de 4 pour l'état de 0 à 15). Là aussi, la position du curseur sera modifiée en conséquence. Là encore, si l'utilisateur garde la souris enfoncée dans la bande, l'action de défilement doit se poursuivre périodiquement, jusqu'à ce qu'il relâche le bouton de la souris. Les deux régions comprises entre le curseur *(Thumb)* et les flèches (quand elles existent) sont appelées *PageUp* et *PageDown*.

Enfin, l'utilisateur a la possibilité de faire glisser le curseur jusqu'à la position qu'il désire. Dès qu'il aura relâché le curseur dans une nouvelle position, le document devra être déplacé proportionnellement au déplacement du curseur.

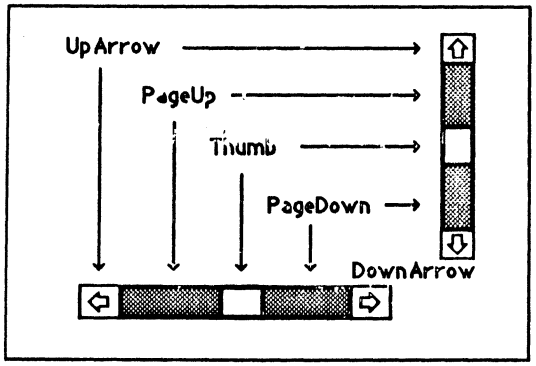


Figure VII.6. Les différentes parties des barres de défilement.

Si vous souhaitez définir vos propres contrôles, aucun problème : le Control Manager vous le permet ! Mais nous n'entrerons pas dans ce genre de considérations. Reportez-vous à la documentation technique pour définir des contrôles autres que les boutons et la barre de défilement.

UTILISATION DU CONTROL MANAGER

Contrôles et fenêtres

Nous avons vu dans le chapitre consacré au Window Manager qu'une fenêtre est créée avec un certain nombre de contrôles standard, définissant l'aspect de sa région contour. Nous avons dit que ces contrôles sont entièrement gérés par le Window Manager, qui utilise un grafport particulier, le *Window Manager port,* pour les dessiner.

On ne confondra pas les contrôles gérés par le Window Manager et les contrôles créés et gérés directement par l'application.

Chaque contrôle créé par l'application appartient à une fenêtre. Quand un contrôle doit être affiché, il apparaît dans la région contenu de la fenêtre. Quand il est manipulé par l'action de la souris, il agit dans cette fenêtre. En conséquence, toutes les coordonnées concernant un contrôle seront définies dans le système de coordonnées locales de la fenêtre.

Les contrôles peuvent être définis dans la zone d'informations d'une fenêtre, mais l'opération présentant la même complexité que la définition des menus déroulants, nous n'en parlerons pas davantage.

Comment la fenêtre garde-t-elle trace de tous ses contrôles ? C'est très simple : elle ne garde en fait trace que du premier contrôle qui lui est défini, par l'intermédiaire d'un handle, et chaque contrôle garde trace du contrôle qui le suit par l'intermédiaire également d'un handle. Deux listes sont associées à chaque fenêtre : la liste des contrôles gérés par le Window Manager, et la liste des contrôles gérés par l'application. De la sorte, les contrôles forment deux chaînes dans lesquelles Window Manager et Control Manager savent se retrouver.

Remarque Avec quelques manipulations pas très orthodoxes, on peut échanger les deux listes, ce qui permet à l'application de manipuler les contrôles créés par le Window Manager. Nous ne donnerons pas cet exemple : plutôt que de jouer à l'apprenti sorcier, il vaut mieux laisser **TaskMaster** faire défiler les fenêtres, ou alors créer soi-même les barres de défilement et programmer l'Apple IIGS comme un Macintosh.

Caractéristiques de contrôles

Tout contrôle possède une structure de stockage interne qui possède les éléments suivants (description non exhaustive) :

- un handle sur le contrôle suivant, pour assurer les chaînages (le Control Manager se débrouille).
- un pointeur sur la fenêtre à laquelle il appartient.
- un rectangle dans lequel il sera dessiné : pour un bouton, ce rectangle englobera la case de contrôle et le titre. Quand on demandera de rendre un contrôle invisible, c'est ce rectangle qui sera effacé. Le rectangle précise à la fois la taille et la localisation du contrôle dans la fenêtre, il sera toujours exprimé dans son système de coordonnées locales.
- un entier précisant quelques caractéristiques internes : si le contrôle est visible ou invisible, s'il est actif ou inactif. D'autres caractéristiques dépendent du type du contrôle : bouton par défaut ou pas en cas des boutons simples, famille d'appartenance pour les boutons radio, composantes et orientation des barres de défilement...
- la valeur courante associée au contrôle : toujours 0 pour un bouton, 1 ou 0 pour une case cochée ou non, un bouton radio sélectionné ou pas, variable pour une barre de défilement, entre deux bornes.
- les données propres au contrôle : un pointeur sur le titre pour un bouton, le montant total de données et le montant qui peut être vu pour une barre (par exemple 500 et 20 si un document texte fait 500 lignes mais que la fenêtre ne peut en montrer que 20 à la fois).
- l'adresse d'une procédure d'action à suivre quand l'utilisateur invoquera le contrôle.
- l'adresse d'une procédure de définition du contrôle (prédéfinie pour les contrôles standard, à fournir pour les contrôles définis par l'application).
- un entier long que l'application pourra utiliser comme bon lui semble.
- un pointeur sur la table de couleurs qui servira à représenter les contrôles.

EXEMPLES D'UTILISATION

Initialisation, création de contrôles

```
...
CtlStartUp(myID, zeropg);                /* initialisation du Control Manager */
...
```

La procédure **CtlStartUp** doit être appelée avant toute utilisation du Control Manager, et après initialisation du Window Manager. Comme d'habitude, le premier argument est l'identifiant de l'application tel qu'il a été retourné par la fonction

MMStartUp du Memory Manager. Le second argument désigne la page zéro qui servira en usage interne au Control Manager. Voir le chapitre XII pour la séquence exacte d'initialisation des outils.

Pour créer un contrôle que l'application va pouvoir manipuler comme bon lui semble, il faut déjà avoir créé une fenêtre, puisque tout contrôle appartient à une fenêtre. On utilise alors la fonction **NewControl**, qui ajoute ce nouveau contrôle à la liste des contrôles de la fenêtre. Celle-ci ne réclame pas moins de dix arguments, et retourne un handle qui identifiera le contrôle pour son utilisation ultérieure (ou la valeur zéro-long en cas d'erreur à la création). Voyons le détail des dix arguments, en fonction du type de contrôle à créer :

– premier argument : un pointeur sur la fenêtre à laquelle appartiendra le contrôle, tel qu'il a été retourné par **NewWindow** ;

– deuxième argument : un pointeur sur le rectangle qui englobera complètement le contrôle et le localisera à l'intérieur de la fenêtre. Les coordonnées du rectangle doivent être exprimées en coordonnées locales de la fenêtre d'appartenance. Pour un simple bouton, ce rectangle définit la taille exacte du bouton (et devrait avoir une hauteur d'au moins 20 pixels). Pour les autres types de boutons, une hauteur de 16 pixels suffit. Pour une barre de défilement, il faut prévoir au minimum 48 pixels dans le sens de la longueur pour que les flèches et le curseur de défilement puissent être dessinés, et une largeur d'au moins 12 pixels. Si on donne une largeur de trente pixels, la barre fera trente pixels et paraîtra énorme ;

– troisième argument : un pointeur sur le titre du contrôle. Une barre de défilement n'ayant pas de titre, on passera simplement un pointeur sur une chaîne de caractères vide. Comme d'habitude, le titre est une chaîne de caractères de type Pascal ;

– quatrième argument : l'entier décrivant les caractéristiques du contrôle. Seuls quelques bits sont définis dans cet entier, en fonction du type de contrôle (laisser à zéro les bits non cités).

Remarque Si les bits 8 à 15 prennent la valeur $FF, le contrôle sera inactif.

- simple bouton. Le bit 0 est à 1 si le bouton doit être ombré ou entouré d'un trait gras, signifiant qu'il s'agit du bouton par défaut. Le bit 1 donne la forme du bouton : 0 pour un bouton arrondi, 1 pour un rectangle normal. Le bit 7 est à 1 si le bouton est invisible.
- case à cocher. Le bit 7 est à 1 si la case est invisible.
- bouton radio. Les bits 0 à 6 donnent le numéro de la famille à laquelle appartient le bouton. Le bit 7 est à 1 si le bouton est invisible.
- barre de défilement. Le bit 0 indique la présence ou non d'une flèche de défilement vers le haut, le bit 1 d'une flèche vers le bas, le bit 2 d'une flèche vers la gauche, le bit 3 d'une flèche vers la droite. Le bit 4 est à zéro si la barre est verticale, à un si la barre est horizontale. Le bit 7 est à 1 si la barre est invisible.

– cinquième argument : la valeur prise par le contrôle à sa création (entier sur 16 bits). Toujours 0 pour un simple bouton, TRUE ou FALSE pour les autres catégories de boutons (mais nous utiliserons plutôt 1 ou 0), quelconque pour une barre de défilement (mais dans un intervalle cohérent avec les arguments suivants) ;

– sixième argument : taille de la partie visualisée, encore appelée, mais improprement, taille minimale (cas d'une barre de défilement) ;

– septième argument : taille de l'ensemble des données, encore appelée, mais improprement, taille maximale (cas d'une barre de défilement) ;

– huitième argument : adresse de la procédure de définition (dans le cas des types de contrôles non standard) ou valeur prédéfinie pour les contrôles standard : *SimpleProc* pour les simples boutons, *CheckProc* pour les cases à cocher, *RadioProc* pour les bouton radios et *ScrollProc* pour les barres de défilement ;

```
#define SimpleProc      0L
#define CheckProc       0x02000000
#define RadioProc       0x04000000
#define ScrollProc      0x06000000
```


– neuvième argument : entier long d'utilisation libre, dans lequel l'application qui crée un contrôle peut stocker n'importe quelle valeur ;

– dixième argument : pointeur sur une table de couleur définissant quelles couleurs seront employées pour dessiner les contrôles (zéro-long désigne la table par défaut). La taille et le contenu de la table sont différents pour chaque type de contrôle. Chaque couleur est définie sur un entier de 16 bits.

Note La documentation est pratiquement inexistante sur le sujet, les renseignements ci-après sont sans doute incomplets ;

- simple bouton. 7 couleurs sont définies : couleur du bord (bits 4 à 7), couleur de l'intérieur à l'état normal (bits 4 à 7), couleur de l'intérieur si sélectionné (bits 4 à 7), couleur du titre si normal (bits 0 à 3 : *ForeColor*, bits 4 à 7 : *BackColor*), couleur du titre si sélectionné (bits 0 à 3 : *ForeColor*, bits 4 à 7 : *BackColor*), couleur spéciale de mise en lumière (???), couleur du double trait pour le bouton par défaut (???)
- case à cocher. 4 couleurs sont définies : la première n'est pas utilisée, couleur de la case à l'état normal (bits 0 à 3 : les traits de la case, bits 4 à 7 : l'intérieur de la case), couleur de la case « en lumière » (bits 0 à 3 : les traits de la case, bits 4 à 7 : l'intérieur de la case), couleur du titre (bits 0 à 3 : *ForeColor*, bits 4 à 7 : *BackColor)*
- bouton radio. Idem case à cocher.
- barre de défilement, 8 couleurs sont définies : couleur du bord (bits 4 à 7), couleur des flèches à l'état normal (bits 0 à 3 : le contour de la flèche, bits 4 à 7 : le reste de la boîte, mais il y a d'autres subtilités), couleur des flèches « en lumière » (bits 0 à 3 : toute la flèche, bits 4 à 7 : tout ce qu'il y a autour), couleur dans le rectangle englobant la flèche (???), couleur du curseur à l'état normal (bits 4 à 7 : intérieur du curseur), couleur du curseur si sélectionné (???), couleur de la bande de défilement (bit 8 : s'il est nul, la bande est de couleur pleine, définie dans les bits 4 à 7 : s'il n'est pas nul, la bande est constituée de points de la couleur définie dans les bits 4 à 7 sur fond de couleur définie dans les bits 0 à 3), couleur de la barre si inactive.

Nous verrons des exemples de contrôles en couleur dans l'exemple complet en fin du chapitre IX consacré au Dialog Manager.

Remarque

– la fonction **NewControl** ne dessine pas le contrôle mais génère un événement de mise à jour, il sera donc judicieux d'inclure un appel à la procédure **DrawControls** (voir plus loin) en réponse à ces événements !

– aucun argument dans **NewControl** ne permettant de définir l'éventuel pointeur sur la procédure d'action (cas des barres de défilement), il faudra appeler explicitement **SetCtlAction** (voir plus loin) pour fixer cette valeur.

A titre d'illustration, les lignes suivantes créent un contrôle visible et actif de chaque type, en noir et blanc (on suppose que la fenêtre d'accueil existe) :

```
Handle  bouton, check, radio1, radio2, barreH, barreV;      /* handles sur contrôles */
Pointer wind;                                               /* pointeur sur fenêtre */
Rect    r;                                                  /* un rectangle */

SetRect(&r,10,10,110,30);
bouton = NewControl(wind,&r,"\6Bouton",3,0,0,0,SimpleProc,0L,0L);
SetRect(&r,10,40,110,56);
check = NewControl(wind,&r,"\10A cocher",0,0,0,0,CheckProc,0L,0L);
SetRect(&r,10,70,110,86);
radio1 = NewControl(wind,&r,"\10Option 1",10,0,0,0,RadioProc,0L,0L);
SetRect(&r,10,90,110,106);
radio2 = NewControl(wind,&r,"\10Option 2",10,1,0,0,RadioProc,0L,0L);
SetRect(&r,10,120,130,136);
barreH = NewControl(wind,&r,"",0x1C,5,1,20,ScrollProc,0L,0L);
SetRect(&r,140,10,156,130);
barreV = NewControl(wind,&r,"",0x03,25,20,200,ScrollProc,0L,0L);
DrawControls(wind);
```

Dans la suite, un contrôle sera toujours repéré par le handle retourné par la fonction **NewControl**. S'il arrive un moment où l'application n'a plus besoin d'un contrôle, elle pourra libérer la place qu'il occupe en mémoire grâce à **DisposeControl**. En outre, cette procédure efface de l'écran et retire de la liste des contrôles de la fenêtre le contrôle dont le handle est passé en argument.

```
Handle barreH, barreV;

DisposeControl(barreH);          /* plus de barre horizontale */
DisposeControl(barreV);          /* plus de barre verticale */
```

Encore plus draconienne, la procédure **KillControls** détruit tous les contrôles associés à la fenêtre dont le pointeur est passé en argument. Là encore, seuls sont détruits les contrôles créés par l'application, et non ceux qui ont été créés pour la fenêtre par le Window Manager. Notons que cette procédure est automatiquement appelée par **CloseWindow**, quand l'application demande la fermeture de la fenêtre.

En réponse au Window Manager

- Après un événement de type *MouseDown*.

On se souvient de la boucle d'événements : quand l'utilisateur a enfoncé le bouton de la souris, il a provoqué un événement de type *MouseDown*. Grâce à **FindWindow**, cet événement a pu être localisé. Supposons qu'il se soit produit dans le contenu d'une fenêtre qui contient des contrôles gérés directement par l'application, c'est au tour du Control Manager d'agir :

```
void    Defilement( );                  /* déclaration de la procédure d'action */
Handle  ctl;                            /* handle désignant un contrôle */
Pointer theWindow;                      /* pointeur sur une fenêtre */
int     part1, part2;                   /* partie du contrôle concernée */

/* après l'appel à FindWindow */
case inContent :                        /* MouseDown dans la région contenu d'une fenêtre */
   if (theWindow != FrontWindow( ))     /* cette fenêtre est-elle active? */
      SelectWindow(theWindow); /* non, alors on l'active et on ne fait rien de plus */
   else
   part1 = FindControl(&ctl, tache.where, theWindow);    /* clic dans un contrôle? */
   if (!part1)
      ...       /* non, l'application répond comme elle doit répondre */
   else         /* ouil */
      {
      part2 = TrackControl(tache.where, Defilement, ctl); /* manipulation du contrôle */
      if (part2==part1)                 /* si le clic est valide... */
         switch(part1)
            {
            case SimpleButton :         /* est-ce dans un bouton simple? */
            ...                         /* oui, identification et réponse appropriée */
            break;

            case CheckBox :             /* est-ce dans une case à cocher? */
            ...                         /* oui, identification et réponse appropriée */
            break;

            case RadioButton :          /* est-ce dans un bouton radio? */
            ...                 /* oui, identification de sa famille et réponse appropriée */
            break;
            }
      }
break;
```


Après que l'on ait vérifié que le clic a eu lieu dans la fenêtre active, la fonction **FindControl** est appelée. Quatre arguments : le premier donne l'adresse mémoire où la fonction va stocker le handle désignant le contrôle dans lequel l'utilisateur a cliqué. Les deuxième et troisième arguments sont l'abscisse et l'ordonnée du point où a eu lieu le clic, en coordonnées globales, tel qu'il a été enregistré dans l'événement de type *MouseDown* (et comme d'habitude, ces deux entiers peuvent être condensés en un seul entier long). Le quatrième argument est un pointeur sur la fenêtre à laquelle le point appartient.

Si l'utilisateur a réellement cliqué dans un contrôle, le handle désignant ce contrôle est stocké à l'adresse désignée par le premier argument, et la fonction **FindControl** retourne en résultat explicite un code identifiant la partie du contrôle dans laquelle on a cliqué.

Si l'utilisateur n'a pas cliqué dans un contrôle, c'est la valeur zéro-long qui est stockée à l'adresse désignée par le premier argument, et la fonction **FindControl** retourne la valeur zéro.

Les valeurs susceptibles d'être retournées par **FindControl** sont prédéfinies : elles indiquent une partie de contrôle dans laquelle l'utilisateur a cliqué.

```
#define NoPart          0       /* aucune partie de contrôle */
#define SimpleButton    2       /* bouton simple */
#define CheckBox        3       /* case à cocher */
#define RadioButton     4       /* bouton radio */
#define UpArrow         5       /* flèche du haut ou de gauche d'une barre */
#define DownArrow       6       /* flèche du bas ou de droite d'une barre */
#define PageUp          7       /* région PageUp d'une barre de défilement */
#define PageDown        8       /* région PageDown d'une barre de défilement */
#define Thumb           129     /* curseur de défilement d'une barre */
```

Les autres valeurs sont soit réservées à l'usage interne du système, soit utilisables par les applications qui définissent leurs propres types de contrôles.

Une fois que la fonction **FindControl** a retourné les renseignements qu'on attend d'elle, il reste à gérer le contrôle, et c'est la fonction **TrackControl** qui s'en charge. Dès que le bouton de la souris est pressé dans un contrôle visible et actif, cette fonction est appelée : elle suit les mouvements de la souris et agit de manière appropriée jusqu'à ce que le bouton soit relâché. S'il y a des parties de contrôles à mettre en lumière, elle le fait, si un curseur de défilement doit être déplacé, il l'est. Par contre, **TrackControl** n'assure pas toute seule le défilement du contenu de la fenêtre (ou toute autre action liée à l'emploi de la barre de défilement), c'est à l'application de la gérer, en fonction de la valeur prise par le contrôle et de la valeur qu'il avait au début de l'action.

La fonction **TrackControl** admet quatre arguments. Les deux premiers représentent l'abscisse et l'ordonnée en coordonnées globales du point où débute l'action, tel qu'il a été enregistré dans l'événement de type *MouseDown*. Le troisième argument donne l'adresse de la procédure qui gère l'action de **TrackControl**. Le quatrième argument est le handle qui désigne le contrôle en cours de manipulation. La fonction retourne zéro si la souris est relâchée en dehors de la partie du contrôle où le bouton a été enfoncé, ce qui signifie que l'utilisateur a changé d'avis en cours de route et qu'il ne faut entreprendre aucune action, ou une valeur égale à celle qu'avait retournée **FindControl** juste auparavant, ce qui signifie que l'utilisateur est allé au bout de son idée et qu'il y a une action à entreprendre.

Pour gérer un bouton, **TrackControl** n'a pas besoin de procédure d'action, et on peut passer zéro-long dans ce cas. Par contre, il est indispensable de créer une procédure d'action pour la gestion des barres de défilement. En effet, tant que l'utilisateur n'a pas relâché le bouton de la souris, **TrackControl** appelle périodiquement cette procédure, ce qui permet une action continue, généralement un défilement.

Il y a deux moyens d'appeler une procédure d'action. Le premier, c'est de passer son adresse à la fonction **TrackControl**, en direct, ainsi que nous l'avons fait dans l'exemple. Le second, c'est de stocker cette adresse au niveau du contrôle lui-même, dans l'un des champs de sa structure. On utilisera pour cela la procédure **SetCtlAction** (voir description plus loin). Pour invoquer la procédure d'action, il suffira de passer un argument négatif à **TrackControl** :

```
part2 = TrackControl(tache.where, -1L, ctl);        /* invoque une procédure
                                                       au niveau du contrôle */
```

TrackControl ira rechercher dans la structure du contrôle s'il y a une procédure d'action, auquel cas il l'exécutera périodiquement. S'il n'y a pas de procédure, pas d'action (tout comme si l'argument avait été zéro-long).

Cette seconde solution est sans doute plus élégante, parce que plus lisible : on a une procédure dédiée pour chaque contrôle de type cadran, et qui se borne aux seuls tests nécessaires. Nous verrons les deux méthodes dans les exemples en fin de chapitre.

Une procédure d'action est une routine de type Pascal, qui admet deux arguments : le code désignant la partie du contrôle invoquée et le handle désignant le contrôle invoqué. C'est **TrackControl** qui lui passe ces arguments. La procédure doit examiner la partie du contrôle invoquée et agir en conséquence : d'une part en ajustant la valeur du contrôle en fonction du défilement (ce qui permet notamment au Control Manager de le redessiner correctement), d'autre part en exécutant l'action réelle du défilement, liée à cette nouvelle valeur. Un modèle de procédure d'action est donné plus loin dans ce chapitre, insistons sur le fait qu'il doit s'agir d'une procédure Pascal, sa déclaration sera donc de la forme suivante :

```
pascal void Defilement(part,control)
int     part;
Handle control;
```

- Après un événement de type *KeyDown*.

Un autre type d'événement doit être pris en compte dans la gestion des contrôles : la touche de clavier enfoncée. Nous avons dit qu'il existait une notion de bouton par défaut, pour lequel il est équivalent de cliquer dedans ou d'enfoncer la touche Retour ou Entrée. Quand un événement de type *KeyDown* est retourné, il faut bien vérifier s'il s'agit d'un Retour, auquel cas une action est à entreprendre, à condition que le bouton par défaut existe et soit visible et actif à ce moment-là.

Rien n'empêche également d'imaginer des touches de commandes liées à l'emploi de tel ou tel bouton, à l'instar des articles des menus déroulants. (Par exemple un bouton pourrait être invoqué par la combinaison des touches Pomme et l'initiale de son titre). C'est évidemment à l'application de gérer de telles commandes, le programmeur gardant toujours à l'esprit qu'il ne doit pas désorienter l'utilisateur (pas de confusions avec les menus déroulants, par exemple) et qu'il ne doit pas compliquer la tâche d'un éventuel traducteur de son application !

```
Handle  boutonDef;          /* handle sur le bouton par défaut */
long    unPoint;            /* un point appartenant à ce bouton */

case KeyDown :
    if (tache.modifiers & AppleKey)
        ...             /* répondre à la commande, généralement en appelant MenuKey */
    else
        {
        if (condition)  /* si on sait qu'on a affaire à une fenêtre contenant des contrôles */
            {
            if (tache.message & 0xFF == 13)    /* touche Retour ou Entrée ? */
                {
                if (TestControl(unPoint,boutonDef))
                    ...             /* action du bouton par défaut */
                }
            }
        ...     /* l'application fait ce qu'elle a à faire avec le caractère sélectionné */
    }
```


Compliqué, n'est-ce pas ? On commence par tester si la touche Pomme est enfoncée (on agit dans ce cas en conséquence). On teste ensuite dans un environnement multifenêtres si une condition est remplie : la fenêtre doit posséder des contrôles gérés par l'application pour qu'il y ait un intérêt à tester la touche Retour. On teste ensuite le caractère correspondant à la touche enfoncée. Et si c'est la touche Retour (code ASCII 13 décimal), on teste enfin l'état du contrôle.

La fonction **TestControl** admet trois arguments : l'abscisse et l'ordonnée d'un point dont on vérifie qu'il appartient à un contrôle dont le handle est passé en troisième argument. Elle retourne la partie du contrôle invoquée, ou zéro si le point n'appartient pas au contrôle. Cette fonction est généralement à usage interne (elle est appelée par **FindControl** et **TrackControl**). Nous nous en servirons pour tester l'état du contrôle, en passant en argument un point dont nous savons pertinemment qu'il appartient au bouton défini par défaut. Si la fonction retourne zéro, c'est que le bouton est soit désactivé, soit invisible, et il serait désastreux de déclencher l'action correspondante alors que l'utilisateur n'a pas le droit de cliquer dedans !

Compliqué, certes, mais heureusement que le Dialog Manager gérera pour nous le bouton par défaut dans les fenêtres d'alerte et de dialogue, en transformant l'événement de type *KeyDown* en événement de type *MouseDown* dans le bouton par défaut si les conditions préalables sont remplies.

Une façon beaucoup plus simple d'agir était d'aller tester directement deux champs du *Control Record*, à savoir *CtlFlag* (pour savoir si le contrôle est visible ou pas) et *CtlHilite* (pour savoir s'il est actif ou pas). Nous n'avons pas voulu donner la définition de cette structure, on constate qu'il y a des cas où on ne peut s'en passer !

On verra plusieurs exemples de manipulation des contrôles : deux localisations, une utilisation des barres de défilement en tant que cadrans avec utilisation de tous les types de boutons. Dans le chapitre consacré au Dialog Manager, comment sont gérés les boutons dans une fenêtre de dialogue. Enfin, dans le chapitre consacré à la routine **TaskMaster**, nous verrons comment laisser le système gérer à notre place le défilement du contenu des fenêtres possédant des barres de défilement dans leur région contour. Il est tout de même plus agréable de laisser le système accomplir tout seul certaines tâches pénibles !

Modification et dessin de contrôles

Dans les exemples qui suivent, la variable *ctl* désigne un handle sur contrôle, tel qu'il a pu être retourné par la fonction **NewControl**.

- On peut modifier le titre d'un bouton (quel que soit son type), grâce à la procédure **SetCtlTitle** (qui en outre redessine le contrôle). Deux arguments : un pointeur sur la chaîne de type Pascal désignant le nouveau titre et le handle repérant le contrôle. On peut par ailleurs connaître le titre d'un contrôle grâce à la fonction **GetCtlTitle**, à qui on donne en argument le handle repérant le contrôle et qui retourne un pointeur sur la chaîne Pascal contenant le titre.

```
char    titre[ ] = "\15Nouveau titre";
Pointer oldTitle;

oldTitle = GetCtlTitle(ctl);        /* on récupère un pointeur sur le titre actuel */
SetCtlTitle(titre,ctl);             /* on change le titre du contrôle */
```

• Un contrôle peut être rendu invisible par la procédure **HideControl** et visible par la procédure **ShowControl**. Un seul argument dans les deux cas : le handle désignant le contrôle. Faire apparaître ou disparaître des contrôles dans une fenêtre doit être motivé par de fortes raisons : il ne s'agit pas de désorienter l'utilisateur. L'offre de nouvelles options à la suite d'un choix de cet utilisateur peut être un exemple : on rend visibles à ce moment-là de nouveaux contrôles, qui avaient été créés invisibles.

```
HideControl(ctl);                /* le contrôle est rendu invisible */
...
ShowControl(ctl);                /* le contrôle est rendu visible */
```

• On préférera généralement créer tous les contrôles visibles, quitte à rendre certains d'entre eux inactifs en fonction de la configuration actuelle. Pour activer ou désactiver un contrôle, pour mettre en lumière une partie de contrôle, une seule procédure : **HiliteControl**. Deux arguments : le second est le handle représentant le contrôle, le premier est un entier sur 16 bits dont la valeur détermine l'action à entreprendre :

– la valeur 0 signifie que le contrôle est actif mais qu'il n'est pas mis en lumière. S'il était précédemment inactif, il est réactivé et redessiné ;

– une valeur entre 1 et 253 est comprise comme une partie d'un contrôle (actif) qui doit être mis en lumière (par exemple la flèche d'une barre de défilement). Ces valeurs ont été vues plus haut ;

– la valeur 254 est réservée ;

– la valeur 255 signifie que le contrôle est rendu inactif, et il est redessiné en conséquence.

On retiendra les deux appels suivants :

```
HiliteControl(255,ctl);          /* le contrôle est désactivé, il apparaît estompé */
HiliteControl(0,ctl);            /* le contrôle est réactivé, il apparaît normal */
```

• Pour dessiner l'ensemble des contrôles associés à une fenêtre, on utilisera la procédure **DrawControls**. Un seul argument : le pointeur désignant la fenêtre. Lieu typique où cette routine doit être employée : dans la réponse à un événement de mise à jour *(update event)*, entre **BeginUpdate** et **EndUpdate**. En effet, les routines telles que **SelectWindow** ou **ShowWindow** n'appellent pas automatiquement **DrawControls**, mais génèrent plutôt de tels événements.

Il n'existe aucune routine permettant de dessiner un seul contrôle. Pour ce faire, on peut utiliser une astuce : déclarer invalide le rectangle contenant le contrôle, par **InvalRect** (voir le Window Manager), ce qui génère un événement de mise à jour pour la fenêtre et provoque l'appel à **DrawControls**, le dessin étant limité à ce rectangle.

• Deux routines permettent l'accès à la valeur libre (un entier long) associée à un contrôle. La procédure **SetCtlRefCon** permet de fixer une telle valeur, la fonction **GetCtlRefCon** retourne la valeur de ce champ pour le contrôle passé en argument.

```
long    valeur;

valeur = GetCtlRefCon(ctl); /* on récupère la valeur contenue dans le champ */
SetCtlRefCon(valeur+1,ctl);      /* on fixe une nouvelle valeur */
```

• Deux routines permettent l'accès à la procédure d'action liée à un contrôle. La procédure **SetCtlAction** permet de donner l'adresse d'une nouvelle procédure d'action, la fonction **GetCtlAction** permet de récupérer l'adresse de la procédure actuellement utilisée par le contrôle (ou zéro-long).


```
void MonAction( );               /* déclaration d'une fonction */
void (*action)( ); /* action est l'adresse d'une fonction qui ne retourne pas d'argument */

action = GetCtlAction(ctl);
SetCtlAction(MonAction, ctl);
```

Un petit mot de C au passage. Pour exécuter explicitement la fonction dont le pointeur est retourné par **GetCtlAction**, on doit déclarer action comme ci-dessus, et on utilisera l'instruction (*action) (arg1, arg2) pour lancer cette fonction. Mais en règle générale, on ne fait que stocker l'adresse, pour un éventuel rétablissement ultérieur (quand on change plusieurs fois de procédure d'action). Alors, on peut simplement déclarer action de type Pointer.

• Notons enfin l'existence de deux procédures qui permettent de changer la position d'un contrôle dans la fenêtre, **MoveControl** et **DragControl**. L'application utilisera ces appels pour des contrôles particuliers qu'elle pourrait gérer elle-même, par exemple des simples boutons vus non pas comme des boutons-poussoirs (cas classique d'utilisation), mais comme des objets dont la localisation peut changer en fonction des événements (pourquoi les pions d'un jeu de dames ou les lettres d'un jeu de scrabble ne seraient-ils pas gérés comme des contrôles ?).

MoveControl ne fait que redessiner un contrôle ailleurs dans la fenêtre après l'avoir effacé de sa position initiale. On donne en argument les nouvelles abscisse et ordonnée du coin supérieur gauche du rectangle englobant le contrôle, ainsi que le handle désignant le contrôle.

DragControl agit en liaison avec l'utilisateur. C'est lui qui déplace le contrôle à l'intérieur de la fenêtre. Durant le déplacement, la silhouette du contrôle suit les mouvements de la souris, de la même manière qu'un déplacement de fenêtre (pratique plus familière à l'ensemble des utilisateurs). **DragControl** termine en appelant **MoveControl** quand le bouton de la souris est relâché.

Six arguments pour **DragControl** : l'abscisse et l'ordonnée du point de départ de l'action, traduites en coordonnées locales ; un pointeur sur un rectangle limitant le champ de déplacement ; un pointeur sur un rectangle de grâce (voir **DragWindow** dans le chapitre consacré au Window Manager pour avoir plus de renseignements à ce sujet) ; un entier désignant une contrainte dans le déplacement (0 : pas de contrainte, 1 : le déplacement est strictement horizontal, 2 : le déplacement est strictement vertical) ; le handle sur le contrôle incriminé.

DragControl comme **DragWindow** font appel à une fonction plus générale du Control Manager, qui s'appelle **DragRect**. Nous n'entrerons pas dans le détail de la description de cette fonction.

Valeurs prises par un contrôle

• Pour fixer la valeur associée à un contrôle (FALSE ou TRUE pour une case à cocher ou un bouton radio, une valeur arbitraire pour une barre de défilement), on emploiera la procédure **SeCtlValue**. Deux arguments : la valeur à donner (entier sur 16 bits) et le handle désignant le contrôle. Notons que la procédure redessine le contrôle en fonction de sa nouvelle valeur (case cochée ou bouton radio plein si 1, case non cochée ou bouton radio vide si 0, curseur de défilement au bon endroit en cas de barre de défilement). Pour connaître la valeur associée à un contrôle, on emploiera la fonction **GetCtlValue**. Elle retourne la valeur (dans un entier sur 16 bits) associée au contrôle dont le handle est passé en argument.

– cas d'un bouton simple. On ne doit pas toucher à sa valeur, qui n'a aucune signification ;

– cas d'une case à cocher. On donne la valeur 0 ou 1 à ce contrôle (ou plus généralement une valeur nulle ou non nulle), et quand l'utilisateur clique dans une telle case (et que la souris est relâchée dans cette case), on agit de la manière suivante :

```
int valeur;

valeur = GetCtlValue(ctl);  /* si le contrôle est une case à cocher... */
SetCtlValue(1-valeur, ctl); /* ...on change son état: sélectionné <--> non sélectionné */
        /* ou bien: SetCtlValue(!valeur, ctl); dans un cas plus général */
...     /* l'application fait ce qu'elle a à faire à la suite de ce changement */
```

Le fait de fixer la valeur redessine la case correctement.

– cas d'un bouton radio. Dès qu'on fixe la valeur d'un bouton radio, le Control Manager redessine tous les boutons appartenant à la même famille. Il suffira donc de donner la valeur 1 (ou toute autre valeur non nulle) au bouton radio sélectionné par l'utilisateur, pour que tout fonctionne correctement : si le bouton était déjà sélectionné, il le reste, s'il ne l'était pas, il le devient et tous les autres sont désélectionnés. On est sûr de la sorte qu'au moins un, et un seul, bouton est sélectionné dans l'opération.

```
SetCtlValue(1,ctl);         /* cas d'un bouton radio: ultra-simple! */
...     /* l'application fait ce qu'elle a à faire à la suite de ce changement */
```

– cas d'une barre de défilement. C'est dans la procédure d'action que la valeur sera fixée, en fonction de la partie sélectionnée du contrôle. Cette procédure sera appelée répétitivement par le Control Manager tant que l'utilisateur n'aura pas relâché le bouton de la souris.

```
pascal void Defilement(part,control)
int     part;
Handle  control;

{
int     valeur, min, max, unitflech, unitpage;

if (part < 5) return; /* if (part == 0) return; ...si on est sûr que le contrôle est une barre */
...     /* calcul de unitflech et unitpage en fonction de ce que représente la barre */
...     /* calcul de min et max grâce à GetCtlParams, voir point suivant */
valeur = GetCtlValue(control);
switch(part)
    {
    case UpArrow:
        valeur -= unitflech;            /* on retire une unité */
        break;

    case DownArrow:
        valeur += unitflech;            /* on ajoute une unité */
        break;

    case PageUp:
        valeur -= unitpage;             /* on retire une unité-page */
        break;

    case PageDown:
        valeur += unitpage;             /* on ajoute une unité-page */
        break;
    }

if (valeur<0) valeur = 0;               /* borne inférieure */
if (valeur>max-min) valeur = max-min;   /* borne supérieure */
SetCtlValue(valeur,control);            /* on fixe la nouvelle valeur */
...     /* action résultant du défilement, tenant compte de la nouvelle valeur */
}
```


On constate que seules les flèches et les bandes de défilement sont prises effectivement en compte. Le Control Manager se débrouille tout seul pour suivre les manipulations du curseur de défilement en fonction du déplacement de la souris, et fixe la nouvelle valeur du contrôle dès que l'utilisateur a relâché le bouton. En fait, seule l'action résultant de la nouvelle valeur prise par le contrôle est commune à toutes les parties de la barre de défilement.

Le test en première ligne vise à exclure tous les boutons, dans le cas où la procédure d'action est appelée à partir de **TrackControl**, sans test préalable sur la nature du contrôle. Si par contre on est sûr qu'on a affaire à une barre de défilement, soit parce qu'on le teste avant, soit parce que la procédure a été mémorisée dans le champ adéquat du contrôle grâce à **SetCtlAction**, le test peut se borner à vérifier que la valeur prise par *part* est non nulle : on ne fait rien si l'utilisateur est en dehors du contrôle.

Nous verrons deux exemples complets d'utilisation des procédures d'action en fin de chapitre.

• Dans le cadre des barres de défilement, le curseur matérialise une valeur entre un minimum et un maximum. Il est parfois nécessaire de changer ce minimum ou ce maximum. Par exemple, si la barre est censée repérer un document de 200 lignes dans une fenêtre affichant 20 lignes, le minimum sera fixé à 20 et le maximum à 200. Si l'utilisateur se sert de son clavier et rajoute une ligne au document, le maximum doit devenir 201. S'il se sert de sa souris et qu'il diminue la hauteur de la fenêtre, ramenant l'affichage à 16 lignes, le minimum doit devenir 16. Notons que la valeur prise par le contrôle ayant un minimum à 16 et un maximum à 201 est comprise entre 0 et 201-16. C'est pourquoi ces termes de minimum et de maximum semblent assez impropres !

La procédure **SetCtlParams** se charge de modifier les bornes associées à une barre de défilement. Trois arguments : la nouvelle valeur maximale, la nouvelle valeur minimale (toutes deux sur 16 bits) et le handle désignant le contrôle. Souvent, une seule des deux bornes doit être modifiée. Si on passe la valeur – 1 dans l'un des arguments, la routine comprendra qu'il s'agit d'un code signifiant : ne pas modifier cette valeur. Les deux exemples cités ci-dessus se coderont donc ainsi :

```
SetCtlParams(201, -1, ctl);/* modification de la borne supérieure du contrôle */
SetCtlParams(-1, 16, ctl);  /* modification de la borne inférieure du contrôle */
```

De plus, cette procédure redessine le contrôle en tenant compte des nouvelles valeurs, notamment en repositionnant et en redimensionnant correctement le curseur de défilement.

Pour connaître la valeur des bornes, on utilisera la fonction **GetCtlParams**, qui retourne dans un entier long le minimum (mot le moins significatif) et le maximum (mot le plus significatif) associés au contrôle repéré par le handle passé en argument.

```
long    val;
int     max, min;

val = GetCtlParams(ctl);
min = (int) val;                /* mot bas */
max = val >> 16;                /* mot haut */
```

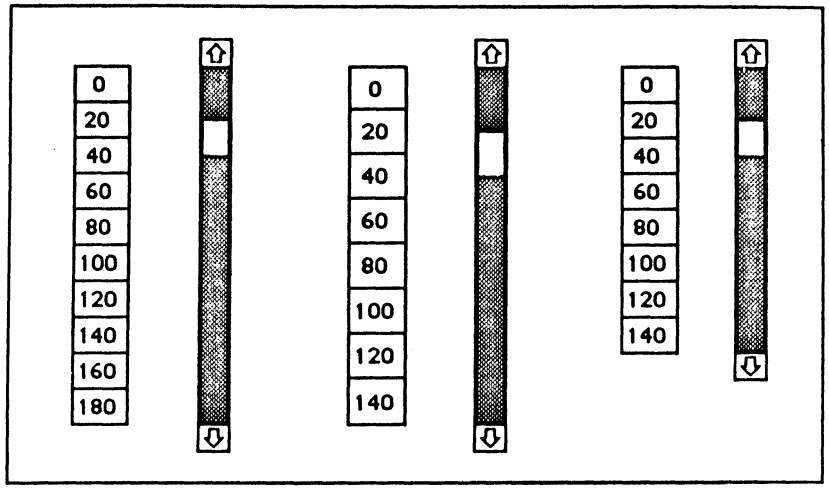


Figure VII.7 Les valeurs prises par une barre de défilement

• Pour mieux comprendre la signification des valeurs associées à une barre de défilement, regardons les figures VII.7 et VII.8.

Dans la figure VII.7, on suppose que la barre traduit un certain nombre de lignes dans un document dont seule une partie peut être visualisée. Au départ (à gauche), le document fait 200 lignes numérotées de 0 à 199 (valeur dite maximale), la partie visualisée correspond à 20 lignes (valeur dite minimale), et la ligne portant le numéro 30 est la première ligne visible. On constate que le curseur est à son maximum dès que la ligne 180 est atteinte, donc le contrôle ne peut prendre une valeur qu'entre 0 et 180 (où 180 = 200 − 20, c'est-à-dire total du document moins nombre visualisé).

Au centre, la barre est représentée après l'effacement de 40 lignes. La valeur ne peut plus être comprise qu'entre 0 et 140. A droite enfin, on a opéré une réduction d'échelle : on visualise toujours 20 lignes à la fois, mais dans un espace plus restreint. Aucune des valeurs n'a changé par rapport au dessin précédent, seule la taille du contrôle a été modifiée (en fait le rectangle qui l'englobe). Dans les trois cas, le curseur chevauche la frontière de deux « pages ».

Dans la figure VII.8, les barres traduisent un état, qui prend huit valeurs en haut, seize valeurs en bas. Ces valeurs sont les valeurs « maximales ». Dans les deux cas, la valeur « minimale » est égale à 1, puisque la barre traduit un état. La valeur prise par le contrôle est comprise entre 0 et max-min. Le curseur de défilement ne pourra jamais chevaucher deux états.

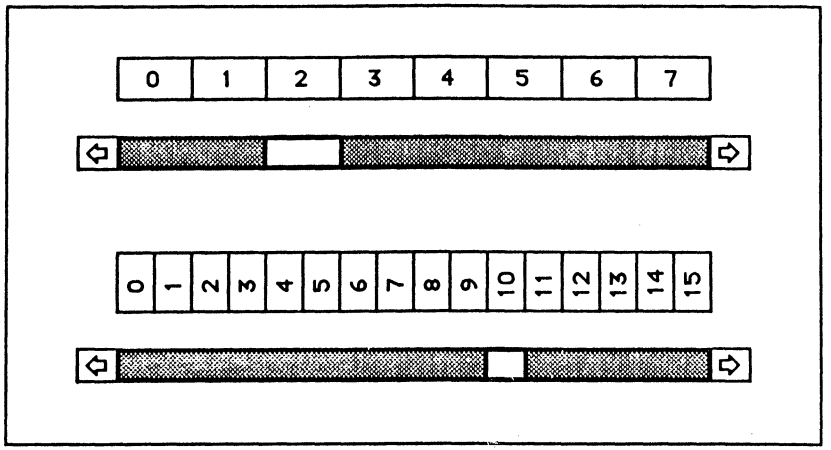


Figure VII.8. Les valeurs prises par une barre de défilement (bis).

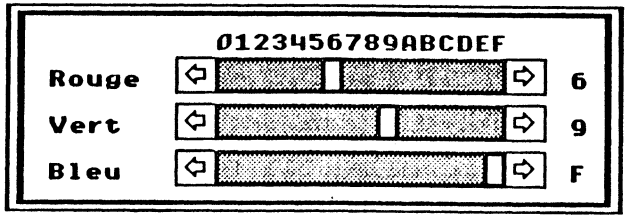


Figure VII.9 La fenêtre de l'exemple.

Exemple complet : contrôler les couleurs

Cet exemple consiste à ouvrir une fenêtre dans laquelle seront manipulés trois cadrans en forme de barre de défilement, permettant de régler le niveau de rouge, le niveau de vert et le niveau de bleu intervenant dans la composition de la couleur externe à la fenêtre... juste pour s'amuser à manipuler des barres de défilement à la main. On pourra ainsi visualiser les 4 096 couleurs possibles de l'Apple IIGS.

```
#include <tools.h>                    /* contient la definition des termes en gras */
#include <entete.h>                   /* contient la definition des termes en italique */

void Defilement();                    /* la procédure d'action */
ParamList maFen = {                   /* la fenêtre qui sera utilisée */
        sizeof(ParamList), 0x20A0, "", 0L,
        {0, 0, 0, 0}, 0L, 0, 0,
        0, 0, 0, 0,
        0, 0, 0, 0,
        0L, 0, 0L, 0L, 0L,
        {70,35,150,285}, -1L, 0L };
TaskRec tache;                        /* ce que manipule GetNextEvent */
Pointer wind;                         /* pointeur sur fenêtre */
int     indic = TRUE;                 /* indicateur de fin de boucle */
Handle  barR, barV, barB;             /* handles sur les 3 barres */
```

```
                    /***** PROGRAMME PRINCIPAL *****/

    main( )
{
Pointer myWind;                     /* pointeur sur fenêtre */
Rect    r;                          /* un rectangle */
int     myID;                       /* identifiant de l'application */

    myID = debut_appl(0);           /* initialisations en mode 320 */
    FlushEvents(EveryEvent,0);      /* la file d'événements est vidée */
    myWind = NewWindow(&maFen);     /* ouverture de la fenêtre... */
    SetPort(myWind);                /* ...qui devient le port courant */
    SetRect(&r,65,18,225,34);                   /* création de la barre des rouges */
    barR = NewControl(myWind,&r,"",0x1C,0,1,16,ScrollProc,0L,0L);
    SetRect(&r,65,38,225,54);                   /* création de la barre des verts */
    barV = NewControl(myWind,&r,"",0x1C,0,1,16,ScrollProc,0L,0L);
    SetRect(&r,65,58,225,74);                   /* création de la barre des bleus */
    barB = NewControl(myWind,&r,"",0x1C,0,1,16,ScrollProc,0L,0L);

    Desktop(5,0x40000022);          /* la couleur 2 sera utilisée pour redessiner le bureau */
    Couleur( );                     /* on lui donne la bonne couleur */

    do {
        if(!GetNextEvent(EveryEvent, &tache)) continue;     /* nouvel événement */
        switch(tache.what)                      /* quel événement? */
            {
            case MouseDown :                    /* bouton souris enfoncé */
                sourisDans(FindWindow(&wind, tache.where));
                break;

            case KeyDown :                      /* touche enfoncée */
                indic = FALSE ;                 /* on sort, c'est fini! */
                break;

            case UpdateEvt :                    /* mise à jour de la fenêtre */
                BeginUpdate(tache.message);
                MoveTo(10,31); DrawCString("Rouge");
                MoveTo(10,51); DrawCString("Vert");
                MoveTo(10,71); DrawCString("Bleu");
                MoveTo(82,15); DrawCString("0123456789ABCDEF");
                DrawControls(tache.message);
                EndUpdate(tache.message);
                break;
            }
        }
    while(indic);                           /* fin de la boucle d'événement */

    CloseWindow(myWind);                    /* on ferme la fenêtre... */
    quitter(myID);                          /* ...et on s'en va */
}

                /***** FONCTION SOURISDANS: réponse à un clic souris *****/

    sourisDans(code)

int     code;                   /* code retourné par FindWindow */

{
    switch (code)                           /* quel code? */
        {
        case inContent :                    /* dans le contenu de la fenêtre */
            RepCtl(wind);                   /* on va répondre aux contrôles */
            break;
        case inDrag :                       /* dans le cadre de la fenêtre */

            DragWindow(0, tache.where, 0, 0L, wind);   /* on suit la souris */
            break;
        }
}

                /***** FONCTION REPCTL: bouton souris enfoncé dans l'une des barres *****/

    RepCtl(fen)

Pointer fen;                            /* fenêtre où la souris a été enfoncée */

{
Handle  ctl;                            /* handle sur contrôle */
int     part;                           /* partie de contrôle */

    part = FindControl(&ctl, tache.where, fen);  /* a-t-on cliqué dans un contrôle? */
    if (part) TrackControl(tache.where, Defilement, ctl); /* oui, alors, gérons-le */
}

    /***** PROCEDURE DEFILEMENT: bouton souris enfoncé dans l'une des barres *****/

    pascal void Defilement(part,control)

int     part;                           /* partie du contrôle */
Handle  control;                        /* handle sur contrôle */

{
int valeur;                             /* valeur prise par le contrôle */
char msg[3];

    if(part < 5) return;                /* dans ces cas, on peut quitter! */
    valeur = GetCtlValue(control);      /* valeur actuelle du contrôle */
    switch(part)                        /* dans quelle partie l'utilisateur est-il? */
        {
        case UpArrow :                  /* flèche de gauche */
            valeur -= 1;                /* on retire une unité */
            break;

        case DownArrow :                /* flèche de droite */
            valeur += 1;                /* on ajoute une unité */
            break;

        case PageUp :                   /* bande de gauche */
            valeur -= 4;                /* on retire 4 unités */
            break;

        case PageDown :                 /* bande de droite */
            valeur += 4;                /* on ajoute 4 unités */
            break;
        }

    if (valeur<0) valeur = 0;
    if (valeur>15) valeur = 15;
    SetCtlValue(valeur,control);
    sprintf(msg,"%x ",valeur);
    if (control == barR) MoveTo(235,31);
    else if (control == barV) MoveTo(235,51);
    else if (control == barB) MoveTo(235,71);
    DrawCString(msg);                   /* on écrit la nouvelle valeur à droite de la flèche */
    Couleur( );                         /* et on change la couleur du fond d'écran */
}
```

```
/***** FONCTION COULEUR: change la couleur du fond d'écran *****/

Couleur( )
{
    int coul;
                                        /* calcul de la nouvelle couleur */
    coul = (GetCtlValue(barR)<<8) + (GetCtlValue(barV)<<4) + GetCtlValue(barB);
    SetColorEntry(0, 2, coul);  /* stockage dans la 2ème couleur de la palette système */
}
```

Exemple complet : des boutons et des barres

Cet exemple permet de visualiser certaines routines du Control Manager, en employant des boutons de tout type et deux barres de défilement. L'intérêt de l'application, hormis son aspect pédagogique, est évidemment nul.

La figure VII.1 donne une idée de l'écran au début de l'application : quatre boutons à droite, le premier pris par défaut, deux cases à cocher et deux boutons radio à gauche, le tout séparé par une grosse barre de défilement verticale, et une barre horizontale, moins épaisse.

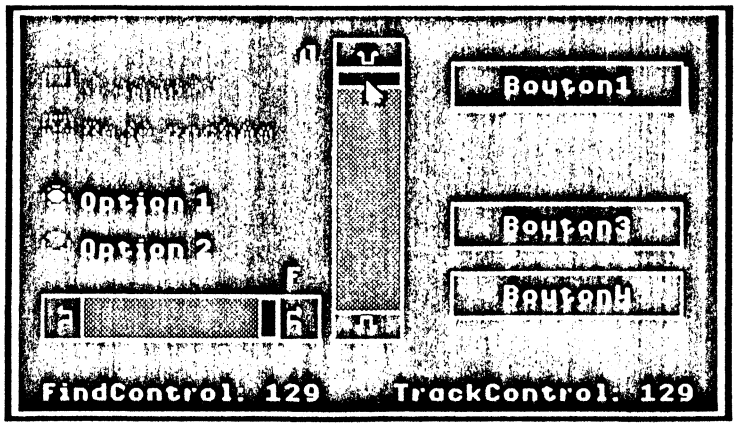


Figure VII-10

Quand on clique dans le bouton numéro 1 (ou si on utilise les touches Retour ou Entrée), on sort de l'application. Le bouton 2 désactive les cases à cocher, réactive les boutons radio et rend visible le bouton 3, s'il y a lieu, et devient invisible. Le bouton 3 désactive les boutons radio, réactive les cases à cocher et rend visible le bouton 2, s'il y a lieu, et devient invisible. Le bouton 4 réactive tout, rend visible tout.

Les cases à cocher et les boutons radio réagissent correctement, mais aucune action n'est liée à leur valeur. Les barres de défilement réagissent correctement, et agissent sur les entrées 0 et 15 de la palette de couleurs utilisée, ce qui permet avec certaines combinaisons de ne plus rien voir du tout ! La figure VII.10 montre une belle inversion du blanc et du noir.

Enfin, dans le bas de la fenêtre sont affichées les valeurs retournées par les fonctions **FindControl** et **TrackControl**.

```
#include <tools.h>                  /* contient la definition des termes en gras */
#include <entete.h>                 /* contient la definition des termes en italique */
```



```
void DefilH( );                     /* la procédure d'action de la barre horizontale */
void DefilV( );                     /* la procédure d'action de la barre verticale */
ParamList maFen = {                 /* la fenêtre qui sera utilisée */
        sizeof(ParamList), 0x20A0, "", 0L,
        {0, 0, 0, 0}, 0L, 0, 0,
        0, 0, 0, 0,
        0, 0, 0, 0,
        0L, 0, 0L, 0L, 0L,
        {20,10,190, 310}, -1L, 0L };
TaskRec tache;                      /* ce que manipule GetNextEvent */
Pointer wind;                       /* pointeur sur fenêtre */
int     indic = TRUE;               /* indicateur de fin de boucle */
Handle  bouton1, bouton2, bouton3, bouton4,
        check1, check2, radio1, radio2, barreH, barreV;     /* handles sur contrôle */


/***** PROGRAMME PRINCIPAL *****/

    main( )
{
Pointer myWind;                     /* pointeur sur fenêtre */
Rect    r;                          /* un rectangle */
int     myID;                       /* identifiant de l'application */
char    tampon[32];                 /* réserve 32 octets en mémoire */

    myID = debut_appl(0);           /* initialisations en mode 320 */
    InitColorTable(tampon);         /* on récupère la table de couleurs par défaut */
    SetColorTable(0,tampon);        /* la table 0 reçoit ces valeurs */
    SetColorTable(1,tampon);        /* la table 1 reçoit ces valeurs */
    SetAllSCBs(1);                  /* on rend active la table 1 */
    FlushEvents(EveryEvent, 0);     /* la file d'événements est vidée */
    myWind = NewWindow(&maFen);     /* la fenêtre est ouverte... */
    SetPort(myWind);                /* ...et on peut dessiner dedans */
    SetRect(&r,185,20,285,40);                          /* création du bouton 1 */
    bouton1 = NewControl(myWind,&r,"\7Bouton1",3,0,0,0,SimpleProc,0L,0L);
    SetRect(&r,185,50,285,70);                          /* création du bouton 2 */
    bouton2 = NewControl(myWind,&r,"\7Bouton2",2,0,0,0,SimpleProc,0L,0L);
    SetRect(&r,185,80,285,100);                         /* création du bouton 3 */
    bouton3 = NewControl(myWind,&r,"\7Bouton3",2,0,0,0,SimpleProc,0L,0L);
    SetRect(&r,185,110,285,130);                        /* création du bouton 4 */
    bouton4 = NewControl(myWind,&r,"\7Bouton4",2,0,0,0,SimpleProc,0L,0L);
    SetRect(&r,10,20,110,36);                           /* création d'une case à cocher */
    check1 = NewControl(myWind,&r,"\10A cocher",0,0,0,0,CheckProc,0L,0L);
    SetRect(&r,10,40,110,56);                           /* création d'une case à cocher */
    check2 = NewControl(myWind,&r,"\13Déjà cochée",0,1,0,0,CheckProc,0L,0L);
    SetRect(&r,10,70,110,86);                           /* création d'un bouton radio */
    radio1 = NewControl(myWind,&r,"\10Option 1",2,1,0,0,RadioProc,0L,0L);
    SetRect(&r,10,90,110,106);                          /* création d'un bouton radio */
    radio2 = NewControl(myWind,&r,"\10Option 2",2,0,0,0,RadioProc,0L,0L);
    SetRect(&r,10,120,130,140);                         /* création de la barre horizontale */
    barreH = NewControl(myWind,&r,"",0x1C,0,1,16,ScrollProc,0L,0L);
    SetCtlAction(DefilH,barreH);                        /* on fixe sa procédure d'action */
    SetRect(&r,135,10,165,140);                         /* création de la barre verticale */
    barreV = NewControl(myWind,&r,"",0x03,15,1,16,ScrollProc,0L,0L);
    SetCtlAction(DefilV,barreV);                        /* on fixe sa procédure d'action */

    do {
        if(!GetNextEvent(EveryEvent, &tache)) continue;     /* événement suivant */
        switch(tache.what)                                  /* de quoi s'agit-il? */
            {
            case MouseDown :                                /* bouton souris enfoncé */
                sourisDans(FindWindow(&wind, tache.where));
                break;
```

```
            case KeyDown :                          /* touche clavier enfoncée */
                if ((tache.message & 0xFF) == 13)   /* Retour ou Entrée? */
                indic = FALSE ;                     /* oui, on sort */
                break;

            case UpdateEvt :                        /* fenêtre à mettre à jour */
                BeginUpdate(tache.message);
                DrawControls(tache.message);        /* dessin des contrôles */
                MoveTo(10,165); DrawCString("FindControl:");
                MoveTo(160,165); DrawCString("TrackControl:");
                EndUpdate(tache.message);
                break;
            }
        }
    while(indic);                                   /* fin de la boucle d'événement */

    CloseWindow(myWind);                            /* on ferme la fenêtre.. */
    quitter(myID);                                  /* ...et on s'en va */
}

        /***** FONCTION SOURISDANS: réponse à un clic souris *****/

    sourisDans(code)

int     code;                                       /* code retourné par FindWindow */
{
    switch (code)
        {
        case inContent :                            /* bouton enfoncé dans le contenu de la fenêtre */
            if (wind != FrontWindow()) SelectWindow(wind); /* inutile, il n'y a qu'une fenêtre! */
            else RepCtl(wind);                      /* réponse aux contrôles */
            break;
        }
}

        /***** FONCTION REPCTL: souris enfoncée dans l'un des contrôles *****/

    RepCtl(fen)

Pointer fen;                                        /* fenêtre où la souris a été enfoncée */

{
Handle ctl;                                         /* handle sur contrôle */
int    part1,part2;                                 /* parties de contrôle */
char   msg[10];

    part1 = FindControl(&ctl,tache.where,fen);      /* dans quel contrôle? */
    sprintf(msg,"%d   ",part1);
    MoveTo(106,165); DrawCString(msg);              /* affichage de la valeur retournée */
    MoveTo(266,165); DrawCString("    ");
    if(part1)                                       /* vraiment dans un contrôle... */
        {
        part2 = TrackControl(tache.where,-1L,ctl);  /* ...alors on le gère */
        sprintf(msg,"%d   ",part2);
        MoveTo(266,165); DrawCString(msg);          /* affichage de la valeur retournée */
        if (part2 == part1)                         /* l'utilisateur a relâché la souris
                                                       dans la bonne partie du contrôle */
            switch(part1)                           /* quelle partie? */
                {
                case SimpleButton :                 /* dans un bouton classique */
                    if (ctl == bouton1) indic = FALSE ;     /* dans le bouton 1: on sort! */
                    else if (ctl == bouton2)                /* dans le bouton 2 */
                    {
```



```
                    HiliteControl(255,check1);  /* on désactive les cases à cocher */
                    HiliteControl(255,check2);
                    HiliteControl(0,radio1);    /* on active les boutons radio */
                    HiliteControl(0,radio2);
                    HideControl(bouton2);       /* on cache le bouton 2 */
                    ShowControl(bouton3);       /* on montre le bouton 3 */
                    }
                    else if (ctl == bouton3)                /* dans le bouton 3 */
                    {
                    HiliteControl(0,check1);    /* on active les cases à cocher */
                    HiliteControl(0,check2);
                    HiliteControl(255,radio1);  /* on désactive les boutons radio */
                    HiliteControl(255,radio2);
                    ShowControl(bouton2);       /* on montre le bouton 2 */
                    HideControl(bouton3);       /* on cache le bouton 3 */
                    }
                    else                                    /* forcément dans le bouton 4 */
                    {
                    HiliteControl(0,check1);    /* on active les cases à cocher */
                    HiliteControl(0,check2);
                    HiliteControl(0,radio1);    /* on active les boutons radio */
                    HiliteControl(0,radio2);
                    ShowControl(0,bouton2);     /* on montre le bouton 2 */
                    ShowControl(0,bouton3);     /* on montre le bouton 3 */
                    }
                    break;

                case CheckBox :                 /* dans une case à cocher */
                    SetCtlValue(1-GetCtlValue(ctl),ctl);    /* on change sa valeur */
                    break;

                case RadioButton :              /* dans un bouton radio */
                    SetCtlValue(1,ctl); /* on force sa valeur à 1 (tous les autres passent à 0) */
                    break;
                }
        }
}


/***** FONCTION DEFILCOMM: partie commune aux deux procédures d'action *****/
/* retourne FAUX si la procédure d'action n'a rien à faire, VRAI dans le cas contraire
et alors la valeur du contrôle est placée à l'adresse donnée par le troisième argument */

int DefilComm(part,control,pval)

int    part;                                    /* partie du contrôle */
Handle control;                                 /* handle sur contrôle */
int    *pval;                                   /* pointeur sur la valeur du contrôle */

{
int valeur;                                     /* la valeur du contrôle */

    if(!part) return FALSE ;                    /* part est nul, on n'a rien à faire */
    valeur = GetCtlValue(control);              /* valeur actuelle du contrôle */
    switch(part)
        {                                       /* la nouvelle valeur est fixée... */
        case UpArrow :
            valeur -= 1;
            break;

        case DownArrow :
            valeur += 1;
            break;
```

```
        case PageUp :
            valeur -= 4;
            break;

        case PageDown :
            valeur += 4;
            break;
        }
    if (valeur<0) valeur = 0;              /* ...dans des bornes correctes... */
    if (valeur>15) valeur = 15;
    SetCtlValue(valeur,control);           /* ...et donnée au contrôle... */
    (*pval) = valeur;                      /* ...et passée à la fonction appelante */
    return TRUE;
}

    /***** PROCEDURE DEFILH: bouton souris enfoncé dans la barre horizontale *****/

    pascal void DefilH(part,control)

int     part;                              /* partie du contrôle */
Handle  control;                           /* handle sur contrôle */

{
int val;                                   /* valeur du contrôle */
char msg[3];

    if (DefilComm(part,control,&val))      /* la procédure doit-elle continuer? */
        {
        sprintf(msg,"%x ",val);
        MoveTo(115,115); DrawCString(msg); /* la valeur est affichée près de la barre */
        SetColorEntry(1,0,GetColorEntry(0,val));    /* l'entrée 0 de la table 1 est modifiée */
        }
}

    /***** PROCEDURE DEFILV: bouton souris enfoncé dans la barre verticale *****/

    pascal void DefilV(part,control)
int     part;                              /* partie du contrôle */
Handle  control;                           /* handle sur contrôle */

{
int val;                                   /* valeur du contrôle */
char msg[3];

    if (DefilComm(part,control,&val))      /* la procédure doit-elle continuer? */
        {
        sprintf(msg,"%x ",val);
        MoveTo(120,20); DrawCString(msg); /* la valeur est affichée près de la barre */
        SetColorEntry(1,15,GetColorEntry(0,val)); /* l'entrée 15 de la table 1 est modifiée */
        }
}
```

CHAPITRE VIII

LINE EDIT

PRINCIPES GÉNÉRAUX

Line Edit est un éditeur de texte qui travaille sur des lignes de 256 caractères au maximum. Par éditeur, comprenons outil qui permet de créer, de modifier et de détruire les lignes de caractères, les sélections de texte étant faites par l'intermédiaire de la souris, et les saisies de caractères par l'intermédiaire du clavier. Une application utilisera l'éditeur pour obtenir de l'utilisateur des renseignements lui permettant de fonctionner : grâce à Line Edit, il est très facile de faire saisir du texte par petites quantités et de le mémoriser. Cependant, nous allons voir que certaines fonctions très utiles n'existent pas dans cet éditeur de base, et qu'il y a donc une importante valeur à ajouter pour créer un magnifique éditeur de programmes !

Parallèlement à l'édition de texte, Line Edit offre des fonctions d'affichage de textes possédant jusqu'à 32 767 caractères. Ces textes ne peuvent pas être édités : on ne pourra pas sélectionner des caractères, en insérer ou en effacer. Il sera par contre possible de justifier à gauche ou à droite, ou de centrer un tel texte dans un rectangle déterminé. Ce mode est idéal pour afficher des messages, par exemple pour gérer des écrans d'aide dans des fenêtres avec barres de défilement : l'utilisateur n'a pas à modifier de tels écrans !

Souvent, une application n'utilisera pas Line Edit directement, mais plutôt au travers des fenêtres d'alerte et de dialogue gérées par le Dialog Manager, objet du chapitre suivant.

UTILISATION DE LINE EDIT

Fonctions de Line Edit

◇ Edition de lignes

La fonction principale de Line Edit est d'assurer l'édition de lignes de caractères, en transformant les manipulations souris ou clavier de l'utilisateur en sélection de textes. Ceci se traduira par la présence soit d'un point d'insertion (généralement une barre verticale clignotante) soit d'une étendue de texte inversée (texte blanc sur fond

noir si les couleurs par défaut sont utilisées, c'est-à-dire l'écriture en noir sur fond blanc). A partir du point d'insertion, des caractères pourront être insérés (soit parce qu'ils sont saisis au clavier, soit parce qu'ils sont collés à partir du presse-papiers), ou détruits (en appuyant sur les touches convenables). A partir d'un texte sélectionné, les mêmes opérations sont possibles, mais la sélection est d'abord détruite. De plus, un texte sélectionné peut être coupé ou copié vers le presse-papiers.

L'affichage de la sélection (inversion des caractères sélectionnés) est entièrement géré par Line Edit. De même le couper-copier-coller, pour lequel plusieurs routines sont utilisables. Notons toutefois que Line Edit utilise un presse-papiers privé pour ses opérations d'édition, et non le presse-papiers commun à toutes les applications. Il y aura donc une manipulation (facile) à faire pour transférer le contenu du presse-papiers de Line Edit dans le presse-papiers général, afin de transmettre du texte entre deux applications, ou entre une application et un accessoire de bureau.

Une ligne est limitée à 256 caractères. C'est vraiment une ligne, et non un texte de 256 caractères, c'est-à-dire qu'une ligne ne peut en aucun cas s'étendre à l'écran sur plusieurs lignes. Si la ligne est plus longue que la largeur de la fenêtre, elle disparaît vers la droite, sans aucune tentative de mise en page. Elle est forcément cadrée à gauche, elle ne connaît pas les tabulations. A une ligne sont affectés une police de caractères, une taille et un style, comme nous le verrons plus loin. Toute modification de ces caractéristiques se fait sur la ligne entière. Enfin, le couper-copier-coller n'est pas intelligent, dans le sens où les espaces ne sont pas ajustés durant l'opération. Cette dernière restriction n'est pas grave, dans la mesure où Line Edit redessine toute la fin de la ligne après un couper ou un coller.

• Nous allons voir les commandes clavier supportées par Line Edit :

– les flèches gauche et droite permettent de déplacer le point d'insertion d'un caractère à la fois ;

– les combinaisons Option-flèche (gauche ou droite) permettent de déplacer le point d'insertion d'un mot (un mot est un ensemble de caractères délimités par des blancs, le blanc est le caractère de code ASCII $20) ;

– les combinaisons Pomme-flèche (gauche ou droite) permettent de déplacer le point d'insertion au début ou à la fin de la ligne courante ;

– les combinaisons Majuscule-flèche (gauche ou droite) permettent d'agrandir ou de diminuer une sélection de texte caractère par caractère (agrandissement à partir du point d'insertion initial, dans les deux sens, diminution à partir du point d'insertion courant, dans le sens contraire) ;

– les combinaisons Option-Majuscule-flèche (gauche ou droite) permettent d'agrandir ou de diminuer une sélection de texte mot par mot (règles similaires aux précédentes) ;

– les combinaisons Pomme-Majuscule-flèche (gauche ou droite) permettent d'étendre la sélection du point d'insertion jusqu'à un bout de la ligne ;

– la touche Effacement efface le texte sélectionné ou le caractère situé immédiatement à gauche du point d'insertion (le presse-papiers n'est pas affecté) ;

– la combinaison Contrôle-F efface le texte sélectionné ou le caractère situé immédiatement à droite du point d'insertion (le presse-papiers n'est pas affecté) ;

– la combinaison Contrôle-X efface le texte sélectionné ou toute la ligne si aucun caractère n'est inversé (le presse-papiers n'est pas affecté).
Remarque La touche Clear (en haut à gauche du pavé numérique) est équivalente à Contrôle-X ;

– la combinaison Contrôle-Y efface le texte sélectionné ou toute la fin de la ligne (tout ce qui est à droite du point d'insertion, le presse-papiers n'est pas affecté).

• Nous allons voir les manipulations souris et les combinaisons clavier/souris supportées par Line Edit :

– un clic dans une ligne positionne le point d'insertion ;

– faire glisser la souris dans une ligne étend ou diminue la sélection de texte (en suivant des règles similaires aux combinaisons Majuscule-flèche vues plus haut) ;

– un double-clic dans un mot sélectionne le mot ;

– un triple-clic dans une ligne sélectionne la ligne entière ;

– la combinaison Majuscule-clic permet d'étendre ou de diminuer une sélection (le point d'insertion initial est pris comme pivot).

◇ Affichage de textes

Line Edit permet l'affichage de textes comprenant jusqu'à 32 767 caractères et s'étendant sur plusieurs lignes, les sauts de ligne n'étant pas gérés automatiquement mais forcés par l'insertion de caractères Retour (code ASCII $D). Aucune opération d'édition n'est permise sur ce genre de textes. De plus, un seul style et une seule police de caractères peuvent être utilisés pour un texte donné. Toutefois, quelques fonctions de mise en page sont permises : le texte s'ajustera dans un rectangle et pourra être justifié à droite ou à gauche, ou centré horizontalement à l'intérieur de ce rectangle. Aucune tabulation n'est gérée.

Données manipulées par Line Edit

Pour permettre l'édition d'une ligne à l'écran, Line Edit a besoin d'un certain nombre de renseignements, tels que l'endroit où afficher le texte, l'endroit où le stocker, l'endroit où se trouve le point d'insertion, quelle est la zone sélectionnée, etc. A chaque ligne est associée une structure de données gérées par Line Edit, qui contient tous ces renseignements. Contrairement à l'habitude, nous allons définir le contenu exact de la structure, en gardant bien présent à l'esprit qu'il est interdit d'aller modifier directement l'un des champs sans passer par une routine appropriée. Certains champs n'étant pas accessibles au travers de routines, il sera parfois nécessaire d'aller les lire directement !

```
struct _LERec {
    Handle  TextHandle;      /* handle sur la ligne de texte éditable */
    int     Length;          /* longueur courante en caractères */
    int     MaxLength;       /* nombre maximal de caractères autorisés */
    Rect    DestRect;        /* rectangle de destination */
    Rect    ViewRect;        /* rectangle de visualisation */
    Pointer PortPtr;         /* pointeur sur le grafport définissant l'environnement */
    int     LineHite;        /* hauteur du rectangle à inverser si sélection */
    int     BaseHite;        /* position verticale du texte dans le rectangle de destination */
    int     SelStart;        /* début de la sélection */
    int     SelEnd;          /* fin de la sélection */
    int     ActFlg;          /* usage interne à Line Edit */
    int     CarAct;          /* usage interne à Line Edit */
    int     CarOn;           /* usage interne à Line Edit */
    long    CarTime;         /* usage interne à Line Edit */
    Pointer HiliteHook;      /* pointeur sur la routine de mise en lumière */
    Pointer CaretHook;       /* pointeur sur la routine de dessin du curseur d'insertion */
    };
#define LERec struct _LERec
```

Nous allons décrire certains de ces champs, ceux qui interviendront dans les manipulations de base du manager. Notons immédiatement qu'une telle structure est repérée par un handle, alloué par Line Edit à sa création.

• Tout affichage de texte suppose un environnement graphique complet, c'est-à-dire un grafport. Ce sont les caractéristiques du grafport qui serviront à l'affichage du texte (police de caractères, couleurs et style notamment), et donc la structure associée à une ligne gardera trace d'un pointeur *(PortPtr)* sur le grafport à utiliser.

• Dans ce grafport, deux rectangles seront définis (coordonnées locales) : un rectangle de destination, dans lequel le texte sera dessiné *(DestRect)*, et un rectangle de visualisation, dans lequel le texte sera visible *(ViewRect)*. Puisque nous travaillons dans un grafport, qui possède une clip region et une région visible, nous constatons

que Line Edit n'affichera du texte visible qu'à l'intersection de ces quatre rectangles ou régions. Souvent, ces deux rectangles seront égaux (ou *ViewRect* sera légèrement plus grand que *DestRect*). C'est le rectangle *ViewRect* qui fixe les frontières de l'activité de la souris sur une ligne. Dans l'exemple de fin de chapitre, nous avons fait ce rectangle volontairement trop haut, et nous voyons à l'exécution que ce n'est pas d'un effet très heureux pour un utilisateur !

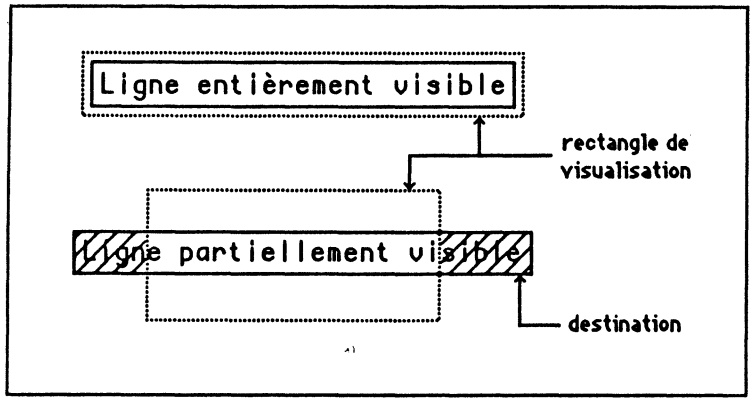


Figure VIII.1. Rectangles de destination et de visualisation.

C'est principalement pour pouvoir accéder au rectangle *ViewRect* que nous avons donné la définition de la structure *LERec*. Pour respecter l'interface utilisateur, il serait souhaitable que le pointeur change de forme dès qu'il passe au-dessus d'un texte éditable, et prenne la forme d'une poutre en I *(I-beam)*. Il faut donc tester l'appartenance du point survolé par le pointeur au rectangle *ViewRect* de chaque ligne éditable présente à l'écran !

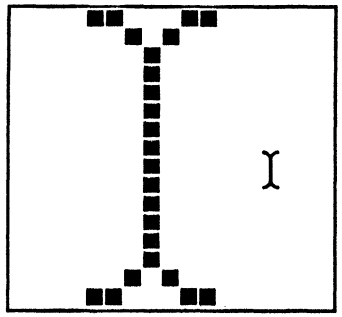


Figure VIII.2. Le curseur en forme de poutre.

Supposons que la variable LEHdl soit un handle repérant une ligne éditable, tel qu'il a été retourné par la fonction LENew (voir la section suivante). Pour obtenir le rectangle de visualisation défini pour cette ligne, on écrira en C :

```
LERec  **LEHdl;          /* LEHdl est déclaré handle sur structure LERec */
Rect   r;                /* r est un rectangle */

r = (*(LEHdl)->ViewRect;  /* rectangle de visualisation d'une ligne éditable */
```


ou encore :

```
Handle LEHdl;            /* LEHdl est déclaré handle sans particularité */
Rect   r;                /* r est un rectangle */

r = (*(LERec **) LEHdl)->ViewRect; /* rectangle de visualisation d'une ligne éditable */
```

et si pt est un point défini en coordonnées globales (et qu'on doit donc convertir en coordonnées locales du grafport définissant l'environnement de la ligne repérée par LEHdl), l'appartenance de ce point au rectangle de visualisation sera testée de la manière suivante :

```
SetPort((*LEHdl)->PortPtr);          /* utile si le grafport est susceptible d'avoir changé */
GlobalToLocal(&pt);                  /* conversion */
if (PtInRect(&pt, &(*LEHdl)->ViewRect)) ... /* test d'appartenance */
```

ou bien :

```
SetPort((*(LERec **) LEHdl)->PortPtr);
GlobalToLocal(&pt);
if (PtInRect(&pt, &(*(LERec **) LEHdl)->ViewRect)) ...
```

Et encore ! ce n'est pas aussi simple, puisque cette construction suppose que LEHdl est connu. Quand une fenêtre contient plusieurs lignes éditables, il appartient à l'application de déterminer sur laquelle l'utilisateur promène le pointeur et peut être amené à cliquer ! C'est ce genre de tests qui rendent très difficile l'écriture d'un éditeur tout simple sur l'Apple IIGS, infiniment plus difficile qu'avec le manager TextEdit sur Macintosh.

- Le texte lui-même ne fait pas partie de la structure, mais est repéré au moyen d'un handle *(TextHandle)*. C'est ce handle qui est manipulé par Line Edit. Un texte n'est ni une chaîne de type Pascal, ni une chaîne de type C, mais une suite quelconque de caractères. A ce texte sont associées deux valeurs, son nombre de caractères courant *(Length)* et le nombre maximal de caractères qu'il est autorisé à posséder *(MaxLength)*.

- Pour placer le texte dans le rectangle de destination, pour assurer l'inversion graphique lors d'une sélection de texte, Line Edit a besoin de deux renseignements : où faut-il placer le crayon par rapport au sommet de ce rectangle pour écrire le texte *(BaseHite)*, et quelle sera la hauteur de la zone à inverser *(LineHite)*. Ces deux élements dépendent de la police de caractères utilisée, et devront être recalculés si celle-ci est modifiée. Ces deux champs obéiront aux règles suivantes :

  BaseHite = leading + ascent

  LineHite = leading + ascent + descent

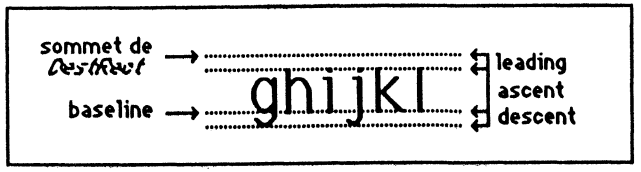


Figure VIII.3. Position du texte dans le rectangle de destination.

- Pour garder trace de la partie de texte sélectionnée ou de la position du point d'insertion, Line Edit utilise deux entiers, *SelStart* et *SelEnd*. Dans ces entiers sont

mémorisées des positions de caractères. Il faut voir une position de caractères comme un index au travers du texte, la position 0 désignant la gauche du premier caractère, la position 1 la gauche du deuxième caractère, etc. Un point d'insertion en position 4 signifie qu'il est situé entre le quatrième et le cinquième caractère de la ligne éditable.

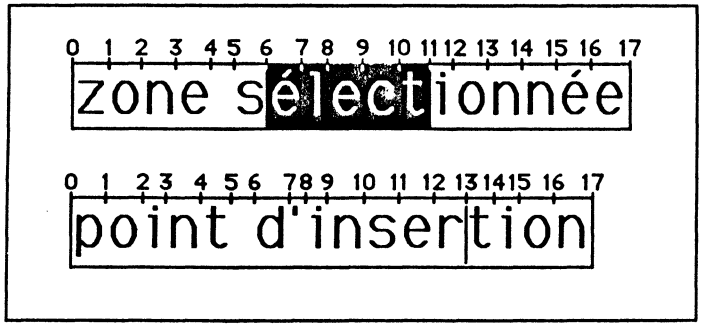


Figure VIII.4. Zone sélectionnée ou point d'insertion.

SelStart et *SelEnd* mémorisent les positions de début et de fin de la sélection. Si ces nombres sont différents, il y a effectivement une sélection de texte (la différence entre les deux donne le nombre de caractères sélectionnés) et Line Edit inverse le rectangle correspondant à cette sélection, en utilisant *LineHite* comme hauteur et en faisant la somme des largeurs des caractères pour obtenir la largeur totale du rectangle. Il s'agit réellement d'une inversion au sens QuickDraw du terme, et non d'un échange entre les couleurs de premier plan et de fond (cette subtilité n'est visible qu'en cas d'utilisation de couleurs autres que le noir et blanc).

Si ces nombres sont égaux, il n'y a pas de sélection de texte : la position désignée sera donc celle du point d'insertion, par défaut une barre clignotante de hauteur égale à *LineHite*.

Vision d'ensemble des routines

On commencera par initialiser Line Edit, comme tout autre outil, avant d'utiliser les routines que nous offre ce manager. Pour créer une ligne éditable et allouer un handle sur cette ligne, on appellera **LENew**. Une fois qu'on n'aura plus besoin de cette ligne, on fera de la place avec **LEDispose**. Une routine devra être appelée régulièrement pour assurer le clignotement du point d'insertion, **LEIdle**.

Pour répondre à un événement de type *MouseDown* dans le rectangle de visualisation d'une ligne éditable, on utilisera **LEClick**. Pour répondre à un événement de type *KeyDown* ou *AutoKey*, on appellera **LEKey**. Pour assurer le couper-copier-coller, il existe **LECut, LECopy** et **LEPaste**. Pour insérer du texte, on pourra utiliser **LEInsert**, et **LEDElete** pour effacer du texte. Ces deux routines ne touchent pas au presse-papiers et peuvent former les bases d'implantation d'une commande Annuler.

Pour transférer les informations copiées entre le presse-papiers privé de Line Edit et le presse-papiers public géré par le Scrap Manager, on utilisera **LEFromScrap** et **LEToScrap**.

En réponse à un événement de type *UpdateEvt*, il faudra appeler **LEUpdate**. En réponse à un événement de type *ActivateEvt*, il faudra utiliser **LEActivate** ou **LEDeactivate**.

Pour forcer le contenu d'une ligne éditable, on appellera **LESetText**. Pour forcer une sélection de texte, on utilisera **LESetSelect**.

Enfin, pour dessiner un texte (éventuellement plus long que 256 caractères) sans possibilité d'édition, il faudra se servir de **LETextBox**.

Nous allons maintenant détailler ces routines, en étudiant leur syntaxe et leur utilisation exacte.

EXEMPLES D'UTILISATION

Initialisation et création de lignes éditables

```
Rect    dest, view;                  /* les deux rectangles */
Handle  hL1;                         /* handle sur ligne éditable */

...                                  /* initialisations précédentes */
LEStartUp(myID, zeropg);             /* initialisation de Line Edit */
...                                  /* suite des initialisations */

SetPort(...);                        /* on se place sur le grafport voulu */
SetRect(&dest, 4, 4, 200, 15);       /* rectangle de destination (coordonnées locales) */
view = dest;                         /* recopie dans le rectangle de visualisation... */
InsetRect(&view, -2, -2);            /* ...qui est légèrement agrandi */
hL1 = LENew(&dest, &view, 30);       /* création d'une ligne éditable: 30 caractères au plus */
...
```

Tout appel à une routine de Line Edit doit être précédé de l'initialisation du manager. C'est **LEStartUp** qui s'en charge. Deux arguments : l'identifiant de l'application et l'adresse d'une page zéro, dont Line Edit a besoin pour fonctionner. (Voir le chapitre XII pour une vision d'ensemble des initialisations). **LEStartUp** s'alloue de manière interne un handle qui repérera son presse-papiers privé, vide à l'initialisation.

Pour créer une ligne éditable, on appelle la fonction **LENew**. On donne trois arguments : un pointeur désignant le rectangle de destination, un pointeur désignant le rectangle de visualisation et un entier contenant le nombre maximal de caractères autorisés dans la ligne (compris entre 1 et 256). La fonction retourne un handle (sur une structure *LERec*) qui permettra de désigner la ligne éditable dans toutes les manipulations futures. Les rectangles sont donnés en coordonnées locales du grafport d'appartenance, c'est-à-dire le grafport courant au moment de l'appel (ce grafport sera généralement le port associé à une fenêtre). Le rectangle de visualisation ne doit pas être vide. Pour rendre le texte complètement invisible, il suffit que l'intersection entre les rectangles de destination et de visualisation soit vide.

Attention, **LENew** ne provoque pas l'apparition du curseur clignotant (voir le paragraphe suivant sur le point d'insertion). **LENew** alloue un handle qui repérera le texte de la ligne, vide à la création (les champs *SelStart* et *SelEnd* sont donc forcément nuls).

Le handle repérant une ligne éditable sera valide jusqu'à ce que la procédure **LEDispose** soit appelée. On précise ce handle en argument. La place occupée en mémoire par la structure *LERec* et par le texte que la ligne contenait est libérée.

Pour affecter du texte à la ligne éditable et remplacer celui qui aurait pu s'y trouver déjà, on fait appel à la procédure **LESetText**, qui réclame trois arguments : un pointeur sur le texte à incorporer, le nombre de caractères de ce texte et le handle sur la ligne éditable où le texte doit être affecté. Le point d'insertion sera fixé après le dernier caractère du texte. Si le nombre de caractères dépasse le maximum autorisé pour la ligne, le texte sera tronqué.

Cette procédure n'affecte absolument pas ce qui est affiché à l'écran : il faut forcer le dessin du nouveau texte. La méthode correcte est la suivante : on déclare invalide le rectangle de visualisation par **InvalRect**, ce qui va provoquer un événement de mise à jour de la fenêtre, et on répond à cet événement par un appel à **LEUpdate**.

Point d'insertion, texte sélectionné

Nous avons dit que l'appel à **LENew** ne provoque pas l'apparition du curseur clignotant. Supposons que nous voulions créer plusieurs fenêtres possédant chacune plusieurs lignes éditables. Il serait assez inopportun de voir plusieurs points d'insertion clignoter en même temps, ou plusieurs zones sélectionnées en divers lieux de l'écran ! La règle est qu'il n'y a qu'un point d'insertion ou zone sélectionnée, et que ce point ou cette zone appartient obligatoirement à la fenêtre active. Il est de la responsabilité de l'application de garder trace de la ligne active ainsi définie.

- Pour faire clignoter le point d'insertion ou inverser la zone sélectionnée dans une ligne éditable, on fera appel à la procédure **LEActivate** (en donnant comme argument le handle sur la ligne visée). On aura pris soin juste avant de rendre son état normal à la précédente ligne sélectionnée, en appelant la procédure **LEDeactivate** (même type d'argument).

Un lieu privilégié pour placer ces procédures est évidemment la réponse aux événements d'activation. Sans se poser la moindre question, on appelle **LEDeactivate** quand une fenêtre est désactivée, ou **LEActivate** quand elle est activée. Cela se complique dès qu'une fenêtre contient plus d'une ligne éditable : ces procédures doivent être appelées dès qu'un clic souris intervient dans une ligne éditable différente de la ligne active !

- Pour répondre au clic souris, on appellera la procédure **LEClick**, à condition que la fenêtre soit active (sinon on active la fenêtre, comme d'habitude). Deux arguments : un pointeur sur l'événement à traiter et un handle sur la ligne où le clic a eu lieu.

Voici une séquence d'instructions typique pour répondre à un *MouseDown* qui a eu lieu dans la région contenu d'une fenêtre contenant des lignes éditables (l'exemple assume qu'il n'y en a que deux) :

```
TaskRec   tache;          /* l'événement auquel il faut répondre */
Handle    hLE;            /* handle sur la ligne courante */
Handle    hL1, hL2;       /* les deux lignes définies */
Pointer   wind;           /* pointeur sur la fenêtre active */

case wInContent:
    if (wind != FrontWindow( )) SelectWindow(wind);
    else
        {
        long pt;                    /* point qu'on va convertir en coordonnées locales */
        pt = tache.where;           /* il est en coordonnées globales */
        GlobalToLocal(&pt);         /* il est converti */
        LEDeactivate(hLE);          /* la ligne courante est désactivée */
                                    /* on va déterminer dans quelle ligne l'utilisateur a cliqué */
        if (PtInRect(&pt, &(*(LERec **) hL1)->ViewRect)) hLE = hL1;
        else if (PtInRect(&pt, &(*(LERec **) hL2)->ViewRect)) hLE = hL2;
        LEActivate(hLE);            /* on active la nouvelle ligne courante... */
        LEClick(&tache, hLE);       /* ...et on laisse Line Edit se débrouiller */
        }
    break;
```

Quand **LEClick** est appelé, l'écran est propre, et la ligne est propre. Toute manipulation de l'utilisateur sera traduite sur la ligne : déplacement du point d'insertion, extension de la sélection, etc. Si le travail préparatoire n'avait pas été

accompli, on pourrait se retrouver avec plusieurs zones sélectionnées. En effet, **LEClick** ne sait agir que sur une ligne, ignorant totalement l'état des autres.

- Pour que le curseur clignote dans le cas d'un point d'insertion, il est nécessaire d'appeler la procédure **LEIdle** périodiquement, disons au moins une fois dans la boucle d'événement, avant **GetNextEvent**. On donnera en argument le handle sur la ligne courante. On pourra se passer d'appeler **LEIdle** si la fenêtre active ne contient pas de ligne éditable.

- L'application peut forcer dans une ligne éditable la position du point d'insertion ou la zone sélectionnée, grâce à la procédure **LESetSelect**. Trois arguments : un entier donnant la nouvelle position de début de sélection, un entier donnant la nouvelle position de fin de sélection, et le handle désignant la ligne. De manière évidente, les deux premiers arguments doivent être compris entre 0 et 256, et le premier argument doit être inférieur ou égal au second. S'ils sont égaux, la sélection est vide, et la position désigne alors le point d'insertion. On constate que cette procédure assure la modification directe des champs *SelStart* et *SelEnd* de la structure *LERec*. L'effet à l'écran est immédiat.

Edition de lignes

Nous venons de voir comment positionner un point d'insertion ou faire une sélection. Dès que l'utilisateur va se servir de son clavier, il va détruire la sélection et insérer des caractères. Il peut aussi invoquer le menu Edition, choisir Couper pour effacer la sélection et la transférer dans le presse-papiers, choisir Copier pour transférer la sélection dans le presse-papiers sans l'effacer, choisir Coller pour remplacer la sélection par le contenu du presse-papiers ou l'insérer à l'endroit du curseur clignotant, ou choisir Effacer (sans affecter le presse-papiers). Notons que dans le cas d'insertion de texte, si le nombre de caractères courant atteint le nombre maximal autorisé, ce seront les caractères les plus à droite qui seront perdus.

- Dès que l'utilisateur a enfoncé une touche alors qu'il existe une ligne éditable courante, l'application appelle **LEKey** avec trois arguments : le code ASCII du caractère (dans un entier), le champ *modifiers* tel qu'il est retourné par l'Event Manager et le handle sur la ligne éditable courante.

LEKey remplace la sélection par le caractère entré ou insère le caractère s'il n'y avait pas de sélection, et positionne le point d'insertion juste derrière. Elle gère complètement les caractères spéciaux, tels la touche d'effacement, les flèches gauche et droite, les caractères Contrôle-F, Contrôle-X et Contrôle-Y, et la touche Clear (équivalente à Contrôle-X). **LEKey** redessine le texte immédiatement.

Tout serait parfait si cette procédure ne présentait pas le défaut de laisser à l'application un certain nombre de manipulations. Dans la mesure où elle ne filtre absolument rien, c'est à l'application de vérifier si par exemple la touche Pomme était enfoncée, et d'agir en conséquence : la touche Pomme était associée à une flèche (droite ou gauche), c'est **LEKey** qu'il faut appeler pour déplacer le point d'insertion en bout de ligne, sinon c'est sans doute un équivalent-clavier de menu déroulant, et il faut appeler **MenuKey**. De la même manière, les caractères optionnels ne sont pas gérés.

Quand la touche Option est enfoncée, il faut forcer à un le bit 7 du code ASCII pour obtenir les caractères internationaux. Enfin, on peut juger indésirables les caractères de contrôle, obtenus quand la touche Contrôle est enfoncée. C'est à l'application à en faire le tri : certains de ces caractères sont imprimables, et il serait dommage de s'en priver ; d'autres sont vraiment spéciaux, et pourraient s'avérer perturbants. A titre d'exemple, savez-vous comment on obtient le caractère « é » sur un clavier QWERTY ? En faisant Contrôle-Option-n, et à condition de forcer à un le bit 7... Nous verrons un exemple de filtrage en fin de chapitre. Il n'est pas parfait, mais satisfait aux règles de l'interface utilisateur. Les caractères de contrôle ne sont pas filtrés.

Rappelons qu'à la fin du Chapitre IV sur l'Event Manager, l'exemple permet de voir le code ASCII de n'importe quelle combinaison de touches, caractères optionnels compris. Ce peut être un élément de réflexion pour l'écriture d'un filtre parfait.

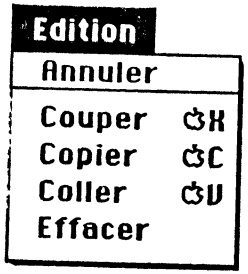


Figure VIII.5. Le menu standard d'édition.

• Pour répondre à la commande Couper, il suffit d'appeler la procédure **LECut** en précisant en argument la ligne éditable concernée. Le contenu de la sélection est placé dans le presse-papiers privé de Line Edit, écrasant ce qui s'y trouvait. La sélection est effacée et la ligne redessinée. Si la selection était vide (présence d'un point d'insertion), le presse-papiers est vidé.

• Pour répondre à la commande Copier, il suffit d'appeler la procédure **LECopy** en précisant en argument la ligne éditable concernée. Le contenu de la sélection est placé dans le presse-papiers privé de Line Edit, écrasant ce qui s'y trouvait. Si la sélection était vide (présence d'un point d'insertion), le presse-papiers est vidé.

• Pour répondre à la commande Coller, il suffit d'appeler la procédure **LEPaste** en précisant en argument la ligne éditable concernée. Le contenu du presse-papiers privé de Line Edit (même vide) vient remplacer la sélection en cours et le point d'insertion est positionné juste derrière le texte ainsi inséré. S'il n'y avait pas de sélection, il y a juste insertion de texte. Le contenu du presse-papiers reste inchangé. Si nécessaire, la ligne est redessinée.

• Pour répondre à la commande Effacer, il suffit d'appeler la procédure **LEDelete** en précisant en argument la ligne éditable concernée. La sélection, si elle existe, est effacée, mais n'est pas transférée dans le presse-papiers, et la ligne est redessinée. S'il n'y a pas de sélection, il ne se passe rien.

• En addition à ces procédures, notons l'existence de **LEInsert**, qui permet d'insérer du texte juste devant la sélection ou devant le point d'insertion. On lui passe trois arguments : un pointeur sur le texte à insérer, le nombre de caractères à insérer et la ligne éditable concernée. Ni la sélection ni le presse-papiers ne sont modifiés par cette procédure.

Pour mettre en œuvre une commande Annuler, il faut commencer par se fixer certaines règles : que doit-on annuler ? On pourra mémoriser l'état d'une ligne au moment de son activation, et considérer l'annulation comme la restauration de cette ligne tant qu'une autre ligne n'est pas activée. Ou alors mémoriser son état avant une commande, pour pouvoir annuler les effets de cette commande. Ou encore mémoriser une séquence de caractères saisis, pour les annuler d'un coup (dans ce cas il serait bien de rétablir la sélection éventuellement écrasée par la saisie du premier caractère de la séquence). Et on n'oubliera pas de laisser la possibilité à l'utilisateur d'annuler son annulation !

Affichage de lignes éditables et de texte non éditable

• Cas des lignes éditables

Nous avons vu que la procédure **LESetText** ne redessine pas le contenu de la ligne éditable, et qu'il faut rendre invalide son rectangle de visualisation pour provoquer un événement d'activation. Dans la réponse à cet événement, on appelle **LEUpdate**, avec comme argument le handle sur la ligne à redessiner.

Quand une ligne éditable est dessinée à l'écran, elle utilise les caractéristiques du grafport dont elle garde trace dans la structure *LERec* qui l'identifie. En particulier, la police de caractères, la taille, le style et les couleurs (fond et premier plan) utilisés sont ceux de ce grafport. On peut imaginer conserver autant de grafports différents qu'il y a de lignes éditables, chacune présentant ses propres caractéristiques. Ce serait un peu lourd ! Il vaut certainement mieux mémoriser ces caractéristiques au niveau de chaque ligne, et utiliser un grafport unique dont on modifiera les champs juste avant d'appeler **LEUpdate**.

Attention, si on modifie les caractéristiques d'un grafport, aucun effet visible ne se répercute à l'écran, puisque Line Edit n'a aucun moyen de savoir ce qui s'est passé. Il faut donc là encore rendre invalide le rectangle de visualisation des lignes concernées par la modification, et appeler **LEUpdate**.

LEUpdate sera appelé classiquement entre les procédures **BeginUpdate** et **EndUpdate**. Juste avant l'appel, s'il concerne la ligne active, on prendra soin d'effacer complètement son rectangle de visualisation, pour éviter de laisser accidentellement à l'écran le fantôme du point d'insertion quand la fenêtre est désactivée.

Reprenons notre exemple précédent où nous avions deux lignes éditables dans une fenêtre. La réponse aux événements de mise à jour pourrait avoir l'allure suivante :

```
long    port;
Pointer precPort;
Handle  hLE, hL1, hL2;              /* hLE désigne la ligne active */
int     style1, style2;             /* styles dans lesquels les lignes sont dessinées */

case UpdateEvt :
    port = tache.message ;
    precPort = GetPort( );          /* on mémorise le grafport courant */
    SetPort(port);                  /* on fixe le grafport de la fenêtre à mettre à jour */
    BeginUpdate(port);
    EraseRect(&(*(LERec **)hLE)->ViewRect); /* on efface le rectangle de la ligne active */
    SetTextFace(style1);            /* le style a été déterminé par ailleurs */
    LEUpdate(hL1);                  /* dessin de la première ligne */
    SetTextFace(style2);            /* le style a été déterminé par ailleurs */
    LEUpdate(hL2);                  /* dessin de la deuxième ligne */
    EndUpdate(port);
    SetPort(precPort);              /* on rétablit le grafport précédent */
    break;
```

• Cas du texte non éditable

Nous n'en avions pas parlé jusqu'à présent, et nous n'en parlerons plus par la suite. En effet, le texte non éditable est géré par une seule procédure, **LETextBox**. Quatre arguments : un pointeur sur le texte à afficher, un entier donnant le nombre de caractères du texte, un pointeur sur le rectangle (coordonnées locales) dans lequel le texte viendra s'afficher et un entier précisant la justification du texte à l'intérieur du rectangle (0 signifie justification à gauche, − 1 justification à droite et 1 texte centré).

LETextBox commence par effacer le rectangle spécifié, puis dessine le texte à l'intérieur. Le rectangle agit comme une clip region, dans le sens où le texte ne pourra pas déborder du rectangle. Les lignes de texte sont définies par l'insertion de caractères Retour (code ASCII 13 décimal) au milieu des caractères éditables. Excepté la justification, aucune tentative de mise en page n'est effectuée : si une ligne dépasse la largeur du rectangle, une de ses extrémités (voire les deux) ne sera pas visible, un point c'est tout. Aucun passage à la ligne suivante ne se fait automatiquement.

Le texte est limité à 32 737 caractères. Il est dessiné avec les caractéristiques du grafport courant. Pour ce faire, Line Edit crée une structure provisoire et l'écrase immédiatement après. C'est pourquoi un tel texte n'est pas éditable : Line Edit n'en

garde pas une copie. C'est pourquoi aussi on aura tout intérêt à appeler cette procédure en réponse à un événement de mise à jour (voir l'exemple en fin de chapitre).

Manipulation de presse-papiers

• Le Scrap Manager (voir chapitre X) gère un presse-papiers grâce auquel les applications et/ou les accessoires de bureau peuvent s'échanger des données, données de tous types. Line Edit gère un presse-papiers qui lui est propre, et qui ne peut contenir autre chose que du texte. Il arrive souvent que deux applications, ou une application et un accessoire de bureau, veuillent s'échanger du texte copié par l'intermédiaire de Line Edit. Cela n'est possible que si le contenu des deux presse-papiers peuvent être échangés.

Line Edit offre deux procédures pour ces échanges. **LEFromScrap** copie le contenu du presse-papiers public dans son propre presse-papiers, **LEToScrap** fait exactement l'inverse. Si une application veut gérer convenablement le copier-coller d'informations de type texte, elle veillera à appeler ces procédures à bon escient, notamment à son démarrage ou après réactivation (elle vérifiera si le presse-papiers public contient du texte, laissé là par l'application précédente ou par l'accessoire de bureau récemment appelé, et en fera éventuellement le transfert), et dans l'autre sens au moment d'une désactivation ou de quitter (l'accessoire appelé ou l'application suivante peuvent vouloir recevoir le texte copié, donc le transfert vers le presse-papiers public est indispensable).

• **LEScrapHandle** est une fonction sans argument qui retourne un handle sur le presse-papiers privé de Line Edit. De la sorte, on peut même s'amuser à éditer le presse-papiers !

• **LEGetScrapLen** est une fonction sans argument qui retourne dans un entier la taille en octets du presse-papiers privé de Line Edit. Cette taille doit être vue comme une limite supérieure au nombre de caractères mémorisés, et non le nombre exact. On peut changer la taille maximale du presse-papiers en passant un entier compris entre 0 et 256 à la procédure **LESetScrapLen**.

Remarque Le presse-papiers de Line Edit est limité à 256 octets, pas le presse-papiers public. Si une tentative de copie de plus de 256 caractères est effectuée par l'intermédiaire de **LEFromScrap**, une erreur sera retournée dans *_errno*, de code $1404 (presse-papiers trop gros pour être copié).

Exemple complet

Exemple complet, mais modeste. Modeste, parce qu'il ne manipule qu'une seule ligne éditable, et un seul texte non éditable sur plusieurs lignes. Notre but n'étant pas d'écrire un éditeur de programme, nous nous sommes limités à la simplicité. Le fait que Line Edit gère des lignes et soit incapable de passer automatiquement à la ligne suivante restreint singulièrement son utilisation ! Sur Macintosh, Text Edit sait éditer plusieurs lignes d'un coup et assurer une certaine mise en page (gestion de *texte*, et non de *ligne*). Aussi a-t-on vu fleurir de magnifiques exemples d'éditeurs, dans tous les livres, dans tous les environnements de développement, quel que soit le langage. La mise en œuvre était si simple ! Gageons que ce ne sera pas le cas avec Line Edit sur l'Apple IIGS, en tout cas les éditeurs « simples » auront des fonctionnalités extrêmement limitées. Par contre, Line Edit est parfait comme outil pour le Dialog Manager, dont nous allons parler dans le chapitre suivant, et nous n'avons pas voulu ici faire de manière compliquée ce que le Dialog Manager permet de faire de manière plus simple.

L'exemple passe en revue la plupart des appels de Line Edit sur une simple ligne éditable ! Toutes les manipulations sont permises sur cette ligne, sauf la commande Annuler et le passage de texte aux accessoires de bureau, non pas que nous trouvons

cela difficile à faire, mais parce qu'il n'existe au moment où nous écrivons ces lignes aucun accessoire de bureau susceptible de recevoir du texte, et qu'il est donc impossible de tester des exemples plus complets !

Le rectangle de visualisation est dessiné, et le pointeur est géré de telle sorte qu'il prend la forme d'une poutre en I dès qu'il passe au-dessus de cette zone éditable.

En addition à la ligne éditable, un texte non éditable est affiché, et grâce à un menu déroulant, on peut modifier sa justification.

Pour manipuler plus d'une ligne éditable par fenêtre, il suffira de combiner (très simplement) les morceaux d'exemples vus plus haut avec l'exemple complet vu ici. On pourra ainsi monter les éléments d'un éditeur pleine page, en définissant autant de handles qu'il y a de lignes. A vous de gérer les caractères Retour pour passer d'une ligne à l'autre, et, beaucoup plus compliqué, la possibilité de sélectionner du texte sur plus d'une ligne, ce qui est tout de même la moindre des choses sur un éditeur pleine page !

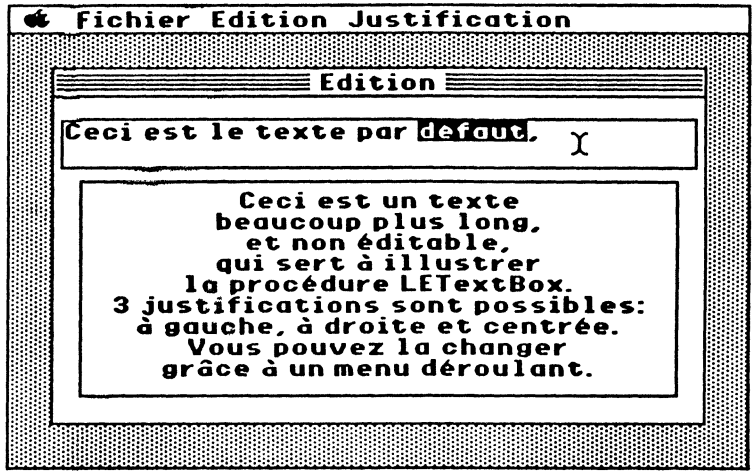


Figure VIII.6. L'écran de l'exemple.

```
#include <tools.h>                    /* définition des termes en gras */
#include <entete.h>                   /* définition des termes en italique */

#define mode 0                        /* 0 si mode 320, 1 si mode 640 */

char Menu1[ ] = "> @\\XN1";
char Menu11[ ] = "- A propos d'édition...\\N257VD";
char Menu19[ ] = ". ";
char Menu2[ ] = "> Fichier \\N2";
char Menu21[ ] = "- Quitter\\N260°Qq";
char Menu29[ ] = ". ";
char Menu3[ ] = "> Edition \\N3";
char Menu31[ ] = "- Annuler\\N250V°Zz";
char Menu32[ ] = "- Couper\\N251°Xx";
char Menu33[ ] = "- Copier\\N252°Cc";
char Menu34[ ] = "- Coller\\N253°Vv";
char Menu35[ ] = "- Effacer\\N254";
char Menu39[ ] = ". ";
char Menu4[ ] = "> Justification \\N4";
char Menu41[ ] = "- Gauche\\N271°GgC\\22";
char Menu42[ ] = "- Droite\\N270°Dd";
```

```
char Menu43[ ] = "- Centrée\\N272*Mm";
char Menu49[ ] = ". ";

#define iQuitter    260
#define iAnnuler    250
#define iCouper     251
#define iCopier     252
#define iColler     253
#define iEffacer    254
#define iGauche     271
#define iDroite     270
#define iCentre     272

int     colfen[ ] = {0,0x0F00,0x020F,0xF0F0,0x00F0};        /* couleurs des fenêtres */

    ParamList maFen = {
            sizeof(ParamList), 0x8080, "\11 Edition ", 0L,
            {0, 0, 0, 0}, colfen, 0, 0,
            0, 0, 0, 0,
            0, 0, 0, 0,
            0L, 0, 0L, 0L, 0L,
            {40,20,180, 300+320*mode}, -1L, 0L };

                                /* curseur en forme de poutre (modèle de définition libre) */
    char IBeam[ ] = { 10,0,3,0,                         /* 10 lignes de 3 mots */
                      0xFF,0,0x0F,0xF0,0,0,             /* image */
                      0,0xF0,0xF0,0,0,0,
                      0,0x0F,0,0,0,0,
                      0,0x0F,0,0,0,0,
                      0,0x0F,0,0,0,0,
                      0,0x0F,0,0,0,0,
                      0,0x0F,0,0,0,0,
                      0,0x0F,0,0,0,0,
                      0,0xF0,0xF0,0,0,0,
                      0xFF,0,0x0F,0xF0,0,0,
                      0,0,0,0,0,0,                      /* masque */
                      0,0,0,0,0,0,
                      0,0,0,0,0,0,
                      0,0,0,0,0,0,
                      0,0,0,0,0,0,
                      0,0,0,0,0,0,
                      0,0,0,0,0,0,
                      0,0,0,0,0,0,
                      0,0,0,0,0,0,
                      0,0,0,0,0,0,
                      5,0,3,0 };                        /* point chaud */

    char    texte[ ] = "Ceci est le texte par défaut, défaut inversé";
    char    text1[ ] = "Ceci est un texte\15beaucoup plus long,\15et non éditable,\15qui sert à
illustrer\15la procédure LETextBox.\0153 justifications sont possibles:\15à gauche, à droite et
centrée.\15Vous pouvez la changer\15grâce à un menu déroulant.";

    TaskRec tache;                                      /* ce que manipule GetNextEvent */
    Pointer fen;                                        /* pointeur sur fenêtre */
    int     indic = TRUE;                               /* indicateur de fin de boucle */
    Pointer wind;                                       /* la fenêtre courante */
    Handle  hLE;                                        /* la ligne éditable courante */
    Rect    rDV = {10,4*(mode+1),30,276+316*mode};      /* rectangle de destination et de visu */
    Rect    box = {40,15+40*mode,125,265+40*mode};      /* rectangle pour le texte à afficher */
    int     justif = 0;                                 /* justification active */
    Pointer arrow;                                      /* curseur en forme de flèche */

                    /***** PROGRAMME PRINCIPAL *****/
```



```
    main( )
{
int     myID;                                   /* identifiant de l'application */

    myID = debut_appl(mode);                    /* initialisations */
    PlaceMenus( );                              /* installe la barre des menus */
    fen = NewWindow(&maFen);                    /* ouverture de la fenêtre (invisible) */
    SetPort(fen);                               /* grafport sur lequel on va travailler */
    hLE = LENew(&rDV, &rDV, 30+50*mode);        /* création d'une ligne éditable... */
    LESetText(texte, 44, hLE);          /* ...où on met du texte par défaut... */
    LESetSelect(22, 28, hLE);           /* ...et où on sélectionne un mot par défaut */
    ShowWindow(fen);                            /* fenêtre maintenant visible */
    FlushEvents(EveryEvent, 0);                 /* ménage dans la file d'événements */
    arrow = GetCursorAdr( );                    /* on garde l'adresse du curseur flèche */

    do {
        SystemTask( );                  /* pour les accessoires de bureau périodiques */
        LEIdle(hLE);                            /* pour le clignotement du point d'insertion */
        AjusteCurs( );                          /* ajuste le dessin du curseur */
        if(!GetNextEvent(EveryEvent, &tache)) continue;

        switch(tache.what)                      /* quel événement? */
            {
            case MouseDown :
                sourisDans(FindWindow(&wind, tache.where)); /* réponse à un clic souris */
                break;

            case KeyDown :
            case AutoKey :
                clavier( );                     /* réponse aux événements de type clavier */
                break;

            case UpdateEvt :

                majFen(tache.message);          /* réponse à un événement de mise à jour */
                break;

            case ActivateEvt :
                actFen(tache.message);          /* réponse à un événement d'activation */
                break;
            }
        }
    while(indic);

    quitter(myID);                              /* fin de l'application */
}

            /***** FONCTION PLACEMENUS: installe la barre des menus *****/

    PlaceMenus( )

{
    InsertMenu(NewMenu(Menu4), 0);
    InsertMenu(NewMenu(Menu3), 0);
    InsertMenu(NewMenu(Menu2), 0);
    InsertMenu(NewMenu(Menu1), 0);
    FixAppleMenu(1);
    FixMenuBar( );
    DrawMenuBar( );
}

            /***** FONCTION EXECMENU: répond au choix d'un article de menu *****/

    int ExecMenu(art, menu)             /* retourne FALSE si quitter est choisi */
```

```
int  art;          /* article choisi */
int  menu;         /* dans ce menu */

{
    if (art>249) switch (art)
        {
        case iQuitter:
            return FALSE;
            break;

        case iAnnuler:
            SystemEdit(Undo);                    /* l'application ne gère pas l'article Annuler */
            break;

        case iCouper:
            if (!SystemEdit(Cut)) LECut(hLE);
            break;

        case iCopier:
            if (!SystemEdit(Copy)) LECopy(hLE);
            break;

        case iColler:
            if (!SystemEdit(Paste)) LEPaste(hLE);
            break;

        case iEffacer:
            if (!SystemEdit(Clear)) LEDelete(hLE);
            break;

        case iGauche:
            CheckMItem(FALSE, iGauche+justif);
            justif = 0;
            CheckMItem(TRUE, iGauche);
            InvalRect(&box); /* le rectangle est rendu invalide, il sera donc redessiné... */
            break;                               /* ...au prochain événement de mise à jour */
        case iDroite:
            CheckMItem(FALSE, iGauche+justif);
            justif = -1;
            CheckMItem(TRUE, iDroite);
            InvalRect(&box);
            break;

        case iCentre:
            CheckMItem(FALSE, iGauche+justif);
            justif = 1;
            CheckMItem(TRUE, iCentre);
            InvalRect(&box);
            break;
        }

    else if (art>0) OpenNDA(art);                /* ouverture accessoire de bureau */
    if (art) HiliteMenu(FALSE, menu);
    return TRUE;
}

        /***** FONCTION MAJFEN: mise à jour du contenu d'une fenêtre *****/

    majFen(port)

Pointer     port;                /* pointeur sur la fenêtre à rafraîchir */

{
Pointer     precPort;            /* pointeur sur le précédent grafport actif */
Rect        r;                   /* rectangle intermédiaire */

    precPort = GetPort( );               /* on mémorise le grafport actif */
    SetPort(port);                       /* on change de grafport */
    BeginUpdate(port);
    EraseRect(&rDV);                     /* on efface toute la ligne éditable */
    r = rDV;
    InsetRect(&r,-1,-1);
    FrameRect(&r);                       /* on dessine un rectangle contour */
    LEUpdate(hLE);                       /* on dessine la ligne éditable */
    r = box;
    InsetRect(&r,-4,-4);
    FrameRect(&r);                       /* on dessine un rectangle contour */
    LETextBox(text1,213,&box,justif); /* on dessine le texte justifié */
    EndUpdate(port);
    SetPort(precPort);           /* on rétablit le précédent grafport */
}

        /***** FONCTION ACTFEN: activation ou désactivation d'une fenêtre *****/

    actFen(port)

Pointer     port;                /* pointeur sur la fenêtre à activer ou désactiver */

{
    if (tache.modifiers & ActiveFlag) LEActivate(hLE);  /* activation */
    else LEDeactivate(hLE);                             /* désactivation */
}

        /***** FONCTION SOURISDANS: réponse à un clic souris *****/

    sourisDans(code)

int     code;        /* code retourné par FindWindow */

{
    if (code<0) SystemClick(&tache, wind, code);
    else switch (code)
        {
        case inMenuBar:
            MenuSelect(&tache, 0L);
            indic = ExecMenu(tache.TaskData);
            break;

        case inContent:
            if (wind != FrontWindow()) SelectWindow(wind);
            else LEClick(&tache, hLE);          /* gestion par Line Edit du clic souris */
            break;

        case inDrag:
            if (wind != FrontWindow() && !(tache.modifiers & AppleKey))
                SelectWindow(wind);
            DragWindow(0, tache.where, 0, 0L, wind);
            break;
        }
}

        /***** FONCTION CLAVIER: réponse à un événement clavier *****/

    clavier( )

{
```

```
int     car;

    car = tache.message & 0xFF;                     /* on garde l'octet le moins significatif */
    if (car == 0x08 || car == 0x15)                 /* flèches droite ou gauche... */
        LEKey(car, tache.modifiers, hLE);           /* ...on laisse faire Line Edit */
    else if (tache.modifiers & AppleKey)
        {
                                                    /* touche Pomme enfoncée... */
        MenuKey(&tache, 0L);                    /* ...c'est pour le Menu Manager */
        indic = ExecMenu(tache.TaskData);
        }
    else
        {
        if(tache.modifiers & OptionKey)             /* touche Option enfoncée... */
            car |= 0x80;                            /* ...on force le bit 7 à 1... */
        LEKey(car, tache.modifiers, hLE); /* ...on laisse faire Line Edit dans tous les cas */
        }
}

        /***** FONCTION AJUSTECURS: ajuste le dessin du curseur *****/

    AjusteCurs( )
{
static  modif;
int     ind;
long    pt;                                 /* le point où se trouve le curseur */

    if (FrontWindow( ) != fen) ind = FALSE;
    else                                    /* notre fenêtre doit être au premier plan */
        {
        GetMouse(&pt);          /* le point est en coordonnées locales du grafport actif */
        ind = PtInRect(&pt,&(*(LERec **) hLE)->ViewRect);
        }
    if (ind == modif) return;
    if (ind) SetCursor(IBeam);
    else SetCursor(arrow);
    modif = ind;
}
```

CHAPITRE IX

DIALOG MANAGER

PRINCIPES GÉNÉRAUX

Nous l'avons vu, une application communique avec l'utilisateur par l'intermédiaire des fenêtres, et l'utilisateur donne ses directives par l'intermédiaire des menus déroulants. Parfois, une commande appelle l'introduction de certains paramètres pour être prise en compte. Dans ce cas, l'application fait appel à une fenêtre de dialogue pour préciser lesdits paramètres. A d'autres moments, l'utilisateur peut faire une fausse manœuvre ou tente une action qui peut se révéler dangereuse. Pour le mettre en garde, l'application affichera une fenêtre d'alerte contenant soit des explications, soit la possibilité de faire marche arrière.

Le Dialog Manager a pour mission de gérer les fenêtres d'alerte et de dialogue. Pour mener à bien cette mission, il utilisera de manière transparente pour le programmeur des appels à QuickDraw, à l'Event Manager, au Window Manager, au Control Manager et à Line Edit. Ce qui explique la localisation de ce chapitre dans cet ouvrage.

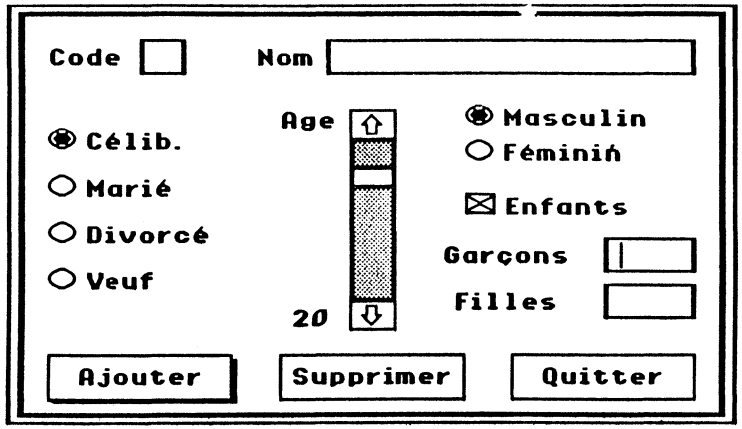


Figure IX.1. Un exemple de modal dialog.

Dans la suite de ce chapitre, on sera amené à distinguer deux sortes de dialogues : le *modal dialog* et le *modeless dialog*. Un modal dialog, au même titre qu'une alerte, interrompt le fonctionnement de l'application jusqu'à ce que l'utilisateur en sorte, soit en validant ses modifications et en acceptant de poursuivre sa commande (bouton OK ou équivalent), soit en annulant purement et simplement sa requête (bouton Annuler ou Cancel en anglais ou équivalent). Un modeless dialog n'interrompt pas le fonctionnement de l'application, mais coexiste avec elle, modifiant son comportement en fonction des options choisies parmi les possibilités qu'elle vient offrir. C'est une fenêtre supplémentaire dans l'environnement de l'application, et à ce titre, elle en a tous les aspects habituels : barre de titre, case de fermeture notamment. Elle pourra passer sous d'autres fenêtres de l'application et donc devenir inactive. La fenêtre d'un modal dialog ou d'une alerte, par contre, n'aura pas ces possibilités : un cadre fait d'un trait double, pas de barre de titre, et impossibilité de passer sous une fenêtre de l'application. Une fenêtre d'alerte sera toujours au premier plan.

Une fenêtre de dialogue pourra contenir toutes sortes d'éléments : du texte d'information et du texte éditable, des contrôles, des images QuickDraw ou des icônes, ainsi que tout autre objet défini par l'application. Un modal dialog contiendra nécessairement un bouton OK et un bouton Annuler, pour poursuivre la commande ou l'annuler. Une fenêtre d'alerte pourra contenir du texte d'information, des images ou icônes et des boutons simples (au moins un pour pouvoir sortir). Dans toutes ces fenêtres, un bouton pourra être privilégié, en ce sens qu'il sera équivalent à l'action d'enfoncer la touche Retour (ou Entrée) du clavier. Ce bouton aura une représentation caractéristique, permettant de le distinguer des autres boutons.

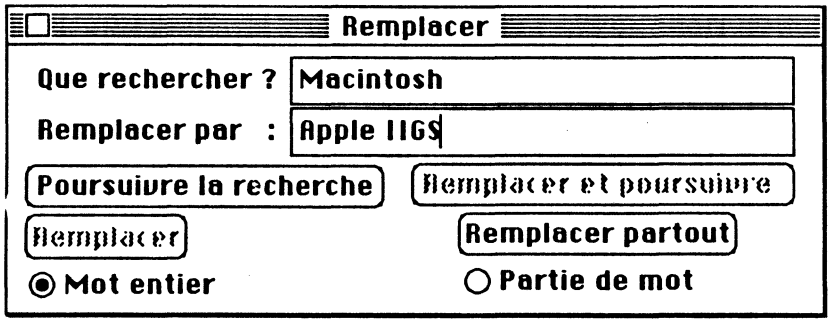


Figure IX.2. Un exemple de modeless dialog.

Dans la figure IX.3, on a un magnifique exemple **à ne pas suivre** : un bouton par défaut qui provoque la perte de toutes les données de l'utilisateur !

Figure IX.3. Un (mauvais) exemple d'alerte.

UTILISATION DU DIALOG MANAGER

Fenêtres de dialogue et d'alerte

Le Dialog Manager utilise ce type de fenêtre particulier pour créer une alerte ou un modal dialog : le cadre est fait d'un trait double, c'est tout : il n'y a pas de barre de titre et aucun des contrôles spécifiques à une fenêtre. L'application précisera la taille du contenu de cette fenêtre (le *PortRect* du grafport associé) et si elle est visible ou pas au moment de sa création.

La fenêtre d'un modeless dialog est tout à fait classique, et toutes les caractéristiques habituelles associées à une fenêtre pourront être utilisées : barre de titre avec case de fermeture et case de zoom, modification de taille, défilement du contenu, changement de plan, activation, invisibilité.

Alors que l'application n'aura aucune flexibilité pour manipuler une fenêtre d'alerte, elle pourra dessiner dans les fenêtres de dialogue comme bon lui semble : on récupérera en effet à leur création un pointeur sur le grafport associé, dans lesquel toute routine QuickDraw pourra être exécutée, avec respect de la notion de clip region.

Composantes (ou items) d'un dialogue ou d'alerte

Nous venons de dire qu'une fenêtre de dialogue est une fenêtre semblable à toutes les autres fenêtres. Quelle est donc la valeur ajoutée qu'apporte le Dialog Manager ? Elle réside dans les listes de composantes apparaissant au sein d'une fenêtre d'alerte ou de dialogue, composantes qu'il gérera automatiquement.

Chaque composante ou item de la liste contiendra les informations suivantes :

– un identifiant (entier sur 16 bits) désignant l'item : tout appel à une routine du Dialog Manager voulant référencer un item se fera par l'intermédiaire de cet identifiant. Cette valeur sera donc unique pour l'ensemble des items d'une même fenêtre de dialogue ou d'alerte ;

– le type de l'item (valeur précodée sur 16 bits), qui précisera sa nature et déterminera donc quel traitement le Dialog Manager doit lui appliquer ;

– un descripteur pour l'item (désigné par un pointeur ou un handle), tel un titre, une image ou une procédure de gestion dans certains cas particuliers ;

– une valeur (entier sur 16 bits) dépendant du type de l'item, telle la valeur initiale d'un contrôle ou la longueur limite d'une chaîne de caractères ;

– un rectangle d'affichage, pour déterminer la taille et la localisation de l'item au sein de la fenêtre ;

– un champ caractéristique (sur 16 bits), dépendant lui aussi partiellement du type de l'item ;

– le numéro d'une table de couleurs permettant de changer les couleurs standard utilisées par le Dialog Manager pour dessiner l'item.

Différentes routines nous permettront, connaissant le pointeur sur le grafport de la fenêtre de dialogue et l'identifiant d'un item à l'intérieur de cette fenêtre, de fixer ou de récupérer le contenu des caractéristiques de cet item. Caractéristiques que nous allons étudier plus en détail maintenant.

Identifiant d'un item

Dans toute liste d'items associés à une fenêtre de dialogue ou d'alerte, chaque item doit posséder un identifiant unique. Attention, aucun identifiant ne doit être égal à

zéro, car le Dialog Manager se sert de cette valeur pour retourner l'équivalent du message « item invalide ou non trouvé » dans certaines routines.

Par convention, dans une fenêtre d'alerte, l'item possédant l'identifiant numéro 1 doit correspondre au bouton *OK* (ou équivalent), et l'item possédant l'identifiant numéro 2 doit correspondre au bouton *Annuler* (ou *Cancel,* ou l'équivalent).

Dans une fenêtre de modal dialog, le Dialog Manager assume que l'item portant l'identifiant numéro 1 sera le bouton par défaut (à moins de spécifier autre chose), c'est-à-dire que si l'utilisateur appuie sur la touche Retour ou Entrée, l'application aura la même information que s'il avait cliqué dans l'item par défaut.

Si l'item numéro 1 est effectivement un bouton, le Dialog Manager l'entourera automatiquement d'un second trait (cas des boutons arrondis) ou l'ombrera (cas des boutons rectangulaires), pour que l'utilisateur sache que c'est le bouton par défaut. Si l'item numéro 1 n'est pas un bouton simple, ou s'il n'existe aucun item portant le numéro 1 dans la liste, le Dialog Manager ne gérera aucun bouton par défaut.

Type de l'item

Nous désignerons le type des items par l'une des constantes prédéfinies suivantes :

```
#define ButtonItem      10      /* bouton simple */
#define CheckItem       11      /* case à cocher */
#define RadioItem       12      /* bouton radio */
#define ScrollBarItem   13      /* barre de défilement */
#define UserCtlItem     14      /* contrôle propre à l'application */
#define StatText        15      /* texte statique */
#define LongStatText    16      /* texte statique long */
#define EditLine        17      /* ligne de texte éditable */
#define IconItem        18      /* icône */
#define PicItem         19      /* picture QuickDraw */
#define UserItem        20      /* item propre à l'application */
#define ItemDisable     0x8000  /* à ajouter pour rendre l'item muet */
```

Les cinq premiers types sont des contrôles tels que nous les avons rencontrés dans l'étude du Control Manager : le bouton simple qui déclenche une action immédiate (et notamment la sortie d'un modal dialog ou d'une alerte), la case à cocher qui prend la valeur 1 (cochée) ou 0 (non cochée), le bouton radio (membre d'une famille de boutons radio parmi lesquels un seul à la fois est sélectionné), la barre de défilement (seul exemple prédéfini de cadran) et tout autre contrôle que l'application aurait créé et gérerait elle-même. Pour agir sur ces items, le Dialog Manager fera évidemment appel aux routines du Control Manager, mais de manière transparente au programmeur.

Note La case à cocher s'appelle *CheckItem* et une procédure du Menu Manager **CheckMItem**. Attention aux homonymies !

Les types suivants sont propres au Dialog Manager.

• Le texte statique *(StatText)* et le texte statique long *(LongStatText)* représentent des chaînes de caractères que l'utilisateur ne pourra pas modifier (non éditables). Elles serviront à afficher des messages, plus ou moins longs. Elles pourront s'étendre sur plusieurs lignes, par insertion de *return* en leur sein. Le *return,* ou retour-chariot, possède le code ASCII 13 et est symbolisé en C par le caractère spécial \r. Un texte statique peut posséder jusqu'à quatre zones paramétrables, repérées par les caractères particuliers ^0, ^1, ^2 et ^3. On donnera une valeur (alphanumérique) à chaque paramètre, et le Dialog Manager substituera les caractères particuliers. Le texte statique est une chaîne de type Pascal. Elle est donc limitée à 255 caractères, et le premier octet contient la longueur de la chaîne. Le texte statique long est une suite de caractères en nombre inférieur à 32767, le nombre de caractères étant stocké ailleurs (dans le champ valeur associé à l'item).

• *EditLine* représente une ligne (et une seule à la fois) de texte éditable, c'est-à-dire que l'utilisateur pourra saisir et modifier à sa guise. Un tel item permettra à l'application d'obtenir de la part de l'utilisateur des informations littérales ou chiffrées, impossibles à renseigner par l'intermédiaire d'un contrôle. Pour gérer ce type d'items, le Dialog Manager fera bien évidemment appel aux routines de LineEdit, ce qui signifie que toutes les commandes applicables en LineEdit sur du texte éditable continueront à s'appliquer lors de l'utilisation de ces items. Cette ligne éditable est une chaîne de caractères de type Pascal. On pourra en limiter le nombre de caractères en précisant cette limite dans le champ valeur associé à l'item. Notons que quand un modal dialog contient des lignes éditables, il y en a toujours une active (matérialisée par du texte sélectionné ou un point d'insertion clignotant). On peut sauter d'une ligne éditable à une autre par l'intermédiaire de la touche de tabulation.

• Les items de type icône et picture sont a priori là pour faire joli. Cependant, comme le Dialog Manager est capable de dire à l'application que l'utilisateur a cliqué dans un tel item, on pourrait très bien imaginer de les utiliser comme des boutons particuliers, pour déclencher un traitement dans un dialogue. Encore faut-il que l'icône soit suffisamment explicite, ou accompagnée d'un commentaire explicite !

• Les items propres à l'application peuvent être n'importe quoi, pourvu qu'ils soient gérés entièrement par l'application.

Quel que soit leur type, les items d'une fenêtre de dialogue peuvent être rendus *muets*. Chaque fois que l'utilisateur clique dans le rectangle de visualisation d'un item ou saisit un caractère dans une ligne éditable, le Dialog Manager renseigne l'application de ces événements. Dès qu'un item est rendu muet *(disabled item),* le Dialog Manager cesse de renseigner l'application sur les événements qui l'affectent. Par conséquent, il est interdit de rendre muet un bouton simple, puisque l'application ne serait alors plus capable de savoir si l'utilisateur a cliqué dedans pour lancer la commande ! A contrario, tous les textes statiques devraient être muets, puisque l'utilisateur ne peut avoir aucune action sur eux. De même les icônes et pictures auxquelles on n'aurait pas accordé de vertu particulière. En règle générale, tout item dont on n'a pas besoin de connaître la valeur durant le dialogue peut être rendu muet. Un cas typique : la ligne de texte éditable. Si l'application veut filtrer ce que rentre l'utilisateur (par exemple accepter les chiffres et rejeter le reste), l'item n'est pas muet et le contrôle peut s'effectuer chaque fois qu'une touche est enfoncée lorsque cet item est sélectionné. Si par contre l'application accepte n'importe quel caractère ou ne souhaite faire un contrôle qu'en fin de saisie, l'item peut être rendu muet.

Un item est normal si son type possède la valeur prédéfinie qui lui est affectée. Pour rendre muet un item, il suffit d'ajouter à son type la valeur *ItemDisable.* Ceci n'est à faire qu'à la création, car le Dialog Manager nous propose deux procédures pour rendre muet ou normal un item quelconque d'une fenêtre de dialogue.

On se gardera bien de confondre les qualificatifs *muet* et *inactif* (ou *désactivé*). Un item muet est un item actif, sur lequel l'utilisateur peut agir, même si l'application n'en a pas connaissance. Visuellement, un item muet n'est pas différent d'un item normal. Par contre, un item inactif (au sens Control Manager du terme) n'est pas accessible par l'utilisateur : il ne répond plus à ses sollicitations. Il est d'ailleurs représenté différemment à l'écran (titres estompés notamment). Pour désactiver un item, il faudra employer la procédure **HiliteControl** du Control Manager. Le Dialog Manager permettra quant à lui de rendre un item invisible.

Descripteur de l'item

Quel que soit le type de l'item, son descripteur donne l'adresse de quelque chose, par l'intermédiaire d'un pointeur ou d'un handle. Le quelque chose dépend du type, comme nous l'allons voir immédiatement.

• Pour un bouton simple, une case à cocher ou un bouton radio, le descripteur contiendra l'adresse du titre associé (chaîne de type Pascal).

• Le descripteur pour une barre de défilement sera un pointeur sur la fonction d'action associée. Cette fonction sera appelée au moment de l'initialisation, et chaque

fois que l'utilisateur provoquera un défilement. Elle devra gérer en temps réel la mise à jour de tout ce qui dépend de la valeur prise par ce contrôle, sans rien apporter à l'application elle-même, de telle sorte que si l'item est rendu muet, l'application ne saura même pas que l'utilisateur a cliqué dedans !

Attention La fonction d'action d'une barre de défilement dans un dialogue s'apparente évidemment à la procédure d'action déjà vue dans le chapitre sur le Control Manager (puisqu'elle va la générer), mais il y a quelques différences notables à prendre en compte (notamment le nombre et la nature des arguments, le fait qu'elle retourne une valeur, le fait qu'elle est appelée pour créer le contrôle).

La fonction d'action d'une barre de défilement appartenant à un dialogue doit suivre des règles très précises : elle sera obligatoirement déclarée de type Pascal, acceptera trois arguments et retournera un résultat sur 16 bits. Si notre fonction s'appelle *Defilement,* elle aura un peu cette allure :

```
pascal int Defilement(commande, dialogue, id)

int     commande;
Pointer dialogue;
int     id;

{
int valeur;

    switch(commande)
        {
        case 1:
            ...                 /* prise en compte de l'identifiant de l'item */
            return valeur;      /* valeur "minimale" contrôlée par le défilement */
            break;

        case 2:
            ...                 /* prise en compte de l'identifiant de l'item */
            return valeur;      /* valeur "maximale" contrôlée par le défilement */
            break;

        case 3:
            ...                 /* prise en compte de l'identifiant de l'item */
            return valeur;      /* valeur initiale du contrôle */
            break;

        case 4:
            valeur = GetItemValue(dialogue, id); /* ancienne valeur (avant l'événement) */
            ...                 /* défilement d'une unité vers le haut */
            return valeur;      /* nouvelle valeur pour le contrôle (modifiée) */
            break;

        case 5:
            valeur = GetItemValue(dialogue, id); /* ancienne valeur (avant l'événement) */
            ...                 /* défilement d'une unité vers le bas */
            return valeur;      /* nouvelle valeur pour le contrôle (modifiée) */
            break;

        case 6:
            valeur = GetItemValue(dialogue, id); /* ancienne valeur (avant l'événement) */
            ...                 /* défilement d'une "page" vers le haut */
            return valeur;      /* nouvelle valeur pour le contrôle (modifiée) */
            break;

        case 7:
            valeur = GetItemValue(dialogue, id); /* ancienne valeur (avant l'événement) */
            ...                 /* défilement d'une "page" vers le bas */
            return valeur;      /* nouvelle valeur pour le contrôle (modifiée) */
            break;

        case 8:
            valeur = GetItemValue(dialogue, id);    /* nouvelle valeur après déplacement
                                                       du curseur de défilement */
            ...                 /* défilement jusqu'à la nouvelle valeur */
            return valeur;      /* nouvelle valeur pour le contrôle (inchangée) */
            break;
        }
}
```


La fonction doit répondre à toutes les sollicitations du Dialog Manager, et ce pour une barre de défilement particulière, ou bien pour un ensemble de barres de défilement : on n'est pas obligé d'en avoir une par item, si leur fonctionnement est semblable. Elle est appelée au moment de l'initialisation de l'item, et lors de la gestion de la fenêtre de dialogue. C'est le Dialog Manager qui fixe la valeur de ses arguments, et notamment le premier, qui définit le résultat que le Dialog Manager attend. Quand les commandes 1, 2 ou 3 sont passées, le contrôle n'existe pas encore : le Dialog Manager a besoin de ces valeurs pour les passer en bloc au Control Manager qui va créer la barre de défilement. Grâce à ces valeurs, la taille et la position du curseur de défilement pourront être gérées par le Control Manager. Quand la commande 4 est passée, c'est que l'utilisateur a cliqué dans la flèche du haut (barre verticale) ou de gauche (barre horizontale). La fonction demande l'ancienne valeur de l'item, répond immédiatement à l'événement en gérant le « défilement », et retourne la nouvelle valeur du contrôle. A l'identique, la commande 5 correspond à un clic dans la flèche du bas (ou de droite), les commandes 6 à 7 à des clics dans la bande de défilement (*PageUp* et *PageDown*). La commande 8 intervient quand l'utilisateur a déplacé le curseur de défilement. La fonction commence par se renseigner sur la nouvelle valeur de l'item (attention : la nouvelle, pas l'ancienne), et fait le travail approprié pour afficher les éléments en concordance avec cette valeur.

En réponse aux commandes 4 à 7, on veillera à ce que la nouvelle valeur appartienne bien à l'intervalle autorisé, c'est-à-dire qu'elle soit comprise entre 0 et max-min (voir le chapitre sur le Control Manager pour plus de précisions).

- Pour un contrôle propre à l'application, le descripteur est un pointeur sur la procédure de définition du contrôle (dont nous avons évité de parler dans le chapitre sur le Control Manager).

- Pour un texte statique, le descripteur pointe sur la chaîne de caractères de type Pascal. Pour un texte statique long, le descripteur pointe sur le début du texte.

- Dans le cas d'une ligne éditable, le descripteur pointe sur la chaîne de caractères qui sera affichée par défaut et que l'utilisateur pourra modifier. S'il n'y a pas d'affichage par défaut, on mettra zéro-long. Le texte par défaut de la première ligne éditable du dialogue sera entièrement sélectionné au moment de la création de l'item.

- Le descripteur en ce qui concerne les icônes et pictures est un handle sur l'icône ou la picture concernée. Une icône est définie par un rectangle (dont la largeur est un multiple de 8) suivi de la pixel image de l'icône, une picture est un objet QuickDraw.

- Pour un item propre à l'application, le descripteur est un pointeur sur la procédure de définition de l'item (dont nous ne parlerons pas plus que les contrôles propres à l'application).

Valeur de l'item

Tout comme le descripteur, la signification de la valeur associée à l'item dépend étroitement de son type. Cette valeur sera fixée à la création de l'item et pourra ou non varier durant le dialogue en fonction des sollicitations de l'utilisateur.

- Pour les boutons, la valeur est toujours nulle.

• Pour les cases à cocher et les boutons radio, la valeur à la création est la valeur initiale du contrôle (TRUE ou FALSE dans le cas général, nous utiliserons plutôt 1 ou 0). Cette valeur se modifie au cours du dialogue et est toujours égale à la valeur actuelle du contrôle.

• Pour les barres de défilement, la valeur est toujours comprise entre 0 et max-min, ainsi que nous l'avons déjà vu dans le chapitre consacré au Control Manager.

• Pour un texte statique long, la valeur donne sa longueur (son nombre de caractères) et donc ne varie pas durant le dialogue.

• Pour une ligne éditable, le champ valeur contient la longueur maximale autorisée pour la chaîne de caractères, entre 0 et 255. L'utilisateur ne peut modifier cette valeur, l'application si oui.

• Dans le cas des textes statiques courts, des icônes et des pictures, le champ valeur est libre d'utilisation par l'application (le Dialog Manager ne s'en sert pas).

Rectangle d'affichage de l'item

Chaque item de la liste est représenté dans un rectangle d'affichage. Ce rectangle est donné dans le système de coordonnées locales et situe donc l'item à l'intérieur de la fenêtre. Le rectangle doit complètement englober l'item, sinon certains types d'items peuvent être coupés, le rectangle pouvant agir comme une clip region pour leur représentation. Quand un item est rendu invisible, le Dialog Manager fait en réalité (entre autres choses) un **EraseRect** du rectangle d'affichage de l'item concerné. Quand un clic souris est détecté, il est censé s'adresser à un item particulier s'il a eu lieu dans le rectangle d'affichage de cet item. Si deux ou plusieurs rectangles ont une intersection non vide, c'est le premier item rencontré dans la liste qui sera pris en compte au cas où le clic souris se produirait dans cette intersection.

• Pour l'ensemble des contrôles (boutons, cases à cocher, boutons radio, barres de défilement), le rectangle d'affichage est celui qui servira à la création du contrôle par le Control Manager (appelé directement par le Dialog Manager).

• Pour une ligne éditable, le rectangle est pris comme le rectangle de visualisation au sens de LineEdit. Donc aucun texte ne peut être saisi ou affiché en dehors de ce rectangle. De plus, le Dialog Manager dessine un cadre autour de ce rectangle, ce qui permet d'identifier les textes éditables.

• Pour les textes statiques (courts ou longs), le rectangle possède le même comportement que pour la ligne éditable, mais peut contenir plusieurs lignes. Aucun cadre n'est dessiné autour du rectangle.

• Pour les icônes et les pictures, le rectangle est pris comme rectangle de destination, conformément à l'un des paramètres des procédures QuickDraw **PaintPixels** ou **DrawPicture**. Les images sont donc mises à l'échelle (avec les déformations qui s'ensuivent) ou clippées pour tenir à l'intérieur du rectangle.

Champ caractéristique de l'item

Pour les véritables contrôles (boutons simples, cases à cocher, boutons radio et barres de défilement), ce champ contient des caractéristiques absolument identiques à celles rencontrées dans le chapitre consacré au Control Manager. Nous ne les redonnerons donc pas ici.

En ce qui concerne les types propres au Dialog Manager, aucune définition n'est disponible. On passera donc la valeur 0.

Couleur de l'item

Pour les véritables contrôles (boutons simples, cases à cocher, boutons radio et barres de défilement), ce champ contient des caractéristiques absolument identiques à

celles rencontrées dans le chapitre consacré au Control Manager. Nous ne les redonnerons donc pas ici.

En ce qui concerne les types propres au Dialog Manager, aucune définition n'est disponible. On passera donc la valeur 0.

Structure de type dialogue

Une application va créer des listes d'items correspondant à des fenêtres d'alerte ou de dialogue. Le Dialog Manager va stocker ces données dans une structure particulière, appelée *dialog record*. Cette structure ne sera pas décrite, pour la bonne et simple raison qu'une application n'a pas le droit d'aller modifier directement les données qu'elle contient. C'est le Dialog Manager et lui seul qui doit y accéder. Pour qu'une application puisse modifier les caractéristiques d'un item, il fournit tout un ensemble de procédures qui rendent inutiles la divulgation du format de cette structure, ce qui permettra éventuellement de le revoir lors d'améliorations futures sans nuire à la compatibilité des applications existantes.

Sur Macintosh, les alertes et dialogues étaient déclarés sous forme de ressources, dans des fichiers séparés du programme source, suivant un format très précis. Sur l'Apple IIGS, on aura deux manières de procéder : soit créer les éléments un à un, soit utiliser des formats prédéfinis *(templates)*, cette dernière possibilité palliant partiellement l'absence du *resource manager* sur cette machine.

Sauf cas particulier, on utilisera les routines du Dialog Manager et non celles du Control Manager pour manipuler les items d'un dialogue. Par construction, tous les items ont une structure de contrôle. C'est évident pour les vrais contrôles (boutons simples, cases à cocher, boutons radio, barres de défilement), mais c'est aussi le cas pour les items propres au Dialog Manager. On risque d'obtenir des résultats surprenants en manipulant ces items comme de vrais contrôles ! Une seule procédure du Control Manager sera vraiment intéressante à utiliser, **HiliteControl** pour désactiver ou réactiver un contrôle (un vrai !), en cours de dialogue.

Cas particulier des alertes

Une alerte est repérée par un identifiant. Elle se présente sous la forme d'une fenêtre sans barre de titre, contenant une liste d'items qui sont soit de simples boutons (au moins un et généralement pas plus de deux), soit des items informatifs (texte statique, icône, picture). Il n'y a pas *dialogue* avec l'utilisateur : celui-ci peut seulement dire à l'application qu'il a bien reçu le message.

Quand l'alerte doit servir à expliquer à l'utilisateur qu'il s'est trompé pour une raison ou pour une autre, on peut avoir envie de donner des messages différents en cas de récidive (plus de renseignements par exemple en cas de persistance dans l'erreur).

Le Dialog Manager offre quatre niveaux pour une alerte donnée. Pour chaque niveau, l'application doit décrire la marche à suivre. A la première erreur, c'est l'action numéro 1 qui sera utilisée (par exemple un simple « bip », sans affichage de fenêtre). A la deuxième erreur identique consécutive, l'action numéro 2 prendra le relais (par exemple en affichant un message explicatif sans « bip » sonore). A la troisième erreur consécutive, l'action numéro 3 prendra le relais (par exemple en lisant par voix synthétisée le message affiché dans la fenêtre). A partir de la quatrième erreur consécutive et pour toutes les suivantes, c'est l'action 4 qui sera appelée.

Pour savoir si deux alertes sont consécutives, le Dialog Manager garde en mémoire l'identifiant de la dernière alerte utilisée. Si l'alerte actuelle porte le même identifiant, il y a consécutivité, sinon le niveau est remis à 1.

Pour définir chaque niveau d'action, on donnera les trois informations suivantes ;
- quel est le bouton par défaut (*OK* ou *Annuler*) ;
- la fenêtre doit-elle être dessinée ou non ;
- lequel des quatre sons possibles doit être émis à ce niveau.

Les sons sont numérotés de 0 à 3. Par défaut, le son 0 est insonore, le son 1 émet un « bip », le son 2 deux « bips » et le son 3 trois « bips ». Si le programmeur souhaite personnaliser ses sons, il pourra le faire en écrivant une procédure style Pascal, admettant un argument qui peut prendre les valeurs 0 à 3. La procédure, si nous l'appelons *MesSons,* aura la forme suivante :

```
pascal void MesSons(quelSon)

int  quelSon;

{
switch(quelSon)
    {
    case 0:
        ...        /* définition du son 0 */
        break;

    case 1:
        ...        /* définition du son 1 */
        break;

    case 2:
        ...        /* définition du son 2 */
        break;

    case 3:
        ...        /* définition du son 3 */
        break;
    }
}
```

Pour respecter l'interface utilisateur, le numéro 1 devrait toujours être un « bip » simple. Il est en effet automatiquement utilisé par le Dialog Manager chaque fois qu'un utilisateur clique en dehors d'une fenêtre d'alerte ou de modal dialog.

EXEMPLES D'UTILISATION

Initialisation

Avant d'initialiser le Dialog Manager, il faut avoir initialisé le Memory Manager, QuickDraw, l'Event Manager, le Window Manager, le Control Manager et LineEdit, dans cet ordre. On peut alors appeler la procédure **DialogStartUp**, qui installe la procédure standard de sons, initialise à vide les chaînes de caractères pouvant servir de paramètres dans les textes statiques, prend pour police de caractères la police système et fixe le niveau d'alerte à 1. Un seul argument est requis, le sempiternel numéro d'application retourné par la fonction **MMStartUp** du Memory Manager.

Pour empêcher le Dialog Manager d'utiliser la police de caractères système pour dessiner les textes (titres des contrôles, textes statiques et lignes éditables), on peut utiliser la procédure **SetDAFont** en passant en argument un handle sur la nouvelle police choisie.

Création d'un modal dialog et de ses items

Il existe plusieurs moyens pour créer un modal dialog, suivant qu'on utilise ou non les modèles prédéfinis *(templates)*. Il existe deux sortes de modèles prédéfinis : les modèles d'items et les modèles de dialogues.

◇ Quand on n'utilise pas de modèle prédéfini, on commence par appeler la fonction **NewModalDialog**, qui va créer la fenêtre de dialogue, puis autant de fois qu'on veut d'items **NewDItem**, pour créer les items de cette fenêtre.

NewModalDialog réclame les trois arguments nécessaires à la création d'une fenêtre dont le type est forcé (pas de barre de titre, aucun contrôle standard associé) :

– le rectangle qui sera le *PortRect* du grafport associé à la fenêtre (en coordonnées globales, pour définir sa taille et sa localisation à l'écran), repéré par un pointeur ;

– un indicateur précisant si la fenêtre sera visible (TRUE) ou pas (FALSE) ;

– la valeur d'utilisation libre associée à chaque fenêtre (*wRefCon,* un entier sur 32 bits).

La fonction **NewModalDialog** retourne un pointeur sur le grafport de cette fenêtre de dialogue, que nous appellerons désormais pointeur sur dialogue, même si le terme est impropre. Elle alloue l'espace nécessaire à la structure dialogue. Si la fenêtre est déclarée visible, elle est dessinée à l'écran. Sinon elle est seulement créée et pourra être rendue visible plus tard par la procédure **ShowWindow** du Window Manager. Cette possibilité est intéressante pour faire apparaître tous les items en même temps, et non au fur et à mesure de leur création. On veillera comme d'habitude à ce que la fenêtre créée n'aille pas se cacher partiellement sous la barre de menus, ce qui fait négligé.

```
Rect    r;                  /* un rectangle */
Pointer dlg;                /* sera le pointeur sur dialogue */

SetRect(&r, 50, 50, 270, 150);          /* on définit le rectangle contenu */
dlg = NewModalDialog(&r, FALSE, 0L); /* on crée la fenêtre du modal dialog (invisible) */
```

Une fois un pointeur sur dialogue récupéré, on peut appeler la procédure **NewDItem** pour ajouter un item à la liste. Cette procédure réclame huit arguments. Le premier indique à quel dialogue appartient l'item, par l'intermédiaire d'un pointeur sur dialogue. Les sept autres arguments décrivent les caractéristiques de l'item : son identifiant, un pointeur sur son rectangle d'affichage, son type, son descripteur, sa valeur initiale, son champ caractéristique et un pointeur sur sa définition de couleur. Nous ne reviendrons pas sur la description de ces arguments (voir plus haut).

```
char    OKstr[ ] = "\2OK";          /* OKstr pointe sur une chaîne Pascal */
Rect    OKrect = {10, 10, 30, 30};  /* rectangle en coordonnées locales */

NewDItem(dlg, 1, &OKrect, ButtonItem, OKstr, 0, 0, 0L);    /* ajout d'un item */
...                                 /* ajout d'autres items */
ShowWindow(dlg);                    /* on rend la fenêtre visible */
SelectWindow(dlg);                  /* on rend la fenêtre active */
```

La procédure **NewDItem** associe l'item à la fenêtre précisée en argument. Il apparaît à l'écran seulement si celle-ci est visible. Rappelons que si à la valeur du type on a ajouté la valeur prédéfinie *ItemDisable,* l'item sera muet et les événements l'affectant ne seront pas rapportés à l'application.

Notons qu'on passe 0 dans le champ caractéristique, et pourtant le bouton sera le bouton par défaut, parce qu'il porte l'identifiant 1. Le Control Manager nous aurait obligé à passer la valeur 1 dans le champ, le Dialog Manager l'ajoute pour nous.

◇ Un modèle prédéfini peut être considéré en C comme une structure particulière, qu'on manipulera classiquement. Manipuler est un bien grand mot, car en général, tout ce que l'application aura à faire sur elle, c'est son initialisation.

• Le modèle d'item pourra être défini de la manière suivante :

```
struct _ItemTemplate {
    int     itemID;          /* identifiant de l'item */
    Rect    itemRect;        /* rectangle d'affichage de l'item */
    int     itemType;        /* type de l'item, éventuellement rendu muet */
    Pointer itemDescr;       /* descripteur de l'item, pointeur ou handle */
    int     itemValue;       /* valeur de l'item à la création */
    int     itemFlag;        /* champ caractéristique de l'item */
    Pointer itemColor;       /* pointeur sur table de couleurs */
    };
#define ItemTemplate struct _ItemTemplate
```

On retrouve intégralement les caractéristiques d'un item, telles qu'elles ont été décrites plus haut et telles qu'elles sont passées en argument à la fonction **NewDItem** (à la différence près que dans cette structure on manipule le rectangle, et non un pointeur sur rectangle). Par contre, aucune référence n'est faite à une fenêtre de dialogue particulière : le modèle d'item est indépendant du dialogue dans lequel il sera utilisé, et rien n'empêche de créer des modèles d'items à utiliser dans plusieurs dialogues différents, tels les boutons *OK* et *Annuler,* par exemple.

– modèle pour un bouton *OK* :

```
ItemTemplate  OKbouton = {
    1,                  /* identifiant: ce sera le bouton par défaut */
    {10, 10, 30, 50},   /* rectangle en coordonnées locales */
    ButtonItem,         /* type bouton, valeur prédéfinie */
    "\2OK",             /* pointeur sur titre, chaîne type Pascal */
    0,                  /* valeur initiale, toujours 0 pour un bouton simple */
    2,                  /* caractéristiques (bouton rectangulaire) */
    0L                  /* couleurs par défaut */
    };
```

– modèle pour une case à cocher :

```
ItemTemplate  cocher = {
    5,                  /* identifiant: strictement supérieur à 1 */
    {35, 10, 50, 130},  /* rectangle en coordonnées locales */
    CheckItem,          /* type case à cocher, valeur prédéfinie */
    "\12Déjà coché",    /* pointeur sur titre, chaîne type Pascal */
    1,                  /* valeur initiale, la case sera cochée */
    0,                  /* caractéristiques (rien à signaler) */
    0L                  /* couleurs par défaut */
    };
```

– modèle pour un bouton radio :

```
ItemTemplate  radio = {
    12,                 /* identifiant: strictement supérieur à 1 */
    {55, 10, 70, 130},  /* rectangle en coordonnées locales */
    RadioItem,          /* type bouton radio, valeur prédéfinie */
    "\10Option 1",      /* pointeur sur titre, chaîne type Pascal */
    1,                  /* valeur initiale, ce sera le bouton sélectionné */
    6,                  /* caractéristiques (appartient à la famille 6) */
    0L                  /* couleurs par défaut */
    };
```


– modèle pour une barre de défilement :

```
int       Defilement( );    /* déclaration de la fonction d'action */
ItemTemplate  barre = {
    23,                     /* identifiant: strictement supérieur à 1 */
    {40, 140, 135, 160},    /* rectangle en coordonnées locales */
    ScrollBarItem + ItemDisable,   /* type barre de défilement (muette) */
    Defilement,             /* pointeur sur la fonction d'action, définie ailleurs */
    19,                     /* valeur initiale, comprise entre 0 et max–min */
    3,                      /* caractéristiques (barre verticale avec deux flèches) */
    0L                      /* couleurs par défaut */
    };
```

– modèle pour un texte statique court :

```
ItemTemplate  textstat = {
    28,                     /* identifiant: strictement supérieur à 1 */
    {95, 10, 110, 130},     /* rectangle en coordonnées locales */
    StatText + ItemDisable,     /* type texte statique (muet) */
    "\10un texte",          /* pointeur sur texte, chaîne type Pascal */
    0,                      /* valeur initiale, ignorée du Dialog Manager */
    0,                      /* caractéristiques (rien à signaler) */
    0L                      /* couleurs par défaut */
    };
```

– modèle pour une ligne éditable :

```
ItemTemplate  ligne = {
    29,                     /* identifiant: strictement supérieur à 1 */
    {115, 10, 130, 130},    /* rectangle en coordonnées locales */
    EditLine + ItemDisable, /* type ligne éditable (muette) */
    "\6défaut",             /* pointeur sur texte par défaut, chaîne type Pascal */
    15,                     /* nombre maximal de caractères de la ligne éditable */
    0,                      /* caractéristiques (rien à signaler) */
    0L                      /* couleurs par défaut */
    };
```

La procédure **GetNewDItem** permet d'ajouter à un dialogue un item défini par un *ItemTemplate* : il suffit de lui passer deux arguments, un pointeur sur le dialogue visé et un pointeur sur l'item désiré.

```
GetNewDItem(dlg, &OKbouton);    /* ajout d'un nouvel item au dialogue dlg */
```

Cette instruction est équivalente à celle vue plus haut, avec **NewDItem**, mais présente l'avantage d'être plus facile à maintenir, puisque les données sont séparées de l'instruction elle-même. Il sera donc préférable d'utiliser les modèles prédéfinis, et même de les initialiser dans des fichiers séparés des fichiers d'instructions programme proprement dits.

• Le modèle de dialogue pourra être défini de la manière suivante :

```
struct _DialogTemplate {
    Rect    BoundsRect;     /* rectangle contenu de la fenêtre */
    int     Visible;        /* TRUE si fenêtre visible, FALSE sinon */
    long    RefCon;         /* valeur libre associée à la fenêtre */
    Pointer Items[ ];       /* liste variable de pointeurs sur item, terminée par 0L */
    };
#define DialogTemplate struct _DialogTemplate
```

On retrouve au début du modèle les trois valeurs qu'on passe en argument à la fonction **NewModalDialog**, à la différence qu'on utilise le rectangle et non un pointeur sur rectangle. On donne ensuite une liste de pointeurs sur items définis par *ItemTemplate*, liste de longueur variable terminée par zéro-long.

Supposons qu'on ait créé cinq items comme ceux vus plus haut, et qu'on les ait appelés *item 1, item 2, item 3, item 4* et *item 5*. On pourra initialiser un *DialogTemplate* ainsi :

```
DialogTemplate dialogue = {
    {50, 50, 150, 270},                    /* le rectangle */
    TRUE,                                  /* fenêtre visible */
    0L,                                    /* valeur d'utilisation libre */
    {&item1, &item2, &item3, &item4, &item5, 0L}  /* liste des items */
    };
```

Il suffira alors d'appeler la fonction **GetNewModalDialog** en lui passant l'adresse de ce *DialogTemplate* pour créer la fenêtre de dialogue avec tous ses items en place. Cette fonction retournera un pointeur sur le dialogue, qui est rappelons-le plus exactement un pointeur sur le grafport de la fenêtre de dialogue.

```
dlg = GetNewModalDialog(&dialogue);      /* création de la fenêtre complète */
```

La fenêtre ayant été déclarée visible dans le template, elle devient immédiatement active. Comme pour les items, le fait de travailler avec des templates présente l'intérêt de pouvoir séparer les données de l'instruction proprement dite, ce qui facilite la maintenance de l'application, et son internationalisation.

- La bonne marche à suivre sera donc la suivante :

1. Déclarer et initialiser toutes les chaînes de caractères (titres, textes statiques, contenus par défaut d'une ligne éditable). Cette étape est optionnelle, nous avons vu que nous pouvons glisser directement les chaînes de caractères à l'intérieur de la définition de l'item, parce que le champ *itemDescr* est déclaré de type *Pointer*, et non de type *long* (à condition que l'environnement de travail l'accepte).

2. Déclarer s'il y a lieu toutes les fonctions d'action dont l'adresse intervient dans la définition des items.

3. Décrire tous les items des dialogues en suivant les modèles prédéfinis.

4. Décrire les fenêtres de dialogues en suivant les modèles prédéfinis. Une règle importante devra être suivie, pour éviter des surprises désagréables. Puisque la structure *DialogTemplate* est de longueur variable, il faut commencer par déclarer celle qui a le plus d'items. Cela fixera une taille maximale, qui ne gênera pas les autres déclarations.

Ces quatre étapes s'effectueront en dehors de toute fonction (variables globales), et même dans un fichier de données séparé, quitte à l'insérer par la suite à l'endroit requis grâce à une directive #include.

Gestion d'un modal dialog

Une fois que la fenêtre de modal dialog est créée, que tous ses items sont en place, qu'elle est visible et active (elle est donc au premier plan), l'utilisateur peut commencer à jouer avec. Tous les événements survenant à partir de ce moment vont être gérés par une fonction unique, **ModalDialog**, qui doit être appelée jusqu'à ce que l'utilisateur quitte le dialogue en cliquant dans un bouton. En d'autres termes, il faut boucler sur cette fonction jusqu'à ce que le dialogue soit fermé.

Comment fonctionne **ModalDialog** ? Si la fenêtre de premier plan n'est pas un modal dialog, elle retourne la valeur zéro et ne fait rien. Si la fenêtre de premier plan est effectivement un modal dialog (c'est tout de même préférable !), **ModalDialog** va intercepter les événements et les gérer elle-même. De plus, si l'événement a lieu dans un item normal (non muet), elle retournera l'identifiant de l'item, de telle sorte que l'application pourra à son tour faire les actions appropriées liées à cet item (par exemple activer ou désactiver d'autres items, changer la valeur d'un contrôle ou ne rien faire du tout), avant de revenir appeler **ModalDialog**.

ModalDialog admet un argument unique, qui est un pointeur sur une fonction type Pascal qui sert de filtre aux événements. Quand la valeur zéro-long est passée en argument, un filtre standard est utilisé : si l'utilisateur appuie sur la touche Retour ou Entrée, **ModalDialog** retournera l'identifiant du bouton par défaut ; et les combinaisons de touches Pomme-X, Pomme-C et Pomme-V permettront le couper-copier-coller à l'intérieur du dialogue.

Si nous voulons utiliser notre propre filtre, nous écrirons une fonction type Pascal possédant trois arguments, de la manière suivante :

```
pascal int monFiltre(dialog, event, itemHit)

Pointer    dialog;          /* pointeur sur dialogue */
TaskRec    * event;         /* pointeur sur événement */
int        * itemHit;       /* pointeur sur l'item concerné */

{
int car;

if ((event->what == KeyDown || event->what == AutoKey) &&
                              !(event->modifiers & AppleKey))
    {
    car = (int) event->message & 0x000000FF;
    car = (car < 'a' || car > 'z') ? car : car + 'A' - 'a';
    event->message = (long) car;
    }
return FALSE;
}
```

La fonction filtre précédente teste si l'événement est une touche enfoncée (*KeyDown* ou *AutoKey*) sans que la touche Pomme le soit elle-même. Si ces conditions sont vraies, elle récupère le code ASCII du caractère enfoncé, et transforme les lettres minuscules en lettres majuscules. Dans tous les cas, la fonction retourne la valeur FALSE, indiquant à **ModalDialog** qu'elle doit traiter l'événement comme si de rien n'était. On constate par cet exemple que les trois arguments ne servent pas systématiquement. Ce filtre (c'est implicite) n'agira que sur les lignes éditables du dialogue, forçant toute saisie de lettres à s'effectuer en majuscules. La valeur (* itemHit) est indéterminée en entrée, puisque la fonction filtre gère tous les événements (y compris l'événement nul) et pas seulement les clics souris. Il est donc impossible de savoir sur quelle ligne éditable le filtre agit, puisque le Dialog Manager n'offre aucune fonction retournant l'actuelle ligne éditable active, et ne permet pas d'aller lire directement le champ du *Dialogrecord* qui contient cette information ! C'est là un très gros manque (voir l'exemple en fin de chapitre).

Si le filtre retourne TRUE, la fonction **ModalDialog** sera interrompue et renverra la valeur (* itemHit). Donc, l'événement est traité comme un clic souris dans l'item identifié par (* itemHit). L'application doit renseigner cette valeur avant de rendre la main : c'est cette possibilité qui oblige le filtre à utiliser un pointeur sur identifiant.

Remarquons au passage que grâce à cette fonction de filtrage, rien n'empêche qu'une action de l'utilisateur dans un item d'un dialogue provoque l'appel d'une alerte pour signaler une erreur. Dans ce cas et uniquement dans ce cas, la fenêtre de modal dialog n'est plus au premier plan. Mais **ModalDialog** gérera toute seule les événements

d'activation et de mise à jour de fenêtre qui résulteront de la fermeture de l'alerte (tout au moins en ce qui concerne les items associés à la fenêtre : si l'application lui rajoute ses propres textes ou dessins, ceux-ci ne seront pas rafraîchis). On pourrait imaginer également qu'un dialogue appelle un autre dialogue, mais ce style de programmation ne devrait pas avoir cours, par respect pour l'utilisateur.

Pour utiliser son propre filtre et conserver les vertus du filtre standard, ce qui est bien pratique, on appellera **ModalDialog** de la façon suivante :

```
int item;
int monFiltre( );              /* déclaration de la fonction filtre */

do
   {
   item = ModalDialog((long) monFiltre | 0x80000000);
   }
while (item > 2);
```

Dans cet exemple, on boucle sur **ModalDialog** en utilisant un filtre propre et le filtre standard (le bit 31 de l'argument a été forcé à 1). On ne fait aucun traitement particulier sur les items (à part forcer les caractères en majuscules). Cet exemple assume qu'il n'y a que deux boutons simples, dont les identifiants sont 1 (bouton par défaut, par exemple *OK*) et 2 (bouton *Annuler*). On sort de la boucle si l'utilisateur clique dans l'un des deux boutons (obligatoirement actif et non muet). L'application va alors tester lequel des deux a été enfoncé, et si c'est le bouton *OK*, récupérer les valeurs contenues dans chaque item susceptible d'avoir été modifié pour poursuivre le traitement. Une fois les valeurs récupérées, le dialogue pourra être fermé. Remarquons incidemment que si tous les items sauf les boutons simples sont muets, il n'y a même pas besoin de boucle !

Les belles applications pourront utiliser la fonction filtre pour changer le dessin du curseur quand il passe au-dessus d'un texte éditable (voir le chapitre VIII). Ce n'est pas très compliqué, puisqu'on a à chaque instant sa position par l'intermédiaire du champ *where* de l'événement : on n'a qu'à convertir en coordonnées locales et vérifier l'appartenance ou non au rectangle de visualisation, suivant une technique déjà vue plusieurs fois.

ModalDialog traite les événements de la manière suivante :

– elle gère complètement les événements d'activation et de mise à jour de la fenêtre de dialogue, mais en ce qui concerne les items uniquement : si l'application va écrire dans la fenêtre directement, le Dialog Manager ne le sait pas !

– si un clic souris intervient dans une ligne éditable, elle agit comme agirait Line Edit : affichage d'un point d'insertion ou sélection d'une partie ou de la totalité de la ligne, réponse au double-clic pour sélectionner un mot, au triple-clic pour sélectionner la ligne entière, etc. Les événements de type clavier (hormis l'invocation des touches Retour et Entrée) ne sont gérés que dans les items de ligne éditable, sauf spécification contraire due à la présence d'un filtre particulier (gestion de touches de commande, par exemple) ;

– si la souris est pressée dans un contrôle de type bouton, elle appelle **TrackControl**. Si la souris est relâchée à l'intérieur du contrôle et que celui-ci n'est pas muet, elle renvoie son identifiant. Sinon, elle ne fait rien. Il appartient à l'application de fixer la nouvelle valeur du contrôle, ce qui provoquera le dessin correct du ou des boutons (voir le chapitre sur le Control Manager) ;

– si la souris est pressée dans une barre de défilement, elle appelle aussi **TrackControl**, mais en passant l'adresse d'une fonction d'action définie par la barre (voir plus haut) ;

– si la souris est pressée dans n'importe quel autre item actif et non muet, elle retourne l'identifiant de l'item ;

– si la souris est pressée dans un item muet ou dans aucun item, elle ne fait rien. De même si aucun événement ne survient ;

– si la souris est pressée à l'extérieur de la fenêtre de dialogue, elle émet le son numéro 1 (un « bip » unique tel que réglé par le Tableau de Bord si la procédure standard de son est utilisée).

Suppression d'items, désallocation d'un dialogue

Il est possible de rendre invisible un item, nous le verrons plus loin. Mais quand un item ne doit plus être utilisé dans un dialogue, pour quelque raison que ce soit, autant le supprimer purement et simplement de la liste des items. C'est le rôle de la procédure **RemoveItem**, qui efface l'item de l'écran (si nécessaire) et supprime du dialogue passé en premier argument l'item dont l'identifiant correspond au second argument.

Une fois qu'on en a terminé avec le dialogue, si l'utilisateur a cliqué dans un bouton simple (cas d'un modal dialog) ou dans la case de fermeture (cas d'un modeless dialog), deux possibilités se présentent : soit on rend le dialogue invisible, soit on le désalloue.

Si l'application risque de le réutiliser rapidement, il est peut-être préférable de simplement le rendre invisible, avec **HideWindow**. Tout reste en place et occupe de la mémoire, mais il sera instantanément présent au prochain appel (il suffira de le rendre visible et d'activer la fenêtre).

Si le dialogue a peu de chances d'être rappelé, ou si l'application a besoin d'espace mémoire, il est préférable de désallouer toutes les structures liées à ce dialogue. La procédure **CloseDialog** est là pour cela. Une fois appelée, il n'en restera plus grand-chose : tous les items (sauf les icônes et pictures, qui pourraient être des objets utilisés ailleurs) seront effacés de la mémoire, ainsi que les éléments qui pourraient s'y rapporter. La fenêtre elle-même est détruite, comme le fait la procédure **CloseWindow**, à laquelle **CloseDialog** fait d'ailleurs appel. Un seul argument pour cette procédure : le pointeur sur le dialogue à détruire. Si l'application désire de nouveau l'utiliser, elle n'aura plus qu'à le recréer de toutes pièces.

Création et gestion d'alertes

Une alerte est d'un maniement beaucoup plus simple qu'un dialogue, car elle se gère toute seule : elle fonctionne comme un modal dialog, sauf que dès que l'utilisateur a provoqué un événement dans un item non muet, elle rend la main à l'application en désallouant tout l'espace mémoire qu'elle occupait. Ainsi, avec une seule fonction, **Alert**, on obtient les actions suivantes : création de l'alerte (fenêtre et items), gestion de l'alerte, destruction de l'alerte. Facile, non ?

Cette fonction réclame deux arguments : un pointeur sur un modèle prédéfini de création d'alerte et un pointeur sur une fonction de filtre. Le modèle prédéfini va servir à appeler (de manière transparente) la procédure **GetNewModalDialog** (pour créer l'alerte), donc on va retrouver avec quelques modifications et ajouts la plupart des éléments qui sont nécessaires à son utilisation. La gestion de l'alerte se faisant par un appel unique (et non dans une boucle) et transparent à la fonction **ModalDialog**, la fonction de filtre en sera l'argument, et la valeur zéro-long pourra être passée pour utiliser le filtre par défaut.

Le modèle d'alerte pourra être défini de la manière suivante :

```
struct _AlertTemplate {
    Rect    BoundsRect;        /* rectangle contenu de la fenêtre */
    int     AlertID;           /* identifiant de l'alerte */
    char    stage1;            /* définition du niveau 1 de l'alerte */
    char    stage2;            /* définition du niveau 2 de l'alerte */
    char    stage3;            /* définition du niveau 3 de l'alerte */
    char    stage4;            /* définition du niveau 4 de l'alerte */
    Pointer Items[ ];          /* liste variable de pointeurs sur item, terminée par 0L */
    };
#define AlertTemplate struct _AlertTemplate
```

• On constate quelques différences par rapport au *DialogTemplate* :
– l'absence du champ *Visible* (la notion de visibilité est gérée pour chaque niveau d'alerte) ;
– l'absence du champ *wRefCon* (l'application n'a absolument pas l'occasion de manipuler la fenêtre d'alerte, donc pas de valeur libre utilisable) ;
– la présence de champs propres à l'alerte.

• Les autres champs sont identiques (la liste des pointeurs doit toujours être terminée par zéro-long). La remarque faite à propos de la définition des dialogues s'applique aussi à la définition des alertes. Si une application doit utiliser plusieurs modèles d'alerte, elle commencera par initialiser celui qui possède le plus d'items, afin que suffisamment d'espace soit réservé pour cette structure de taille variable.

• L'identifiant de l'alerte est un nombre unique pour l'ensemble des alertes d'une application : si l'application définit 10 *AlertTemplate*, chacun aura un identifiant différent. Grâce au paramétrage des textes statiques, un modèle d'alerte peut servir à l'ensemble des messages d'erreurs d'une application !

• Les quatre niveaux d'alerte seront définis chacun sur un simple octet, les bits ayant la signification suivante :
– bits 0 à 1 : numéro du son à émettre au niveau donné (compris entre 0 et 3) ;
– bits 2 à 5 : inutilisés ;
– bit 6 : identifiant du bouton par défaut, moins 1 : seuls les boutons d'identifiant 1 (bit à 0) et 2 (bit à 1) sont donc possibles ;
– bit 7 : à 1 si la fenêtre doit être affichée à ce niveau, 0 pour avoir simplement le son.

La fonction **Alert** retourne l'identifiant de l'item non muet dans lequel l'utilisateur a cliqué. Rappelons encore une fois comment fonctionne **Alert** (les appels sont implicites) :

1. Création d'une fenêtre par appel à **GetNewModalDialog** ;
2. Comparaison avec la dernière alerte appelée pour déterminer son niveau ;
3. Action en fonction du niveau : procédure sonore puis procédure visuelle ;

Si procédure visuelle (bit 7 positionné) :

4. Affichage de la fenêtre et appel de la fonction **ModalDialog** ;
5. Dès que l'utilisateur clique dans un item non muet, **ModalDialog** retourne son identifiant ;
6. Appel de **CloseDialog** pour purger items et fenêtre de la mémoire ;
7. **Alert** retourne l'identifiant de l'item.

Dans cette succession d'actions, on constate que rien n'empêche l'alerte de posséder les mêmes items qu'un dialogue. Cependant, la destruction des items avant que l'application ne retrouve le contrôle des opérations interdit la présence d'items pouvant prendre plusieurs valeurs, puisqu'on n'a aucun moyen d'aller lire la valeur finale. Par ailleurs, il serait d'un effet assez désastreux que l'utilisateur sorte d'une alerte en cliquant sur un bouton radio ! On exclura donc des alertes tout contrôle autre que les simples boutons. Et une ligne éditable muette ? Elle ne générait pas, mais l'application serait incapable de lire son contenu avant sa destruction ! Donc, règle absolue : une alerte ne contient que des boutons et du texte statique, éventuellement paramétré.

Voici un exemple tout simple d'alerte. Elle affiche le message « Vous allez détruire tout votre répertoire » et deux boutons. « D'accord » et « Pas d'accord ». C'est évidemment le bouton « Pas d'accord » qui est pris par défaut. Pas de fioriture particulière : deux « bips » et affichage de la fenêtre quel que soit le niveau (qui sont tous égaux, valeur binaire 11000010, soit C2 en hexadécimal).

```
ItemTemplate  bouton1 = { 1, {50, 10, 70, 110}, ButtonItem, "\10D'accord", 0, 2, 0L };
ItemTemplate  bouton2 = { 2, {50, 130, 70, 240}, ButtonItem, "\14Pas d'accord", 0, 2, 0L };
char str[ ] = "\51Vous allez détruire\rtout votre répertoire";      /* sur 2 lignes! */
ItemTemplate  textalerte = { 3, {10, 10, 40, 230}, StatText + ItemDisable, str, 0, 0, 0L };
AlertTemplate  attention = {
    {40, 30, 120, 290},          /* rectangle contenu de la fenêtre */
    1,                           /* identifiant de l'alerte */
    0xC2, 0xC2, 0xC2, 0xC2,      /* les quatre niveaux d'alerte */
    { &bouton1,                  /* adresse du premier item */
    &bouton2,                    /* adresse du deuxième item */
    &textalerte,                 /* adresse du troisième item */
    0L }                         /* il n'y a plus d'item */
    };
int  item;

item = Alert(&attention, 0L);          /* tout se fait avec cette instruction unique */
if (item == 1) destruction_du_repertoire( );
```


Derrière l'appel à la fonction **Alert**, un seul test est suffisant : si *item* vaut 1, l'utilisateur a cliqué sur le bouton « D'accord » et on exécute la commande. Sinon, on n'a rien à faire puisqu'il n'est plus d'accord ! La valeur retournée ne pourra jamais être égale à 3, puisque le troisième item a été déclaré muet.

La fonction **Alert** possède trois variantes, de fonctionnement absolument identique. La différence tient dans le fait qu'elles dessinent en plus des items déclarés par l'application une icône prédéfinie dans le coin haut gauche de la fenêtre. Ces fonctions s'appellent **StopAlert**, **NoteAlert** et **CautionAlert**.

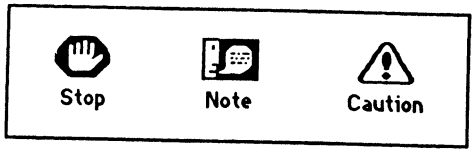


Figure IX.4. Les icônes prédéfinies.

Une quatrième variante est prévue : **TalkAlert**, qui présentera de plus la particularité de *parler :* elle énoncera d'une voix synthétique les textes statiques de l'alerte.

Citons enfin les deux routines qui permettent de contrôler le niveau d'une alerte. Nous avons dit que le Dialog Manager stockait dans un coin l'identifiant de la dernière alerte appelée (ainsi que son niveau d'appel). A l'appel d'une nouvelle alerte, si l'identifiant est identique, il s'agit d'une alerte consécutive, et le niveau est incrémenté, sinon le niveau est remis à zéro. Ceci est automatique. Mais supposons que l'application utilise un seul *AlertTemplate* pour gérer plusieurs alertes différentes (grâce aux textes statiques paramétrables). Elle peut avoir besoin de gérer elle-même le niveau d'une telle alerte.

La procédure **ResetAlertStage**, sans argument, fera en sorte que la prochaine alerte sera traitée à son premier niveau. La fonction **GetAlertStage**, sans argument non plus, retournera le niveau actuellement stocké par le Dialog Manager (entier compris entre 0 et 3, 0 signifiant premier niveau et 3 quatrième niveau).

Utilisation d'un modeless dialog

La création d'un modeless dialog se fera par la fonction **NewModelessDialog**, dont les six arguments vont permettre de créer une fenêtre classique, avec les contrôles habituels liés aux fenêtres :

– un pointeur sur le rectangle définissant la région contenu de la fenêtre à la création (*PortRect* du grafport associé) ;

– un pointeur sur le titre de la fenêtre de dialogue (chaîne de type Pascal) ;

– un pointeur sur la fenêtre derrière laquelle sera créé le dialogue (passer la valeur – 1L pour que le dialogue soit créé au premier plan) ;

– un entier décrivant la région contour de la fenêtre (donc les contrôles associés, champ *wFrame*) ;

– un entier long d'utilisation libre pour l'application (champ *wRefCon*) ;

– un pointeur sur le rectangle définissant la région contenu de la fenêtre après un zoom, nécessaire si la case de zoom est présente (passer zéro-long sinon).

Voir le chapitre consacré au Window Manager pour avoir plus de détails sur ces arguments, notamment la position des bits permettant la définition des contrôles dans sa région contour.

La fonction **NewModelessDialog** retourne un pointeur sur dialogue, qui servira de manière identique au pointeur retourné par **NewModalDialog**. Une fois la fenêtre créée, on lui ajoute des items, grâce aux procédures **GetNewDItem** ou **NewDItem** vues plus haut.

Une fenêtre de modeless dialog est une fenêtre classique, qui peut passer derrière d'autres fenêtres. Il y aura donc des événements d'activation et de mise à jour à gérer, mais heureusement, c'est le Dialog Manager qui va s'en occuper. Quand une application manipule des fenêtres de modeless dialog, elle doit appeler la fonction **IsDialogEvent** juste après avoir appelé **GetNextEvent** ou **TaskMaster**. En argument, le pointeur sur l'événement qui vient d'être alimenté par ces routines. Cette fonction va déterminer si l'événement doit être traité par le Dialog Manager comme intervenant dans une partie du dialogue : événement d'activation ou de mise à jour de la fenêtre de dialogue, clic-souris ou touche enfoncée si la fenêtre de dialogue est active. Dans ce cas, **IsDialogEvent** retourne TRUE. Si l'événement n'a rien à voir avec le dialogue, la valeur retournée est FALSE.

Si la réponse est FALSE (donc égale à zéro), l'application gère son événement de manière normale, tel que cela a été vu dans la boucle d'événement. Si la réponse est TRUE (donc différente de zéro), l'application appellera la fonction **DialogSelect** (après avoir éventuellement exécuté quelques instructions propres, équivalentes par exemple, à ce qu'aurait fait la fonction filtre dans un modal dialog).

DialogSelect réclame trois arguments, trois pointeurs :

– un pointeur sur l'événement auquel on répond (identique à celui de **IsDialogEvent**) ;

– l'adresse de la variable dans laquelle **DialogSelect** va stocker le pointeur sur dialogue concerné ;

– l'adresse de la variable dans laquelle **DialogSelect** va stocker le pointeur sur l'item concerné.

Si l'événement concerne un item non muet dans un dialogue actif (la fenêtre de modeless dialog était active quand l'événement est survenu), **DialogSelect** retourne TRUE après avoir alimenté les variables donnant le pointeur sur dialogue et l'identifiant de l'item concernés, pour permettre à l'application de gérer cet événement. Dans tous les autres cas, **DialogSelect** retourne FALSE et l'application n'a rien à faire (un item muet a été utilisé, un événement d'activation/désactivation ou de mise à jour est intervenu, une touche a été enfoncée alors qu'il n'y a aucune ligne éditable dans le dialogue, etc.).

Quand le dialogue contient des lignes éditables, les fonctions **IsDialogEvent** et **DialogSelect** doivent toujours être appelées après **GetNextEvent** ou **TaskMaster**, même si ces fonctions retournent l'événement nul, afin de permettre au point d'insertion de clignoter si le dialogue est actif (ce sont ces fonctions qui appellent la procédure de Line Edit **LEIdle**).

Quatre procédures peuvent être utilisées quand le dialogue contient des lignes éditables, pour répondre aux commandes d'édition de texte *Couper, Copier, Coller* et *Effacer*. Ce sont **DlgCut**, **DlgCopy**, **DlgPaste** et **DlgDelete**, dont le seul argument est un pointeur sur le dialogue. Ces procédures agissent sur la ligne éditable sélectionnée, en appelant les procédures de Line Edit **LECut**, **LECopy**, **LEPaste et LEDelete**. C'est donc le presse-papiers privé de Line Edit qui est utilisé.

Pour désallouer la fenêtre de dialogue et ses items, on procédera comme pour un modal dialog.

La gestion d'un modeless dialog aura donc à peu près cette allure :

```
TaskRec   tache;          /* un événement */
Pointer   dlg;            /* pointeur sur le dialogue éventuellement utilisé */
int       it;             /* l'item éventuellement invoqué */

if (!GetNextEvent(EveryEvent, &tache)) continue;
if (IsDialogEvent(&tache))          /* si l'événement concerne un modeless dialog... */
   {
   if(DialogSelect(&tache, &dlg, &item)) /* ...et si le Dialog Manager ne peut pas le gérer... */
      ...                           /* ... on répond à l'item it du dialogue dlg renvoyé */
   }
else ...          /* réponse classique à l'événement, qui ne concerne pas un dialogue */
```

Gestion des items

Les routines que nous allons voir dans ce paragraphe s'appliquent à tous les items, qu'ils appartiennent à un dialogue ou une alerte (quand c'est possible). Elles permettent de connaître ou de modifier les caractéristiques d'un item dont l'identifiant et la fenêtre d'appartenance sont donnés. Seul l'identifiant ne peut pas être modifié. Dans les exemples qui suivent, nous ne répéterons pas les déclarations suivantes :

```
Pointer dlg;              /* pointeur sur dialogue */
int     item;             /* identifiant de l'item considéré */
```

• On peut connaître le type d'un item grâce à la fonction **GetItemType**, qui retourne l'une des constantes prédéfinies vues plus haut. La procédure **SetItemType** permet de changer le type de l'item, son utilisation est à proscrire absolument.

```
int type;

type = GetItemType(dlg, item);      /* retourne le type de l'item */
```

• On peut connaître et changer le rectangle d'affichage d'un item, grâce aux procédures **GetItemBox** et **SetItemBox**. Dans les deux cas, le rectangle sera manipulé par l'intermédiaire de son adresse, en troisième argument. Sauf effets spéciaux, on évitera de modifier la taille ou la localisation d'un item.

```
Rect    r;                  /* un rectangle */

GetItemBox(dlg, item, &r);  /* on récupère le rectangle à l'adresse indiquée */
InsetRect(&r, -10, 0);      /* on l'élargit de 20 points */
SetItemBox(dlg, item, &r);  /* l'item a une nouvelle taille */
```

• On peut connaître la valeur actuelle d'un item grâce à la fonction **GetItemValue** et changer la valeur d'un item grâce à la procédure **SetItemValue**. Pour les contrôles standard, la valeur de l'item est vraiment la valeur du contrôle. Pour les types propres au Dialog Manager, les valeurs peuvent avoir un sens particulier ou être libres d'utilisation. Ce sont ces routines qu'il faudra utiliser pour les gérer.

Exemple de gestion des cases à cocher et des boutons radio, conforme à ce que nous avons déjà vu dans le chapitre consacré au Control Manager.

```
int  val, type;

item = ModalDialog(0L);                 /* on est dans un modal dialog */
type = GetItemType(dlg, item);          /* quel est le type de l'item choisi? */
switch(type)
    {
    case CheckItem :                    /* est-ce une case à cocher? */
    val = GetItemValue(dlg, item);
    SetItemValue(1 – val, dlg, item);   /* oui, on change sa valeur */
          /* ou bien: SetItemValue(!val, dlg, item); */
    ...
    break;

    case RadioItem :
    SetItemValue(1, dlg, item);         /* oui, on change sa valeur */
    ...
    break;
    }
    ...       /* et on boucle! */
```

Notons un point important : la procédure **SetItemValue** redessine l'item dès que la valeur est fixée, donc les contrôles standard présenteront leur aspect tel qu'il découle de leur valeur. Si un utilisateur clique dans une case à cocher et que la valeur de cette case n'est pas modifiée, il n'y aura rien de visible à l'écran ! De même, la modification de la valeur d'un bouton radio provoque le redessin de toute la famille à laquelle il appartient.

• Pour rendre visible ou invisible un item, on utilisera les procédures **ShowDItem** et **HideDItem**. Quand un item est rendu invisible, il reste dans la liste des items mais il n'apparaît plus dans la fenêtre et ne peut être atteint par l'utilisateur. La procédure **HideDItem** ne fait rien si l'item considéré est déjà invisible. La procédure **ShowDItem** ne fait rien si l'item considéré est déjà visible.

```
HideDItem(dlg, item);           /* on rend l'item invisible */
...
ShowDItem(dlg, item);           /* on rend l'item visible */
```

Rappelons qu'un item peut être détruit par **RemoveItem** plutôt que rendu invisible s'il n'a aucune chance de réapparaître avant la fin du dialogue.

• Pour rendre muet un item, on pourra utiliser la procédure **DisableDItem**. Il ne se passe rien si l'item est déjà muet. Pour rendre parlant un item, on pourra utiliser la procédure **EnableDItem**. Il ne se passera rien si l'item est déjà parlant. Dans les deux cas, rien ne vient modifier l'aspect visuel de l'item.

```
DisableDItem(dlg, item);        /* on rend l'item muet */
...
EnableDItem(dlg, item);         /* on rend l'item parlant */
```

Sauf cas particulier, ces deux procédures n'auront pas à être appelées. En effet, un item est créé muet ou non, et il n'y a pas beaucoup de raisons pour que cet état doivent changer subitement !

• Aucune routine du Dialog Manager ne permet de désactiver et réactiver un contrôle standard dans une fenêtre de dialogue. Il faudra donc utiliser la routine **HiliteControl** du Control Manager en cas de besoin. Mais cette routine réclame en argument un handle sur le contrôle visé ! Pour connaître le handle d'un contrôle appartenant à un dialogue, on utilisera la fonction **GetControlItem**.


```
Handle hdl;                     /* handle sur contrôle */

hdl = GetControlItem(dlg, item);  /* on récupère le handle */
HiliteControl(255, hdl);        /* l'item est désactivé */
...
HiliteControl(0, hdl);          /* l'item est réactivé */
```

Cet exemple mis à part, on se gardera bien de faire n'importe quoi en utilisant directement le Control Manager. En effet le Dialog Manager utilise certains champs de manière non standard, et on pourrait perdre des informations essentielles en manipulant un peu trop directement certains items. Il est toutefois permis (puisqu'il n'y a pas d'autre solution) d'aller manipuler la couleur des contrôles directement.

• Nous avons dit que les textes statiques pouvaient être paramétrés en leur incluant les caractères spéciaux ^0, ^1, ^2 et ^3. La procédure **ParamText** permet de donner les chaînes de caractères qui se substitueront aux caractères spéciaux lors de l'affichage des textes statiques. Elle admet quatre arguments : quatre pointeurs sur chaîne de caractères type Pascal donnant les substitutions. Ces chaînes seront éventuellement vides.

```
char fichstr[17];               /* réserve de la place pour le nom de fichier */
char repstr[65];                /* réserve de la place pour le chemin d'accès */
                                /* le texte statique à afficher (chaîne Pascal) */
char statstr[ ] = "\66Le fichier\r<^0>\rest introuvable\rdans le répertoire\r^1\r";

/* l'utilisateur a demandé un nom de fichier, ce nom est écrit à l'adresse fichstr */
/* l'application vérifie l'existence du fichier dans le répertoire en cours (appel ProDOS) */
/* elle ne le trouve pas, elle écrit le pathname à l'adresse repstr, et... */
ParamText(fichstr, repstr, 0L, 0L);          /* paramétrage du texte statique */
```

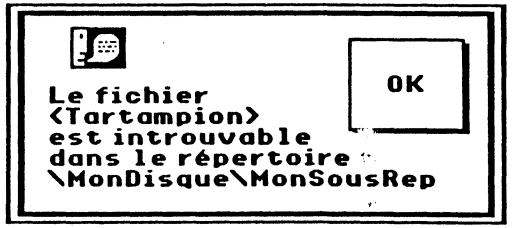


Figure IX.5. Ce que cela pourrait donner.

Rappelons que les caractères de retour à la ligne sont permis dans les textes statiques. Ils correspondent au code ASCII 13 en décimal, encore codé \r en langage C. Pour calculer la longueur de la chaîne contenant le texte paramétré, on compte chaque caractère pour 1, la combinaison \r pour 1, et les combinaisons de type ^0 pour 2. Il est conseillé de terminer la chaîne par un caractère de retour à la ligne, surtout quand d'autres retours à la ligne figurent déjà à l'intérieur.

Remarque Le Dialog Manager conserve la trace des quatre adresses dans un coin de sa mémoire. Si la procédure **ParamText** n'est pas rappelée lors d'un affichage ultérieur d'un texte statique qui fait appel à des substitutions, les anciennes adresses seront utilisées. Si entre-temps elles ont servi à stocker autre chose (si par exemple les chaînes ont été déclarées variables locales), des caractères plus que bizarres risquent d'apparaître dans le texte statique concerné ! On peut passer zéro-long dans l'un des arguments, si l'adresse correspondante sur laquelle pointait précédemment **ParamText** doit être conservée ou si le paramètre n'est pas utilisé.

• Intéressons-nous maintenant aux items de type ligne éditable. Si nous demandons la valeur d'un tel champ, nous obtenons le nombre précise au moment de la création de l'item, qui désigne le nombre maximal de caractères que l'utilisateur pourra saisir

dans cet item, et en aucun cas les caractères eux-mêmes. Pour obtenir cette chaîne de caractères (style Pascal), il faudra s'allouer la mémoire nécessaire pour la recevoir, et appeler la procédure **GetIText**. Pour allouer la mémoire nécessaire, le plus simple sera de déclarer une chaîne de longueur m + 1 (si m désigne le nombre maximal de caractères autorisé), le caractère supplémentaire servant à accueillir le nombre exact de caractères en début de chaîne Pascal.

```
char editstr[17];          /* on réserve 17 octets pour la chaîne, assume m=16 */

GetIText(dlg, item, editstr);          /* la chaîne est recopiée à l'adresse editstr */
```

Éventuellement, cette procédure peut servir à retrouver le contenu d'un texte statique, mais c'est nettement moins utile !

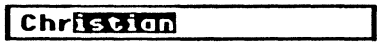
Du point de vue de l'application, on peut également mettre du texte dans une ligne éditable ou un texte statique, pourquoi pas, si la possibilité du paramétrage ne suffit pas (c'est intéressant, par exemple, pour afficher en clair la valeur prise par une barre de défilement).

Supposons que l'application autorise l'utilisateur à taper les premières lettres d'un texte, et complète elle-même le texte par défaut en tenant compte de ces premières lettres (un moyen élégant et pas trop coûteux pour proposer un choix, l'autre moyen élégant étant de proposer une liste dans une fenêtre avec barre de défilement et choix par double-clic). Pour afficher ce texte par défaut, la procédure **SetIText** sera utilisée.

```
char editstr[17];          /* on réserve 17 octets pour la chaîne, assume m=16 */

GetIText(dlg, item, editstr);          /* les premières lettres sont récupérées */
...          /* l'application calcule sa chaîne par défaut et la place à l'adresse editstr */
SetIText(dlg, item, editstr);          /* la chaîne est affichée */
```

Continuons sur l'exemple précédent. L'utilisateur a tapé quelques lettres, l'application a complété son texte. Mais si le choix n'était pas unique ? Il serait agréable à l'utilisateur de n'avoir qu'à taper une lettre supplémentaire pour faire un nouveau choix. C'est possible si l'application lui rend la main après avoir sélectionné la partie du texte qu'elle a complété elle-même, laissant normal ce que lui a saisi. La procédure **SelIText** permet cela. Si nous poursuivons notre exemple, et en admettant que l'utilisateur a saisi trois caractères, on écrira :

```
SelIText(dlg, item, 3, 256);  /* tous les caractères après le troisième seront inversés */
```



Le troisième argument de **SelIText** correspond au début de la sélection (les positions débutent à la valeur zéro et tombent entre les caractères, donc la position 3 se situe entre les troisième et quatrième lettres). Le quatrième argument correspond à la fin de la sélection. Si la valeur de fin est plus grande que le nombre de caractères de la chaîne, peu importe : la sélection ira jusqu'au bout du texte, sans plus. C'est le cas dans l'exemple, une ligne éditable ne pouvant avoir plus de 255 caractères. Si les deux arguments sont égaux, il n'y a pas de portion de texte sélectionnée, mais le point d'insertion clignotant est placé à la position désignée. Et si par hasard aucun texte ne se trouvait dans la ligne éditable, le point d'insertion serait encore affiché, sans plus, au début du champ.

• On a dit et répété que si un bouton simple portait l'identifiant 1, il était considéré dans un dialogue comme le bouton par défaut. Si pour une raison précise, l'application doit changer le bouton par défaut (par exemple un même dialogue pouvant être utilisé dans deux contextes différents qui nécessitent de permuter les boutons par défaut), elle le fera avec la procédure **SetDefButton**. Et pour connaître l'identifiant du bouton par défaut, on appelle la fonction **GetDefButton**. Cette fonction retourne zéro s'il n'y a pas de bouton par défaut.


```
item = GetDefButton(dlg);          /* item sera vraisemblablement égal à 1 */
SetDefButton(item+1, dlg);         /* on change le bouton par défaut */
item = GetDefButton(dlg);          /* item vaut maintenant 2 */
```

Attention Le bouton par défaut doit être un bouton « sûr ». Si l'application passe un message du style : « Voulez-vous sauvegarder vos données avant de quitter ? » et qu'il y a trois boutons possibles : OUI, NON et ANNULER, il ne faudra surtout pas déclarer le bouton NON par défaut ! Le bouton OUI est un choix meilleur, le bouton ANNULER est sans doute le meilleur des choix, puisqu'il oblige l'utilisateur à prendre une décision : dans 95 % des cas, ce sera OUI, dans 5 % des cas, ce sera NON, mais il vaut peut-être mieux ne pas lui forcer la main sur ces 5 % là. Remarquons au passage la clarté du message : la question ne doit occasionner aucune ambiguïté !

• Supposons qu'on veuille connaître tous les identifiants d'une liste d'items (peu importe qu'ils soient muets, inactifs ou invisibles, ils font partie de la liste), on a pour cela deux fonctions, **GetFirstItem** et **GetNextItem**. Gageons que peu nombreuses seront les applications qui utiliseront ces fonctions ! Si un dialogue contient les items 8, 3 et 6, dans cet ordre à la création, on pourra avoir l'enchaînement suivant :

```
item = GetFirstItem(dlg);          /* item reçoit la valeur 8 */
item = GetNextItem(dlg, item);     /* item prend la valeur 3 */
item = GetNextItem(dlg, item);     /* item prend la valeur 6 */
item = GetNextItem(dlg, item);     /* item prend la valeur 0 */
```

S'il n'y a aucun item dans la liste, **GetFirstItem** retourne zéro. Si l'item spécifié en deuxième argument dans **GetNextItem** est le dernier de la liste, cette fonction retourne également zéro.

• Pour savoir si un point, donné en coordonnées globales, appartient à un item d'un dialogue particulier, on peut se servir de la fonction **FindDItem**, qui retourne l'identifiant de l'item ou zéro si le point n'est inclus dans aucun item ou se trouve à l'extérieur de la fenêtre de dialogue. Cette fonction n'a pas de grandes raisons d'être employée par une application.

Exemple complet : un modal dialog

L'exemple suivant est destiné à manipuler un certain nombre de routines offertes par le Dialog Manager. Il n'a pas grande application pratique, puisqu'il est limité à la saisie de neuf fiches, mais il contient les bases de ce que pourrait être un écran de saisie d'un institut de sondage. On va entrer un code (compris entre 1 et 9) dans une ligne éditable, ce code correspondant à un enregistrement dans une structure de données qui mémorise un identifiant, le statut matrimonial, le sexe, l'âge, le nombre d'enfants (garçons et filles).

L'enregistrement 0 sert d'enregistrement par défaut, on n'a donc pas le droit de le modifier. Si l'utilisateur entre le code 0, une alerte est affichée. S'il entre un autre code, les données associées à cet enregistrement sont affichées dans les différents items : ligne éditable pour l'identifiant, famille de boutons radio pour le statut, autre famille de boutons radio pour le sexe, barre de défilement pour l'âge. Une case à cocher est également présente : si la personne a des enfants, elle est cochée et apparaissent en dessous deux lignes éditables (nombre de garçons, nombre de filles). Si la personne n'a pas d'enfant, la case n'est pas cochée, et les lignes garçons et filles sont invisibles.

L'utilisateur pourra modifier les données associées à un code, et les mettre en mémoire en cliquant sur le bouton Ajouter (bouton par défaut) Dans ce cas, le champ identifiant ne devra pas être vide, sinon une alerte est affichée. L'utilisateur pourra également supprimer l'enregistrement, par le bouton Supprimer. Enfin, il pourra visualiser ce qu'il a saisi, en cliquant sur le bouton Quitter (ou en faisant Pomme-Q,

équivalent-clavier défini pour ce bouton dans une fonction filtre). Dans ce cas est affiché un dialogue secondaire, avec deux boutons, l'un permettant de revenir à la saisie, l'autre de quitter définitivement l'application (aucun bouton par défaut). L'exemple montre donc comment on peut passer d'un dialogue à un autre, puis revenir.

Tous les événements clavier du dialogue principal sont interceptés par une fonction filtre, qui non seulement définit l'équivalent-clavier Pomme-O, mais aussi empêche la saisie de tout caractère différent des chiffres. Quand l'utilisateur tape un caractère non valide, une alerte est affichée. Le filtre standard est également utilisé, ce qui permet le copier-coller entre lignes éditables, et le caractère Retour en équivalent du bouton par défaut.

Dans la barre des menus est affiché en permanence l'identifiant du dernier item renvoyé par la fonction **ModalDialog**, ainsi que son type.

Cet exemple a été mis au point avec beaucoup de mal : il semble bien que dans sa version 1.01, le Dialog Manager soit encore fâché avec les lignes éditables et même les textes statiques. Cela est évident quand on introduit des couleurs dans les contrôles standard. Les textes affichés ensuite deviennent multicolores, alors qu'on ne leur a rien demandé ! Plusieurs autres bogues, plus graves, ont été mis à jour, mais ces défauts de jeunesse seront sans doute réparés quand vous lirez ces lignes.

Remarquons qu'un dialogue avec des lignes éditables provoque souvent le besoin de convertir des chaînes de caractères en nombres et réciproquement. Nous avons plusieurs fois joué sur le fait que le Dialog Manager manipule des chaînes de type Pascal (où le premier caractère donne la longueur de la chaîne). Nous avons également joué sur les caractères comme le langage C nous le permet : puisque 'a' désigne le code ASCII du caractère a minuscule, 'a' + 1 désigne le code ASCII suivant, donc le caractère b minuscule, et ainsi de suite. La condition car<'0' || car>'9' assure que le caractère contenu dans la variable car n'est pas un chiffre.

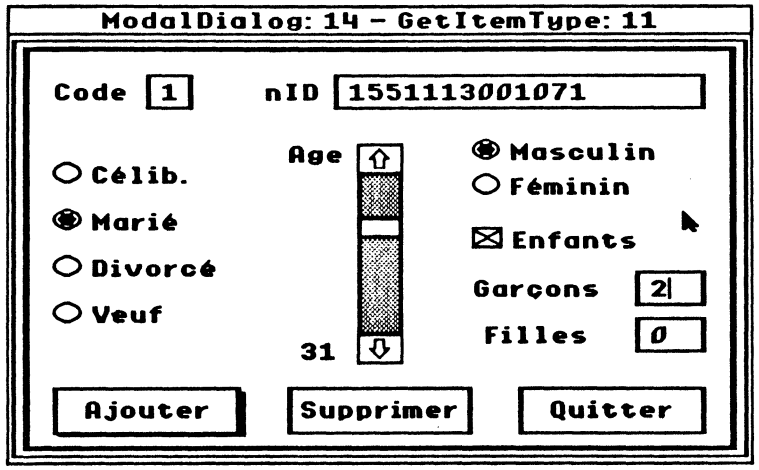


Figure IX.7. L'écran de l'exemple.

```
#include <tools.h>                  /* définition des termes en gras */
#include <entete.h>                 /* définition des termes en italique */

#define mode 0                      /* 0 si mode 320, 1 si mode 640 */

int     Defilement( );              /* fonction d'action pour une barre de défilement */
int     monFiltre( );               /* filtre pour un dialogue */
```



```
                        /* couleurs pour les boutons, la case à cocher, les boutons radio et la barre */
int     colb[ ] = {0x20, 0x90, 0x70, 0x9D, 0x7A, 0x00, 0x00};
int     colc[ ] = {0x00, 0x92, 0x97, 0xF0};
int     colr1[ ] = {0x00, 0x92, 0x97, 0xF0};
int     colr2[ ] = {0x00, 0x82, 0x83, 0xF0};
int     cols[ ] = {0x20, 0x97, 0xA2, 0xFF, 0xCF, 0xFF, 0x1DF, 0xFF};

                                                    /* définition du dialogue principal */
ItemTemplate    bEnreg={1, {145,10,165,90}, ButtonItem, "\7Ajouter", 0, 2, colb};
ItemTemplate    bSuppr={2, {145,110,165,190}, ButtonItem, "\11Supprimer", 0, 2, colb};
ItemTemplate    bQuit={3, {145,210,165,290}, ButtonItem, "\7Quitter", 0, 2, colb};
ItemTemplate    stCode={4, {13,10,28,50}, StatText + ItemDisable, "\4Code", 0, 0, 0L};
ItemTemplate    elCode={5, {10,50,25,70}, EditLine, "", 1, 0, 0L};
ItemTemplate    stnID={6, {13,100,28,130}, StatText + ItemDisable, "\3nID", 0, 0, 0L};
ItemTemplate    elnID={7, {10,130,25,290}, EditLine + ItemDisable, "", 13, 0, 0L};
ItemTemplate    r11={8, {45,10,60,90}, RadioItem, "\6Célib.", 1, 1, colr1};
ItemTemplate    r12={9, {65,10,80,90}, RadioItem, "\5Marié", 0, 1, colr1};
ItemTemplate    r13={10, {85,10,100,90}, RadioItem, "\7Divorcé", 0, 1, colr1};
ItemTemplate    r14={11, {105,10,120,90}, RadioItem, "\4Veuf", 0, 1, colr1};
ItemTemplate    r21={12, {35,190,50,290}, RadioItem, "\10Masculin", 1, 2, colr2};
ItemTemplate    r22={13, {50,190,65,290}, RadioItem, "\7Féminin", 0, 2, colr2};
ItemTemplate    cEnf={14, {73,190,88,290}, CheckItem, "\7Enfants", 0, 0, colc};
ItemTemplate    stGar={15, {98,190,113,250}, StatText + ItemDisable,
                                                    "\7Garçons", 0, 0x80, 0L};
ItemTemplate    elGar={16, {95,260,110,290}, EditLine + ItemDisable, "", 1, 0, 0L};
ItemTemplate    stFil={17, {118,195,133,250}, StatText + ItemDisable, "\6Filles", 0, 0x80, 0L};
ItemTemplate    elFil={18, {115,260,130,290}, EditLine + ItemDisable, "", 1, 0, 0L};
ItemTemplate    stAge1={19, {40,110,50,135}, StatText + ItemDisable, "\3Age", 0, 0, 0L};
ItemTemplate    sbAge={20, {40,140,135,160}, ScrollBarItem + ItemDisable,
                                                    Defilement, 19, 3, cols};
ItemTemplate    stAge2={21, {125,115,140,135}, StatText + ItemDisable, "\00220", 0, 0, 0L};

DialogTemplate  monDial1={ {20,10,190,310}, TRUE, 0L,
                        {&bEnreg, &bSuppr, &bQuit, &stCode, &elCode,
                        &stnID, &elnID, &r11, &r12, &r13, &r14, &r21, &r22,
                        &cEnf, &stGar, &stFil, &stAge1, &sbAge, &stAge2, 0L}};
                                                    /* définition d'une alerte avec texte paramétrable */
ItemTemplate    bOK={1, {10,140,50,190}, ButtonItem, "\2OK", 0, 2, 0L};
ItemTemplate    stMes1={2, {45,10,60,190}, StatText + ItemDisable, "\7ERREUR:", 0, 0, 0L};
ItemTemplate    stMes2={3, {60,10,80,190}, StatText + ItemDisable, "\3^0\r", 0, 0, 0L};

AlertTemplate   monAlerte={ {40,60,125,260}, 1, 0x80, 0x80, 0x80, 0x80,
                                                    {&bOK, &stMes1, &stMes2, 0L}};

                                                    /* définition du dialogue secondaire */
ItemTemplate    bRev={2, {145,10,165,90}, ButtonItem, "\7Revenir", 0, 2, 0L};
ItemTemplate    bOut={3, {145,210,165,290}, ButtonItem, "\7Quitter", 0, 2, 0L};

DialogTemplate  monDial2={ {20,10,190,310}, TRUE, 0L,
                        {&bRev, &bOut, 0L}};


Pointer     dlg;                    /* pointeur sur le dialogue principal */
int         ind;                    /* indice courant dans notre tableau de données */

struct Sondage {                    /* la structure manipulée par notre programme */
        int     bon;                /* état de l'enregistrement (TRUE ou FALSE) */
        char    nID[15];            /* l'identifiant de l'individu (chaîne type Pascal) */
        int     statut;             /* son statut matrimonial */
        int     sexe;               /* son sexe */
        int     age;                /* son age */
        int     nbG;                /* nombre de garçons */
        int     nbF;                /* nombre de filles */
```

```
    } fiche[10];            /* l'indice ne peut varier que de 0 à 9 */

                /***** PROGRAMME PRINCIPAL *****/

main( )
{
int     myID;                       /* identifiant de l'application */
int     i;

    myID = debut_appl(mode);            /* initialisations */
    FlushEvents(EveryEvent, 0);         /* ménage dans la file d'événements */
    dlg = GetNewModalDialog(&monDial1); /* on ouvre le dialogue */
    SetPort(GetMenuMgrPort( ));         /* on va écrire dans la barre des menus... */
    SetTextMode(0);                     /* ...en mode Copy */

    for (i=0; i<10; ++i)                /* initialisation de nos données */
        {
        fiche[i].bon = FALSE ;
        fiche[i].nID[0] = 0;
        fiche[i].statut = 0;
        fiche[i].sexe = 0;
        fiche[i].age = 20;
        fiche[i].nbG = 0;
        fiche[i].nbF = 0;
        }

    do gereDialogue( );                 /* gestion du dialogue... */
    while (afficher());                 /* ...tant qu'on revient de l'affichage */

    CloseDialog(dlg);                   /* on ferme le dialogue... */
    quitter(myID);                      /* ...et on s'en va */
}

        /***** FONCTION GEREDIALOGUE: gestion du dialogue principal *****/

    gereDialogue( )
{
int         item, ietype;               /* item sélectionné, son type */
char        msg[50];

    do {
        item = ModalDialog(0x80000000 | (long) monFiltre);  /* le standard et le filtre */
        ietype = GetItemType(dlg, item);                    /* le type de l'item à traiter */
        sprintf(msg,"ModalDialog: %d - GetItemType: %d  ", item, ietype);
        MoveTo(40,10); DrawCString(msg);        /* on écrit dans la barre des menus */

        switch (ietype)
            {
            case ButtonItem :                   /* un bouton simple */
                if (item == 1) ajouter( );
                else if (item == 2) supprimer( );
                break;

            case CheckItem :                    /* une case à cocher */
                if (item == 14) repEnf( );
                    break;

                case RadioItem :                    /* un bouton radio */
                    SetItemValue(1, dlg, item);
                    break;

                case EditLine :                     /* une ligne éditable non muette */
                    if (item == 5) repCode( );
                    break;
```



```
            }
        }
    while(item != 3);           /* on arrête de boucler quand le bouton 3 est sélectionné */
}

            /***** FONCTION REPCODE: action quand un nouveau code est entré *****/

    repCode( )

{
char    msg[3];
int     indprov;

    GetIText(dlg, 5, msg);                  /* on récupère le code (chaîne Pascal) */
    if (msg[0] == 1 && msg[1] == '0')       /* le code 0 a été saisi... */
        {
        ParamText("\27Le code 0\rest interdit!", 0L, 0L, 0L);
        CautionAlert(&monAlerte, 0L);       /* ...on affiche une alerte */
        }
    else
        {
        indprov = msg[0] ? msg[1] - '0' : 0;    /* le code est traduit en numérique */
        if (indprov == ind && ind !=0) return;  /* s'il n'a pas changé, on sort */
        ind = indprov;                          /* sinon, c'est le nouvel indice */
                                                /* on va afficher les données de cet
enregistrement */
        SetIText(dlg, 7, fiche[ind].nID);
        SetItemValue(1, dlg, 8+fiche[ind].statut);
        SetItemValue(1, dlg, 12+fiche[ind].sexe);
        SetItemValue(fiche[ind].age - 1, dlg, 20);
        convNP(fiche[ind].age, msg);
        SetIText(dlg, 21, msg);
        SetItemValue(!(fiche[ind].nbG + fiche[ind].nbF), dlg, 14);
        repEnf( );              /* on fait comme si on avait cliqué dans la case à cocher */
        }
    SelIText(dlg, 5, 0, 9);                 /* le code est inversé */
}

            /***** FONCTION REPENF: action quand la case Enfants est cochée ou non *****/

    repEnf( )

{
int     val;
char    msg[2];

    val = GetItemValue(dlg,14);     /* on récupère l'ancienne valeur (TRUE ou FALSE) */
    SetItemValue(!val,dlg,14);              /* et on l'inverse */
    if (val)
        {                                   /* la marque est retirée */
        HideDItem(dlg,15);                  /* on rend invisibles les textes statiques */
        HideDItem(dlg,17);
        RemoveItem(dlg,16);                 /* on supprime complètement les lignes éditables */
        RemoveItem(dlg,18);
        fiche[ind].nbG = 0;                 /* on met l'enregistrement à jour */
        fiche[ind].nbF = 0;
        }
    else                                    /* la case est cochée */
        {
        GetNewDItem(dlg, &elGar);           /* on recrée les lignes éditables */
        GetNewDItem(dlg, &elFil);
        ShowDItem(dlg,15);                  /* on rend visibles les textes statiques */
        ShowDItem(dlg,17);
```

```
                                /* on affiche les données contenues dans l'enregistrement */
            msg[0] = 1; msg[1] = '0' + fiche[ind].nbF;
            SetIText(dlg, 18, msg);
            msg[0] = 1; msg[1] = '0' + fiche[ind].nbG;
            SetIText(dlg, 16, msg);
            }
    }

        /***** FONCTION AJOUTER: met à jour l'enregistrement courant dans la structure *****/

        ajouter( )
{
char    msg[15];
int     i;

    GetIText(dlg, 7, msg);                  /* on récupère le contenu du champ nID*/
    if (ind == 0)                           /* l'indice courant est nul... */
        {
        ParamText("\30Le code\rest obligatoire!", 0L, 0L, 0L);
        StopAlert(&monAlerte, 0L);          /* ...on affiche une alerte */
        }
    else if (msg[0] == 0)                   /* le champ nID est vide... */
        {
        ParamText("\35Le champ nID\rest obligatoire!", 0L, 0L, 0L);
        StopAlert(&monAlerte, 0L);          /* ...on affiche une alerte */
        }
    else
        {
        fiche[ind].bon = TRUE ;             /* l'enregistrement est marqué correct */
        ptocstr(msg);                       /* conversion Pascal vers C */
        sprintf(fiche[ind].nID, "%s", msg); /* alimentation du champ dans la fiche... */
        ctopstr(fiche[ind].nID);            /* ...retraduite en chaîne Pascal */
        for (i=0; i<4; ++i)
            {                               /* on récupère la valeur du statut */
            if (GetItemValue(dlg, 8+i))
                {
                fiche[ind].statut = i;
                break;
                }
            }
        fiche[ind].sexe = (GetItemValue(dlg, 12)) ? 0 : 1;      /* on récupère le texte */
        fiche[ind].age = GetItemValue(dlg, 20) + 1;             /* on récupère l'âge */
        if (GetItemValue(dlg, 14))                      /* s'il y a des enfants... */
            {
            GetIText(dlg, 16, msg);
            fiche[ind].nbG = msg[1] - '0';              /* ...nombre de garçons */
            GetIText(dlg, 18, msg);
            fiche[ind].nbF = msg[1] - '0';              /* ...nombre de filles */
            }
        ecrandef( );                        /* on revient à l'écran par défaut */
        }
}

        /***** FONCTION SUPPRIMER: efface les données de l'enregistrement courant */

    supprimer( )

{
    fiche[ind] = fiche[0];          /* on met les données de l'enreg. 0 dans l'enreg. courant */
    ecrandef( );                    /* on revient à l'écran par défaut */
}

        /***** FONCTION ECRANDEF: force l'affichage de l'écran par défaut *****/
```



```
    ecrandef( )

{
    ind = 0;                        /* on force l'enregistrement 0 */
    SetIText(dlg, 5, "");           /* on force le code à blanc */
    SelIText(dlg, 5, 0, 0);         /* on place le point d'insertion dans cette case */
    repCode( );                     /* on fait comme si un nouveau code avait été entré */
}

        /***** FONCTION AFFICHER: réponse au bouton Quitter du dialogue principal *****/

    afficher( )                     /* va ouvrir le dialogue secondaire */

{
Pointer     dial;                   /* pointeur sur le dialogue secondaire */
Pointer     port;                   /* ancien grafport */
int         i, flag;
char        st[4];                  /* les quatre statuts possibles */
char        sx[2];                  /* les deux sexes possibles */
char        msg[10];

    st[0] = 'C'; st[1] = 'M'; st[2] = 'D'; st[3] = 'V';     /* initialisation des 4 statuts */
    sx[0] = 'H'; sx[1] = 'F';                               /* initialisation des 2 sexes */
    port = GetPort( );                  /* on mémorise le grafport où on écrivait */
    HideWindow(dlg);                    /* on rend invisible le dialogue principal */
    dial = GetNewModalDialog(&monDial2);/* on ouvre le dialogue secondaire... */
    SetPort(dial);                      /* ...dans lequel on va écrire directement... */
                                        /* ...tous les enregistrements marqués TRUE */
    MoveTo(5,10); DrawCString("Code nID     Statut Age Sexe nbG nbF");
    for (i=1; i<10; ++i)
        {
        if (!fiche[i].bon) break;
        sprintf(msg, "%d", i);
        MoveTo(5,10*(i+2)); DrawCString(msg);
        MoveTo(25,10*(i+2)); DrawString(fiche[i].nID);
        MoveTo(142,10*(i+2)); DrawChar(st[fiche[i].statut]);
        sprintf(msg, "%d", fiche[i].age);
        MoveTo(167,10*(i+2)); DrawCString(msg);
        MoveTo(210,10*(i+2)); DrawChar(sx[fiche[i].sexe]);
        sprintf(msg, "%d", fiche[i].nbG);
        MoveTo(240,10*(i+2)); DrawCString(msg);
        sprintf(msg, "%d", fiche[i].nbF);
        MoveTo(270,10*(i+2)); DrawCString(msg);
        }                                       /* fin de la page d'écriture */

    flag = (ModalDialog(0L) == 2) ? TRUE : FALSE ; /* dans quel bouton l'utilisateur a cliqué */
    CloseDialog(dial);                  /* fermeture du dialogue secondaire */
    SetPort(port);                      /* on rétablit le grafport précédent */
    if (flag)                           /* si on doit revenir au dialogue principal... */
        {
        ShowWindow(dlg);                /* on lui rend sa visibilité... */
        ecrandef();                     /* ...et on revient à l'écran par défaut */
        }
    return flag;                        /* on retourne TRUE ou FALSE */
}

/***** FONCTION DEFILEMENT: sera appelée automatiquement
                                        dès que la barre sera sollicitée *****/

    pascal int Defilement(comm,dialg,id)

int         comm;                       /* code de la commande */
Pointer     dialg;                      /* pointeur sur dialogue courant */
int         id;                         /* identifiant de la barre */
```

```
{
int         val;
char        msg[4];

    switch(comm)                            /* on répond à quelle commande */
        {
        case 1:                             /* taille de ce qui est vu */
            return 1;                       /* l'âge varie année par année */
            break;

        case 2:                             /* taille de ce qui est manipulé */
            return 99;                      /* on gère jusqu'à 99 années */
            break;

        case 3:                             /* valeur à l'initialisation */
            return 19;                  /* on commence à 1 an, donc 19 correspond à 20 ans */
            break;

        case 4:                             /* défilement vers le haut (flèche) */
            val = GetItemValue(dialg, id) - 1;  /* ancienne valeur dont on retire 1 */
            if (val < 0) val = 0;           /* on se borde */
            convNP(val+1, msg);             /* conversion en chaîne Pascal */
            SetIText(dialg, 21, msg);       /* affichage de l'âge en clair */
            return val;                     /* on retourne la nouvelle valeur */
            break;

        case 5:                             /* défilement vers le bas (flèche) */
            val = GetItemValue(dialg, id) + 1;  /* ancienne valeur à laquelle on ajoute 1 */
            if (val > 98) val = 98;
            convNP(val+1, msg);
            SetIText(dialg, 21, msg);           /* idem */
            return val;
            break;

        case 6:                             /* défilement vers le haut (bande) */
            val = GetItemValue(dialg, id) - 10;  /* ancienne valeur dont on retire 10 */
            if (val < 0) val = 0;
            convNP(val+1, msg);
            SetIText(dialg, 21, msg);           /* idem */
            return val;
            break;

        case 7:                             /* défilement vers le bas (bande) */
            val = GetItemValue(dialg, id) + 10;  /* ancienne valeur à laquelle on ajoute 10 */
            if (val > 98) val = 98;
            convNP(val+1, msg);
            SetIText(dialg, 21, msg);           /* idem */
            return val;
            break;

        case 8:                             /* utilisation du curseur */
            val = GetItemValue(dialg, id);      /* nouvelle valeur, qu'on ne touche pas */
            convNP(val+1, msg);
            SetIText(dialg, 21, msg);           /* idem */
            return val;
            break;
        }
}

/***** FONCTION MONFILTRE: filtre tous les événements clavier
                                                    dans le dialogue principal *****/

    pascal int monFiltre(dialog, event, itemHit)
```



```
Pointer     dialog;                 /* le dialogue courant */
TaskRec     *event;                 /* pointeur sur l'événement courant */
int         *itemHit;               /* à forcer quand on retourne TRUE */

{
int     car;
char    msg[30];

    /* on laisse le Dialog Manager gérer les événements autres que les événements clavier */
    if (event->what != KeyDown && event->what != AutoKey) return FALSE;
    car = event->message & 0xFF;            /* code ASCII du caractère tapé */

    if (car == 0x06                         /* contrôle-F */
        || car == 0x08                      /* flèche gauche */
        || car == 0x09                      /* tabulation */
        || car == 0x0D                      /* retour */
        || car == 0x15                      /* flèche droite */
        || car == 0x18                      /* clear */
        || car == 0x19                      /* contrôle-Y */
        || car == 0x7F)                     /* effacement */
        return FALSE;           /* on laisse le Dialog Manager gérer tous ces caractères */

    if (event->modifiers & AppleKey)        /* la touche Pomme est-elle enfoncée? */
        {
        if (car == 'Q' || car == 'q')       /* l'utilisateur a tapé Pomme-Q ou Pomme-q... */
            {
            *itemHit = 3;               /* ...c'est équivalent à avoir cliqué dans l'item 3... */
            return TRUE;                /* ...et on empêche le DM de gérer l'événement */
            }
        else return FALSE;      /* on laisse le DM gérer les autres combinaisons Pomme */
        }

    if (car<'0' || car>'9')                 /* si ce n'est pas un chiffre... */
        {
        ParamText("\47Vous ne pouvez\rentrer que des chiffres!",0L,0L,0L);
        NoteAlert(&monAlerte, 0L);          /* ...on affiche une alerte... */
        event->what = 0;        /* ...et on transforme l'événement clavier en événement nul */
        }
    return FALSE;                       /* on laisse le DM gérer les bons caractères */
}

        /***** FONCTION CONVNP: conversion d'un nombre en chaîne Pascal *****/

    convNP(nbre,pstr)

int         nbre;                       /* le nombre à convertir */
Pointer     pstr;                       /* l'adresse où stocker la chaîne */

{
    sprintf(pstr, "%d", nbre);          /* conversion du nombre en chaîne C */
    ctopstr(pstr);                      /* conversion de la chaîne C en chaîne Pascal */
}
```

CHAPITRE X

POUR QUELQUES OUTILS DE PLUS

Dans ce chapitre, nous évoquerons quelques outils supplémentaires, sans prétendre à être exhaustif. Les routines évoquées pouvant être d'un grand intérêt, il aurait été dommage de les laisser de côté sous prétexte qu'elles n'entraient pas dans le cadre des chapitres précédents.

MISCELLANEOUS TOOLS

Les Miscellaneous Tools regroupent, comme le nom permet de le supposer, diverses routines sans lien apparent, toutes en ROM. Par exemple les quatre procédures qui vont lire et écrire dans les 256 octets de mémoire vive sauvegardée par pile *(Battery Ram)* ; les deux procédures permettant d'installer et de retirer une fonction s'exécutant en tâche de fond à chaque interruption VBL ; les routines gérant les erreurs système, au nom évocateur de System Death Manager ; les deux routines gérant le compactage des images ; la fonction **Munger**, qui permet certaines manipulations d'octets dans des chaînes d'octets, etc.

Nous nous contenterons de décrire une procédure qui permet de lire l'horloge de l'Apple IIGS, en donnant un résultat lisible presque directement.

La procédure **ReadAsciiTime** renvoie la date et l'heure dans une chaîne de 20 caractères dont l'adresse est passée en argument. Deux pièges à contourner. Premièrement, chacun des caractères a son bit le plus significatif à 1, ce qui est intéressant quand on fait du texte dans les anciens modes de résolution Apple II, mais plutôt gênant en application desktop : il faut remettre ce bit à 0 pour obtenir un résultat lisible. Deuxièmement, la chaîne de caractères contenant le résultat n'est ni une chaîne Pascal ni une chaîne C : les 20 caractères sont significatifs.

L'exemple suivant montre comment afficher l'heure exacte en permanence à l'écran.

```
char    date[20];
int     j;

do {
ReadAsciiTime(date);            /* on lit la date et l'heure */
for(j=0; j<20; ++j)
    {
    date[j] &= 0x7F;            /* force à 0 le bit significatif */
    MoveTo(40,40);
    DrawText(date, 20);         /* dessin de 20 caractères formant un texte */
    }
}
while (!Button(0));             /* tant que le bouton n'est pas enfoncé */
```

Le format de sortie est identique à celui que l'utilisateur a choisi dans l'accessoire de bureau Tableau de Bord. Un bon Français aura choisi JJ/MM/AA HH:MM:SS, où HH varie de 0 à 23. Un Anglo-Saxon utilisera peut-être MM/JJ/AA HH:MM:SSxx, où HH varie de 1 à 12 et xx prend la valeur AM ou PM. Six format sont possibles (3 variantes pour la date, 2 pour l'heure).

Remarque L'accessoire de bureau Clock (voir figure X.1) est une illustration parfaite de la procédure **ReadAsciiTime**.

Figure X.1. A l'horloge, il est minuit !

DESK MANAGER

Nous avons déjà évoqué plusieurs fois certaines routines du Desk Manager tout au long de cet ouvrage. Nous n'entrerons pas dans les détails de l'écriture d'un accessoire de bureau, mais nous allons décrire les routines permettant à une application de les utiliser. De même, nous ne donnerons pas d'exemple complet d'utilisation, puisque deux exemples ailleurs dans l'ouvrage gèrent complètement lesdits accessoires (voir le chapitre V sur le Window Manager et le chapitre XI sur **TaskMaster**).

Accessoires classiques et nouveaux accessoires

Il existe deux sortes d'accessoires de bureau sur l'Apple IIGS : les accessoires de bureau classiques et les nouveaux accessoires de bureau.

• Les accessoires classiques tournent dans un environnement qui n'est pas basé sur le concept de desktop et d'événements. Ils interrompent le déroulement de l'application qui les appelle, sauvegardent l'environnement, installent leur propre environnement (généralement un écran mode texte 40 ou 80 colonnes), font ce qu'ils ont à faire, puis restaurent l'environnement de l'application et lui rendent la main. Quand l'utilisateur appuie sur la combinaison de touches Pomme-Contrôle-Escape, un menu contenant la liste des accessoires de bureau classiques est affiché. Le menu se compose automatiquement : tout fichier ProDOS de type $B9 présent dans un dossier particulier de la disquette système sera ajouté à la liste (*/system/desk.accs).
Remarque Le Tableau de Bord et l'accessoire Affichage Alternatif sont résidents dans le système. Jusqu'à onze accessoires supplémentaires peuvent être chargés.

Pour gérer les accessoires classiques, une application n'a strictement rien à faire : si elle utilise une boucle d'événements (**GetNextEvent** ou **TaskMaster**), la combinaison

Pomme-Contrôle-Escape sera interceptée par ces fonctions, et la main sera donnée à leur menu. En effet, **GetNextEvent** appelle une fonction du Desk Manager, **SystemEvent**, qui sert en quelque sorte de filtre aux événements : si le Desk Manager veut traiter le nouvel événement, il l'intercepte et l'application ne le recevra pas. Entre autres choses, **SystemEvent** intercepte la combinaison Contrôle-Pomme-Escape. Cette fonction ne doit pas être appelée explicitement par l'application.

• Les nouveaux accessoires de bureau fonctionnent à la manière du Macintosh : concurremment avec l'application qui les invoque, dans l'environnement desktop (fenêtres, menus déroulants, contrôles) et événements. Un accessoire a la main dès que la fenêtre qui le représente est au premier plan, certains accessoires peuvent avoir une action périodique (telle l'horloge) qui ne nécessitent pas l'intervention de l'utilisateur. La liste des nouveaux accessoires de bureau est toujours présente dans le menu . Comme pour les accessoires classiques, elle se compose automatiquement (à condition que l'application ait appelé la procédure **FixAppleMenu**) : tout fichier ProDOS de type $B8 présent dans un dossier particulier de la disquette système sera ajouté à la liste (*/system/desk.accs).

Pour gérer les nouveaux accessoires de bureau, une application n'a pas beaucoup de travail à faire. Si elle utilise **TaskMaster** (voir le chapitre suivant), elle n'aura qu'à initialiser le Desk Manager (avec **DeskStartUp**) et placer la liste des accessoires dans le menu (avec **FixAppleMenu**). Si elle n'utilise pas **TaskMaster**, elle devra en plus ouvrir l'accessoire en réponse aux sollicitations de l'utilisateur, appeler à l'intérieur de la boucle d'événements la procédure **SystemTask** pour gérer les accessoires à action périodique, appeler **SystemClick** pour répondre à un clic-souris dans une fenêtre système, appeler **SystemEdit** quand l'utilisateur choisit l'un des articles standard du menu Edition (Annuler, Couper, Copier, Coller et Effacer) alors qu'un accessoire est actif, et enfin fermer l'accessoire du premier plan quand l'utilisateur choisit l'article Fermer du menu Fichier (deux routines sont disponibles : **CloseNDA** et **CloseNDAbyWinPtr**).

Ce sont ces routines que nous allons détailler maintenant.

Gestion des nouveaux accessoires de bureau

• L'initialisation du Desk Manager se fait par la procédure **DeskStartUp**, qui ne réclame aucun argument. Pour fonctionner, le Desk Manager a besoin de beaucoup d'autres outils : QuickDraw (tout le monde a besoin de QuickDraw), l'Event Manager (un accessoire répond à des événements), le Window Manager (un accessoire est susceptible d'être représenté dans une fenêtre), le Control Manager (la région contour d'une fenêtre contient des contrôles), le Menu Manager (on va chercher les accessoires dans le menu , certains accessoires ajoutent des menus déroulants aux applications) et éventuellement Line Edit et le Dialog Manager. Consulter le chapitre XII pour une vision d'ensemble de l'initialisation des outils.

• L'installation des titres d'accessoires dans le menu se fait par la procédure **FixAppleMenu**. En argument, on passe un entier qui sera l'identifiant du menu dans lequel les accessoires doivent être placés. Les accessoires seront identifiés par les numéros 1, 2, ... Rappelons que les identifiants d'articles compris entre 1 et 255 sont réservés par le système.

Supposons que le dossier spécial des accessoires de bureau de la disquette système contienne 10 fichiers de type $B8 et que l'appel suivant soit passé :

```
FixAppleMenu(1);                /* 10 accessoires placés dans le menu 1 */
```

Après cet appel, le menu 1 (il vaut mieux pour l'interface utilisateur que ce soit le menu) contiendra 10 articles supplémentaires (le nom des 10 accessoires) dont les identifiants seront compris entre 1 et 10.
Remarque Pour connaître le nombre d'accessoires installés, on peut appeler la fonction **GetNumNDAs**, sans argument, qui retourne ce nombre dans un entier.

• L'utilisateur a déroulé le menu et choisi un accessoire à ouvrir. L'application a appelé **MenuSelect** (voir le Menu Manager) et connaît l'identifiant de l'article sélectionné (inférieur à 250). Cet identifiant est exactement l'argument à donner à la fonction **OpenNDA** pour que le Desk Manager ouvre l'accessoire choisi. La fonction retourne dans un entier un numéro de référence pour cet accessoire, qui permettra de le fermer après utilisation.

• Supposons que notre accessoire de bureau ouvert est une horloge, précise à la seconde près. Il serait désolant que, sous prétexte que la fenêtre qui la contient n'est pas active, elle cesse de tourner, surtout si elle est visible ! La procédure **SystemTask**, sans argument, permet à chaque accessoire ouvert d'exécuter la tâche périodique pour laquelle il a été programmé, quand elle existe et si c'est le moment d'agir. Cette notion de périodicité est incluse dans la définition de l'accessoire. Elle est variable avec l'accessoire : la même horloge qui n'afficherait que les minutes aurait une fréquence d'appel 60 fois moins importante. Le Desk Manager garde trace de la périodicité de chaque accessoire, et leur donnera la main au travers de **SystemTask** uniquement si le moment est venu d'exécuter l'action périodique.

Remarque Imaginons un accessoire de bureau qui affiche en permanence l'état de la mémoire (nombre d'octets disponibles pour allouer le plus grand bloc possible, par exemple). L'action de cet accessoire est périodique, puisqu'il doit sans cesse scruter la mémoire vive, mais il n'y a aucune raison que la période soit fixe. Dans ce cas, le Desk Manager offre la notion de « aussi souvent que possible » en guise de périodicité : **SystemTask** donnera systématiquement la main à un tel accessoire.

• Un accessoire n'a pas forcément une action périodique. Le plus souvent, il n'en a pas, mais attend plutôt une intervention de l'utilisateur dans la fenêtre qu'il gère. Après un clic souris, dès que la fonction **FindWindow** du Window Manager retourne un nombre négatif, la procédure **SystemClick** doit être appelée. Elle admet trois arguments : un pointeur sur l'événement que l'application est en train de traiter, le pointeur sur la fenêtre désignée par **FindWindow** et le code retourné par cette fonction. C'est alors **SystemClick** qui prend en charge le traitement complet de l'événement, l'application n'a rien de spécial à faire.

```
TaskRec   tache;          /* l'événement en cours de traitement */
Pointer   wind;           /* la fenêtre dans laquelle l'utilisateur a cliqué */
int       code;           /* code retourné par FindWindow */

...                       /* on est en train de traiter un nouvel événement */
case MouseDown :                        /* c'est un clic souris */
    code = FindWindow(&wind, tache.where);   /* si le code retourné est négatif... */
    if (code<0) SystemClick(&tache, wind, code); /* ...on est dans une fenêtre système! */
    else
    ...                   /* sinon on procède comme d'habitude */
    break;
```

Notons que l'application n'a rien à faire si l'utilisateur choisit un article dans un menu déroulant qui est géré par un accessoire de bureau : c'est le Menu Manager qui se chargera de transmettre l'information au Desk Manager pour sa prise en compte par l'accessoire.

De même, l'application n'a pas à se préoccuper des événements de type clavier pour un accessoire : si la fenêtre de premier plan est une fenêtre système, ces événements sont interceptés par la fonction **SystemEvent** et passés directement à l'accessoire qui les gérera comme bon lui semble. Malheureusement, même les équivalents-clavier sont passés au Desk Manager... qui n'exécute pas pour autant la commande dans un menu géré par l'application !

• Une fenêtre d'accessoire de bureau est au premier plan et l'utilisateur a choisi l'un des articles standard du menu Edition. L'application doit faire savoir à l'accessoire quelle action il doit entreprendre, et utilise pour cela la fonction **SystemEdit**. Celle-ci commence par vérifier la nature de la fenêtre située au premier plan, et retourne TRUE s'il s'agit d'une fenêtre système (elle accepte de prendre en

compte la commande) ou FALSE si la fenêtre appartient à l'application (elle refuse la commande, c'est à l'application de la gérer). L'argument passé à **SystemEdit** est l'une des valeurs prédéfinies suivantes :

```
#define Undo      1
#define Cut       2
#define Copy      3
#define Paste     4
#define Clear     5
```

Les réponses de l'application aux choix du menu Edition auront donc l'aspect suivant :

```
case Annuler:
    if (!SystemEdit(Undo))
        ...               /* l'application prend la commande en charge */
    break;

case Couper:
    if (!SystemEdit(Cut))
        ...               /* l'application prend la commande en charge */
    break;

case Copier:
    if (!SystemEdit(Copy))
        ...               /* l'application prend la commande en charge */
    break;

case Coller:
    if (!SystemEdit(Paste))
        ...               /* l'application prend la commande en charge */
    break;

case Effacer:
    if (!SystemEdit(Clear))
        ...               /* l'application prend la commande en charge */
    break;
```

• Pour fermer l'accessoire de bureau, l'utilisateur a deux possibilités : cliquer dans la case de fermeture de sa fenêtre (si elle existe) ou choisir l'article Fermer dans le menu Fichier. Dans le premier cas, **SystemClick** se charge de la fermeture. Dans le second cas, l'application a deux possibilités. Ou bien elle appelle la procédure **CloseNDA**, mais pour cela elle doit connaître le numéro de référence de l'accessoire à fermer (celui qu'a retourné **OpenNDA**) et le passer en argument ; ou bien elle récupère un pointeur sur la fenêtre de l'accessoire grâce à la fonction **FrontWindow** du Window Manager et appelle la procédure **CloseNDAbyWinPtr** avec ce pointeur en argument.

```
Pointer port;

port = FrontWindow( );                  /* pointeur sur la fenêtre à fermer */
if (GetWKind(port) < 0)
    CloseNDAbyWinPtr(port);             /* la fenêtre appartient à un accessoire */
else CloseWindow(port);                 /* la fenêtre appartient à l'application */
```

Remarque Le fait d'appeler **CloseWindow** pour fermer un accessoire de bureau provoque bien la fermeture de la fenêtre de cet accessoire, mais pas la fermeture de l'accessoire lui-même ! Il est toujours ouvert, occupe de la mémoire mais devient inaccessible. On veillera donc à employer **CloseNDAbyWinPtr** et non **CloseWindow** quand la fenêtre à fermer est une fenêtre système.

En cas de nécessité (besoin d'espace mémoire par exemple), une application peut fermer d'un coup tous les accessoires ouverts, grâce à la procédure **CloseAllNDAs**, qui ne réclame aucun argument.

SCRAP MANAGER

Principes généraux

Le Scrap Manager est l'outil qui permet le copier-coller entre deux applications, entre une application et un accessoire de bureau, entre deux documents d'une même application, ou encore entre deux parties du même document. Du point de vue de l'utilisateur, les données transitent par un fichier dit presse-papiers (*Clipboard* en anglais). Quand il copie de l'information, il la copie vers le presse-papiers. Quand il coupe de l'information, il la transfère vers le presse-papierrs en faisant place nette derrière lui. Quand il colle de l'information, il va la chercher dans le presse-papiers et en fait une copie dans son document, le presse-papiers n'étant pas affecté. Le presse-papiers n'est pas un meuble à tiroirs : dès que l'utilisateur met quelque chose dedans, il perd ce qui s'y trouvait précédemment.

Côté application, la vision est plutôt différente : le presse-papiers n'est pas forcément un fichier, mais plutôt un bloc de données en mémoire, repéré par un handle. Si l'information est unique, elle peut tout de même être mémorisée sous plusieurs aspects.

Considérons un traitement de texte : il manipule du texte, dans lequel les caractères sont tous affectés d'un style, d'une taille, voire d'une couleur, et peuvent appartenir à des polices différentes. Ce texte est éditable, dans le sens où on peut venir lui insérer des caractères, lui en retirer, etc. Enfin, on peut faire du copier-coller sur des portions de texte.

Comment l'application doit-elle mémoriser cette portion de texte, sachant qu'une fois dans le presse-papiers, cette information peut repartir soit vers le même document ou un autre document du même traitement de texte, soit vers un éditeur de texte qui ne reconnaît pas les formats propres à cette application, soit vers une application graphique qui ne connaît que la couleur des pixels.

Pour ses documents propres, le traitement de texte doit mémoriser ses données dans leur format, afin de ne perdre aucune information durant le transfert. Mais pour les documents manipulés par d'autres applications, qui n'ont aucune raison de connaître ce format, il faut également les mémoriser d'une manière qui soit universelle, dans des formats (on dit des types) qui sont censés être connus de tout le monde. Le Scrap Manager permet de faire figurer dans le presse-papiers une même information sous plusieurs formats différents : à l'application qui reçoit le contenu du presse-papiers de choisir le format qui lui convient le mieux, et d'ignorer le coller si aucun ne lui convient.

Deux types sont universels sur Apple IIGS (comme sur Macintosh), le type *texte* et le type *picture*. Les données de type texte ne contiennent rien d'autre que les codes ASCII des caractère composant le texte (elles sont donc compatibles entre les deux machines, sans restriction). Les données de type picture sont comme leur nom l'indique une picture au sens QuickDraw du terme (on pourrait imaginer qu'elles soient également compatibles entre les deux machines, puisqu'on a affaire à une suite d'instructions interprétables par QuickDraw, mais les problèmes de couleurs et de résolution graphique rendent l'exercice beaucoup plus périlleux... et moins intéressant).

Il appartient à toute application désirant gérer le copier-coller de connaître l'un et/ou l'autre de ces types, pour recevoir comme pour transférer. Si notre traitement de texte fait les choses correctement, il transférera trois types d'informations dans le presse-papiers : son format propre (qui est son format préféré), le type texte et le type picture. L'éditeur de texte qui recevra cette information choisira le type texte, sachant qu'il perd toute notion de présentation du texte, l'application graphique choisira le type picture, sachant qu'elle perd la faculté d'éditer les caractères reçus. Par rapport au format propre, il y a forcément perte d'information.

Dans l'autre sens, si notre traitement de texte reçoit une information du presse-papiers, il va commencer par chercher son type préféré. S'il le trouve, tant mieux : aucune perte d'information ne sera à déplorer. Sinon, il cherchera le type texte et lui donnera un format par défaut (police système, taille, style et couleurs standard, vraisemblablement). Ce texte sera éditable. Sinon, il ira chercher le type picture, et insérera l'image obtenue (à condition de savoir gérer les images) au milieu de son document, sans qu'on puisse faire autre chose sur cette image que la supprimer. Sinon, ne trouvant aucun type connu, il ignorera le contenu du presse-papiers.

Le presse-papiers peut résider en mémoire, ou sur disquette. L'application n'a pas à se préoccuper de l'endroit où il réside, c'est le travail du Scrap Manager de savoir à tout moment où il est. Quand il est sur disque, c'est réellement un fichier, qui porte le nom Clipboard et qui se trouve dans le dossier Système. Quand il est en mémoire, il s'y trouve sous forme de blocs relogeables : un bloc par type de données qui y réside.

Le presse-papiers est unique pour assurer la communication entre différentes applications. Il est possible toutefois pour une application de gérer son propre presse-papiers privé (Line Edit ne s'en prive d'ailleurs pas, voir le chapitre VIII). Le format est libre, puisque seule l'application l'utilisera. Elle pourra par exemple gérer un simple pointeur sur un bloc d'informations à copier, dans son format préféré. Ce sera à elle de traduire ensuite l'information contenue dans un presse-papiers privé et de la transférer dans le presse-papiers public, pour que d'autres applications ou les accessoires de bureau puissent l'utiliser.

Routines du Scrap Manager

• Pour être utilisé, le Scrap Manager doit avoir été initialisé, par la procédure **ScrapStartUp**, sans argument.

• Deux procédures sans argument permettent le transfert du presse-papiers entre le disque et la mémoire : **LoadScrap** et **UnloadScrap**. **LoadScrap** charge le presse-papiers en mémoire. Si aucun fichier Clipboard n'est trouvé sur disque (soit qu'il n'existe pas, soit qu'il n'est pas dans le bon dossier), aucune erreur n'est rapportée : le presse-papiers est considéré vide, tout simplement. Si le presse-papiers réside déjà en mémoire, il ne se passe rien. **UnloadScrap** copie le presse-papiers sur disque, et libère la place qu'il occupait en mémoire. Si le presse-papiers se trouve déjà sur disque, il ne se passe rien.

Notons que ces deux procédures peuvent générer des erreurs ProDOS ou Memory Manager (un disque protégé en écriture, une mémoire saturée, etc.), dont l'application doit tenir compte. Cependant, l'application n'appellera vraisemblablement pas ces procédures directement. Toute action sur le presse-papiers s'effectuant en mémoire vive, le presse-papiers sera automatiquement chargé dès qu'on l'invoquera. Une application pourra toutefois le transférer sur disque, si elle a besoin de mémoire pour fonctionner (mais alors il risque de devenir inaccessible).

Une fonction, **GetScrapState**, sans argument, retournera TRUE si le presse-papiers réside en mémoire, et FALSE s'il est censé se trouver sur disque (censé seulement, parce que l'utilisateur peut l'avoir détruit).

Une fonction, **GetScrapPath**, sans argument, retourne un pointeur sur le *pathname*, chemin d'accès ProDOS, utilisé pour le fichier Clipboard sur disque. Une procédure, **SetScrapPath**, fixe un nouveau chemin d'accès (le pointeur sur cette chaîne de caractères est passé en argument). Ces routines n'ont a priori aucune raison d'être utilisées.

• Pour effacer le contenu du presse-papiers public, on appelle la procédure **ZeroScrap**, sans argument. Si le presse-papiers réside sur disque, il est chargé en mémoire et vidé. Cette procédure change le compteur géré par le Scrap Manager (voir plus loin).

• Pour écrire quelque chose dans le presse-papiers public, on appelle la procédure **PutScrap**. Trois arguments : la longueur en octets de l'information à écrire (entier long), le type de l'information (entier sur 16 bits) et un pointeur sur l'information. Pour mettre quelque chose dans le presse-papiers, on pourra procéder ainsi : s'allouer un bloc fixe en mémoire, écrire à cette adresse l'information dans un format correct, et appeler **PutScrap** en lui précisant l'adresse des données, le nombre d'octets qu'elles contiennent et le format dans lequel elles sont écrites. **PutScrap** fera une copie de ces données dans le presse-papiers en les ajoutant à celles du même type qui pouvaient déjà s'y trouver. L'opération se passe en mémoire : si le presse-papiers se trouve sur disque, la procédure appelle elle-même **LoadScrap**. Pour placer dans le presse-papiers plusieurs versions de la même information, on appellera plusieurs fois **PutScrap**. Le type texte porte le code 0, le type picture le code 1 : interdiction formelle d'utiliser ces types pour d'autres conventions de format !

```
long    taille, taille0, taille1;
Pointer ptr, ptr0, ptr1;

/* l'utilisateur vient de passer la commande Copier */
ZeroScrap( );                       /* on vide le presse-papiers */
...                 /* calcul de la taille des données, détermination de leur adresse */
PutScrap(taille, 1543, ptr); /* 1543 est le type choisi par l'application pour son format */
...                 /* conversion des données au format texte, à une adresse spécifiée */
PutScrap(taille0, 0, ptr0);         /* 0 est le format texte */
...                 /* conversion des données au format picture, à une adresse spécifiée */
PutScrap(taille1, 1, ptr1);         /* 1 est le format picture */
/* le presse-papiers contient maintenant une information sous trois formats différents */
```

Note La procédure **LEToScrap** copie le contenu du presse-papiers privé de Line Edit dans le presse-papiers public, après l'avoir vidé. Seul le type texte est alors disponible.

• Pour lire quelque chose dans le presse-papiers public, on utilise la procédure **GetScrap**. Deux arguments : un handle précisant la destination des données, et le type des données recherché. Le handle doit avoir été alloué par l'application (il peut être vide). **GetScrap** chargera le presse-papiers en mémoire, si nécessaire, et copiera le contenu du presse-papiers correspondant au type désigné dans le bloc désigné par le handle, ajustant sa taille comme nécessaire. Si la taille du bloc après l'appel est zéro-long, c'est que le type n'a pas été trouvé, soit parce qu'il n'existait pas, soit parce que le presse-papiers était vide. L'application pourra éventuellement demander la lecture d'un nouveau type.

Avant de lire un type, il sera toutefois plus astucieux d'appeler la fonction **GetScrapSize**, qui retournera dans un entier long la taille des données correspondant au type passé en argument, zéro-long signifiant que le type en question n'est pas disponible.

```
Handle hdl;

/* l'utilisateur vient de passer la commande Coller */
hdl = NewHandle(0L, myId, 0, 0L);       /* allocation d'un bloc vide */
if (GetScrapSize(1543) != 0L)           /* recherche du type propre à l'application */
   {
   GetScrap(hdl, 1543);                 /* il y en a: on les lit... */
   ...                                  /* ...et on les traite */
   }
else if (GetScrapSize(0) != 0L)         /* recherche du type texte */
   {
   GetScrap(hdl, 0);                    /* il y en a: on les lit... */
   ...                                  /* ...et on les traite */
   }
/* le presse-papiers reste intact après ces appels */
```

Dans l'exemple précédent, notre application ne sait pas gérer le type picture, elle ne cherche donc pas à le lire ! Ce n'est pas incompatible avec le fait qu'elle écrive le type picture dans le presse-papiers, loin de là ! Elle ignorera le Coller si elle ne trouve pas de données à son goût.

Note La procédure **LEFromScrap** vient chercher les données de type texte du presse-papiers public et les copie dans le presse-papiers privé de Line Edit. Cette opération est limitée à 256 caractères.

• Supposons qu'une application gère le copier-coller, et ait inclus dans son menu Edition la commande Afficher le presse-papiers. Elle gère donc une fenêtre qui affiche en permanence son contenu, quand elle est ouverte. L'application a donc besoin de savoir à chaque instant si ce contenu a changé, pour pouvoir remettre à jour ce qu'affiche la fenêtre. La fonction **GetScrapCount**, sans argument, le permet. Elle retourne la valeur d'un compteur (entier sur 16 bits) qui change à chaque fois que la procédure **ZeroScrap** est appelée. Puisque l'écriture d'une information nouvelle dans le presse-papiers (information, pas type !) doit être obligatoirement précédée de l'appel à cette procédure, si le compteur a changé entre deux appels, c'est que le contenu du presse-papiers public a également changé.

• Le presse-papiers est géré en mémoire par le Scrap Manager comme un bloc relogeable classique pour chaque type de données qu'il contient, repéré par un handle. Il peut être intéressant d'aller lire ou écrire directement dans ces blocs, plutôt que de passer par les procédures **GetScrap** et **PutScrap**, quand l'espace mémoire est limité, puisque ces procédures font une copie de l'information. La fonction **GetScrapHandle** retourne le handle sur le bloc de données dont le type est passé en argument.

STANDARD FILE

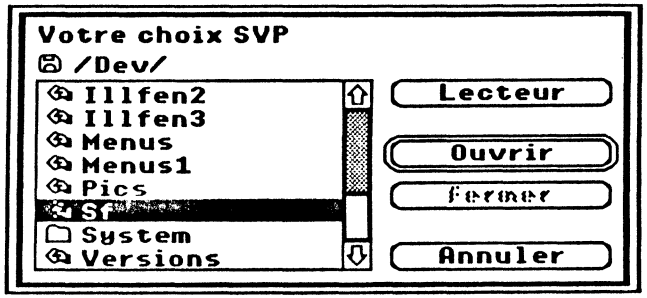
L'accès à la disquette dans une application Apple IIGS se fait toujours par l'intermédiaire de dialogues banalisés, qu'il s'agisse d'ouvrir un document (lecture d'un fichier) ou de l'enregistrer (écriture d'un fichier). Ces fenêtres de dialogue sont gérées par l'outil Standard File Operations, qui met 4 routines à notre disposition : deux pour le choix du fichier en lecture, deux pour le choix en écriture.

Initialisation

L'outil sera initialisé grâce à la procédure **SFStartUp**, qui réclame deux arguments : l'identifiant de l'application (tel que retourné par le Memory Manager) et l'adresse de la page zéro dont l'outil se servira de manière interne. Consulter le chapitre XII pour une vision d'ensemble de l'initialisation des outils.

Choix du fichier à lire

Deux procédures sont à notre disposition pour présenter le dialogue qui permettra à l'utilisateur de choisir le fichier (application ou document) qu'il désire ouvrir. La fenêtre standard présente quatre boutons, permettant de changer de lecteur (équivalent-clavier la touche de tabulation), d'ouvrir un fichier ou un dossier (équivalent-clavier la touche Retour ou Entrée), de fermer un dossier (équivalent-clavier la touche Escape), d'annuler l'action d'ouvrir. La liste qui défile contient le nom de tout ou partie des fichiers présents sur la disquette, certains pouvant éventuellement être estompés, ainsi que les noms des dossiers (qu'on peut ouvrir par double-clic). En haut apparaît une indication contrôlée par l'application, ainsi que le nom du répertoire actif. Quand on tape un caractère au clavier, la barre de sélection se déplace sur le premier nom de fichier non estompé qui commence par ce caractère.



• **SFGetFile** est l'appel standard. Cette procédure affiche le dialogue standard et permet à l'application de déterminer quel fichier l'utilisateur a choisi. Elle réclame six arguments :

– les deux premiers sont l'abscisse et l'ordonnée en coordonnées globales du coin supérieur gauche de la fenêtre de dialogue, permettant ainsi à l'application de la positionner comme bon lui semble ;

– le troisième argument est un pointeur sur une chaîne de caractères type Pascal qui apparaîtra en haut de la fenêtre de dialogue. On veillera à ce qu'elle ne soit pas trop longue étant donné la largeur de la fenêtre et le mode de résolution utilisé. Cette chaîne peut être vide ;

– le quatrième argument donne l'adresse d'une fonction filtre, qui va permettre de choisir si tel ou tel fichier doit ou non être affiché comme élément de choix. Passer zéro-long pour empêcher **SFGetFile** d'appeler une fonction filtre.

La fonction filtre est une fonction de type Pascal, qui admet un seul argument, l'adresse d'une entrée dans le directory sur lequel se trouve l'utilisateur (c'est **SFGetFile** qui la passe). En fonction de ce renseignement, la fonction filtre doit dire ce qu'elle fait de ce fichier : 0 pour ne pas l'afficher, 1 pour l'afficher estompé (de telle sorte qu'il sera visible mais non sélectionnable), 2 pour permettre à l'utilisateur de le choisir. C'est grâce à une fonction filtre qu'un compilateur C, par exemple, pourra ne retenir que les noms de fichiers qui possèdent le suffixe .c. D'autres applications pourront se limiter à des fichiers possédant certaines caractéristiques, combinaisons de type et de type auxiliaire, par exemple.

Comme nous n'avons pas l'intention d'entrer dans les détails de ProDOS, nous ne parlerons pas davantage de la fonction filtre, et nous nous contenterons de l'argument suivant pour limiter les sélections.

– le cinquième argument est un pointeur sur une liste de types (le type du fichier au sens ProDOS du terme). Seuls les fichiers correspondant aux types présents dans cette liste seront affichés. Mettre zéro-long pour utiliser la liste par défaut, qui affiche tout ce que contient la disquette. Une autre manière d'afficher tous les fichiers, quel que soit leur type, est de pointer sur une liste nulle.

Une liste de types aura la forme suivante : le premier octet donne le nombre d'entrées dans la liste, les octets suivants correspondent chacun à un type. C'est parfaitement similaire à la façon dont est définie une chaîne de caractères style Pascal.

```
char listenulle[ ] = {0};          /* la liste nulle: tous les fichiers seront affichés */
char listeappl[ ] = {1, 0xB3};     /* fichiers de type $B3: applications sous ProDOS 16 */
char listeda[ ] ={2, 0xB8, 0xB9};  /* accessoires de bureau:
                                      nouveaux ($B8) et classiques ($B9) */
```

– le sixième argument est un pointeur sur une structure particulière, le *Reply record*, que nous pouvons définir de la manière suivante :


```
struct _SFReply {
    int     good;          /* valeur booléenne, FALSE s'il n'y a rien à faire */
    int     type;          /* type ProDOS du fichier */
    int     auxtype;       /* type auxiliaire pour le fichier ProDOS */
    char    filename[15];  /* nom du fichier dans le préfixe 0 (chaîne type Pascal) */
    char    fullname[128]; /* chemin d'accès complet au fichier sélectionné */
    };
#define SFReply struct _SFReply
```

Note Dans sa version 1.0, il semble que l'outil Standard File Operations n'alimente pas le champ *fullname*. Comme nous le verrons plus loin, il est de toute façon d'une inutilité flagrante.

C'est au travers de cet argument que nous allons recevoir le choix de l'utilisateur. Si celui-ci a cliqué dans le bouton Annuler, le champ *good* contient la valeur FALSE (zéro). Il n'y a donc rien de plus à faire, aucun fichier à lire sur la disquette. Si par contre l'utilisateur est sorti en cliquant dans le bouton Ouvrir, c'est qu'un fichier est sélectionné. Le champ *good* contient une valeur non nulle (TRUE) et les autres champs donnent quelques caractéristiques du fichier sélectionné, dont son nom complet, pour en permettre l'ouverture. A l'application de déterminer si ce fichier lui appartient ou pas, si possible sans provoquer de plantage !

Quand la fenêtre apparaît, elle contient les fichiers affichables du préfixe 0, c'est-à-dire du répertoire par défaut. Quand l'utilisateur a choisi un fichier, le répertoire par défaut devient celui auquel appartient le fichier. Pour faire référence au fichier sélectionné, il suffira donc d'utiliser la chaîne de caractères « 0/nomtich », où *nomfich* est le nom retourné dans le champ *filename* de la structure *SFReply*. Voici une façon parmi tant d'autres d'obtenir cette chaîne en C (une nouvelle fois, on utilise la fonction sprintf de la bibliothèque C, on aurait pu utiliser concat) :

```
SFReply  reply;                          /* une structure SFReply */
char     fichsel[20];                    /* 20 caractères réservés en mémoire */

SFGetFile(30, 45, "\17Votre choix SVP", 0L, 0L, &reply);    /* dialogue standard */
if (reply.good)
    {                                    /* l'utilisateur a-t-il cliqué dans Ouvrir? */
    ptocstr(reply.filename);             /* conversion type chaîne Pascal -> C */
    sprintf(fichsel, "0/%s", reply.filename); /* fichsel pointe sur le nom du fichier complet */
    open(fichsel, 0);                    /* ouverture du fichier */
    ...
    }
```

Le bouton Lecteur est géré par **SFGetFile**. Quand l'utilisateur clique dedans, le système commence par regarder dans le lecteur sur lequel il était positionné, et c'est seulement s'il constate que la disquette n'a pas changé qu'il va examiner la disquette du lecteur suivant, ceci parce que l'Apple IIGS, contrairement au Macintosh, ne sait pas gérer les événements de type disque et que l'utilisateur peut changer à son insu une disquette dans un lecteur.

• **SFPGetFile** permet de gérer un dialogue semblable au précédent et de récupérer les mêmes informations, mais l'aspect de la fenêtre est défini par l'application. Nous n'entrerons pas dans les détails de cette procédure.

Choix du fichier à écrire

Deux procédures sont à notre disposition pour présenter le dialogue qui permettra à l'utilisateur de choisir le nom du document (et le répertoire) dans lequel il désire enregistrer son travail. La fenêtre standard présente six boutons, permettant de changer de lecteur (équivalent touche Tabulation), de créer un nouveau répertoire (dossier), d'ouvrir ou de fermer un dossier (équivalent touche Escape pour fermer), de confirmer ou d'annuler l'action d'enregistrer (équivalent touche Retour ou Entrée

pour confirmer). La liste qui défile contient le nom de tous les fichiers présents sur la disquette, estompés, ainsi que les noms des dossiers non estompés (on peut les ouvrir par double-clic). Une ligne éditable apparaît en dessous, avec un titre (optionnel) par défaut. Entre les deux, une phrase que contrôle l'application. En haut, on voit en clair le nom du répertoire actif, ainsi que la place disponible sur la disquette.

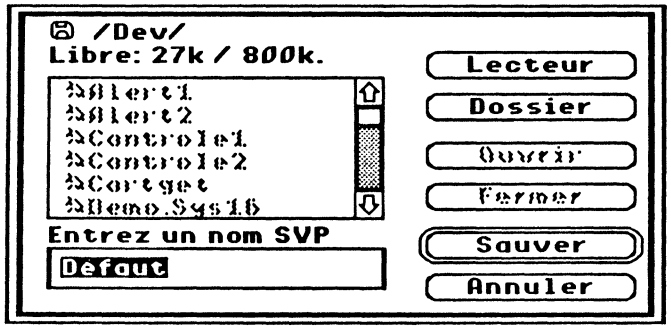


Figure X.3. Dialogue standard SFPutFile.

• **SFPutFile** est l'appel standard. Cette procédure affiche le dialogue standard et permet à l'application de déterminer dans quel fichier l'utilisateur veut écrire ses données. Là encore, six arguments sont nécessaires :

– les deux premiers sont l'abscisse et l'ordonnée en coordonnées globales du coin supérieur gauche de la fenêtre de dialogue, permettant ainsi à l'application de la positionner comme bon lui semble ;

– le troisième argument est un pointeur sur une chaîne de caractères type Pascal qui apparaîtra juste au-dessus du rectangle dans lequel l'utilisateur doit entrer le nom du fichier. Même remarque que plus haut, l'espace étant encore plus restreint ;

– le quatrième argument est un pointeur sur une chaîne de caractères type Pascal qui contiendra le nom par défaut que propose l'application pour le document (la chaîne peut être vide). On veillera à donner un nom conforme à la syntaxe ProDOS ;

– le cinquième argument est un entier contenant le nombre maximal de caractères pouvant être tapés par l'utilisateur. Rappelons que ProDOS tolère 15 caractères au plus, ce nombre ne devra donc pas excéder cette valeur, et il sera nécessairement plus petit si l'application ajoute automatiquement un suffixe au nom de fichier ;

– le sixième argument est un pointeur sur une structure *SFReply*. Comme vu plus haut, le champ *good* permettra de savoir si l'utilisateur a cliqué dans le bouton Enregistrer ou dans le bouton Annuler. Dans le premier cas, le nom complet sera disponible pour assurer l'écriture sur disquette.

Ici, pas de fonction filtre : tous les noms de fichiers sont affichés, ce qui permet à l'utilisateur de ne pas choisir un nom qui existe déjà. La procédure **SFPutFile** agit comme **SFGetFile** en ce qui concerne le répertoire par défaut et la prise en compte du bouton Lecteur.

Remarque La procédure vérifie le nom de fichier saisi par l'utilisateur, et affiche une alerte si l'un des caractères n'est pas accepté par ProDOS. Dans l'illustration, le caractère « é » donné dans le nom par défaut « Défaut » provoquera une telle alerte. La procédure filtrant les caractères au fur et à mesure de leur frappe par l'utilisateur, un mauvais caractère ne peut provenir que d'un mauvais nom par défaut, voire d'un copier-coller.

Si le nom saisi correspond à un fichier qui existe déjà dans le répertoire en cours, une alerte est affichée, demandant confirmation de la destruction du fichier existant et son remplacement par le nouveau fichier à enregistrer.

• **SFPutFile** permet de gérer un dialogue semblable au précédent et de récupérer les mêmes informations, mais l'aspect de la fenêtre est défini par l'application. Là encore, nous n'entrerons pas dans les détails de cette procédure.

Routine de plus

La présentation des noms de fichiers dans les fenêtres de dialogue précédemment décrites se fait par défaut en lettre minuscules, sauf la première et celles qui suivent un point, qui sont en majuscule. Pour changer cette présentation et afficher toutes les lettres en majuscule, on peut utiliser la procédure **SFAllCaps**, en donnant en argument la valeur $8000. Pour rétablir l'option par défaut, on donnera 0 en argument.

Exemple complet

L'exemple suivant montre le fonctionnement de **SFGetFile** et **SFPutFile**. Trois boucles : les deux premières sur **SFGetFile**, l'une sans liste de types, l'autre avec ; la troisième sur **SFPutFile**. Quand un fichier est sélectionné, la fenêtre de dialogue disparaît et le contenu de la structure *SFReply* est affiché en clair, jusqu'au prochain clic. Pour sortir de chaque boucle, choisir Annuler dans le dialogue.

Remarque En aucun cas cet exemple n'essaie de lire ou d'écrire un fichier. Les manipulations n'auront donc aucun effet destructeur sur les disquettes.

```
#include <tools.h>                    /* définition des termes en gras */
#include <entete.h>                   /* définition des termes en italique */

SFReply     reponse;                  /* ce que manipulent les 2 procédures étudiées */
char        liste[ ] = {1, 0xB3};     /* liste de types ProDOS */
Pointer     windPort;                 /* grafport où dessine le Window Manager */

                    /**** PROGRAMME PRINCIPAL ****/

    main( )
{
int     myID;                         /* identifiant de l'application */

    myID = debut_appl(1);             /* initialisations en mode 640 */
    Desktop(5,0x400000FF);            /* le fond d'écran est blanc */
    windPort = GetWMgrPort( );        /* on récupère le Window Manager port */

    do {
        FlushEvents(EveryEvent, 0);           /* plus d'événement en attente */
        SFGetFile(30, 45, "\17Votre choix SVP", 0L, 0L, &reponse);
        Resultat( );                          /* on va écrire ce qu'on a récupéré */
        }
    while (reponse.good);                     /* on boucle tant que Annuler n'est pas choisi */

    do {
        FlushEvents(EveryEvent, 0);           /* plus d'événement en attente */
        SFAllCaps(0x8000);                    /* que des lettres majuscules */
        SFGetFile(30,45,"\17Votre choix SVP",0L,liste,&reponse);
        Resultat( );                          /* on va écrire ce qu'on a récupéré */
        }
    while (reponse.good);                     /* on boucle tant que Annuler n'est pas choisi */

    do {
        FlushEvents(EveryEvent, 0);           /* plus d'événement en attente */
        SFAllCaps(0);                         /* seules les initiales sont en majuscule */
        SFPutFile(30,25,"\21Entrez un nom SVP","\6DEFAUT",15,&reponse);
```

```
        Resultat( );                        /* on va écrire ce qu'on a récupéré */
        }
    while (reponse.good);                   /* on boucle tant que Annuler n'est pas choisi */

    quitter(myID);                          /* au revoir! */
}

        /***** FONCTION RESULTAT: écrit le contenu de la structure SFReply *****/

    Resultat( )
{
char    msg[20];

    if(reponse.good)                        /* un fichier sélectionné? */
        {
        SetPort(windPort);                  /* on écrit directement sur le bureau... */
        MoveTo(10,25); DrawCString("Fichier sélectionné");
        sprintf(msg,"type: %x",reponse.type);       /*... le type du fichier... */
        MoveTo(10,45); DrawCString(msg);
        sprintf(msg,"aux: %x",reponse.auxtype);     /* ...son type auxiliaire... */
        MoveTo(10,60); DrawCString(msg);
        MoveTo(10,75); DrawString(reponse.filename); /* ...le nom du fichier... */
        MoveTo(10,90); DrawString(reponse.fullname); /* ... et rien du tout! */

        while(!Button(0));                  /* on attend un clic souris... */
        Refresh(0L);                        /* ...et on efface l'écran */
        }
}
```

FONT MANAGER

Le Font Manager s'occupe de toute la gestion des polices de caractères, et notamment de la recherche sur disque de la meilleure police à utiliser quand une demande est faite, précisant une famille de polices, une taille et un style. QuickDraw se contente de dessiner des caractères à l'écran, grâce aux informations passées par le Font Manager.

Pour que le Font Manager puisse retrouver les fichiers définissant les polices de caractères, ceux-ci doivent se trouver dans un dossier spécial de la disquette système (*/system/fonts) et posséder le type ProDOS $C8.

Nous n'entrerons pas dans les détails du Font Manager, le but de ce paragraphe étant simplement de donner une façon d'écrire autrement qu'avec le jeu de caractères système, dans une autre taille et un autre style que la taille et le style par défaut.

Nous nous contenterons de dire, c'est que les polices de caractères se rangent dans des grandes catégories : les familles. Quand un développeur crée une police de caractères, il déclare quelque part dans sa définition un identifiant (le numéro de la famille), la taille et le style dans lequel la police est créée.

Quand une application veut utiliser une police, avec une taille et un style, de deux choses l'une : soit la police existe, avec la taille et le style précisés, et tout est parfait, soit elle n'existe pas, et il faut alors se débrouiller autrement. C'est le Font Manager qui se débrouille, en cherchant la police la plus proche correspondant aux caractéristiques spécifiées, essayant de créer des effets de style artificiels à partir du style normal quand le style n'existe pas, essayant de faire des mises à l'échelle à partir de tailles connues quand la taille n'existe pas, et en désespoir de cause prenant la police système si la famille de la police choisie n'existe pas.

Au moment où nous écrivons, le Font Manager n'est pas complètement réalisé. Une routine permettra l'installation automatique des noms de familles de caractères dans un menu déroulant, une autre permettra l'affichage automatique d'une fenêtre de dialogue permettant à l'utilisateur de choisir une famille, une taille et un style, ces caractéristiques étant immédiatement répercutées dans les champs adéquats du grafport courant.

Pour l'heure, contentons-nous de deux routines.

- L'initialisation du Font Manager est réalisée par la procédure **FMStartUp**, qui réclame trois arguments :
  - un pointeur sur une chaîne de caractères Pascal qui précisera le nom de la police système (on passera zéro-long pour accepter le nom par défaut) ;
  - l'identifiant de l'application ;
  - l'adresse d'une page zéro, nécessaire au Font Manager pour son fonctionnement interne.
- Pour fixer dans le grafport une nouvelle police de caractères et l'utiliser avec les procédures **DrawChar**, **DrawString**, **DrawCString** et **DrawText**, on utilisera la procédure **InstallFont**, qui rend caducs les appels à **SetFont**, **SetTextFace** et **SetTextSize**.

InstallFont réclame trois arguments :

- un entier précisant la taille et style choisis, taille dans l'octet haut et le style dans l'octet bas. La taille est un nombre compris entre 1 et 255, et joue sur la hauteur du caractère. Le style est défini par la valeur des bits qui le composent, définition que nous avons déjà vue dans QuickDraw :

bit 0 : gras
bit 1 : *italique*
bit 2 : souligné
bit 3 : relief
bit 4 : ombré

Notons que dans la version actuelle de QuickDraw (1.02), seuls le gras et le souligné sont gérés.

- un entier précisant la famille de caractères désirée (zéro signifie la police système). Logiquement, les mêmes caractères que sur le Macintosh existent, donc le numéro de famille devrait être identique. En voici une liste partielle (les trois dernières familles sont connues de l'imprimante LaserWriter) :

3 : Geneva
5 : Venice
20 : Times
21 : Helvetica
22 : Courrier

- un entier donnant quelques directives à la procédure **InstallFont**. Pour l'instant, seul le bit 0 est défini, et encore ne fonctionne-t-il pas dans la version prototype du Font Manager : s'il est à zéro, la mise à l'échelle est possible, s'il est à un, la mise à l'échelle est impossible. Supposons qu'aucune police correspondant parfaitement aux deux premiers arguments ne soit trouvée. Le Font Manager va choisir à la place la police qui se rapproche le mieux de la demande (en se conformant à un algorithme défini). Si la mise à l'échelle n'est pas permise, il en restera là. Si la mise à l'échelle est permise (et possible), il créera de toute pièce une nouvelle police de caractères à la bonne taille, par extrapolation de la police choisie. L'esthétique du résultat étant souvent médiocre et la mise à l'échelle une opération assez lente, on se contentera le plus souvent d'interdire cette option. Cependant que les champs correspondants du grafport seront correctement mis à jour, même si l'affichage n'est pas en accord avec eux. Rien n'empêche qu'à l'impression, par exemple, l'imprimante LASER connaisse une taille et un style qui n'étaient pas présents sur la disquette système au moment de l'affichage écran !

En résumé, on écrira ceci :

```
char    taille, style;          /* deux entiers sur 8 bits */
int     famille;                /* un entier sur 16 bits */

InstallFont(taille*256+style, famille, 1);      /* choix d'une taille, d'un style et d'une famille */
```

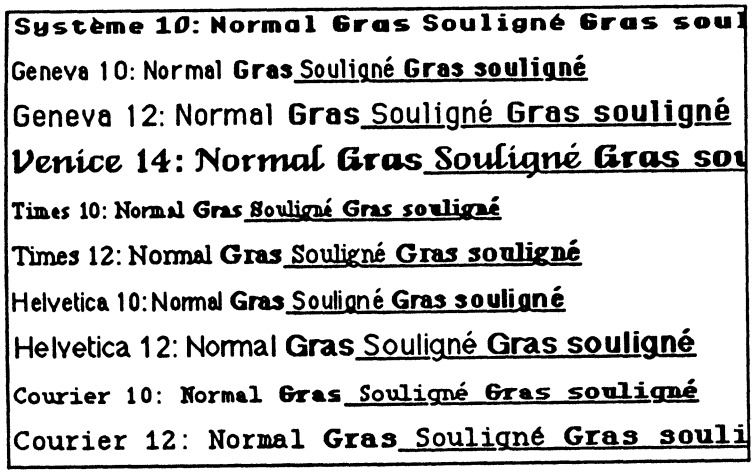


Figure X.4. Quelques caractères différents...

Pour obtenir l'écran de la figure X.4, nous avons utilisé les instructions suivantes :

```
ecrit(10,0,10, "Système 10");
ecrit(10,3,30, "Geneva 10");
ecrit(12,3,50, "Geneva 12");
ecrit(14,5,70, "Venice 14");
ecrit(10,20,90, "Times 10");
ecrit(12,20,110, "Times 12");
ecrit(10,21,130, "Helvetica 10");
ecrit(12,21,150, "Helvetica 12");
ecrit(10,22,170, "Courier 10");
ecrit(12,22,190, "Courier 12");
```

La fonction écrit( ) étant définie de la manière suivante :

```
    ecrit(taille, famille, ligne, str)

int     taille, famille, ligne;
Pointer str;

{
    InstallFont(taille*256+0, famille, 1);
    MoveTo(3,ligne); DrawCString(str); DrawCString(": Normal");
    InstallFont(taille*256+1, famille, 1); DrawCString(" Gras");
    InstallFont(taille*256+4, famille, 1); DrawCString(" Souligné");
    InstallFont(taille*256+5, famille, 1); DrawCString(" Gras souligné");
}
```

On remarquera une nouvelle fois que la police système est définie de telle sorte que ses caractères ne peuvent pas être soulignés.

CHAPITRE XI

TASKMASTER

GÉNÉRALITÉS

Nous avons vu dans les chapitres précédents un certain nombre d'exemples articulés autour de la fonction **GetNextEvent**, qui renseignait l'application sur l'événement à traiter, et qui laissait l'application traiter l'événement. Sur un Macintosh, il n'y a pas d'alternative. Sur l'Apple IIGS, il y en a une : **TaskMaster**.

Comme vous avez pu le constater, tous les exemples d'applications tournent autour de la même architecture : on appelle les mêmes séquences d'instructions pour tester où s'est produit un événement de type *MouseDown*, on fait toujours la même chose quand il s'agit de déplacer une fenêtre, de la redimensionner, de la zoomer, de l'activer, on gère toujours de la même manière les accessoires de bureau, etc. De plus, l'architecture du Window Manager a été conçue de telle sorte que les manipulations les plus compliquées, le défilement des fenêtres, sont pratiquement impossibles à réaliser en dehors de **TaskMaster** (ou alors on programme comme sur Macintosh).

TaskMaster offre au développeur d'applications une facilité déconcertante de programmation. La plupart des événements sont gérés automatiquement par la fonction. Seuls ceux qui sont spécifiques à l'application doivent être traités explicitement. Grâce à **TaskMaster**, le programmeur se concentre sur son travail, et plus sur l'interface utilisateur. Il faut très peu de temps pour obtenir une maquette opérationnelle d'un projet ardu.

De plus, il y a un intérêt évident à utiliser **TaskMaster** : si des améliorations notables interviennent dans le futur concernant tel ou tel manager, tel ou tel aspect de l'interface, il y a gros à parier que les applications écrites à base de **TaskMaster** en bénéficieront sans modification, ou avec très peu de modifications, alors que celles écrites à base de **GetNextEvent** n'auront pas cette chance.

Dans les améliorations souhaitables, ne serait-il pas formidable de gérer automatiquement certains contrôles, ou bien les menus déroulants en zone d'informations, ou bien le double-clic ?

Pourquoi avoir fait dix chapitres avant celui-ci, s'il faut absolument utiliser **TaskMaster**, pouvez-vous demander. Parce que si vous voulez comprendre comment marche cette fonction, il faut avoir compris les chapitres précédents. Cela étant, on peut toujours utiliser **TaskMaster** sans comprendre comment elle fonctionne, et écrire malgré tout d'excellentes applications ! Dans ce cas, il vaut mieux ne pas s'éloigner des sentiers battus...

FONCTIONNEMENT

Structure manipulée

Puisque nous avons utilisé la structure *TaskRec* tout au long de cet ouvrage, nous ne serons pas dépaysés : c'est cette structure, et uniquement elle, qui est en effet utilisée par **TaskMaster**. A la structure d'événement classique (rencontrée sur Macintosh) contenant le type d'événement *(what)*, la donnée additionnelle *(message)*, la date relative *(when)*, le lieu *(where)* et les combinaisons de modification *(modifiers)* viennent s'ajouter deux champs (entiers longs), une autre donnée additionnelle *(TaskData)* et un masque *(TaskMask)*. La structure *TaskRec* a été définie dans le chapitre IV.

Nous avons déjà utilisé *TaskData* dans le Menu Manager. C'est le plus souvent la duplication du champ *message*. Ce champ est là pour laisser intacts les champs remplis par **GetNextEvent**. Si **TaskMaster** doit faire des manipulations sur la donnée additionnelle, elle le fera sur *TaskData*, et non sur *message*. Par exemple, dès que **TaskMaster** appelle **FindWindow**, le pointeur sur fenêtre que cette fonction identifie est placé dans le champ *TaskData*.

Masquer TaskMaster

Le champ *TaskMask* est là pour limiter le champ d'action de **TaskMaster**. Puisque **TaskMaster** manipule lui-même un certain nombre d'événements, il utilise des procédures standard et toutes les options par défaut que nous avons pu rencontrer. Parfois, une application peut vouloir aller au-delà de ces procédures par défaut pour gérer un cas particulier. Faut-il pour cela se passer de **TaskMaster** ? Pas du tout : il suffit de masquer une partie de son utilisation et gérer soi-même cette partie, et laisser **TaskMaster** gérer le reste.

Au moment où nous écrivons ces lignes, 13 des 32 bits du masque sont définis :

- bit 0 : gestion de **MenuKey** ;
- bit 1 : gestion des événements de mise à jour ;
- bit 2 : gestion de **FindWindow** ;
- bit 3 : gestion de **MenuSelect** ;
- bit 4 : gestion de **OpenNDA** ;
- bit 5 : gestion de **SystemClick** ;
- bit 6 : gestion de **DrawWindow** ;
- bit 7 : gestion de **SelectWindow** (quand **FindWindow** retourne *wInContent*) ;
- bit 8 : gestion de **TrackGoAway** ;
- bit 9 : gestion de **TrackZoom** ;
- bit 10 : gestion de **GrowWindow** ;
- bit 11 : gestion du défilement ;
- bit 12 : gestion des articles spéciaux des menus.

Les bits 13 à 31 doivent être à zéro, sinon **TaskMaster** retournera l'erreur $E03 dans *_errno*. Pour les autres bits, la valeur un signifie que **TaskMaster** devra gérer l'action précisée, la valeur zéro que l'application recevra tous les éléments nécessaires à la gestion de cette action, **TaskMaster** ne faisant que traduire les éléments reçus de **GetNextEvent**.

La valeur classique du masque en l'état actuel des choses sera donc $00001FFF, signifiant que l'application laisse **TaskMaster** faire le maximum. Si le masque est égal à zéro, autant appeler **GetNextEvent** directement !

Ce que retourne TaskMaster

TaskMaster s'emploie exactement comme nous avons dit que **GetNextEvent** s'employait dans le chapitre IV. Deux arguments : un masque d'événement et un pointeur sur une structure *TaskRec*. A cela rien d'étonnant, puisque l'une des premières choses que fait **TaskMaster** est justement d'appeler **GetNextEvent**, avec ces arguments. Ne seront donc traités que les événements passés dans le masque donné en premier argument.

Note On ne confondra pas le masque d'événement, argument de **GetNextEvent**, qui sert à sélectionner dans la file d'événements ceux qu'on veut traiter, et le masque *TaskMask*, qui sert à masquer certaines des fonctionnalités du **TaskMaster**.

TaskMaster est une fonction : elle va retourner un entier, un code que l'application doit prendre en compte pour répondre aux événements qu'elle doit gérer. Si **TaskMaster** retourne la valeur zéro, c'est parfait : soit il n'y avait plus d'événement à traiter, soit elle l'a complètement traité. Dans les deux cas, l'application n'a plus rien à faire.

Les autres valeurs susceptibles d'être retournées sont soit des types d'événements, soit un code résultant de **FindWindow**, soit des codes créés de toute pièce par **TaskMaster**.

TaskMaster retourne :

- *MouseDown* si le bit 2 de *TaskMaster* est à zéro. Dans ce cas, l'application appelle elle-même **FindWindow**, si nécessaire, et **TaskMaster** est de peu d'utilité !
- *MouseUp* chaque fois que cet événement intervient, puisqu'elle ne le traite pas. L'application peut ou non en tenir compte (par exemple pour gérer les double-clics).
- *KeyDown* dans les cas suivants : le bit 0 de *TaskMask* est à zéro, ou bien **MenuKey** a été appelée mais n'a rencontré aucun équivalent clavier dans la barre de menus système. A l'application de gérer le caractère, comme elle le faisait dans les chapitres précédents (et notamment dans le chapitre VIII sur Line Edit).
Remarque Aucun caractère n'est retourné quand une fenêtre système est au premier plan. Il semble préférable de récupérer le caractère tapé dans le champ *message* de l'événement plutôt que dans le champ *TaskData*, qui est incorrect quand le bit 0 n'est pas nul.
- *AutoKey* chaque fois que cet événement intervient, puisqu'elle ne le traite pas. A l'application de gérer le caractère en répétition.
Remarque MenuKey n'est pas appelée pour un tel événement, la valeur du bit 0 est donc indifférente.
- *UpdateEvt* dans les cas suivants : le bit 1 de *TaskMask* est à zéro, ou bien la fenêtre a été créée sans donner l'adresse d'une procédure automatique de dessin de son contenu (champ *wContDefProc* de la *ParamList*). Une telle fenêtre ne peut pas posséder de barre de défilement. L'application doit alors dessiner elle-même le contenu de la fenêtre dont le pointeur est précisé dans *TaskData*, entre l'appel aux routines **BeginUpdate** et **EndUpdate**.
- *ActivateEvt* chaque fois que cet événement intervient, puisqu'elle ne le traite pas. A l'application de gérer (ou de ne pas gérer) un tel événement, le pointeur sur la fenêtre incriminée se trouvant dans *TaskData*.
- *DeskAccEvt* chaque fois que cet événement intervient, c'est-à-dire quand l'utilisateur revient à l'application après avoir utilisé un accessoire de bureau classique. L'application n'a rien à faire.
- *wInDesk* chaque fois que **FindWindow** retourne cette valeur, puisque cette occurrence n'est pas traitée. L'application fera comme elle fait d'habitude, vraisemblablement rien.
- *wInMenuBar* dans les cas suivants :
  - le bit 3 de *TaskMask* est à zéro et l'utilisateur a cliqué dans la barre de menus système ;

– **MenuSelect** a été appelée, un article a été sélectionné et doit être géré par l'application (son identifiant est supérieur à 255) ;
– **MenuKey** a été appelée, un article a été sélectionné et doit être géré par l'application (son identifiant est supérieur à 255).

Dans tous les cas, *TaskData* contient l'identifiant de l'article (mot-bas) et l'identifiant du menu (mot-haut) auxquels l'application doit répondre.

- *wInContent* dans les cas suivants :
  – si le bit 7 de *TaskMask* est à zéro, dès que **FindWindow** retourne *wInContent* (la fenêtre peut donc être ailleurs qu'au premier plan, *TaskData* contient le pointeur qui la désigne) ;
  – idem si le bit 7 est à un, mais la fenêtre est alors forcément la fenêtre du premier plan (active).

Quand le bit 7 est à un, un clic dans le contenu d'une fenêtre non active provoque l'appel automatique de **SelectWindow**. Si le bit *F_QCONTENT* du champ *wFrame* de la fenêtre est à un, **TaskMaster** retourne malgré tout *wInContent,* de telle sorte que le clic qui a servi à activer la fenêtre sert encore une fois.

- *wInDrag* si le bit 6 de *TaskMask* est à zéro. Sinon, **DragWindow** est appelée avec des arguments par défaut (grille à 4 dans le mode 320, à 8 dans le mode 640, distance de grâce à 8, le rectangle frontière étant fixé à la totalité du desktop disponible, moins quatre pixels de chaque côté). La fenêtre est sélectionnée si nécessaire (à moins que la touche Pomme ne soit enfoncée au moment du clic souris).

- *wInGrow* si le bit 10 de *TaskMask* est à zéro. Sinon, **GrowWindow** est appelée avec des arguments par défaut pour la taille minimale autorisée pour la région contenu (largeur 100, hauteur 40 en mode 320 comme en mode 640).

- *wInGoAway* dans les cas suivants :
  – le bit 8 de *TaskMask* est à zéro (il est alors de la responsabilité de l'application d'appeler la fonction **TrackGoAway**) ;
  – le bit 8 de *TaskMask* est à un, et **TrackGoAway**, appelée par **TaskMaster**, a retourné TRUE. Dans ce cas, l'application doit fermer la fenêtre dont le pointeur se trouve dans *TaskData.*

On constate que dans tous les cas l'application doit agir : **TaskMaster** ne peut prendre la responsabilité de fermer une fenêtre, puisque l'application est susceptible de provoquer quelques actions au préalable. Ceci ne concerne pas les fenêtres système.

- *wInZoom* si le bit 9 de *TaskMask* est à zéro. Sinon, **TrackZoom** est appelée, et si elle retourne TRUE, **ZoomWindow** est appelée à son tour.

- *wInInfo* chaque fois qu'un clic souris intervient dans la zone d'informations de la fenêtre de premier plan, puisque cette occurrence n'est pas traitée. *TaskData* contient le pointeur sur la fenêtre à laquelle la zone d'informations appartient.
Remarque Quand on clique dans la zone d'informations d'une fenêtre qui n'est pas au premier plan, cette fenêtre est activée quel que soit l'état du bit 7 de *TaskMask.*

wInSpecial quand l'application doit gérer un article de menu dont l'identifiant est compris entre 250 et 255 inclusivement, soit parce qu'il s'applique à une fenêtre de l'application, soit parce qu'il s'applique à une fenêtre système alors que le bit 12 de *TaskMask* est à zéro, soit parce qu'un appel automatique à la fonction **SystemEdit** s'est soldé par un échec. Rappelons que les articles spéciaux sont Annuler (250), Couper (251), Copier (252), Coller (253), Effacer (254) et Fermer (255).

- *wInDeskItem* quand l'application doit gérer un article de menu dont l'identifiant est strictement inférieur à 250 (le bit 4 de *TaskMask* est à zéro). L'application peut appeler elle-même **OpenNDA** dans ce cas.

- *wInFrame* si et seulement si le bit 11 de *TaskMask* est à zéro. Dans ce cas, le clic peut avoir eu lieu dans une barre de défilement, dans la partie gauche du cadre de la fenêtre ou dans la barre de titre si la fenêtre n'a pas le droit de se déplacer. (Si le bit 11 est à un, cliquer ailleurs que dans une barre de défilement est complètement ignoré par **TaskMaster**, qui ne retourne même pas l'événement.)

Si le bit 11 est à zéro et si l'utilisateur clique *in frame* dans une fenêtre qui n'est pas active, celle-ci n'est pas activée automatiquement. Si la fenêtre est active et que le clic s'est produit dans une barre de défilement, **TaskMaster** gère le défilement du contenu de la fenêtre, en accord avec les diverses valeurs passées dans les champs *wScrollVer, wScrollHor, wLageVer, wPageHor* au moment de la création de la fenêtre ou fixés par la suite.

Quand l'application reçoit *wInFrame,* elle ne fait généralement rien.

TaskMaster peut retourner une valeur négative quand le bit 5 de *TaskMask* est à zéro. Dans ce cas, c'est à l'application de gérer l'accessoire de bureau qui se trouve au premier plan, en appelant **SystemClick**.

Deux constantes nouvelles sont apparues dans cette liste. Nous pouvons les définir ainsi :

```
#define wInSpecial      25
#define wInDeskItem     26
```

Sauf cas très particulier, le masque *TaskMask* sera fixé au maximum de ses possibilités, c'est-à-dire à $00001FFF dans la version 1.03 du Window Manager. Si par exemple une application décide de ne pas supporter les accessoires de bureau, inutile de toucher au masque ! Il suffit de ne pas inclure l'instruction **FixAppleMenu**. Il serait toutefois regrettable de ne pas profiter des accessoires de bureau, puisqu'ils sont complètement gérés par **TaskMaster** si quelques règles simples sont respectées (concernant l'identifiant des articles spéciaux, notamment).

On masquera **TaskMaster** là où les options par défaut qu'elle utilise ne s'accordent pas avec l'application (**DragWindow, GrowWindow**...).

Avec un masque maximal, l'application ne reçoit plus que les codes suivants :

– *MouseUp :* l'application a la responsabilité de gérer (ou de ne pas gérer) ces événements ;

– *KeyDown* ou *AutoKey :* l'application doit gérer le caractère reçu (champ *message*), elle est assurée que la fenêtre de premier plan lui appartient et qu'il ne s'agit pas d'un équivalent-clavier d'un menu déroulant. Elle ignorera vraisemblablement cet événement si elle n'utilise pas Line Edit ;

– *UpdateEvt :* si la procédure automatique de dessin du contenu de la fenêtre existe, l'application ne reçoit même pas cet événement (ce sera le cas si la fenêtre possède des barres de défilement). Sinon, elle y répond classiquement ;

– *ActivateEvt :* l'application a la responsabilité de gérer (ou de ne pas gérer) ces événements ;

– *wInDesk :* l'application a la responsabilité de gérer (ou de ne pas gérer) un clic souris dans le desktop ;

– *wInMenuBar :* un article est sélectionné dans un menu déroulant, soit par la souris, soit par équivalent clavier. Ni l'article ni le menu ne peuvent avoir un code nul, puisqu'on est sûr qu'une sélection valide a été faite. L'article a un code supérieur à 255. L'application doit répondre à la commande, avec une fonction de type ExecMenu par exemple ;

– *wInContent :* un clic souris a eu lieu dans la région contenu d'une fenêtre de l'application et on est sûr que cette fenêtre est active. L'application répond à cet événement comme elle l'entend ;

– *wInGoAway :* l'utilisateur désire fermer une fenêtre (qui appartient forcément à l'application). Il a cliqué dans la case de fermeture et relâché le bouton dans cette même case. Il est de la responsabilité de l'application de faire le nécessaire ;

– *wInInfo :* un clic souris a eu lieu dans la zone d'informations d'une fenêtre de l'application et on est sûr que cette fenêtre est active. L'application répond à cet événement comme elle l'entend ;

– *wInSpecial :* un article spécial concernant l'application a été sélectionné. C'est soit la demande de fermeture d'une fenêtre de l'application, soit un Annuler, un

Couper, un Copier, un Coller ou un Effacer. Si l'application ne gère pas le copier-coller, ces articles devraient être estompés quand une fenêtre de l'application est sélectionnée, de telle sorte que l'utilisateur comprenne qu'il ne peut pas s'en servir.

L'application doit répondre à la commande, avec une fonction de type ExecMenu par exemple. Rien n'empêche de partager ExecMenu entre les réponses à *wInMenuBar* et *wInSpecial*.

Nous ne l'avons pas dit, mais cela coule de source : quand **TaskMaster** reçoit un événement peu courant, tels les événements définis directement par l'application, qu'elle ne sait pas gérer, elle renvoie en code le type de l'événement. A l'application de faire le nécessaire ! Notons également que **TaskMaster** appelle elle-même **SystemTask** pour assurer le bon fonctionnement des accessoires de bureau périodiques.

Défilement du contenu d'une fenêtre

Ce n'est pas parce que **TaskMaster** gère automatiquement le défilement du contenu d'une fenêtre qu'il faut « mourir idiot » et ne pas comprendre ce qui se passe. De plus, certaines applications doivent forcer le défilement, par exemple quand l'utilisateur fait glisser la souris en dehors de la fenêtre (cas des applications graphiques, ou des tableurs). C'est pourquoi nous allons essayer de voir le fonctionnement du mécanisme de défilement.

L'écran définit un système de coordonnées, dites coordonnées globales. L'origine en est le coin supérieur gauche de l'écran. Quand on définit une fenêtre, on en donne la région contenu, qui, pour une fenêtre normale, est un rectangle. Les coordonnées de ce rectangle (globales) définissent la taille et la position de la fenêtre à l'écran, et autour de ce rectangle sera dessinée la région contour. Ce rectangle définit lui aussi un système de coordonnées, dites coordonnées locales. L'origine en est le coin supérieur gauche du rectangle.

L'écran est vu comme le rectangle frontière *(BoundsRect)* d'une pixel image (la mémoire écran), la région contenu de la fenêtre est vue comme le rectangle *PortRect* d'un grafport dont la pixel image associée est l'écran.

Dans la région contenu de la fenêtre, une image doit être affichée. Si elle est plus grande, elle n'est pas visible en totalité, mais elle ne déborde pas. Considérons une feuille de papier imaginaire qui aurait la taille de l'image complète, et une très grande feuille de carton non transparent au milieu de laquelle un trou rectangulaire de la taille du contenu de la fenêtre aurait été fait (la lucarne). Si nous plaçons la feuille de carton au-dessus de la feuille de papier, nous ne voyons plus qu'une partie de l'image à travers la lucarne : c'est cette partie visible qui est reportée à l'écran (figure XI.1.a).

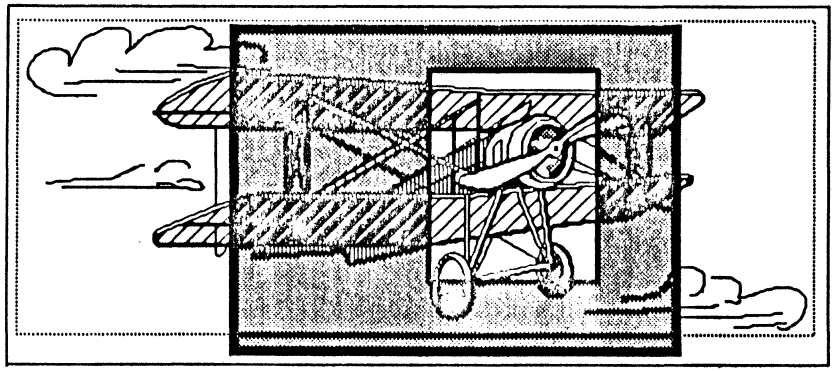


Figure XI.1.a. L'état initial. Le contour pointillé représente la feuille de dessin, le rectangle grisé l'écran et la lucarne le contenu de la fenêtre.

Si nous déplaçons solidairement la feuille de papier et la feuille de carton, la même partie de l'image reste visible dans la lucarne, mais la lucarne a changé de position. C'est le déplacement d'une fenêtre (figure XI.1.b).

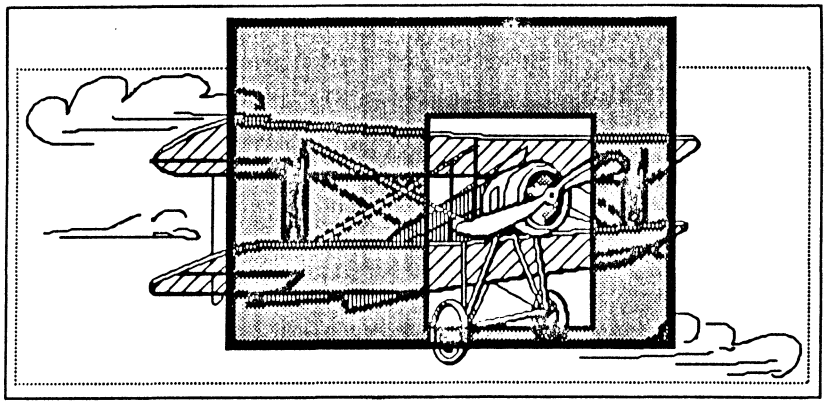


Figure XI.1.b. Déplacement de la fenêtre par rapport à l'état initial.

Si nous déplaçons la feuille de papier sous la feuille de carton (soit horizontalement, soit verticalement), cette dernière restant immobile, nous ne voyons plus la même partie de l'image : celle-ci défile à travers la lucarne. Le défilement traduit donc une translation horizontale ou verticale du dessin par rapport à la lucarne, mais la lucarne n'a pas changé de position (figure XI.1.c).

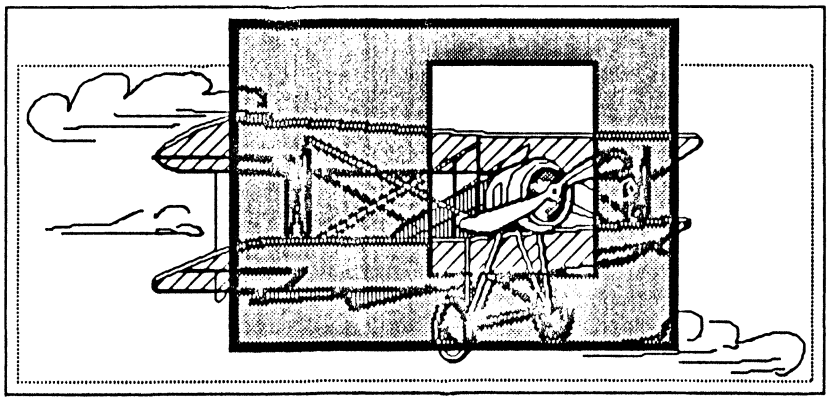


Figure XI.1.c. Défilement de la fenêtre par rapport à l'état initial.

Notre lucarne définit le système de coordonnées locales, origine en haut à gauche. La feuille de papier possède elle-même son propre système de coordonnées relatives, origine en haut à gauche. A un instant précis, un point et un seul de la feuille de papier coïncide avec le point origine du système de coordonnées locales. Pour faire action de défilement, il suffit donc de dire que le point de coïncidence a changé, donc qu'on fixe une nouvelle origine.

QuickDraw offre une telle procédure : **SetOrigin**. Elle fait coïncider l'origine des coordonnées locales avec le point défini dans le système de coordonnées relatives (qui n'est autre qu'un deuxième système de coordonnées globales), mais ne génère aucun événement de mise à jour. On peut accompagner cet appel de **ScrollRect**, par exemple, et de **InvalRgn** pour générer le dessin de la partie apparue. Il y a

malheureusement une difficulté supplémentaire à prendre en compte : quand le Window Manager est invoqué, l'origine de chaque fenêtre doit obligatoirement être le point (0,0) !

Voici donc comment le défilement est effectué (au travers de la procédure de mise à jour passée à la création de la fenêtre) :

Tout d'abord, **TaskMaster** fixe le bon grafport.

Après, la procédure QuickDraw **SetOrigin** est appelée pour fixer la bonne origine.

Ensuite, on peut dessiner dans le système de coordonnées relatives (notre feuille de dessin), sans se préoccuper de ce qui sera visible ou pas, puisque ce dessin est effectué durant un événement de mise à jour. C'est le rôle de notre procédure de dessin du contenu de la fenêtre.

Dès que toute la région visible a été redessinée, l'origine est rétablie à (0,0) par **SetOrigin**, mais la véritable origine est conservée dans un coin de la mémoire.

Enfin, le grafport précédent est restauré.

Cette complication est à l'origine de nombreux malentendus. Nous avons appelé système de coordonnées locales le système dont l'origine est au coin supérieur gauche du contenu de la fenêtre, et système de coordonnées relatives celui dont l'origine est le coin supérieur gauche de la feuille de dessin. Certains ne feront pas la différence. Peu importe. Ce qui est primordial, c'est de dessiner dans le système de coordonnées relatives alors que l'origine est correctement fixée, et de remettre l'origine à zéro dès qu'on a terminé.

L'ambiguïté provient du fait qu'il y a deux sytèmes de coordonnées globales pour un système de coordonnées locales. Quand on dessine sur une feuille de papier, cette feuille est réellement une pixel image dont les frontières *(BoundsRect)* définissent un système global. La lucarne (le *PortRect* du grafport) ne définit qu'un système local. Quand on dessine dans la mémoire écran, l'écran définit un système global, et chaque fenêtre un système local. Il suffit de ne pas se perdre dans ces subtilités pour faire de jolis défilements !

Quand l'application voudra dessiner dans une fenêtre qui contient des barres de défilement en dehors d'un événement de mise à jour (cas des applications graphiques, par exemple), cette règle devra absolument être respectée. C'est pourquoi le Window Manager met à notre disposition une procédure dédiée, **StartDrawing**. On donne en argument un pointeur sur la fenêtre dans laquelle on veut dessiner, et la procédure se charge de fixer le bon grafport et la bonne origine. Dans le cas où une fenêtre ne comporte pas de barre de défilement, la procédure **SetPort** suffit. Dès que l'action de dessiner sera terminée, il faudra rétablir l'origine à (0,0), avec toujours la procédure QuickDraw **SetOrigin.**

Attention à l'emploi des procédures **GetMouse**, **LocalToGlobal** et **GlobalToLocal**, et en règle générale à tout ce qui touche aux coordonnées. La séquence :

```
SetPort(wind);
GetMouse(&pt);
```

donnera un point dans le système de coordonnées locales de la fenêtre wind, alors que la séquence :

```
StartDrawing(wind);
GetMouse(&pt);
SetOrigin(0,0);
```

donnera un point dans le système de coordonnées relatives de la fenêtre wind. Voir l'exemple en fin de chapitre.

- Le Window Manager offre deux routines dont nous n'avons pas encore parlé : **GetCOrigin** et **SetCOrigin**. La fonction **GetCOrigin** retourne dans un entier long les

coordonnées relatives du point qui coïncide avec le coin supérieur gauche du rectangle contenu de la fenêtre dont un pointeur est passé en argument (abscisse dans le mot haut, ordonnée dans le mot bas). La procédure **SetCOrigin** permet de redessiner le contenu de la fenêtre en accord avec l'origine passée en argument. Les barres sont réajustées en conséquence, et l'origine est remise à (0,0). Seul inconvénient, aucun événement de mise à jour n'est généré. C'est donc à l'application de le faire (vous souvenez-vous de la procédure **InvalRect** ?). Fabuleux pour forcer un défilement alors que l'utilisateur n'est pas en train de manipuler les barres de défilement ! On peut même faire défiler une fenêtre non active.

Une précaution à prendre, toutefois, vérifier que l'origine est un point possible, en fonction de la taille du dessin et de celle du contenu de la fenêtre. Le Window Manager ne se posera pas de questions : il fera ses calculs, et si l'origine est mauvaise, il risque de dessiner les curseurs de défilement en dehors des barres de défilement !

Le bout d'exemple suivant assume qu'on travaille avec deux fenêtres suffisamment grandes pour que l'origine fixée reste dans des limites convenables. L'une est active, l'autre pas, elles possèdent toutes deux, deux barres de défilement et obligatoirement une procédure automatique de dessin. On va faire défiler leur contenu simultanément et en diagonale, de cinq pixels chaque fois, et revenir à la position initiale, l'utilisateur restant simple spectateur !

On remarquera comment les événements de mise à jour sont générés, et surtout comment ils sont traités au sein de la boucle : on appelle **TaskMaster** avec un masque d'événement limitant son fonctionnement aux seuls événements de mise à jour, tant que de tels événements sont disponibles. C'est **TaskMaster** qui appelle la procédure de dessin du contenu de la fenêtre !
Note Cette fonction pourra être intégrée dans l'exemple en fin de chapitre.

```
    defilauto( )
{
int     i;
Rect    r;

    SetRect(&r, 0, 0, 1000, 1000);      /* on fixe un rectangle arbitrairement grand */
    for (i=0; i<40; ++i)
        {
        SetCOrigin(5*i, 5*i, fen1);     /* origine changée, barres redessinées */
        SetPort(fen1); InvalRect(&r);   /* contenu de la fenêtre 1 entièrement invalidé */
        SetCOrigin(5*i, 5*i, fen2);             /* idem fenêtre 2 */
        SetPort(fen2); InvalRect(&r);
        while (EventAvail(UpdateMask, &tache)) TaskMaster(UpdateMask, &tache);
        }
    for (i=40; i>=0; i--)               /* idem en sens inverse */
        {
        SetCOrigin(5*i, 5*i, fen1);
        SetPort(fen1); InvalRect(&r);
        SetCOrigin(5*i, 5*i, fen2);
        SetPort(fen2); InvalRect(&r);
        while (EventAvail(UpdateMask, &tache)) TaskMaster(UpdateMask, &tache);
        }
}
```

Une autre solution était de dessiner soi-même, en employant la séquence suivante (où Paint est la procédure qui dessine le contenu des fenêtres) :

```
SetCOrigin(5*i, 5*i, fen1);         /* on fixe une nouvelle origine */
StartDrawing(fen1);                 /* on fixe le bon grafport, avec la bonne origine */
Paint( );                           /* on dessine */
SetOrigin(0,0);                     /* on rétablit (0,0) quand on a terminé */
```

• En mode 640 (et uniquement dans ce mode-là), nous savons que la couleur d'un pixel dépend de sa position horizontale à l'écran (voir le chapitre III). Supposons que nous nous amusions à créer des patterns de couleurs pour définir des teintes plus agréables que les teintes par défaut dans ce mode. Ces patterns n'auront pas la même apparence si les pixels qui les utilisent sont alignés sur des frontières de mots ou pas.

Résultat : quand on déplace une fenêtre ou quand on fait défiler son contenu, le dessin à l'intérieur risque de changer de couleur inexpliquablement ! (Voir l'exemple avec les formes en fin de chapitre VI.) Pour éviter ce genre de désagréments, le Window Manager nous offre une procédure, **SetOrgnMask**. Elle n'a aucun effet sur le déplacement d'une fenêtre, puisqu'elle agit uniquement sur la composante horizontale de l'origine : en cas de défilement, l'origine sera forcée à rester sur certaines frontières. Le premier argument de **SetOrgnMask** agit comme un masque : un « et logique » est effectué entre ce masque et la coordonnée horizontale de l'origine, le second argument désigne la fenêtre. La valeur par défaut du masque est $FFFF, donc le masque est neutre, l'origine n'est pas affectée. Pour forcer une abscisse de l'origine paire (dernier bit forcé à 0), on utilisera $FFFE ; pour une abscisse multiple de 4 (les deux derniers bits forcés à 0), le masque sera $FFFC ; pour une abscisse multiple de 8 (les 3 derniers bits forcés à 0, ce qui garantit l'intégrité de tous les patterns, puisqu'un pattern est défini sur huit pixels horizontaux), le masque sera $FFF8. Inutile d'aller plus loin !

La séquence suivante (où wind désigne une fenêtre avec barres de défilement) :

```
SetOrgnMask(0xFFF8, wind);
SetCOrigin(53, 31, wind);
```

forcera l'origine à (48,31), garantissant une couleur conforme à tous les patterns. Un exemple concret d'utilisation est donné en fin de chapitre.

Procédure de mise à jour du contenu d'une fenêtre

Dès qu'une fenêtre possède des barres de défilement, elle est obligée de posséder une procédure qui dessine le contenu de la fenêtre, de la même manière que dès qu'une fenêtre possède une zone d'informations, elle est obligée de posséder une procédure qui dessine le contenu de cette zone. L'analogie est parfaite, à la différence que la procédure de mise à jour ne doit recevoir aucun argument.

Par l'intermédiaire de cette procédure, l'application va faire son dessin, complet, dans le système de coordonnées relatives. Le Window Manager appellera cette procédure dans deux cas précis : l'utilisateur a cliqué dans une barre de défilement et le contenu de la fenêtre doit défiler, ou bien une partie de la fenêtre vient d'être rendue visible et doit être rafraîchie. Dans les deux cas, il s'agit d'une mise à jour, dans les deux cas le Window Manager va opérer comme expliqué plus haut : il fixe le bon grafport, il fixe la bonne origine, il utilise la procédure pour dessiner la partie à mettre à jour (la région invalide), il rétablit l'origine à (0,0) et il restitue le grafport précédent.

Comme avec la procédure de dessin de la zone d'informations, on ne peut pas faire n'importe quoi, car l'environnement au moment de l'appel de ces procédures est propre au Window Manager, et l'application n'est plus capable de retrouver ses propres variables directement (problème déjà évoqué de page directe et de data bank register).

Une façon d'agir pourra être la suivante, en attendant des compilateurs C capables de résoudre ce problème : on fait une procédure qui ne comporte qu'une instruction, un appel à une fonction C qui fera réellement le dessin. Par exemple, si Paint1 et Paint2 sont les procédures de mise à jour de deux fenêtres dont le contenu est une picture QuickDraw repérée par un handle (disons pict1 et pict2), on pourra écrire quelque chose du style (ce n'est qu'un exemple) :


```
Handle pict1, pict2;          /* ce sont des variables globales */

    pascal void Paint1( )     /* procédure fenêtre 1 */
{
dessine(&pict1);
}

    pascal void Paint2( )     /* procédure fenêtre 2 */
{
dessine(&pict2);
}

    dessine(phdl)             /* procédure commune */

Handle *phdl;

{
Rect    r;

    SetRect(&r,...);
    DrawPicture(*phdl, &r);
}
```

DEUX EXEMPLES COMPLETS

Affichage des coordonnées (nouvelle version)

Nous avons vu en fin de chapitre V un exemple d'application où les coordonnées locales et globales du pointeur étaient affichées en permanence dans la barre de menus. Nous reprenons cet exemple avec quelques variantes, et surtout avec la gestion du défilement par **TaskMaster**. Nous affichons cette fois les trois sytèmes de coordonnées : coordonnées globales (origine en haut à gauche de l'écran), coordonnées relatives (origine en haut à gauche du dessin) et coordonnées locales (origine en haut à gauche de la région contenu de la fenêtre). On remarquera l'emploi de **StartDrawing** et de **SetOrigin**, routines sur lesquelles s'articule tout l'exemple.

Le contenu de chaque fenêtre étant la juxtaposition de trois rangées de cinq carrés de 100 pixels de côté, tous de couleurs différentes, il est facile de repérer les coordonnées relatives ! De plus, le défilement est fixé à 10 pixels pour les flèches, 100 pixels pour les bandes.

Par dessin direct, nous affichons en permanence dans la zone d'informations de la fenêtre de premier plan la valeur retournée par **GetCOrigin**, qui est donc le point en coordonnées relatives qui correpond à l'origine des coordonnées locales. Puisque la procédure appelée par le Window Manager pour dessiner automatiquement la zone d'informations ne fait rien, il n'y a aucune chance de voir apparaître le moindre message dans la fenêtre au second plan !

Seule différence entre les deux fenêtres : le bit *F_FLEX* du champ *wFrame* est à zéro dans la fenêtre 1, et à un dans la fenêtre 2. Voyez-vous la différence ? Tentez l'expérience suivante : pour chaque fenêtre, faites défiler jusqu'à visualiser le coin inférieur droit du dessin (les deux barres de défilement sont donc en position maximale), puis agrandissez la fenêtre. Dans le cas de la fenêtre 1, le dessin est refait proprement. Dans le cas de la fenêtre 2, le dessin n'est pas refait, mais la zone des données est agrandie, laissant des bandes blanches à droite et en bas du dessin... bandes qu'on ne peut plus faire disparaître. C'est cela, la flexibilité des données !

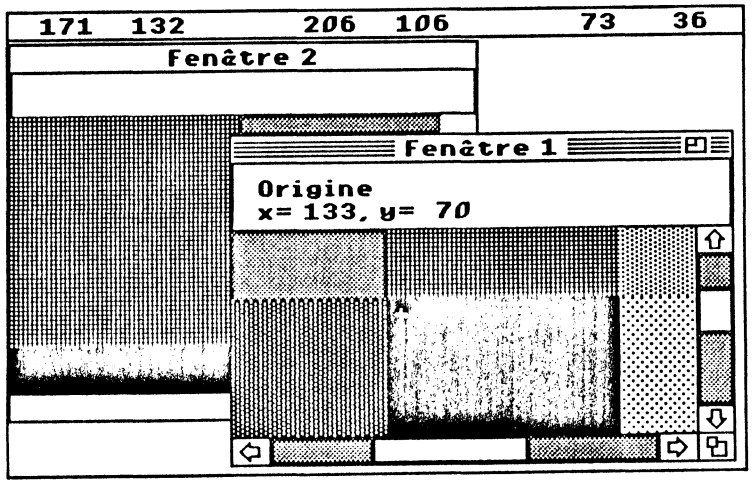


Figure XI.2. L'écran de l'exemple.

```
#define mode 0                          /* 0 si mode 320, 1 si mode 640 */

#include <tools.h>                      /* définition des termes en gras */
#include <entete.h>                     /* définition des termes en italique */

void    Info( );                        /* dessin des zones d'information */
void    Paint( );                       /* dessin des régions contenu */
int     colfen[ ] = {0,0x0F00,0x020F,0xF0F0,0x00F0};        /* couleur des fenêtres */

ParamList  maFen1 = {
        sizeof(ParamList), 0x9DB0, "\13 Fenêtre 1 ", 1L,
        {56, 2,187, 302+320*mode}, colfen, 0, 0,
        300, 500, 131, 300+320*mode,
        10, 10, 100, 100,
        0L, 30, 0L, Info, Paint,
        {60,20,130, 196+320*mode}, -1L, 0L };

ParamList  maFen2 = {
        sizeof(ParamList), 0x9FB0, "\13 Fenêtre 2 ", 2L,
        {46, 2,187, 302+320*mode}, colfen, 100, 100,
        300, 500, 141, 300+320*mode,
        10, 10, 100, 100,
        0L, 20, 0L, Info, Paint,
        {80,40,145, 216+320*mode}, -1L, 0L };
                                                /* définition d'un curseur */
char croix[ ] = { 5,0,3,0,                      /* 5 lignes de 3 mots */
                  0,0xF0,0,0,0,0,               /* image */
                  0,0xF0,0,0,0,0,
                  0xFF,0xFF,0xF0,0,0,0,
                  0,0xF0,0,0,0,0,
                  0,0xF0,0,0,0,0,
                  0,0,0,0,0,0,                  /* début du masque */
                  0,0,0,0,0,0,
                  0,0,0,0,0,0,
                  0,0,0,0,0,0,
                  0,0,0,0,0,0,
                  2,0,2,0 };                            /* point chaud */

TaskRec    tache;                               /* ce que manipule TaskMaster */
```



```
Pointer     fen1, fen2, wind;                   /* pointeurs sur fenêtre */
int         indic = TRUE;                       /* indicateur de fin de boucle */
Pointer     menuPort;                           /* Menu Manager port */
Handle      cont;                               /* handle sur région contenu */
Pointer     arrow;                              /* le curseur en forme de flèche */

                    /***** PROGRAMME PRINCIPAL *****/

main( )
{
int     code;                                   /* code retourné par TaskMaster */
int     myID;                                   /* identifiant de l'application */

    myID = debut_appl(mode);                    /* début standard */
    fen1 = NewWindow(&maFen1);                  /* ouverture fenêtre 1 */
    fen2 = NewWindow(&maFen2);                  /* ouverture fenêtre 2 */
    menuPort = GetMenuMgrPort( );               /* on récupère le menu manager port */
    SetPort(menuPort);
    SetTextMode(0);                             /* on lui impose le mode Copy */
    arrow = GetCursorAdr( );                    /* on garde l'adresse du curseur système */
    tache.TaskMask = 0x00001FFF;                /* on fixe le masque maximal à TaskMaster */
    FlushEvents(EveryEvent, 0);         /* on supprime tous les événements en attente */

/* defilauto( )   on peut appeler ici cette fonction définie plus haut,
                  pour assister à du défilement automatique, si on le désire */

    do {
        code = TaskMaster(EveryEvent, &tache);  /* traitement de l'événement suivant */
        AffichCoord( );                         /* affichage des coordonnées */
        AjusteCurs( );                          /* dessin du curseur ajusté */
        if(!code) continue;                     /* pas d'événement à traiter */

        switch(code)
            {
            case KeyDown :                      /* une touche enfoncée... */
                indic = FALSE;                  /* ...on quitte l'application */
                break;

            case ActivateEvt :          /* un événement d'activation ou de désactivation */
                if(tache.modifiers & ActiveFlag)        /* si activation... */

                cont = GetContRgn(tache.TaskData); /* ...on calcule la région contenu... */
                break;                          /* ...de la fenêtre activée */
            }
        }
    while(indic);

    quitter(myID);
}

/***** FONCTION AFFICHCOORD: affiche les coordonnées souris
                                                dans trois systèmes différents *****/

    AffichCoord( )
{
Point   pt, pt2;                        /* deux points */
char    msg[20];
Pointer port;                           /* la fenêtre active */
long    orig;                           /* équivalent point */
Rect    r;                              /* un rectangle */

port = FrontWindow( );              /* on mémorise la fenêtre active */
orig = GetCOrigIn(port);            /* on récupère la bonne origine */
```

```
    StartInfoDrawing(&r, port);          /* on va écrire dans la zone d'information */
    MoveTo(10, r.bottom - 13); DrawCString("Origine");
    sprintf(msg, "x=%4d, y=%4d    ", getbits(orig,31,16), getbits(orig,15,16));
    MoveTo(10, r.bottom - 3); DrawCString(msg);
    EndInfoDrawing( );                   /* on a fini d'écrire dans la zone d'information */

    StartDrawing(port);                  /* on fixe le bon grafport et la bonne origine */
    GetMouse(&pt2);                      /* le point en coordonnées relatives */
    SetOrigin(0, 0);                     /* on rétablit l'origine en (0,0) */
    GetMouse(&pt);                       /* le point en coordonnées locales */
    SetPort(menuPort);                   /* on va écrire dans la barre des menus... */
    sprintf(msg, "%4d ", pt.H);
    MoveTo(240,10); DrawCString(msg);
    sprintf(msg, "%4d ", pt.V);                  /* ...les coordonnées locales */
    MoveTo(280,10); DrawCString(msg);

    sprintf(msg, "%4d ", pt2.H);
    MoveTo(125,10); DrawCString(msg);
    sprintf(msg," %4d ", pt2.V);                 /* ...les coordonnées relatives */
    MoveTo(165,10); DrawCString(msg);

    sprintf(msg, "%4d ", getbits(tache.where,31,16));
    MoveTo(10,10); DrawCString(msg);
    sprintf(msg, "%4d ", getbits(tache.where,15,16));   /* ...les coordonnées globales */
    MoveTo(50,10); DrawCString(msg);
}

/***** FONCTION AJUSTECURS: ajuste le dessin du curseur
                                        en fonction de sa position à l'écran *****/

    AjusteCurs( )
{
static  modif;
int     ind;

    ind = PtInRgn(&tache.where, cont); /* est-il au dessus du contenu de la fenêtre active? */
    if (ind == modif) return;
    if (ind) SetCursor(croix);
    else SetCursor(arrow);
    modif = ind;
}

        /***** FONCTION GETBITS *****/

    getbits(x,p,n)              /*** prend n bits à partir de la position p ***/

unsigned long x;
unsigned int  p,n;

{   return( (x>>(p+1-n)) & ~(~0<<n) );   }

        /***** PROCEDURE PAINT: dessine le contenu des fenêtres *****/

    void Paint( )
{
int     i = 0;
int     j, k;
Rect    r;

    SetRect(&r,0,0,100,100);            /* le rectangle initial */
    for(k=0; k<3; ++k)
        {
        for(j=0; j<5; ++j)
            {
            SetSolidPenPat(++i);        /* une nouvelle couleur */
            PaintRect(&r);              /* on peint le rectangle... */
            OffsetRect(&r,100,0);       /* ...et on le déplace */
            }
        OffsetRect(&r,-500,100);        /* on passe à la rangée suivante */
        }
}

        /***** PROCEDURE INFO: dessine les zones d'information *****/

    pascal void Info(bar,data,wnd)

long    bar, data, wnd;

{
    /* on ne fait rien */
}
```


Manipulations avec TaskMaster

Voici l'exemple le plus achevé de cet ouvrage (même s'il est susceptible de contenir quelques bogues de par la liberté laissée à l'utilisateur), celui qui offre les possibilités les plus intéressantes pour découvrir le fonctionnement de la boîte à outils de l'Apple IIGS. La taille des données manipulées est relativement importante (on gère quatre dialogues et une alerte, trois fenêtres normales, des menus déroulants...), ce qui justifie la séparation du code en deux parties : un fichier « data », qui contient la définition de toutes les données utilisées par l'application, et un fichier « instructions », qui contient toutes les instructions en langage C. Pour définir la frontière entre les deux, mettez-vous à la place d'une personne qui serait chargée de traduire votre programme dans une langue étrangère : il ne faudrait pas qu'elle ait à toucher le moindre caractère dans la partie « instructions ».

Cet exemple ne fonctionne qu'en mode 640, certaines fenêtres de dialogue dépassant la largeur de 320 pixels. Il va permettre de comprendre le fonctionnement de **TaskMaster**, en permettant en direct le paramétrage du champ *TaskMask*. A gauche de l'écran, une fenêtre affiche en permanence les caractéristiques des cinq derniers codes renvoyés par **TaskMaster** : le code lui-même, le type d'événement qui l'a généré et le contenu du champ *TaskData*. Avec le masque maximal, on verra principalement des *MouseUp* et des événements d'activation sur les fenêtres. Cette fenêtre d'informations possède une routine de mise à jour qui ne fait rien : elle ne sera donc pas rafraîchie si une fenêtre vient à la recouvrir. On ne peut pas la fermer, on ne peut pas la déplacer.

Deux autres fenêtres sont présentes. Elles peuvent être ouvertes et fermées à volonté, mais ce qui est remarquable, c'est que leur région contour peut être modifiée à volonté par l'utilisateur : suppression de la zone d'informations ou de la barre de titre, passage en type alerte, etc. On pourra ainsi tester toutes les combinaisons possibles, et constater que la case de contrôle de taille est une entité plutôt capricieuse ! Souvenez-vous, nous avions déjà eu des ennuis dans le chapitre V avec la fenêtre contenant des ovales (elle possédait une case de contrôle de taille et une barre horizontale, mais pas de barre verticale). La région contour n'est pas mémorisée si la fenêtre est fermée. Sa réouverture se fait comme elle a été définie à l'origine.
Note La valeur étant prise avant l'ouverture de la fenêtre de dialogue, le bit *F_HILITED* est toujours positionné. Nous avons interdit la modification des bits *F_HILITED, F_ZOOMED, F_ALLOCATED* et *F_VIS*.

Le contenu de ces fenêtres est une série de traits épais en diagonale, chacun réalisé avec un pattern différent. Ce qui nous donne la possibilité de constater la dépendance de la couleur d'un pixel vis-à-vis de sa position horizontale à l'écran en mode 640. Bien que la procédure de dessin soit la même pour les deux fenêtres, les couleurs

affichées ne sont pas identiques. En effet, l'une a été créée sur un pixel pair, l'autre sur un pixel impair. Et si on fait défiler le contenu, les couleurs changent. C'est pourquoi un dialogue permet de fixer pour chaque fenêtre la valeur du masque de la coordonnée horizontale de l'origine. Du fait que nous utilisons la palette standard des couleurs en mode 640, un masque forçant les multiples de 2 suffit à rétablir un défilement correct. Ce masque est mémorisé, même si la fenêtre est fermée puis rouverte.
Remarque L'unité de défilement horizontal a été fixée à 7 pixels pour la fenêtre 1 ; si le masque n'autorise que les multiples de 8, elle ne pourra pas défiler en utilisant la flèche droite !

La zone d'informations contient le seul libellé « zone d'informations », invariable.

Le titre de chaque fenêtre peut être modifié grâce à un troisième dialogue. Jusqu'à 20 caractères sont autorisés, et puisque nous écrivons directement à l'adresse où la fenêtre stocke le titre, celui-ci est mémorisé de manière permanente, même si la fenêtre est fermée puis rouverte.

Une autre fenêtre de dialogue permet de modifier le champ *TaskMask* pour tester le comportement de **TaskMaster** dans différentes situations. Les 13 bits définis sont accessibles, mais on fera attention à certaines combinaisons. Par exemple, l'utilisateur se bloque complètement s'il interdit à la fois la gestion automatique de **MenuSelect** et **MenuKey** : il n'y a plus moyen de choisir un menu déroulant ! Pratiquement tous les articles possèdent un équivalent-clavier. Souvenez-vous qu'ils sont inopérants quand une fenêtre d'accessoire de bureau est située au premier plan, état de fait désolant qui, nous l'espérons, sera bientôt modifié par Apple.

Les accessoires de bureau sont gérés par **TaskMaster**... à condition que le champs Taskmask le permette. L'application ne gère pas le copier-coller, pour ses propres fenêtres. De plus, dans le menu , l'article « A propos de » est géré. Une alerte toute simple pour donner à l'utilisateur toutes les informations intéressantes (ou plublicitaires) concernant l'application. On notera un bogue irritant dans **TaskMaster** : quand cette fonction gère elle-même la réponse à une commande dans un menu déroulant (cas des articles spéciaux), elle oublie d'appeler **HiliteMenu** pour rendre au titre son aspect normal !

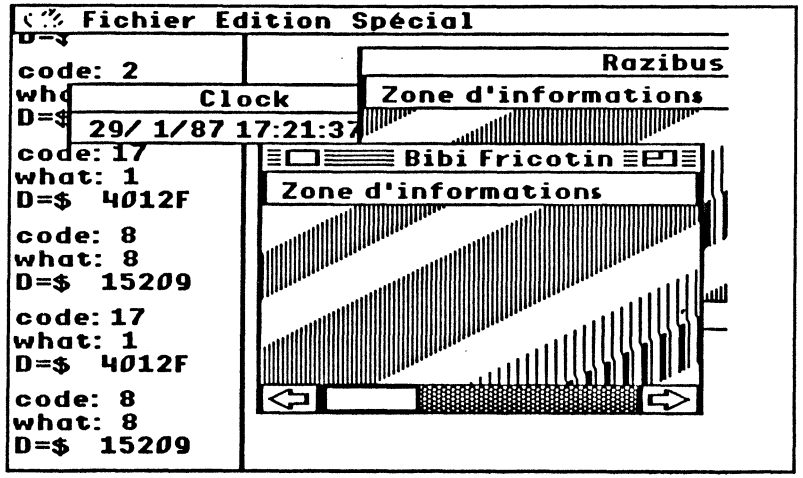
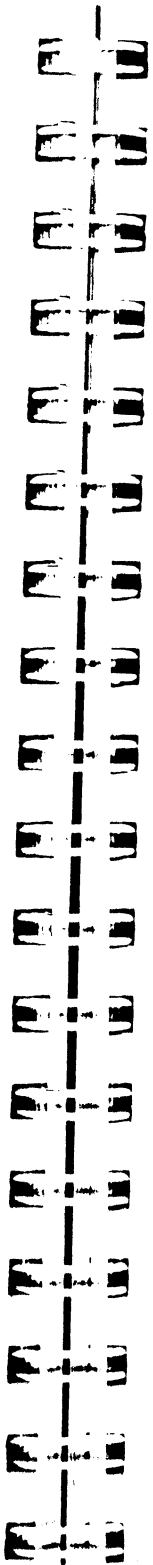


Figure XI.3. Un morceau d'écran tiré de l'exemple.

```
/********** FICHIER EXTM.DATA: les données nécessaires à l'application **********/

                                                    /* définition des menus déroulants */
char Menu1[ ] = "> @\\XN1";
char Menu11[ ] = "- A propos de...\\N256V*Aa";
char Menu19[ ] = ". ";
char Menu2[ ] = "> Fichier \\N2";
char Menu21[ ] = "- Ouvrir fen1\\N257*1 ";
char Menu22[ ] = "- Ouvrir fen2\\N258*2 ";
char Menu23[ ] = "- Fermer\\N255*Ff";
char Menu24[ ] = "- Quitter\\N260*Qq";
char Menu29[ ] = ". ";
char Menu3[ ] = "> Edition \\N3";
char Menu31[ ] = "- Annuler\\N250V*Zz";
char Menu32[ ] = "- Couper\\N251*Xx";
char Menu33[ ] = "- Copier\\N252*Cc";
char Menu34[ ] = "- Coller\\N253*Vv";
char Menu35[ ] = "- Effacer\\N254";
char Menu39[ ] = ". ";
char Menu4[ ] = "> Spécial \\N4";
char Menu41[ ] = "- Fixer TaskMask...\\N301*Mm";
char Menu42[ ] = "- Fixer OrgnMask...\\N302*Oo";
char Menu43[ ] = "- Fixer wFrame...\\N303*Ww";
char Menu44[ ] = "- Changer le titre...\\N304*Tt";
char Menu49[ ] = ". ";
                                                    /* définition de quelques constantes */
#define iAbout      256
#define iFen1       257
#define iFen2       258
#define iFermer     255
#define iQuitter    260
#define iTM         301
#define iSOM        302
#define iFrame      303
#define iTitre      304
                                                    /* déclaration de trois procédures */
void        updFen0( );
void        Paint( );
void        Info( );

int         colfen[ ] = {0,0x0F00,0x020F,0xF0F0,0x00F0};    /* couleurs des fenêtres */
char        infoStr[ ] = "Zone d'information";               /* une chaîne de caractères */

                                                    /* définition de la fenêtre d'information */
ParamList   maFen0 = {
        sizeof(ParamList), 0x0020, "", 0L,
        {0,0,0,0}, colfen, 0, 0,
        0,0,0,0,
        0,0,0,0,
        0L, 0, 0L, 0L, updFen0,
        {12,0,201,100}, -1L, 0L };
                                                    /* définition de la fenêtre 1 */
char titre1[22] = "\13 Fenêtre 1 ";
ParamList   maFen1 = {
        sizeof(ParamList), 0xDDB0, titre1, 1L,
        {45,110,186,610}, colfen, 0, 0,
        300, 600, 141, 496,
        7, 7, 77, 77,
        0L, 15, 0L, Info, Paint,
        {50,120,130,570}, -1L, 0L };
                                                    /* définition de la fenêtre 2 */
char titre2[22] = "\13 Fenêtre 2 ";
ParamList   maFen2 = {
        sizeof(ParamList), 0xDDF0, titre2, 2L,
        {45,110,186,610}, colfen, 0, 0,
        300, 600, 141, 496,
        11, 11, 99, 99,
```

```
    0L, 15, 0L, Info, Paint,
    {75,141,165,570}, -1L, 0L };

char pat[8][16] = {                                  /* définition des huit patterns utilisés */

    {0x77,0x77,0x77,0x77,0x77,0x77,0x77,0x77,0x77,0x77,0x77,0x77,0x77,0x77,0x77,0x77,0x77},
                                                     /* rouge/blanc */

    {0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB},
                                                     /* vert/blanc */

    {0xDD,0xDD,0xDD,0xDD,0xDD,0xDD,0xDD,0xDD,0xDD,0xDD,0xDD,0xDD,0xDD,0xDD,0xDD,0xDD},
                                                     /* blanc/bleu */

    {0xEE,0xEE,0xEE,0xEE,0xEE,0xEE,0xEE,0xEE,0xEE,0xEE,0xEE,0xEE,0xEE,0xEE,0xEE,0xEE},
                                                     /* blanc/jaune */

    {0x66,0x66,0x66,0x66,0x66,0x66,0x66,0x66,0x66,0x66,0x66,0x66,0x66,0x66,0x66,0x66},
                                                     /* rouge/jaune */

    {0x7D,0x7D,0x7D,0x7D,0x7D,0x7D,0x7D,0x7D,0x7D,0x7D,0x7D,0x7D,0x7D,0x7D,0x7D,0x7D},
                                                     /* rouge/blanc/blanc/bleu */

    {0x7B,0x7B,0x7B,0x7B,0x7B,0x7B,0x7B,0x7B,0x7B,0x7B,0x7B,0x7B,0x7B,0x7B,0x7B,0x7B},
                                                     /* rouge/blanc/vert/blanc */

    {0xF6,0x06,0xF6,0x06,0xF6,0x06,0xF6,0x06,0xF6,0x06,0xF6,0x06,0xF6,0x06,0xF6,0x06}
                                                     /* blanc/blanc/rouge/jaune/noir/noir/rouge/jaune */
};

                                                     /* définition de l'alerte "A propos de..." */
ItemTemplate  bOK={1,{100,110,120,190}, ButtonItem, "\2OK", 0, 2, 0L};
ItemTemplate  lig1={2,{10,10,25,290}, StatText + ItemDisable,
                                                     "\44Exemple écrit par Jean-Pierre CURCIO", 0, 0, 0L};
ItemTemplate  lig2={3,{30,60,45,290}, StatText + ItemDisable,
                                                     "\30pour illustrer l'ouvrage", 0, 0, 0L};
ItemTemplate  lig3={4,{50,20,65,290}, StatText + ItemDisable,
                                                     "\43'LA BOITE A OUTILS DE L'APPLE IIGS'", 0, 0, 0L};
ItemTemplate  lig4={5,{70,90,85,290}, StatText + ItemDisable,
                                                     "\21(Editions du PSI)", 0, 0, 0L};

AlertTemplate about= { {40,172,170,468}, 1, 0x80, 0x80, 0x80, 0x80,
                                                     {&bOK, &lig1, &lig2, &lig3, &lig4, 0L} };

                                                     /* définition du dialogue "Fixer TaskMask..." */
ItemTemplate  TMOK={1, {135,10,155,90}, ButtonItem, "\2OK", 0, 2, 0L};
ItemTemplate  TMannul={2, {135,270,155,350}, ButtonItem, "\7Annuler", 0, 2, 0L};
ItemTemplate  TMtxt1={3,{135,120,145,260}, StatText + ItemDisable,
                                                     "\20Nouvelle valeur:", 0, 0, 0L};
ItemTemplate  TMtxt2={4,{145,150,155,250}, StatText + ItemDisable, "", 0, 0, 0L};
ItemTemplate  TMtxt3={5,{1,50,10,350}, StatText + ItemDisable,
                                                     "\36Changer le mot bas de TaskMask", 0, 0, 0L};
ItemTemplate  TM0={10, {10,10,24,180}, CheckItem, "\0120: MenuKey", 0, 0, 0L};
ItemTemplate  TM1={11, {25,10,39,180}, CheckItem, "\0111: Update", 0, 0, 0L};
ItemTemplate  TM2={12, {40,10,54,180}, CheckItem, "\0152: FindWindow", 0, 0, 0L};
ItemTemplate  TM3={13, {55,10,69,180}, CheckItem, "\0153: MenuSelect", 0, 0, 0L};
ItemTemplate  TM4={14, {70,10,84,180}, CheckItem, "\0124: OpenNDA", 0, 0, 0L};
ItemTemplate  TM5={15, {85,10,99,180}, CheckItem, "\0165: SystemClick", 0, 0, 0L};
ItemTemplate  TM6={16, {100,10,114,180}, CheckItem, "\0156: DragWindow", 0, 0, 0L};
ItemTemplate  TM7={17, {115,10,129,180}, CheckItem, "\0177: SelectWindow", 0, 0, 0L};
ItemTemplate  TM8={18, {10,190,24,350}, CheckItem, "\0168: TrackGoAway", 0, 0, 0L};
ItemTemplate  TM9={19, {25,190,39,350}, CheckItem, "\0149: TrackZoom", 0, 0, 0L};
```



```
ItemTemplate  TM10={20, {40,190,54,350}, CheckItem, "\015A: GrowWindow", 0, 0, 0L};
ItemTemplate  TM11={21, {55,190,69,350}, CheckItem, "\015B: défilement", 0, 0, 0L};
ItemTemplate  TM12={22, {70,190,84,350}, CheckItem, "\013C: spéciaux", 0, 0, 0L};
ItemTemplate  TM13={23, {85,190,99,350}, CheckItem, "\014D: inutilisé", 0, 0xFF00, 0L};
ItemTemplate  TM14={24, {100,190,114,350}, CheckItem, "\014E: inutilisé", 0, 0xFF00, 0L};
ItemTemplate  TM15={25, {115,190,129,350}, CheckItem, "\014F: inutilisé", 0, 0xFF00, 0L};

DialogTemplate TMdlg={ {25,140,190,500}, TRUE, 0L,
                    {&TMOK, &TMannul, &TMtxt1, &TMtxt2, &TMtxt3,
                    &TM0, &TM1, &TM2, &TM3, &TM4, &TM5, &TM6, &TM7, &TM8,
                    &TM9, &TM10, &TM11, &TM12, &TM13, &TM14, &TM15, 0L} };

                                                     /* définition du dialogue "Fixer OrgnMask..." */
ItemTemplate  OMOK={1, {130,10,150,90}, ButtonItem, "\2OK", 0, 2, 0L};
ItemTemplate  OMannul={2, {130,130,150,210}, ButtonItem, "\7Annuler", 0, 2, 0L};
ItemTemplate  OMtxt1={3,{110,20,120,150}, StatText + ItemDisable,
                                                     "\20Nouvelle valeur:", 0, 0, 0L};
ItemTemplate  OMtxt2={4,{110,150,120,190}, StatText + ItemDisable, "", 0, 0, 0L};
ItemTemplate  OMtxt3={5,{10,30,20,200}, StatText + ItemDisable,
                                                     "\24Faire un SetOrgnMask", 0, 0, 0L};
ItemTemplate  OM0={10, {35,35,49,200}, RadioItem, "\015Pas de masque", 1, 1, 0L};
ItemTemplate  OM1={11, {50,35,64,200}, RadioItem, "\015Multiple de 2", 0, 1, 0L};
ItemTemplate  OM2={12, {65,35,79,200}, RadioItem, "\015Multiple de 4", 0, 1, 0L};
ItemTemplate  OM3={13, {80,35,94,200}, RadioItem, "\015Multiple de 8", 0, 1, 0L};

DialogTemplate OMdlg={ {30,210,190,430}, TRUE, 0L,
                    {&OMOK, &OMannul, &OMtxt1, &OMtxt2, &OMtxt3,
                    &OM0, &OM1, &OM2, &OM3, 0L} };

                                                     /* définition du dialogue "Changer le titre..." */
ItemTemplate  TITOK={1, {60,10,80,90}, ButtonItem, "\2OK", 0, 2, 0L};
ItemTemplate  TITannul={2, {60,150,80,230}, ButtonItem, "\7Annuler", 0, 2, 0L};
ItemTemplate  TITtxt={3, {10,10,20,230}, StatText + ItemDisable,
                                                     "\35Nouveau titre pour la fenêtre", 0, 0, 0L};
ItemTemplate  TITline={4, {30,10,45,230}, EditLine + ItemDisable, "", 20, 0, 0L};

DialogTemplate TITdlg={ {30,200,120,440}, TRUE, 0L,
                                                     {&TITOK, &TITannul, &TITtxt, &TITline, 0L} };

                                                     /* définition du dialogue "Fixer wFrame..." */
ItemTemplate  wFOK={1, {135,10,155,90}, ButtonItem, "\2OK", 0, 2, 0L};
ItemTemplate  wFannul={2, {135,270,155,350}, ButtonItem, "\7Annuler", 0, 2, 0L};
ItemTemplate  wFtxt1={3,{135,120,145,260}, StatText + ItemDisable,
                                                     "\20Nouvelle valeur:", 0, 0, 0L};
ItemTemplate  wFtxt2={4,{145,150,155,250}, StatText + ItemDisable, "", 0, 0, 0L};
ItemTemplate  wFtxt3={5,{1,50,10,350}, StatText + ItemDisable,
                                                     "\36Changer le cadre de la fenêtre", 0, 0, 0L};
ItemTemplate  wF0={10, {10,10,24,180}, CheckItem, "\0140: F_HILITED", 0, 0xFF00, 0L};
ItemTemplate  wF1={11, {25,10,39,180}, CheckItem, "\0131: F_ZOOMED", 0, 0xFF00, 0L};
ItemTemplate  wF2={12, {40,10,54,180}, CheckItem, "\0162: F_ALLOCATED", 0, 0xFF00, 0L};
ItemTemplate  wF3={13, {55,10,69,180}, CheckItem, "\0153: F_CTRL_TIE", 0, 0, 0L};
ItemTemplate  wF4={14, {70,10,84,180}, CheckItem, "\0114: F_INFO", 0, 0, 0L};
ItemTemplate  wF5={15, {85,10,99,180}, CheckItem, "\0105: F_VIS", 0, 0xFF00, 0L};
ItemTemplate  wF6={16, {100,10,114,180}, CheckItem, "\0156: F_QCONTENT", 0, 0, 0L};
ItemTemplate  wF7={17, {115,10,129,180}, CheckItem, "\0117: F_MOVE", 0, 0, 0L};
ItemTemplate  wF8={18, {10,190,24,350}, CheckItem, "\0118: F_ZOOM", 0, 0, 0L};
ItemTemplate  wF9={19, {25,190,39,350}, CheckItem, "\0119: F_FLEX", 0, 0, 0L};
ItemTemplate  wF10={20, {40,190,54,350}, CheckItem, "\011A: F_GROW", 0, 0, 0L};
ItemTemplate  wF11={21, {55,190,69,350}, CheckItem, "\012B: F_BSCRL", 0, 0, 0L};
ItemTemplate  wF12={22, {70,190,84,350}, CheckItem, "\012C: F_RSCRL", 0, 0, 0L};
ItemTemplate  wF13={23, {85,190,99,350}, CheckItem, "\012D: F_ALERT", 0, 0, 0L};
ItemTemplate  wF14={24, {100,190,114,350}, CheckItem, "\012E: F_CLOSE", 0, 0, 0L};
ItemTemplate  wF15={25, {115,190,129,350}, CheckItem, "\012F: F_TITLE", 0, 0, 0L};
```

```
DialogTemplate wFdlg={ {25,140,190,500}, TRUE, 0L,
                       {&wFOK, &wFannul, &wFtxt1, &wFtxt2, &wFtxt3,
                       &wF0, &wF1, &wF2, &wF3, &wF4, &wF5, &wF6, &wF7, &wF8,
                       &wF9, &wF10, &wF11, &wF12, &wF13, &wF14, &wF15, 0L} };


        /********** FICHIER EXTM.C: les instructions de programme **********/

#include <tools.h>                      /* définition des termes en gras */
#include <entete.h>                     /* définition des termes en italique */
#include <exTM.data>                    /* données modifiables utilisées par l'application */

TaskRec     tache;                      /* ce que manipule TaskMaster */
Pointer     fen0, fen1, fen2;           /* pointeurs sur fenêtre */
Handle      dummyRgn;                   /* handle sur région */
Rect        scroll;                     /* rectangle contenu de la fenêtre 0 */
int         indic = TRUE;               /* indicateur de fin de boucle */
int         myID;                       /* identifiant de l'application */
                                        /* les 4 masques possibles pour SetOrgnMask */
int         masques[4] = {0xFFFF, 0xFFFE, 0xFFFC, 0xFFF8};
int         indM[2] = {0,0};            /* indices masque courant */

                    /***** PROGRAMME PRINCIPAL *****/

    main( )

{
int     code;                           /* code retourné par TaskMaster */
char    msg[20];                        /* réservation de 20 octets */
    myID = debut_appl(1);               /* cette application ne tourne qu'en mode 640 */
    PlaceMenus( );                      /* installe la barre des menus système */
    FlushEvents(EveryEvent, 0);         /* supprime tous les événements en attente */
    dummyRgn = NewRgn( );               /* alloue une nouvelle région */
    fen0 = NewWindow(&maFen0);          /* ouvre la fenêtre d'information... */
    SetPort(fen0);                      /* ...dans laquelle on va écrire en permanence */
    GetPortRect(&scroll);               /* ...et dont on récupère le rectangle contenu */
    tache.TaskMask = 0x00001FFF;        /* initialisation du masque */

    do {                                /* début de la boucle d'événements */
        code = TaskMaster(EveryEvent, &tache); /* on va chercher l'événement suivant */
        if(!code) continue;             /* l'application n'a rien à faire, on boucle! */

        ScrollRect(&scroll, 0, -35, dummyRgn); /* on fait défiler la fenêtre d'information */
        sprintf(msg, "code: %2d", code);
        MoveTo(3,160); DrawCString(msg);    /* on écrit le code retourné */
        sprintf(msg, "what: %2d", tache.what);
        MoveTo(3,170); DrawCString(msg);    /* et le type d'événement qui l'a généré */
        sprintf(msg, "D=$%8lx", tache.TaskData);
        MoveTo(3,180); DrawCString(msg);    /* et les données associées */

        switch(code)                    /* code à traiter par l'application */
            {
            case wInMenuBar :           /* réponse à un article de menu */
                indic = ExecMenu(tache.TaskData);
                break;

            case wInGoAway :            /* une fenêtre à fermer */
                ferme(tache.TaskData);
                break;

            case wInSpecial :           /* réponse à un article spécial */
                indic = ExecMenu(tache.TaskData);
```



```
                break;
            }
        }
    while(indic);                       /* fin de la boucle d'événements */

    DisposeRgn(dummyRgn);               /* on évacue la région allouée */
    quitter(myID);                      /* et on retourne au finder */
}

        /***** FONCTION PLACEMENUS: installe la barre des menus *****/

    PlaceMenus( )

{
    InsertMenu(NewMenu(Menu4), 0);      /* le menu 4 dans la barre */
    InsertMenu(NewMenu(Menu3), 0);      /* le menu 3 à sa gauche */
    InsertMenu(NewMenu(Menu2), 0);      /* le menu 2 à sa gauche */
    InsertMenu(NewMenu(Menu1), 0);      /* le menu  à sa gauche */
    FixAppleMenu(1);                    /* accessoires de bureau dans le menu  */
    FixMenuBar( );                      /* calcule la taille des menus */
    DrawMenuBar( );                     /* dessine la barre des menus */
}

        /***** FONCTION EXECMENU: répond au choix d'un article de menu *****/

    int ExecMenu(art, menu)             /* retourne FALSE si quitter est choisi */

int     art;                            /* article choisi */
int     menu;                           /* dans ce menu */

{
int     val;

    switch (art)                        /* quel article choisi? */
        {
        case iAbout:                    /* "A propos de..." */
            Alert(&about, 0L);          /* on lance l'alerte publicitaire */
            break;

        case iFen1:                     /* "Ouvrir fen1" */
            fen1 = NewWindow(&maFen1);  /* on ouvre la fenêtre 1 */
            val = masques[indM[0]];     /* on cherche la valeur du masque... */
            SetOrgnMask(val,fen1);      /* ...et on le fixe */
            DisableMItem(iFen1);        /* on rend l'article inactif */
            break;

        case iFen2:                     /* "Ouvrir fen2" */
            fen2 = NewWindow(&maFen2);
            val = masques[indM[1]];
            SetOrgnMask(val,fen2);      /* idem */
            DisableMItem(iFen2);
            break;

        case iFermer:                   /* une fenêtre à fermer... */
            ferme(FrontWindow( ));      /* ...l'actuelle fenêtre active */
            break;

        case iQuitter:                  /* l'utilisateur en a assez, on sort */
            return FALSE;
            break;

        case iTM:                       /* "Fixer TaskMask..." */
            gereTM( );                  /* on lance le dialogue */
```

```
            break;

        case iSOM:                          /* "Fixer OrgnMask..." */
            gereSOM();                      /* on lance le dialogue */
            break;

        case iFrame:                        /* "Fixer wFrame..." */
            gereFrame();                    /* on lance le dialogue */
            break;

        case iTitre:                        /* "Changer le titre..." */
            gereTitre();                    /* on lance le dialogue */
            break;
        }

    if (art) HiliteMenu(FALSE, menu);       /* on rend le titre du menu normal */
    return TRUE;                            /* et on continue */
}

        /***** FONCTION FERME: fermeture d'une fenêtre *****/

    ferme(port)

Pointer     port;                           /* pointeur sur la fenêtre à fermer */

{
    if (port == 0L) return;                 /* il n'y a rien à fermer */
    if (port == fen0) return;               /* on ne ferme pas la fenêtre d'information */
    if (GetWKind(port)) return;             /* on ne ferme pas les fenêtres système */
                                            /* c'est enfin une bonne fenêtre, on rétablit...*/
    EnableMItem(iFen1-1 + (int) GetWRefCon(port)); /* ...l'activité de l'article Ouvrir */
    if (port == fen1) fen1 = 0L;            /* on élimine le contenu... */
    else if (port == fen2) fen2 = 0L;       /* ...du bon pointeur... */
    CloseWindow(port);                      /* ...et on ferme la fenêtre */
}

    /***** PROCEDURE UPDFEN0: dessine le contenu de la fenêtre d'information *****/

    void    updFen0( )                      /* cette procédure ne fait rien */
                                            /* mais TaskMaster est contente */
{ }                                 /* et ne nous envoie pas d'événement de mise à jour */

    /***** PROCEDURE PAINT: dessine le contenu des fenêtres 1 et 2 *****/

    void    Paint( )

{
int     i, j;

    SetPenSize(40,20);                      /* un crayon de belle taille */
    for (i=0, j=0; i<15; ++i)
        {
        SetPenPat(pat[j]);                  /* on change le pattern du crayon */
        MoveTo(40*i,0);
        LineTo(0,20*i);                     /* on trace une ligne oblique */
        j = (j + 1) % 8;                    /* le prochain pattern sera différent */
        }
}

        /***** PROCEDURE INFO: dessine les zones d'information *****/

    pascal void Info(bar,data,wnd)
```



```
long    bar, data, wnd;

{                                       /* on se place deux lignes au-dessus du bas de la zone */
    MoveTo(10, ((Rect*) bar)->bottom - 2);
    DrawCString(infoStr);               /* et on écrit un message */
}
    /***** FONCTION GERETM: gestion du dialogue permettant de fixer TaskMask *****/

    gereTM( )
{
Pointer     dlg;                        /* pointeur sur dialogue */
int         it;                         /* l'item courant */
int         i;
long        val;
char        msg[10];

    val = tache.TaskMask;               /* on récupère la valeur actuelle du masque */
    dlg = GetNewModalDialog(&TMdlg); /* on ouvre la fenêtre de dialogue */
    sprintf(msg, "$%8lx", val);         /* la valeur est traduite en chaîne type C... */
    ctopstr(msg);                       /* ...convertie en chaîne type Pascal... */
    SetIText(dlg, 4, msg);              /* ...et affichée dans un texte statique */
                                        /* chaque bit est traduit au niveau... */
    for(i=0; i<16; ++i)
        SetItemValue(getbits(val,i,1), dlg, 10+i);      /* ...d'une case à cocher */

    do {
        it = ModalDialog(0L);           /* l'utilisateur a choisi un item */
        if (GetItemType(dlg, it) == CheckItem)  /* est-ce une case à cocher? */
            {
            SetItemValue(1-GetItemValue(dlg, it), dlg, it);     /* la case est changée... */
            val ^= (1<<(it-10));        /* ...et la valeur est changée en conséquence... */
            sprintf(msg, "$%8lx", val); ctopstr(msg);
            SetIText(dlg, 4, msg);      /* ...et affichée comme précédemment */
            }
        }
    while (it>2);       /* on boucle tant que l'utilisateur n'a pas cliqué dans un bouton */

    if (it == 1) tache.TaskMask = val;  /* il a cliqué dans OK, on change le masque */

    CloseDialog(dlg);               /* on peut fermer la fenêtre de dialogue */
}

    /***** FONCTION GERESOM: gestion du dialogue permettant d'appeler SetOrgnMask *****/

    gereSOM( )
{
int         indprov;                    /* indice provisoire */
Pointer     dlg;                        /* pointeur sur dialogue */
Pointer     port;                       /* pointeur sur fenêtre */
int         it;                         /* l'item courant */
int         i, k;
long        val;
char        msg[10];

    port = FrontWindow( );              /* quelle est la fenêtre active */
    if (port == 0L) return;             /* pour la forme: on est sûr qu'il y en a au moins une */
    if (GetWKind(port)) return;         /* on ne veut pas d'une fenêtre système */
    if (port == fen1) k=0;              /* fenêtre 1... */
    else if (port == fen2) k=1;         /* ...ou fenêtre 2 */
    else return;                        /* on ne veut pas non plus de la fenêtre d'info */
    val = masques[indM[k]];             /* ancienne valeur du masque */
    dlg = GetNewModalDialog(&OMdlg);/* on ouvre le dialogue */
    sprintf(msg, "$%x", val); ctopstr(msg);
    SetIText(dlg,4,msg);                /* on affiche la valeur comme précédemment ...*/
```

```
    SetItemValue(1, dlg, 10+indM[k]);  /* ...et on marque le bon bouton radio */

    do {
        it = ModalDialog(0L);                   /* l'utilisateur a choisi un item */
        if (GetItemType(dlg,it) == RadioItem)   /* est-ce un bouton radio? */
            {
            SetItemValue(1, dlg, it);           /* on force sa valeur à 1... */
            val = masques[indprov=it-10];       /* ...on change la valeur du masque... */
            sprintf(msg, "$%x", val); ctopstr(msg);
            SetIText(dlg, 4, msg);              /* ...et on l'affiche */
            }
        }
    while (it>2);     /* on boucle tant que l'utilisateur n'a pas cliqué dans un bouton */

    if (it == 1)                                /* si c'est le bouton OK... */
        {
        SetOrgnMask(val, port);                 /* ...on fixe un nouveau masque... */
        indM[k] = indprov;                      /* ...et on le mémorise sous forme d'indice */
        }

    CloseDialog(dlg);                           /* on ferme le dialogue */
}

/***** FONCTION GEREFRAME: gestion du dialogue
                                    permettant de changer le contour des fenêtres *****/
    gereFrame( )
{
Pointer     dlg;                                /* pointeur sur dialogue */
Pointer     port, oldport;                      /* une fenêtre et un grafport */
int         it;                                 /* l'item courant */
int         i;
int         val;
char        msg[10];
Rect        r;                                  /* rectangle contenu de la fenêtre */

    port = FrontWindow( );                      /* quelle est la fenêtre active */
    if (port == 0L) return;                     /* pour la forme: on est sûr qu'il y en a au moins une */
    val = GetWFrame(port);                      /* les caractéristiques actuelles */
    dlg = GetNewModalDialog(&wFdlg);            /* on ouvre le dialogue */
    if (port != fen1 && port != fen2)           /* si la fenêtre n'est ni la fenêtre 1 ni la fenêtre 2... */
        {                                       /* ...on désactive le bouton OK... */
        HiliteControl(255, GetControlItem(dlg, 1));
        SetDefButton(2, dlg);                   /* ...et on fait d'Annuler le bouton par défaut */
        }
    sprintf(msg, "$%8x", val); ctopstr(msg);
    SetIText(dlg, 4, msg);                      /* on affiche la valeur actuelle de wFrame */
    for(i=0; i<16; ++i)                         /* chaque bit est traduit au niveau... */
        SetItemValue(getbits((long)val,i,1),dlg,10+i);   /* ...d'une case à cocher */

    do {
        it = ModalDialog(0L);                   /* un item est choisi */
        if (GetItemType(dlg, it) == CheckItem)  /* est-ce une case à cocher? */
            {
            SetItemValue(1-GetItemValue(dlg, it), dlg, it);      /* on la change... */
            val ^= (1<<(it-10));  /* ...et on change la valeur provisoire en conséquence */
            sprintf(msg, "$%8x", val); ctopstr(msg);
            SetIText(dlg, 4, msg);              /* on affiche cette valeur */
            }
        }
    while (it>2);     /* on boucle tant que l'utilisateur n'a pas cliqué dans un bouton */

    if (it == 1)                                /* si c'est le bouton OK... */
        {
```



```
        SetWFrame(val, port);           /* ...on change les caractéristiques */
        oldport = GetPort( );           /* on mémorise le grafport actif */
        SetPort(port);                  /* on fixe le grafport de la fenêtre */
        GetPortRect(&r);                /* on calcule le rectangle contenu */
        /* on force le Window Manager à redessiner le contour en agrandissant le contenu... */
        SizeWindow(r.right - r.left + 1, r.bottom - r.top + 1, port); /* ...d'un pixel
                                                                    dans les 2 directions */
        SetPort(oldport);               /* on rétablit l'ancien grafport */
        }

    CloseDialog(dlg);                   /* on ferme le dialogue */
}

/***** FONCTION GERETITRE: gestion du dialogue permettant
                                    de changer le titre des fenêtres *****/

    gereTitre( )
{
Pointer     dlg;                        /* pointeur sur dialogue */
Pointer     port;                       /* pointeur sur fenêtre */
int         it;                         /* l'item courant */
int         k;
Pointer     titre;                      /* pointeur sur titre */

    port = FrontWindow( );              /* quelle est la fenêtre active */
    if (port == 0L) return;             /* pour la forme: on est sûr qu'il y en a au moins une */
    if (GetWKind(port)) return;         /* on ne veut pas des fenêtres système... */
    if (port == fen1) k=0;
    else if (port == fen2) k=1;
    else return;                        /* ...ni de la fenêtre d'information */
    titre = GetWTitle(port);            /* on récupère un pointeur sur le titre actuel */
    dlg = GetNewModalDialog(&TITdlg); /* on ouvre le dialogue */
    SetIText(dlg, 4, titre);            /* on affiche le titre actuel... */
    SelIText(dlg, 4, 0, 50);            /* ...et on le sélectionne entièrement */

    it = ModalDialog(0L); /* pas besoin de boucle ici: seuls les boutons ne sont pas muets */

    if (it == 1)                        /* si le bouton OK a été choisi... */
        GetIText(dlg, 4, k ? titre2 : titre1); /* ...on va remplacer le titre de la fenêtre */
                /* même pas besoin de SetWTitle, puisqu'on écrit à l'adresse du précédent */
                /* dès que le dialogue sera fermé, la fenêtre sera réactivée, */
                /* et son contour redessiné... avec le nouveau titre! */
    CloseDialog(dlg);                   /* on ferme le dialogue */
}

        /***** FONCTION GETBITS: une vieille connaissance *****/

    getbits(x, p, n)            /*** prend n bits à partir de la position p ***/

unsigned long   x;
unsigned int    p, n;

{   return( (x>>(p+1-n)) & ~(~0<<n) );   }
```

CHAPITRE XII

DÉBUT ET FIN D'UNE APPLICATION

GÉNÉRALITÉS

• Nous l'avons dit et répété, il faut initialiser un outil avant d'utiliser la moindre des routines qui le composent. Les routines ne vérifiant pas si l'outil est actif au moment où elles sont appelées, elles conduiront soit à des résultats aberrants, soit plus vraisemblablement au plantage du système, en cas de non-initialisation.

La première chose que doit faire une application, c'est initialiser le Tool Locator. A partir de là, deux possibilités : soit elle impose une version aux outils résidant en mémoire morte, auquel cas elle doit les charger explicitement par l'une des procédures **LoadTools** et **LoadOneTool**, soit elle accepte le chargement par défaut, tel qu'il résulte de l'état des fichiers système qui contiennent les patches des outils en ROM. Nous avons choisi cette deuxième solution.

L'application initialise ensuite le Memory Manager, grâce auquel l'application va recevoir un identifiant. Puis le Miscellaneous Tools, qui sera utilisé par l'Event Manager. Elle va également se réserver un bloc de mémoire vive entièrement situé dans la banque 0, qui va constituer tout ou partie des pages zéro dont les outils auront besoin pour fonctionner. Pour notre chargement standard, nous avons besoin de huit pages zéro, nous réserverons donc un bloc de 8 × 256 octets (soit $800).

Ensuite nous initialisons QuickDraw et l'Event Manager. QuickDraw réclame trois pages zéro, ce qui explique que l'adresse de page zéro donnée à l'Event Manager soit z + 0×300. On remarquera également une façon simple d'écrire ces initialisations pour qu'elles soient valables aussi bien en mode 320 qu'en mode 640, en utilisant les décalages de bits. Toute application réclamera au minimum ces initialisations.

La procédure **InitCursor** est ensuite appelée, pour faire apparaître le curseur système. Pourquoi ici ? Parce que juste après, nous allons charger les outils résidant sur disque, et que s'il y a une erreur au chargement, une pseudo-alerte sera affichée, qui réclamera l'utilisation du curseur pour un clic souris dans un bouton. Puisque nous n'imposons pas de numéro de version à QuickDraw et à l'Event Manager, qui sont des outils résidant en mémoire morte, nous sommes à peu près sûr qu'ils seront initialisés (il faudrait un manque de mémoire complet pour que ce ne soit pas le cas, et **NewHandle** aurait retourné zéro-long : rien n'empêche de tester ce résultat !), et nous

pouvons donc utiliser la fonction **TLMountVolume**. Nous ne nous fatiguerons pas trop : s'il y a une erreur de chargement, nous affichons l'alerte, et quel que soit le bouton choisi, nous annulons tout ce que nous avons déjà fait et nous retournons vers le programme qui a lancé notre application.

Nous avons pris l'option de charger les sept outils principaux, laissant de côté le Font Manager. Cela n'est évidemment pas une règle immuable, et cette règle doit être adaptée en fonction de l'utilisation qui en est faite au sein de l'application. Certaines règles sont à suivre absolument : le Window Manager risque d'avoir besoin du Control Manager, si les fenêtres possèdent des contrôles dans leur région contour, le Control Manager a forcément besoin du Window Manager, Line Edit a besoin du Window Manager et peut avoir besoin du Scrap Manager pour gérer son presse-papiers privé, le Dialog Manager a forcément besoin du Control Manager et de Line Edit, le Standard File Operations a besoin du Dialog Manager, le Desk Manager a besoin du Window Manager, de Line Edit, du Scrap Manager, du Menu Manager, etc. On constate très vite qu'une application a besoin des sept outils principaux pour pouvoir fonctionner. (Le Font Manager est un peu à part : il ne réclame que QuickDraw, et n'est réclamé par personne, tant que les autres outils se contentent du jeu de caractères système).

Maintenant que nos outils RAM sont chargés, nous allons pouvoir les initialiser. L'ordre des initialisations n'est pas primordial, sauf quand deux outils se partagent la même page zéro. Ainsi, le Window Manager doit obligatoirement être initialisé après l'Event Manager, et le Dialog Manager après le Control Manager. De même, le Menu Manager devra être initialisé après le Window Manager, de telle sorte que le haut de l'écran (là où viendra s'installer la barre système) puisse être déclaré zone interdite par la fonction **Desktop**.

Chaque fois qu'un outil est initialisé, nous testons le code erreur, et nous affichons un message donnant la valeur de ce code s'il n'est pas nul. Nous attendons un clic souris pour poursuivre, mais il y a des chances pour que l'application se poursuive mal ! On pourrait à ce moment-là proposer ou imposer à l'utilisateur un retour au programme appelant... nous ne l'avons pas fait.

Quand tous ces outils sont initialisés, on peut considérer que la phase « début d'application » est terminée. Ceux qui le veulent pourraient y inclure l'appel à la fonction **FlushEvent**, plutôt que de l'appeler à partir d'un autre fichier programme de l'application... ou de ne pas l'appeler du tout.

• Au moment de quitter l'application, il faudra faire un peu de ménage et laisser place nette à la suivante (généralement le programme superviseur, à moins que l'application ne permette le transfert direct à une autre application... mais cela, il faut le gérer !). Nous allons procéder en sens inverse de nos initialisations, en fermant les uns après les autres les outils que nous avons initialisés.

L'appel à **MMShutDown** est primordial : c'est lui qui va rendre purgeables tous les blocs de mémoire réservés explicitement ou implicitement par l'application, puisqu'ils sont marqués de l'identifiant de l'application. C'est grâce à cette procédure qu'on peut se permettre par exemple, et nous ne nous en sommes pas privés, de ne pas fermer les fenêtres gérées par l'application : le Memory Manager s'en chargera tout seul ! (C'est le Desk Manager qui se charge de dire au Memory Manager quelles sont les fenêtres système qui pourraient ne pas avoir été fermées.)

Dernière étape, on appelle la commande QUIT de ProDOS. En attendant des *glue routines* C pour appeler directement ProDOS, ou mieux, un File Manager du type de celui du Macintosh, c'est par un appel direct à quelques instructions assembleur que nous conclurons. Dans les environnements Megamax, on les remplacera par la simple instruction exit(0) (exit est une fonction de la bibliothèque standard).

FICHIER PROGRAMME

Il s'appelle DebFin.c et est valable quel que soit le mode super hi-res choisi. Pour être utilisé, il suffit de le compiler tout seul une fois pour toutes, ce qui crée un fichier objet DebFin.o. Chaque fois qu'une application sera créée :

– elle débutera par l'appel à la fonction debut_appl, en passant le mode de résolution en argument (suivant le code : 0 pour le mode 320 × 200, 1 pour le mode 640 × 200). Cette fonction retourne l'identifiant de l'application ;

– elle finira par l'appel à la fonction quitter, en passant l'identifiant de l'application en argument (qu'elle aura donc conservé quelque part en mémoire, même si elle n'en a pas besoin) ;

– au moment de l'édition des liens *(link)*, elle inclura le fichier DebFin.o à la liste des fichiers à relier, de telle sorte que les fonctions préalablement citées soient incluses dans le code exécutable généré.

```
/********** Fichier DEBFIN.C **********/


#include    <tools.h>                   /* définition des termes en gras */

int     _errno;                 /* variable qui reçoit les codes d'erreur */
char    **pgzero;               /* handle sur pages zéro système */
long    quitparms[ ] = {0, 0};  /* paramètres pour la commande ProDOS QUIT */

int  tools[ ] = {7,                     /* sept outils chargés en permanence */
            14, 0x103,                  /* Window Manager version 1.03 ou postérieure */
            15, 0x103,                  /* Menu Manager version 1.03 ou postérieure */
            16, 0x103,                  /* Control Manager version 1.03 ou postérieure */
            20, 0x100,                  /* Line Edit version 1.00 ou postérieure */
            21, 0x101,                  /* Dialog Manager version 1.01 ou postérieure */
            22, 0x101,                  /* Scrap Manager version 1.01 ou postérieure */
            23, 0x101 };                /* Standard File version 1.01 ou postérieure */

char    ligne1[ ] = "\42Certains outils ne sont pas à jour";
char    ligne2[ ] = "\41Démarrer avec un autre système...";
char    btn1[ ] = "\2OK";
char    btn2[ ] = "\7Annuler";


                /***** fonction debut_appl *****/

/*  cette fonction assure le chargement des principaux outils
    en mémoire vive ainsi que leur initialisation  */

    int debut_appl(mode)                /* retourne l'identifiant de l'application */

int     mode;                           /* reçoit 0 si le mode 320 est utilisé
                                                1 si le mode 640 est utilisé */

{
int     myID;                           /* identifiant de l'application */
char    *z;                             /* pointeur sur les pages zéro */

    TLStartUp( );                       /* initialisation du Tool Locator */
    myID = MMStartUp( );                /* initialisation du Memory Manager */
    MTStartUp( );                       /* initialisation de Miscellaneous Tools */
    pgzero = NewHandle(0x800L, myID, 0xC001, 0L);   /* réservation d'un bloc
                                                       de 8 pages zéro */
    z = *pgzero;                        /* on déréférence le handle pour obtenir un pointeur */
    QDStartUp((int) z, mode<<7, 160, myID);   /* initialisation de QuickDraw */
                                        /* initialisation du Memory Manager */
    EMStartUp((int) (z + 0x300), 20, 0, 10*(32<<mode), 0, 200, myID);
    InitCursor( );                      /* le curseur est rendu visible en forme de flèche */

    LoadTools(tools);                   /* chargement des outils à partir du disque système */
    if (_errno)                         /* si autre chose que zéro, donc si erreur... */
        {                               /* ...on affiche une pseudo-alerte... */
```

```
            TLMountVolume(20+160*mode, 60, ligne1, ligne2, btn1, btn2);
            retourfinder(myID);          /* ...et on retourne au finder */
            }

    WindStartUp(myID);                   /* initialisation du Window Manager */
    if(_errno) erreur(10,45, "WindStartUp");    /* si erreur, on l'affiche à l'écran */
    Refresh(0L);                         /* on se place en environnement desktop */
    CtlStartUp(myID, (int) (z + 0x400)); /* initialisation du Control Manager */
    if(_errno) erreur(10,60, "CtrlStartUp");
    LEStartUp(myID, (int) (z + 0x500)); /* initialisation de Line Edit */
    if(_errno) erreur(10,75, "LEStartUp");
    DialogStartUp(myID);                 /* initialisation du Dialog Manager */
    if(_errno) erreur(10,90, "DialogStartUp");
    MenuStartUp(myID, (int) (z + 0x600));    /* initialisation du Menu Manager */
    if(_errno) erreur(10,105, "MenuStartUp");
    ScrapStartUp( );                     /* initialisation du Scrap Manager */
    if(_errno) erreur(10,120, "ScrapStartUp");
    SFStartUp(myID,(int) (z + 0x700));   /* initialisation du Standard File Operations */
    if(_errno) erreur(10,135, "SFStartUp");
    DeskStartUp( );                      /* initialisation du Desk Manager */
    if(_errno) erreur(10,150, "DeskStartUp");

    return myID;                         /* la fonction retourne l'identifiant de l'application */
}


                /***** fonction quitter *****/

/*  cette fonction assure la désallocation des principaux outils
    en mémoire vive et le retour au finder      */
    quitter(myID)

int     myID;                   /* identifiant de l'application */


{
    GrafOff( );                          /* on quitte le mode super haute résolution */
    DeskShutDown( );                     /* on ferme le Desk Manager */
    SFShutDown( );                       /* on ferme le Standard File Operations */
    ScrapShutDown( );                    /* on ferme le Scrap Manager */
    MenuShutDown( );                     /* on ferme le Menu Manager */
    DialogShutDown( );                   /* on ferme le Dialog Manager */
    LEShutDown( );                       /* on ferme Line Edit */
    CtlShutDown( );                      /* on ferme le Control Manager */
    WindShutDown( );                     /* on ferme le Window Manager */
    retourfinder(myID);                  /* retour au finder */
}

            /***** FONCTION RETOURFINDER *****/

    /* cette fonction assure le retour au programme général ayant permis
    le lancement de l'application, que ce soit MouseDesk,
    le finder d'Apple, le program launcher ou toute autre chose... */

    retourfinder(myID)

{
    EMShutDown( );                       /* on ferme l'Event Manager */
    QDShutDown( );                       /* on ferme QuickDraw */
    DisposeHandle(pgzero);               /* on n'a plus besoin de nos pages zéro */
```



```
    MTShutDown( );                       /* on ferme le Miscellaneous Tools */
    MMShutDown(myID);                    /* on ferme le Memory Manager */
    TLShutDown( );                       /* on ferme le Tool Locator */

            /* pour quitter proprement, on appelle en assembleur la commande ProDOS QUIT */
    asm
        {
        jsl     0xE100A8                /* ProDOS dispatcher    */
        dcw     0x29                    /* commande QUIT        */
        dcl     quitparms               /* parameter list       */
        dcw     0xF000                  /* BRK                  */
        }
}


                /***** Fonction erreur *****/

/*  assure l'affichage d'un message d'erreur quand un outil ne peut être initialisé  */

    erreur(x,y,msg)

int     x, y;                   /* localisation du message */
char    msg[ ];                 /* morceau de message */

{
char    mess[50];               /* message complet */

    sprintf(mess, "Erreur %x dans %s", _errno, msg);   /* préparation du message */
    MoveTo(x,y); DrawCString(mess);                    /* écriture sur le desktop */
    while (!Button(0));                                /* attente d'un clic souris */
}
```

EXEMPLE COMPLET : LA VERSION DES OUTILS

Nous avons parlé plusieurs fois de la version des outils utilisés. Dans cet exemple, nous allons enfin les afficher ! Rien de bien compliqué là-dedans, l'application se passe de commentaire. On remarquera que le Font Manager, qui n'est pas dans la liste des outils manipulés par debut_appl, est chargé et initialisé tout seul comme un grand. Ce qui nous permet en plus de l'utiliser, et de faire notre affichage dans la police Venice 14. On remarquera également comment **TaskMaster** est utilisée pour gérer l'option Quitter du menu déroulant, étant donné que c'est le seul article actif.

Version des outils

System Loader vers. 101	Window Manager vers. 103
Tool Locator vers. 102	Menu Manager vers. 103
Memory Manager vers. 102	Control Manager vers. 103
Miscellaneous vers. 102	Line Edit vers. 100
Desk Manager vers. 102	Dialog Manager vers. 101
QuickDraw vers. 102	Standard File vers. 101
Event Manager vers. 100	Scrap Manager vers. 101
	Font Manager vers. 100

Figure XII.1. Les outils que nous avons utilisés tout au long de cet ouvrage.

Les numéros de version affichés dans l'illustration XII.1 sont ceux des outils qui étaient à notre disposition quand nous avons terminé la rédaction de cet ouvrage (c'est par erreur que le Font Manager affiche une version 1.00, erreur qui lui est propre !). Si vous utilisez des outils plus récents, il est possible que certaines modifications aient été apportées, non dans la syntaxe des appels, mais plutôt dans le fonctionnement : améliorations, corrections de bogues, voire nouvelles routines. En aucun cas vous ne devriez utiliser des versions d'outils antérieures à celles listées dans cet écran, sinon vous ne pourriez pas charger les outils à partir du disque, étant donné les numéros de version imposés par notre fonction debut_appl.

```
#include <tools.h>                  /* définition des caractères en gras */
#include <entete.h>                 /* définition des caractères en italique */

char Menu2[ ] = "> Version des outils\\N2";
char Menu21[ ] = "- par Jean-Pierre CURCIO\\N271D";
char Menu22[ ] = "- Quitter\\N272*Qq";
char Menu29[ ] = ". ";

extern _errno;                      /* la variable qui reçoit les erreurs */

main( )
{
int         myID;                   /* identifiant de l'application */
TaskRec     tache;                  /* ce que manipule TaskMaster */
char        msg[50];

    tache.TaskMask = 0x00001FFF;
    myID = debut_appl(1);           /* mode 640 */

    LoadOneTool(27, 0x100);         /* chargement et initialisation du Font Manager */
    if(_errno) erreur(10,30, "Load27");
    FMStartUp(0L, myID, (int) *NewHandle(0x100L, myID, 0xC001, 0L));
    if(_errno) erreur(10,165, "FMStartUp");

    Desktop(5,0x400000FF);          /* le desktop sera tout blanc */
    SetTitleStart(120);             /* 120 pixels laissés à gauche dans la barre de menus */
    InsertMenu(NewMenu(Menu2), 0);  /* l'unique menu de la barre */
    FixMenuBar( );                  /* dimension de la barre */
    DrawMenuBar( );                 /* dessin de la barre */

    InstallFont(14*256+0, 5, 1);    /* on va écrire en Venice 14 */
    sprintf(msg, "System Loader vers. %x", LoaderVersion( ));
    MoveTo(10,40); DrawCString(msg);
    sprintf(msg, "Tool Locator vers. %x", TLVersion( ));
    MoveTo(10,60); DrawCString(msg);
    sprintf(msg, "Memory Manager vers. %x", MMVersion( ));
    MoveTo(10,80); DrawCString(msg);
    sprintf(msg, "Miscellaneous vers. %x", MTVersion( ));
    MoveTo(10,100); DrawCString(msg);
    sprintf(msg, "Desk Manager vers. %x", DeskVersion( ));
    MoveTo(10,120); DrawCString(msg);
    sprintf(msg, "QuickDraw vers. %x", QDVersion( ));
    MoveTo(10,140); DrawCString(msg);
    sprintf(msg, "Event Manager vers. %x", EMVersion( ));
    MoveTo(10,160); DrawCString(msg);
    sprintf(msg, "Window Manager vers. %x", WindVersion( ));
    MoveTo(250,40); DrawCString(msg);
    sprintf(msg, "Menu Manager vers. %x", MenuVersion( ));
    MoveTo(250,60); DrawCString(msg);
    sprintf(msg, "Control Manager vers. %x", CtrlVersion( ));
```



```
    MoveTo(250,80); DrawCString(msg);
    sprintf(msg, "Line Edit vers. %x", LEVersion( ));
    MoveTo(250,100); DrawCString(msg);
    sprintf(msg, "Dialog Manager vers. %x", DialogVersion( ));
    MoveTo(250,120); DrawCString(msg);
    sprintf(msg, "Standard File vers. %x", SFVersion( ));
    MoveTo(250,140); DrawCString(msg);
    sprintf(msg, "Scrap Manager vers. %x", ScrapVersion( ));
    MoveTo(250,160); DrawCString(msg);
    sprintf(msg, "Font Manager vers. %x", FMVersion( ));
    MoveTo(250,180); DrawCString(msg);

    FlushEvents(EveryEvent, 0);     /* on supprime les événements en attente */
    while (TaskMaster(EveryEvent, &tache) != wInMenuBar); /* quelle belle boucle! */

    FMShutDown( );                  /* on ferme le Font Manager */
    UnloadOneTool(27);              /* on évacue le Font Manager de la mémoire */
    quitter(myID);                  /* on quitte l'application */
}
```

CHAPITRE XIII

LISTE DES ROUTINES

RAPPEL DES OUTILS DISPONIBLES

Voici la liste des outils disponibles ou annoncés à l'heure où nous écrivons ces lignes. Chaque outil porte un numéro (décimal) et une abréviation que nous utiliserons dans la liste des routines (si une routine au moins appartient à l'outil). Pour connaître le numéro exact d'identification d'une routine utilisée par le dispatcher, on rapprochera ce numéro et le code de la routine, et on utilisera la fonction spéciale *inline* pour appeler la routine.

Outils en ROM		**Outils en RAM**	
1. Tool Locator	TL	14. Window Manager	Wind
2. Memory Manager	Mem	15. Menu Manager	Menu
3. Miscellaneous Tools	Misc	16. Control Manager	Ctl
4. QuickDraw II	QD	17. System Loader	
5. Desk Manager	Desk	18. QuickDraw auxilliary	
6. Event Manager	Evnt	19. Printer Driver	
7. Scheduler		20. Line Edit	LE
8. Sound Tools		21. Dialog Manager	Dlg
9. Apple Desktop Bus		22. Scrap Manager	Scrp
10. SANE		23. Standard File	SF
11. Integer Math		24. Disk Utilities	
12. Text Tools		25. Note Synthesizer	
13. *Utilisation interne*		26. Note Sequencer	
		27. Font Manager	Font

LISTE DES ROUTINES

Voici la liste des 479 routines dont nous avons parlé dans cet ouvrage. Elle est loin d'être exhaustive, mais nombreuses sont les applications qui n'auront pas besoin d'autre chose. Pour chaque routine, on donnera : le type de données qu'elle retourne (type *void* si c'est une procédure), son nom, le type de chacun de ses arguments, l'abréviation de l'outil à laquelle elle appartient, le numéro d'identification à l'intérieur de l'outil et la page où elle est expliquée ou évoquée. Les routines sont classées par ordre alphabétique.

Les conventions suivantes sont employées :

– *int* désigne un entier sur 16 bits (c'est-à-dire un mot machine, qui peut également désigner une valeur booléenne) ;

– *long* désigne un entier sur 32 bits (un entier long, soit deux mots machine) ;

– *Ptr* désigne un pointeur ou une adresse (codé sur 32 bits, l'octet le plus significatif étant à zéro) ;

– *Hdl* désigne un handle (pointeur sur pointeur, donc mêmes spécifications) ;

– *void* désigne une routine qui ne retourne aucun résultat (ce type est peu utilisé en C, où toute fonction est censée retourner un résultat... mais rappelons que les routines *Toolbox* sont de type Pascal).

void	**AddPt(Ptr,Ptr)**	QD	128	74
int	**Alert(Ptr,Ptr)**	Dlg	23	258
void	**BeginUpdate(Ptr)**	Wind	30	147
void	**BlockMove(Ptr,Ptr,long)**	Mem	43	41
int	**Button(int)**	Evnt	13	109
void	**CalcMenuSize(int,int,int)**	Menu	28	179
int	**CautionAlert(Ptr,Ptr)**	Dlg	26	259
void	**CharBounds(int,Ptr)**	QD	172	85
int	**CharWidth(int)**	QD	168	85
void	**CheckMItem(int,int)**	Menu	50	176
void	**ClipRect(Ptr)**	QD	38	65
void	**CloseAllNDAs()**	Desk	29	279
void	**CloseDialog(Ptr)**	Dlg	12	256
void	**CloseNDA(int)**	Desk	22	279
void	**CloseNDAbyWinPtr(Ptr)**	Desk	28	279
void	**ClosePicture()**	QD	185	94
void	**ClosePoly()**	QD	194	77
void	**ClosePort(Ptr)**	QD	26	63
void	**CloseRgn(Hdl)**	QD	110	79
void	**CloseWindow(Ptr)**	Wind	11	134
void	**CompactMem()**	Mem	31	39
void	**CopyRgn(Hdl,Hdl)**	QD	105	80
int	**CountMItems(int)**	Menu	20	181
void	**CStringBounds(Ptr,Ptr)**	QD	174	85
int	**CStringWidth(Ptr)**	QD	170	85
void	**CtlShutDown()**	Ctl	3	320
void	**CtlStartUp(int,int)**	Ctl	2	203
int	**CtlVersion()**	Ctl	4	322
void	**DeleteItem(int)**	Menu	16	179
void	**DeleteMenu(int)**	Menu	14	179
void	**DeskShutDown()**	Desk	3	320
void	**DeskStartUp()**	Desk	2	277
long	**Desktop(int,long)**	Wind	12	150
int	**DeskVersion()**	Desk	4	322
int	**DialogSelect(Ptr,Ptr,Ptr)**	Dlg	17	260
void	**DialogShutDown()**	Dlg	3	320
void	**DialogStartUp(int)**	Dlg	2	250
int	**DialogVersion()**	Dlg	4	322
void	**DiffRgn(Hdl,Hdl,Hdl)**	QD	115	82
void	**DisableDItem(Ptr,int)**	Dlg	57	261
void	**DisableMItem(int)**	Menu	49	172
void	**DisposeAll(int)**	Mem	17	38
void	**DisposeControl(Hdl)**	Ctl	10	205
void	**DisposeHandle(Hdl)**	Mem	16	38
void	**DisposeMenu(Hdl)**	Menu	46	179
void	**DisposeRgn(Hdl)**	QD	104	79
void	**DlgCopy(Ptr)**	Dlg	19	260
void	**DlgCut(Ptr)**	Dlg	18	260
void	**DlgDelete(Ptr)**	Dlg	21	260
void	**DlgPaste(Ptr)**	Dlg	20	260
void	**DragControl(int,int,Ptr,Ptr,int,Hdl)**	Ctl	23	210
long	**DragRect(long,Ptr,int,int,Ptr,Ptr,Ptr,int)**	Ctl	29	211
void	**DragWindow(int,int,int,int,Ptr,Ptr)**	Wind	26	142
void	**DrawChar(int)**	QD	164	93
void	**DrawControls(Ptr)**	Ctl	16	209
void	**DrawCString(Ptr)**	QD	166	93
void	**DrawMenuBar()**	Menu	42	173
void	**DrawPicture(Hdl,Ptr)**	QD	186	97
void	**DrawString(Ptr)**	QD	165	93
void	**DrawText(Ptr,int)**	QD	167	93
int	**EmptyRect(Ptr)**	QD	82	76
int	**EmptyRgn(Hdl)**	QD	120	82
void	**EMShutDown()**	Evnt	3	321
void	**EMStartUp(int,int,int,int,int,int,int)**	Evnt	2	108
int	**EMVersion()**	Evnt	4	322
void	**EnableDItem(Ptr,int)**	Dlg	58	261
void	**EnableMItem(int)**	Menu	48	176
void	**EndInfoDrawing()**	Wind	81	149
void	**EndUpdate(Ptr)**	Wind	31	147
int	**EqualPt(Ptr,Ptr)**	QD	131	74
int	**EqualRect(Ptr,Ptr)**	QD	81	77
int	**EqualRgn(Hdl,Hdl)**	QD	119	83
void	**EraseArc(Ptr,int,int)**	QD	100	92
void	**EraseOval(Ptr)**	QD	90	92
void	**ErasePoly(Hdl)**	QD	190	93
void	**EraseRect(Ptr)**	QD	85	88
void	**EraseRgn(Hdl)**	QD	123	92
void	**EraseRRect(Ptr,int,int)**	QD	95	92
int	**EventAvail(int,Ptr)**	Evnt	11	108
void	**FillArc(Ptr,int,int,Ptr)**	QD	102	92
void	**FillOval(Ptr,Ptr)**	QD	92	92
void	**FillPoly(Hdl,Ptr)**	QD	192	93
void	**FillRect(Ptr,Ptr)**	QD	87	89
void	**FillRgn(Hdl,Ptr)**	QD	125	92
void	**FillRRect(Ptr,int,int,Ptr)**	QD	97	90
int	**FindControl(Ptr,int,int,Ptr)**	Ctl	19	206
int	**FindDItem(Ptr,long)**	Dlg	36	265
Hdl	**FindHandle(Ptr)**	Mem	26	36
int	**FindWindow(Ptr,int,int)**	Wind	23	141
void	**FixAppleMenu(int)**	Desk	30	277
int	**FixMenuBar()**	Menu	19	173
void	**FlashMenuBar()**	Menu	12	181
int	**FlushEvents(int,int)**	Evnt	21	108
void	**FMShutDown()**	Font	3	322
void	**FMStartUp(Ptr,int,int)**	Font	2	280
int	**FMVersion()**	Font	4	322
void	**FrameArc(Ptr,int,int)**	QD	98	92
void	**FrameOval(Ptr)**	QD	88	92
void	**FramePoly(Hdl)**	QD	188	93
void	**FrameRect(Ptr)**	QD	83	88
void	**FrameRgn(Hdl)**	QD	121	92
void	**FrameRRect(Ptr,int,int)**	QD	93	90
long	**FreeMem()**	Mem	27	39
Ptr	**FrontWindow()**	Wind	21	141
int	**GetAlertStage()**	Dlg	52	259
int	**GetBackColor()**	QD	163	72
void	**GetBackPat(Ptr)**	QD	53	67
long	**GetBarColors()**	Menu	24	181
long	**GetCaretTime()**	Evnt	18	112
Ptr	**GetCDraw(Ptr)**	Wind	72	139
long	**GetCharExtra()**	QD	213	87

void	**GetClip**(Hdl)	QD	37	65
Hdl	**GetClipHandle**()	QD	199	65
int	**GetColorEntry**(int,int)	QD	17	40
void	**GetColorTable**(int,Ptr)	QD	15	40
Hdl	**GetContRgn**(Ptr)	Wind	47	140
Hdl	**GetControlItem**(Ptr,int)	Dlg	30	261
long	**GetCOrigin**(Ptr)	Wind	62	136
Ptr	**GetCtlAction**(Hdl)	Clt	33	210
long	**GetCtlParams**(Hdl)	Ctl	28	213
long	**GetCtlRefCon**(Hdl)	Ctl	35	210
Ptr	**GetCtlTitle**(Hdl)	Ctl	13	209
int	**GetCtlValue**(Hdl)	Ctl	26	211
Ptr	**GetCursorAdr**()	Qd	143	99
long	**GetDataSize**(Ptr)	Wind	64	136
long	**GetDblTime**()	Evnt	17	112
int	**GetDefButton**(Ptr)	Dlg	55	264
Ptr	**GetDefProc**(Ptr)	Wind	49	139
int	**GetFirstItem**(Ptr)	Dlg	42	265
Hdl	**GetFont**()	QD	149	71
int	**GetFontFlags**()	QD	153	71
long	**GetFontID**()	QD	209	72
int	**GetForeColor**()	QD	161	72
void	**GetFrameColor**(Ptr,Ptr)	Wind	16	136
Ptr	**GetFullRect**(Ptr)	Wind	55	136
long	**GetHandleSize**(Hdl)	Mem	24	39
Ptr	**GetInfoDraw**(Ptr)	Wind	74	139
long	**GetInfoRefCon**(Ptr)	Wind	53	139
Ptr	**GetItem**(int)	Menu	37	178
void	**GetItemBox**(Ptr,int,Ptr)	Dlg	40	260
int	**GetItemMark**(int)	Menu	52	176
int	**GetItemStyle**(int)	Menu	54	177
int	**GetItemType**(Ptr,int)	Dlg	38	260
int	**GetItemValue**(Ptr,int)	Dlg	46	261
void	**GetIText**(Ptr,int,Ptr)	Dlg	31	263
int	**GetMasterSCB**()	QD	23	48
long	**GetMaxGrow**(Ptr)	Wind	66	137
Hdl	**GetMenuBar**()	Menu	10	183
Ptr	**GetMenuMgrPort**()	Menu	27	181
Ptr	**GetMenuTitle**(int)	Menu	34	176
Hdl	**GetMHandle**(int)	Menu	22	179
void	**GetMouse**(Ptr)	Evnt	12	109
void	**GetNewDItem**(Ptr,Ptr)	Dlg	51	253
Ptr	**GetNewModalDialog**(Ptr)	Dlg	50	254
int	**GetNextEvent**(int,Ptr)	Evnt	10	108
int	**GetNextItem**(Ptr,int)	Dlg	43	265
Ptr	**GetNextWindow**(Ptr)	Wind	42	151
int	**GetNumNDAs**()	Desk	27	278
long	**GetPage**(Ptr)	Wind	70	139
void	**GetPen**(Ptr)	QD	41	68
void	**GetPenMask**(Ptr)	QD	51	70
int	**GetPenMode**()	QD	47	68
void	**GetPenPat**(Ptr)	QD	49	69
void	**GetPenSize**(Ptr)	QD	45	68
void	**GetPenState**(Ptr)	QD	43	70
int	**GetPixel**(int,int)	QD	136	98
Ptr	**GetPort**()	QD	28	62
void	**GetPortLoc**(Ptr)	QD	30	64
void	**GetPortRect**(Ptr)	QD	32	65
void	**GetRectInfo**(Ptr,Ptr)	Wind	79	150
int	**GetSCB**(int)	QD	19	48
void	**GetScrap**(Hdl,int)	Scrp	13	281

int	**GetScrapCount**()	Scrp	18	283
Hdl	**GetScrapHandle**(int)	Scrp	14	283
Ptr	**GetScrapPath**()	Scrp	16	280
long	**GetScrapSize**(int)	Scrp	15	281
int	**GetScrapState**()	Scrp	19	280
long	**GetScroll**(Ptr)	Wind	68	87
long	**GetSpaceExtra**()	QD	159	48
int	**GetStandardSCB**()	QD	12	140
Hdl	**GetStructRgn**(Ptr)	Wind	46	183
Hdl	**GetSysBar**()	Menu	17	
Hdl	**GetSysFont**()	QD	179	186
int	**GetSysWFlag**(Ptr)	Wind	76	72
int	**GetTextFace**()	QD	155	72
int	**GetTextMode**()	QD	157	72
int	**GetTextSize**()	QD	211	72
int	**GetTitleStart**()	Menu	26	181
int	**GetTitleWidth**(int)	Menu	30	181
Hdl	**GetUpdateRgn**(Ptr)	Wind	48	140
long	**GetUserField**()	QD	71	72
Hdl	**GetVisHandle**()	QD	201	67
void	**GetVisRgn**(Hdl)	QD	181	67
int	**GetWFrame**(Ptr)	Wind	44	135
int	**GetWKind**(Ptr)	Wind	43	140
Ptr	**GetWMgrPort**()	Wind	32	150
long	**GetWRefCon**(Ptr)	Wind	41	136
Ptr	**GetWTitle**(Ptr)	Wind	14	135
void	**GlobalToLocal**(Ptr)	QD	133	74
long	**GrowWindow**(int,int,int,int,Ptr)	Wind	27	144
void	**HandToHand**(Hdl,Hdl,long)	Mem	42	41
void	**HandToPtr**(Hdl,Ptr,long)	Mem	41	40
void	**HideControl**(Hdl)	Ctl	14	209
void	**HideCursor**()	QD	144	99
void	**HideDItem**(Ptr,int)	Dlg	34	261
void	**HidePen**()	QD	39	71
void	**HideWindow**(Ptr)	Wind	18	135
void	**HiliteControl**(int,Hdl)	Ctl	17	209
void	**HiliteMenu**(int,int)	Menu	44	174
void	**HLock**(Hdl)	Mem	32	38
void	**HLockAll**(int)	Mem	33	38
void	**HUnlock**(Hdl)	Mem	34	38
void	**HUnlockAll**(int)	Mem	35	38
void	**InitColorTable**(Ptr)	QD	13	48
void	**InitCursor**()	QD	202	99
void	**InitPort**(Ptr)	QD	25	
void	**InsertItem**(Ptr,int,int)	Menu	15	179
void	**InsertMenu**(Hdl,int)	Menu	13	173
void	**InsertRect**(Ptr,int,int)	QD	76	76
void	**InsertRgn**(Hdl,int,int)	QD	112	82
void	**InstallFont**(long,int)	Font	14	289
void	**InvalRect**(Ptr)	Wind	58	147
void	**InvalRgn**(Hdl)	Wind	59	147
void	**InvertArc**(Ptr,int,int)	QD	101	92
void	**InvertOval**(Ptr)	QD	91	92
void	**InvertPoly**(Hdl)	QD	191	93
void	**InvertRect**(Ptr)	QD	86	88
void	**InvertRgn**(Hdl)	QD	124	92
void	**InvertRRect**(Ptr,int,int)	QD	96	90
int	**IsDialogEvent**(Ptr)	Dlg	16	260
void	**KillControls**(Ptr)	Ctl	11	205
void	**KillPicture**(Hdl)	QD	187	94
void	**KillPoly**(Hdl)	QD	195	77

void	**LEActivate(Hdl)**	LE	15	229
void	**LEClick(Ptr,Hdl)**	LE	13	229
void	**LECopy(Hdl)**	LE	19	231
void	**LECut(Hdl)**	LE	18	231
void	**LEDeactivate(Hdl)**	LE	16	230
void	**LEDelete(Hdl)**	LE	21	231
void	**LEDispose(Hdl)**	LE	10	229
void	**LEFromScrap()**	LE	25	234
void	**LEGetScrapLen()**	LE	28	234
int	**LEIdle(Hdl)**	LE	12	230
void	**LEInsert(Ptr,int,Hdl)**	LE	22	232
void	**LEKey(int,int,Hdl)**	LE	17	230
void	**LENew(Ptr,Ptr,int)**	LE	9	229
Hdl	**LEPaste(Hdl)**	LE	20	231
void	**LEScrapHandle()**	LE	27	234
Hdl	**LESetScrapLen(int)**	LE	29	234
void	**LESetSelect(int,int,Hdl)**	LE	14	230
void	**LESetText(Ptr,int,Hdl)**	LE	11	229
void	**LEShutDown()**	LE	3	320
void	**LEStartUp(int,int)**	LE	2	229
void	**LETextBox(Ptr,int,Ptr,int)**	LE	24	233
void	**LEToScrap()**	LE	26	234
void	**LEUpdate(Hdl)**	LE	23	232
void	**LEVersion()**	LE	4	322
int	**Line(int,int)**	QD	61	87
void	**LineTo(int,int)**	QD	60	87
void	**LoadOneTool(int,int)**	TL	15	28
void	**LoadScrap()**	Scrp	10	280
void	**LoadTools(Ptr)**	TL	14	29
void	**LocalToGlobal(Ptr)**	QD	132	74
void	**MapPoly(Hdl,Ptr,Ptr)**	QD	197	84
void	**MapPt(Ptr,Ptr,Ptr)**	QD	138	84
void	**MapRect(Ptr,Ptr,Ptr)**	QD	139	84
void	**MapRgn(Hdl,Ptr,Ptr)**	QD	140	84
void	**MaxBlock()**	Mem	28	39
long	**MenuKey(Ptr,Hdl)**	Menu	9	175
void	**MenuSelect(Ptr,Hdl)**	Menu	43	174
void	**MenuShutDown()**	Menu	3	320
void	**MenuStartUp(int,int)**	Menu	2	173
void	**MenuVersion()**	Menu	4	322
int	**MMShutDown(int)**	Mem	3	321
void	**MMStartUp()**	Mem	2	36
int	**MMVersion()**	Mem	4	322
int	**ModalDialog(Ptr)**	Dlg	15	254
int	**Move(int,int)**	QD	59	68
void	**MoveControl(int,int,Hdl)**	Ctl	22	210
void	**MovePortTo(int,int)**	QD	34	65
void	**MoveTo(int,int)**	QD	58	68
void	**MoveWindow(int,int,Ptr)**	Wind	25	144
void	**MTVersion()**	Misc	4	322
int	**NewControl(Ptr,Ptr,Ptr,int,int,int,int,Ptr,long,Ptr)**	Ctl	9	204
Hdl	**NewDItem(Ptr,int,Ptr,int,Ptr,int,int,Ptr)**	Dlg	13	251
void	**NewHandle(long,int,int,Ptr)**	Mem	9	36
Hdl	**NewMenu(Ptr)**	Menu	45	173
Hdl	**NewMenuBar(Ptr)**	Menu	21	183
Hdl	**NewModalDialog(Ptr,int,long)**	Dlg	10	250
Ptr	**NewModelessDialog(Ptr,Ptr,Ptr,int,long,Ptr)**	Dlg	11	259
Hdl	**NewRgn()**	QD	103	79
Ptr	**NewWindow(Ptr)**	Wind	9	134
int	**NoteAlert(Ptr,Ptr)**	Dlg	25	259
void	**ObscureCursor()**	QD	146	99

void	**OffsetPoly(Hdl,int,int)**	QD	196	78
void	**OffsetRect(Ptr,int,int)**	QD	75	75
void	**OffsetRgn(Hdl,int,int)**	QD	111	80
int	**OpenNDA(int)**	Desk	21	278
Hdl	**OpenPicture(Ptr)**	QD	183	94
Hdl	**OpenPoly()**	QD	193	77
void	**OpenPort(Ptr)**	QD	24	63
void	**OpenRgn()**	QD	109	79
void	**PaintArc(Ptr,int,int)**	QD	99	92
void	**PaintOval(Ptr)**	QD	89	92
void	**PaintPixels(Ptr)**	QD	127	97
void	**PaintPoly(Hdl)**	QD	189	93
void	**PaintRect(Ptr)**	QD	84	88
void	**PaintRgn(Hdl)**	QD	122	92
void	**PaintRRect(Ptr,int,int)**	QD	94	90
void	**ParamText(Ptr,Ptr,Ptr,Ptr)**	Dlg	27	261
void	**PenNormal()**	QD	54	67
long	**PenRect(int,int,Ptr)**	Wind	33	
int	**PostEvent(int,long)**	Evnt	20	
void	**PPToPort(Ptr,Ptr,int,int,int)**	QD	214	98
void	**Pt2Rect(Ptr,Ptr)**	QD	80	75
int	**PtInRect(Ptr,Ptr)**	QD	79	83
int	**PtInRgn(Ptr,Hdl)**	QD	117	83
void	**PtrToHand(Ptr,Hdl,long)**	Mem	40	39
void	**PurgeAll(int)**	Mem	19	37
void	**PurgeHandle(Hdl)**	Mem	18	37
void	**PutScrap(long,int,Ptr)**	Scrp	12	280
void	**QDShutDown()**	QD	3	321
void	**QDStartUp(int,int,int,int)**	QD	2	47
int	**QDVersion()**	QD	4	322
void	**ReadAsciiTime(Ptr)**	Misc	15	275
void	**ReallocHandle(long,int,int,Ptr,Hdl)**	Mem	10	37
int	**RectInRgn(Ptr,Hdl)**	QD	118	83
void	**RectRgn(Hdl,Ptr)**	QD	108	84
void	**Refresh(Ptr)**	Wind	57	133
void	**RemoveItem(Ptr,int)**	Dlg	14	256
void	**ResetAlertStage()**	Dlg	53	259
void	**RestoreHandle(Hdl)**	Mem	11	37
void	**ScalePt(Ptr,Ptr,Ptr)**	QD	137	84
void	**ScrapShutDown()**	Scrp	3	320
void	**ScrapStartUp()**	Scrp	2	280
int	**ScrapVersion()**	Scrp	4	322
void	**ScrollRect(Ptr,int,int,Hdl)**	QD	126	97
int	**SectRect(Ptr,Ptr,Ptr)**	QD	77	77
void	**SectRgn(Hdl,Hdl,Hdl)**	QD	113	82
void	**SelectWindow(Ptr)**	Wind	17	135
void	**SelIText(Ptr,int,int,int)**	Dlg	33	264
void	**SendBehind(Ptr,Ptr)**	Wind	20	151
void	**SetAllSCBs(int)**	QD	20	48
void	**SetBackColor(int)**	QD	162	72
void	**SetBackPat(Ptr)**	QD	52	67
void	**SetBarColors(int,int,int)**	Menu	23	181
void	**SetCDraw(Ptr,Ptr)**	Wind	73	139
void	**SetCharExtra(long)**	QD	212	87
void	**SetClip(Hdl)**	QD	36	65
void	**SetClipHandle(Hdl)**	QD	198	65
void	**SetColorEntry(int,int,int)**	QD	16	50
void	**SetColorTable(int,Ptr)**	QD	14	50
void	**SetCOrigin(int,int,Ptr)**	Wind	63	136
void	**SetCtlAction(Ptr,Hdl)**	Ctl	32	210
void	**SetCtlParams(int,int,Hdl)**	Ctl	27	213

void	**SetCtlRefCon**(long,Hdl)	Ctl	34	210
void	**SetCtlTitle**(Ptr,Hdl)	Ctl	12	209
void	**SetCtlValue**(int,Hdl)	Ctl	25	211
void	**SetCursor**(Ptr)	QD	142	99
void	**SetDAFont**(Hdl)	Dlg	8	250
void	**SetDataSize**(int,int,Ptr)	Wind	65	136
void	**SetDefButton**(int,Ptr)	Dlg	56	264
void	**SetDefProc**(Ptr,Ptr)	Wind	50	139
void	**SetEmptyRgn**(Hdl)	QD	106	80
void	**SetFont**(Hdl)	QD	148	71
void	**SetFontFlags**(int)	QD	152	71
void	**SetFontID**(long)	QD	208	72
void	**SetForeColor**(int)	QD	160	72
void	**SetFrameColor**(Ptr,Ptr)	Wind	15	136
void	**SetFullRect**(Ptr,Ptr)	Wind	56	136
void	**SetHandleSize**(long,Hdl)	Mem	25	39
void	**SetInfoDraw**(Ptr,Ptr)	Wind	22	139
void	**SetInfoRefCon**(long,Ptr)	Wind	54	139
void	**SetItem**(Ptr,int)	Menu	36	177
void	**SetItemBox**(Ptr,int,Ptr)	Dlg	41	260
void	**SetItemFlag**(int,int)	Menu	38	178
void	**SetItemID**(int,int)	Menu	56	178
void	**SetItemMark**(int,int)	Menu	51	176
void	**SetItemStyle**(int,int)	Menu	53	177
void	**SetItemType**(int,Ptr,int)	Dlg	39	260
void	**SetItemValue**(int,Ptr,int)	Dlg	47	261
void	**SetIText**(Ptr,int,Ptr)	Dlg	32	264
void	**SetMasterSCB**(int)	QD	22	48
void	**SetMaxGrow**(int,int,Ptr)	Wind	67	137
void	**SetMenuBar**(Hdl)	Menu	57	183
void	**SetMenuFlag**(int,int)	Menu	31	176
void	**SetMenuID**(int,int)	Menu	55	176
void	**SetMenuTitle**(Ptr,int)	Menu	33	176
void	**SetOrgnMask**(int,Ptr)	Wind	52	299
void	**SetOrigin**(int,int)	QD	35	297
void	**SetPage**(int,int,Ptr)	Wind	71	139
void	**SetPenMask**(Ptr)	QD	50	70
void	**SetPenMode**(int)	QD	46	68
void	**SetPenPat**(Ptr)	QD	48	69
void	**SetPenSize**(int,int)	QD	44	68
void	**SetPenState**(Ptr)	QD	42	70
void	**SetPort**(Ptr)	QD	27	63
void	**SetPortLoc**(Ptr)	QD	29	64
void	**SetPortRect**(Ptr)	QD	31	65
void	**SetPortSize**(int,int)	QD	33	65
void	**SetPt**(Ptr,int,int)	QD	130	73
void	**SetPurge**(int,Hdl)	Mem	36	38
void	**SetPurgeAll**(int,int)	Mem	37	38
void	**SetRect**(Ptr,int,int,int,int)	QD	74	75
void	**SetRectRgn**(Hdl,int,int,int,int)	QD	107	80
void	**SetSCB**(int,int)	QD	18	48
void	**SetScrapPath**(Ptr)	Scrp	17	280
void	**SetScroll**(int,int,Ptr)	Wind	69	137
void	**SetSolidBackPat**(int)	QD	56	67
void	**SetSolidPenPat**(int)	QD	55	69
void	**SetSpaceExtra**(long)	QD	158	87
void	**SetSysBar**(Hdl)	Menu	18	183
void	**SetSysFont**(Hdl)	QD	178	
void	**SetSysWindow**(Ptr)	Wind	15	140
void	**SetTextFace**(int)	QD	154	72
void	**SetTextMode**(int)	QD	156	72
void	**SetTextSize**(int)	QD	210	72

void	**SetTitleStart**(int)	Menu	25	181
void	**SetTitleWidth**(int,int)	Menu	29	181
void	**SetUserField**(long)	QD	70	72
void	**SetVisHandle**(Hdl)	QD	200	67
void	**SetVisRgn**(Hdl)	QD	180	67
void	**SetWFrame**(int,Ptr)	Wind	45	135
void	**SetWRefCon**(long,Ptr)	Wind	40	136
void	**SetWTitle**(Ptr,Ptr)	Wind	13	135
void	**SFAllCaps**(int)	SF	13	278
void	**SFGetFile**(int,int,Ptr,Ptr,Ptr,Ptr)	SF	9	283
void	**SFPGetFile**(int,int,Ptr,Ptr,Ptr,Ptr,Ptr,Ptr)	SF	11	285
void	**SFPPutFile**(int,int,Ptr,Ptr,int,Ptr,Ptr,Ptr)	SF	12	284
void	**SFPutFile**(int,int,Ptr,Ptr,int,Ptr)	SF	10	284
void	**SFShutDown**()	SF	3	320
void	**SFStartUp**(int,int)	SF	2	283
int	**SFVersion**()	SF	4	322
void	**ShowControl**(Hdl)	Ctl	15	209
void	**ShowCursor**()	QD	145	99
void	**ShowDItem**(Ptr,int)	Dlg	35	261
void	**ShowHide**(int,Ptr)	Wind	35	135
void	**ShowPen**()	QD	40	71
void	**ShowWindow**(Ptr)	Wind	19	135
void	**SizeWindow**(int,int,Ptr)	Wind	28	144
void	**SolidPattern**(Ptr,int)	QD	57	70
void	**StartDrawing**(Ptr)	Wind	77	298
void	**StartInfoDrawing**(Ptr,Ptr)	Wind	80	149
int	**StillDown**(int)	Evnt	14	110
int	**StopAlert**(Ptr,Ptr)	Dlg	24	259
void	**StringBounds**(Ptr,Ptr)	QD	173	85
int	**StringWidth**(Ptr)	QD	169	85
void	**SubPt**(Ptr,Ptr)	D	129	74
void	**SystemClick**(Ptr,Ptr,int)	Desk	23	278
int	**SystemEdit**(int)	Desk	24	278
int	**SystemEvent**(int,long,long,long,int)	Desk	26	278
void	**SystemTask**()	Desk	25	278
int	**TaskMaster**(int,Ptr)	Wind	29	291
int	**TestControl**(int,int,Hdl)	Ctl	20	209
void	**TextBounds**(Ptr,int,Ptr)	QD	175	85
int	**TextWidth**(Ptr,int)	QD	171	85
long	**TickCount**()	Evnt	16	111
int	**TLMountVolume**(int,int,Ptr,Ptr,Ptr,Ptr)	TL	17	28
void	**TLShutDown**()	TL	3	321
void	**TLStartUp**()	TL	2	28
int	**TLTextMountVolume**(Ptr,Ptr,Ptr,Ptr)	TL	18	28
int	**TLVersion**()	TL	4	322
long	**TotalMem**()	Mem	29	39
int	**TrackControl**(int,int,Ptr,Hdl)	Ctl	21	207
int	**TrackGoAway**(int,int,Ptr)	Wind	24	145
int	**TrackZoom**(int,int,Ptr)	Wind	38	145
void	**UnionRect**(Ptr,Ptr,Ptr)	QD	78	77
void	**UnionRgn**(Hdl,Hdl,Hdl)	QD	114	82
void	**UnloadOneTool**(int)	TL	16	29
void	**UnloadScrap**()	Scrp	9	280
void	**ValidRect**(Ptr)	Wind	60	147
void	**ValidRgn**(Ptr)	Wind	61	147
int	**WaitMouseUp**(int)	Evnt	15	110
void	**WindShutDown**()	Wind	3	320
void	**WindStartUp**(int)	ind	2	133
int	**WindVersion**()	Wind	4	322
void	**XorRgn**(Hdl,Hdl,Hdl)	QD	116	82
void	**ZeroScrap**()	Scrp	11	280
void	**ZoomWindow**(Ptr)	Wind	39	145

INDEX