P.S.I.

MATERIEL

BOITE A OUTILS DE L'APPLE IIGS

JEAN-PIERRE CURCIO

PROGRAMMATION

- Toolbox
- Quickdraw
- Exemples complets

BOITE A OUTILS DE L'APPLE IIGS

Pour connaître les dernières nouveautés P.S.I.,
ou nous soumettre un problème technique,
nous mettons à votre disposition un service Minitel

Service Minitel

Sur le 3615 tapez OI puis PSI

Pour tout problème rencontré dans les ouvrages P.S.I.
vous pouvez nous contacter au numéro ci-dessous :

Numéro Vert/Appel Gratuit en France

05 21 22 01

(Composer tous les chiffres, même en région parisienne)

CEP Communication

© *Éditions P.S.I., une société du groupe*
5 place du Colonel Fabien 75491 Paris Cédex 10
1987
ISBN: 2-86595-428-5

MATERIEL

BOITE A OUTILS DE L'APPLE IIGS

JEAN-PIERRE CURCIO

ÉDITIONS P.S.I.
1987

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 :

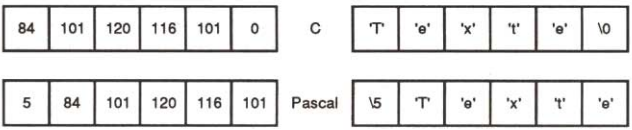
```
struct Rect r;          /* r est déclaré comme un rectangle */

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.

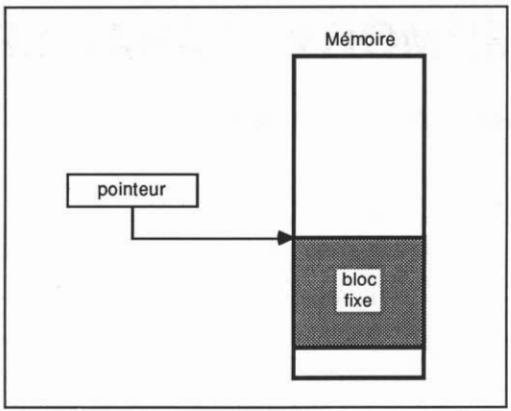


Figure II.1. Pointeur et bloc fixe

• 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.

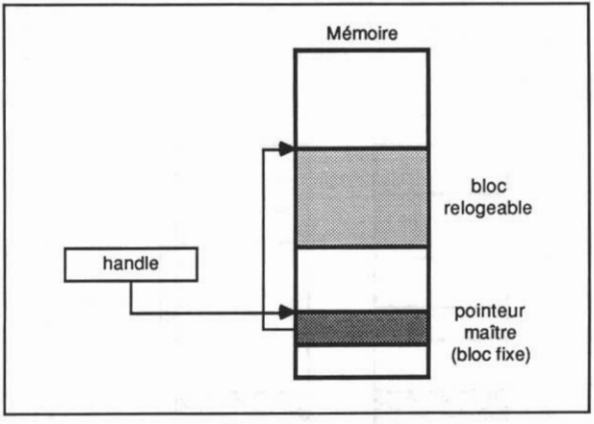


Figure II.2. Handle et bloc relogeable.

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).

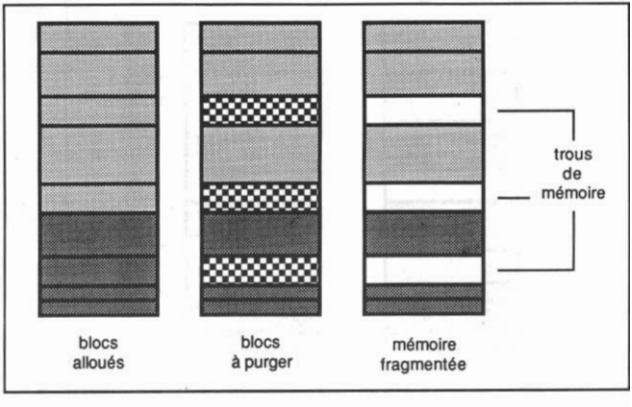


Figure II.3. Fragmentation de la mémoire

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).

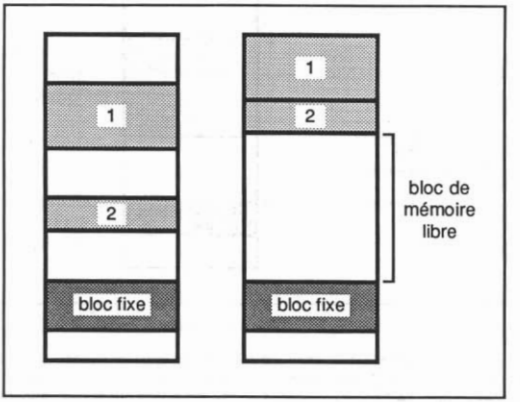
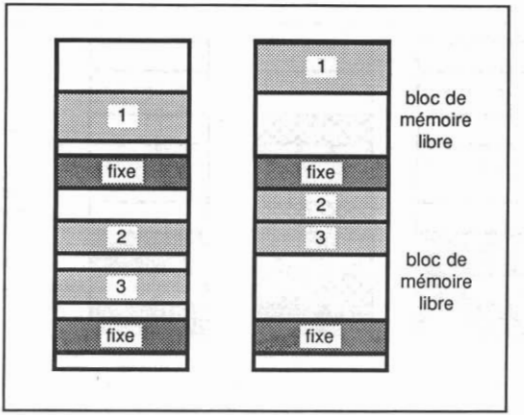


Figure II.4. Compactage de la mémoire.



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.

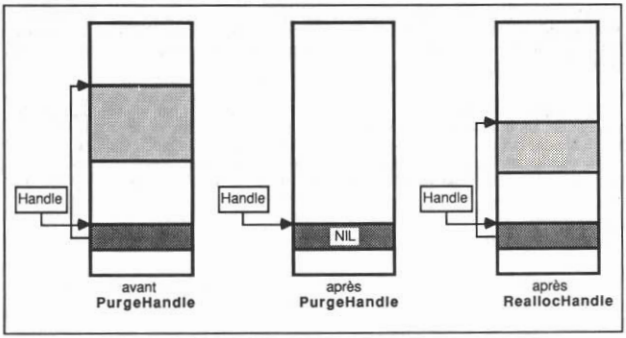


Figure II.6. Fonctionnement de PurgeHandle et ReallocHandle.

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.

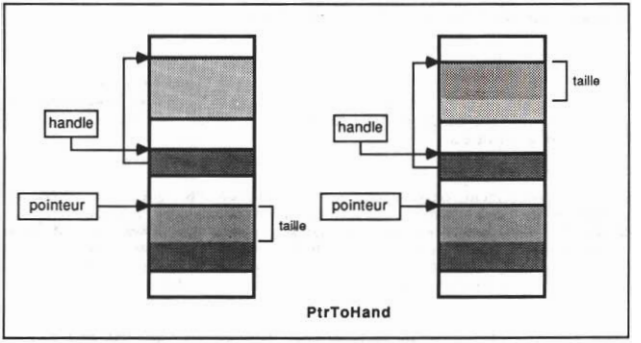


Figure II.7. État mémoire avant et après PtrToHand.

• **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.

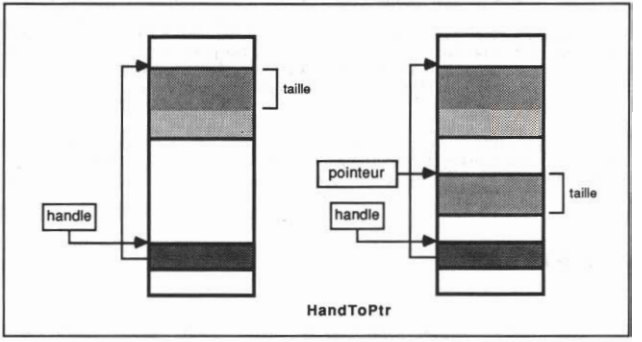


Figure II.8. État mémoire avant et après HandToPtr.

• **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.

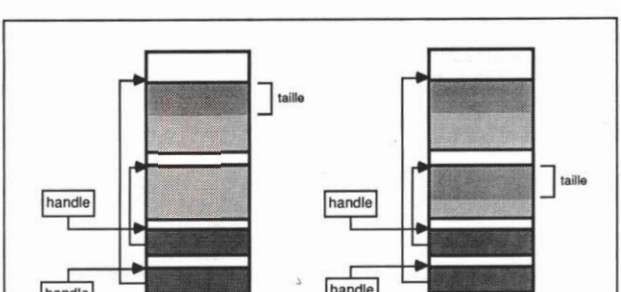


Figure II.9. État mémoire avant et après HandToHand.

• **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.

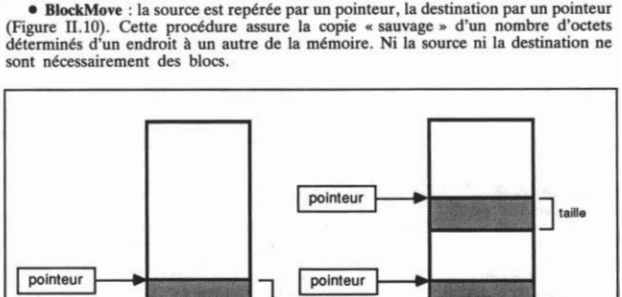


Figure II.10. État mémoire avant et après BlockMove.

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.

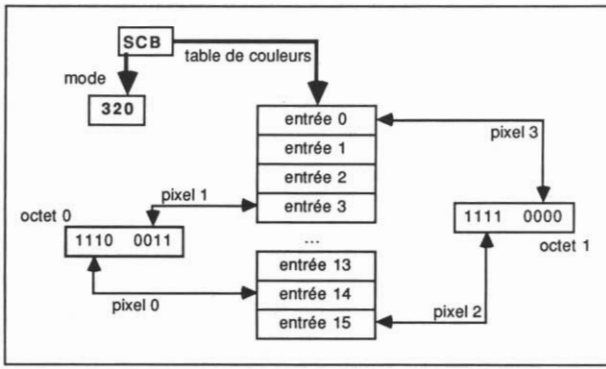


Figure III.1. Le mode 320.

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.

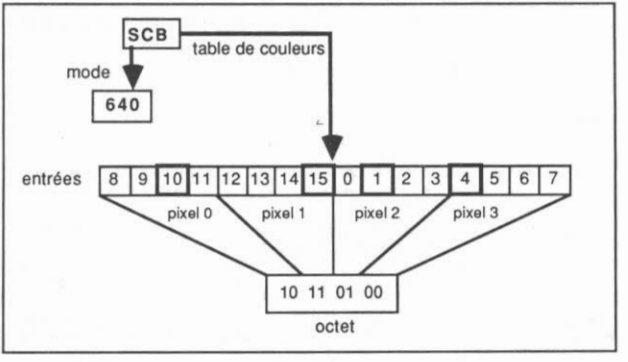


Figure III.2. Le mode 640.

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.

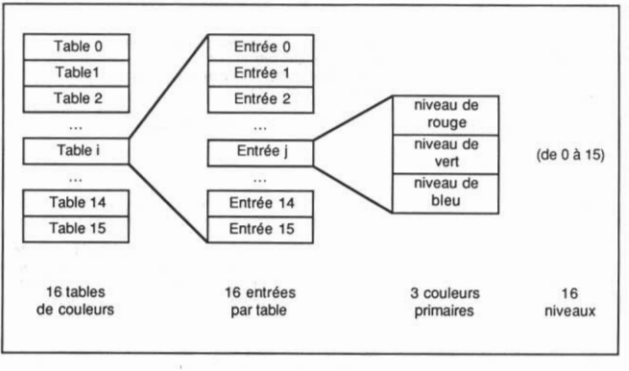


Figure III.3. Tables de couleurs.

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

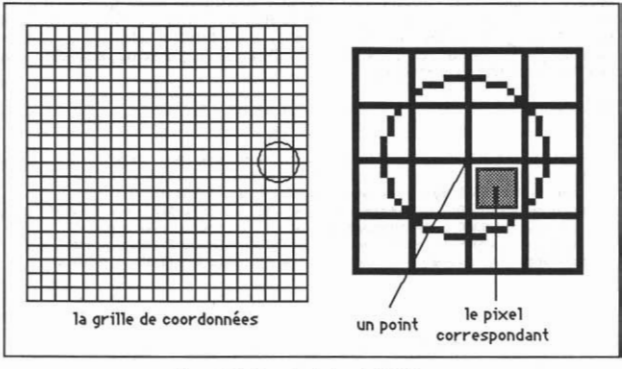


Figure III.4. Concepts de base de QuickDraw.

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 $n \times m$ englobe exactement $(n-1) \times (m-1)$ 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.

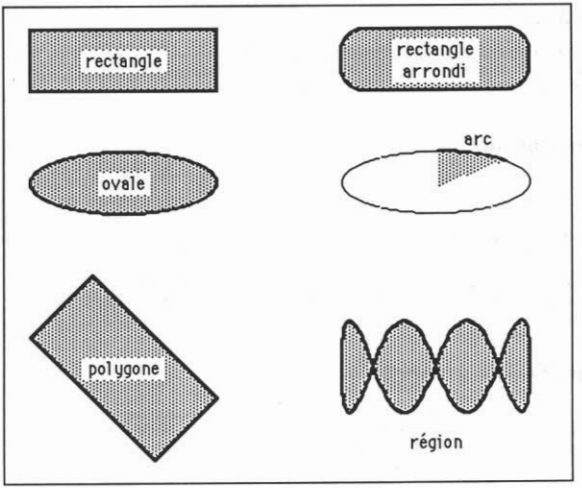


Figure III.5. Concepts géométriques.

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

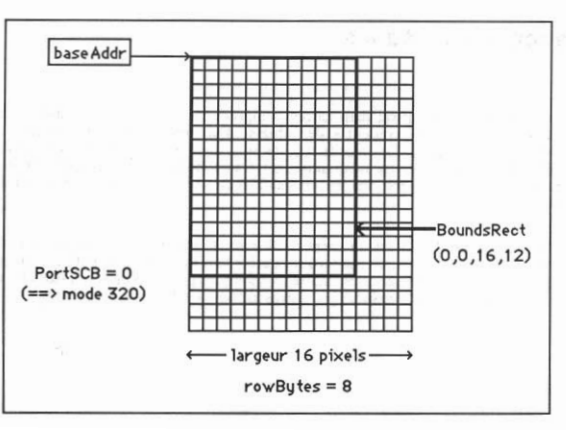
Une pixel map est une portion de mémoire contenant une image graphique composée de pixels, chaque pixel ayant la même taille (2 ou 4 bits). QuickDraw sait dessiner n'importe où, et pas seulement dans la mémoire écran. Il suffit de lui dire où dessiner. Le fait de dessiner ne consiste réellement qu'en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



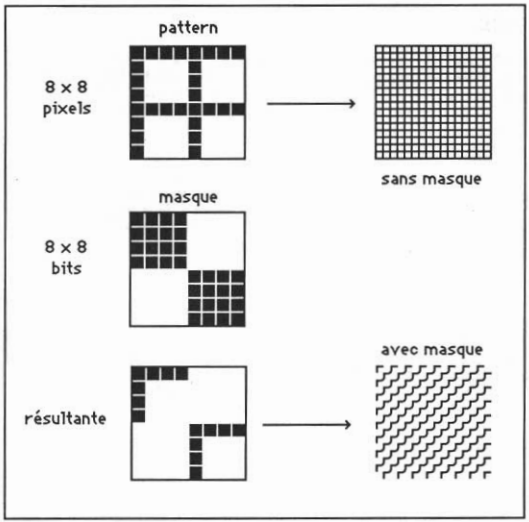


Figure III.7. Pattern et masque.

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).

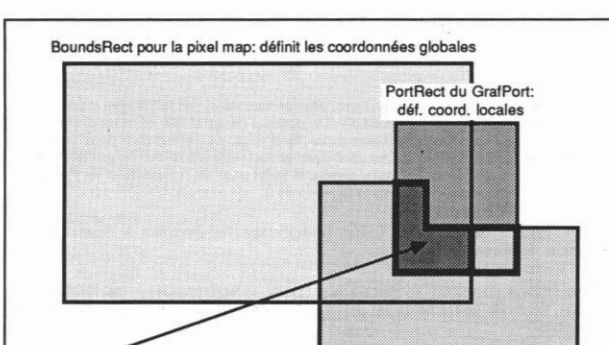


Figure III.8. Les régions du grafport.

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 */
    };
```

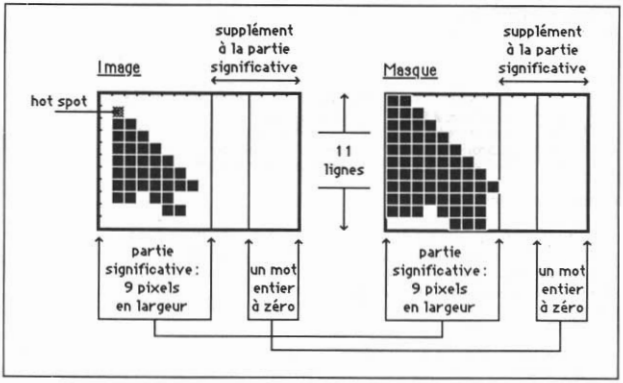


Figure III.9. Le curseur.

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,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x0F,0xF0,0x00,0x00,0x00,
    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,0xFF,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.

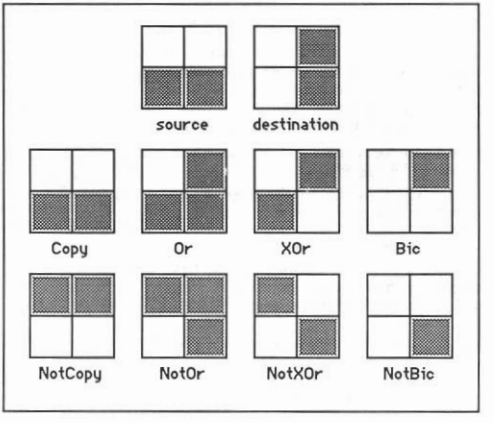


Figure III.10. Les modes de transferts normaux (blanc = 0, gris = 1).

• 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 = GetClipttandle () ;               /* 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.

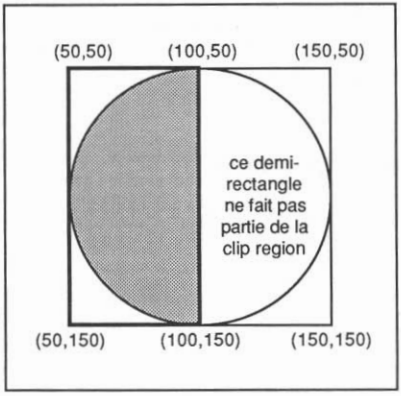


Figure III.11. Dessin d'un demi-cercle.

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 */
```

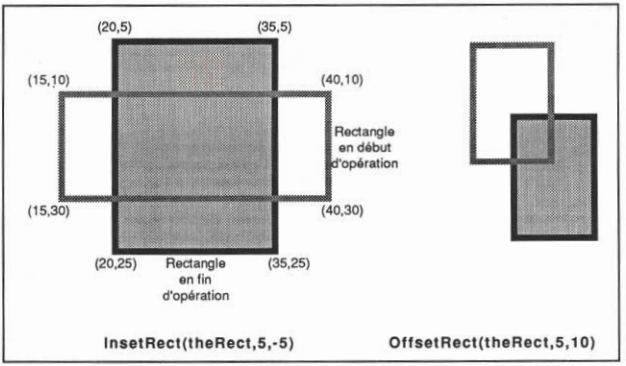


Figure III.12. Déformation et déplacement d'un rectangle.

• 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) */
```

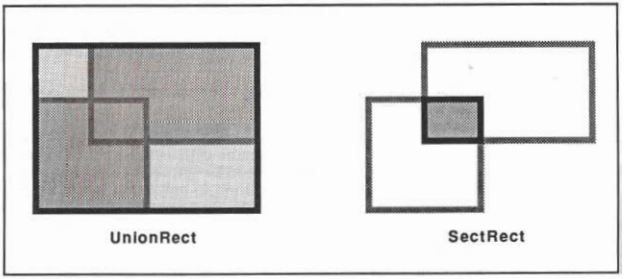


Figure III.13. Union et intersection de deux rectangles.

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.

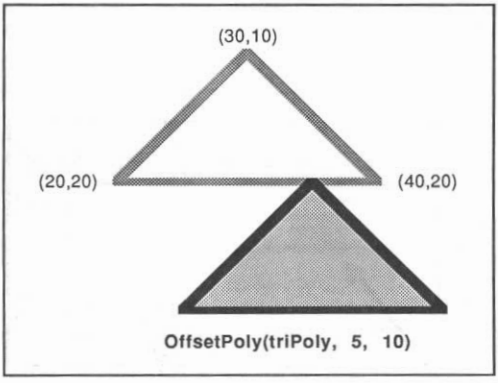


Figure III.14. Déplacement d'un polygone.

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.

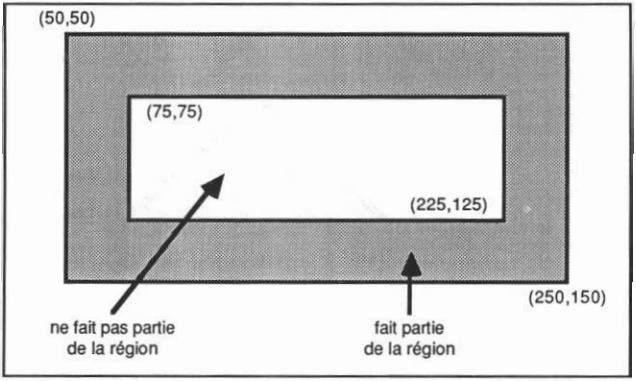


Figure III.15. La région de l'exemple.

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 */
```

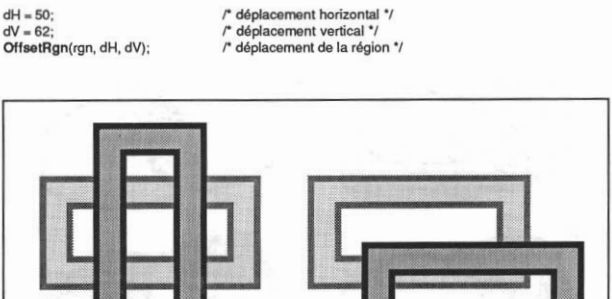


Figure III.16. Déformation et déplacement d'une région

• 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).

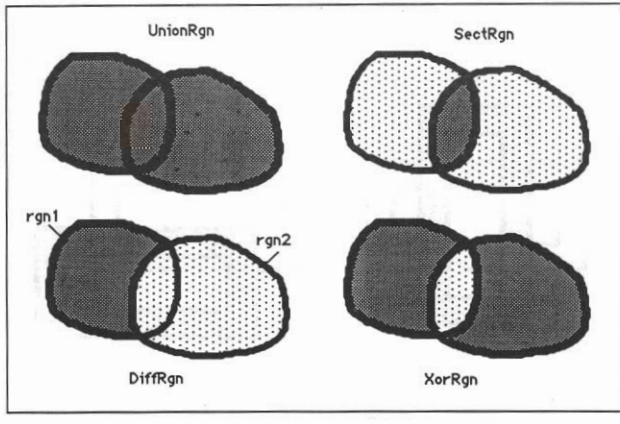


Figure III.17. Union, intersection, différence et « ou » exclusif de deux régions.

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) */
```

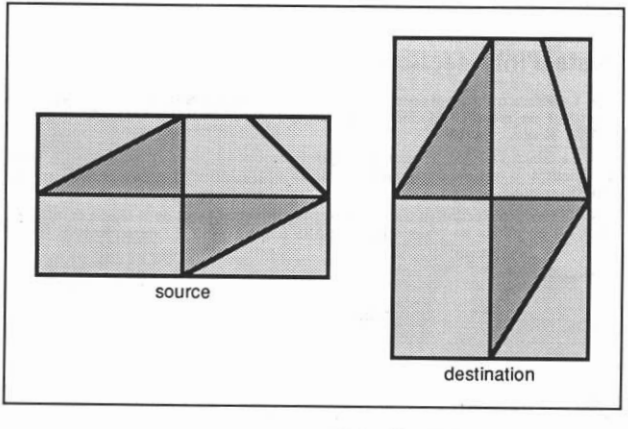


Figure III.18. La mise à l'échelle (procédures Map).

• 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 :

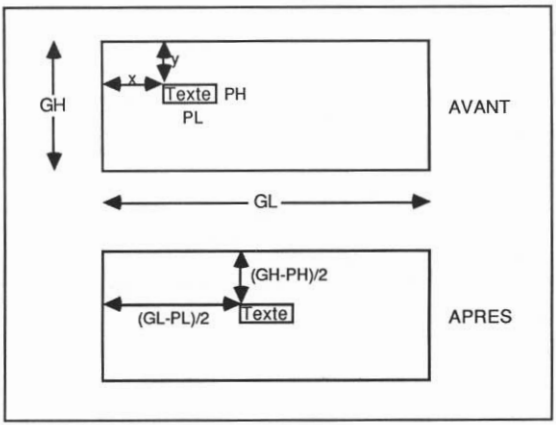


Figure III.19. Centrer un texte.

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) */
```

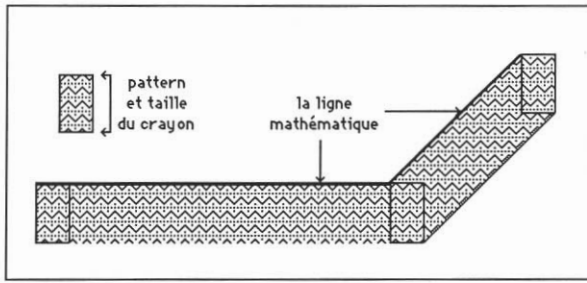


Figure III.20. Dessin de deux lignes.

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*.

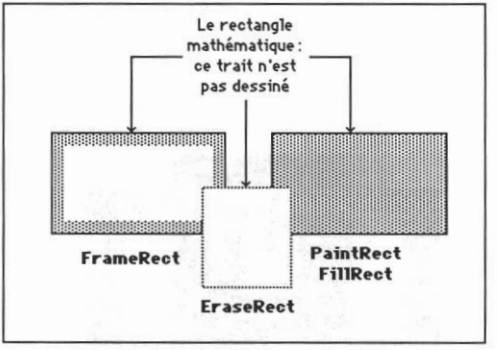


Figure III.21. Dessins de rectangles.

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

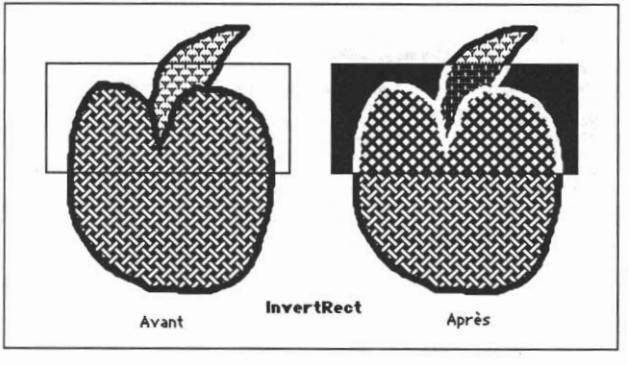


Figure III.22. Inversion du rectangle.

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.

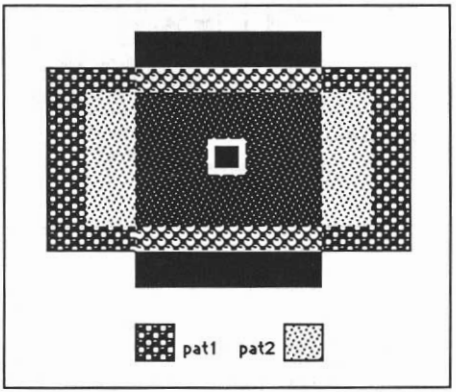


Figure III.23. Dessins avec rectangles.

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.

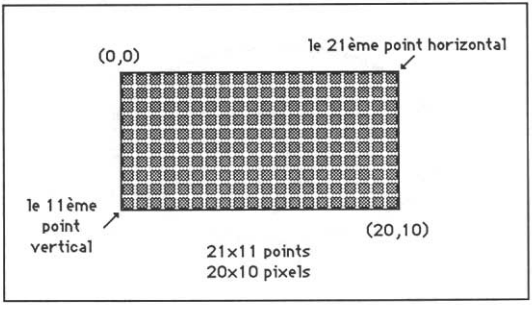


Figure III.24. Les pixels d'un rectangle.

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.

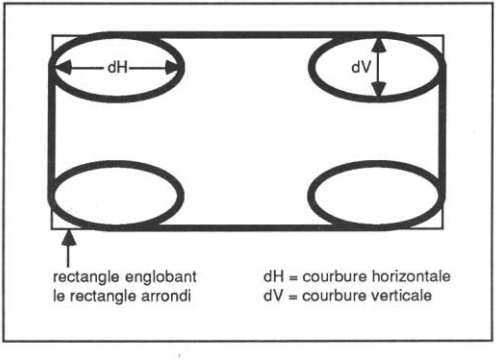


Figure III.25. Les éléments constituant le rectangle arrondi.

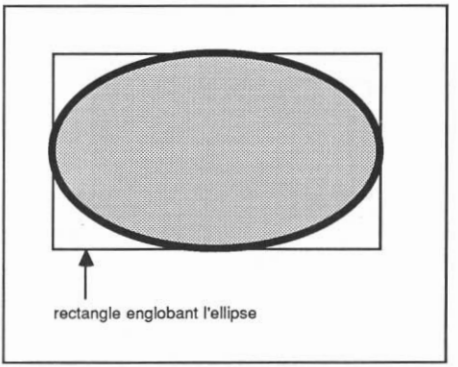


Figure III.26. L'ellipse est parfaitement définie par le rectangle circonscrit.

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.

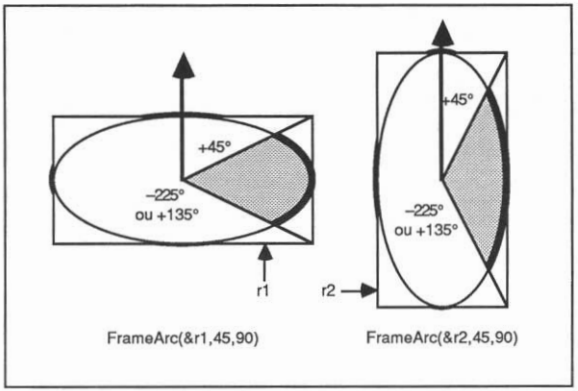


Figure III.27. Les angles sont calculés par référence au rectangle circonscrit.

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.

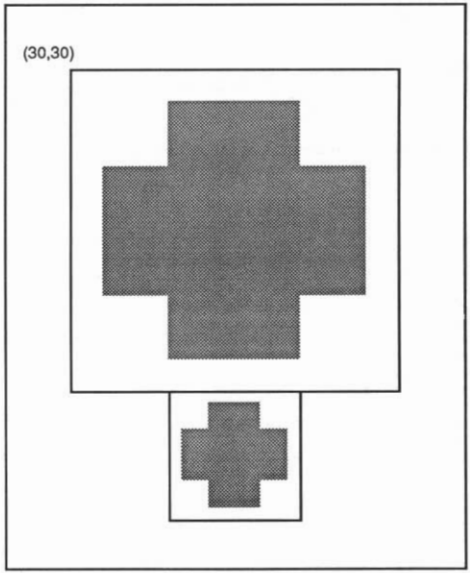
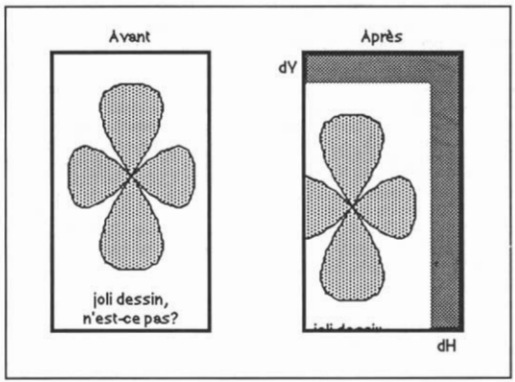


Figure III.28. La Croix-Rouge (imaginez le rouge !).



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 tellement 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 *MouseDown* 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) :

$$|x_2 - x_1| + |y_2 - y_1| < 5 \qquad \text{par exemple.}$$

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.

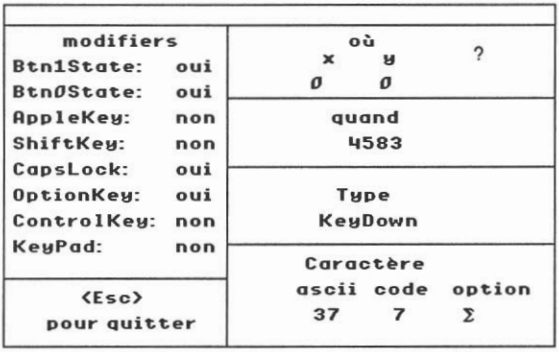


Figure IV. 1. L'écran de l'exemple.

```
#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 :

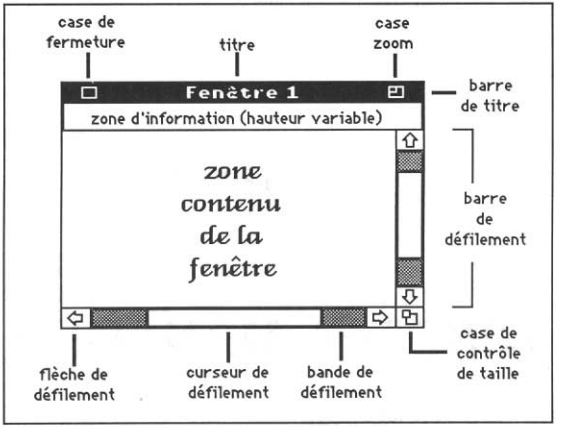


Figure V.1. La fenêtre document standard.

– 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.

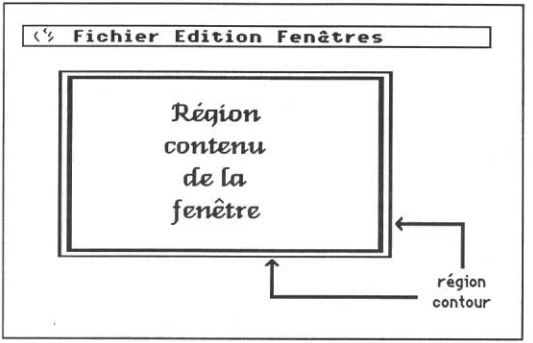


Figure V.2. L'autre type de fenêtre.

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.

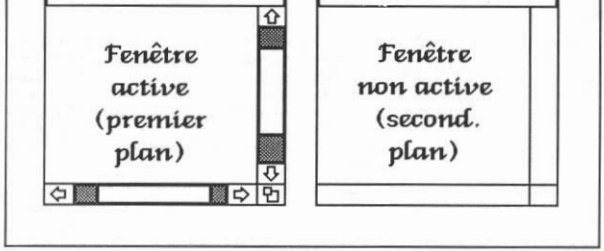
UTILISATION DU WINDOW MANAGER

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.



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*.

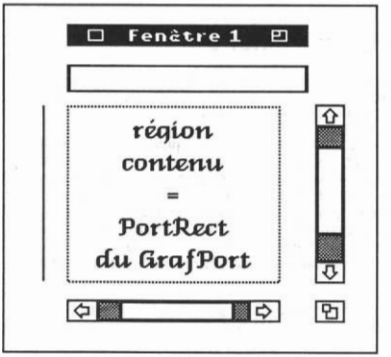


Figure V.4. La fenêtre : un contenu et un contour

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.

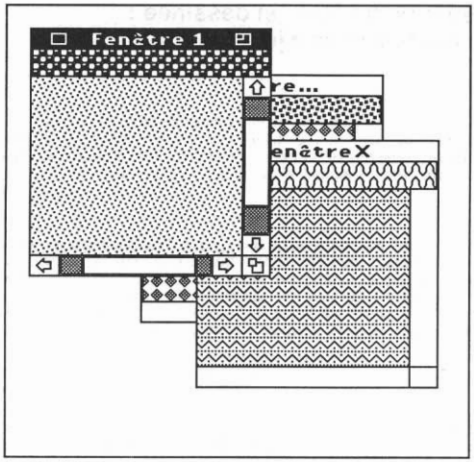


Figure V.5.a. Situation initiale.

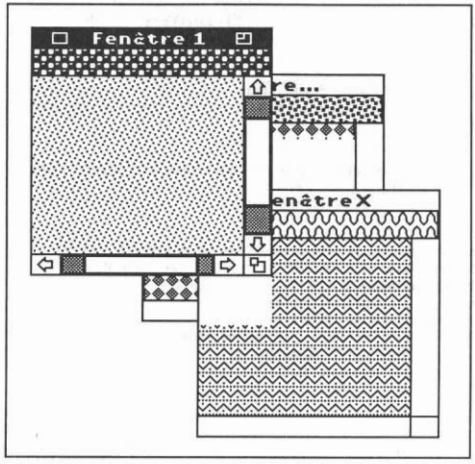


Figure V.5.b. La fenêtre X est déplacée, touche Pomme enfoncée.

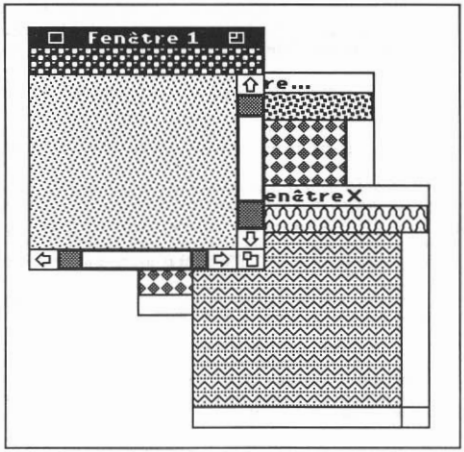


Figure V.5.c. Après les événements de mise à jour.

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.

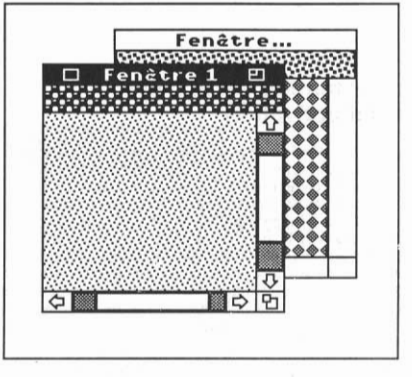


Figure V.6.a. Situation initiale.

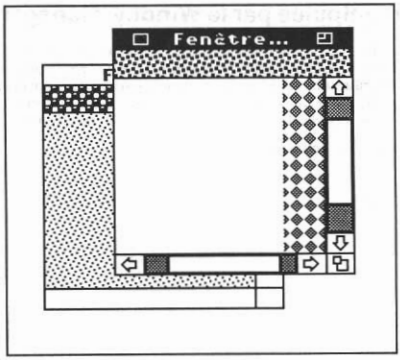


Figure V.6.b. Après les événements d'activation/désactivation.

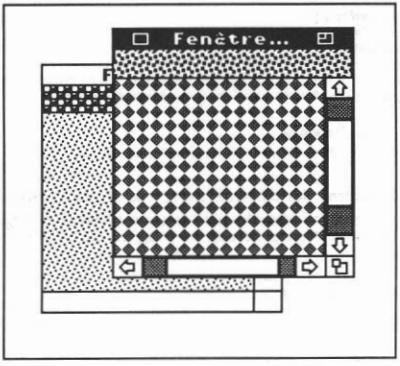


Figure V.6.c. Après l'événement de mise à jour.

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).

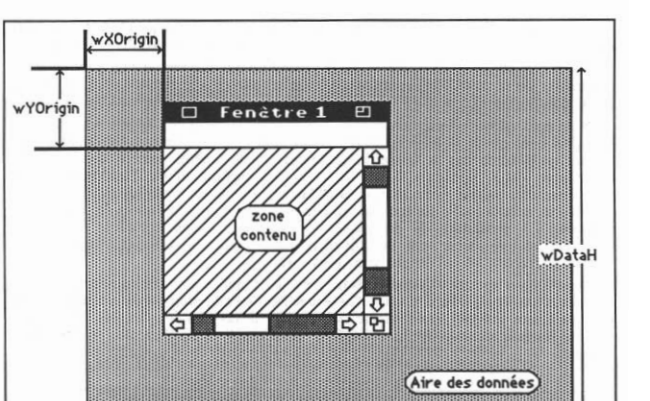


Figure V.7. Contenu de la fenêtre et aire des données.

• *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.

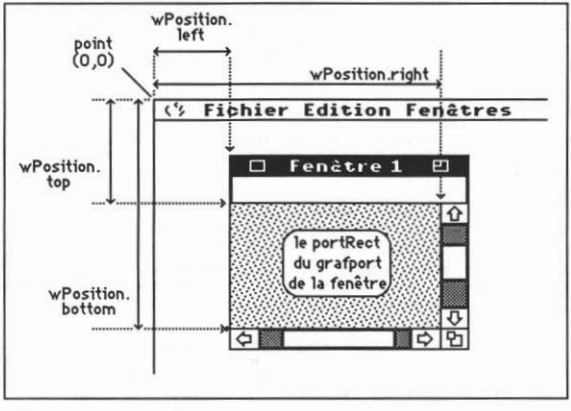


Figure V.8. Position de la fenêtre à l'écran (coordonnées globales).

• *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 !

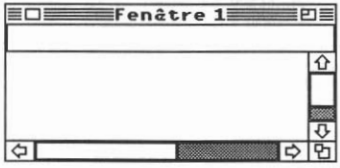


Figure V.9. Barre de titre dans le style Macintosh.

```
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'informations 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** :

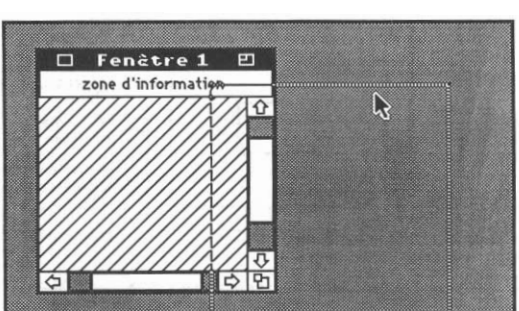


Figure V.10. Fenêtre en cours de déplacement.

– 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.

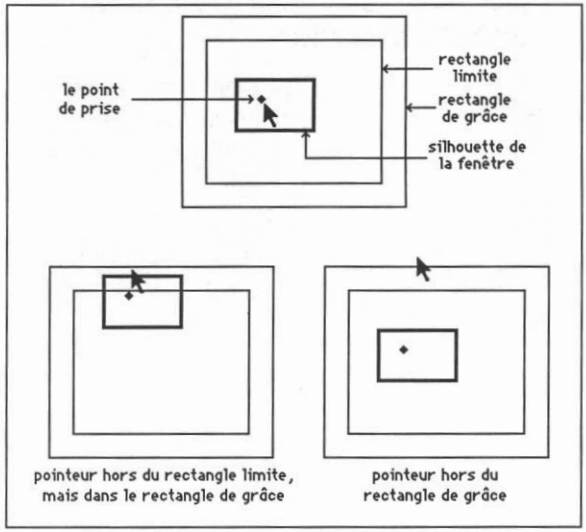


Figure V.11. Rectangle limite et rectangle de grâce.

```
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 en 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).

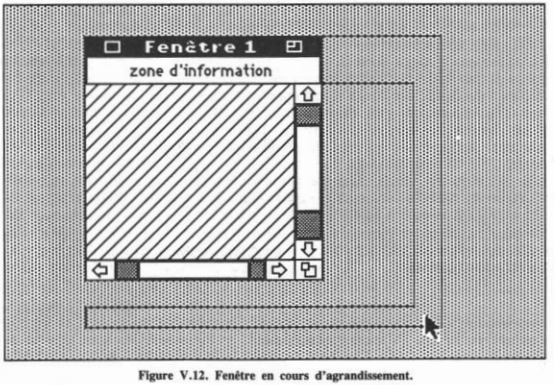


Figure V.12. Fenêtre en cours d'agrandissement.

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 inGoAway :
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 *inZoom*, 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 inZoom :
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 inInfo :
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 !

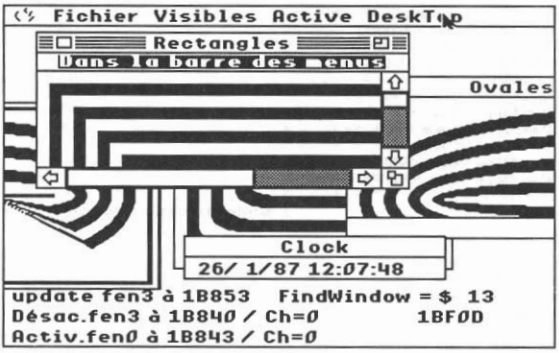


Figure V.13. Ecran tiré de l'exemple.

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,"¶/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).

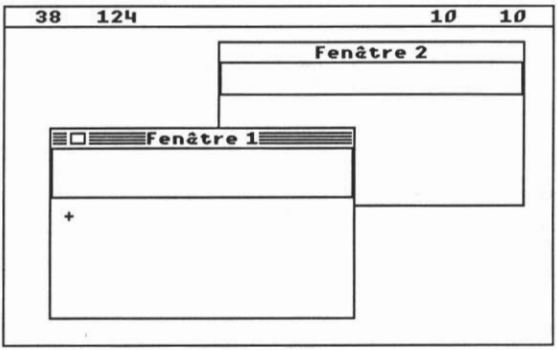


Figure V.14. Ecran tiré de l'exemple.

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 */
}
```