Boris BERTELSONS /
Mathias RASCH

PC INTERDIT

MICRO APPLICATION

Auteurs Boris BERTELSONS / Mathias RASCH

Traducteurs Christophe STEHLY / Marc NICHANIAN

ISBN : 2-7429-0352-6
Ref. DB : 441117

Préface

Il y a deux ans encore, les texture mappings, les Copper list, les sons échantillonnés sur 16 bits et les ombres de Gouraud semblaient encore appartenir à un monde de rêve, qui ne se réaliserait pas avant longtemps.

Les choses ont (heureusement) évolué fondamentalement. Aujourd'hui, ces techniques sont devenues indispensables à tout projet graphique d'envergure, augmentant indirectement, et dans des proportions impressionnantes, les compétences minimales exigées des développeurs.

Alors que tout le monde se contentait des effets classiques, le public averti des derniers progrès techniques ne cesse de s'élargir et de réclamer ce qu'il considère dorénavant comme un minimum. L'époque du développeur génial et isolé, travaillant avec acharnement, est terminée, laissant la place à des équipes de développeurs, musiciens, graphistes et designers, qui doivent travailler avec le même acharnement, mais en groupe. Une démo n'est plus un projet de deux jours, elle requiert des temps de développement du même ordre de grandeur que les applications et les jeux.

Cette évolution était difficilement envisageable sur le PC, qui était il n'y a pas si longtemps, le cheval de trait des travaux administratifs et non la machine polyvalente d'aujourd'hui. La souplesse du concept du PC est telle qu'il devient pratiquement impossible de tracer des limites à ses domaines d'application futurs.

Un signe qui ne trompe pas est le changement d'attitude des utilisateurs d'autres systèmes : après avoir considéré le monde du PC avec condescendance, leur regard se teinte d'envie. L'image multitalent du PC provient, davantage que des progrès de la technique, des nombreux utilisateurs enthousiastes, ne se lassant jamais de tenter de nouvelles expériences pour améliorer la machine et les logiciels existants. Ceci explique d'ailleurs que les géants des logiciels fassent de plus en plus fréquemment appel à des équipes auteurs de démos pour leurs nouveaux projets.

Pour les fanas, l'attrait du gain est largement dépassé par le prestige et par la reconnaissance de leurs pairs, qui ne manquent pas de saluer les projets réussis. Mais attention : si vous pensez avoir votre mot à dire, vous devez savoir de quoi vous parlez. La passion de ces fanas a fait du PC ce qu'il est aujourd'hui. Sans les compétences et les milliers d'heures de ces précurseurs, le niveau technique de nos PC serait certainement inférieur à ce qui est devenu la norme.

En quoi suis-je concerné par ce sujet ?

La question mérite d'être posée. La programmation des cartes graphiques du PC est une longue histoire. L'établissement de la carte VGA comme norme de facto a fait évoluer la situation car elle a enfin permis de considérer le PC comme autre chose qu'un appareil de saisie et d'affichage de texte. Le potentiel des cartes VGA permet d'effectuer des opérations graphiques très complexes, dont nous bénéficions sous forme d'effets visuels particulièrement esthétiques. L'arrivée des cartes son vient confirmer cette évolution vers une machine plus conviviale. L'utilisation de ces cartes peut être considérée comme un progrès parvenu très rapidement à maturité.

L'histoire du son sur le PC a débuté avec la carte AdLib, encore utilisée dans de nombreuses applications. Les sons synthétisés par modulation de fréquence ont rapidement montré leurs limites, et l'échantillonnage est venu renforcer la puissance des cartes son. Actuellement, ces cartes sont équipées de tables d'ondes (Wave Tables) dont la qualité sonore est comparable à celle des enregistrements en studio. La programmation des cartes son, qu'elles soient à synthèse FM ou à tables d'ondes, demeure fascinante. L'harmonie des projets de programmation des sons repose sur l'usage équilibré de la vision et du son.

Après ces paragraphes généraux, revenons au contenu du livre, qui a pour ambition de rapprocher le lecteur ou la lectrice de ces sujets relativement complexes.

Rédigé et conçu par des acteurs en prise sur l'évolution du PC, ce livre traite de la programmation vectorielle d'objets sur lesquels des textures sont plaquées. A première vue, l'utilisation de la technique du placage de textures (Texture Mapping) est une matière complexe, mais elle vaut la peine que l'on y accorde toute son attention car elle est très appréciée par le public et permet de réaliser des effets d'un réalisme inconnu jusqu'ici.

L'accompagnement, suggéré plus haut, par une musique et par des effets sonores adéquats trouve son application dans les players de fichiers MOD. Le format MOD des fichiers musicaux est la norme actuellement en vigueur. Quand on l'a compris, il devient très facile d'utiliser les connaissances acquises avec d'autres formats. Ceci n'est naturellement qu'un petit aperçu de ce que vous réserve ce livre. La programmation de Voxels, d'effets wobbler, de feu et l'assimilation des procédés de mots de passe sont d'autres facettes intéressantes de l'ouvrage, sans oublier l'écriture de programmes "Game Trainers".

Qu'est-ce qui m'attend ?

Vous l'avez compris après avoir lu les lignes qui précèdent, le livre ***PC Interdit*** n'est pas un manuel ordinaire. Il ne donne aucune solution toute prête pour telle ou telle application, ni d'algorithmes passe-partout. Au contraire, ce livre tente de faire partager au lecteur une partie de la fascination qui dévore les nuits des gourous d'aujourd'hui. Des connaissances élémentaires en Pascal et en Assembleur sont indispensables à la compréhension de certaines parties. Que ceci n'effraie pas le débutant, ce livre restera une source de références durant toute l'acquisition de vos connaissances.

Assez parlé, amusez-vous bien, de la part de

Brian, a.k.a Michael Diefenthaeler, Legend Design

PS : Special Greets fly to :

> *All Members of Blank, DUST, KLF, Legend Design, THE COESISTENCE & Xography, to Alpha. AXL, Angelika, Atan, Daryl. AD, Dose, Ela, Frank B., Ines, Killer Voop, Soft Ones, Moppel, Osti & Co., Pogo, Roger, SSL, Sandra, Stepp-Vera, Wolfgang*

Greets fly to :

> *Access Denied, Admire, Anarchy, Cascada VRS, Epical, Extreme, Gollum, Grif, Iguana, Impact, Infiny, Imp/Cda, Majic 12, Mental Design, Silicon Worlds! Prods., Tran, Triton*

Sommaire

1. A propos de bytes et de freaking : le langage de la scène

Ces derniers temps, la "scène" a vu se développer un langage particulier. Un profane qui écouterait une conversation entre deux membres de la scène ne comprendrait pas de quoi ils s'entretiennent. Voici un exemple de dialogue :

Coder à GFX-Man :

```
"Quand finis-tu le graphique pour le nouveau BBS-ADDY ?"
```

GFX-Man à Coder :

```
"Je te le charge cette nuit. Logue-toi dans WHQ. Tu y trouveras un
messy de ma part avec  les infos de codage"
```

Coder à Music-man :

```
"Où en es-tu avec ta 'Mucke' ?"
```

Music-man à Coder :

```
"Je t'appelle per voice et te l'envoie"
```

C'était un extrait d'une conversation typique entre membres de la scène, appartenant à l'un des groupes de démos du genre "The CoExistence", "Legend Design", "Xography" ou "KLF".

Un individu "normal" ne peut pas suivre un tel échange, dans la plupart des cas il sera incapable d'en interpréter la teneur à moins qu'il ne tienne carrément les interlocuteurs pour fous. Pour vous éviter cette déconvenue, nous avons rassemblé ci-dessous quelques expressions du vocabulaire de base de la scène.

Intro	Effets graphiques, parfois avec sons, taille de 4 Ko au maximum
Démo	Effets graphiques avec animations, taille de 100 Ko au maximum
Mégadémo	Effets graphiques avec animations, de taille quelconque
BBS-Addy	Petit programme de pub dans une boîte aux lettres de messagerie électronique, 64 Ko au maximum
WHQ	World Headquarter, boîte aux lettres servant de quartier général à un groupe de démos (au niveau mondial)
EHQ	European Headquarter, idem au niveau européen
Mucke	Musique
Voice	Appel téléphonique normal, pas de modem
Call	L'appel téléphonique proprement dit
Upload, up	Télécharger des fichiers (en émission)
Download, down	Télécharger des fichiers (en réception)
BBS	Bulletin Board System, messagerie électronique
Coder	Programmeur
GFX-Man	Graphiste du groupe de démos
Music-Man	Musicien du groupe de démos
Messy	Message
VectorBobs	Objets graphiques se déplaçant librement dans l'espace 3D
ShadeBobs	Petits objets graphiques fantomatiques
Anim	Animation, image graphique mobile
Scrolly	Texte défilant dans une démo ou une intro

2. Le véritable langage des programmeurs de la scène : l'assembleur

Même à l'époque des langages de programmation structurés de haut niveau comme Pascal ou C et en dépit de l'avènement des systèmes d'exploitation orientés utilisateurs comme Windows, l'assembleur conserve sa raison d'être. Proche du langage machine, l'assembleur donne accès au coeur de l'ordinateur comme aucun langage orienté vers la résolution des problèmes n'est capable de le faire.

Certaines tâches ne peuvent être programmées qu'en assembleur. Ainsi, Pascal permet de maîtriser correctement le maniement des interruptions tant qu'on n'implique pas le gestionnaire d'origine pour, par exemple, détourner les entrées au clavier. Il n'existe aucune instruction prévue à cet effet, il faut donc faire appel à un module en assembleur. Vous pouvez toujours essayer de retirer de la mémoire un programme résident : en pur Pascal, ce n'est pas possible. L'assembleur est incontournable.

Aujourd'hui encore, la taille d'un programme peut jouer un rôle crucial. Ainsi, lorsqu'on met au point un programme résident, il ne faut pas gaspiller les octets de mémoire situés en dessous des 640 Ko conventionnels. Les langages évolués qui traînent inévitablement avec eux des bibliothèques de plusieurs Ko se trouvent défavorisés par rapport à l'assembleur qui n'exploite que la stricte mémoire véritablement nécessaire.

Mais le plus grand avantage de l'assembleur est sa vitesse d'exécution. S'il est vrai que les langages évolués ont fait des progrès considérables sur le plan de l'optimisation des exécutables, une optimisation automatique ne peut se substituer à l'expérience du programmeur.

Pour ne mentionner qu'un seul exemple : si en Pascal on met à zéro deux variables qui se suivent, le compilateur génère le code suivant :

```
xor ax,ax
mov var1,ax
xor ax,ax
mov var2,ax
```

Ici, Pascal "optimise" bien la mise à zéro du registre ax (avec l'instruction XOR) mais la rigidité du compilateur conduit à l'introduction d'un deuxième XOR complètement inutile.

Ainsi, les parties de programme critiques comme celles qui s'occupent par exemple de la représentation des Sprites (objet graphique mobile) devront obligatoirement être codées en assembleur. Sur ce terrain, deux approches sont possibles. On peut se servir de l'assembleur interne de Borland Pascal (directives asm) ou faire appel à un assembleur externe (TASM, MASM). Ces méthodes ont chacune leurs avantages et leurs inconvénients, mais la plupart du temps il est préférable de recourir à l'assembleur externe.

Ce dernier est par nature mieux apte à exploiter les subtilités du langage machine. Il dispose par ailleurs de possibilités accrues. Une directive du type db 20 dup facilite grandement la vie, et ne peut pas être écrite avec l'assembleur interne. Les macros constituent également un avantage appréciable offert par les assembleurs externes. Aussi souples d'usage que les sous-programmes, elles sont meilleures sur le plan de la vitesse d'exécution. C'est pourquoi, dans cet ouvrage, nous aurons presque toujours recours à un assembleur externe. La seule situation dans laquelle l'assembleur interne est préférable à la version externe se présente lorsqu'une partie de programme écrite en assembleur doit accéder à des variables locales à l'intérieur d'une procédure (comme c'est le cas par exemple dans les procédures GetSprite et PutSprite de l'unité Sprites). Si Borland pouvait rechercher une solution pour ce type de situation...

Bien que l'assembleur soit par définition très rapide, il est facile de gaspiller ce bénéfice en écrivant un code maladroit. Très souvent, les débutants imitent des constructions héritées du Basic dont les effets sur la vitesse d'exécution ressemblent à ceux produits par l'optimisation automatique d'un langage évolué. Vous trouverez dans les chapitres suivants quelques conseils de base pour aborder efficacement ce type de programmation.

2.1. Multiplications et divisions en assembleur

Bien que les multiplications soient traitées très vite par les processeurs récents (en six cycles d'horloges par un 486 DX4 ou un Pentium), il vaut mieux les remplacer par des décalages avec la fonction Shift lorsqu'elles portent sur des puissances de deux. Le nombre de bits à décaler correspond à l'exposant en base 2 du multiplicateur. Ainsi, une multiplication par 16 demandera un décalage de 4 bits, car 2 puissance 4 égale 16. Pour multiplier par 8 le contenu du registre AX, le mieux est d'écrire l'instruction :

```
SHL AX,3
```

Chaque bit se trouve alors déplacé dans une position qui multiplie sa valeur par huit (2 puissance 3). Les divisions se traitent de la même façon, en introduisant un décalage vers la droite.

On conseillait autrefois de procéder à des décalages systématiques, même pour des nombres qui n'étaient pas exactement des puissances de deux. Ainsi, une multiplication par 320 était décomposée en une multiplication par 256 (décalage de 8 bits) suivie d'une autre par 64 (décalage de 6 bits). Après sommation des deux produits partiels, on obtenait le résultat recherché. Cette décomposition se justifiait peut-être encore à l'époque des XT qui consommaient plus de 100 cycles d'horloge pour effectuer une multiplication. Mais avec l'apparition du processeur 286, le nombre de cycles est descendu à une vingtaine, de sorte que cette manière de procéder devient plutôt une cause de ralentissement.

2.2. Calculs en virgule fixe

Les calculs ne peuvent pas toujours porter sur des variables entières. Lorsqu'on trace une droite, sa pente s'exprime rarement par un nombre entier. Dans ces cas, on fait normalement appel à des nombres de type real (Pascal) ou float (C) qui reposent sur la représentation appelée "virgule flottante". La virgule décimale peut se placer librement à l'intérieur d'un certain nombre de positions (de là l'expression virgule flottante). Mais le maniement de ces nombres est affreusement lent. Les processeurs ordinaires ne disposent pas d'instructions spécialisées. En l'absence de coprocesseur, chaque calcul doit se faire par un sous-programme, ce qui est très pénalisant sur le plan de la vitesse.

Par ailleurs, ce type de nombre est très difficile à manipuler en assembleur. Il faudrait utiliser les opérations en langage évolué mais ce n'est pas simple car les opérations de base ne sont pas déclarées "public" en Pascal. Ou alors écrire ses propres procédures ? Mais l'une et l'autre de ces méthodes nécessitent un investissement important. On est donc conduit à rechercher une alternative.

En fait, la plupart du temps, la précision des nombres en virgule flottante est quelque peu excessive. Qui a vraiment besoin de onze chiffres significatifs ? Si on ne parcourt pas des intervalles énormes, il est parfaitement raisonnable de renoncer aux virgules flottantes. La solution passe donc par les nombres à virgule fixe.

Un tel nombre se compose structurellement de deux entiers. Le premier représente la partie entière qui se trouve avant la virgule. Le second est constitué par les décimales, étant entendu qu'il faut en préciser le nombre. Si la partie entière est par exemple 17, et la partie décimale 1, on aura le nombre 17,01 s'il existe deux décimales (facteur décimal 100) et 17,1 avec une seule décimale (facteur décimal 10).

Avec cette représentation, il faut aussi s'habituer à une autre façon de voir les signes. Ecrit en virgule fixe, le nombre -100,3 est décomposé en -100 et -3 (avec une seule décimale ou un facteur décimal de 10). En effet, les deux nombres partiels doivent donner par addition le nombre d'origine. Si on fait la somme de -100 et -0,3, on retombe bien sur le nombre de départ.

L'avantage de cette représentation est évident : les traitements vont concerner des nombres entiers. Lorsque la partie décimale déborde, on augmente la partie entière. Le processeur est capable de traiter avec brio et célérité les nombres entiers, ceci en l'absence de tout coprocesseur et avec relativement peu de cycles d'horloge.

Bien entendu, les procédures de calcul doivent être programmées manuellement, étant donné que le processeur ne connaît pas à proprement parler les nombres à virgule fixe. Mais la représentation en virgule fixe est si proche du monde des entiers que la programmation des différentes opérations ne pose pas grand problème. Parmi celles-ci, on peut évidemment prévoir des fonctions aussi compliquées que l'extraction d'une racine carrée par une méthode itérative : le gain en vitesse d'exécution sera alors très sensible.

Les quatre opérations de base

Les nombres à virgule fixe restant proches des nombres entiers, il n'est pas difficile d'adapter à leur intention les quatre opérations de base. Le processeur dispose des instructions appropriées, il suffit de relier correctement les composants entier et décimal.

Pour en démontrer le principe, nous allons présenter dans ce chapitre un programme qui implémente les quatre opérations de base en Pascal. On pourrait encore accroître considérablement la vitesse d'exécution en effectuant la programmation en assembleur mais le but de cet exemple est simplement de montrer la méthode.

Addition

L'addition est l'opération la plus simple à mener à bien. Il suffit d'additionner respectivement les parties entières et décimales. Il peut se produire un débordement lorsque l'addition des parties décimales donne une somme supérieure à 1 (soit 100 si le nombre de décimales est de 2). Dans ce cas, il suffit d'augmenter de 1 la partie entière et de diminuer de 1 (c'est-à-dire de 100 dans notre exemple) la partie décimale. Avec les nombres négatifs, le processus s'inverse : il faut intercepter les dépassements en dessous de -1 et diminuer en conséquence la partie entière.

Soustraction

Le principe est le même que pour l'addition mais les nombres partiels sont alors soustraits l'un de l'autre au lieu d'être additionnés l'un à l'autre.

Multiplication

Pour la multiplication, on a choisi dans cet exemple une voie un peu différente. Multiplicande et multiplicateur sont convertis en nombres entiers, ces derniers sont multipliés l'un par l'autre puis le produit est reconverti en virgule fixe. Il faut cependant prendre la précaution d'effectuer une division par le facteur décimal car celui-ci intervient une fois de trop dans la conversion en nombres entiers.

Division

La division est effectuée selon le même principe que la multiplication. Les versions entières des deux nombres sont divisées l'une par l'autre avec multiplication du dividende par le facteur décimal pour garantir la précision (n'oublions pas que par définition le résultat ne comporte pas de décimales).

Le programme OPBASE.PAS effectue la démonstration de ces méthodes :

```
Type Fixe=Record                {Structure d'un nombre à virgule fixe}
           PEnt,
           PDec:Integer
         End;
Var Var1,                       {Variables d'exemple }
    Var2:Fixe;

Const FacDec=100;               {2 chiffres après la virgule }
      NbDec=2;
 Function Strg(FNombre:Fixe):String;
 {transforme un nombre à virgule fixe en chaîne de chiffres}
Var PDec_Str,                   {Sous-chaîne de la partie décimale}
    PEnt_Str:String;            {Sous-chaîne de la partie entière}
    i:Word;
Begin
  If FNombre.PDec < 0 Then      {pas de signe après la virgule}
    FNombre.PDec:=-FNombre.PDec;
  Str(FNombre.PDec:NbDec,PDec_Str);
                                {crée la sous-chaîne décimale
                                 (après la virgule)}
  For i:=0 to NbDec do          {remplace les espaces par des 0}
    If PDec_Str[i] = ' ' Then PDec_Str[i]:='0';
  Str(FNombre.PEnt,PEnt_Str);   {crée la sous-chaîne de la partie
                                 entière
                                 avant la virgule}
  Strg:=PEnt_Str+','+PDec_Str;  {réunit les deux sous-chaînes}
End;

 Procedure Convert(RNombre:Real;Var FNombre:Fixe);
{convertit un nombre Real RNombre en nombre à virgule fixe FNombre}
 Begin
   FNombre.PEnt:=Trunc(RNombre);
                                {Partie entière}
   FNombre.PDec:=Trunc(Round(Frac(RNombre)*FacDec));
                                {Partie décimale stockée comme
                                 entier }
 End;
```

```
 Procedure Adjust(Var FNombre:Fixe);
 {remet le nombre à virgule fixe transmis en format légal}
Begin
  If FNombre.PDec > FacDec Then Begin
    Dec(FNombre.PDec,FacDec);     {si la partie décimale est
                                   trop grande}
    Inc(FNombre.PEnt);            {la diminuer et augmenter la
                                   partie entière  }
  End;
  If FNombre.PDec < -FacDec Then Begin
    Inc(FNombre.PDec,FacDec);     {si la partie décimale est trop
                                   petite}
    Dec(FNombre.PEnt);            {l'augmenter et diminuer la
                                   partie entière }
  End;
End;

Procedure Add(Var Somme:Fixe;FNombre1,FNombre2:Fixe);
{Additionne FNombre1 und FNombre2 et mémorise le résultat dans Somme}
Var Resultat:Fixe;
Begin
  Resultat.PDec:=FNombre1.PDec+FNombre2.PDec;
                                  {Somme les parties décimales}
  Resultat.PEnt:=FNombre1.PEnt+FNombre2.PEnt;
                                  {Somme les parties entières}
  Adjust(Resultat);
                                  {Traduit le résultat dans un
                                   format correct }
  Somme:=Resultat;
End;

Procedure Sub(Var Difference:Fixe;FNombre1,FNombre2:Fixe);
{Soustrait FNombre2 de FNombre1 et mémorise le résultat dans la
 variable Difference }
 Var Resultat:Fixe;
Begin
  Resultat.PDec:=FNombre1.PDec-FNombre2.PDec;
                                 {Soustraction des parties décimales }
```

```
  Resultat.PEnt:=FNombre1.PEnt-FNombre2.PEnt;
                                {Soustraction des parties entières }
  Adjust(Resultat);             {Traduit le résultat dans un
                                 format correct }
  Difference:=Resultat;
End;

Procedure Mul(Var Produit:Fixe;FNombre1,FNombre2:Fixe);
{Multiplie FNombre1 par FNombre et mémorise le résultat dans la
 variable Produit }
Var Resultat:LongInt;
Begin
  Resultat:=Var1.PEnt*FacDec + Var1.PDec; {Multiplicande}
  Resultat:=Resultat * (Var2.PEnt*FacDec + Var2.PDec);
                                {Multiplicateur}
  Resultat:=Resultat div FacDec;
                                {Redivise par le facteur décimal
                                 FacDec }
  Produit.PEnt:=Resultat div FacDec;
                                {Extrait les parties entières et
                                 décimales }
  Produit.PDec:=Resultat mod FacDec;
End;

Procedure Divi(Var Quotient:Fixe;FNombre1,FNombre2:Fixe);
{divise FNombre1 par FNombre2 et mémorise le résultat dans la
 variable Quotient }
Var Resultat:LongInt;           {Résultat intermédiaire}
Begin
  Resultat:=FNombre1.PEnt*FacDec + FNombre1.PDec;    {Dividende... }
  Resultat:=Resultat * FacDec div (FNombre2.PEnt*FacDec+
                                   FNombre2.PDec);
                                {... multiplié par le facteur
                                 décimal pour plus de précision
                                 et divisé par le diviseur }
  Quotient.PEnt:=Resultat div FacDec; {Extrait les parties entières
                                       et décimale }
```

```
  Quotient.PDec:=Resultat mod FacDec;
End;

Begin
  WriteLn;
  Convert(-10.2,Var1);          {Deux nombres pour la démonstration)}
  Convert(25.3,Var2);
                                {Quelques calculs pour tester
                                 le programme:}
   Write(Strg(Var1),'*',Strg(Var2),'= ');
  Mul(Var1,Var1,Var2);
  WriteLn(Strg(Var1));

  Write(Strg(Var1),'-',Strg(Var2),'= ');
  Sub(Var1,Var1,Var2);
  WriteLn(Strg(Var1));

  Write(Strg(Var1),'/',Strg(Var2),'= ');
  Divi(Var1,Var1,Var2);
  WriteLn(Strg(Var1));

  Write(Strg(Var1),'+',Strg(Var2),'= ');
  Add(Var1,Var1,Var2);
  WriteLn(Strg(Var1));
End.
```

Addition, soustraction, multiplication et division ont été implémentées dans les procédures Add, Sub, Mul et Divi en appliquant les méthodes décrites. Le programme principal effectue quelques calculs en chaîne à titre de démonstration. En quatre étapes et avec seulement deux décimales, on ne constate aucune erreur d'arrondi.

Les ajustements nécessaires après l'addition et la soustraction sont pris en charge par une procédure nommée Adjust. Convert transforme un nombre à virgule flottante en nombre à virgule fixe. A partir d'un nombre à virgule flottante, Strg produit une chaîne qui servira notamment à des fins d'affichage.

Exemple d'application

Le programme précédent montrait le fonctionnement des quatre opérations de base sur les nombres à virgule fixe. Le programme qui suit en propose une application. Le développement porte sur un algorithme de tracé de segment qui en termes d'efficacité se rapproche de la méthode de Bresenham. Notre algorithme va fonctionner en fait avec des formats fixes mais de façon peu apparente.

La méthode s'appuie sur l'une des définitions mathématiques les plus simples de la droite, à savoir l'équation y=ax+b. Nous nous préoccuperons essentiellement de la pente a qui donne l'accroissement vertical de la droite pour un accroissement horizontal de 1. Ce paramètre étant rarement entier, les calculs en virgule fixe trouvent ici une application particulièrement heureuse. La procédure d'exemple appelée Line est capable de tracer des segments dont la pente est comprise entre 0 et 1. Pour d'autres valeurs, il faudrait simplement introduire des symétries (cf chapitre 8.4 : Modèles en fil de fer avec l'algorithme de Bresenham).

Ce programme exploite la procédure PutPixel décrite au chapitre 4.2 lorsque nous ferons défiler des étoiles. Disons simplement ici qu'elle dessine en mode 13h un point de coordonnées (x, y) dans la couleur col.

Sur le CD, vous trouverez le programme correspondant sous le nom de SEGMENT.PAS, écrit en assembleur.

```
Uses Crt;

Var x:Word;

Procedure PutPixel(x,y,col:word);assembler;
{dessine le point (x,y) en couleur col (Mode 13h)}
asm
  mov ax,0a000h                    {Charge le segment}
  mov es,ax
  mov ax,320                       {Offset = Y*320 + X}
  mul y
  add ax,x
  mov di,ax                        {Charge l'offset}
  mov al,byte ptr col              {... et la couleur}
  mov es:[di],al                   {puis dessine le point}
End;
```

```
Procedure Line(x1,y1,x2,y2,col:Word);assembler;
asm
  {registres exploités : bx/cx: parties entière/décimale de
   l'ordonnée y
    si   : partie décimale de la pente}
  mov si,x1                    {initialise x }
  mov x,si
  sub si,x2                    {et mémorise en si la différence
                                des abscisses}

  mov ax,y1                    {initialise y (en bx) }
  mov bx,ax
  sub ax,y2                    {et mémorise en ax la différence
                                des ordonnées}
    mov cx,100                 {étend la différence des
                                ordonnées pour une meilleure
                                précision}
  imul cx
  idiv si                      {divise par la différence
                                des abscisses (pente)}
  mov si,ax                    {mémorise la pente en si }

  xor cx,cx                    {met à zéro la partie décimale
                                des ordonnées}

@lp:
  push x                       {transmet x et la partie  entière }
  push bx                      { de y à PutPixel      }
  push col
  call PutPixel

  add cx,si                    {augmente la partie décimale de y }
  cmp cx,100                   {débordement de la partie
                                décimale ?}
  jb @no_debordmt              {non, on continue }
  sub cx,100                   {sinon diminue la partie décimale }
  inc bx                       {et augmente la partie entière}
```

```
@no_debordmt:
  inc x                          {incrémente x }
  mov ax,x
  cmp ax,x2                      {déjà terminé ?}
  jb @lp                         {non, on recommence }
end;

Begin
  asm  mov ax,0013h; int 10h end;{branche le mode 13h }
  Line(10,10,100,50,1);          {trace un segment }
  ReadLn;
  Textmode(3);
End.
```

Le programme principal initialise par le BIOS le mode graphique 13h et trace un segment dont les coordonnées d'origine sont (10,10), qui s'arrête en (100,50), et qui présente la couleur col 1. La procédure Line profite de ce que la pente prise en considération est inférieure à 1 : elle n'a donc pas de partie entière et sa partie décimale tient dans le registre SI. L'ordonnée y qui est à traiter comme un nombre décimal est décomposée en sa partie entière (mémorisée dans le registre BX) et sa partie décimale (mémorisée en CX).

On commence par charger la valeur initiale x1 de l'abscisse puis on calcule la longueur du segment selon l'axe x, soit (x1-x2). On procède de même pour y. On passe alors au calcul de la pente. Pour déterminer la partie décimale, la différence des y est d'abord multipliée par 100 (facteur décimal pour deux décimales) puis divisée par la différence des x, le résultat étant stocké en SI.

A chaque itération, la boucle dessine un point aux coordonnées courantes et calcule le point suivant. La partie décimale des ordonnées y et celle de la pente sont à chaque fois augmentées. En cas de débordement (>100), la partie entière est incrémentée de 1 et la partie décimale diminuée de 100. Ensuite on augmente encore l'abscisse x de 1, et si le point de destination n'est pas encore atteint on recommence la boucle.

2.3. Fonctions mathématiques personnalisées

Tant qu'on utilise des nombres à virgule flottante, Pascal propose un assez grand choix de fonctions (pas pour un mathématicien, je sais). Les sinus, cosinus, racines carrées et autres fonctions sont là pour rendre plus confortable la vie du programmeur mais elles sont des facteurs de ralentissement des programmes. Les fonctions mathématiques comptent en effet parmi les plus lentes offertes par les langages, à moins que l'on dispose d'un coprocesseur.

Pour la plupart des tâches de la programmation courante, les nombres entiers sont largement suffisants, à supposer que l'intervalle soit approprié (un sinus entier compris entre -1 et +1 est de peu d'intérêt). Si on utilise les fonctions internes de Pascal pour un nombre entier, ce dernier est en fait converti est réel avant d'être soumis aux algorithmes nonchalants du type Real. En fin de compte, les calculs sur les entiers sont encore plus lents que lorsqu'on opère directement sur des équivalents à virgule flottante. Une seule parade : rédiger soi-même ces fonctions.

On peut suivre à cet égard deux méthodes. La première consiste à initialiser au début du programme une table qui contient les résultats de l'opération et à s'y référer par la suite. La seconde méthode consiste à déterminer les fonctions par approximations successives.

Méthode par construction d'une table

La recherche d'une valeur dans une table vous rappelle sans doute des souvenirs de collège. En informatique aussi, on est parfois conduit à utiliser une table dans laquelle sont rassemblées des valeurs à consulter. Avec certaines fonctions, cette manière de procéder est de beaucoup la plus rapide. C'est notamment le cas de toutes les fonctions périodiques du type sinus. Ceci dit, il faut malgré tout s'obliger à calculer une fois la série des valeurs, un travail qui n'est pas amusant.

C'est ce que nous allons appliquer dans notre programme. Au début, il faudra passer par une phase un peu désagréable pour construire la table. Avec les procédures internes de Pascal, cette création coûtera plusieurs millisecondes. Mais par la suite, la consultation sera très rapide.

Comme exemple, nous prendrons la fonction sinus dont nous aurons besoin par la suite dans cet ouvrage (dans la partie consacrée aux graphiques). L'unité TOOLS.PAS contient une procédure générale appelée Sin_Gen :

```
procedure sin_gen(var table:Array of  word;periode,
                  amplitude,offset:word);
{calcule une table des sinus de longueur periode , et la  mémorise
 dans le tableau table. "hauteur" est stockée    dans la variable
 Amplitude et la situation du point 0    en   offset }
Var i:Word;
Begin
  for i:=0 to periode-1 do
   table[i]:=round(sin(i*2*pi/periode)*amplitude)+offset;
End;
```

Cette procédure reçoit d'abord comme paramètres le nom du tableau qui va contenir la table et la longueur de la période de la fonction sinus, celle-ci correspondant au nombre d'entrées de la table puisqu'on ne calcule qu'une seule période. L'amplitude donne la valeur du maximum. Ainsi, pour une amplitude de 30, la table contiendra des valeurs de -30 à +30. Le dernier paramètre est l'offset qui représente le décalage de la fonction sinus dans la direction y. Dans l'exemple précédent, un offset de 10 renvoie une table de valeurs comprises entre -20 et 40. La boucle parcourt les entrées de la table et calcule les valeurs qui conviennent avec la fonction sinus classique de Pascal.

L'application consiste ici à tracer des cercles en mode texte, pour conserver le maximum de simplicité. Plus précisément, le programme SINTEST.PAS dessine deux fois 26 cercles superposés : une première fois avec les fonctions sinus et cosinus ordinaires et une autre fois avec la table. Le coprocesseur est désactivé au début de l'exécution : il est en effet rarement utilisé dans la pratique de l'assembleur et il fausserait ici les résultats de l'expérience. Faites tourner le test et savourez la différence de vitesse !

```
{$n-}                              {désactive le coprocesseur}
Uses Crt, Tools;

Var phi,                           {angle}
    x,y:Word;                      {coordonnées}
    Caractere:Byte;                {caractère utilisé}
    Sinus:Array[1..360] of Word; {mémorise la table des sinus}
```

```
Procedure Sinus_Real;              {dessine 26 fois un cercle}
Begin
  For Caractere:=Ord('A') to Ord('Z')do {26 itérations}
    For phi:=1 to 360 do Begin
      x:=Trunc(Round(Sin(phi/180*pi)*20+40)); {calcule l'abscisse x}
      y:=Trunc(Round(Cos(phi/180*pi)*10+12)); {calcule l'abscisse y}
      mem[$b800:y*160+x*2]:=Caractere;        {affiche le caractère }
    End;
End;
Procedure Sinus_nouvo;             {dessine 26 fois un cercle }
Begin
  For Caractere:=Ord('A') to Ord('Z')do {26 itérations}
    For phi:=1 to 360 do Begin
      x:=Sinus[phi]+40;            {calcule l'abscisse x}
      If phi<=270 Then             {calcule l'ordonnée y}
        y:=Sinus[phi+90] div 2 + 12 Else {cosinus = sinus déphasé }

        y:=Sinus[phi-270] div 2 + 12;
      mem[$b800:y*160+x*2]:=Caractere; {affiche le caractère }
    End;
End;

Begin
  Sin_Gen(Sinus,360,20,0);         {prépare la table des sinus}
  ClrScr;                          {efface l'écran }
  Sinus_real;                      {dessine les cercles}
Repeat until keypressed            {Attente de l'appui d'une touche
                                   avant d'exécuter la procédure
                                   suivante}
  ClrScr;                          {efface l'écran }
  Sinus_nouvo;                     {dessine les cercles}
End.
```

Au départ, le programme principal établit la table des sinus. Ce processus ne s'effectue qu'une seule fois pour chaque table après quoi elles sont, l'une et l'autre, à la libre disposition du programmeur. C'est alors que sont invoquées les deux procédures qui dessinent les cercles. La première, Sinus_Real, calcule les coordonnées du point courant avec les fonctions sinus et cosinus d'origine (l'angle est transmis en radians, d'où le facteur /180*pi). Le rayon a pour valeur 20 selon l'axe x et 10 selon l'axe y. Le cercle est tracé au milieu de

l'écran (+40, +12). L'affichage de chaque caractère se fait par accès direct à la mémoire d'écran.

La deuxième procédure Sinus_nouvo recherche les valeurs de x et de y dans la table préconstituée. Le cosinus est calculé à partir du sinus par un déphasage de 90 degrés, ce qui nécessite toutefois une surveillance des limites de la table. Malgré la présence de l'alternative if, cette procédure se montre beaucoup plus rapide que la précédente : l'exécution du programme est très convaincante à cet égard.

Méthode par approximations successives

Les tables sont parfaites pour les fonctions dont le domaine de définition est limité, comme c'est le cas du sinus. Avec des fonctions du type racine carrée, qui doivent couvrir un nombre infini de nombres, les choses deviennent un peu plus difficiles. Pour avoir un domaine de définition suffisamment vaste, il faudrait réduire fortement la résolution de la table et la précision des calculs.

Mais les formules d'approximation trouvent ici un terrain idéal. Pour la fonction racine, il suffit de consulter un bon ouvrage de mathématiques pour y trouver une formule très efficace :

$$x_{n+1}=1/2(x_n+a/x_n)$$

a est le nombre dont on souhaite extraire la racine (radicande), x_n une valeur initiale quelconque (par exemple 1). x_{n+1} représente alors une approximation meilleure. Ce petit jeu peut durer aussi longtemps que le demande la précision souhaitée. Avec les nombres entiers, on s'arrêtera quand la différence entre la nouvelle approximation et la précédente sera de 0 ou 1. Il faut tolérer explicitement une différence de 1 sinon on risque de poursuivre le processus sans fin. Ce cas se présente lorsque, en raison des problèmes d'arrondi, le résultat oscille entre deux entiers consécutifs.

Par ailleurs, cet algorithme est "autocorrecteur", ce qui est surtout intéressant pour les calculs manuels. Si l'une des valeurs calculées est fausse et si elle est reprise comme valeur de x_n à l'itération suivante, l'approximation subit un ralentissement mais le résultat final à l'issue du processus sera quand même le bon.

L'algorithme évoqué se trouve dans la procédure en assembleur appelée Racine (fichier RACINE.ASM). Cette procédure n'a pas été mise dans une unité car elle sera exploitée plus tard (avec les graphiques 3D) comme procédure Near. Une procédure Far générée par une unité Pascal serait ici trop lente du fait de son mécanisme d'appel. Le source en assembleur contient deux procédures : celle qui s'appelle Racine contient le calcul proprement dit. Elle travaille étroitement avec les registres étant donné que la transmission

des paramètres se fait par les registres DX:AX. Plus tard, la procédure 3D l'appellera directement de cette manière.

Pour pouvoir accéder à la racine carrée dans des cas non critiques par un appel Pascal, le fichier contient également une fonction "cadre" appelée Racfct qui accueille le radicande à la façon d'un paramètre traditionnel et renvoie la racine comme résultat.

```
.286                            ;active au moins les registres
                                ;du 286
e equ db 66h                    ;Operand Size Prefix
                                ;(instructions 32 bits)
w equ word ptr

code segment public
assume cs:code
public racine
public Racfct                    ;valeur du radicande en dx:ax
racine proc pascal               ;résultat en ax (fonction)
e                                ;effectue le calcul sur 32 bits
  xor si,si                      ;efface esi (pour résultat
                                 ;intermédiaire)
  db 66h,0fh,0ach,0d3h,10h       ;shrd ebx,edx,16d - dx en ebx
                                 ;(16 bits sup.)
  mov bx,ax                      ;ax en ebx (en bas) - dx:ax
                                 ;est en ebx
e
  xor dx,dx                      ;efface edx
e
  mov cx,bx                      ;sauve la valeur initiale en ecx
e
  mov ax,bx                      ;charge aussi eax

iterat:
e
  idiv bx                        ;divise par Xn
e
  xor dx,dx                      ;reste sans intérêt
e
  add ax,bx                      ;additionne Xn
```

```
e
  shr ax,1                      ;divise pat 2
e
  sub si,ax                     ;différence avec le résultat
                                ;précédent
e
  cmp si,1                      ;inférieure à 1
  jbe fini                      ;c'est fini
e
  mov si,ax                     ;mémorise le résultat comme
                                ;résultat précédent
e
  mov bx,ax                     ;c'est-à-dire Xn
e
  mov ax,cx                     ;valeur initiale pour division
  jmp iterat                    ;retourne au début de la boucle
fini:
  ret                           ;le résultat se trouve en eax
racine endp

Racfct  proc pascal a:dword     ;traduit la procedure en
                                ;fonction Pascal
  mov ax,word ptr a             ;transfère les paramètres dans
                                ;registres
  mov dx,word ptr a+2
  call racine                   ;extrait la racine
  ret
Racfct endp

code ends
end
```

Ce qui frappe tout de suite lorsqu'on examine la procédure Racine, ce sont les "e" en tête de ligne. Définis au début du programme, ils sont équivalents à db 66h. Il s'agit de l'Operand Size Prefix des processeurs 386 qui étend à 32 bits l'instruction qui suit. Ici, les opérandes à 32 bits accroissent considérablement la vitesse d'exécution car les résultats de type LongInt n'ont pas besoin d'être répartis sur deux registres. Malheureusement, Pascal n'est pas

encore capable de traiter directement ce type d'instructions même si elles sont présentées par un assembleur externe.

Il ne reste donc plus qu'à convertir manuellement chaque instruction en 32 bits.

Avec l'instruction 386 shrd, le contenu du registre DX est mémorisé dans la partie supérieure de EBX, tandis que la partie inférieure est chargée avec le contenu de AX. ECX sert ici à stocker le radicande a qui sera réutilisé par la suite.

La boucle d'itération conduit alors aux étapes décrites dans l'étude de la méthode de résolution. Après division du radicande par la dernière approximation (en EBX), cette dernière est encore additionnée au quotient, ce qui clôture la parenthèse. Après une division par deux, le résultat est comparé au précédent. En cas d'identité (différence maximale de 1), on s'arrête sinon Xn (en BX) est chargé avec le nouveau résultat et l'itération se poursuit. A la fin, la racine se trouve dans le registre AX. C'est là qu'une fonction de type Pascal dépose également son résultat.

A titre d'exemple, nous effectuons un test de rapidité :

```
{$n-}                          {désactive le coprocesseur}
Function Racfct(Radicande:LongInt):Integer;external;
{$l racine}
                               {reporter ici le chemin d'accès
                                au module assembleur racine .obj!}

var i:word;                    {compteur d'itérations}
    n:Integer;                 {résultat du calcul entier }
    r:Real;                    {résultat du calcul real }
Procedure racine_nouvo;        {calcule la racine par
                                 approximations entières }
Begin
  For i:=1 to 10000 do         {10000 fois  }
    n:=Racfct(87654321);       {pour pouvoir comparer la vitesse }
End;
Procedure racine_real;         {calcule la racine par fonction
                                Pascal }
Begin
  For i:=1 to 10000 do         {10000 fois}
    r:=Sqrt(87654321);         {pour pouvoir comparer la vitesse }
```

```
End;

Begin
  writeLn;
  WriteLn('Extraction de racine par fonction Pascal:');
  Racine_Real;
  WriteLn('Résultat: ',r:0:0);
  WriteLn('Extraction de racine carrée par méthode d'approximations
          entières:');
  racine_nouvo;
  WriteLn('Racine: ',n);
End.
```

Ce programme RATEST.PAS calcule de deux manières différentes 10000 fois la racine de 87654321. Même avec un 486 (avec coprocesseur désactivé, directive $n-), il faut plusieurs secondes pour obtenir le résultat. La deuxième partie (selon nos propres méthodes) ne nécessite par contre que quelques fractions de secondes.

2.4. High Speed Tuning - Optimisation des comparaisons

Après les calculs mathématiques, les comparaisons sont la deuxième cause de ralentissement des processeurs. C'est pourquoi il ne faut les mettre en service que parcimonieusement et en les optimisant le mieux possible.

OR au lieu de CMP

Les opérations logiques offrent un moyen simple d'accélérer les exécutions. Au nombre de ces opérations, il faut aussi compter l'instruction TEST qui exploite en fait une opération AD. Pour tester une certaine combinaison de bits, il est avantageux d'y recourir. La comparaison à 0 se prête encore à une meilleure optimisation grâce à la mise en oeuvre de la commande OR. Ainsi *OR AL,AL* met à un l'indicateur de zéro si AL est à 0 (sinon l'indicateur est mis à 0), ce qui permet de prévoir un branchement par jn, jnz.

Comme cette commande agit sur tous les indicateurs du registre des indicateurs, il est possible entre autres de tester en conséquence le signe d'un nombre : en cas de signe négatif, le branchement js renvoie à l'adresse indiquée.

Comparaison avec débordement

Lorsqu'on teste des coordonnées, il est souvent nécessaire d'encadrer une variable qui, par exemple, devra être comprise entre 0 et 320. Utiliser deux comparaisons à cette fin serait du pur gaspillage de calcul. Il est plus élégant d'exploiter une comparaison non signée. Si on teste la variable en tant que nombre non signé, le fait d'être supérieure à 320 inclut pour elle la possibilité d'être négative. Un nombre négatif considéré sans signe a en effet une valeur supérieure à 32767 ce qui remplit la condition précédente.

Comparaison de chaînes de caractères

Pour beaucoup de programmeurs, les comparaisons de caractères recèlent encore bien des mystères. Elles ont cependant une importance déterminante, par exemple dans le cas d'un programme résident qui doit tester s'il n'est pas déjà installé en mémoire centrale. Pour bien appréhender les comparaisons, il faut avant tout se pencher sur le fonctionnement de l'instruction JCXZ. Cette instruction a pour effet de déclencher un branchement à l'adresse indiquée lorsque le contenu du registre CX devient nul. En association avec l'instruction de répétition REPE CMPSB, les comparaisons de chaînes de caractères sont en fin de compte faciles à programmer.

Il faut d'abord initialiser les pointeurs ES:DI et DS:SI en les faisant pointer sur les deux chaînes. Dans CX, on stocke la longueur des chaînes. On exécute ensuite l'instruction REPE CMPSB qui tourne aussi longtemps que les chaînes ne présentent pas de différence (REPE s'interrompt lorsque l'indicateur de zéro est à 0) jusqu'à ce que les chaînes aient été entièrement parcourues (CX=0 met fin à la boucle). L'instruction JCXZ exploite judicieusement ce comportement. Si les chaînes ne sont pas identiques, CX n'est pas encore nul, aucun branchement n'a lieu. Ce n'est qu'avec des chaînes strictement égales que CX aboutit à la valeur 0, ce qui occasionne un branchement.

2.5. Les variables en assembleur

Il est conseillé de maintenir autant de valeurs que possible dans les registres, car le processeur sait y accéder très rapidement. Il ne faut pas hésiter à détourner de leur vocation de base les registres spécialisés comme SI, DI, BP pour les exploiter comme des registres de stockage ordinaires. Mais même en faisant preuve d'habileté, on manque toujours de registres, leur nombre n'étant pas assez élevé (en application de la loi de Murphy : tout ce qui peut mal tourner ne manquera pas de le faire). Il faut alors recourir à la mémoire centrale, ce qui nous ramène à l'usage des variables ordinaires.

Accès aux variables Pascal

Accéder aux variables Pascal à partir de l'assembleur n'est plus un problème de nos jours même si quelques programmeurs se compliquent inutilement la vie avec des constructions du genre

```
mov AX,[offset variable].
```

Il est plus simple d'écrire directement :

```
mov AX,variable.
```

Si on désire effectuer en même temps un changement de type (par exemple en mettant un offset de pointeur dans un registre 16 bits), il faut ajouter un Word ptr ou Byte ptr, par exemple : mov AX, word ptr Pointeur + 2.

Accès aux tableaux et enregistrements

Les tableaux sont faciles à gérer directement en assembleur, il suffit de s'occuper soi-même de leur indexation. Il faut évidemment tenir compte de la taille de chaque élément. Les éléments de type Word demandent 2 octets, tandis que les Doubleword en réclament 4. D'autres offsets sont imaginables, par exemple lorsqu'on exploite un tableau d'enregistrements (Array of Record). Le 386 a été préparé à ces fins, il peut adresser des variables sous la forme mov AX,[2*ecx]. Mais en Pascal (limité au code 286 !), cette prouesse est difficile à réaliser, il faudrait présenter les instructions correspondantes sous forme de simples suites d'octets. Il vaut mieux effectuer la multiplication à l'extérieur par une commande ***shl SI,1***.

La plupart des assembleurs permettent d'indiquer l'offset d'un tableau sous la forme usuelle en le mettant devant l'indice.

```
mov AX, word ptr Arr[SI]
```

est interprétée par l'assembleur comme

```
mov Ax, word ptr [SI+offset Arr].
```

Les enregistrements (Records) ne nécessitent plus des indications d'offsets longues et fastidieuses. Dans Pascal-ASM, il est maintenant possible d'accéder directement aux enregistrements tout comme à partir de Pascal : mov AX, word ptr rec.a. TASM et MASM exploitent une variable semblable aux enregistrements et appelée structure.

Elle permet de reproduire directement un enregistrement Pascal pour y accéder par la suite :

```
data segment
rec_typ struc
  a dw ?
  b db ?
rec_typ ends

extrn rec:rec_typ
```

Variables de segment de code

Il n'y a jamais suffisamment de registres dans un processeur pour satisfaire les besoins d'un programmeur. Il faut donc exploiter chacun d'eux, y compris par exemple le registre BP qui après tout est un registre comme les autres. Mais il se pose quand même un problème : BP est utilisé pour adresser les variables locales de la procédure. Faut-il alors y renoncer ou les mettre ailleurs ? Les variables globales peuvent elles aussi s'avérer inaccessibles, surtout dans les procédures graphiques, lorsque par exemple le registre DS ne pointe plus sur le segment de données mais sur la description d'un sprite. Pour sortir de cette impasse, le recours aux variables de segment de code s'impose.

Ces dernières se trouvent avec le code de programme dans le segment de code courant et sont adressées au niveau machine par le segment override prefix. Au niveau de la programmation, l'assembleur prend en charge une partie du travail de sorte qu'on ne remarque presque rien de spécial. Mais tout accès de l'extérieur, par exemple à partir d'autres procédures ou modules, est impossible : il est vrai qu'il en est de même pour les variables locales. Sous MASM ou TASM, les variables peuvent être stockées dans le segment de code au lieu du segment de données. L'assembleur s'occupe alors automatiquement de la bonne gestion des adresses.

Pascal, par contre, complique un peu la vie du programmeur car il ne permet pas de mettre des données dans le segment de code à l'extérieur des procédures et fonctions. Il faut alors faire appel à une astuce. Au début de la procédure, on met une instruction jmp qui renvoie au code de la procédure, et les variables sont disposées juste après dans le cadre d'étiquettes, comme le montre l'exemple suivant :

```
Procedure Test;assembler;
asm
  jmp @vasy
 @Var1: dw 0
```

```
@Var2: db 0
 @vasy:
  {Reste de la procédure}
End;
```

Tableaux circulaires

Les tableaux ne sont pas toujours traités du début à la fin, il leur arrive d'être parcourus d'une sorte de mouvement circulaire, du début à la fin puis en recommençant à partir du début. Tel sera par exemple le cas d'une table des sinus, pour qu'on puisse y rechercher sans problème un angle, disons de 700 degrés. La manière la plus simple mais aussi la plus inefficace pour implémenter cette caractéristique consiste à ramener l'indice à une valeur acceptable lorsqu'il dépasse la valeur limite de la table. Dans le cas précédent, on retrancherait 360 degrés ce qui ramène l'indice à la valeur acceptable 340.

Mais il est plus rapide de construire le tableau de façon que le nombre de ses éléments soit une puissance de deux : 32, 64, 128. Dans ce cas, l'indice peut être projeté dans le domaine autorisé par simple masquage binaire à l'aide de l'opérateur AND. Si le tableau possède 64 éléments (0-63), chaque indice est combiné à 63 par l'opérateur AND, ce qui supprime les deux bits supérieurs et ne laisse intervenir que les six bits inférieurs.

Pour créer un tableau qui convienne à ce type d'exploitation, il faut cependant introduire un nombre d'élément approprié. Dans notre exemple, la table des sinus serait à construire avec une période de 64. Rien à voir avec le nombre de degrés usuels mais après tout qu'importe : pourquoi faudrait-il obligatoirement découper un cercle en 360 sections plutôt qu'en 64 ?

Masquage binaire tournant

La programmation système fait souvent appel au masquage binaire. Ainsi, les différents bits d'un nombre écrit dans un registre VGA auront une signification précise et individuelle (par exemple l'activation d'un plan binaire d'affichage). Pour rester dans le cadre de cet exemple, il sera nécessaire d'activer successivement les plans 0,1,2,3 puis de sélectionner à nouveau le plan 0, ce qui consiste donc à mettre à un les bits 0,1,2,3 et à nouveau 0.

Le problème qui se pose est le suivant : comment revenir au bit 0 après le bit 3 ? La méthode brute consisterait à exploiter simplement une instruction CMP, mais il est possible de faire plus vite. Au niveau 8 bits, on introduirait une rotation d'octet qui envoie le bit 7 automatiquement dans le bit 0. Ici, nous sommes au niveau 4 bits, nous pourrions appliquer le même principe mais la rotation d'un demi-octet n'est pas directement possible. Nous utiliserons donc une astuce qui consiste à initialiser l'octet non pas avec la valeur 01h mais avec 11h. Dans ce cas, les moitiés supérieure et inférieure de l'octet

(nibbles) contiennent la même valeur. Si on effectue des rotations en ne masquant à chaque fois que le nibble inférieur, les bits 0-3 subiront l'effet recherché. A la fin du cycle, l'octet contient la valeur 88h. Si on poursuit la rotation, on retombe sur 11h (bit 7 en bit 0 et bit 3 en bit 4).

Masquage d'un certain nombre de bits.

Certaines situations exigent de sélectionner un certain nombre de bits dans un mot (word) ou un octet (byte), par exemple dans le chargeur GIF du chapitre 4.3. Il s'agit de sortir d'un octet un certain nombre de bits non encore traités. Au lieu de passer par une boucle, on peut ici encore exploiter une voie arithmétique. On charge un registre (par exemple A) avec la valeur 1 et on décale ce registre vers la gauche du nombre de bits à masquer.

Du résultat, on soustrait 1 ce qui donne un masque prêt à l'emploi, qui ne contient que des 1 aux endroits qu'il faut, et 0 ailleurs. La formule pourra s'écrire :

```
Masquage := (1 shl n) - 1
```

Pour sélectionner 6 bits, on crée le masque (1 shl 6) - 1 = 63, ce qui met à 1 les bits 0-5 et à 0 les bits 6 et 7. L'octet à masquer est à combiner à ce nombre par l'opérateur AND, ce qui conduit à l'extraction les bits souhaités.

Il est intéressant de signaler à ce propos une curieuse propriété des instructions SHR et SHL sur les processeurs 386 et au-delà. Seuls les 5 bits inférieurs sont pris en compte dans le nombre de bits demandés au décalage. Le décalage maximal ne peut porter que sur 31 bits, indépendamment de la dimension du registre utilisé. Si on demande par exemple à décaler AX de 34 bits, ce qui revient normalement à effacer le registre (qui n'a que 16 bits), on constate que l'instruction SHL AX,34d provoque un décalage de 2 bits seulement.

Cette anomalie n'est pas très importante mais elle m'a coûté une demi-heure de recherche d'erreur car je croyais fermement qu'un décalage excessif annulait le contenu du registre.

2.6. Le secret des cracks : les interruptions

Les interruptions font l'objet d'une programmation aussi intéressante que variée mais leur domaine apparaît souvent mystérieux à ceux qui souhaitent l'aborder. Cette réputation est sûrement due pour une large part aux plantages généralement occasionnés par leur mise au point. Mais avec de bonnes connaissances de base et quelques bons exemples d'initiation, on peut apprendre rapidement à maîtriser la matière. Même si votre ordinateur se plante

de temps à autre, sachez que les programmeurs les plus expérimentés subissent régulièrement le même avatar.

Les interruptions ne sont rien d'autre que des demandes urgentes adressées au processeur. Selon leur cheminement, on distingue les interruptions logicielles et les interruptions matérielles. Les premières sont déclenchées explicitement par l'instruction Int et sont comparables à des sous-programmes ordinaires. La seconde catégorie est nettement plus intéressante : les interruptions matérielles en provenance de périphériques externes parviennent au processeur par le moyen de deux contrôleurs d'interruption. Par exemple, chaque frappe de touche déclenche une interruption qui conduit le processeur à déclencher un programme spécialisé (gestionnaire d'interruption) qui réceptionne le caractère tapé.

Pliage de vecteurs

C'est ici que commence le travail du programmeur. Il peut accrocher son propre programme à chaque interruption pour déclencher une action particulière, comme par exemple dans le cas de l'interruption du clavier l'émission d'un clic par le haut-parleur à chaque frappe de touche. L'adresse de la procédure à exécuter par chaque interruption est stockée dans un vecteur d'interruption situé tout au début de la mémoire vive. Le vecteur d'interruption peut être interrogé et modifié par les fonctions 35h et 25h de DOS. On transmet à ces fonctions le numéro de l'interruption par le registre AL. La fonction 35h renvoie le vecteur en ES:BX, tandis que la fonction 25h en prend connaissance dans les registres DS:DX. Pour lire le vecteur de l'interruption 9, on écrit :

```
mov ax,3509h    ; numéro de la fonction et de l'interruption
int 21h         ; exécute la fonction Dos
```

Le vecteur se trouve à présent en es :bx. Il est modifié par les instructions suivantes :

```
lds dx,Vecteur ; lit le vecteur (comme pointeur)
mov ax,2509h   ; numéro de fonction et d'interruption
int 21h        ; exécute la fonction Dos
```

Avant de détourner un vecteur d'interruption vers votre propre procédure, il est recommandé de lire l'ancien vecteur et de le sauvegarder. Vous en aurez besoin à la fin du programme. Par exemple, si c'est un programme résident, il faudra lors de la désinstallation rétablir la situation d'origine. Dans bien des cas, il faut appeler l'ancien gestionnaire à partir du nouveau.

Appel de l'ancien gestionnaire et terminaison

Dans l'exemple précédent du clavier, il ne suffit pas d'émettre un clic à chaque frappe de touche, le caractère tapé doit également apparaître sur l'écran. Il est donc nécessaire d'invoquer l'ancien gestionnaire avant ou après l'émission du clic, à moins que vous n'ayez décidé d'écrire votre propre driver de clavier. Comme l'ancien vecteur a été sauvegardé, on peut pratiquement y faire appel directement par un far call. La voie n'est pas tout à fait directe car il faut simuler une interruption. La seule particularité d'un appel d'interruption par rapport à un far call ordinaire tient à la manière particulière de sauvegarder les indicateurs du processeur, ce qui se fait facilement avec une instruction pushf. Un appel complet se présente alors de la façon suivante :

```
pushf
call dword ptr [ancien_vecteur]
```

le vecteur d'origine ayant été au préalable sauvegardé dans le pointeur ancien_vecteur.

Une interruption se termine toujours par l'instruction IRET mais il faut auparavant rétablir les registres du processeur dans leur état initial. Il se pourrait en effet que l'interruption ait été déclenchée au milieu d'une routine qui repose sur l'exploitation de certains de ces registres.

Masquage des interruptions

Pour masquer toutes les interruptions en même temps, le processeur dispose d'une instruction CLI qui inhibe l'intégralité des entrées d'interruption. L'instruction symétrique STI les libère à nouveau. Mais il est parfois souhaitable de désactiver certaines interruptions tout en autorisant certaines autres. Pour cela, il est nécessaire de reprogrammer le contrôleur d'interruptions. Par rapport aux vecteurs, ce circuit adopte une autre convention de numérotation. Les interruptions matérielles portent les numéros 0-7 (contrôleur 1) et 8-15 (contrôleur 2). On parle alors des requêtes d'interruption (Interrupt Request ou IRQ) 0-15 : les interruptions en question se référant aux numéros des vecteurs.

Les IRQ 0-7 sont prises en charge par le contrôleur 1 qui les renvoie comme interruptions 8-0fh au processeur tandis que les IRQ 8-15 passent par le contrôleur 2 et sont transmises comme interruptions 70-77h. Les deux contrôleurs sont reliés en cascade par l'IRQ2 : lorsque le contrôleur 1 reçoit une demande d'interruption, il la transmet par l'entrée 2 au contrôleur 2.

La disposition des IRQ sur les contrôleurs est la suivante :

Contrôleur 1 :	
IRQ	**Propriétaire**
0	Timer
1	Clavier
2	relié en cascade au contrôleur 2
3	Com 2, Com 4
4	Com 1, Com 3
5	LPT 2
6	Disquette
7	LPT 1

Contrôleur 2 :	
IRQ	**Propriétaire**
8	Horloge temps réel
9	VGA (souvent inactif), réseau
a	-
b	-
c	-
d	Coprocesseur
e	Disque dur
f	

Les interruptions sont hiérarchisées, autrement dit les IRQ sont d'autant plus prioritaires que leur numéro est plus petit. Cette hiérarchie peut être modifiée par programme mais il vaut mieux ne pas y toucher car le Bios et DOS reposent sur cette structure.

Les contrôleurs disposent de registres de masquage (IMR = Interrupt Mask Register) qui permettent d'inhiber telle ou telle interruption. L'IMR du premier contrôleur se trouve à l'adresse de port 21h, l'IMR du second contrôleur à l'adresse a1h. Un bit mis à 1 signifie que l'interruption est masquée.

Ainsi pour désactiver l'horloge temps réel, on écrira la séquence suivante :

```
in  al,0a1h     ; charge le contenu du registre IMR 2
or  al,01h      ; met à 1 le bit 0
out 0a1h,al     ; écrit le résultat dans le registre
```

Les deux contrôleurs disposent d'une autre adresse de port (20h resp. a0h) qui leur permet de recevoir des commandes. La plus importante est l'instruction EoI (End of Interrupt) codée 20h qui signale la fin d'un gestionnaire d'interruption et libère à nouveau le contrôleur pour l'interruption suivante. Si à l'intérieur de son gestionnaire personnalisé on appelle le gestionnaire d'origine, c'est lui qui s'occupe de ces détails. Mais si on décide d'écrire de fond en comble un nouveau gestionnaire, il faudra veiller à la fin du programme à envoyer une commande 20h sur le port 20 h (resp. a0h) :

```
move al,20h
out 20h,al
```

Deuxième entrée dans DOS : la réentrance

Selon le cas, un gestionnaire d'interruption pourra prendre quelques cycles d'horloge ou durer plusieurs secondes (cas de la copie d'écran Impr écran interruption 5). Lorsque le temps d'exécution est long, il faut si possible empêcher que le gestionnaire soit soumis à une nouvelle interruption pendant son déroulement. Faute de cette précaution, on obtient au mieux un résultat inattendu (deux copies d'écran imbriquées avec l'interruption 5), au pire un plantage du système. En effet, le nouveau gestionnaire qui se déclenche accède à des variables dont le premier a encore besoin.

La manière la plus simple d'éviter cet inconvénient est d'instituer un indicateur qui signale si le gestionnaire a été lancé et qui peut être consulté en cas de nouveau déclenchement pour éviter ce qu'on appelle la réentrance.

La situation est un peu plus compliquée pour les projets résidents qui sont plus gros et qui activent un programme complet avec menus à la simple frappe d'une touche. Le problème tient à ce que DOS ne permet pas d'activer quasi simultanément plusieurs fonctions DOS. Si une interruption perturbe une fonction DOS en invoquant d'autres fonctions DOS durant son exécution (par exemple pour effectuer un affichage à l'écran), cet appel détruit la pile de DOS de sorte qu'à la sortie du gestionnaire, au retour dans la fonction DOS interrompue, l'ordinateur se plante.

Eviter cet inconvénient n'est pas trivial : il faut commencer par consulter l'indicateur appelé InDos. Cette consultation se fait avant installation du gestionnaire à l'aide de la fonction non documentée de DOS numéro 34h, qui renvoie un pointeur far dans les registres ES:BX. Ce n'est que si cet indicateur

est à 0 au moment de l'activation qu'il est possible d'invoquer sans inconvénient un gestionnaire qui fait appel à des fonctions de DOS.

Il faut également installer un gestionnaire pour l'interruption 28h qui sera appelé tant que l'interpréteur COMMAND.COM attend une entrée de l'utilisateur. Dans ce cas, l'indicateur InDos pourra être à 1 car COMMAND.COM compte lui-même comme fonction de DOS.

Ces mesures de sécurité quelque peu compliquées peuvent être laissées de côté si le gestionnaire n'appelle pas de fonction de DOS, ce qui est quand même le cas d'un certain nombre de programmes résidents.

Interception de Ctrl C et Reset

Un programme véritablement professionnel ne devrait pas permettre à l'utilisateur de sortir par une issue dérobée. Car dans ce cas, s'il reste des fichiers ouverts, la démarche se solderait inévitablement par des pertes de données.

Il est par ailleurs tout à fait désagréable qu'une démonstration puisse être interrompue par n'importe quel assistant qui s'amuserait à taper **CTRL-PAUSE** ou **CTRL-ALT-SUPPR**. Comment peut-on intercepter infailliblement ces fonctions du BIOS ?

Pour **CTRL-C** resp. **CTRL-PAUSE**, DOS propose un début de solution : lorsqu'on appuie sur l'une des combinaisons de touches précitées, l'interruption 23h se déclenche et conduit à l'abandon du programme en cours. Il est donc possible de détourner ce vecteur vers une routine qui se charge de renvoyer le contrôle à l'appelant. L'inconvénient est que ça ne marche pas toujours, surtout avec **CTRL-PAUSE** et que de temps en temps l'ordinateur se plante malgré tout.

On peut faire appel à une méthode plus sûre qui intercepte également la réinitialisation par **CTRL-ALT-SUPPR**. On écrit un gestionnaire d'interruption de clavier qui teste avant intervention de l'original si l'une des combinaisons de touches critiques n'a pas été déclenchée. Si tel est le cas, on met fin à l'exécution du gestionnaire de sorte que rien ne se passe. Par contre, les frappes de touches licites sont renvoyées à l'ancien gestionnaire qui les transmet au programme principal. A titre de démonstration de cette méthode, vous trouverez sur le CD le programme NO_RST.ASM, à assembler avec TASM ou MASM.

```
data segment public
 start_message: db 'Plus de Reset possible',0dh,0ah,'$'
 buffer:     db 40d                ;taille du buffer d'entrée
             db 40 dup (0)         ;buffer
 old_int9   dd 0                   ;ancien gestionnaire d'interruption
```

```
data ends

code segment public
assume cs:code,ds:data
handler9 proc near              ;nouveau gestionnaire de
                                ;l'interruption 9
  push ax                       ;sauvegarde les registres exploités
  push bx
  push ds
  push es
  mov ax,data                   ;charge ds
  mov ds,ax

  in al,60h                     ;lit les caractères tapés en al

  xor bx,bx                     ;es pointe sur le segment 0
  mov es,bx
  mov bl,byte ptr es:[417h]     ;charge en bl l'état du clavier

  cmp al,83d                    ;scan code de la touche Suppr ?
  jne pasde_reset               ;non,pas de reset

  and bl,0ch                    ;masque Ctrl et Alt
  cmp bl,0ch                    ;enfoncées simultanément ?
  jne pasde_reset               ;non pas de reset

bloque:                         ;Reset ou break, il faut
                                 intervenir
  mov al,20h                    ;envoie EoI au contrôleur
                                ;d'interruption
  out 20h,al
  jmp fini                      ;et quitte l'interruption

pasde_reset:                    ;pas de Reset, teste break
  cmp al,224d                   ;touche étendue  ?
  je evtl_break                 ;oui -> peut-être break
  cmp al,46d                    ;touche 'C' ?
  jne licite                    ;non -> touche licite
```

```
evtl_break:
  test bl,4                     ;teste l'état du clavier
  jne bloque                    ;touche Ctrl, alors bloquer


licite:                         ;touche licite -> appeler
                                ;l'ancien gestionnaire du clavier
  pushf
  call dword ptr [old_int9]     ;gestionnaire d'origine
fini:
  pop es
  pop ds                        ;reprend les registres
  pop bx
  pop ax
  iret
handler9 endp
start proc near
  mov ax,data                   ;initialise ds
  mov ds,ax
  mov dx,offset start_message   ;met en dx l'offset du message
  mov ah,09h ;affiche le message
  int 21h

  mov ax,3509h                  ;lis l'ancien vecteur d'interruption
  int 21h
  mov word ptr old_int9,bx      ;et le mémorise
  mov word ptr old_int9 + 2, es

  push ds                       ;sauvegarde ds
  mov ax,cs                     ;et y charge cs
  mov ds,ax
  mov dx,offset handler9        ;charge aussi l'offset du
                                ;gestionnaire
  mov ax,2509h                  ;fixe le vecteur
  int 21h
  pop ds
;-----------------------------------------------------------------
; Ici à la place de l'appel DOS vous pouvez mettre un
```

```
; call vers votre programme principal

  mov ah,0ah                    ;lis une chaîne de caractères
  lea dx,buffer                 ;à titre d'exemple
  int 21h
;-----------------------------------------------------------------

  push ds
  lds dx,old_int9               ;rétablit l'ancien vecteur
  mov ax,2509h
  int 21h
  pop ds

  mov ax,4c00h                  ;met fin au programme  int 21h
start endp

code ends
end start
```

Le programme principal (start) commence par afficher un message explicatif, lit l'ancien vecteur de l'interruption 9 du clavier et dirige le nouveau vecteur sur la procédure Handler9. A la place de la partie de programme à protéger (donc en fait de la routine de démonstration), on déclenche ici un affichage sous DOS qui traite une chaîne de 40 caractères. A la fin du programme, l'ancien gestionnaire est restauré.

Le gestionnaire nouveau est appelé à chaque frappe de touche. Il sauvegarde d'abord tous les registres exploités pour que le programme interrompu ne s'aperçoive pas de ses activités. Puis le port de données du contrôleur de clavier est interrogé pour fournir le scan code de la touche enfoncée (dans le registre AL). La variable d'état du clavier est consultée pour connaître la position des touches **CTRL** et **ALT** (en BBL). Cette variable se trouve à l'adresse 0:417h avec la structure suivante :

Bit	Signification
7	Ins
6	Caps Lock (Verrouillage des majuscules)
5	Num Lock
4	Scroll Lock (Verrouillage du défilement)
3	Alt

Bit	Signification
2	Ctrl
1	Shift (Majuscule) gauche
0	Shift (Majuscule) droit

On teste d'abord si la touche **SUPPR** a été enfoncée (ce qui pourrait conduire à un Reset) et, si tel est le cas, si les touches **CTRL** et **ALT** (bits 2 et 3 de BL) sont également en position d'enfoncement. Si tout cela est vérifié, l'exécution se poursuit à l'étiquette "bloque" qui conduit à l'envoi d'un signal EoI au contrôleur d'interruption 1 et à la clôture du gestionnaire. Si aucun Reset n'a été détecté, le programme teste les combinaisons **CTRL-PAUSE** et **CTRL-C** à partir de l'étiquette "pasde_reset". Si ni la touche **C** (scan code 46) ni aucune touche étendue (scan code 224) n'a été actionnée, on peut supposer qu'on se trouve en présence d'une frappe licite, ce qui renvoie le contrôle à l'étiquette "licite" où l'ancien gestionnaire est invoqué avant clôture de l'interruption. Si la touche **PAUSE** ou la touche **C** a été enfoncée, il faut regarder si la touche **CTRL** est également activée. Si tel est le cas, on bloque toute transmission vers le programme, sinon il s'agit d'une touche licite.

Si vous voulez utiliser ce traitement pour vos propres besoins, il suffit de remplacer l'affichage du message par un appel de votre programme principal.

2.7. Conseils pour l'utilisation des boucles

Même un domaine aussi simple que l'usage des boucles peut donner lieu à bien des optimisations en langage machine. Commençons par examiner les boucles simples du type "loop label". Cette instruction ne connaît plus guère la faveur des développeurs. Sur les 386, une construction du genre

```
dec CX
jne label
```

est 10 % plus rapide, sur les 486 l'écart atteint 40%. Et ceci malgré la présence dans ce dernier cas d'un octet supplémentaire à tirer d'une mémoire centrale forcément longue à réagir. L'instruction loop ne se justifie plus que dans le cas où la décrémentation de CX ne doit pas influencer les indicateurs, notamment dans certaines comparaisons compliquées de chaînes qui ne sont pas solubles par REP CMPSB.

L'indicateur de direction dans les instructions sur chaînes de caractères

La mauvaise gestion de l'indicateur de direction est une source fréquente d'erreurs dans les instructions sur chaînes (lodsb, cmpsb) qui sont en fait des boucles. Normalement, cet indicateur est toujours à 0. Mais si à un endroit quelconque du programme vous le positionnez à 1 pour traiter des chaînes d'arrière en avant, il ne faut jamais oublier de le remettre à 0.

Imbrication

Lorsqu'on met en service des boucles imbriquées, il faut toujours trouver un compromis entre vitesse et registres. Il faut employer le plus possible de registres comme compteurs d'itérations après quoi il faudra introduire des variables mémoires. Les limites et le sens de progression de la boucle doivent faire l'objet d'une réflexion soigneuse. En principe, il vaut mieux décompter par décrémentation pour qu'un test de l'indicateur de zéro suffise à gérer la fin des itérations. Dans certains cas, lorsque le compteur devient nul, il faudra encore effectuer une dernière itération. On fera alors appel à l'indicateur de signe qui détecte les nombres négatifs. C'est en atteignant la valeur -1 que le compteur provoquera alors la clôture de la boucle.

Accès 16/32 bits

Chaque fois que c'est possible, il est recommandé d'introduire des instructions de 16 ou 32 bits qui réduisent le nombre des accès à la mémoire. Sur le plan de la vitesse d'exécution, notez bien qu'à partir du 386 un accès du processeur à un octet est converti en un accès à un double mot (doubleword), de sorte qu'il est plus rapide de déplacer un double mot qu'un octet isolé.

Dans certains cas cependant, on sera encore obligé d'avoir recours à des instructions 8 bits. Il peut s'agir par exemple de recopier un nombre d'octets qui n'est pas un multiple de 4 (ainsi 9 octets = 2 Doubleword + 1 octet), ou encore de satisfaire des contraintes matérielles. Ainsi les cartes VGA n'aiment pas que l'on effectue dans les modes à base de plans (cf le mode X expliqué plus loin) des accès supérieurs à 8 bits, étant donné que les registres de plans internes (bascules ou latches) n'ont qu'une capacité de 8 bits.

2.8. Quelques instructions pratiques sur le 386

En plus de ses caractéristiques proprement innovatrices (mode virtuel, pagination, registres de 32 bits), le processeur a également vu s'étendre son jeu d'instructions. Quelques unes de ces instructions sont très précieuses, notamment lorsqu'elles regroupent plusieurs instructions du 286, ce qui conduit à une excellente efficacité dans telle ou telle situation critique. N'hésitez pas à exploiter ces instructions, même en mode réel, ne les laissez pas prendre la poussière dans un recoin de votre processeur.

Les instructions MOVSX et MOVZX

Considérons d'abord les deux instructions MOVSX et MOVZX. Toutes les deux sont capables de déplacer le contenu d'un registre de 8 bits dans un registre de mot, ou le contenu d'un registre de 16 bits dans un registre de 32 bits, ce qui normalement nécessite deux instructions. Les lettres "S" et "Z" qui figurent dans les mnémoniques signifient "signed" et "zero" et font allusion à la partie supérieure du registre de destination. Avec MOVSX, tous les bits de cette partie supérieure sont remplis de 0 ou de 1 en fonction du signe du registre source, de façon que le signe de départ reste valable. MOVZX, par contre, efface scrupuleusement la partie supérieure du registre de destination.

Supposons par exemple que BL ait pour contenu -1 (ffh) :

```
movzx ax,bl             ; ax contient 255 (00ffh)
movsx ax,bl             ; ax contient à présent -1 (ffffh)
```

Instructions SET

Sur les 386, on peut encore optimiser les comparaisons dans certains cas. Ce processeur dispose en effet de 30 instructions SETxx qui combinent à la fois un CMP, un branchement conditionnel, et un MOVE. A chaque type de branchement conditionnel correspond en effet un type d'instruction SET (SETz, SETnz, SETs, ...). Si le test est vérifié, l'octet transmis comme opérande est mis à 1, sinon il devient nul :

```
dec cx                  ; décrémente le compteur de boucle
sete al                 ; exploite al comme indicateur
```

Dans cet exemple qui pourrait être extrait d'une boucle, al est normalement mis à 0 (car CX>0). Ce n'est que si CX devient nul que Al devient égal à 1. On peut ainsi initialiser une variable booléenne de Pascal en fonction d'une condition rédigée en assembleur (SETxx byte ptr variable).

Accélération des multiplications et divisions avec SHRD et SHRL

Le 386 permet de faire directement des calculs dans des registres de 32 bits (notamment des multiplications et des divisions). Cette méthode est évidemment beaucoup plus rapide que la voie traditionnelle par les registres DX:AX.

Mais comment ranger dans un registre étendu (par exemple EAX) des nombres transmis dans le format DX:AX ?

S'il est vrai que par malheur les parties supérieures de ces registres ne peuvent pas être adressées directement, il ne faut pas oublier que le 386 possède des instructions capables de faire davantage qu'un chargement : il s'agit des instructions de décalage étendu SHLD et SHRL.

Ces dernières reçoivent comme arguments, en plus du nombre de bits à décaler, deux opérandes. Le premier (opérande de destination) subit le décalage demandé. Mais au lieu de mettre à 0 les bits libérés à droite (décalage vers la gauche) ou à gauche (décalage vers la droite), il les tire par rotation du deuxième opérande (qualifié de source). Mais l'opérande source lui-même n'est pas altéré.

Si AX contient, par exemple, le nombre 3 (0000 0000 0000 0011) et BX le nombre 23 (0000 0000 0001 0111), l'instruction

```
SHRD BX,AX,3
```

a pour effet de décaler de trois bits vers la droite le contenu de BX (=2). Mais en même temps, la partie supérieure reçoit 3 bits en provenance de AX, de sorte que le résultat dans l'opérande de destination (BX) est finalement

```
0110 0000 0000 0010 = 6002h = 24578
```

Ces instructions sont surtout utilisées pour charger des registres de 32 bits (ici EBX) avec le contenu de deux registres de 16 bits (DX:AX dans notre exemple). Plus précisément, la méthode employée est la suivante : la partie supérieure est d'abord chargée par un SHRD :

```
SHRD EBX,EDX,16d
```

Cet ordre fait rentrer "par le haut" DX dans le registre EBX. La partie inférieure est ensuite directement chargée avec la valeur souhaitée, ce qui préserve la partie supérieure : MOV BX,AX. On trouve une application de cette méthode dans la procédure Racine du chapitre 2.3.

Multiplication étendue par IMUL.

Les nouvelles instructions de multiplication constituent également une mine d'améliorations à ne pas négliger. Avec le 386, on peut pratiquement multiplier n'importe quel registre par n'importe quel nombre. IMUL DX,3 (triplement de DX) est tout aussi valable que IMUL AX,DX,3 (le résultat du triplement de DX est mémorisé en AX). Par rapport aux anciennes formes de l'instruction IMUL, ces techniques nouvelles permettent d'économiser beaucoup de temps.

Utilisation d'instructions 386 dans les programmes Pascal

Les instructions 386 ont en commun de ne pouvoir être prises en compte dans Borland Pascal, ni par l'assembleur interne ni par le moyen de modules assemblés à l'extérieur (si un code objet est lié par la directive $L, le processeur indiqué dans le code doit correspondre à celui qui est paramétré dans Pascal).

On en est alors réduit à tromper le compilateur par de l'assembleur en ligne trafiquée. On procède de la façon suivante. L'instruction souhaitée est tapée dans sa forme définitive sous Turbo Debugger. Ce dernier affiche le code hexadécimal de l'instruction dont on prend soigneusement note. Ledit code est alors inséré dans le programme à la suite d'une directive db :

```
db 66h,0fh,0ach,0d3h,10h  ; shrd ebx,edx,16d
```

Par la suite, il sera difficile d'effectuer des modifications. Dans notre exemple, on peut certes remplacer l'opérande 16d (10h) par 8 puisqu'il s'agit visiblement du dernier octet de la séquence. Mais d'une façon générale, il faudra réassembler manuellement l'instruction avec Turbo Debugger.

A long terme, il faut espérer que les services de développement de Borland reconnaîtront que le processeur 386 est le nouveau standard (ce qui n'est même plus vrai aujourd'hui) et qu'il convient de pouvoir en exploiter les avantages sans tarder.

3. La pratique concrète de l'assembleur

Ce chapitre traite d'applications pratiques de la programmation en assembleur. Les exemples présentés donnent une idée de ce qu'il est possible de faire en assembleur avec un PC. Vous y trouverez la programmation des ports parallèle et série, du haut-parleur (même pour des échantillonnages) et des programmes résidents.

3.1. Le port parallèle

Avec l'écran et le clavier, le port parallèle est pour la majorité des utilisateurs le point de contact le plus important avec le monde extérieur, en l'occurrence l'imprimante. Le port parallèle est remarquablement desservi par le Bios, de sorte qu'il n'est pas forcément indispensable d'écrire des gestionnaires nouveaux. Il est capable de bien autre chose que la commande des imprimantes : il peut ainsi être exploité pour la transmission des données vers un autre ordinateur, et même comme une carte sonore si on ajoute quelques composants électroniques passifs. Au point de vue matériel, le port parallèle est un connecteur Sub-D muni des broches suivantes :

1	-Strobe	14	-Auto LF
2-9	D0 - D7	15	-Error
10	-ACK	16	-Reset
11	-Busy	17	SLCT IN
12	PE	18-25	GND
13	SLCT		

Programmation directe du port parallèle

Tout port parallèle dispose de trois registres situés à trois adresses successives. Les adresses utilisées se déduisent ainsi d'une adresse dite de base. La plupart du temps, l'adresse de base du premier port parallèle est 378h (resp. 278h pour le deuxième). Mais ces valeurs peuvent varier d'une configuration à l'autre. Ainsi une carte Hercules avec port parallèle intégré attribuera au premier port l'adresse 3bch. En cas de doute, on trouve à partir de l'adresse 0:0408h la série des adresses de base pour LPT1, LPT2, etc...

A l'adresse de base se trouve en principe le registre de données. Ce dernier reçoit les données à émettre sur le port (il est "write only"). Pour chaque "1" transmis, la ligne de données associée est mise à la tension High, un "0" l'abaissant au niveau Low.

A l'adresse qui suit directement l'adresse de base se trouve le registre d'état qui donne des informations retournées par l'imprimante. Les lignes de commande correspondantes du câble sont en prise directe avec ce registre (seul BUSY est représenté sous forme inversée) qui est du type Read Only.

Bit	Signification
7	Busy (0 = l'imprimante ne peut plus accepter de données pour le moment)
6	ACK (0 = l'imprimante a lu le caractère)
5	PE, Paper empty (1 = plus de papier)
4	SLCT (0 = l'imprimante est hors ligne)
3	Error (0 = une erreur est survenue)
2-0	réservé

L'indicateur Busy signale que l'imprimante est occupée (busy) et qu'elle n'est donc plus en situation d'accepter de nouvelles données (le plus souvent parce que son buffer est saturé). Le signal d'acquittement Acknowledge (ACK) est remis à 0 à chaque fois que l'imprimante a prélevé le caractère envoyé sur la ligne de données qui peut donc être réutilisée. PE et Error indiquent des erreurs à corriger par l'utilisateur. SLCT reflète l'état courant du commutateur On Line de l'imprimante. Lorsque l'imprimante est hors ligne (Off line) elle ne peut recevoir de données.

Le dernier registre de tout port parallèle est le registre de contrôle associé à des lignes de commande. Il peut être lu ou au contraire recevoir des informations. Sa structure est la suivante :

Bit	Signification
7-5	Réservé
4	IRQ enable (1 = IRQ actif)
3	SLCT (0 = met l'imprimante hors ligne)
2	Reset (0 = effectue une réinitialisation de l'imprimante)
1	Auto LF (1 = l'imprimante ajoute à tout CR un LF)
0	Strobe (0 = données disponibles)

En mettant à 1 le bit 4 on peut théoriquement activer l'IRQ 5 ou 7 déclenché à chaque signal d'acquittement (ACK). Mais pour éviter les interférences avec les cartes sonores, il est généralement préférable de désactiver cette caractéristique. La plupart du temps le port parallèle est exploré en mode polling (le processeur attend la modification d'un indicateur).

La ligne SLCT permet de mettre hors ligne certaines imprimantes à partir de l'ordinateur. La ligne Auto LF sert parfois (sur certaines imprimantes seulement) à demander la génération automatique du caractère LF. Toutes les imprimantes par contre exploitent les lignes Reset et Strobe. Cette dernière indique au récepteur qu'un octet valide est disponible sur les lignes de données.

Pilotage de l'imprimante

Il est donc tout à fait simple d'émettre un caractère vers l'imprimante. Il suffit d'attendre que le bit Busy soit à 1 (ligne basse), puis d'écrire le caractère sur le port de données et de lancer un signal Strobe (activer la ligne puis la désactiver immédiatement). A titre de démonstration vous trouverez sur le CD, sous le nom PAR_TEST.PAS, un programme qui imprime une chaîne formée de tous les caractères existants :

```
Const Base=$378;                {adresse de base du port parallèle }

Procedure PutChar_Par(c:Char);
{transmet un caractère au port parallèle (adresse de base en "Base")}
Begin
  While Port[Base+1] and 128 = 0 Do; {Attend la fin du signal Busy }
  Port[Base]:=Ord(c);             {envoie le caractère sur le port }
   Port[Base+2]:=Port[Base+2] or 1; {émet un Strobe }
   Port[Base+2]:=Port[Base+2] and not 1;
   While Port[Base+1] and 64 = 1 do; {attend l'acquittement Ack }
End;
```

```
Procedure PutString_Par(s:String);
{transmet une chaîne de caractère au port parallèle en utilisant
 PutChar_Par)}
Var i:Integer;                    {compteur de caractères }
Begin
  For i:=1 to Length(s) do        {prend chaque caractère }
    PutChar_Par(s[i]);            {et l'envoie sur le port }
End;

Begin
  PutString_Par('Bonjour, ceci est un test d'impression de
                 Micro Application'#13#10);
  PutString_Par('abcdefghijklmnopqrstuvwxyz0123456789'#13#10);
End.
```

3.2. Autres applications

Ce n'est pas pour rien que le port parallèle ne s'appelle pas port d'imprimante. Il peut en effet servir à d'autres fins. Il dispose de 8 sorties (lignes de données), 5 entrées (lignes de commande) et 4 canaux d'entrée/sortie (dans le registre de contrôle) qui peuvent selon le cas servir comme entrée ou comme sortie.

L'une des applications fréquemment rencontrées est la mise en réseau de deux ordinateurs par l'intermédiaire d'un câble parallèle dit "null modem" avec le support du driver INTERLNK.EXE qui est fourni avec DOS (depuis la version 6.0). Ce driver exploite un protocole de transmission sur 4 bits qui utilise en sortie les registres de données D0-D3 et en entrée les lignes Error, SLCT, PE et ACK. La fonction de la ligne Strobe (qui est d'annoncer la présence de données) est reprise en sortie par D4 et en entrée par Busy. Le câble est donc connecté comme suit :

Broche avec broche	
2	15
3	13
4	12
5	10

Broche avec broche	
6	11
15	2
13	3
12	4

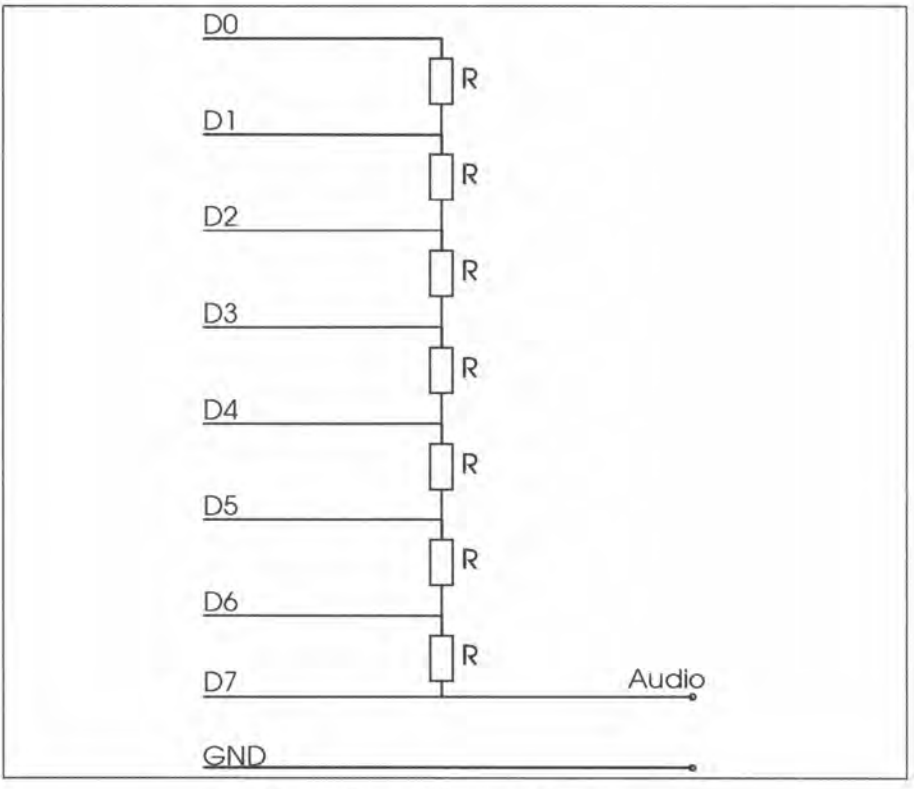
Broche avec broche	
11	6
10	5

Ce câble permet aussi de faire fonctionner d'autres programmes commerciaux de transfert, qui ne se différencient que par le logiciel employé. Il constitue

donc une sorte de standard. Il est évidemment possible d'écrire ses propres programmes de transfert mais il en existe aussi un grand nombre sur le marché du shareware.

Une autre possibilité offerte par le port parallèle mais de moins en moins usitée est son exploitation comme succédané de carte sonore. Un port parallèle n'est en fait rien d'autre qu'une sortie digitale sur 8 bits. Si les données émises sont converties en signaux analogiques, leur audition devient possible.

La conversion digitale/analogique demande quelques circuits spécialisés mais au-delà on risque de monter une véritable carte sonore. Il est en fait beaucoup moins coûteux de mettre en service quelques résistances pour obtenir un résultat presque identique.



Convertisseur DA avec résistances

Les résistances sont montées de façon que la ligne D7, dont le poids est le plus élevé, contribue le plus intensément à la formation du signal de sortie analogique. Les autres lignes (D6-D0) interviennent ensuite de façon décroissante.

Cet adaptateur (Covox) est simplement enfiché dans le port parallèle et relié à un amplificateur. Il peut être piloté par n'importe quel programme Mod Player, le principe de la commande étant on ne peut plus simple : les données sonores exploitent un certain nombre d'octets par secondes (jusqu'à 44000) qui reproduisent l'évolution des oscillations analogiques. Le PC utilisant à l'inverse de l'Amiga un format non signé, ces données peuvent être émises directement sur le port parallèle (registre de données, en LPT1 le plus souvent à l'adresse 378h, cf plus haut). Il n'y a pas à s'occuper des deux autres registres qui servent simplement à gérer la communication avec les imprimantes et autres choses semblables. Les lignes de données reproduisent exactement le contenu du port de sortie. Si on respecte le taux d'échantillonnage (sampling rate) en émettant les données à la même vitesse (par exemple pour 22 Khz le rythme sera de 22000 octets par seconde), un signal audio se fera entendre en sortie. Mais le processeur est très sollicité car les accès au port durent une "éternité". Il n'est donc généralement pas recommandable de faire en même temps des tâches consommatrices de ressources de calcul, comme par exemple des opérations de calcul. Dans ce cas, il vaut mieux acheter une carte sonore.

4. Savoir-faire secret en matière de graphique

Nous commencerons par donner un petit glossaire des termes techniques de base utilisés dans la programmation graphique du PC. Cette liste n'a aucune prétention à l'exhaustivité, elle a simplement pour but de permettre au développeur novice d'assimiler le vocabulaire de ce monde particulier. Ceux d'entre vous qui sont déjà des pros regretteront çà et là quelques simplifications mais il ne faut pas oublier que cet ouvrage s'adresse aux débutants, non aux ingénieurs des Grandes Ecoles.

4.1. Le vocabulaire incontournable

Voici quelques termes techniques avec leur explication.

pixel, pel (point)

Les écrans des ordinateurs affichent des images composées de points élémentaires ou pixels. En mode graphique les pixels peuvent être manipulés isolément avec leur couleur, tandis que le mode texte gère des blocs indivisibles de pixels qui représentent les caractères affichables.

Palette

Les modes VGA à 256 couleurs sont toujours obligés de recourir à une palette. A l'inverse, les modes TrueColor (16,7 millions de couleurs) ou même HiColor (65536 couleurs) associent à chaque pixel un ensemble de trois valeurs exprimant sa couleur à partir des composants de base : rouge, vert et bleu. En présence d'une palette, chaque pixel est associé à une sorte de pointeur qui renvoie à une entrée de palette. La palette comporte un jeu de 256 couleurs représentées sous la forme de leurs proportions élémentaires en

couleurs de base. Toutes les images de l'écran utilisent les seules couleurs de la palette.

Ce procédé a pour avantage d'économiser beaucoup de mémoire. Une entrée de palette occupe 18 bits alors que le pointeur associé à chaque pixel n'en prend que 8. Lorsque le mode graphique comporte davantage de couleurs, cet avantage ne joue plus car la palette prendrait des dimensions démesurées et chaque pixel demanderait 16 ou 24 bits.

Les modes basés sur la palette permettent certains effets graphiques intéressants. On peut par exemple modifier brutalement la couleur de tous les points associés à la couleur 1 en changeant l'entrée de palette correspondante (3 octets). On peut aussi faire apparaître ou disparaître progressivement une image sans manipuler chaque pixel : il suffit d'amener progressivement les 256 entrées de la palette de 0 à leur valeur finale. Vous trouverez une étude plus détaillée de ces effets et d'autres encore au chapitre 6.

Sprite

Un sprite n'est au fond rien d'autre qu'un dessin librement déplaçable sur l'écran. Mais il peut être transparent à certains endroits ce qui lui permet de se déplacer par dessus un fond. Certains ordinateurs familiaux possèdent des circuits spécialisés dans la gestion des sprites, ce qui présente l'avantage de décharger le processeur de calculs fastidieux, mais en contrepartie le sprite doit avoir des paramètres strictement définis (par exemple une taille fixe).

Rayon cathodique

Rayon produit par des électrons accélérés qui provoque un éclair lumineux d'une certaine intensité et d'une certaine couleur à l'endroit où il rencontre le revêtement de l'écran. C'est à partir de ces points d'impact qu'une image est dessinée ligne par ligne (de gauche à droite, si on se place devant l'écran) et de haut en bas.

Retour de balayage

Il s'agit d'un mouvement du rayon cathodique lors du dessin d'une image. Un retour de balayage peut être horizontal ou vertical. Le retour de balayage horizontal a lieu à la fin d'une ligne lorsque le rayon cathodique passe au début de la ligne suivante. Le retour de balayage vertical est déclenché lorsque le rayon a atteint l'extrémité inférieure de l'écran et qu'il est ramené sur la première ligne.

En principe, toute modification du contenu d'un écran (y compris sur certains registres VGA) ne doit avoir lieu que pendant un retour de balayage (retrace) pour éviter des perturbations génératrices de scintillement. Que le retour de balayage soit vertical ou horizontal n'a pas d'importance mais si les modifi-

cations sont importantes ou si les manipulations concernent les registres VGA (lents à réagir), il est préférable d'attendre un retour de balayage vertical qui dure plus longtemps (près de 200 fois).

Pour attendre le retour de balayage vertical, on utilise la procédure WaitRetrace (dans le module MODEXLIB.ASM, mais valable pour tout mode graphique ou texte). Cette procédure exploite le fait que le registre 1 de l'Input Status (adresse de port 3dah) signale l'exécution d'un retour de balayage vertical dans son bit 3, ce qui permet à tout moment de le détecter.

En fait, il ne suffit pas de vérifier qu'un retour de balayage est en cours. Il se pourrait en effet que ce retour au moment où on le surprend à l'appel de la procédure WaitRetrace soit déjà proche de sa fin. Dans ce cas, WaitRetrace rend la main sans tarder mais les modifications d'écran entreprises n'auront peut-être pas le temps de s'exécuter complètement. On est donc obligé d'attendre explicitement un début de retour de balayage. La boucle @wait1 laisse éventuellement passer un retour de balayage déjà entamé. Une fois le rayon cathodique de retour sur l'écran, c'est la boucle @wait2 qui attend un nouveau retour, celui qui sera exploité.

```
WaitRetrace proc pascal far
  mov dx,3dah ;Input Status Register 1
@wait1:
  in al,dx    ;bit 3 = 0 si le rayon est en train de
              ;dessiner l'image de l'écran
  test al,8h
  jnz @wait1
@wait2:
  in al,dx    ;bit 3 = 1 si le rayon est en retour de
              ;balayage
  test al,8h
  jz @wait2
  ret         ;le rayon est au bas de l'écran
```

Double scan

Division de la résolution verticale par deux, de 400 à 200, par dédoublement de chaque ligne : il s'agit d'émuler le mode 200 lignes que la carte VGA ne maîtrise pas de façon satisfaisante.

Vitesse d'affichage

C'est l'alpha et l'oméga de la programmation graphique. Depuis les débuts de l'ère informatique, les programmeurs ont toujours eu à lutter contre la

lenteur des affichages graphiques. Il est vrai que les processeurs et les cartes deviennent de jour en jour plus rapides, et que les graphiques CGA ne posent plus de problème de vitesse. Mais pour maintenir la qualité à laquelle les utilisateurs sont habitués de nos jours, il faut connaître des astuces et ne pas hésiter à programmer directement les circuits graphiques. Souvent, l'accroissement de vitesse par rapport aux routines du Bios est dû à la suppression de certains contrôles de sécurité. Ainsi, les procédures de ce livre ne vérifient jamais les coordonnées. Il est possible de dessiner un point aux coordonnées (5000, 7000) ce qui donne évidemment des résultats imprévisibles.

Mais comme la plupart du temps c'est le programmeur qui a en main le destin des pixels qu'il commande, les tests de ce type sont superflus (le défileur d'étoiles éteint les pixels au bord gauche de l'écran et les rallume à droite). S'il s'avérait cependant nécessaire d'instituer des contrôles renforcés, notamment dans le cadre d'un programme interactif qui offrirait un espace de liberté à la souris de l'utilisateur, il faudrait placer les contrôles en dehors des procédures d'affichages (PutPixel, etc) de façon que ces dernières ne soient appelées qu'à bon escient.

4.2. Les bases : le mode 13h du Bios

En développant le standard VGA, IBM a fourni un moyen confortable d'adressage de l'écran dans le nouveau mode à 256 couleurs. Les programmeurs qui exploitent le Bios ont rapidement adopté cette méthode en mode vidéo 13h : il s'agit du chaînage des plans de bits dans un espace d'adressage linéaire.

Organisation de la mémoire

Pour le processeur (et donc pour le programmeur en mode 13h), cet espace commence au segment A000 de la mémoire système avec la plus simple des structures. A chaque pixel est associé un octet qui représente sa couleur en pointant sur une couleur de la palette. Les adresses se suivent dans l'ordre de parcours du rayon cathodique quand il dessine l'écran : à l'offset 0 se trouve le point de coordonnées (0,0), à l'offset 320 le point (0,1) jusqu'au point (319,199) qui est situé à l'offset 63999.

Tout pixel a donc une adresse de formule :

```
offset = Y *320 + X
```

En application voici déjà un programme simple (ETOILES.PAS) qui fait défiler des étoiles en allumant et en éteignant des pixels :

```
Uses Crt;
Var Etoiles:Array[0..500] of Record
                               x,y,Plan:Integer;
                             End;
  nb_et:Word;

Procedure PutPixel(x,y,col:word);assembler;
{dessine le point (x,y) en couleur col (Mode 13h)}
asm
  mov ax,0a000h                  {charge le segment}
  mov es,ax
  mov ax,320                     {offset = Y*320 + X}
  mul y
  add ax,x
  mov di,ax                      {charge l'offset }
  mov al,byte ptr col            {charge la couleur}
  mov es:[di],al                 {et place le point}
End;

Begin
  Randomize;                     {initialise le générateur de
                                  nombres aléatoires}
  asm mov ax,13h; int 10h End;   {déclenche le mode 13h }
  Repeat                         {exécution une fois par dessin
                                  d'écran}
    For nb_et:=0 to 500 do Begin {nouvelle position pour chaque
                                  étoile}
      With Etoiles[nb_et] do Begin
        PutPixel(x,y,0);         {efface l'ancien pixel}
        Dec(x,Plan shr 5 + 1);   {effectue le déplacement}
        if x <= 0 Then Begin     {sortie à gauche  ?}
          x:=319;                {alors réinitialisation }
          y:=Random(200);
          Plan:=Random(256);
        End;
        PutPixel(x,y,Plan shr 4 + 16); {dessine le nouveau pixel}
```

```
      End;
    End;
  Until KeyPressed;                {continue jusqu'à ce qu'on
                                    frappe une touche}
TextMode(3);
End.
```

La partie importante de ce programme est constituée par la boucle interne qui pour chacune des étoiles efface l'ancienne position puis calcule la nouvelle en fonction de sa vitesse (évaluée à partir de son plan). Lorsqu'une étoile atteint le bord gauche de l'écran (x<=0), elle réapparaît à droite avec une nouvelle ordonnée et une nouvelle vitesse tirées au hasard.

Une fois sa nouvelle position établie, l'étoile est rallumée en tenant compte de son plan : les étoiles les plus lentes se trouvent à l'arrière-plan avec une luminosité moindre. On exploite à cet effet une propriété de la palette standard sur toute carte VGA réinitialisée : les entrées 16 (noir) à 31 (blanc) contiennent les différents niveaux de gris, qu'il n'est donc pas nécessaire de programmer.

La gestion interne du mode 13h

L'adressage linéaire qui rend si simple la programmation du mode 13h n'est en fait qu'une simulation à l'usage du processeur. En réalité, la carte VGA reconvertit l'adresse linéaire en une adresse de plans. Les deux lignes d'adresse inférieures (bits 0 et 1 de l'offset) servent à sélectionner le plan d'écriture et de lecture. Les six bits restants (2-7) représentent l'adresse physique à l'intérieur des plans, après mise à 0 des bits 0 et 1.

Les modes textes utilisent en fait des méthodes similaires (adressage pair/impair). Dans ce cas, la ligne d'adresse inférieure sert à sélectionner l'un des plans 0 ou 1. Ainsi, du point de vue du processeur, les octets de caractère et d'attribut se trouvent côte à côte, alors qu'en interne les caractères sont situés dans le plan 0 et les attributs dans le plan 1. Les plans 2 et 3 servent de mémoire pour le jeu de caractères.

4.3. Le format d'image GIF

GIF est le format graphique le plus usité sur les PC, ceci non sans raison. Ce format a été développé en 1987 par les exploitants de CompuServe pour permettre aux utilisateurs des messageries électroniques d'échanger efficacement et au moindre prix des fichiers d'images. Il présente des avantages énormes sur les autres formats tels que PCX. Pour les échanges de données, il est important que le format utilisé soit indépendant du système. Il ne doit pas être lié à certains modes graphiques mais contenir les données sous une forme accessible à tout système graphique. GIF est capable d'enregistrer des images de résolution maximale 16000 x 16000 pixels avec une palette de 256 couleurs prélevées dans un fonds de 16,7 millions de couleurs. Plusieurs images peuvent exploiter la même palette globale ou locale, mais cette caractéristique est peu utilisée.

L'avantage décisif est ailleurs : GIF autorise en effet une excellente compression avec une vitesse de décompression très importante. La méthode employée est la méthode LZW qui sert d'ailleurs aussi à compresser les fichiers non graphiques.

Telle est la raison pour laquelle les programmes présentés dans cet ouvrage chargent leurs images dans le format GIF. Les démos font un usage raisonnable de la mémoire et le chargement des images est assez rapide.

Le standard de la scène

GIF utilise une structure de blocs, qui n'est pas aussi développée qu'avec TIFF mais qui permet de manipuler facilement les paramètres de base tels que la résolution et le nombre de couleurs. La structure précise est la suivante :

Offset	Longueur	Contenu
0	3	signature du format "GIF"
3	3	numéro de version, act. "87a" ou "89a"
6	7	Logical Screen Descriptor Block
0dh	n	Global Color Map (optionnel), en mode 256 couleurs compatible VGA avec (n=768 octets)
30dh	n	Extension Block (optionnel)
30dh	10	Image Descriptor Block
317h	n	Local Color Map (optionnel), en mode 256 couleurs compatible VGA (n=768 octets)
317h	n	Raster Data Block (données graphiques compressées LZW)
?	1	signature terminale (89h)

Les offsets indiqués supposent la présence d'une palette globale (Global Color Map) mais les blocs d'extension et la palette locale (Local Color Map) ne sont pas décomptés. A la place de la signature terminale, on peut trouver d'autres blocs descripteurs d'image, avec leurs palettes et les données graphiques. La signature terminale caractérise la fin du fichier. En fait, il est assez rare que l'on mémorise plusieurs images dans un même fichier. Notre préoccupation étant avant tout de disposer non pas d'un visualiseur de GIF universel mais d'une routine de chargement d'image rapide, nous supposerons par la suite qu'un fichier ne contient qu'une image.

Aux 6 octets de l'identification du format succède le bloc descripteur d'écran logique (Logical Screen Descriptor Block, LSDB) qui comme son nom l'indique définit la logique de l'écran, c'est-à-dire sa résolution et les caractéristiques de couleur. La structure de ce bloc est la suivante :

Offset	Longueur	Contenu
0	2	Largeur de l'écran
2	2	Hauteur de l'écran
4	1	Indicateur de résolution :
		Bit 7 : 1 = palette de couleurs globale disponible
		Bits 6-4 : profondeur de couleur -1
		Bit 3 : réservé (0)
		Bits 2-0 : nombre de bits/pixel -1
5	1	Couleur de fond (numéro dans la palette)
6	1	Pixel Aspect Ratio :
		Bit 7 : ordre de classement de la palette globale
		Bits 6-0 : Pixel Aspect Ratio

Ce bloc est suivi de la palette globale (Global Color Map) qui d'après le schéma VGA associe 3 octets à chacune de 256 couleurs (proportion des composants de base rouge, vert et bleu). Notez bien que chaque composant est codé sur 8 bits ce qui donne en tout 2 ^ 24 = 16,7 millions de couleurs. Sur une carte VGA, les indications doivent être décalées de deux bits vers la droite avant d'être envoyées au contrôleur VGA/DAC car VGA ne tient compte que des 6 bits inférieurs.

Une autre particularité qui ne joue généralement aucun rôle sur les PC est l'ordre de présentation de la palette. Qu'elles soient locales ou globales, les palettes peuvent être rangées dans l'ordre VGA : la couleur 0 est alors décrite par ses composants rouge bleu vert, puis la couleur 1 et ainsi de suite. C'est

là l'ordre normal. Mais bien que cette possibilité soit rarement exploitée, on peut aussi trier la palette par fréquence en mémorisant d'abord les proportions de rouge pour les couleurs 0 à 255, ensuite les proportions de vert et enfin de bleu.

Jusqu'ici, toutes les données sont globales, elles sont valables pour l'ensemble des images contenues dans le fichier GIF. Les indications qui suivent sont spécifiques aux images et doivent être rechargées pour chaque nouvelle image tirée du fichier.

Ce sont d'abord les blocs d'extension qui servent à stocker des informations quelconques, quelques programmes de numérisation ou de dessin y déposent par exemple des avis de copyright. Un chargeur de GIF élémentaire, comme nous le présentons dans ce chapitre, ne s'intéresse pas à ces données mais il doit pouvoir les sauter. A l'intérieur de la structure, une signature caractérise clairement les blocs d'extension : elle est constituée d'un point d'exclamation ("!", ascii 21h), suivi d'un code de fonction dont nous ne parlerons pas et de l'indication de la taille du bloc proprement dit. Un bloc d'extension peut inclure un nombre quelconque de blocs de données dont la taille est indiquée à chaque fois. L'ensemble du bloc d'extension se termine par un 0.

Il est suivi du bloc descripteur d'image qui possède la structure suivante :

Offset	Longueur	Taille
0	1	"," (2ch) Image Separator Header
1	2	abscisse x du coin supérieur gauche de l'écran logique
3	2	ordonnée y du coin supérieur gauche de l'écran logique
5	2	Largeur de l'image en pixels
7	2	Hauteur de l'image en pixel
9	1	indicateur :
		Bit 7 : 1 = palette locale disponible
		Bit 6 : 1 = image entrelacée
		Bit 5 : ordre' de placement de la palette locale
		Bits 3-4 : réservé (0)
		Bits 0-2 : nombre de bits/pixel -1

C'est l'indicateur qui constitue ici l'information la plus intéressante. Il signale d'abord si une palette locale fait suite au bloc descripteur d'image. Elle sera prioritaire par rapport à la palette globale. L'ordre de classement et le nombre de pixels peuvent également être spécifiés à part. Le bit 6 indique si l'image

est entrelacée, ce terme ayant le même sens que dans la technologie des moniteurs : les lignes d'écran paires (0, 2, 4 etc) sont dessinées avant les impaires. Ainsi, en cas de transmission lente (chargement sur messagerie ou sur réseau de type CompuServe, ou à partir d'une disquette), on obtient quand même au bout d'un certain temps une approximation du contenu de l'image.

Après le bloc descripteur vient, si elle existe, la palette locale qui se décrit comme la palette globale. En dernière place, on trouve les blocs de données graphiques qui constituent la trame de l'image en compression LZW. Comme la description de l'image, ces données sont également mémorisées par bloc. La longueur est codée sur 8 bits, ce qui donne une taille maximale de 256 octets par bloc (y compris l'octet qui donne la taille).

La compression LZW

Le premier bloc de données graphiques contient une indication particulière : il s'agit de la répétition de l'octet qui donne le nombre de bits par pixel. Cette indication est nécessitée par la méthode de compression LZW qui l'utilise pour bâtir son algorithme.

Pour nous, la très faible taille du fichier d'image obtenu par la méthode LZW est un avantage incontestable. S'il est vrai que cette méthode n'est pas rapide dans la phase de compression, elle se comporte très bien dans la phase de décompression.

Contrairement aux méthodes de compression simples comme le codage RLE (Run Length Encoding) utilisé par le format PCX, la méthode LZW n'est pas seulement capable de regrouper des suites d'octets identiques mais également des séquences de longueur quelconques non consécutives. Cette performance est due à l'utilisation d'un alphabet étendu qui en plus des 8 bits usuels exploite plusieurs bits supplémentaires pour le codage. Si on prend par exemple 9 bits par caractère, le fichier contiendra, en plus des codes 0-255, les codes 256-511 qui pourront correspondre à des chaînes entières de caractères.

Cette extension n'est pas affectée à priori, elle ne reçoit son contenu que pendant la compression (de même pour la décompression). En lui-même, l'algorithme de compression est relativement simple.

Il consiste à lire les caractères dans le fichier source (non compressé) jusqu'à ce que la chaîne formée ne se trouve plus dans l'alphabet. Au début, ceci se produit au deuxième caractère (le premier caractère étant dans le domaine 0-255, et aucune chaîne n'étant encore prise en considération).

La chaîne est alors ajoutée à l'alphabet et à la prochaine occurrence, elle sera remplacée par le code correspondant ce qui constituera une compression.

Le code de la chaîne la plus ancienne dans l'alphabet est écrit dans le fichier de destination et la chaîne est initialisée avec le dernier caractère lu. On lui accroche d'autres caractères jusqu'à avoir une nouvelle chaîne ne figurant pas dans l'alphabet.

Il se pose donc la question de savoir combien de bits il faut prendre pour l'alphabet. Si leur nombre est trop faible, l'alphabet déborde rapidement et il faut le réinitialiser, ce qui diminue le taux de compression. Si le nombre de bits est trop élevé, on provoque un gaspillage d'espace puisque les bits supérieurs ne seront pas utilisés. La solution est apportée par la méthode LZW modifiée qui introduit un code de taille variable. On commence avec 9 bits, ce qui ouvre un alphabet de 512 éléments. Si cette limite est dépassée, on ajoute un bit supplémentaire.

Il n'est cependant pas judicieux de pratiquer une extension sans fin car un nombre excessif de bits est consommé alors que les chaînes déjà recensées dans l'alphabet ne servent plus. C'est pourquoi, lorsque les douze bits sont dépassés, on envoie tôt ou tard un code d'assainissement (clear code) qui efface complètement l'alphabet, ramène le code à 9 bits et recommence en fait la compression.

Il apparaît clairement que l'efficacité de cet algorithme dépend étroitement de la taille du fichier à comprimer. Ce n'est que pour de grandes quantités de données que l'alphabet pourra resservir, ce qui conduit à loger de longues chaînes dans seulement quelques bits. Les images rentrent bien dans ce schéma car elles occupent le plus souvent des fichiers de bonne taille.

Il est important pour nous de nous pencher sur l'algorithme de décompression qui restitue les données dans leur structure d'origine, en formant à nouveau des images. Mais on ne peut comprendre la décompression si on n'a pas étudié auparavant l'opération inverse de compression.

Notons bien que dans la méthode LZW, l'alphabet n'est pas mémorisé dans le fichier compressé : il est en fait reconstitué pendant l'opération de décompression. Seules sont en effet codées des chaînes de caractères effectivement présentes dans les données d'origine.

Le décompresseur commence par tester si le caractère lu est dans une forme non compressée, ce qui se traduit par un code inférieur à 256. Dans ce cas, il peut être écrit directement dans le fichier de destination (ou la mémoire d'écran). Si c'est un code étendu, il faut rechercher dans l'alphabet la chaîne qui lui correspond. Il est donc nécessaire de reconstruire simultanément l'alphabet, ce qui se fait de la façon suivante : la dernière chaîne décompressée (le caractère non compressé) est combinée au premier caractère de la chaîne juste décodée et reportée dans l'alphabet.

C'est en fait la même démarche que pour la compression mais dans l'ordre inverse. A tout moment, les alphabets de compression et de décompression se correspondent - ou presque, car il existe comme toujours une exception au cours de laquelle apparaît un caractère dont le code n'est pas encore dans l'alphabet. Si une chaîne de type AbcAbcA est compressée et si la chaîne Abc existe déjà dans l'alphabet, le compresseur écrit le code de cette entrée dans le fichier de destination et constitue le nouvel élément d'alphabet AbcA qui réapparaît aussitôt après et se trouve réutilisé avec écriture dans le fichier de destination.

Mais à ce moment, le décompresseur ne connaît pas encore cette chaîne, comment pourrait-il savoir qu'il vient un A puisqu'il n'est pas écrit dans le fichier de destination ? Ce cas ne survient qu'avec des chaînes du type cité, ce qui permet de le détecter sans peine. S'il apparaît un code qui n'est pas encore dans l'alphabet, la dernière chaîne décodée augmentée de son premier caractère est écrite dans la mémoire d'écran et la nouvelle chaîne est notée dans l'alphabet.

Autrefois, le manque de mémoire posait problème car la gestion d'un alphabet en consomme beaucoup. L'algorithme a été amélioré en conséquence, il est devenu en apparence plus compliqué mais en réalité certaines choses se sont simplifiées. Comme nous l'avons vu, chaque nouvelle entrée d'alphabet est créée, tant à la compression qu'à la décompression, par addition d'un caractère à une chaîne déjà disponible. On ne mémorise donc que le code de l'ancienne chaîne et celui du nouveau caractère, ce qui ne demande plus que deux entrées (appelées préfixe et suffixe)

Chargeur de GIF optimisé pour la résolution 320 x 200

De toute évidence, le format GIF est un format universel qui autorise de nombreuses résolutions et niveaux de couleurs et reste par ailleurs complètement indépendant de tout système. Un bon visualiseur de GIF devra connaître et supporter toutes les variantes du format, ce qui n'est guère favorable à une grande vitesse d'exécution. Mais il existe déjà toute une gamme de visualiseurs de GIF dans le monde du shareware, de sorte que l'objet de ce chapitre n'est sûrement pas d'en rajouter un nième. Nous allons plutôt développer un chargeur optimisé pour certaine variante de format qui aura l'avantage de compenser sa spécialisation par sa grande vitesse d'exécution.

Dans le monde des jeux et des démos, la résolution préférée est sans aucun doute de 320 x 200 en 256 couleurs. Nous travaillerons par préférence dans ce mode sans exclure une extension facile à d'autres modes en 256 couleurs. Mais nous laisserons tomber le mode 16 couleurs car il faudrait réécrire le décompresseur de fond en comble (niveaux de couleurs sur 4 bits au lieu de 8).

La routine LoadGIF de l'unité GIF utilisée par divers programmes de démonstration de cet ouvrage va nous servir d'exemple. Elle utilise l'unité MODEXLIB.TPU que vous obtiendrez en compilant MODEXLIB.PAS :

```
unit gif;                          {en-tête pour gif.asm}

Interface
uses modexlib;                      {à cause de SetPal}
var
  vram_pos,                         {position courante dans la RAM VGA}
  rest, numerr:word;                {autres octets en mémoire centrale
                                     et numéro d'erreur }
  gifname:String;                   {Nom avec #0}
 Procedure LoadGif(GName:String);{charge le fichier Gig
                                     "GName.gif" dans vscreen}
  Procedure LoadGif_Pos(GName:String;Posit:Word);
{charge le fichier Gif à l'offset d'écran Posit}

Implementation
  Procedure ReadGif;external;
  {chargeur de Gif, compl. en Asm}
  {$l gif}                          {indique le chemin d'accès
                                     de gif.obj si nécessaire}

  Procedure LoadGif;
  {charge le fichier "GName.gif" dans vscreen}
  Begin
    If pos('.',gname) = 0 then      {ajoute si nécessaire
                                     l'extension ".gif" }
      gname:=gname+'.gif';
    Gifname:=GName+#0;;             {crée une chaîne ASCIIZ }
    vram_pos:=0;                    {commence à l'offset 0 dans
                                     la mémoire VGA }
    ReadGif;                        {charge l'image }
    If Numerr <> 0 Then            {s'arrête en cas d'erreur }
      Halt(Numerr);
    SetPal;                         {fixe la palette chargée }
  End;
```

```
Procedure LoadGif_pos;
{charge le fichier Gif à l'offset d'écran Posit}
Begin
  If pos('.',gname) = 0 then   {ajoute si nécessaire
                                l'extension ".gif" }
    gname:=gname+'.gif';
  Gifname:=GName+#0;           {crée une chaîne ASCIIZ }
  vram_pos:=posit;             {commence en mémoire VGA à la
                                position indiquée }
  ReadGif;                     {charge l'image }
  If Numerr <> 0 Then          {s'arrête en cas d'erreur }
    Halt(Numerr);
  SetPal;                      {fixe la palette chargée }
End;

Begin
  numerr:=0;                   {normalement pas d'erreur }
  GetMem(VScreen,64000);       {alloue un écran virtuel }
End.
```

Les seules procédures de cette unité LoadGIF et LoadGIF_Pos ne servent pratiquement que de cadre à l'appel de la partie en assembleur pour simplifier le processus de chargement. LoadGIF est la procédure de chargement normale : elle transfère simplement une image dans l'écran virtuel et teste les débordements (image supérieure à 320 x 200) qui déclenchent un déchargement en mémoire d'écran à partir de l'offset 0. LoadGIF_Pos est en plus capable de traiter comme argument l'offset où débute le déchargement.

Les deux procédures ajoutent en cas de besoin l'extension .gif au nom du fichier transmis et elles clôturent la chaîne par un zéro terminal réclamé par DOS. Une gestion d'erreur simplifiée a été implémentée, elle se contente d'empêcher un plantage du système en mettant fin au programme lorsque ReadGIF transmet un code d'erreur différent de 0. A l'inverse d'un visualiseur GIF universel, une démo peut supposer que ses fichiers sont situés dans le répertoire courant. La dernière partie fait encore appel à SetPal qui réalise la palette de couleurs chargée.

La procédure de chargement proprement dite ReadGIF se trouve dans le fichier GIF.ASM qui étant écrit en assembleur permet d'atteindre une vitesse d'exécution maximale.

```
.286

Clr=256                         ;code d'effacement de l'alphabet
eof=257                         ;code de fin de fichier
w equ word ptr
b equ byte ptr
data segment public
  extrn gifname:dataptr         ;Nom du fichier Gif, avec
                                ;".gif" + db 0
  extrn vscreen:dword           ;pointe sur la mémoire de
                                ;destination
  extrn palette:dataptr         ;palette de destination
  extrn vram_pos:word           ;position à l'intérieur de la
                                ;mémoire d'écran
  extrn rest:word               ;nbre de codes restant à copier
  extrn numerr:word;            ;indicateur d'erreur

  handle     dw 0               ;handle DOS pour fichier Gif
  buff       db 768 dup (0)     ;buffer des données lues
  buffInd     dw 0              ;pointeur à l'intérieur de ce
                                ;buffer
  abStack    db 1281 dup (0)    ;pile de décodage
  ab_prfx    dw 4096 dup (0)    ;alphabet, préfixe
  ab_suffx    dw 4096 dup (0)   ;alphabet, suffixe
  free      dw 0                ;position libre dans l'alphabet
  nbbit    dw 0                 ;nombre de bits dans un code
  max        dw 0               ;nombre maximum d'entrées dans
                                ;un alphabet à nbbit
  stackp    dw 0                ;pointeur dans la pile de l'alphabet
  restbits  dw 0                ;nombre de bits à lire
  restbyte  dw 0                ;nombre d'octets disponibles dans
                                ;le buffer
  casspecial dw 0               ;mémoire pour le cas spécial
  act_code  dw 0                ;code traité
  old_code  dw 0                ;code précédent
  readbyt   dw 0                ;code en cours de lecture
  lbyte      dw 0               ;dernier code lu
data ends
```

```
extrn p13_2_modex:far          ;utilisé en cas de dépassement du
                               ;segment

code segment public
assume cs:code,ds:data

public readgif
GifRead proc pascal n:word     ;lit n octets dans le fichier
  mov ax,03f00h                ;fonction 3fh de l'interruption 21h
  mov bx,handle                ;charge le handle
  mov cx,n                     ;et le nombre d'octets à lire
  lea dx,buff                  ;pointe sur le buffer de destination
  int 21h                      ;exécute l'interruption
  ret
gifread endp

GifOpen proc pascal
;ouvre le fichier Gif pour y accéder en lecture
  mov ax,03d00h                ;fonction 3dh: ouvrir
  lea dx,gifname + 1           ;pointe sur le nom
                               ;(saute l'octet qui mémorise
                               ;la taille)
  int 21h                      ;exécute l'interruption
  mov handle,ax                ;sauvegarde le handle
  ret
gifopen endp
GifClose proc pascal
;ferme le fichier Gif
  mov ax,03e00h                ;fonction 3eh: fermer
  mov bx,handle                ;charge le handle
  int 21h                      ;exécute l'interruption
  ret
gifclose endp

GifSeek proc pascal ofs:dword
;positionne le pointeur du fichier
  mov ax,04200h  ;fonction 42h,
```

```
  mov bx,w handle                ;sous-fonction 0: Seek
                                 ;relatif au début du fichier
  mov cx,word ptr ofs + 2        ;charge l'offset
  mov dx,word ptr ofs
  int 21h                        ;exécute l'interruption
  ret
Endp
ShiftPal proc pascal
;adapte la palette 24 bits au format VGA (18 bits)
  mov ax,ds                      ;segment de données pour
                                 ;tables source et destination
  mov es,ax
  mov si,offset buff             ;lecture dans le buffer de données
  lea di,palette                 ;écriture dans la palette de
                                 ;destination
  mov cx,768d                    ;traite 786 octets
@l1:
  lodsb                          ;lit un octet
  shr al,2                       ;le convertit
  stosb                          ;et l'enregistre
  loop @l1
  ret
Endp
FillBuff proc pascal
;lit tout un bloc du fichier en Buf
  call gifread pascal,1          ;lit un octet
  mov al,b buff[0]               ;transfère la taille en al
  xor ah,ah
  mov w restbyte,ax              ;et la mémorise en restbyte
  call gifread pascal, ax        ;lit les octets
  ret
Endp

GetPhysByte proc pascal
;lit un octet dans le buffer
  push bx                        ;bx préservé pour l'appelant
  cmp w restbyte,0               ;plus de données dans le buffer ?
  ja @yadurab
```

```
  pusha                          ;remplit à nouveau le buffer
  call fillbuff
  popa
  mov w buffInd,0                ;réinitialise le pointeur
@yadurab:                        ;il y a des données dans le buffer
  mov bx,w buffInd               ;charge le pointeur
  mov al,b buff[bx]              ;lit un octet
  inc w buffInd                  ;déplace le pointeur
  pop bx                         ;c'est tout
  ret
Endp
GetLogByte proc pascal
;lit un code dans le buffer en exploitant GetPhysByte
  push si                        ;registre "si" préservé pour
                                 ;l'appelant
  mov ax,w nbbit;lit la taille en bits du code
  mov si,ax ;et la sauvegarde
  mov dx,w restbits              ;décale lbyte de 8-restbits
                                 ;vers la droite
  mov cx,8
  sub cx,dx                      ;calcule la différence
  mov ax,w lByte
  shr ax,cl                      ;et lance le décalage
  mov w act_code,ax              ;sauve le code
  sub si,dx                      ;restbits déjà lus -> soustraire
@nextbyte:
  call getphysbyte               ;lit un nouvel octet
  xor ah,ah
  mov w lByte,ax                 ;le met en lByte pour le
                                 ;prochain code
  dec w restbyte                 ;octet traité
   mov bx,1                      ;masque les bits restants
                                 ;dans l'octet
  mov cx,si                      ;fixe le nombre de bits
  shl bx,cl                      ;décale 1 du nombre de bits
  dec bx                         ;et décrémente
  and ax,bx                      ;masque le code
```

```
  mov cx,dx                    ;effectue le décalage recherché
  shl ax,cl                    ;de restbits vers la gauche
  add w act_code,ax            ;ajoute le résultat

  sbb dx,w nbbit               ;diminue le nombre restbits
  add dx,8                     ;de ce qui dépasse 8 bits
  jns @positif
  add dx,8
@positif:
  sub si,8                     ;jusqu'à 8 bits cherchés ->
                               ;soustraire
  jle @fini                    ;<= 0 -> tout est fini
  add dx,w nbbit;sinon augmente restbits du nombre de bits manquants
  sub dx,8
  jmp @nextbyte                ;et on continue
@fini:
  mov w restbits,dx            ;sauve le nombre de bits
                               ;restants pour le prochain appel
  mov ax,w act_code            ;et charge ax
  pop si
  ret
Endp

ReadGif proc pascal
;lit en vscreen un fichier Gif appelé gifname,
;en cas de dépassement, stockage en mémoire d'écran
  push ds                      ;sauvegarde ds
  call GifOpen                 ;ouvre le fichier
  jnc ok                       ;erreur ?
  mov numerr,1                 ;la signaler et conclure
  pop ds
  ret

ok:
  call gifseek pascal, 0,13d   ;saute les 13 premiers octets
  push 768d                    ;lit les 768 octets de la palette
  call gifread
  call shiftpal                ;et l'enregistre avec conversion
```

```
  call gifread pascal,1          ;passe un octet

@extloop:                        ;laisse tomber les blocs d'extension
  cmp w buff[0],21h              ;encore un bloc d'extension ? ?
  jne @noext;non, on continue
    call gifread pascal,2        ;les deux premiers octets...
    mov al,b buff[1]             ;donnent la longueur du bloc
    inc al                       ;...si on ajoute 1
    xor ah,ah
    call gifread pascal, ax      ;on passe par dessus
  jmp @extloop

@noext:
  call gifread pascal, 10d       ;lit le reste de l'IDB
  test b buff[8],128             ;une palette locale  ?
  je @noloc                      ;non on poursuit
    push 768                     ;lit la palette
    call gifread
    call shiftpal                ;et la convertit

@noloc:
  les di,dword ptr vscreen       ;charge l'adresse de destination

  mov w lbyte,0                  ;dernier code lu 0
  mov w free,258                 ;première entrée libre 258
  mov w nbbit,9                  ;nb de bits d'un code = 9
  mov w max,511                  ;donc maximum d'entrées = 511
  mov w stackp,0                 ;pointeur sur début
  mov w restbits,0               ;pas de bits restants
  mov w restbyte,0               ;rien à chercher
@mainloop:                       ;boucle pour chaque code
  call getlogByte                ;lit un code
  cmp ax,eof                     ;signature fin de fichier ?
  jne @no_abandon
  jmp @abandon                   ;oui, abandonne
@no_abandon:
  cmp ax,clr                     ;code clr  ?
  jne @no_clear
```

```
  jmp @clear                    ;oui efface l'alphabet
@no_clear:
  mov w readbyt,ax              ;sauve le code actuel
  cmp ax,w free                 ;est-il déjà dans l'alphabet ?
                                ; (<free)
  jb @code_in_ab                ;oui, passe au traitement
  mov ax,w old_code             ;non, donc cas spécial, transmettre
   mov w act_code,ax            ;au traitement la dernière chaîne
  mov bx,w stackp
  mov cx,w casspecial           ;et accrocher le premier caractère
                                ;(toujours concret)
  mov w abstack[bx],cx          ;gère la pile
  inc w stackp                  ;en conséquence
@code_in_ab:                    ;code disponible dans l'alphabet :
  cmp ax,clr                    ;< code clr ?
  jb @concret                   ;oui, caractère concret
@fillstack_loop:                ;sinon passe au décodage
  mov bx,w act_code             ;le code est un pointeur dans
                                ;l'alphabet
  shl bx,1                      ;Word Array (!)
  push bx
  mov ax,w ab_suffx[bx]         ;cherche le suffixe, qui est concret
  mov bx,w stackp               ;le place sur la pile
  shl bx,1                      ;considérée comme Word Array
  mov w abstack[bx],ax
  inc w stackp
  pop bx
  mov ax,w ab_prfx[bx]          ;cherche le préfixe
  mov w act_code,ax             ;le prend comme code courant
  cmp ax,clr                    ;> code clr
  ja @fillstack_loop            ;poursuit le décodage
@concret:                       ;plus que des caractères
                                ;concrets sur la pile
  mov bx,w stackp               ;empile le dernier code
  shl bx,1                      ;comme Word Array
  mov w abstack[bx],ax
  mov w casspecial,ax           ;en prend note aussi pour le
                                ;cas spécial
```

```
  inc w stackp                    ;fait progresser le pointeur
  mov bx,w stackp                 ;prépare la lecture de la pile
  dec bx                          ;décrémente le pointeur
  shl bx,1                        ;sur Word Array
@readstack_loop:                  ;traite la pile
  mov ax,w abstack[bx]            ;prend un caractère
  stosb                           ;et l'écrit dans la mémoire de
                                  ;destination

  cmp di,0                        ;dépasse-t-on le segment ?
  jne @noovl1
  call p13_2_modex pascal,vram_pos,16384d
  add vram_pos,16384d             ;exploite la mémoire d'écran
  les di,dword ptr vscreen        ;nouvelle position en mémoire VGA

@noovl1:
  dec bx                          ;pointeur de pile sur l'élément
                                  ;suivant
  dec bx
  jns @readstack_loop             ;fini ? non on poursuit
  mov w stackp,0                  ;réinitialise le pointeur de pile
  mov bx,w free                   ;met à jour l'alphabet
  shl bx,1                        ;à la position "free"
  mov ax,w old_code               ;écrit le dernier code dans préfixe
  mov w ab_prfx[bx],ax
  mov ax,w act_code               ;et le code courant dans suffixe
  mov w ab_suffx[bx],ax
  mov ax,w readbyt                ;le code  lu est le plus récent
  mov w old_code,ax
  inc w free                      ;met à jour la dernière
                                  ;position libre dans l'alphabet
  mov ax,w free
  cmp ax,w max                    ;déjà saturé ??
  ja @no_mainloop
  jmp @mainloop                   ;non, on continue
@no_mainloop:
  cmp b nbbit,12                  ;nombre de bits déjà égal à 12 ?
  jb @no_mainloop2
```

```
  jmp @mainloop                ;oui, alors on recommence simplement
@no_mainloop2:
  inc w nbbit                  ;sinon on l'augmente
  mov cl,b nbbit               ;nouveau maximum d'entrées
  mov ax,1                     ;1 décalé de nbbit vers la gauche
  shl ax,cl
  dec ax                       ;puis décrémenté
  mov w max,ax                 ;enregistre le maximum
  jmp @mainloop                ;revient à la boucle principale
@clear:                        ;réinitialise l'alphabet:
  mov w nbbit,9                ;nbbit reprend la valeur de départ
  mov w max,511                ;le maximum d'entrées est 511
  mov w free,258               ;première place libre = 258
  call getlogbyte              ;lit le code suivant
  mov w casspecial,ax          ;le note pour le cas spécial
  mov w old_code,ax            ;et comme ancien code
  stosb                        ;passe directement en mémoire
                               ;car concret

  cmp di,0                     ;dépassement de segment ?
  jne @noovl2
  call p13_2_modex pascal,vram_pos,16384d
  add vram_pos,16384d          ;stockage dans la mémoire d'écran
  les di,dword ptr vscreen     ;avance le pointeur en mémoire  VGA

@noovl2:
  jmp @mainloop                ;retour à la boucle principale
@abandon:                      ;abandon par code Eof
  call gifclose                ;ferme le fichier
  mov rest,di                  ;mémorise le nombre de codes
                               ;restant à copier
  pop ds                       ;et clôture
  ret
Endp

code ends
end
```

Les premières procédures de ce module servent à la gestion du fichier : GIFOpen, GIFRead, GIFSeek et GIFClose procèdent à l'ouverture, à la lecture (de n octets), au positionnement et à la fermeture du fichier.

La procédure ShiftPal quant à elle n'a rien à voir avec le chargement : elle lit la palette dans le buffer où elle a été transférée par GIFRead et la met sous forme de table appelée "palette", directement manipulable. Une conversion est effectuée en même temps, le format des entrées passant de 8 bits (norme GIF) à 6 bits (norme VGA) par décalage binaire à droite.

Le niveau suivant du chargeur GIF est constitué par la procédure GetPhysByte qui a pour mission de tirer un octet du fichier. On pourrait obtenir ce résultat avec GIFRead mais, malgré l'existence de mémoire cache, les accès individuels au disque sont toujours plus lents qu'une gestion de buffer intermédiaire. GetPhysByte teste d'abord s'il reste des données dans le buffer, ce dont témoigne le compteur restbyte. Si tel est le cas, l'octet suivant est lu sinon le buffer doit être rechargé par FillBuff. Ce dernier sous-programme lit la taille du bloc courant et ensuite le nombre correspondant de données dans le fichier.

La méthode de compression demande que les données lues fassent l'objet d'un traitement supplémentaire. Un "octet" de code n'a plus 8 bits mais au moins 9, selon l'état courant de l'alphabet. A partir de ce flot continu de bits (transmis par getPhysByte par blocs de 8 bits ou octets physiques), il faut isoler le nombre de bits constituant un code. Cette tâche est prise en charge par GetLogByte.

Comme on travaille avec des octets partiellement pris en compte, les variables lByte et restbits jouent un rôle important. lByte contient le dernier octet physique fourni par GetPhysByte. De cet octet, il reste à traiter restbits bits. A cet effet, on décale lByte de 8 - restbits bits vers la droite, ce nombre étant stocké en act_code.

A l'étape suivante, un nouvel octet est lu dont les bits inférieurs nécessaires sont extraits par masque et ajoutés à act_code. Avec des codes de plus de 10 bits, il peut être nécessaire de prélever un autre octet physique. Dans ce cas, on retourne à l'étiquette @nextbyte pour en tenir compte.

La plus importante procédure de ce module (et la seule déclarée publique) est ReadGIF. Elle reçoit comme arguments de portée globale : GIFName qui contient le nom complet du fichier avec caractère nul terminal et vscreen qui pointe sur l'écran virtuel supposé alloué au préalable (tâche assurée au démarrage de l'unité).

Après ouverture du fichier (avec test d'erreur), on survole les treize premiers octets qui contiennent des données sans intérêt pour nous (signature et descripteur d'écran logique), étant donné qu'on se limite à un format de 320 x 200. Les autres formats ne sont pas reconnus et donnent n'importe quoi

sur l'écran mais nous avons déjà dit que nous ne recherchions pas l'universalité mais avant tout la vitesse d'exécution.

La présence d'une palette globale est également supposée. Après chargement, elle est convertie par ShiftPal. La signature du bloc suivant est lue, les blocs d'extension sont ignorés. S'il n'y a plus de bloc d'extension, on se trouve en présence du bloc descripteur d'image (IDB). On lit alors 10 octets : 9 en provenance de l'IDB et le premier octet tiré des données graphiques. Si une palette locale est disponible (bit 7 du registre des indicateurs de l'IDB), elle est chargée et prise en compte. L'étiquette @noloc introduit le processus de décodage proprement dit des données graphiques. On commence par initialiser quelques variables : free indique le premier emplacement libre de l'alphabet, nbbit le nombre de bits formant un code, max la dernière position de l'alphabet courant. La boucle principale @mainloop est ensuite parcourue pour chaque code. Chaque code lu est d'abord comparé aux codes spéciaux eof (end of file) et clr (clear alphabet). Dans le premier cas, le fichier se trouve complètement chargé, dans le deuxième cas la compression a provoqué un débordement de l'alphabet qui nécessite son effacement et une réinitialisation. Pour que la compression et la décompression fonctionnent en synchronisation, il faut aussi réinitialiser l'alphabet dans la phase de décompression.

Le code lu doit à présent être confronté à l'alphabet. S'il ne s'y trouve pas, on est en présence du cas particulier étudié plus haut qui est traité en conséquence : la dernière chaîne comprimée (disponible dans old_code) est combinée avec l'avant dernier caractère et déposée sur la pile (abstack). Cette pile sera retraitée plus tard dans l'ordre inverse de sa création pour être transférée dans la mémoire principale.

Qu'il y ait ou non un cas spécial, la section code_ini_ab prend ensuite le contrôle des opérations. Elle teste d'abord si le code est inférieur à clr, c'est-à-dire dans le domaine de l'alphabet ordinaire (0-255). Il s'agit alors d'un code concret qui est empilé à partir de l'étiquette @concret. Sinon il faut procéder à un décodage.

La pile précitée entre alors en jeu. Le décompresseur exploite l'alphabet compressé, qui possède une sorte de structure récursive. Le suffixe est toujours concret : à la compression, on y a disposé un véritable caractère. Par contre, le préfixe est souvent compressé et doit être décompressé par une entrée d'alphabet, elle-même constituée d'une partie comprimée et d'une partie qui ne l'est pas. Ce jeu se poursuit jusqu'à ce qu'une entrée d'alphabet soit constituée de deux parties concrètes. Jusque là, tous les suffixes ont été empilés car ils n'avaient pas encore été utilisés et ils peuvent donc être enregistrés dans la destination.

La boucle qui développe les codes se trouve à l'étiquette @fillstack_loop. C'est là qu'on applique scrupuleusement la méthode mentionnée. Le suffixe du code courant est empilé et le préfixe, s'il est comprimé, est soumis à la

décompression, jusqu'à ce qu'il devienne concret ce qui met fin à la boucle. Dans la section introduite par l'étiquette @concret, le dernier préfixe est encore mis sur la pile, car il pourrait encore servir à un éventuel cas particulier qui suivrait, et on vide la pile par une boucle (readstack_loop).

On tient compte des images plus grandes que 320x200 à l'endroit qui suit stosb. Le seul problème que posent ces images est qu'elles ne rentrent plus dans le segment de l'écran virtuel. Elles sont donc déchargées dans la mémoire d'écran et le pointeur associé est mis à jour pour la suite des opérations (nouveau déchargement ou écriture de la suite).

A partir de l'étiquette @noovl1, la routine se consacre à nouveau à la décompression proprement dite. La boucle est parcourue jusqu'à ce que toute la pile soit vidée. Ensuite, il faut actualiser l'alphabet en y reportant le dernier code comme préfixe et le code courant comme suffixe. Le pointeur sur le premier emplacement libre (free) est aussitôt mis à jour. S'il franchit la limite autorisée (max), l'alphabet est étendu, un nouveau maximum est calculé et les opérations se poursuivent. Si le nombre de bits atteint 12, on ne fait plus d'extension mais on retourne dans la boucle principale. Il est probable que peu de temps après sera émis le code clr qui est traité à l'étiquette @clear.

Les variables de gestion de l'alphabet sont réinitialisées et le caractère suivant qui est à coup sûr dans une forme non compressée est directement enregistré. Il faut ici aussi veiller à ne pas dépasser la capacité et prévoir un déchargement éventuel.

Pour terminer, il convient encore de procéder à la fermeture du fichier et d'indiquer que l'écran virtuel est rempli. Le programme appelant n'a plus rien d'autre à faire qu'à copier le reste de l'écran virtuel sur l'écran réel :

```
p13_2_ModeX(vram_pos, rest div 4);
```

Dans le format 320x200. vous pouvez simplifier en écrivant :

```
p13_2_ModeX(0,16000);
```

PCX simple et rapide

A côté de GIF, il existe de nombreux autres formats aux caractéristiques variées et liées à diverses méthodes de compression. L'un des plus connus est le format PCX développé par Zsoft pour son programme de dessin Paintbrush. Ce format n'est guère convaincant par son universalité ou son taux de compression. Mais il brille par sa simplicité. Les données sont disposées très clairement et l'algorithme de compression (RLE) n'est pas très compliqué.

PCX a connu plusieurs versions aux performances assez contrastées, nous nous intéresserons pour notre part à la version 3.0 qui supporte les images en 256 couleurs.

Indépendamment de sa version, un fichier PCX possède un en-tête de 128 octets qui contient divers paramètres comme la taille et la position de l'image. Dans la version 3.0, seuls 70 octets sont véritablement utilisés, le reste ne sert pas encore et est rempli de zéros.

La structure de l'en-tête est très simple :

Offset	Longueur	Contenu
0	1	Signature du format 0ah
1	1	Version (5 pour version 3.0)
2	1	Avec (1) ou sans (0) compression RLE
3	1	Nombre de bits par pixel
4	4	Coordonnées du coin supérieur gauche (x,y comme Words)
8	4	coordonnée du coin inférieur droit (x,y)
12	2	Résolution horizontale en dpi (la plupart du temps largeur de l'image)
14	2	Résolution horizontale en dpi (la plupart du temps hauteur de l'image)
16	48	Palette (16 couleurs)
64	1	réservé
65	1	Nombre de plans (un seul avec 256 couleurs)
66	2	Nombre d'octets par ligne d'image
68	2	Type de palette (1=couleur ou n/b
70	58	réservé (la plupart du temps 0

Après la signature du format et le numéro de version, on trouve un témoin de compression : s'il est à 0, aucune compression n'a été réalisée, ce qui conduit parfois à des fichiers ... plus petits comme nous le verrons plus loin. Si le témoin est à 1, l'image a été soumise à une compression RLE.

A la profondeur de couleur (8 bits pour 256 couleurs) succèdent les coordonnées de l'image et la résolution. Cette dernière est comprise dans un sens très large, car certains programmes l'interprètent directement comme la hauteur et la largeur de l'image. La palette qui suit est un héritage de l'époque des images en 16 couleurs et ne sert à rien si on travaille en 256 couleurs. Par souci de sécurité, il convient néanmoins d'y enregistrer les 16 premières couleurs de la palette.

A l'offset 65 se trouve le nombre de plans de bits exploités (1 pour 256 couleurs). On relève ensuite le nombre d'octets formant une ligne d'image, sous forme d'arrondi pair. Le mot qui suit (offset 68) indique si la palette contient des couleurs ou des niveaux de gris : il vaut 1 si l'image utilise des couleurs ou est en noir et blanc. Le reste est constitué d'octets de remplissage.

Les données proprement dites de la palette sont situées en fin de fichier pour les images en 256 couleurs. Elles sont introduites par l'octet Och, ce qui ajoute 769 octets à la suite des données graphiques.

Après l'en-tête viennent les données graphiques qui sont éventuellement compressées par la méthode RLE. Cette méthode transforme une succession de caractères semblables en une combinaison de 2 octets. Les octets isolés sont repris tels quels. Les répétitions sont introduites par un octet dont les deux bits supérieurs sont égaux à 1. Il s'agit alors d'un octet qui indique le nombre de répétitions, et qui est suivi du caractère à répéter.

De toute évidence, la méthode ne peut pas traiter convenablement les caractères isolés dont les deux premiers bits valent 1 : ils sont considérés comme des caractères répétés une fois. Le processus de compression conduit donc à leur dédoublement ce qui augmente la taille du fichier au lieu de la réduire. Si une image ne comporte pas beaucoup de zones monochromes, ce qui est souvent le cas des images en 256 couleurs, le fichier compressé est effectivement un peu plus long que son homologue naturel.

Le grand avantage de cette méthode de compression tient à sa vitesse de traitement. Les octets répétitifs sont beaucoup plus faciles à décoder que dans le format LZW associé aux fichiers GIF, qui nécessite un perpétuel tripatouillage de l'alphabet.

L'objet de ce chapitre est de présenter un screen grabber universel (qui copie l'écran dans un fichier) valable pour les deux formats en 320x200x256. Ce programme doit être en mesure de traiter des petites tâches amusantes comme le partage d'écran. Pour une fois, nous n'accorderons pas trop d'importance à la vitesse mais nous insisterons sur la clarté de la programmation. C'est ainsi que le compresseur est écrit en Pascal. Il s'agit de montrer les principes de traitement et d'affichage d'un fichier PCX.

Une application de ce programme pourrait consister par exemple à enregistrer quelques copies d'écran d'une démo pour les réafficher ensuite sous forme de générique final avec le nom des programmeurs, dessinateurs et musiciens.

Le code source du programme se trouve dans le fichier GRABBER.PAS :

```
{$m 1024,0,0}                    {pile réduite, pas de tas}
Uses ModeXLib,Crt,Dos;

Var OldInt9:Pointer;             {pointe sur l'ancien gestionnaire
                                  de clavier }
    actif:Boolean;               {vrai s'il y a déjà une copie
                                  d'écran en cours}
    nr:Word;                     {numéro de l'image, pour le nom du
                                  fichier}
    installe:Boolean;            {déjà installé ?}

    mode,                        {mode VGA actuel : 13h, ffh
                                  (mode X)}
                                 {ou 0 (aucun des deux)}
    Split_at,                    {ligne de fractionnement
                                  (ligne graphique)}
    LSA,                         {Linear Starting Address}
    Skip:Word;                   {Nombre d'octets à sauter }
 Procedure GetMode;
{détermine le mode graphique courant 13h ou mode X (Numéro 255)
 ainsi que ses paramètres ligne de fractionnement, adresse de
 départ)}
Begin
  mode:=$13;                     {mode 13h standard}
  asm                            {détermine le mode BIOS}
    mov ax,0f00h                 {fonction : infos vidéo}
    int 10h
    cmp al,13h                   {est-ce le mode BIOS 13h ?}
    je @Bios_ok
    mov mode,0                   {non -> ni mode 13h ni X actif}
  @bios_ok:
  End;
  If mode=0 Then Exit;           {mode erroné -> abandonner}

  Port[$3c4]:=4;                 {lit le registre 4 du TS Memory
                                  Mode}
  If Port[$3c5] and 8 = 0 Then   {Chain 4 (Bit 3) inactif ?}
```

```
    mode:=$ff;                      {alors mode X}

  Port[$3d4]:=$0d;                  {Linear Starting Address Low
                                    (CRTC 0dh)}
  LSA:=Port[$3d5];                  {à lire}
  Port[$3d4]:=$0c;                  {Linear Starting Address High
                                     (CRTC 0ch)}
  LSA:=LSA or Port[$3d5] shl 8;     {à lire et à inclure }

  Port[$3d4]:=$18;                  {Line Compare CRTC 18h}
  Split_at:=Port[$3d5];             {à lire}
  Port[$3d4]:=7;                    {Overflow Low}
  Split_at:=Split_at or             {extrait par masque le bit 4 et le
                                     décale en bit 8 }
    (Port[$3d5] and 16) shl 4;
  Port[$3d4]:=9;                    {Maximum Row Address}
  Split_at:=Split_at or             {extrait par masque le bit 6 et le
                                     décale en bit 9}
    (Port[$3d5] and 64) shl 3;
  Split_at:=Split_at shr 1;         {calcule en lignes d'écran }

  Port[$3d4]:=$13;                  {Row Offset (CRTC Register 13h)}
  Skip:=Port[$3d5];                 {à lire}
  Skip:=Skip*2-80                   {différence avec l'interligne
                                     "normal" }
End;

Procedure PCXShift;assembler;
{convertit la palette courante au format PCX (décalage de 2 bits
 vers la gauche )}
asm
  mov si,offset palette             {ds:si pointe sur la palette}
  mov cx,768                        { 768 octets à traiter }
@lp:  lodsb20 =                     {prend une valeur}
  shl al,2                          {la décale}
  mov ds:[si-1],al                  {la remet dans l'ancienne position }
  loop @lp                          {teste la fin de boucle }
End;
```

```
Var pcx:File;                          {Fichier PCX sur disque }

Procedure Hardcopy(Startadr,splt:Word;s : string);
{copie un graphique 320x200 (mode 13 ou X) dans un fichier PCX
 appelé s. Début de l'écran courant (Linear Starting Address) en
 Startadr. Ligne de fractionnement en splt}
Var Buf:Array[0..57] of Byte;      {mémorise des données avant
                                    enregistrement}
    Aux_Ofs:Word;
const
  Header1:Array[0..15] of Byte     {en-tête PCX, première partie }
    =($0a,5,1,8, 0,0, 0,0, $3f,1, 199,0,$40,1,200,0);
  Header2:Array[0..5] of Byte      {en-tête PCX, deuxième partie }
    =(0,1,$40,1,0,0);
  plan:Byte=0;                     {plan courant}

var count:Byte;                    {facteur de répétition }
    valeur ,                       {octet en cours de traitement }
    lastbyt:Byte;                  {octet précédent}
    i:word;                        {compteur }
begin
asm                                {lecture de la palette}
  xor al,al                        {commence par la couleur 0 }
  mov dx,3c7h                      {vers le DAC par Pixel Read Address}
  out dx,al

  push ds                          {es:di pointe sur la palette}
  pop es
  mov di,offset palette
  mov cx,768                       {768 octets à lire }
  mov dx,3c9h                      {Pixel Color Value}
  rep insb                         {on lit }

  cmp mode,13h                     {mode X ?}
  je @Linear{alors:}
  mov dx,03ceh                     {fixe le mode d'écriture et de
                                    lecture 0 }
```

```
  mov ax,4005h                    {par le registre 5 du GDC
                                   (GDC Mode)}
  out dx,ax
@Linear:End; = ign(pcx,s);        {ouvre le fichier en écriture }
  Rewrite(pcx,1);

  BlockWrite(pcx,Header1,16);     {enregistre la 1ère partie de
                                   l'en-tête }
  PCXShift; {prépare la palette}
  BlockWrite(pcx,palette,48);     {enregistre les 16 premières
                                   couleurs }
  BlockWrite(pcx,Header2,6);      {enregistre la 2ème partie de
                                   l'en-tête }
  FillChar(buf,58,0);             {58 zéros de remplissage }
  BlockWrite(pcx,buf,58);
  plan:=0;                        {commence par le plan 0 }
  count:=1;                       {initialise le facteur de
                                   répétition}
  If splt<200 Then
    If mode = $ff Then
      splt:=splt*80 Else          {calcule l'offset de fractionnement}
      splt:=splt*320 Else         {dépend du mode}
    splt:=$ffff;
  If mode=$13 Then                {LSA se réfère au modèle des plans!}
    Startadr:=Startadr*4;
  for i:=0 to 64000 do Begin      {traite chaque pixel }
  If i shr 2 < splt Then
  aux_ofs:=(i div 320) * skip     {fixe l'offset auxiliaire en tenant
                                   compte de la longueur des lignes }
  Else
  aux_ofs:=((i shr 2 - splt) div 320) * skip;
                                  {en cas de fractionnement }
asm                               {lit un pixel}
  mov ax,0a000h                   {d'abord le segment}
  mov es,ax
  mov si,i                        {puis l'offset}
  cmp mode,13h                    {mode 13h ?}
  je @Linear1
```

```
  shr si,2                          {non, calcul l'offset }
@Linear1:  cmp = ,splt              {ligne de fractionnement atteinte ?}
  jb @suite                         {non on continue}
  sub si,splt                       {sinon référence au début }
  sub si,startadr                   {de l'écran }
@suite:  add s = tartadr            {adresse de début }
  add si,aux_ofs                    { + offset auxiliaire }

  cmp mode,13h                      {mode 13h ?}
  je @Linear2                       {non, lecture en mode X }
  mov dx,03ceh                      {active le registre 4 du GDC
                                     (Read Plane Select)}
  mov ah,plan                       {sélectionne le plan courant }
  inc plan                          {et passe au suivant }
  mov al,4
  and ah,03h
  out dx,ax
@Linear2:  mov = ,es:[si]           {lit un octet}
  mov valeur,al                     {le met dans la variable valeur }
End;
  If i<>0 Then Begin                {pas de compression pour le premier
                                     octet }
  If (Valeur = lastbyt) Then Begin{ octets identiques?}
    Inc(Count);                     {incrémente le facteur de
                                     répétition }
    If (Count=64) or                {facteur trop grand ? }
     (i mod 320 =0)  Then Begin     {ou début de ligne?}
      buf[0]:=$c0 or (count-1);     {alors on stocke }
      buf[1]:=lastbyt;              {le facteur et la valeur de l'octet}
      count:=1;                     {réinitialise le facteur de
                                     répétition }
      BlockWrite(pcx,buf,2);        {enregistre le tout sur disque }
    End;
  End Else                          {octets divers :}
    If (Count > 1) or               {plusieurs octets identiques ?}
   (lastbyt and $c0 <> 0) Then {octet trop grand ? }
    Begin
```

```
      buf[0]:=$c0 or count;      {stocke le facteur et l'octet dans
                                  le fichier}
      buf[1]:=lastbyt;
      lastbyt:=Valeur;           {sauve l'octet courant pour la
                                  suite de la compression }
      Count:=1;                  {et réinitialise }
      BlockWrite(pcx,buf,2);
    End Else Begin               {octet isolé à ... }
      buf[0]:=lastbyt;           {... enregistrer directement }
      lastbyt:=Valeur;           {sauve l'octet courant pour la
                                  suite de la compression }
      BlockWrite(pcx,buf,1);
    End;

  End Else lastbyt:=valeur;      {premier octet juste à sauver }
End;
  buf[0]:=$0c;                   {enregistre la signature de la
                                  palette }
  blockwrite(pcx,buf[0],1);
  blockwrite(pcx,palette,256*3); {puis la palette }
  Close(pcx);                    {ferme le fichier }
End;

 Procedure Action;
{appelé lorsqu'on appuie sur la touche de déclenchement (Hot-Key) }
Var nrs:String;                  {chaîne pour le nom }
Begin
  if not actif Then Begin        {pas encore chargé }
    actif:=true;                 {maintenant actif }
    str(nr,nrs);                 {convertit le numéro en chaîne et
                                  l'incrémente }
    Inc(nr);
    GetMode;                     {consulte le mode graphique, etc}
    If mode <> 0 Then
     HardCopy(LSA,Split_at,'hard'+nrs+'.pcx');
                                 {exécute la copie d'écran }
    actif:=false;                {activité terminée }
  End;
```

```
End;

Procedure Handler9;interrupt;assembler;
{nouveau gestionnaire d'interruption de l'IRQ clavier }
asm
   pushf
   call [oldint9]                {appelle l'ancien gestionnaire de
                                  l'IRQ 1 }

   cli                           {inhibe toute nouvelle interruption}
   in al,60h                     {lit le scan code }
   cmp al,34d                    {G ?}
   jne @fini                     {non -> c'est terminé }
   xor ax,ax                     {charge le segment 0}
   mov es,ax
   mov al,es:[417h]              {lit l'état du clavier }
   test al,8                     {Bit 8 à 1 (touche Alt) ?}
   je @fini{non -> c'est terminé}
   call action                   {exécute la copie d'écran }
@fini:  sti =                    {autorise à nouveau les
                                  interruptions}
End;

Procedure signature;assembler;
{procédure fantôme, contient un avis de copyright pour le test
 d'installation }
{ CODE NON EXECUTABLE !}
asm
  db 'Screen-Grabber, (c) Micro Application 1994';
End;

Procedure Check_Inst;assembler;
{teste si le grabber est déjà installé }
asm
  mov installe,1                 {a priori: oui }
  push ds                        {ds va resservir!}
  les di,oldint9                 {charge un pointeur sur l'ancien
                                  gestionnaire }
```

```
  mov di,offset signature        {la procedure signature est dans le
                                  même segment }
  mov ax,cs                      {ds:si va pointer sur la signature
                                  du programme }
  mov ds,ax
  mov si,offset signature
  mov cx,20                      {compare 20 caractères }
  repe cmpsb
  pop ds                         {restaure ds }
  jcxz @installe                 {égalité -> déjà installé}
  mov installe,0                 {pas encore installé }
@installe:End; = Begin
  nr:=0;                         {premier nom de fichier : hard0.pcx}
  GetIntVec(9,OldInt9);          {lit l'ancien vecteur
                                  d'interruption }
  Check_Inst;                    {teste si déjà installé }
  If not installe Then Begin     {si ce n'est pas le cas :}
    SetIntVec(9,@Handler9);      {installe le nouveau gestionnaire }
    WriteLn('Grabber installé');
    WriteLn('(c) Micro Application 1994');
    WriteLn('Déclenchement par <Alt> g');
    Keep(0);{affichage et mise en résidence }
  End;
  WriteLn('Grabber déjà installé ');
      {on s'en va }
End.
```

Le programme principal présente les caractéristiques typiques d'un programme résident même si les limites de Pascal obligent à faire l'impasse sur la désinstallation. On teste si le programme n'est pas déjà installé, et s'il s'avère que ce n'est pas le cas, le vecteur d'interruption du clavier est détourné, un message est affiché et l'exécution est abandonnée. Sinon l'utilisateur est averti qu'il existe déjà un exemplaire du programme en mémoire centrale. Le test se fait par une procédure Check_inst qui effectue la comparaison d'une partie du programme chargé avec le contenu du gestionnaire de clavier actuel.

La séquence qui est introduite dans l'interruption du clavier (IRQ 1, donc interruption 9) a pour tâche de restituer à l'ancien gestionnaire les touches frappées pour assurer un fonctionnement normal du système, tout en détectant la combinaison **ALT-G** qui doit déclencher le grabber (Hotkey ou touche de déclenchement). Chaque scan code est comparé à celui de la touche **G**. En

cas d'identité, l'indicateur d'état du clavier qui se trouve à l'adresse 00417h permet de savoir si la touche ALT a été pressée. Si tel est le cas, on lance la procédure action.

Cette dernière s'assure d'abord que pendant l'enregistrement d'une image, la touche de déclenchement ne sera pas actionnée une seconde fois ce qui conduirait à des complications. Le nom du fichier est fixé à partir d'un numéro courant, pour le cas où on procéderait à toute une série de copies d'écran. Mais avant d'appeler la procédure HardCopy, il faut déterminer le mode graphique courant, car le grabber est prévu pour 320x200 pixels.

C'est la procédure GetMode qui prend en charge cet aspect du problème. Elle recherche d'abord le mode Bios qui est 13h dans les deux cas. Si le Bios fournit une autre valeur, la copie d'écran ne peut se faire et la procédure se termine avec une valeur de mode égale à 0. Le bit Chain4 du registre 4 du Timing Sequencer permet de faire la distinction entre mode 13h et mode X.

Pour que le grabber fournisse des images acceptables dans toutes les situations d'effets, il faut encore lire quelques registres du CRTC. D'abord la Linear Starting Adress, utilisée pour le défilement, ensuite l'indicateur Line Compare pour la ligne de fractionnement (cet indicateur est réparti sur trois registres CRTC qu'il faut réunir), et ensuite le Row Offset qui est utilisé pour des résolutions virtuelles plus élevées (640x400). Grâce au paramètre Row Offset, on peut calculer la distance skip qui sépare deux lignes. Normalement, en résolution 320x200, cet intervalle est nul ; en résolution virtuelle 640x400, il vaut 80.

La procédure Action déclenche alors la copie proprement dite qui reçoit comme arguments les paramètres du CRTC précédemment déterminés ainsi que le nom du fichier. La palette courante est lue dans le circuit DAC et si le mode X est actif, on enclenche le mode de lecture 0. La palette est ramenée à un format de 8 bits par la procédure PCXShift (le traitement est inversé par rapport à la lecture des fichiers GIF : chaque nombre est décalé de deux bits vers la gauche).

L'en-tête soigneusement préparé est maintenant enregistré dans le fichier avec les 16 premiers éléments de la palette et rempli jusqu'à 128 octets. Quelques variables supplémentaires font l'objet d'une évaluation : reçu comme ligne split est exprimé en offset car la boucle de copie travaille en orientation offset. En mode 13h, l'adresse de début doit être multipliée par 4 pour respecter la structure (Chain4) de ce mode.

Dans la boucle principale (i), un autre offset est encore calculé (aux_ofs) qui servira à la gestion du défilement des images excessivement larges. Si l'offset courant se trouve avant la ligne de fractionnement, le nombre de lignes traité est simplement multiplié par l'interligne. Après la ligne de fractionnement,

cette dernière doit être prise en compte pour avoir une référence sur le début de la mémoire.

Le bloc asm qui suit sert à lire un octet dans la mémoire d'écran. Le couple de registres es:si est chargé avec un pointeur sur le pixel courant. En mode X, comme d'habitude, l'offset est divisé par 4.

Si la ligne de fractionnement est déjà dépassée, la référence au début de la mémoire d'écran est rétablie pour ajustement.

En mode X, le plan courant est déterminé et fixé. Le pixel est amené dans la variable valeur pour être comprimé par la suite.

Le premier n'est pratiquement lu que pour être traité à l'itération suivante comme "Last byte". Chaque octet supplémentaire est comparé au précédent et en cas de coïncidence, le compteur count est incrémenté. Quand ce dernier atteint la valeur 63, facteur de répétition maximal, il faut prendre des mesures spéciales. Le compteur est enregistré dans le fichier et le comptage reprend à zéro. Les fins de ligne posent aussi un problème spécial. A l'origine, PCX fonctionnait ligne par ligne, ce qui se répercutait sur la compression. Les répétitions de caractères ne pouvaient pas franchir une fin de ligne. Comme il existe des programmes de dessin anciens qui conservent ce handicap, la compression doit en tenir compte.

Si le caractère n'est pas identique à son prédécesseur, l'alternative Else prend le contrôle. Elle teste d'abord si une répétition était en cours (count>1) ou si le caractère exploite les deux bits supérieurs. Dans les deux cas, un caractère comprimé avec facteur de répétition et octet à répéter est à enregistrer (le facteur de répétition est de 1 dans le deuxième cas). Lastbyt est remis à jour pour permettre une compression à l'itération suivante. Si count vaut 1 et si le dernier octet est un octet ordinaire (bits supérieurs égaux à 0), l'octet est directement écrit dans le fichier, sans compression, et Lastbyt est réactualisé.

Quand l'intégralité de l'image a été enregistrée dans le fichier, il reste à ajouter la palette (avec la marque 0ch) et à clôturer le fichier. Le gestionnaire est alors en fin d'exécution.

Comme nous l'avons déjà dit, ce programme n'est pas optimisé pour être le plus rapide possible : pour un screen grabber, ce n'est pas très important. Nous avons voulu montrer la gestion d'un fichier PCX au moyen d'un exemple impliquant un enregistrement, et non simplement une lecture.

Quand il est appelé, le programme s'installe dans la mémoire comme programme résident, attendant patiemment une frappe de la touche de déclenchement. Il ne peut pas être retiré autrement qu'en réinitialisant l'ordinateur. Un programme résident muni d'une faculté de désinstallation ne peut être réalisé confortablement qu'en assembleur.

Cependant la version en Pascal évite bien la réinstallation accidentelle comme le montre la procédure Check_Inst.

4.4. VGA pressé jusqu'au dernier bit

Dans la suite de cet ouvrage, vous allez encore apprendre de nombreux effets graphiques qui reposent pour la plupart sur des manipulations de registres. Ces registres offrent une gamme infinie de possibilités qui influencent profondément les caractéristiques les plus intimes de VGA. Beaucoup de registres paraissent au premier abord inutiles mais, en les explorant, on parvient presque toujours à en tirer des effets intéressants.

Pour faciliter la compréhension des manipulations décrites plus loin, nous commencerons par étudier l'intégralité des registres VGA du premier jusqu'au dernier bit.

En principe, vous pourrez jouer librement avec ces registres en les testant comme vous l'entendez. Seuls les registres de timing (Registres 0-7 du CRTC) sont à manier avec précaution car si par l'intermédiaire de leurs circuits vous imposez des paramètres néfastes à votre moniteur, il va se mettre à siffler horriblement et risque d'être endommagé.

Quelques méchants personnages peuvent exploiter cette faculté de nuire en détraquant votre moniteur. Soyez donc prudent et dès que le moniteur commence à siffler bruyamment, éteignez-le immédiatement. Normalement, il faut plusieurs secondes pour que le moniteur soit réellement coupé de l'extérieur. Pendant les expériences, il est recommandé de garder un doigt sur l'interrupteur d'arrêt.

Les autres registres sont moins dangereux : au pire, ils peuvent conduire au plantage du système. On peut les classer en deux catégories. Les registres isolés peuvent être adressés par une adresse de port spéciale. Les registres indexés ne possèdent pas d'adresse propre. Ils font partie de l'un des circuits de l'ensemble VGA et sont sélectionnés par une adresse fixe, et traités par une autre.

Miscellaneous Output Register
Lecture : Port 3cch, écriture : 3c2h

Bit	Signification	Accès
7	polarité du retour de balayage vertical	RW
6	polarité horizontale (bit 7 = résolution vert.)	RW
5	Sélection de page pour adressage pair/impair	RW
4	réservé	
3, 2	sélection d'horloge (résolution horiz.)	RW
1	Enable Ram, 1 = accès RAM par CPU possible	RW
0	Adresse E/S, 1 = monochrome (3bxh), 0 = couleur (3dxh)	RW

Explication

La résolution verticale physique est fixée par la polarité des deux signaux de retour de balayage : 00 est réservé, 01 signifie 350 lignes, 10 400 lignes de pixels (également 320x200, voir registre 9 du CRTC), 11 480 lignes.

Bit 5 : Donne en mode pair/impair le bit 0. Dans chaque plan, on traite soit les adresses paires (bit 5=0) ou impaires (bit 5=1), voir registre 4 du TS.

Bits 3, 2 : Définissent par la fréquence des pixels la résolution horizontale. 00 signifie 640 pixels horizontaux, 01 720 pixels, 10 800 pixels, 11b étant réservé.

Bit 1 : Régit l'accès à la mémoire VGA de la part du processeur (0= pas d'accès possible).

Bit 0 : Indique à quelle adresse se trouvent le CRTC (3x4h/3x5h) et le registre 1 de l'Input Status (3xah). Si le bit contient 1, "x" est à remplacer par "d", s'il est à 0, "x" est à remplacer par "b".

Input Status Register 0
Lecture : Port 3c2h

Bit	Signification	Accès
7	Interruption CRT	RO
6-0	spécifiques au fabricant, la plupart du temps Feature Code etc.	RO

Explication

bit 7 : signale un retour de balayage vertical, à condition que l'interruption correspondante ait été activée (la plupart du temps par micro-interrupteur DIP sur la carte). Effacé par le registre 4 du CRTC.

```
Input Status Register 1
Port 3dah (couleur) ou 3bah (monochrome), selon
le bit 0 du registre Miscellaneous Output
```

Bit	Signification	Accès
7	6	réservé
5	4	bits de test, dépendant de la carte
3	Retour de balayage vertical	RO
2	1	réservé
0	Display enable complement	RO

Explication

Bit 3 : S'il est égal à 1, un retour de balayage (retrace) vertical est en cours. Ce bit permet donc de synchroniser le dessin de l'écran pour éviter le scintillement dû à l'arrivée inopportune du retour de balayage (cf 4.1 le vocabulaire incontournable)

Bit 0 : C'est un signal d'autorisation d'affichage inversé qui indique, s'il est égal à 1, qu'un retour de balayage horizontal ou vertical est en cours. On peut exploiter ce bit pour connaître la ligne en cours d'affichage. A la suite d'un retour de balayage vertical (bit 3), on peut détecter une nouvelle ligne s'il passe de 1 à 0. Les célèbres barres de Copper (chapitre 6.8) constituent une application de ce principe.

Selon le fabricant, il existe encore différents autres registres mais ils ne sont pas normalisés et reflètent des particularités d'une carte spécifique (par exemple Feature Connector). Ils ne sont donc pas à prendre en considération pour une programmation de portée générale.

Le contrôleur du tube de rayons cathodiques (Cathod Ray Tube Controller ou CRTC)

Le circuit VGA qui possède le plus de registres est le contrôleur du tube de rayons cathodiques (CRTC pour Cathod Ray Tube Controller) qui est responsable de la génération du signal vidéo. Le CRTC est très librement programmable et le rayon cathodique peut être manipulé avec beaucoup de liberté, ce qui peut se révéler dangereux pour votre moniteur si vous ne manifestez pas un minimum de prudence. C'est d'ailleurs pour cette raison que les registres sont protégés par un bit de protection (registre 11 du CRTC) qui empêche toute écriture accidentelle. Mais le CRTC dispose aussi de beaucoup d'autres registres qui n'ont rien à voir avec le timing du rayon (par exemple Linear Starting Address), et qui peuvent donc être manipulés sans crainte.

On entre en contact avec ces registres au moyen de deux adresses : 3d4h sert d'index et 3d5h est destiné aux données (bit 0 du registre Miscellaneous Output à 0). Les adresses deviennent 3b4h et 3b5h. On atteint un registre donné en écrivant d'abord son numéro dans le registre d'index et en accédant ensuite, en écriture ou en lecture, au registre de données qui correspond alors au registre CRTC recherché. Une fois l'index fixé, il reste valable et les accès peuvent se succéder autant de fois qu'on le désire. Pour des raisons d'efficacité, on peut effectuer un accès unique en écriture en émettant un mot (word) sur le port d'index : l'octet de poids faible doit alors contenir le numéro du registre et l'octet de poids fort la valeur à donner au registre.

Registre 0 du CRTC : Horizontal Total

Bit	Signification	Accès
7-0	Nombre de caractères par ligne de balayage (-5 en VGA)	RW

Bits 7-0 : Ils donnent la taille de la ligne en nombre de caractères (Character Times Units). Une unité de ce type correspond soit à 8 pixels (registre 1 du TS : bit 0=1, par exemple mode 320x200) ou 9 pixels (registre 1 du TS : bit 0=0, par exemple mode texte 3). La valeur effective de ce registre doit être diminuée de 5 (2 en mode EGA).

Registre 1 du CRTC : Horizontal Display End

Bit	Signification	Accès
7-0	Fin du signal Display enable en caractères	RW

Bits 7-0 : Ils indiquent en pratique le nombre de caractères visibles (8 ou 9 pixels, voir registre 0).

Registre 2 du CRTC : Horizontal Blank Start

Bit	Signification	Accès
7-0	Début de la période de blanc horizontal	RW

Bits 7-0 : Ils définissent la position à partir de laquelle le CRTC désactive le rayon cathodique lors de l'affichage d'une ligne. La période de blanc contient la période de retour de balayage et crée un cadre noir à droite et à gauche de l'écran.

Register 3 du CRTC : Horizontal Blank End

Bit	Signification	Accès
7	bit de test (exploitation normale : 1)	RW
6-5	Display Enable Skew (délai)	RW
4-0	Horizontal Blank End	RW

Bits 6-5 : Ces bits donnent le nombre de caractères que le CRTC scrute à l'avance dans la mémoire (le signal Display Enable est retardé en conséquence) pour être toujours prêt à fournir le prochain caractère. En VGA, ils valent normalement 0.

Bits 4-0 : Les 5 bits inférieurs du nombre à 6 bits qui représente la fin de la période de blanc se retrouvent ici. Le bit 5 de ce nombre se trouve dans le bit 7 du registre 5 du CRTC.

Registre 4 du CRTC : Horizontal Sync Start

Bit	Signification	Accès
7-0	Position de début du retour de balayage horizontal	RW

Bit 7-0 : Ce registre donne la position (en caractères) où va débuter le retour de balayage horizontal.

Registre 5 du CRTC : Horizontal Sync End

Bit	Signification	Accès
7	Bit 5 du signal Horizontal Blank End (registre 3)	RW
6-5	Horizontal Sync Skew (délai)	RW
4-0	Fin du retour de balayage horizontal	RW

Un retour de balayage horizontal peut aussi être suivi d'un délai : voir registre 3, bits 6-5, le plus souvent 0 en VGA.

Bits 4-0 : définissent la fin du retour de balayage horizontal, relativement au début. La première fois que les 5 bits inférieurs du compteur de caractères interne sont identiques au contenu de ce registre, le retour de balayage est achevé.

Registre 6 du CRTC : Vertical Total

Bit	Signification	Accès
7-0	Nombre total de lignes balayées par image -2 (Bits 7-0)	RW

Explication

Bits 7-0 : Donnent la hauteur totale de l'image en lignes de balayage (diminuée de 2, de 1 en EGA) C'est un nombre de 10 bits (11 en Super VGA). Les bits 9-8 se trouvent dans le registre d'Overflow (registre 7).

Registre 7 du CRTC : Overflow

Bit	Signification	Accès
7	Vertical Sync Start (bit 9)	RW
6	Vertical Display Enable End (bit 9)	RW
5	Vertical Total (bit 9)	RW
4	Line Compare (Split Screen) (bit 8)	RW
3	Vertical Blank Start (bit 8)	RW
2	Vertical Sync Start (bit 8)	RW
1	Vertical Display Enable End (bit 8)	RW
0	Vertical Total (bit 8)	RW

Explication

Bits 7-0 : Ce registre contient les bits 8 et 9 de la plupart des paramètres verticaux qui ne trouvent plus de place dans les registres d'origine.

Registre 8 du CRTC : Initial Row Address

Bit	Signification	Accès
7	réservé	
6-5	Byte Panning	RW
4-0	Initial Row Address	RW

Explication

Bits 6-5 : Ce registre permet de décaler le contenu de l'écran vers la gauche. Le décalage peut aller jusqu'à 3 octets (en mode X 4 pixels) mais il peut se faire avec plus de liberté en maniant l'adresse de début linéaire (Linear Starting Address, registre 0ch/0dh).

Bits 4-0 : Indiquent au CRTC la ligne de déclenchement du retour de balayage vertical, normalement la ligne 0. Si on augmente ce paramètre, le CRTC commence par une ligne située plus bas, ce qui déplace le contenu de l'écran vers le haut. Ce registre fonctionne de la même façon en mode texte, de sorte que grâce à lui on peut réaliser un défilement continu vertical (smooth scrolling, voir chapitre 6.6).

Une autre application concerne le mode 320x200. Pour ralentir sensiblement le défilement vertical, tout en conservant 70 décalages par seconde pour éviter des tressautements, on procède de la façon suivante : l'adresse de début n'est incrémentée que tous les deux balayages pour diminuer la vitesse. Le défilement fin est ensuite effectué à l'aide de ce registre en le faisant osciller entre les valeurs 0 et 1. A chaque fois, le défilement se fait sur une ligne de balayage (= une demi ligne d'image en mode 320x200).

Registre 9 du CRTC : Maximum Row Address

Bit	Signification	Accès
7	Double Scan (dédoublement des lignes)	RW
6	Line Compare (Split screen, registre 18h) bit 9	RW
5	Vertical Blank Start bit 9	RW
4-0	Nombre de lignes de balayage par ligne de caractères -1	RW

Explication

Bit 7 : Si on met ce bit à 1, le rythme d'horloge vertical (clock rate) est divisé par deux, ce qui dédouble l'affichage de chaque ligne. Prévu pour la génération des modes 200 lignes dans une résolution physique de 400 lignes. La plupart des Bios atteignent ce résultat en mettant à 1 le bit 0, voir à cet endroit.

Bits 4-0 : Donne la hauteur des caractères en mode texte, diminuée de 1 (15 dans le mode 3 de VGA, 9 x 16 pixels par caractères). Egalement utilisé en mode graphique pour diminuer la résolution verticale (affichage multiple de chaque ligne si les bits 4-0 >0), voir chapitre 6.7 : (une autre manière de nettoyer l'écran : l'image qui s'écoule)

Registre 0ah du CRTC : Ligne de début du curseur

Bit	Signification	Accès
7-6	réservé	
5	curseur activé (0) / désactivé (1)	RW
4-0	ligne de début	RW

Bit 4-0 : ligne de balayage à l'intérieur du caractère où commence l'affichage du curseur

Registre 0bh : Ligne de fin du curseur

Bit	Signification	Accès
7	réservé	
6-5	Cursor skew	RW
4-0	ligne de fin	RW

Explication

Bits 6-5 : L'affichage du curseur peut également donner lieu à un retardement pour garantir l'affichage à l'extrémité gauche ou droite de l'écran, en VGA vaut la plupart du temps 0.

Bits 4-0 : Ligne de balayage à l'intérieur du caractère correspondant à la fin de l'affichage du curseur.

Registre 0ch du CRTC : Linear Starting Address High		
Bit	**Signification**	**Accès**
7-0	Linear Starting Address bits 15-8	RW

Explication

Bits 7-0 : Ce registre contient les bits 15-8 de l'adresse de début à 16 bits. Cette adresse indique l'offset à l'intérieur de la mémoire d'écran, où le CRTC commence à lire les données graphiques. En la manipulant, on peut provoquer un défilement horizontal et vertical de l'écran (voir chapitre 6.3 Défilement tout azimut). En mode texte, le défilement se fait cependant caractère par caractère, le paramétrage fin se faisant par le registre 13h de l'ATC (Horizontal Pixel Panning) et le registre 8 du CRTC (Initial Row Address), voir chapitre 6.6 Défilement continu en mode texte. Avant d'écrire dans ce registre, il faut impérativement diviser par deux l'adresse réelle en mode pair/impair et par quatre en mode Chain 4 (par exemple 13h).

Registre 0dh du CRTC : Linear Starting Address Low		
Bit	**Signification**	**Accès**
7-0	Linear Starting Address bits 7-0	RW

Explication

Bits 7-0 : Donnent le mot de poids inférieur de l'adresse de début de l'écran, voir registre 0dh.

Registre 0eh du CRTC : Cursor Address High		
Bit	**Signification**	**Accès**
7-0	Position du curseur comme offset (bits 15-8)	RW

Explication

Bit 7-0 : Indiquent la position courante du curseur dans la mémoire d'écran.

Registre 0fh du CRTC : Cursor Address Low		
Bit	**Signification**	**Accès**
7-0	Position du cursor comme offset (bits 7-0)	RW

Explication

Bit 7-0 : Mot de poids faible de l'adresse du curseur, voir registre 0eh.

Registre 10h du CRTC : Vertical Sync Start		
Bit	**Signification**	**Accès**
7-0	Ligne de déclenchement du balayage de retour vertical	RW

Explication

Bit 7-0 : Contiennent les bits 7-0 de la ligne de balayage (nombre à 10 bits) où se déclenche le retour vertical. Les bits 9 et 8 se trouvent dans le registre d'Overflow.

Registre 11h du CRTC : Vertical Sync End		
Bit	**Signification**	**Accès**
7	Protection Bit	RW
6	réservé	
5	Activation de l'interruption de retour de balayage vertical (0)	RW
4	Remise à zéro de l'interruption de retour de balayage vertical (0)	RW
3-0	Ligne où s'arrête le retour de balayage vertical	RW

Explication

Bit 7 : Le bit de protection, s'il est égal à 1, protège contre l'écriture les registres 0-7 du CRTC à l'exception du bit 4 du registre d'overflow (registre 7). Presque tous les Bios mettent à 1 ce bit de sorte qu'il doit être annulé avant les manipulations de timing.

Bit 5 : Si ce bit est à 0 et l'interruption précédente réinitialisée par effacement du bit 4, le CRTC déclenche un IRQ 2 au prochain retour de balayage vertical. Mais la plupart des cartes VGA ne génèrent pas cette interruption, soit parce qu'elles ne sont simplement pas en mesure de le faire soit parce que le fabricant configure ainsi les micro-interrupteurs DIP pour des raisons de compatibilité.

Bit 4 : A l'issue de chaque interruption, ce bit doit être remis à 0, sans quoi aucune autre interruption ne peut se déclencher (voir bit 5).

Bits 3-0 : La fin du retour de balayage est également déterminée par rapport à son début, étant donné que seuls 4 bits sont utilisés à cet effet. La première fois que les quatre bits inférieurs du compteur de lignes internes sont identiques à ce nombre, un retour de balayage en cours doit être considéré comme achevé.

Registre 12h du CRTC : Vertical Display End

Bit	Signification	Accès
7-0	dernière ligne de balayage affichée (bits 7-0)	RW

Explication

Bits 7-0 : A l'issue de la ligne correspondant à ce numéro, le CRTC coupe le rayon cathodique. Les bits 9-8 de ce nombre se trouvent dans le registre d'overflow

Registre 13h du CRTC : Row Offset

Bit	Signification	Accès
7-0	Offset entre deux lignes d'écran	RW

Explication

Bits 7-0 : Définissent la distance entre deux lignes dans la mémoire, autrement dit leur longueur. L'unité de référence dépend du type d'adressage courant: en mode double word, on compte en blocs de 8 octets, par exemple dans le mode 13h si le registre contient 40 la largeur effective est de 40*8=320 octets. En mode adressage word, l'unité est de 4 octets (par exemple mode texte 3, contenu du registre 40, donne 40*4=160 octets). En mode byte, l'unité est de 2 octets, (par exemple mode X, contenu du registre=40, donne 40*2=80 octets). Ce registre permet également d'exécuter un défilement horizontal. On lui donne par exemple la valeur 80. La distance entre les lignes est alors dédoublée et "à côté " de l'extrait visible on obtient des zones invisibles de la mémoire d'écran. Si on déplace le début de l'écran de quelques octets, on peut effectuer un défilement horizontal de l'écran virtuel (voir chapitre 6.3, Défilement tout azimut).

Registre 14h du CRTC : Underline Location

Bit	Signification	Accès
7	réservé	
6	Doubleword Addressing	RW
5	Linear Adress Count by 4	RW
4-0	Ligne de début du soulignement	

Bit 6 : Enclenche le mode doubleword si on le met à 1. Dans ce mode, l'adresse courante subit une rotation de deux bits vers la gauche, avant d'être envoyée à la mémoire d'écran pour accès en lecture. De ce fait, la plupart du temps, ce sont d'abord les offsets 0, 4, 8 puis 1, 5, 9 qui sont lus en mémoire. Ce fonctionnement a par exemple lieu en mode 13h.

Bit 5 : Si ce bit vaut 1, la fréquence de lecture dès lors des accès à la mémoire d'écran est divisée par 4. Employé le plus souvent avec le mode doubleword.

Bits 4-0 : En mode monochrome, ces bits déterminent la ligne de balayage de l'ACT qui exécute le soulignement. Si on diminue cette valeur, les caractères peuvent se trouver barrés ou surlignés. Si on dépasse la hauteur d'un caractère, le soulignement est désactivé.

Registre 15h du CRTC : Vertical Blank Start

Bit	Signification	Accès
7-0	Vertical Blank Start bits 7-0	RW

Explication

Bits 7-0 : Ce registre indique à partir de quelle position verticale le CRTC doit désactiver le rayon cathodique. C'est à l'intérieur de cette période de blanc que le retour de balayage vertical va être déclenché. Les bits 8 et 9 de ce registre se trouvent dans le registre d'overflow.

Registre 16h du CRTC : Vertical Blank End

Bit	Signification	Accès
7-0	Ligne de fin de blanc vertical	RW

Explication

Bits 7-0 : Ce registre constitue une référence relative au début du blanc, étant donné que seuls 8 bits sont utilisés. La première fois que les huit bits inférieurs du compteur de lignes internes correspondent au contenu de ce registre la mise à blanc est terminée.

Registre 17h du CRTC : CRTC Mode

Bit	Signification	Accès
7	Hold Control	RW
6	Word Mode (0) / Byte Mode (1)	RW
5	autre emplacement pour le lit d'adresse 0	RW
4	réservé	
3	Linear Adress count by 2	RW
2	Line Counter count by 2	
1	autre emplacement pour le lit d'adresse 14	RW
0		

Explication

Bit 7 : Une valeur de 0 arrête l'ensemble du timing horizontal et vertical.

Bit 6 : S'il vaut 0, le mode word est actif, sinon c'est le mode byte. Si le bit 6 du registre 14h est à 1 (mode doubleword), ce bit ne joue plus aucun rôle.

Bit 5 : S'il est nul, le bit 15 n'est pas transféré dans le bit 13 en direction du bit 0 (rotation) : ce procédé permettait aux cartes EGA munies de 64 Ko de ne pas déborder en mode word.

Bit 3 : Mis à 1, il divise par deux la fréquence d'accès aux caractères dans la mémoire d'écran. Rôle semblable à celui du bit 5 du registre 14h.

Bit 2 : Si ce bit vaut 1, le compteur de lignes n'est incrémenté que toutes les deux lignes, ce qui dédouble le timing vertical.

Bits 1, 0 : Si on les met à 0, ces deux bits reprogramment les lignes d'adresse 14 et 13 : le bit 0 est reproduit en bit 13, le bit 1 en bit 14 (uniquement si le bit 1 de ce registre vaut 0). En conséquence, les adresses impaires sont lues 8Ko après les paires, les deux bits inférieurs déterminant un bloc. On peut ainsi émuler le contrôleur 6845 qui adresse la mémoire d'écran de cette manière. En CGA, le bit 0 est mis à 0, et de même le bit 1 en émulation Hercules.

Registre 18h du CRTC : Line Compare (Split Screen)

Bit	Signification	Accès
7-0	Line Compare bits 7-0	RW

Explication

Bits 7-0 : Indiquent la ligne de balayage physique où le CRTC recommence à tirer ses données du début de la mémoire d'écran. On peut ainsi réaliser un écran partagé en deux : dans la partie supérieure s'affiche le domaine de la mémoire fixé par la Linear Starting Address, dans la partie inférieure on fixe le début de la mémoire. Voir à cet effet le chapitre 6.2 consacré au partage d'écran. Le bit 8 de ce registre se trouve dans le registre d'overflow, le bit 9 dans le registre Maximum Row Address.

Timing Sequencer (TS)

Le séquenceur de timing (Timing sequencer) joue avant tout un rôle dans la gestion de la mémoire. En fonction de la configuration en cours, les accès sont dirigés vers des plans donnés et combinés avec le jeu de caractères courant. Le TS est aussi responsable du rafraîchissement de la mémoire d'écran.

Le TS est adressé comme le CRTC par l'intermédiaire d'un registre d'index (adresse 3c4h) et d'un registre de données (adresse 3c5h). On peut écrire une donnée par un seul ordre Out Word.

Registre 0 du TS : Reset synchrone

Bit	Signification	Accès
7-2	réservé	
1	Reset synchrone	RW
0	Reset asynchrone	RW

Explication

Bit 1 : Si on le met à 0, le TS déclenche une réinitialisation synchrone, c'est-à-dire qu'il réinitialise tous les registres et les déconnecte, jusqu'à ce que les bits 0 et 1 soient à nouveau mis à 1. Pendant ce temps, le rafraîchissement de la mémoire est également désactivé, il faudra donc mettre fin au Reset le plus rapidement possible pour éviter toute perte de données. La réinitialisation est à lancer chaque fois qu'on souhaite modifier le registre du TS.

Bit 0 : La mise à 0 déclenche un réinitialisation asynchrone. Les caractéristiques d'une réinitialisation asynchrone sont semblables à celles d'une réinitialisation synchrone sauf que le registre de sélection de fonte n'est pas effacé, ce qui préserve le jeu de caractères courant.

Registre 1 du TS : TS Mode

Bit	Signification	Accès
7-6	réservé	RW
5	Screen off	RW
4	Shift 4	RW
3	Dot Clocks / 2	RW
2	Video Load / 2	RW
1	réservé	
0	TS State	RW

Explication

Bit 5 : Mis à 1, il provoque la désactivation de l'écran, ce qui permet au processeur d'accéder plus rapidement à l'écran.

Bit 4 : S'il est égal à 1, les bascules sont chargées au quart de la fréquence

Bit 3 : Mis à 1 dans certains modes 320x200 pour réduire la résolution horizontale de 640 à 320.

Bit 2 : S'il est égal à 1, les bascules sont chargées à la moitié de la fréquence.

Bit 0 : Un état nul fixe la largeur des caractères à 9 pixels (1 correspondant à une largeur de 8 pixels). Ce bit joue surtout un rôle dans le calcul du timing horizontal dont les registres reposent tous sur une unité de caractère.

Registre 2 du TS : Write Plane Mask

Bit	Signification	Accès
7-4	réservé	
3	Accès en écriture au plan 3	RW
2	Accès en écriture au plan 2	RW
1	Accès en écriture au plan 1	RW
0	Accès en écriture au plan 0	RW

Explication

Bits 3-0 : Ce registre permet d'exclure (0) certains plans de l'accès en écriture ou au contraire de les y ouvrir (1).

Registre 3 du TS : Font Select

Bit	Signification	Accès
7-6	réservé	
5	Fonte B bit 2	RW
4	Fonte A bit 2	RW
3-2	Fonte B bits 0 et 1	RW
1-0	Fonte A Bits 0 et 1	RW

Explication

Bits 5-0 : Indique la situation du jeu de caractères en mémoire avec la codification suivante :

Bits	Offset
0 0 0	0
0 0 1	16 Ko
0 1 0	32 Ko
0 1 1	48 Ko
1 0 0	8 Ko
1 0 1	24 Ko
1 1 0	40 Ko
1 1 1	56 Ko

La fonte A détermine l'aspect des caractères dont le bit d'attribut, le bit 3, n'est pas à 1. La fonte B est activée lorsque ce bit est à 1.

Registre 4 du TS : Memory Mode

Bit	Signification	Accès
7-4	réservé	
3	Chain 4	RW
2	Mode pair/impair	RW
1	Extended Memory	RW
0	réservé	

Explication

Bit 3 : Enclenche le mode Chain 4 (1). Les lignes d'adresse 0 et 1 sont alors utilisées pour sélectionner les plans lors des accès en lecture ou en écriture du processeur. Ce mode trouve une application en mode 13h. Si on le désactive et si on met en outre le CRTC en mode byte, on se trouve dans le mode X (voir chapitre 5.2).

Bit 2 : Si ce bit est nul, le mode pair/impair est actif. Ce dernier fonctionne un peu de façon semblable au mode Chain 4. Le bit 0 de la ligne d'adresse est utilisé pour sélectionner les plans pairs ou impairs. En mode texte, ceci signifie par exemple que les codes ASCII des caractères seront mémorisés dans les plans 0 et 2 tandis que les plans 1 et 3 accueilleront les octets représentatifs des attributs. Ce bit devrait toujours correspondre au bit 4 du registre 5 du GDC (inversé !).

Bit 1 : Indique la mémoire disponible : 0 signifie 64 Ko de mémoire d'écran, et 1 correspond à 256 Ko. Doit être à 1 si on veut utiliser le bit 2 de la sélection du jeu de caractères.

Contrôleur de données graphiques (Graphics Data Controller, GDC)

Ce circuit est destiné à commander les accès mémoire du processeur à un niveau plus élevé que le TS. Il régit les manipulations que subit un octet du processeur dans les différents modes de lecture ou d'écriture. Il contient par ailleurs les quatre bascules (latches) qui ne sont pas directement accessibles de l'extérieur : celles-ci captent un octet de chaque plan lors de chaque accès processeur. Si l'accès est en lecture, l'un des plans (en fonction du registre 4 du GDC) est envoyé au processeur, en cas d'écriture, les bascules sont reliées à cet octet (en fonction du registre 3) et émises vers le plan. Le GDC est adressé comme le CRTC par un port d'index (adresse 3ceh) et un port de données (3cfh).

Registre 0 du GDC : Set /reset

Bit	Signification	Accès
7-4	réservé	
3	Valeur de set /reset pour le plan 3	RW
2	valeur de set /reset pour le plan 2	RW
1	Valeur de set /reset pour le plan 1	RW
0	Valeur de set /reset pour le plan 0	RW

Explication

Bit 3-0 : Indiquent la valeur de Set/Reset pour les différents plans dont la fonction set/reset a été activée (voir registre 1).

Registre 1 du GDC : Enable Set /reset

Bit	Signification	Accès
7-4	réservé	
3	Active la fonction set /reset (1) pour le plan 3	RW
2	Active la fonction set /reset (1) pour le plan 2	RW
1	Active la fonction set /reset (1) pour le plan 1	RW
0	Active la fonction set /reset (1) pour le plan 0	RW

Explication

Bit 3-0 : Le bit correspondant à un plan indique que la fonction set/reset est activée pour ce plan. Dans ce cas, la bascule associée n'est pas reliée à l'octet du processeur mais à la valeur 0 (bit correspondant dans le registre 0 = 0) ou 0ffh (bit = 1). En mode écriture 1, cette fonction n'est pas disponible.

Registre 2 du GDC : Color Compare

Bit	Signification	Accès
7-4	réservé	
3	Valeur de comparaison de couleur pour le plan 3	RW
2	Valeur de comparaison de couleur pour le plan 2	RW
1	Valeur de comparaison de couleur pour le plan 1	RW
0	Valeur de comparaison de couleur pour le plan 0	RW

Explication

Bits 3-0 : Ces bits jouent un rôle dans le mode de lecture 1. Voir registre 5, bit 3 (read mode).

Registre 3 du GDC : Function Select

Bit	Signification	Accès
7-5	réservé	
4-3	Function Select	RW
2-0	Compteur de rotation	RW

Explication

Bit 4-3 : Précise le type de lien entre l'octet du processeur et les quatre bascules :

0	Move (écriture destructive)
1	AND
2	OR
3	XOR

Ces indications sont ignorées dans le mode d'écriture 1 où l'écriture est toujours directe.

Bit 2-0 : Nombre de bits de la rotation vers la droite à laquelle on souhaite soumettre un octet du processeur, avant qu'il ne soit mis au contact des bascules.

Registre 4 du GDC : Read Plane Select

Bit	Signification	Accès
7-2	réservé	
1-0	Plane Select	RW

Explication

Bit 1-0 : Indiquent sous forme d'entier à 2 bits le plan concerné par un accès en lecture du processeur.

Registre 5 du GDC : GDC Mode

Bit	Signification	Accès
7	réservé	
6	Mode 256 couleurs	RW
5	Shift	
4	Mode pair / impair	RW
3	Mode de lecture	RW
2	réservé	
1-0	Mode d'écriture	RW

Explication

Bit 6 : Active le mode 256 couleurs, dont la répartition des plans est spécifique.

Bit 5 : Mis à 1 dans le mode CGA 320x200 pour générer 4 couleurs.

Bit 4 : Active le mode pair/impair du point de vue du GDC. A faire correspondre au bit 2 du registre 4 du TS (bit inversé !).

Bit 3 : Indique le mode de lecture actif.

Mode 0 : Le GDC lit les quatre bascules et envoie au processeur le contenu de celui qui a été sélectionné par le registre 4.

Mode 1 : Le GDC lit les quatre bascules et compare leurs bits avec ceux du registre Color Compare (sauf exclusion par le registre 7 Color Care). La comparaison porte d'abord sur les bits 0 des quatre bascules, puis sur les bits 1, etc. En cas d'identité complète, le bit correspondant est mis à 1 dans l'octet envoyé au processeur.

Bits 1-0 : Indique le mode d'écriture courant

Mode 0 : L'octet du processeur est mis en contact avec les quatre bascules et écrit dans les plans fixés par le registre 2 du TS. Seuls les bits autorisés par le registre 8 sont envoyés. Si la fonction set /reset est activée pour le plan (voir registre 1), ce n'est pas l'octet du processeur mais la valeur de set / reset qui est utilisée. Ce mode convient parfaitement aux manipulations de bits isolés.

Mode 1 : dans ce mode, le contenu des bascules est envoyé directement dans les plans indiqués par le registre 2 du TS. L'octet du processeur ne joue aucun rôle. Ce mode est intéressant pour copier rapidement des parties d'écran importantes.

Mode 2 : Les quatre bits inférieurs du processeur sont étendus pour donner un octet (0 devient 0 et 1 se transforme en 0ffh) puis reliés aux bascules correspondantes comme dans le mode d'écriture 0.

Mode 3 : Chaque bit du processeur qui n'est pas masqué par le registre 8 est relié à l'un des bits du registre set/reset et écrit dans le plan correspondant à ce bit. Ainsi le bit 0 est d'abord relié au bit 0 du registre set/reset selon le registre 3 dit Function Select et écrit dans le plan 0 comme bit 0. Puis le bit 0 est relié au bit 1 du registre set/reset et écrit dans le plan 1. L'opération se répète pour tous les bits sélectionnés de l'octet du processeur.

Registre 6 du GDC : Miscellaneous

Bit	Signification	Accès
7-4	réservé	
3-2	Memory Map	RW
1	Mode pair/impair	RW
0	Mode graphique	RW

Bits 3-2 : Indique l'emplacement de la mémoire d'écran:

0	0a0000h-0bffffh
1	0a0000h-0affffh
2	0b0000h-0b7fffh
3	0b8000h-0bffffh

Explication

Bit 1 : Egal à 1 si le mode pair/impair est actif.

Bit 0 : Egal à 1 si un mode graphique est actif, sinon c'est un mode texte.

Registre 7 du GDC : Color Care

Bit	Signification	Accès
7-4	réservé	
3	Color Compare actif pour le plan 3	RW
2	Color Compare actif pour le plan 2	RW
1	Color Compare actif pour le plan 1	RW
0	Color Compare actif pour le plan 0	RW

Explication

Bits 3-0 : Un bit égal à 1 inclut le plan correspondant dans la comparaison de couleur en mode de lecture 1 (voir registre 5), un 0 provoque l'exclusion.

Registre 8 du GDC : Bit Mask

Bit	Signification	Accès
7-0	Masque pour l'octet du processeur	RW

Explication

Bits 7-0 : Un bit égal à 1 signifie que le bit correspondant du processeur est relié aux bascules. Avec un bit égal à 0, le contenu des bascules est repris sans changement.

Le contrôleur d'attributs (Attribute Controller, ATC)

Ce contrôleur a pour tâche de gérer les couleurs en avant-dernière instance pour les cartes VGA. L'exploitation de ce contrôleur est un peu plus compliquée : pour les accès en écriture, on ne dispose que d'un seul port. En 3C0h se trouve un flip-flop index/données, qui à chaque écriture change de rôle en commutant donc entre index et données. Pour fixer un premier état de sortie, il convient d'activer explicitement le mode index par un accès en lecture du registre 1 de l'Input Status (port 3dah et 3bah en mode couleur et en Monochrome). On peut ensuite effectuer un accès en écriture en écrivant d'abord l'index en 3C0h puis en émettant sur la même adresse les données souhaitées, après quoi l'index suivant peut être envoyé directement.

L'accès en lecture s'effectue un peu différemment : après écriture de l'index, le port 3c1h permet de lire l'octet de données. La lecture du port 3C0h renverrait l'index.

La structure du registre d'index/données en 3c0h est également particulière. Les bits 4-0 donnent l'index, comme d'habitude, mais le bit 5 a une autre signification.

Registre de l'ATC : Index/Data port 3c0h

Bit	Signification	Accès
7-6	réservé	
5	Accès à la mémoire de la palette	RW
4-0	Index de l'ATC	RW

Explication

Bit 5 : Si ce bit est égal à 0, il ouvre l'accès du processeur à la mémoire vive de la palette (registres 0-f) mais en découple l'ATC, de sorte qu'il mette l'image dans la couleur de cadre. Après modification, il faut donc toujours ramener ce bit à 1.

Bits 4-0 : Ils renvoient à un registre interne de l'ATC. La valeur est ensuite écrite sur le port 3c0h, la lecture s'effectue sur le port 3c1h.

Registres 0-f de l'ATC : Palette Ram

Bit	Signification	Accès
7-0	Couleur DAC	RW

Explication

Bit 7 : Ce registre permet d'attribuer à chaque valeur de couleur EGA une couleur proprement dite. Avec les cartes EGA, on trouve ici directement les composantes rouge-vert-bleu. Avec les cartes VGA, le convertisseur digital/analogique (Digital to Analog Converter, DAC) s'interpose en simulant cette structure dans ses 16 premières entrées de palette. On peut exploiter ce registre pour fixer les couleurs de texte avec de nouvelles valeurs DAC.

Registre 10h de l'ATC : Mode Control		
Bit	**Signification**	**Accès**
7	Origine des lignes de couleur 4 et 5	RW
6	PelClock / 2	RW
5	Enable Pixel Panning	RW
4	réservé	
3	Clignotement	RW
2	Line Graphics enable	RW
1	Attributs monochromes (1) / couleur (0)	RW
0	Graphique (1)/ texte (0)	RW

Explication

Bit 7 : Lorsque le mode graphique comporte 16 couleurs ou moins, ce bit décide de l'origine des lignes de couleur 4 et 5. S'il est égal à 1, ces indications sont tirées des bits 0 et 1 du registre 14 (Color Select). S'il est égal à 0, elles proviennent des registres de palette. Si on exploite la méthode Color Select, on peut déplacer la zone des palettes DAC par blocs de 16. Ceci évidemment aussi dans les modes EGA et CGA.

Bit 6 : Egal à 1, il divise par deux la vitesse de transfert des données pixels au DAC. On utilise cette propriété en mode 320x200x256 car, dans ce mode, la résolution horizontale est moitié moindre.

Bit 5 : S'il est égal à 1, ce bit empêche le Pixel Panning en dessous de la ligne de partage d'écran : un 0 provoque un panoramique de l'ensemble de l'écran.

Bit 3 : Si ce bit est mis à 0, les 16 couleurs de la palette AC sont librement exploitables. S'il est égal à 1, il se produit un clignotement. En mode texte comme en mode graphique, le plan d'intensité passe alors constamment de la moitié supérieure de la palette à la moitié inférieure et vice-versa, il ne reste donc plus que huit couleurs disponibles. Mais le contenu de la palette demeure libre, de sorte que le clignotement permet d'obtenir toutes sortes d'effets.

Bit 2 : Si ce bit contient 1, la huitième colonne des caractères dont le code ASCII est compris entre 0c0h et 0dfh est dédoublée en mode texte à 9 pixels. Les caractères qui dessinent des filets peuvent alors être juxtaposés sans lacune. Si le bit contient 0, la neuvième colonne est effacée ou tirée du plan d'intensité (plan 3).

Bit 1 : 1 donne des attributs monochromes (clignotement, soulignement), 0 des attributs couleur.

Bit 0 : 1 déclenche le mode graphique, 0 le mode texte

Registre 11h de l'ATC : Overscan Color

Bit	Signification	Accès
7-0	Couleur DAC pour le cadre d'écran	RW

Explication

Bits 7-0 : Fixent le numéro de la couleur mise en service dans le domaine de l'overscan. Sur les cartes EGA, contient une valeur RGB à 6 bits. Avec les cartes VGA, le mélange des couleurs se fait par le DAC.

Registre 12h de l'ATC : Color Plane Enable

Bit	Signification	Accès
7-6	réservé	
5-4	réservé, souvent configuration pour bits de test	RW
3-0	Enable Plane	

Explication

Bits 3-0 : Activent le plan correspondant (1) ou le désactivent, ce qui exclut de l'affichage certains composants de couleur (modèle à 16 couleurs) ou certains pixels (modèle à 256 couleurs).

Registre 13h de l'ATC : Horizontal Pixel Panning

Bit	Signification	Accès
7-4	réservé	
3-0	Horizontal Pixel Panning	RW

Explication

Bits 3-0 : Donnent le nombre de pixels dont on déplace l'image globale vers la gauche (mode texte ou graphique). Les valeurs n'ont pas la même signification selon le mode. En mode 9 pixels, une valeur de 8 signifie qu'aucun déplacement ne doit avoir lieu, les valeurs de 0-7 provoquent un déplacement de 1 à 8 pixels. En mode graphique, ce paramétrage n'est pas très intéressant car il est plus simple de travailler avec l'adresse de début. Mais en mode texte, on peut gérer ainsi un défilement très soigné (voir chapitre 6.6 - Défilement continu en mode texte).

Registre 14 h de l'ATC : Color Select

Bit	Signification	Accès
7-4	réservé	
3-2	Ligne de couleur 7-6	RW
1-0	Ligne de couleur 5-4	RW

Explication

Bits 3-2 : Dans les modes graphiques à moins de 256 couleurs, ces bits donnent les deux bits supérieurs de chaque couleur envoyée au DAC. En reprogrammant ces registres, on peut déplacer les 16 couleurs EGA vers les offsets de palette 0, 64, 128 et 192.

Bits 1-0 : Si le bit 7 du registre 10 h (Mode Control) est à 1 et si on se trouve dans un mode graphique à moins de 256 couleurs, ces deux bits correspondent aux bits de couleurs 5 et 4. Ainsi on peut déplacer, en liaison avec les bits 3-2, les couleurs EGA aux offsets de palette multiples de 16.

Convertisseur Analogique Digital (Digital to Analog Converter DAC)

Le DAC représente la dernière instance dans la génération des couleurs. C'est ici que les valeurs de couleurs exprimées en 8 bits, qu'elles proviennent directement de la mémoire (256 couleurs) ou de l'ATC (conversion à partir de 4 bits) sont transformées en signaux analogiques grâce à la palette externe et envoyées vers le moniteur en composantes séparées rouge, verte et bleue. Le mot "externe" signifie que la mémoire vive se trouvait autrefois en dehors du jeu de circuits proprement dit. Entre-temps, l'intégration a également touché ce circuit. Le DAC exploite donc une palette externe de 256 couleurs, chaque couleur étant codée sur trois octets représentant les composantes rouge, verte et bleue, seuls les 6 bits inférieurs étant significatifs. Chaque couleur peut donc donner une nuance issue de 2^18=262144 possibilités différentes.

Pour lire ou entrer une palette, il faut d'abord indiquer la couleur de début, à partir de laquelle on souhaite effectuer la lecture ou l'écriture. Puis on utilise des commandes successives Out et In pour lire ou écrire dans l'ordre les composantes rouge, verte et bleue de chaque couleur. Toutes ces valeurs peuvent être écrites bout à bout car le DAC incrémente automatiquement le pointeur. Il n'est pas recommandé de gérer la méthode par le Bios. Prétendue "compatible", cette méthode ne l'est pas tellement, et par ailleurs sa durée d'exécution est dix fois supérieure.

Le DAC permet d'accéder à la palette par le moyen de 5 registres.

Pixel Mask	**RW**
Port:	3c6h

Explication

Ce registre contient habituellement la valeur 0ffh. Il représente un masque pour combiner des couleurs. Chaque fois que le DAC lit une entrée de palette au moment de la génération de l'écran, il soumet la couleur lue à une opération AND de combinaison avec ce registre.

Pixel Write Address	**RW**
Port	3c8h

Explication

Avant tout accès en écriture, il faut communiquer au DAC par l'intermédiaire de ce registre quelle est la couleur que l'on souhaite modifier. On peut ensuite écrire dans la palette ou une partie de la palette par le port d'adresse 3c9h (Pixel Color Value).

Pixel Read Address	**WO**
Port	3c7h

Explication

Avant tout accès en lecture, il faut communiquer au DAC par l'intermédiaire de ce registre la couleur à lire. On peut ensuite lire les valeurs des couleurs par le port 3c9h (Pixel Color Value).

Pixel Color Value	**RW**
Port	3c9h

Explication

C'est ici que le DAC envoie les valeurs des couleurs à lire ou à écrire, après initialisation du processus d'écriture ou de lecture par les adresses Pixel Read Address ou Pixel Write Address.

DAC State	R0
Port	3c7h

Les deux bits inférieurs de ce registre indiquent l'état d'accès courant du DAC : 0 signifie que le DAC a préparé des données à lire, 11 signifie que le DAC est prêt pour une opération d'écriture.

Les cartes Super VGA disposent encore de bien plus de registres, aussi bien des registres isolés supplémentaires que des registres de contrôle étendus. Il existe même parfois des circuits supplémentaires chargés par exemple de gérer la mémoire. Comme ces registres ne sont pas normalisés, il est malheureusement impossible de les décrire d'une façon générale. Mais comme nous l'avons vu, il est déjà possible de programmer un grand nombre d'effets avec les registres standard. N'hésitez pas à vous lancer dans un voyage à travers les registres, vous trouverez peut-être un nouvel effet.

5. Le mode X, ou le secret le mieux gardé des codeurs de la scène

Si l'application du mode 13h paraît facile, il présente cependant de graves inconvénients qui excluent pratiquement la programmation d'effets spectaculaires. Malgré les énormes progrès réalisés, les processeurs et les architectures de bus ne sont pas encore capables de calculer des zones d'écran étendues dans l'intervalle d'un retour de balayage.

5.1. La puissance graphique grâce au mode X

S'il n'est pas possible de calculer un dessin d'écran pendant un retour de balayage, la qualité d'image en souffre énormément. Un exemple simple : un sprite se déplace horizontalement sur l'écran. Si le rayon cathodique traverse une zone d'écran en cours de modification, il représente l'image nouvelle dans la partie supérieure et l'image ancienne dans la partie inférieure. Dans le meilleur des cas, on observe un scintillement mais la plupart du temps on obtient un sprite déchiré et déformé. S'il s'avère interdit de faire des modifications pendant la génération de l'écran et qu'il est impossible de les faire pendant le retour de balayage, il ne reste qu'une solution : la page d'écran doit être calculée de façon quasi invisible et activée pendant le retour de balayage vertical. Nous avons donc besoin de deux pages d'écran, l'une en cours d'affichage pendant que l'autre est en cours de calcul.

Ces pages peuvent être stockées l'une derrière l'autre dans la mémoire d'écran, leur sélection s'opérant par les registres 0ch et 0dh (Linear Starting Address). Mais en mode 13h, l'image prend presque 64 Ko (64000 octets pour être précis). Une page d'écran occupe donc presque toute la mémoire vidéo

projetée dans la mémoire principale (0a0000h-0affffh), ce qui empêche le processeur d'adresser la deuxième page d'écran. Les modifications ne sont possibles qu'à travers les sélecteurs de segment de la carte VGA mais ceux-ci se programment différemment selon le fabricant. La carte VGA d'IBM n'offre à cet égard aucune possibilité. La solution de ce problème réside dans le mode X qui est décrit ci-après.

Un autre inconvénient de taille est dû à la vitesse d'accès. Les images animées se déroulent presque toujours sur un fond statique qui doit être recopié à chaque génération d'écran sur la page courante. Pour copier une pleine page d'écran, il faut 10% plus de temps en mode 13h avec accès rapide doubleword par le bus local VESA que dans le mode X. Avec les autres bus systèmes, la différence peut encore être plus importante. Si on passe à l'accès word ne serait-ce que pour supporter le processeur 286, le processus de copie devient insupportablement lent. En mode X, comme nous le verrons dans les pages qui suivent, chaque accès byte copie quatre pixels à la fois. Par ailleurs, on exploite le mode lecture 0 et le mode d'écriture 1 qui ne demandent pas de conversion d'adresse interne importante et n'envoient pas non plus de données au processeur, ce qui donne bien un avantage de vitesse par rapport aux accès 32 bits du processeur.

5.2. INITIALISATION

Quelles doivent être les propriétés du mode X ? La désignation mode X est devenue entre-temps quasi officielle et se réfère toujours au même modèle de mémoire. Des résolutions supérieures à 320x400 sont possibles sur toutes les cartes VGA. Même le 320x480 peut être désormais programmé sur presque toutes les cartes, mais il s'agit bien de "presque toutes" les cartes. Dans ce cas, une page d'écran exige plus de 128 Ko ce qui fait perdre l'avantage de l'exploitation de différentes pages. Il est essentiel de désactiver le mécanisme Chain 4 de façon que l'accès individuel aux différents plans soit libre. Il faut aussi s'assurer que le mode pair/impair (sélection de plan par le bit d'offset inférieur) est inactif, il suffit à cet effet d'effacer le bit 3 (Enable Chain 4) du registre 4 du TS (Memory Mode) et de mettre à 1 le bit 2 (Mode pair/impair).

```
port[$3c4]:=4;
port[$3c5]:=port[$3c5] (and (not 8)) or 4;
```

Selon la carte graphique dont on dispose (vive la compatibilité), il faut encore activer l'adressage par byte de l'accès mémoire : mettre d'abord à zéro le bit 6 du registre 14h du CRTC (Underline Row Address) qui représente l'adressage doubleword et mettre ensuite à 1 le bit 6 du registre 17h du CRTC (Mode CRTC).

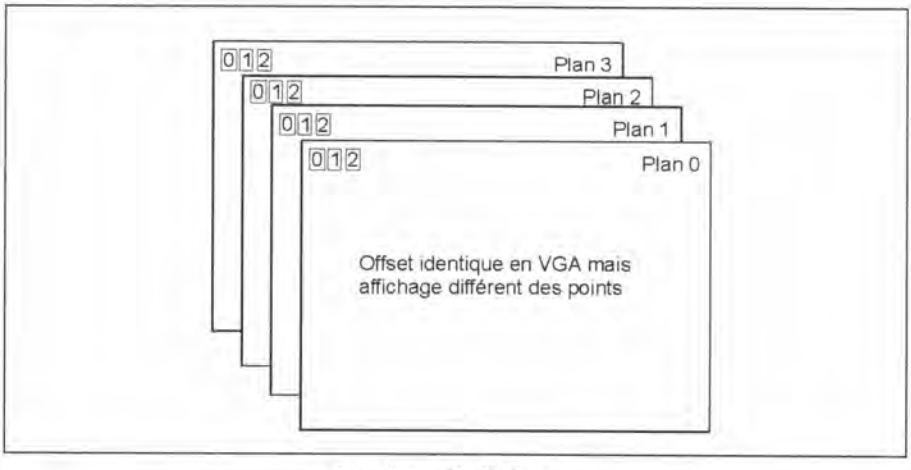
```
port[$3d4]:=$14;
port[$3d5]:=port[$3d5] and (not 64);
port[$3d4]:=$17;
port[$3d5]:=port[$3d5] or 64;
```

Il convient à présent d'effacer la mémoire d'écran car les endroits qui ne sont pas exploités par le mode 13h vont devenir visibles et peuvent présenter des résidus en provenance d'autres modes vidéo. Pour effectuer cet effacement, nous allons nous aider du registre 2 du séquenceur de timing (Write Plane Mask) : tous les plans de la mémoire d'écran vont être activés, de sorte que 32000 accès word ou 16000 accès Dword suffiront pour effacer l'ensemble des quatre pages d'écran.

Nous avons maintenant un écran vierge en mode X. Il nous faut trouver quelques idées pour l'exploiter car aucun BGI ni sous-programme du Bios ne vont nous apporter la moindre assistance, mis à part pour la sortie du mode X et le retour à un mode texte. Pour faire apparaître des images sur le moniteur, nous devons encore une fois étudier en détail la structure en cause.

5.3. STRUCTURE

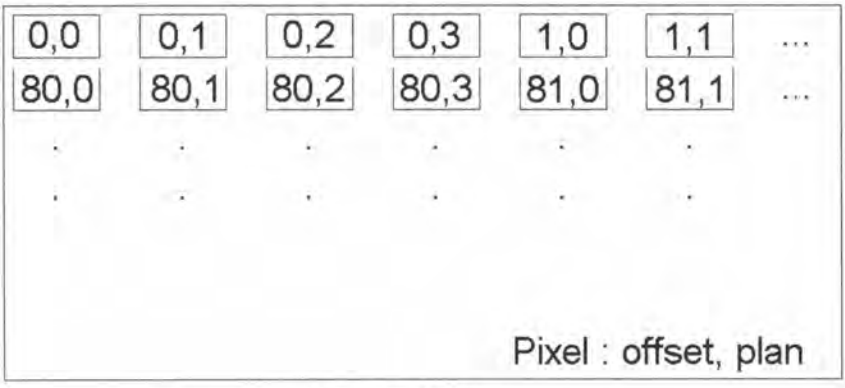
Comme nous l'avons déjà évoqué, dans tous les modes graphiques basés sur la représentation par plans, chaque adresse mémoire correspond à quatre octets, un par plan. Ces quatre octets sont pratiquement superposés à la même adresse, d'où la notion de plan.



Les plans d'affichage

Les plans sont des mémoires quasi indépendantes qui sont adressables séparément mais qui sont exploitées parallèlement lorsque l'image est affichée sur l'écran, autrement dit les données en provenance des quatre plans sont lues simultanément.

Dans le détail, il existe des différences de taille entre les modes à 16 couleurs et le mode X. Pour représenter 16 couleurs, il suffit de 4 bits, d'où les quatre plans. Le bit 0 d'un pixel est stocké dans le bit correspondant du plan 0, le bit 1 dans le plan 1 et ainsi de suite. En mode X, les choses sont différentes. Les 4 bits sont insuffisants pour l'adressage d'un pixel, il faut désormais un octet complet. Les plans sont ainsi remplis octet par octet : le pixel 0 se trouve à l'offset 0 du plan 0, le pixel 1 au même offset dans le plan 1, c'est le pixel 4 qui se trouve à l'offset 1, de nouveau dans le plan 0.



Structure des plans

Pour l'adressage en mode X, il faut d'abord calculer l'offset d'un pixel en divisant par quatre son numéro (obtenu comme dans le mode 13h par la formule 320*y+x). Cette division peut être effectuée par un décalage de deux bits vers la droite et intégrer immédiatement les considérations d'offsets : l'ordonnée y ne sera multipliée que par 80 (320/4) et l'abscisse x subira une division supplémentaire par 4. Le reste (numéro du pixel AND 3) donne le plan à sélectionner. On aura donc :

```
plan:=X mod 4      (donne le reste)
offset:=Y*80 + X div 4
```

Cette méthode correspond exactement à celle mise en oeuvre par le mode 13h, à une seule différence près : l'offset est formé par décalage de 2 bits à droite du numéro de pixel, et non par masquage. Ainsi, la mémoire ne présente pas de lacune entre les pixels et on peut stocker quatre pages dans les 256 Ko de la mémoire d'écran.

Lors de la sélection des plans en mode X, il faut cependant faire la différence entre les accès en écriture et en lecture. A la lecture, le numéro de plan est écrit dans le registre 4 du GDC (Read Select). A l'écriture, il est par contre possible d'adresser plusieurs plans en même temps (comme nous l'avons déjà vu dans la procédure d'effacement de l'écran). C'est pourquoi on fixe un masque dans le registre 2 du TS (Write Plane Mask), à déduire du numéro de plan par la formule :

```
Masque:=1 shl Numéro de plan
```

Si on prend le plan 2, on aura par exemple 1 shl 2 = 4 = 0100.

Encore un mot à propos de l'adressage par octet du côté du processeur. La tentation est grande d'accéder aux données graphiques par le moyen d'accès sur 32 bits (à quoi bon posséder un 386 ?), ou au moins d'utiliser 16 bits (sur un XT).

Si vous faites la tentative, vous en tirerez une amère leçon. L'image est complètement déformée, ce qui s'explique par le mode d'accès du processeur. Lorsque le processeur copie un mot (word) par une instruction du type movsb (le même principe étant valable pour un double mot), il ne décompose évidemment pas le processus en déplacements d'octets mais lit un mot complet pour le réécrire intégralement ailleurs.

Mais la carte VGA, avec ses quatre bascules qui servent de mémoire intermédiaire, ne peut prendre que quatre octets. Après l'accès en lecture, les bascules ne contiennent que les quatre octets de poids fort du mot lu. Lors de l'accès en écriture qui suit, les quatre octets de poids fort et les quatre octets de poids faible possèdent la même valeur issue de la bascule. A l'écran, on observe que des quartets de pixels successifs (et même quatre quartets en accès 32 bits) ont le même contenu, ce qui rend impossible tout affichage sensé.

5.4. Davantage de résolution en mode X

Le mode X présente l'avantage de pouvoir fonctionner avec quatre pages d'écran, mais rien n'est changé sur le plan de la résolution par rapport à son prédécesseur le mode 13h. Pour certaines applications, il est cependant nécessaire de disposer d'une meilleure résolution. Il existe évidemment les cartes SuperVGA. S'il est vrai qu'elles ont chacune leur adressage spécifique, on peut en cas de besoin recourir à un driver VESA. Mais lorsqu'on utilise plusieurs pages d'écran en haute résolution, il se pose un problème. Les cartes actuelles ont presque toujours 1Mo, ce qui permet de mémoriser en théorie deux pages d'écran en mode 800x600, mais la norme VESA ne prévoit pas la possibilité de gérer les pages d'écran à la façon du registre Linear Starting

Address. La programmation directe se heurte donc à nouveau au problème de compatibilité de la carte SuperVGA.

En recherchant un mode qui offre une meilleure résolution tout en supportant la notion de page, on ne peut manquer de s'apercevoir que le mode X supporte quatre pleines pages alors que seules deux sont nécessaires la plupart du temps. Il paraît donc possible de doubler la résolution en la faisant passer à 320x400. Mais pourquoi 320x400 et non 640x200? Ce choix dépend en fait de la structure propre au mode 200 lignes sur les cartes VGA. Il ne faut pas oublier en effet que les cartes VGA ne sont pas capables d'adresser explicitement 200 lignes (les bits 6-7 du registre Miscellaneous Output n'autorisent comme nombre de lignes en résolution verticale que les valeurs 350, 400, 480 et partiellement 768).

Mais il existe la faculté de double scan. En résolution verticale de 400 lignes, chaque ligne est affichée deux fois donc dédoublée dans l'axe des ordonnées, ce qui conduit à une division par deux de la résolution. Le timing horizontal ou vertical n'a pas besoin d'être changé car physiquement ce sont toujours 400 lignes de 320 pixels qui sont représentées.

Il est très facile de désactiver ce mécanisme. Il suffit de mettre à 0 cinq bits du registre 9 du CRTC (Maximum Row Address) : le bit 7 et les bits 0-3. Selon le Bios VGA, le dédoublement se fait par le bit 7, normalement prévu à cet effet (Double Scan Enable) ou par les bits 0-3 qui indiquent en mode texte le nombre de lignes de balayage impliquées dans une ligne de caractères, à diminuer de 1 avant écriture dans le registre. En mode graphique, ce registre correspond au nombre de copies d'affichage supplémentaires d'une ligne. En augmentant sa valeur, on peut diminuer progressivement la résolution verticale, jusqu'à 320x25, et même en liaison avec le bit 7 jusqu'à 320x12,5. Ces valeurs sont dénuées d'intérêt mais sont là pour donner une idée du fonctionnement du registre.

Pour revenir aux 400 lignes d'origine, les bits du registre précités sont à remettre à 0. En pseudo-code :

```
CRTC[9]:=CRTC[9] AND 01110000
```

Cette instruction se trouve en assembleur dans la procédure Enter400 du module MODEXLIB.ASM :

```
Enter400 proc pascal far        ;active le mode X (200 lignes)
  mov dx,3d4h                   ;et l'étend sur 400 lignes
  mov al,9                      ;registre 9 du CRTC (Maximum
                                ;Row Adress)
  out dx,al                     ;sélectionne le registre
  inc dx                        ;lit le contenu
```

```
    in al,dx
    and al,01110000b            ;met à 0 les bits 7 et 3-0
    out dx,al                   ;renvoie la nouvelle valeur
    ret
Enter400 endp
```

Comme nous l'avons dit, aucune modification de timing ne s'avère nécessaire et même les routines se référant au mode X n'ont pas besoin d'être retouchées, la structure des données restant identique. En principe, les 200 lignes du mode X normal sont rassemblées vers le haut lorsqu'on enclenche les 400 lignes, libérant la vue sur les 200 lignes situées en dessous, qui suivent directement en mémoire VGA.

Les routines disposent désormais d'un domaine de coordonnées plus étendu, à exploiter librement puisqu'aucun test de validité n'a été installé pour accroître la vitesse d'exécution. La seule modification qui nous intéresse consiste à passer de 4 pages de 64000 octets à 2 pages de 128000 octets?

Au moment de la commutation, ce changement ne joue aucun rôle car l'adresse de début (vpage) n'oscille plus entre 0 et 16000 mais entre 0 et 128000. Par ailleurs, il n'existe plus de page de réserve pour le fond de l'écran, étant donné que les 256 Ko se trouvent entièrement occupés. Il reste en toute rigueur 256 Ko-256000 octets = 6 Ko, qui peuvent être exploités pour de tout petits fonds de sprite.

Dans ce mode, les fonds d'écran doivent donc être copiés de la mémoire principale vers la mémoire VGA (à l'aide de p13_2_ModeX, voir chapitre 5.6 Extension du chargeur de Gif en mode X).

Pour dessiner un fond d'écran, il faudra bien tenir compte du rapport 320/400 qui est assez inhabituel et qui écrase les formes : les rectangles sont aplatis et les cercles deviennent des ellipses allongées. A notre connaissance, aucun programme de dessin n'exploite le mode 320x400, c'est pourquoi il nous semble recommandé de faire les dessins dans une résolution "carrée" (par exemple 640x480) puis de les ramener par calcul en 320x400.

A côté de cette nouvelle résolution, il est également possible d'atteindre une finesse encore plus grande. On peut ainsi travailler en 512x400 ou 320x480 mais on est alors dans le domaine des curiosités. Si on tient compte de l'initialisation complexe de ces modes, qui nécessitent une reprogrammation complète des registres de timing, on s'aperçoit qu'ils ne présentent aucun avantage déterminant par rapport au mode SuperVGA : la mémoire consommée est trop importante pour laisser de la place à la gestion des pages.

Une page unique peut être affichée sur n'importe quelle carte au standard VGA mais, avec deux pages, on dépasse les 256 Ko qui correspondent à la limite supérieure d'adressage par les méthodes usuelles. Les adresses situées au-delà posent le problème de l'incompatibilité des cartes SuperVGA, de sorte qu'il est préférable de recourir directement au mode SuperVGA, qui présente au moins une capacité de pagination raisonnable.

5.5. Application du mode X

Même dans le mode X, il est parfois nécessaire d'adresser des pixels isolés et de modifier leur couleur. Lorsque les applications reposent fondamentalement sur ce type de modification (par exemple un défileur d'étoiles), il vaut mieux utiliser le mode 13h, à supposer que l'on puisse se passer des quatre pages d'écran. Non seulement ce mode rend plus facile l'adressage individuel des pixels mais il se révèle également plus rapide car la décomposition de l'offset en offset de mémoire et numéro de plan a lieu à l'intérieur des circuits VGA et échappe au processeur.

Par ailleurs, l'émission du numéro de plan sur les adresses des ports concernés du TS et du GDC se révèle parfois assez lente sur certains circuits implantés sur la carte mère. Sur les ordinateurs performants (386 et au-delà), les circuits rajoutent parfois des états d'attente en nombre considérable, pour laisser à l'électronique le temps de digérer les données, ce qui est totalement inutile avec les matériels actuels.

Dessin de pixel

Pour modifier directement de temps à autre quelques pixels, on peut suivre la procédure décrite ci-après qui convient également au défileur d'étoiles à condition que l'on initialise le mode X à la place du mode 13h.

```
Procedure PutPixel(x,y,col:word);assembler;
{met le pixel (x,y) en couleur col (mode X)}
asm
  mov ax,0a000h                    {charge le segment }
  mov es,ax

  mov cx,x                         {détermine le plan d'écriture}
  and cx,3                         {comme x mov 4}
  mov ax,1
  shl ax,cl                        {met à 1 le bit correspondant}
```

```
  mov ah,al
  mov dx,03c4h                    {registre 2  - Write Plane Mask -}
  mov al,2                        {du séquenceur de timing }
  out dx,ax

  mov ax,80                       {Offset = Y*80 + X div 4}
  mul y
  mov di,ax
  mov ax,x
  shr ax,2
  add di,ax                       {charge l'offset }
  mov al,byte ptr col             {charge la couleur }
  mov es:[di],al                  {puis dessine le pixel }
End;
```

Les deux bits inférieurs de l'abscisse x sont extraits par masquage ce qui détermine le plan. Mais ce dernier est à convertir au format du registre Write Plane Mask : il faut mettre à 11 le bit correspondant au numéro du plan. Il suffit ensuite de calculer l'offset (décalage de 2 bits vers la droite par rapport au mode 13h) et de fixer la couleur.

Changement de page

Le grand avantage du mode X est de permettre l'exploitation de plusieurs pages. La plupart du temps, on effectuera une commutation incessante entre les pages 0 et 1. Une nouvelle image (où les sprites auront par exemple progressé) est dessinée dans la page présentement invisible puis celle-ci est amenée à l'écran, et on inverse le rôle des pages pour poursuivre indéfiniment le processus.

La fonction 5 du Bios offre ses services pour la commutation des pages en mode texte, mais en mode X elle ne nous est d'aucune utilité. Les choses restent pourtant simples : comme nous l'avons déjà dit, les quatre pages d'écran se suivent linéairement en mémoire. La page 0 occupe les offsets 0-15999, la page 1 les offsets 16000-31999, etc. Il nous suffit de trouver un moyen de fixer l'adresse où le CRTC commence son affichage. Mais c'est là précisément le rôle des registres 0ch et 0dh du CRTC (Linear Starting Address). Le registre 0ch contient l'octet supérieur et le registre 0dh l'octet inférieur de l'adresse où commence l'image.

D'une manière générale, l'adresse de début d'affichage sera fixée par la procédure SetStart de ModeXlib.asm, qui attend comme argument l'adresse souhaitée. Celle-ci est décomposée en deux octets envoyés dans les registres ad hoc.

La procédure Switch présente une structure un peu particulière. La variable globale vpage y est utilisée pour sauvegarder l'adresse de début de la page couramment invisible. La procédure prend donc également soin de gérer les pages. Vous pouvez systématiquement dessiner votre image dans la page vpage, présentement invisible. L'appel à Switch la rend visible et vpage donne la nouvelle page invisible.

Comme nous l'avons déjà vu, les registres 0ch et 0dh contiennent l'offset du début d'écran et non pas un numéro de page. On peut donc y disposer des valeurs intermédiaires, voir à ce sujet le chapitre 6.3 Défilement tous azimuts.

Le jeux de caractères se trouvant dans PFONT4.GIF

Pour montrer le principe de l'exploitation d'un jeu de caractères, le programme DRAW_FON.PAS présenté ci-dessous ne fait rien d'autre que d'afficher les entrées de clavier en utilisant le jeu PFONT4.GIF. Une chaîne complète est encore affichée en supplément dès que l'utilisateur a frappé la touche d'échappement.

```
Uses Crt,ModeXLib,Gif,Font;

Var Entree:Char;                    {caractère tapé }

Begin
  Init_ModeX;                       {initialise le mode X }
  LoadGif('pfont4');                {charge le jeu de caractères}
  p13_2_ModeX(48000,16000);         {et le copie en page 3 }

  Repeat                            {boucle d'affichage des frappes}
```

```
    Entree:=ReadKey;              {lit un caractère }
    Print_Char(Entree);           {et l'affiche }
  Until Entree=#27;               {jusqu'à ce qu'on tape Esc }

  Print_String('Ceci est un test');{pour terminer affiche toute
                                    une chaîne }
  ReadLn;
TextMode(3);
End.
```

Le jeu de caractères disponible sous forme de fichier GIF est d'abord chargé en page 3. Cette page est située tout à fait à l'"arrière" de la mémoire d'écran. Elle ne sert donc pas dans les animations à deux pages ou en cas de défilement (limité) de la page d'écran. La boucle a pour seule tâche d'afficher le caractère tapé par l'utilisateur. A la fin, le programme affiche encore la chaîne "Ceci est un test".

Les deux procédures centrales Print_Char et Print_String se trouvent dans l'unité FONT.PAS :

```
Unit Font;

Interface

Procedure Print_Char(Chr:Char);
{affiche un caractère en mode X}
Procedure Print_String(Str:String);
{affiche une chaîne de caractères en mode X }

Procedure Scrl_Move;
{déplace la partie visible du Scrolling vers la gauche }
Procedure Scrl_Append;
{ajoute des données nouvelles au bord droit du Scrolling}

Var Scrl_Y:Word;                  {position verticale du Scrolling}

Const
  Scrl_Nombre=4;                  {nombre de chaînes disponibles dans
                                   Scrl_Txt }
  Scrl_Txt:Array [1..Scrl_Nombre] of String =
```

```
{Ceci n'est qu'un texte de démonstration que vous pouvez modifier ou
 compléter librement.!}
  ('Bonjour, ceci est un defileur de demonstration tire du '
    livre P C I N T E R D I T' de Micro Application. Bon '
  +'d accord ce n est pas le grand luxe mais il se programme
   'et sa vitesse est avec un minimum d effort 'bonne.
  +'Meme sur les ordinateurs lents, il est possible d''adjoindre',
   'des effets supplementaires. Notez bien que ce programme '
  +'ne consomme que 10 pour 100 des ressources de calcul '
   'disponibles sur un 486-40 avec Turbo desactive. Attention '
  +'Scrolly va démarrer
  ------------------------------------------------------------');

Implementation
Uses ModeXLib;
Const
  CharPos:Array[' '..'Z', 0..1] of Word= {positions et largeurs des
                                          différents caractères,
                                          décomptés en mots-CPU }

   ((71,4),(0,0),(0,0),(0,0),(0,0),(0,0),
    (0,0),(0,0),(0,0),(0,0),(0,0),(0,0),
    (1906,3),(1909,3),(1912,3),(1915,4),        {,-./}
    (3600,5),(3605,3),(3608,5),(3613,5),        {0..3}
    (3618,5),(3623,5),(3628,5),(3633,5),        {4..7}
    (3638,5),(3643,5),(3648,3),(3651,3),        {8..;}
    (3654,5),(3659,5),(3664,5),(3669,4),        {<..?}
    (0,0),(0,5),(5,5),(10,5),(15,6),(21,5),     {@..E}
    (26,4),(30,7),(37,5),(42,3),(45,4),(49,5),  {F..K}
    (54,4),(58,8),(66,5),(1840,7),(1847,5),     {L..P}
    (1852,7),(1859,5),(1864,4),(1868,4),        {Q..T}
    (1872,5),(1877,6),(1883,8),(1891,5),        {U..X}
    (1896,5),(1901,5));                         {YZ}

Var Cur_X,                      {coordonnées courantes }
    Cur_Y:Integer;              {du curseur }
```

```
    Scrl_Number,                {numéro de la chaîne de défilement
                                 présentement active}
    Scrl_Pos,                   {position à l'intérieur de cette
                                 chaîne }
    Scrl_ChrPos:Word;           {position à l'intérieur du
                                 caractère }

Procedure Print_Char(Chr:Char);
{affiche un caractère en mode X et avance le curseur}
Begin
  Chr:=UpCase(Chr);             {ne prend que des majuscules }
  If Chr in [' '..'Z'] Then Begin{le caractère est-il dans la
                                 table?, oui:}
    If 80- Cur_X <              {reste-t-il assez de place?}
    CharPos[Chr,1] Then Begin
      Cur_X:=0;                 {non, ligne suivante, x à 0}
      Inc(Cur_Y,25);            {y augmenté de la hauteur d'un
                                 caractère}
    End;
 Copy_Block(Cur_Y*80+Cur_X, 48000+Charpos[Chr,0], CharPos[Chr,1],22);
      {copie le caractère de la position de fonte (lue dans la table
       CharPos) à la position du curseur (Cur_Y * 80 byte par ligne
       + Cur_X) (hauteur = 22 lignes }
  Inc(Cur_X,CharPos[Chr,1]);    {avance le curseur de la largeur du
                                 caractère }
  End;
End;

Procedure Print_String(Str:String);
{affiche une chaîne en mode X, utilise Print_Char}
Var i:Word;
Begin
  For i:=1 to Length(Str) do    {parcourt l'ensemble de la chaîne }
    Print_Char(Str[i]);         {et l'envoie caractère par
                                 caractère à Print_Char}
End;
```

```
Procedure Scrl_Move;                {déplace simplement le contenu
                                     de l'image à l'emplacement du
                                     Scrolling d'une position vers
                                     la gauche c'est-à-dire copie
                                     79 bytes de l'abscisse  1 à
                                     l'abscisse 0}
Begin
  Copy_Block(Scrl_y*80, Scrl_Y*80 +1, 79,22);
End;

Procedure Scrl_Append;
Var Chr:Char;                       {caractère courant }
Begin
  Chr:=UpCase(Scrl_txt[Scrl_Number,Scrl_pos]);
                                    {lit le caractère, le convertir
                                     en majuscule }
  If Chr in [' '..'Z'] Then Begin{le caractère est-il dans le
                                     jeu ?, oui:}
    If CharPos[Chr,1] > 0 Then  {n'affiche que les caractères
                                     disponibles }
      Copy_Block(Scrl_y*80+79, 48000+CharPos[Chr,0]+Scrl_ChrPos,
      1, 22);
                                    {puis copie 1 colonne du jeu
                                     de caractères au bord droit
                                     de l'écran }
    Inc(Scrl_ChrPos);               {colonne suivante à l'intérieur
                                     du caractère }
    If Scrl_ChrPos >= CharPos[Chr,1] Then Begin
      Inc(Scrl_Pos);                {caractère traité, au suivant }
      Scrl_ChrPos:=0;               {réinitialise la colonne à 0}
      If Scrl_Pos > Length(Scrl_Txt[Scrl_Number]) Then Begin
        Inc(Scrl_Number);           {chaîne traitée, à la suivante}
        Scrl_Pos:=1;                {remet la position à 0}
        If Scrl_Number > Scrl_Nombre Then Begin
          Scrl_Number:=1;           {texte traité, on recommence }
          Scrl_Pos:=1;
          Scrl_ChrPos:=0;
        End;
```

```
      End;
    End;
  End;
End;

Begin
  Cur_X:=0;                    {curseur au coin supérieur gauche }
  Cur_Y:=0;

  Scrl_Y:=50;                  {Valeur par défaut de l'ordonnée}
  Scrl_Number:=1;              {on commence par la chaîne 1,
                                le caractère 1, la colonne 0 }
  Scrl_Pos:=1;
  Scrl_ChrPos:=0;
End.
```

Nous nous intéresserons d'abord à la table des positions et des largeurs des caractères. De l'espace au Z majuscule, on indique pour chaque caractère l'offset en mémoire d'écran et la largeur en mots-CPU ("bytes"). Il s'agit ici de valeurs décomptées en unités adressables par le processeur, c'est-à-dire des ensembles de quatre octets. L'offset de 71 associé au caractère d'espacement signifie que le caractère commence en ligne 0 à l'abscisse 71*4=284. Par ailleurs, sa largeur est de 4*4 pixels.

Une largeur de 0 indique que le caractère n'existe pas. Dans ce cas, la copie porte sur 0 bytes ce qui équivaut à ne pas représenter le caractère. On laisse ainsi tomber la série des caractères compris entre le point d'interrogation et le signe plus, mais il est facile de les rajouter en cas de besoin.

Les procédures citées utilisent les deux variables Cur_X et Cur_Y qui mémorisent la position courante du "curseur" et indiquent donc les coordonnées d'affichage x et y du prochain caractère.

Print_Char est responsable de l'affichage d'un caractère unique. Ce dernier est d'abord converti en majuscule. Bien entendu, cette conversion devient inutile si on complète la table des caractères avec les minuscules. Si le caractère converti se trouve dans la table, on teste s'il reste assez de place dans la ligne pour l'afficher complètement. Si tel n'est pas le cas, on passe à la ligne suivante.

Les caractères qui ne sont pas présents dans la table (par exemple les caractères accentués) n'apparaissent pas à l'écran mais vous pouvez compléter la table à loisir.

L'instruction Copy_Block ne sert qu'à transférer les données constituant les caractères en page 0. La destination indiquée résulte du calcul cur_Y*80+Cur_X qui donne l'offset de la position courante du curseur. La source est l'offset lu dans la table, augmenté de l'offset de la page 3 du jeu de caractères. La largeur est également lue dans la table tandis que la hauteur est toujours prise égale à 22.

Une fois que la position du curseur a été déplacée de 1 vers la droite, la procédure est achevée.

Print_String ne fait rien d'autre que d'appeler Print_Char pour chaque caractère de la chaîne transmise afin de procéder à son affichage. Le reste de l'unité sert à réaliser le Scrolling qui est expliqué un peu plus loin. Mais voici d'abord la procédure copy_block responsable de la copie des caractères. Elle se trouve dans MODEXLIB.ASM.

```
copy_block proc pascal far destination,source,largeur,hauteur:word
local interligne:word
  mov dx,3ceh                   ;GDC
  mov ax,4105h                  ;mode de lecture 0, mode d'écriture 1
  out dx,ax                     ;émis sur le registre 5 : GDC Mode
  mov dx,3c4h                   ;TS
  mov ax,0f02h                  ;active tous les plans
  out dx,ax                     ;émis sur le registre 2 :
                                ;Write Plane Mask
  push des
  mov ax,0a000h                 ;copie à l'intérieur de VGA
  mov es,ax                     ;-> les deux segments à 0a000h
  mov des,ax
  mov si,source                 ;données d'origine
  mov di,destination            ;copiée en destination
  mov dx,hauteur                ;copie "hauteur" lignes

  mov ax,80                     ;calcule l'intervalle entre
                                ;deux lignes
  sub ax,largeur                ;(= 80-Largeur)
  mov interligne,ax

line_lp:
  mov cx,largeur                ;charge la largeur
  rep movsb                     ;copie une ligne
```

```
    add si,interligne
    add di,interligne

    dec dx                          ;compteur de lignes
    jne line_lp

    pop des
    ret
copy_block endp
```

Cette procédure commence par activer sur le registre 5 du GDC le mode de lecture 0 et le mode d'écriture 1 qui permettent de copier rapidement des ensembles de quatre pixels. Sur le registre 2 du TS, l'intégralité des plans est activée.

On fait alors pointer des:si sur le coin supérieur gauche du bloc à copier et es:di sur les coordonnées de destination. Dans les deux cas, les coordonnées sont transmises sous forme d'offsets prêts à être chargés dans les registres appropriés. Comme compteur de lignes, on exploite dx, tandis que cx est chargé avec la largeur du caractère.

A la fin de chaque ligne copiée, il faut se rendre au début de la suivante. L'intervalle correspondant est calculé : il est de 80 bytes entre deux lignes, la distance à parcourir correspond donc à ce nombre diminué de la largeur du bloc. Le reste de la procédure est constitué des deux boucles imbriquées traitant les coordonnées x et y (rep movsb).

Mais en réalité, vous ne souhaitez pas seulement recopier des caractères statiques sur l'écran. L'objet de ce chapitre était de réaliser un Scrolling. Les autres procédures et variables de l'unité FONT.PAS servent à cette fin.

Comme Pascal ne sait pas gérer les chaînes de plus de 255 caractères, les textes plus longs doivent être répartis sur plusieurs Strings. Le nombre de ces chaînes partielles est mémorisé dans la constante Scrl_Nombre. Le texte proprement dit se trouve dans la constante typée Scrl_Txt.

De nouvelles variables apparaissent : Sclr_Number, Scrl_Pos et Scrl_ChrPos. La première retient le numéro de la chaîne partielle présentement active. La seconde contient le numéro du caractère courant à l'intérieur de cette chaîne et la dernière mémorise la colonne en cours à l'intérieur de ce caractère. La plupart des caractères s'étendent sur plusieurs colonnes et doivent donc être portés à l'écran en plusieurs étapes.

La première chose que fait la boucle du scrolling est de déplacer d'une position vers la gauche les caractères déjà visibles. Elle utilise à cet effet la procédure Copy_block qui décale le contenu du scrolling de l'abscisse 1 à l'abscisse 0.

Scrl_Append ajoute finalement les nouvelles données au bord droit de l'écran. Le caractère courant (majuscule) est à nouveau lu dans le tableau et confronté à la table des caractères existants. Nous avons vu que certains caractères tout en étant disponibles ne sont pas définis : il faut donc détecter à l'étape suivante si leur largeur est nulle pour qu'ils ne tombent pas à l'eau.

Les caractères à la fois définis et disponibles sont recopiés par Copy_Block au bord droit de l'écran (Scrl_y*80+79 donc abscisse 79) en tenant compte de la colonne courante à l'intérieur du caractère (Scrl_Pos). Cette colonne est incrémentée. Si le caractère est entièrement visible, le compteur de colonnes est remis à 0 et le compteur de caractères incrémenté. Ce dernier est remis à 0 si on se trouve à la fin de la chaîne, en même temps le compteur de chaînes est incrémenté. Lorsque ce dernier est en fin de parcours, on recommence au début du texte en réinitialisant les variables appropriées.

Le fichier SCROLLY.PAS pilote un programme de démonstration :

```
Uses Crt,Tools,ModeXLib,Gif,Font;

Var Sinus:Array[0..127] of Word; {table de sinus pour oscillations
                                  verticales }
    t:Word;                      {"temps", paramètre de position
                                  à l'intérieur du sinus}

Begin
  Init_ModeX;                    {active le mode X }
  LoadGif('pfont4');             {charge le jeu de caractères }
  p13_2_ModeX(48000,16000);      {et le transfère en page 3 }
  Sin_Gen(Sinus,128,Scrl_y div 2,Scrl_y div 2);
                                 {prépare la table des sinus pour
                                  déplacement vertical }
  t:=0;                          {au commencement était le temps 0 }
  Repeat
    WaitRetrace;                 {synchronisation}
    Scrl_Move;                   {déplace la partie visible vers
                                  la droite }
    Scrl_Append;                 {ajoute une nouvelle colonne
                                  à droite }
```

```
    SetStart(Sinus[t and 127]*80);{assure le mouvement vertical}
    Inc(t);                       {progresse dans la table des sinus}
  Until KeyPressed;
TextMode(3);
End.
```

On commence par charger le jeu de caractères utilisé par la suite pour afficher les caractères du scrolling. La boucle principale attend d'abord le retour de balayage, puis déplace les données visibles d'une colonne (4 pixels) vers la gauche, et ajoute à droite de nouvelles données par Scrl_Append.

En même temps, on introduit un déplacement vertical en décalant vers le haut et vers le bas le début de l'écran, grâce à une table de sinus. Ce processus est guidé par la variable t qui mémorise la position courante à l'intérieur de la table des sinus.

5.6. Adaptation du chargeur de GIF au mode X

Comme nous l'avons déjà vu, le format GIF est le format graphique le plus adapté aux démos. Il nous faut donc également en mode X la possibilité de lire et d'afficher des images GIF. Ce n'est pas un problème de taille car le chargeur est a priori conçu pour transférer l'image en mémoire centrale. De là, elle est copiée en mode 13h dans la mémoire d'écran par l'instruction Pascal Move. C'est cette dernière étape, à savoir la copie, qu'il convient de réécrire pour tenir compte de la nouvelle organisation de la mémoire en mode X.

Le rôle est dévolu à la procédure p13_2_ModeX dans MODEXLIB.ASM. Elle copie une image de taille pic_size*4 depuis la mémoire centrale (pointeur en Vscreen) vers l'adresse de début start du mode X. Ainsi à titre d'exemple, pour copier en page 2 une image chargée par LoadGIF avec 320x200 pixels, l'appel serait :

```
p13_2_modex(2*16000,64000 div 4);
```

Le problème posé par une telle procédure tient à la répartition de l'image dans les différents plans. La manière la plus simple mais aussi la plus lente de le résoudre consiste à opérer pixel par pixel : on lit un pixel, on sélectionne le plan de destination, on y réécrit le pixel. Mais il est plus élégant de copier d'abord tous les points du plan 0 (qui se trouvent aux offsets 0, 4, 8, de l'image source), puis les points du plan 1 et ainsi de suite. On évite ainsi de solliciter trop souvent le séquenceur de timing.

```
p13_2_modex proc pascal start,pic_size:word
    mov dx,03ceh                    ;fixe Write Mode 0
    mov ax,4005h                    ;sous le registre GDC5 (GDC Mode)
    out dx,ax
    mov b plan_l,1                  ;mémorise le masque de plan
    push des
    lds si,dword ptr des:vscreen ;charge l'adresse source
    mov w plan_pos,si               ;et la sauvegarde
    mov ax,0a000h                   ;fixe l'adresse de destination
    mov es,ax
    mov di,start
    mov cx,pic_size                 ;lit le nombre de pixels
@lpplan:
    mov al,02h                      ;dans le registre 2 du TS...
                                    ;(Write Plane Mask)
    mov ah,b plan_l                 ;masque le plan
    mov dx,3c4h
    out dx,ax

  @lp1:
    movsb                           ;copie un byte
    add si,3                        ;prochain byte source
    loop @lp1

    mov di,start                    ;relit l'adresse de destination
    inc w plan_pos                  ;nouvelle adresse source
    mov si,w plan_pos
    mov cx,pic_size                 ;lit la taille
    shl b plan_l,1                  ;masque le plan suivant
    cmp b plan_l,10h                ;les 4 plans sont copiés ?
    jne @lpplan

    pop des
    ret
Endp
```

La variable plan_l qui contient le masque de plan courant est d'abord initialisée avec la valeur 1 pour adresser le plan 0. Les registres des:si pointent sur les données source dont l'offset est sauvegardé en plan_pos. Après

chargement de l'adresse de destination en es:di et de la taille de l'image en cx, la boucle des plans @lpplan sélectionne d'abord le bon plan en appliquant le masque courant au registre 2 du timing sequencer (Write Plane Pask). La boucle @lp1 copie un octet à chaque passage et saute ensuite trois octets plus loin dans l'image source (movsb a déjà incrémenté si de 1) pour pouvoir lire l'octet suivant du même plan. Quand cette boucle est achevée, l'un des plans se trouve déjà casé au bon endroit. Pour copier le plan suivant, le pointeur de destination est remis au début, l'adresse de destination augmentée de 1 pour engager par exemple les offsets 1, 5, 9 s'il s'agit du plan 1. Cx est rechargé et le nouveau plan est soumis à un masquage par rotation de 1 vers la gauche. La condition de fin de boucle porte sur le masque du cinquième plan qui n'est pas disponible, lorsque les quatre plans souhaités ont été copiés.

A l'issue de ce processus, une page d'écran de la mémoire centrale déposée dans le format du mode 13h se trouve reproduite dans l'écran en mode X. De là, elle pourra être transférée sur n'importe quelle page d'écran, par une copie très rapide. On pourra ainsi effacer une image périmée ou restaurer un fond d'écran.

La méthode de copie plan par plan sera aussi utilisée pour afficher les sprites, comme nous le verrons au chapitre 7. Mais comme nous serons dans une situation complètement différente, il faudra mettre en service une procédure toute autre.

5.7. Un simple scrolling ou texte défilant

Quiconque a déjà passé du temps en compagnie d'un C-64, d'un ST ou d'une autre machine de ce genre s'en souvient encore : les scrollings étaient des textes défilants qui donnaient des indications souvent très intéressantes, à lire impérativement. Aujourd'hui, le contenu ne joue plus qu'un rôle secondaire, la présentation exploitant davantage de ressources et d'effets. Les principes de base demeurent cependant valables et, pour les comprendre, le mieux est de les appliquer en réalisant un scrolling simple et classique.

Un scrolling en mode X

La fonction d'un scrolling est essentiellement de faire défiler un texte sur l'écran de droite à gauche, de façon à le faire passer devant les yeux de l'observateur. En mode texte, la programmation de ce genre de chose ne prend que quelques minutes : on crée une chaîne de caractères et un pointeur qui retient la position courante du bord gauche de l'écran dans la chaîne. En partant de la position courante, on écrit par exemple 70 fois par seconde 80 caractères dans une ligne d'écran donnée et on incrémente le pointeur d'une

position. La fenêtre constituée par la partie visible du texte se déplace par-dessus la chaîne dans la direction de la lecture et on dispose d'un scrolling.

En mode graphique, la programmation est un peu plus difficile. Si le principe reste le même, le programmeur ne dispose que de routines du Bios fort rares (en mode 13h) voire inexistantes (en mode X) pour afficher des caractères. Une fois de plus, il faut faire tout soi-même, mais de toute façon l'obtention d'une bonne vitesse d'exécution exige qu'on mette soi-même la main à la pâte.

Par rapport au scrolling en mode texte, il faut prendre en compte les contraintes du déplacement du contenu de l'écran en mode graphique. En mode texte, nous pouvions nous permettre de réécrire complètement la chaîne de caractère à partir de la nouvelle position courante, pendant le retour de balayage. En mode graphique, le facteur vitesse se fait plus contraignant, et on ne peut plus commencer par reconstituer le caractère à partir de son jeu d'origine.

En fait, comme le texte qui se trouve sur l'écran ne change plus (dans cette version simple de scrolling) et se contente de défiler, on peut le déplacer graphiquement d'une position. Il suffit de décaler les colonnes 4-319 de 4 pixels vers la gauche et de disposer dans les colonnes libérées 316-319 une partie calculée supplémentaire. Les quatre pixels de décalage autorisent une exploitation efficace des avantages du mode X. En effet, dans le mode d'écriture 1, on peut copier facilement des blocs de quatre octets.

Si on avait choisi un déplacement élémentaire qui ne soit pas un multiple de 4 (par exemple trois), il aurait fallu copier des contenus d'un plan à l'autre, ce qui n'est possible qu'en accédant individuellement aux différents pixels. A l'inverse, une copie d'un offset à l'autre sans changement de plan s'applique à quatre points à la fois, ce qui est évidemment intéressant sur le plan de la vitesse.

Jeu de caractères pour texte défilant

Au bord droit de l'écran, il s'agit de représenter une bande de quatre pixels de largeur avec une nouvelle fraction du texte. Les données à représenter seront issues d'un jeu de caractères. Mais où le trouver ? On pourrait exploiter le jeu de caractères contenu dans la ROM de la carte VGA mais il n'est pas en couleur et son design n'est pas renversant. Dessiner son propre jeu de caractères n'est pas du goût de tout le monde, mais constitue un préalable inévitable pour une présentation tant soit peu ambitieuse.

Beaucoup de développeurs prennent la peine de programmer à cet effet un éditeur personnel qui possède tous les raffinements d'un logiciel de dessin avec la capacité de générer les jeux de caractères les plus fous. Mais les moyens à mettre en jeu pour un tel développement sont un peu démesurés par rapport au but poursuivi. Est-il raisonnable d'écrire un éditeur en y travaillant

d'arrache-pied pendant des semaines voire des mois alors que la démo, à laquelle est destinée le jeu de caractères, est achevée en moitié moins de temps ? Pourquoi un programmeur se croirait-il obligé de réinventer la roue en développant les fonctions graphiques de base (segments, cercles, polygones) que maîtrise à la perfection n'importe quel programme de dessin du domaine public ?

La solution la plus simple se trouve effectivement dans ces programmes de dessin. En stockant un alphabet complet dans une seule image, nous pourrons aussi économiser de l'espace disque, étant entendu que nous choisirons le format GIF.

A chacun d'imaginer les formes de caractères qu'il souhaite. Si vous êtes talentueux et doué en dessin, vous pourrez tirer du néant une fonte toute nouvelle, mais il est plus facile de partir d'une fonte existante fournie et de la modifier. Il faut veiller à ce que les caractères soient disposés convenablement pour permettre leur exploitation ultérieure. Comme la routine qui ajoute de nouvelles colonnes fonctionne à base de quartets, il faudra adapter la disposition des caractères en conséquence. L'abscisse et la largeur de chaque caractère devront être divisibles par 4 mais on peut utiliser des largeurs différentes d'un caractère à l'autre : les fontes proportionnelles sont donc possibles.

Une fois les caractères mis en place, il faut procéder à la tâche fastidieuse qui consiste à saisir leurs coordonnées. Ces dernières seront exploitées plus tard par la procédure d'affichage qui lira à la fois la position de chaque caractère dans la mémoire d'écran et sa largeur dans une table.

Pour réaliser la table, on prend un papier, un stylo et un programme de dessin qui donne en permanence la position du curseur.

Attention : certains programmes de dessin fixent l'origine des coordonnées en (1,1) pour le coin supérieur gauche de l'écran. Peut-être les développeurs de ces programmes ont-ils jugé que c'était plus ergonomique, mais ce choix contredit les réalités mathématiques et les habitudes informatiques. Les coordonnées doivent se référer à l'origine (0,0). On sera donc éventuellement amené à retrancher 1 des chiffres lus (également lors du positionnement des quartets). A partir des coordonnées du coin supérieur gauche de l'écran, on pourra calculer l'offset ultérieur dans la mémoire d'écran au moyen de la formule déjà évoquée :

```
Offset:= Y*80 +  X div 4
```

Cette valeur est à noter dans la table pour chaque caractère. Le plan est toujours le plan 0, ce qui constituait bien la condition nécessaire à l'exploitation du mode d'écriture rapide. Pour déterminer la largeur, on la mesure d'abord en pixels (de tête ou avec une fonction d'évaluation de longueur du

programme de dessin) et on l'arrondit par excès au multiple de 4 le plus proche pour obtenir un décompte en quartets car comme vous le savez ce processus fonctionne par blocs de quatre pixels. La largeur est également reportée dans la table.

Une fois cette corvée menée à bien, il reste à transférer la table dans le programme. Pour simplifier, nous conserverons l'ordre ASCII. On prendra par exemple les codes ASCII de 32 (espace) à 96 (accent grave), les caractères non disponibles (par exemple l'arrobas @ de code ASCII 64) étant associés à une largeur de 0. Si on l'estime nécessaire, on pourra évidemment intégrer les minuscules (97-122) ou même la totalité de la table ASCII en associant librement à chaque caractère un petit graphique. Mais ce travail est assez long et rarement justifié : la plupart du temps, les majuscules et les chiffres auxquels on ajoute quelques caractères spéciaux suffisent amplement.

Vous trouverez bien entendu sur le CD d'accompagnement une petite fonte de démonstration, qui est exploitée par les programmes imprimés ici.

6. Ecran fractionné et autres effets brûlants

Que serait la programmation graphique sans effets spéciaux ? On pourrait regarder de belles images, peut-être en dessiner d'autres. Mais la moindre animation et le plus petit sprite constituent déjà des effets graphiques. A vrai dire, nous ne parlerons pas ici de ce type d'effet, qui nécessite avant tout des calculs de la part du processeur. Nous allons nous intéresser aux effets traités par la seule carte VGA, le processeur n'intervenant que pour déclencher la commande.

6.1. Les bases

Tous les effets traités pourraient également être pris en charge par le processeur mais il faudrait alors déplacer d'énormes quantités de données dans la mémoire d'écran. Par nature, les transferts de mémoire sont lents, ils ne peuvent être que modérément accélérés par l'accès DMA (sur les ordinateurs récents, l'accès direct par processeur est même plus rapide) et ils se heurtent inévitablement au goulet d'étranglement constitué par le bus de données.

Sur l'ancien bus ISA, les vitesses de transfert exigées n'étaient pas envisageables car pour des raisons de "compatibilité descendante", même les ordinateurs les plus rapides n'étaient exploités qu'à la cadence de bus de 8 Mhz. En cas de présence d'une carte Hercules supplémentaire, la carte VGA 16 bits était rabaissée au niveau d'une 8 bits, ce qui doublait encore une fois le temps d'accès.

Les systèmes de bus modernes travaillent sur 16 et même 32 bits, à la cadence minimale de 33 Mhz. Mais ils sont toujours obligés de calmer les données en les soumettant à des protocoles rigoureux, ajoutant de ci de là quantité de cycles d'attente. Certaines cartes VGA n'arrivent pas à suivre et ajoutent de leur côté des cycles d'attente supplémentaires.

Il faut donc chercher un autre moyen de susciter des effets spéciaux. D'innombrables démos attestent qu'il est possible de déplacer et de déformer des objets tridimensionnels tout en effectuant des calculs compliqués et en jouant de la musique. C'est la carte qui permet d'atteindre ce résultat.

Certes, les cartes graphiques du PC ne disposent pas encore de contrôleurs programmables ou de processeurs spécialisés conçus pour les effets graphiques, comme il en existe sur l'Amiga. Dans ce cas, le processeur se limite à des tâches générales de contrôle et de surveillance. Et dans le nombre inépuisable des registres des circuits associés et l'infinie faculté de combiner leurs contenus se cachent des potentialités auxquelles le programmeur confronté à la carte VGA ne peut que rêver.

Il faudra veiller à se limiter aux registres compatibles avec la carte VGA originale. Adapter les manipulations de registres spécifiques à un fabricant sera toujours un exercice difficile voire impossible : sur la plate forme de destination, le registre recherché peut tout simplement briller par son absence.

Si on recherche de nouvelles idées d'exploitation des registres, on peut toujours essayer de deviner, à l'aide d'une description complète de leurs propriétés, quel sera l'effet de telle ou telle modification de contenu, mais il sera difficile de se soustraire à l'expérimentation directe.

Tant que vous ne toucherez pas aux registres de timing du CRTC, vos expériences ne seront pas dangereuses et ne sauraient porter préjudice à votre matériel. Dans le pire des cas, vous obtiendrez une image confuse, déformée ou même plus d'image du tout. Vous appuierez alors sur la touche d'échappement **ESC** pour revenir à l'éditeur de Pascal.

Ce sont les registres 0-7 du CRTC qui peuvent se révéler dangereux à manipuler. En charge du timing horizontal, ils peuvent fusiller votre moniteur ou votre carte graphique si vous leur demandez n'importe quoi. Si le timing se trouve bouleversé, le rayon cathodique peut se mettre à parcourir votre écran de façon incontrôlée, avec des fréquences excessives qui provoquent la surchauffe du moniteur et son décès s'il n'est pas très robuste.

C'est pourquoi d'ailleurs ces registres sont protégés par le bit 7 du registre 11h du CRTC. Avant toute manipulation, il faut mettre à 0 ce bit. En cas d'erreur et si on joue avec la vie de son moniteur, le pauvre instrument manifeste la plupart du temps son agonie par des scintillements désagréables et un horrible sifflement. Lorsqu'on procède à des expériences qui risquent de mal se terminer, il faut donc garder le doigt sur l'interrupteur d'arrêt de façon à pouvoir couper le courant de toute urgence.

En règle générale, il est recommandé de ne pas prendre ce risque et de ne pas manipuler ces registres. Pour ne rien vous cacher, je n'ai jamais obtenu d'effet

digne d'être retenu en jouant sur le timing, la plupart du temps les images sont affreuses et dénuées d'intérêt.

Mais cet avertissement n'est pas valable pour les autres registres et ne devrait pas vous empêcher de les tester. Comme nous le verrons, vous pouvez découvrir des propriétés intéressantes qui n'étaient pas prévues au départ.

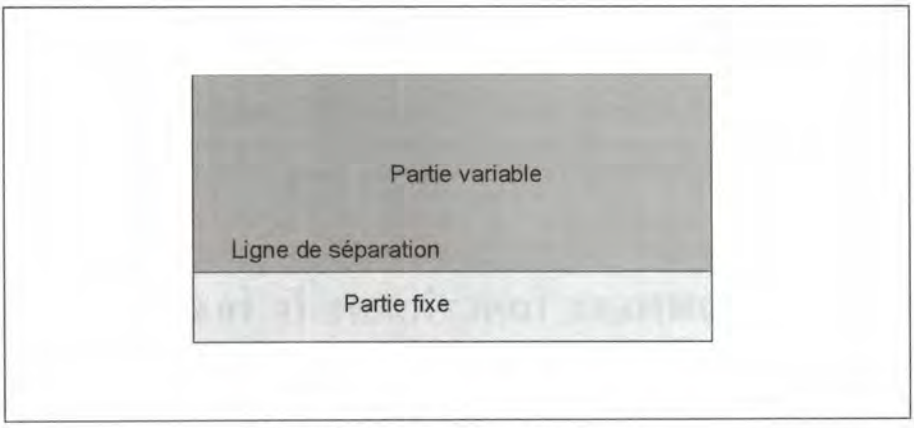
6.2. Comment fonctionne le fractionnement d'écran

Il existe bien des situations où il faut afficher à l'écran deux zones indépendantes, par exemple une zone de défilement dans la partie supérieure de l'écran et une ligne d'état fixe tout en bas. Bien entendu, vous pouvez confier cette tâche au processeur. Mais si vous vous souciez quelque peu de la vitesse d'exécution, vous serez amené à vous pencher sur l'opportunité de confier cette tâche au matériel, c'est-à-dire de la refiler à la carte VGA.

C'est le registre Line Compare -ou Split Screen- qui va nous montrer le chemin. Les fonctions offertes ne sont pas aussi puissantes que sur les ordinateurs familiaux mais elles suffisent à la plupart des applications.

Ce registre fixe la ligne de fractionnement de l'écran en créant deux zones : la supérieure est mobile, la zone inférieure statique. En mode 200 lignes, la valeur indiquée est à dédoubler.

Le fonctionnement de ce registre est très simple : pendant la génération de l'image, la carte VGA suit constamment le numéro de la ligne courante. Le numéro en question est le numéro physique qui va de 0 à 400 même dans les modes à 200 lignes comme le mode 13h ou le mode X. Il se trouve dans un registre malheureusement inaccessible à la lecture directe sur un port, mais il est exploité à titre interne pour comparaison avec le registre Line Compare (CRTC, index 18h). S'il atteint la valeur définie dans ce registre, le compteur d'adresses est initialisé à 0, ce qui veut dire que l'affichage dans cette ligne commence aux données situées à l'offset 0 de la mémoire d'écran.



Organisation d'un écran fractionné

En fait, le numéro d'une ligne ne tient pas dans 8 bits. La carte VGA ne connaît aucun mode où le nombre de lignes est inférieur à 350, même les modes à 200 lignes sont produits par le moyen de 400 lignes physiques. Par conséquent, le registre Line Compare est en réalité réparti sur 3 registres (et même sur 4 dans les cartes SuperVGA). Il existe d'autres registres (surtout ceux qui concernent le timing vertical) qui dépassent 8 bits. Les bits en "excès" sont regroupés dans deux registres supplémentaires (trois sur la carte SuperVGA).

Le premier de ces registres est fort judicieusement appelé registre d'overflow. Il contient des bits liés aux registres de timing vertical. On y trouve notamment en position binaire 4 le bit 8 du registre Line Compare (c'est le seul bit ici présent qui ne soit pas protégé par le bit de protection du registre 11h). Le bit 9 du registre Line Compare se trouve dans le bit 6 du deuxième registre de débordement appelé Maximum Row : il n'est pas utilisé dans les résolutions évoquées ici. Pour définir le contenu de Line Compare, il faut donc répartir sur trois registres les bits du numéro de ligne, en instituant décalages et masquages appropriés. Cette tâche est effectuée par la procédure Split, que vous trouverez dans MODEXLIB.ASM :

```
Split proc pascal far row:byte  ;fractionne l'écran à la ligne row
 mov bl,row
 xor bh,bh
 shl bx,1                       ;*2 à cause du dédoublement des
                                ;lignes
 mov cx,bx

 mov dx,3d4h                    ;registre 7 du CRTC
 mov al,07h                     ;(Overflow low)
 out dx,al
```

```
    inc dx
    in al,dx
    and al,11101111b                ;met le bit 8 de la ligne dans
                                    ;le bit 4
    shr cx,4
    and cl,16
    or al,cl
    out dx,al                       ;écriture

    dec dx
    mov al,09h                      ;registre 9 (Maximum Row Adress)
    out dx,al
    inc dx
    in al,dx
    and al,10111111b                ;charge le bit 6 avec le bit 9 de
                                    ;la ligne
    shr bl,3
    and bl,64
    or al,bl
    out dx,al                       ;écriture

    dec dx
    mov al,18h                      ;registre 18h
                                    ;(Line Compare/Split Screen)
    mov ah,row                      ;fixe les 8 autres bits
    shl ah,1
    out dx,ax
    ret
  Endp
```

Cette procédure commence par multiplier par deux l'argument reçu, pour que l'intervalle des ordonnées aille de 0 à 199 malgré le double scan appliqué au mode 200 lignes. Après sélection du registre d'overflow (registre 7 du CRTC), on efface le bit 4 pour pouvoir y introduire le bit 8 du numéro de ligne (par masquage et décalage à droite de 4 bits). De la même façon, on copie le bit 9 du numéro de ligne dans le bit 6 du registre Maximum Row Address (registre 9 du CRTC). Pour terminer, on transfère les 8 bits restants du numéro de ligne dans le registre 18h (Line Compare ou Split Screen).

Si, à l'aide des registres Linear Starting Address 0ch et 0dh, on reporte plus loin le début de l'écran, par exemple jusqu'à l'entrée de la page 2, cette zone

est affichée dans la partie supérieure de l'écran, tandis que la partie inférieure, au-dessous de la ligne indiquée dans le registre Line Compare, présente le contenu du début de la mémoire d'écran.

On peut donc déplacer librement la zone supérieure en jouant sur le registre Linear Starting Address : elle peut défiler dans toutes les directions (voir chapitre suivant) cependant que la zone inférieure conserve un aspect constant.

Par rapport aux extraordinaires possibilités de fractionnement d'écran des ordinateurs de jeu, ce processus est plutôt restrictif mais il est suffisant pour donner des effets intéressants.

Au-delà du simple partage de l'écran en deux zones, on peut aussi imaginer d'autres effets. Que se passe-t-il par exemple si on supprime la division statique en une partie mobile et une partie fixe ? La ligne de partage peut être fixée librement et se prête à une modification continue à l'intérieur d'une boucle. Cette ligne, rappelons-le, commence toujours à l'offset 0 de la mémoire d'écran, en conséquence l'intégralité de la zone inférieure de l'écran se déplace si on altère le registre Line Compare. Une soustraction de 2 (double scan !) remonte l'écran inférieur d'une ligne graphique de sorte que tout en bas apparaît une nouvelle ligne.

On peut ainsi faire monter une image par-dessus un fond d'écran : par exemple, pour introduire une nouvelle partie d'une démo. On charge l'image qui constitue le fond d'écran en page 1 (par LoadGIF et p13_2_ModeX) et l'image à superposer en page 0. Puis on active le fractionnement de l'écran, d'abord avec la ligne 400, de sorte que l'image supérieure reste invisible et qu'on ne voie que le fond d'écran.

Si on déplace progressivement la ligne de fractionnement vers le haut (décrémentation de 2 du registre Line Compare), le contenu de la page 0 remonte en glissant par-dessus le fond d'écran, jusqu'à atteindre le bord supérieur (Line Compare 0). On peut alors commuter tranquillement la page 0 et désactiver le fractionnement. Il faut faire ces opérations dans l'ordre indiqué sinon l'ancien fond resurgit un court instant car la désactivation du fractionnement envoie sur l'écran la page courante, qui à ce moment-là doit déjà être la page 0.

Inversement, il est également possible de faire descendre une image de l'écran, de façon à faire apparaître, comme dans un jeu de cartes, l'image située en dessous. Il s'agit là d'un bel effet pour un diaporama (présentation programmée d'une séquence d'images toutes prêtes, par exemple dans une vitrine). A cet effet, on met à nouveau le premier plan en page 0 et le fond d'écran en page 1. On active d'abord le fractionnement, on appelle la page 1 et on augmente à l'intérieur d'une boucle la valeur du registre Line Compare. La ligne de fractionnement se déplace alors vers le bas. Au-dessus de cette ligne,

le contenu de la page 1 apparaît sous forme de fond d'écran (la page 1 est la page active à ce moment).

Comme pour la plupart des autres effets, l'incrémentation et la décrémentation du registre Line Compare doivent en principe être synchronisées avec le signal de retour de balayage : on évite ainsi des irrégularités et des scintillements dans l'affichage. Il suffit à cet effet d'insérer dans la boucle la commande WaitRetrace, ce qui contribue à l'indispensable ralentissement du processus dans un contexte où le processeur est inactif et capable d'avaler l'ensemble du traitement en une fraction de seconde. La synchronisation ramène la cadence à 70 images par seconde (fréquence de rafraîchissement du mode 13h et du mode X). L'ensemble de l'opération dure donc 2,8 secondes (200 lignes / 70 lignes par seconde).

Si vous faites des expériences avec ces effets, vous constaterez bien vite que l'une des deux images prend des teintes bizarres. Cette altération est due à la limite des 256 couleurs imposées par la palette en mode graphique. Au moment de son chargement, la première image remplit la palette. Lorsqu'arrive la deuxième image, elle remplit à son tour la palette en écrasant les anciennes couleurs : une palette ne peut gérer que 256 couleurs en même temps. La seule issue est d'utiliser la même palette pour les deux images.

Si on dessine soi-même les images, on peut veiller à utiliser la même palette pour les deux (les logiciels de dessin permettent généralement de la charger et de la mémoriser à part). Mais si on utilise des images toutes faites issues de collections ou numérisées par un scanner, il faut regrouper les deux palettes en une seule. Beaucoup de programmes de conversion graphique maîtrisent cette opération. Ils pratiquent généralement un "dithering" médiocre qui conduit à des colorations erronées et des pixels mal placés.

Il est alors préférable de réduire à 128 le nombre de couleurs des deux images. Si l'une des images est complexe et l'autre beaucoup plus simple, on peut aussi choisir 192 et 64 couleurs, l'essentiel est que la somme donne 256. Avec un programme de conversion graphique, on peut alors sans dithering recomposer une palette complète.

Pendant la démo, on peut charger les deux images dans n'importe quel ordre, puisque les palettes sont identiques.

Dans cet effet, il peut être intéressant de modifier le registre Line Compare par pas de 1. Ce sont alors des lignes de balayage simples, représentant des demi- lignes graphiques, qui forment l'unité élémentaire de déplacement, de sorte que le processus dure deux fois plus longtemps. Dans le cadre d'un mouvement continu, l'oeil ne peut pas distinguer une demi-ligne. Mais s'il s'agit d'un simple fractionnement d'écran, l'apparition au milieu de l'image d'une demi-ligne graphique est dérangeante. Il faut alors utiliser des nombres impairs (l'affichage de la deuxième image commence à la ligne qui suit

immédiatement celle dont le numéro est indiqué dans le registre Line Compare. La ligne de début devant être paire, le registre doit présenter un nombre impair).

Le petit programme de démonstration qui suit est destiné à présenter une application de la procédure de fractionnement. Il ne fait rien d'autre que de faire glisser l'une sur l'autre deux pages qui par simplification sont uniformément colorées. On commence par remplir les deux pages d'écran et la page 1 (adresse de début : 16000) est activée. Les boucles présentes se contentent d'incrémenter et de décrémenter le registre Line Compare, ce qui provoque le déplacement recherché (couverture puis séparation). L'ensemble du processus tient compte du signal de retour de balayage. La lecture du clavier est combinée avec une instruction exit qui contrevient aux principes de la programmation structurée et ne manquera pas d'horrifier l'un ou l'autre informaticien. Mais à quoi bon se compliquer la vie (et la ralentir) en introduisant des variables encombrantes destinées simplement à gérer la fin de la boucle ?

```
{SPLIT.PAS}
Uses Crt,Gif,ModeXLib;
Var i:Word;
begin
  Init_ModeX;
  LoadGif('boule');                {initialisation du mode x}
  p13_2_ModeX(16000,16000);
  LoadGif('coin');                 {chargement de la 2ème image}
  p13_2_modex(0,16000);
  SetStart(16000);                 {affiche l'image de fond}
  Repeat
    For i:=200 downto 0 do Begin {déplacement vers le haut}
      WaitRetrace;
      Split(i);
      If KeyPressed Then Exit;
    End;
    For i:=0 to 200 do Begin       {déplacement ver le bas}
      WaitRetrace;
      Split(i);
      If KeyPressed Then Exit;
    End;
  Until KeyPressed;
TextMode(3);
End.
```

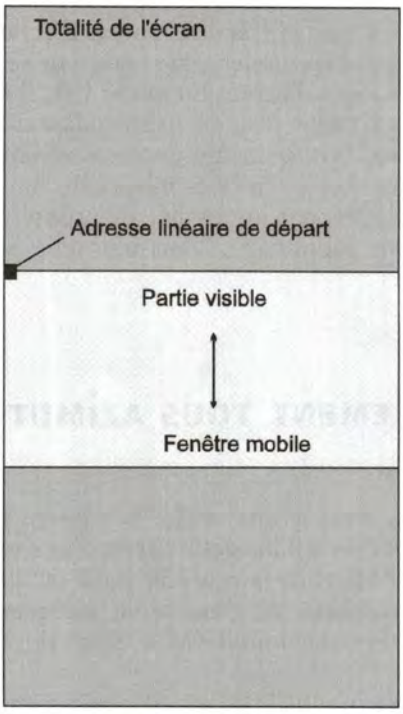
Notez bien que le fractionnement d'écran fonctionne dans tous les modes graphiques et les modes texte mais il ne se justifie vraiment qu'en présence de plusieurs pages d'écran. En mode 13h, il ne reste pas assez de mémoire pour gérer deux pages d'écran indépendantes. Un écran fractionné représentera donc deux fois le même contenu réparti simplement dans deux fenêtres différentes. Les remarques faites plus haut se rapportent en fait au mode X même s'il n'est pas impossible de produire un écran fractionné dans le cadre d'un mode monopage. Mais une telle démarche présente en principe peu d'intérêt.

6.3. DÉFILEMENT TOUS AZIMUTS

Jusqu'ici, nous avons utilisé le registre Linear Starting address du CRTC (registre N° 0ch/0dh) pour passer d'une page à une autre. Nous calculions à cet effet l'offset de la nouvelle page (n° de page * 16000) pour l'écrire dans le registre. Mais on peut aussi indiquer des valeurs intermédiaires pour déplacer librement en RAM le début de la zone affichée.

Lorsqu'une nouvelle image est générée par le rayon cathodique, ce dernier ne tire pas les données du début de la mémoire d'écran mais de l'endroit spécifié dans le registre Linear Starting Address. L'écran physique, c'est-à-dire la zone finalement affichée à l'écran, peut ainsi se déplacer à la façon d'une fenêtre par-dessus toute la mémoire vive de la carte VGA : on verra toujours une section au choix d'une image de 256 Ko.

Si on déplace ce début d'écran à l'intérieur de la mémoire vive, la fenêtre se meut dans la direction correspondante de sorte que le graphique à l'écran part dans la direction opposée. En incrémentant le registre Linear Starting Address de 80, on déplace le contenu de l'écran d'une ligne vers le haut (320 pixels / 4 plans = 80 octets). Le contenu de la ligne 0 se trouve en dehors de la fenêtre et la ligne 200 apparaît tout en bas On voit alors les lignes 1-200. En continuant d'incrémenter le registre par pas de 80, on obtient un défilement vertical de l'ensemble de l'écran, tout en consommant un minimum de ressources de calcul.



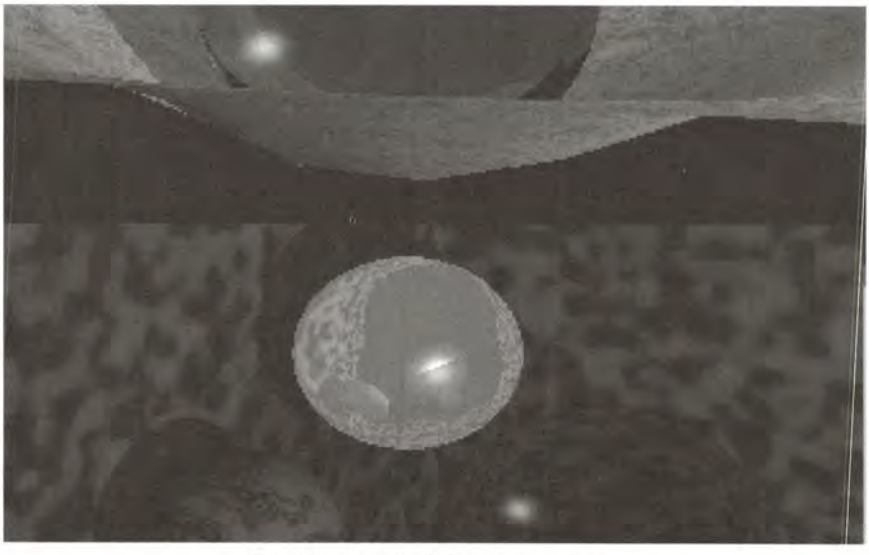
Structure de la mémoire d'écran avec quatre pages verticales

Le programme de démonstration suivant illustre exactement ces propos :

```
{SCROLLV.PAS}
Uses Crt,Gif,ModeXLib;
Var y,                      {valeur courante de la Linear Start.
                             Address}
    y_dir:word;             {direction du défilement }
Begin
  Init_ModeX;               {active le mode X }
  LoadGif('boule');         {charge la première image en page 0 et 2 }
  p13_2_ModeX(0,16000);
  p13_2_ModeX(32000,16000);
  LoadGif('coin');          {charge la deuxième image en page 1 et 3 }
  p13_2_ModeX(16000,16000);
  p13_2_ModeX(48000,16000);
  y:=80;                    {on commence en ligne 1}
  y_dir:=80;                {direction de déplacement + 80 octets par
                             itération }
```

```
  Repeat
    Inc(y,y_dir);           {déplacement }
    WaitRetrace;            {attend le retour de balayage }
    SetStart(y);            {écrit la nouvelle valeur de début dans
                             le registre}
    if (y >= 600*80)        {bord atteint -> on change de direction }
    or (y <= 80) Then y_dir:=-y_dir;
  Until KeyPressed;         {attend une frappe de touche }
End.
```

Après chargement des deux images et répartition sur quatre pages d'écran (on peut aussi utiliser des images multiples), le compteur des ordonnées est initialisé à 80, valeur correspondant à l'adresse de début de la ligne 1 de l'écran. Y_Dir sert à définir la direction de défilement : un nombre positif indiquant que le registre Linear Starting Address est à incrémenter (ce qui oriente le défilement vers le haut). Inversement, si Y_Dir est négatif, le défilement se fera en sens inverse.



Scrolling vertical des deux images

Dans la boucle principale l'adresse, de début (y) est constamment incrémentée (Décrémentée si y_dir est négatif). Arrivé au bord d'image, la direction de défilement est inversée et le défilement reprend dans l'autre sens.

Si on souhaite faire défiler en continu un texte de plus de quatre pages (par exemple pour un générique de fin), les choses se compliquent. Pendant le défilement, la page d'écran à venir doit être chargée depuis le disque dur et

transférée dans la partie déjà lue et disparue de l'écran. Ceci nous mènerait trop loin ici. Nous étendrons les fonctions du défilement dans le cadre de l'effet "pyrotechnique" examiné plus tard.

Il est souvent utile de ne pas se restreindre à un défilement vertical mais de prévoir également un défilement horizontal : pensez par exemple à un logo de quatre pages d'écran qui ne serait jamais visible en totalité et dont les différentes parties viendraient se dérouler devant vos yeux de bas en haut et de droite à gauche.

La solution paraît simple. Jusqu'ici, le registre Linear Starting Address a toujours été incrémenté par pas de 80, pourquoi ne pas introduire un pas de 1 ? Mais dans ce cas, la partie qui disparaît à gauche réapparaît à droite puisque les lignes sont juxtaposées directement en mémoire.

Si on a déplacé, par exemple d'un octet, le début de l'écran (déplacement à gauche de 40 pixels), le CRTC représente dans la première ligne de balayage les octets 1 à 80 (au lieu de 0-79). L'octet 80 appartient en fait déjà à la deuxième ligne du modèle. Donc à chaque octet qui sort du champ à gauche est rajouté une ligne plus haut à droite.

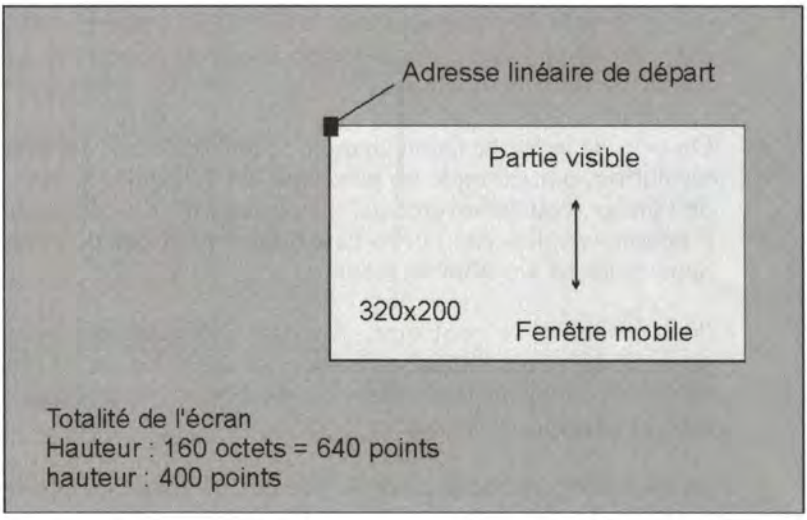
La solution consiste à agrandir virtuellement l'écran en portant sa largeur à 640 pixels. Seuls 320 pixels seront réellement affichés, mais à droite, "à côté" du moniteur, d'autres pixels viendront s'ajouter de sorte que les quatre pages ne seront plus superposées comme dans le défilement vertical mais formeront un carré. Les pixels disparus du champ de vision à gauche iront à droite, dans la zone invisible (abscisse 636-639), de nouveaux pixels étant déplacés de la zone invisible à la zone visible.

En d'autres termes, la fenêtre qui représente en fait l'écran en mode X peut aussi bouger dans le sens vertical tout simplement parce qu'il y a maintenant de la place (pages d'écran disposées côte à côte dans le sens horizontal).

Mais comment activer ce mode particulier ? La carte VGA dispose à cet effet d'un registre. Il s'agit du registre 13h du CRTC appel Row Offset. Ce nom anodin cache des possibilités insoupçonnées. On y indique en effet la distance que doit parcourir le pointeur interne des données (Linear Counter) lorsque le rayon cathodique atteint le bord droit de l'écran. Il s'agit donc de l'intervalle qui sépare les lignes à l'intérieur de la mémoire d'écran, ou en termes plus simples de la longueur des lignes !

Normalement, ce registre contient la valeur 40, tant en mode 13h qu'en mode X, ce qui correspond à une longueur de 80 octets. Le registre compte en mots, et 80 octets correspondent bien à 40 mots. En mode 13h, les lignes ont 320 octets mais l'unité de calcul qui correspond à l'élément maximal adressable par le processeur est quadruplée, passant de un octet à un double mot, ce qui nous ramène à la valeur 320/8=40.

On peut naturellement mettre en service des valeurs supérieures. Si on prend 80, les lignes ont une longueur de 160 octets (ou 640 pixels). Des lacunes de 80 octets vont donc se produire entre les différentes lignes de 80 octets qui sont remplies par la droite par les moitiés restantes de la zone invisible.



Fenêtre de visibilité dans la mémoire d'écran

La figure 6 montre la structure de la mémoire d'écran après activation du mode 160 octets. Pour déclencher le doublement des lignes, on utilise la procédure Double du module ModeXLib :

```
double proc pascal far
  mov dx,3d4h           ;registre 13h du CRTC (Row Offset)
  mov ax,5013h          ;positionné à 80 (Largeur double)
  out dx,ax             ;et écriture
  ret
double endp
```

Comme vous pouvez le voir, cette procédure ne fait rien d'autre que de charger le nombre 50h=80d dans le registre 13h du CRTC, afin de faire passer la taille des lignes à 160 octets.

Utilisation du chargeur de GIF pour des images de grande taille

Vous avez certainement envie de charger des images GIF dans ce mode étendu. Le chargeur normal peut en principe servir à cet usage puisqu'il fonctionne indépendamment du format de l'écran. Il décompresse les données dans l'ordre de la lecture, les dispose en mémoire centrale et les transfère ensuite dans la mémoire graphique. Pour pouvoir charger une image en mode 640 pixels, il suffit d'en disposer sous forme d'une image de 640 pixels. Sinon une ligne sur deux passe dans la moitié droite de l'écran, ce qui n'est sûrement pas le but de la manoeuvre.

On peut de la même façon charger en une seule fois des images de très haute résolution, par exemple un générique de 320x800. Si vous calculez la taille de l'image, vous trouverez qu'elle dépasse 64 Ko : le pointeur ^VScreen ne fonctionnera plus car Turbo Pascal ne permet pas de définir un champ qui dépasse les 64 Ko d'un segment.

Pour résoudre ce problème, il existe plusieurs approches possibles. On pourrait découper l'image en morceaux de 64 Ko, ce qui donne par exemple en double longueur des tailles de 640x100. Cette méthode n'est pas confortable et manque d'élégance.

La deuxième approche consiste à répartir l'image sur plusieurs pointeurs au moment du chargement. En cas de débordement d'un segment, on met en service la variable suivante. Mais la programmation de cette méthode n'est pas aisée.

En troisième lieu, on pourrait contourner complètement la gestion de la mémoire de Turbo Pascal et allouer la mémoire par DOS. Lors du franchissement d'une limite de segment, il suffirait d'incrémenter le registre de segment de 1000h. Mais où prendre la mémoire ? Une image de deux pages nécessite déjà 128 Ko qui seront la plupart du temps excessifs, alors que 64 Ko sont encore à la limite du supportable.

A notre avis, la meilleure solution se trouve dans la procédure ReadGIF dans ModeXLib. Les deux instructions stosb qui écrivent le pixel en fin de décodage sont suivies d'un test de débordement. Si ce test s'avère positif, l'image calculée jusqu'ici est copiée dans la mémoire d'écran. Comme les grandes pages d'écran ne se justifient qu'en mode X, on appelle à cet effet la procédure p13_2_ModeX qui transfère Vscreen dans la mémoire d'écran à la position définie par VRAM_Pos et remet ensuite le pointeur de destination à 0. La variable VRAM_Pos est ensuite déplacée dans la mémoire d'écran à la fin de la zone copiée afin qu'un éventuel deuxième débordement puisse être pris en compte.

Cette image en résolution 640 x 400 peut être lue par la procédure ReadGIF

A la fin de la procédure ReadGIF, la variable Rest est chargée de l'état de remplissage de Vscreen (se trouve en di et pointe directement sur le nombre d'octets copiés car Pascal place toujours les variables allouées à des adresses de début de segment). Ainsi, dans le programme principal, un nouvel appel de p13_2_ModeX avec ce reste comme nombre d'octets à copier provoquera le transfert en mémoire d'écran de ce reste.

Ce procédé n'est peut être pas le plus élégant sur le plan algorithmique mais il présente deux avantages. D'abord la procédure ReadGIF est 100% compatible avec l'ancienne. Il n'est pas nécessaire de préciser la taille de l'image en transmettant un argument ou en lisant l'en-tête GIF, de sorte que théoriquement tous les formats d'image peuvent être pris en compte (même une extension aux résolutions SuperVGA ne pose aucun problème). Le deuxième avantage tient à l'économie de mémoire : un seul segment est sollicité en mémoire centrale, il peut être libéré du tas à l'issue du chargement et servir au chargement de l'image suivante.

Lorsqu'on a chargé en mémoire graphique une image de la taille souhaitée, le défilement proprement dit commence. On se sert encore à cet effet du registre Linear Starting Address qui peut maintenant recevoir n'importe quelle valeur. Le défilement peut être horizontal, vertical et même diagonal. Pour le défilement vertical, on procède presque exactement comme dans le chapitre précédent : le registre Linear Starting Address est à chaque fois incrémenté (puis décrémenté) de la longueur d'une ligne. Il faut simplement tenir compte de la modification de cette dernière de sorte que le défilement vertical fonctionne par pas de 160 octets.

Le défilement horizontal s'obtient simplement en incrémentant ou décrémentant le registre par pas de 1. Un octet représentant à nouveau quatre pixels, le défilement ne se fait que par tranches de 4 pixels (mais il est possible de faire autrement comme le montre la description du registre 13h du contrôleur d'attributs Pixel Panning).

Le défilement diagonal résulte de la composition des deux directions. Par exemple, pour se déplacer de quatre pixels vers le haut et vers la gauche, on augmente le registre Linear Starting Address de 641 (160 octets par ligne * 4 lignes + 1 octet).

Les capacités de ce registre sont démontrées par le programme SCROLL4.PAS :

```
Uses Crt,Gif,ModeXLib;
Var x,                    {offset courant dans le sens horizontal }
    x_dir,                {direction du défilement dans le sens
                           horizontal }
    y,                    {offset courant dans le sens vertical }
    y_dir:word;           {direction du défilement dans le sens
                           vertical }
Begin
  Init_ModeX;             {active le mode X }
  double;                 {sélectionne le mode 160 octets
                           (640*400 pixels) }
  LoadGif('640400');      {charge l'image }
  p13_2_ModeX(vram_pos,rest div 4);
                          {reste de l'image en mémoired'écran }
  x:=1;                   {x commence en colonne 1}
  x_dir:=1;               {direction horizontale = 1 octet par
                           itération}
  y:=160;                 {y commence en ligne 1}
  y_dir:=160;             {direction verticale = 160 octets par
                           itération }
  Repeat
    Inc(x,x_dir);         {défilement horizontal }
    Inc(y,y_dir);         {défilement vertical }
    WaitRetrace;          {attend le retour de balayage}
    SetStart(y+x);        {écrit le nouveau début dans le registre }
    if (x >= 80)          {bord vertical atteint -> changer de
                           direction}
    or (x <= 1) Then x_dir:=-x_dir;
    if (y >= 200*160)     {bord horizontal atteint -> changer de
                           direction }
    or (y <= 160) Then y_dir:=-y_dir;
  Until KeyPressed;       {tourne jusqu'à ce qu'on frappe une touche }
```

```
TextMode(3);
End.
```

Cette version met en service, en plus des variables de position et de direction verticale, des variables semblables pour le sens horizontal. Elles s'appellent x et x_dir et remplissent la même fonction que leurs homologues en y : x donne l'abscisse de la position courante, l'unité correspondant à quatre plans parallèles et occasionnant en cas d'incrémentation ou de décrémentation un déplacement de quatre pixels. x_dir décrit la direction du défilement dans le sens horizontal : +1 correspond à un déplacement du contenu de l'image de quatre pixels vers la gauche, -1 à un déplacement de quatre pixels vers la droite.

Après l'initialisation classique, la routine appelle encore double qui enclenche le mode 160 octets de la carte VGA, ce qui dédouble la largeur de l'écran virtuel et autorise un défilement horizontal. L'image est ensuite chargée par LoadGIF. A cause des dimensions de l'image (640x400 pixels = 256000 octets), une partie en est copiée dans la mémoire d'écran (on pourrait aussi dire que, par manque de place, elle y est remisée).

Comme ce remisage n'a lieu qu'en cas de franchissement de la frontière des 64 Ko, la procédure p13_2_ModeX transfère le reste en mémoire d'écran à partir de la position qui suit le dernier octet copié (vram_pos) et en quantité Rest (variable remplie par LoadGIF).Le nombre d'octets doit être divisé par quatre car p13_2_ModeX travaille en mots-CPU : avec une longueur de 1 ce sont quatre octets qui sont copiés à cause des quatre plans présents. Le contenu de la variable Rest est par contre un nombre d'octets réels, qui correspond donc au nombre de pixels.

La direction de défilement horizontale est initialisée à 1 comme nous l'avons vu, de sorte que le défilement se fait d'abord vers la gauche. La position est également prise égale à 1 pour commencer avec la colonne 1 (avant la boucle le début de l'écran se trouve encore en ligne 0 et colonne 0). Les variables verticales sont initialisées à 160 en raison du dédoublement des lignes qui ont une longueur de 160 octets.

Dans la boucle proprement dite, il se passe relativement peu de choses. Le déplacement horizontal est suscité par une instruction Inc. Le début de l'écran est défini comme la somme des offsets x et y, de façon à prendre en compte les deux directions. Une condition supplémentaire a été introduite qui a pour effet d'inverser le sens de déplacement lorsque le bord droit, gauche, inférieur ou supérieur est atteint. La zone de défilement reste donc toujours dans le cadre du rectangle virtuel fixé par la mémoire d'écran.

6.4. TOUT À LA FOIS : FRACTIONNEMENT D'ÉCRAN AVEC DÉFILEMENT

Le fractionnement de l'écran et le défilement multidirectionnel peuvent donner individuellement des effets agréables. En les combinant, on augmente énormément l'impact optique d'une présentation. Nous prendrons comme exemple le programme SCRL_SPT.PAS :

```
uses crt,Gif,ModeXLib;
Var x,                      {offset courant dans la direction
                             horizontale }
    x_dir,                  {direction de défilement dans le sens
                             horizontal }
    y,                      {offset courant dans la direction verticale}
    y_dir:word;             {direction de défilement dans le sens
                             vertical}
    split_line:word;        {position courante de la ligne de
                             fractionnement }
    split_dir:word;         {direction de déplacement de la ligne de
                             fractionnement }
Begin
  Init_ModeX;               {active le mode X }
  double;                   {sélectionne le mode 160 octets }
  Screen_Off;               {éteint l'écran }
  LoadGif_Pos('640400',160*50); {charge une grande image en position
                                 (0, 50)}
  p13_2_ModeX(vram_pos,rest div 4);  {copie le reste en mémoire VGA }
  LoadGif('boule');         {charge une petite image en position (0,0)}
  p13_2_ModeX(0,160*50);    {et la copie sur l'écran }
  Screen_On;                {réactive l'écran }
  split_line:=150;          {met la séparation en ligne 150 }
  split_dir:=1;             {déplace d'abord la ligne de fractionnement
                             vers le bas }
  x:=1;                     {x commence en colonne 1}
  x_dir:=1;                 {direction horizontale = 1 octet par
                             itération }
  y:=160;                   {y commence en ligne 1}
```

```
  y_dir:=160;              {direction verticale = 160 octets par
                            itération }
  Repeat
    Inc(x,x_dir);          {défilement horizontal }
    Inc(y,y_dir);          {défilement vertical }
    Inc(Split_line,Split_dir);  {déplace la ligne de fractionnement }
    WaitRetrace;           {attend le retour de balayage }
    SetStart(50*160+y+x);{écrit le nouveau début d'écran dans le
                            registre }
                           {en sautant les 50 premières lignes }
    Split(Split_line);     {fractionne l'écran }
    if (x >= 80)           {bord vertical atteint -> changer de
                            direction }
    or (x <= 1) Then x_dir:=-x_dir;
    if (y >= 200*160)      {bord horizontal atteint -> changer de
                            direction }
    or (y <= 160) Then y_dir:=-y_dir;
    if (split_line >= 200){ligne de fractionnement touche le
                            bord -> changer de direction }
    or (split_line <= 150) then split_dir:=-split_dir
  Until KeyPressed;        {tourne jusqu'à ce qu'on frappe une touche}
TextMode(3);
End.
```

Ici l'écran n'est pas seulement coupé en deux zones statiques comme précédemment mais la ligne de fractionnement se déplace elle-même. Ce déplacement est décrit par les variables Split_line et Split_dir qui fonctionnent comme les variables x et y correspondantes. Split_line mémorise la ligne de fractionnement courante et split_dir contient la valeur 1 pour un déplacement vers les ordonnées croissantes (vers le bas) ou -1 pour un déplacement vers le haut (ordonnées décroissantes).

Le programme SCRL_SPT : fractionnement d'écran avec défilement.

Après activation du mode X en lignes de 160 octets, l'écran est éteint pour éviter les effets dérangeants pendant la procédure de chargement. Ce n'est pas la vitesse qui importe ici mais il est plus agréable que l'écran reste noir pendant la génération de l'image. Si vous désirez observer le processus de chargement (pour détecter des erreurs), laissez simplement tomber cette instruction. Mais dans la version définitive d'une démo, la composition de l'image pendant le chargement ne doit pas se voir.

La grande image qui par la suite défilera tous azimuts est chargée avec LoadGIF_Pos en position 160*50, c'est-à-dire 50 lignes après le début de la RAM. La petite image est simplement chargée par LoadGIF qui la dépose en début de mémoire. LoadGIF_Pos fonctionne comme LoadGIF mais avec un argument supplémentaire : l'offset qui indique à quel endroit remiser les parties d'images qui débordent.

Le décalage de 50 lignes tient compte du fonctionnement de l'écran fractionné. Comme à l'endroit de la ligne de fractionnement, le Linear Counter est remis à 0, la partie située au-dessous de cette ligne correspond au contenu du début de la mémoire d'écran : c'est là que se trouve la petite image. Lors du défilement, on prendra comme origine l'adresse de début de la grande image.

L'écran est maintenant rallumé. La ligne de fractionnement est initialisée à 150. Cette valeur est convertie en lignes de balayage de façon à accorder son indépendance au quart inférieur de l'écran. Split_dir reçoit la valeur 1 pour que, dans la boucle, la partie inférieure de l'écran s'esquive par le bas.

Lorsque la ligne de fractionnement a la valeur 200, la partie "splittée" est arrivée tout en bas et la direction de déplacement est inversée. Au bord supérieur (150), on programme le même phénomène dans l'autre sens.

Pendant que la partie supérieure de l'écran défile, la partie inférieure se déplace verticalement.

On a ajouté ici la constante 50*160 à l'argument de la fonction SetStart qui paramètre le défilement : on évite ainsi que les 50 premières lignes de la mémoire d'écran deviennent visibles au cours du défilement. Elles contiennent en effet la partie inférieure du fractionnement et l'image supérieure ne commence que 50 lignes plus loin.

6.5. FERMETURE DE PORTE : ASSEMBLAGE DYNAMIQUE DE DEUX MOITIÉS D'UNE IMAGE

Une autre application de la combinaison du fractionnement et du défilement d'écran consiste à assembler une image en deux parties. Cet effet est idéal pour ouvrir une démo.

La moitié supérieure de l'image, qui descend vers le milieu de l'écran, est contrôlée par le registre Linear Starting Address qui commande simplement un défilement vertical (vers le bas). Il faut veiller à ce que la page d'écran 1 soit vide ou colorée uniformément car elle apparaît pour moitié en début de scénario.

La moitié inférieure est commandée par fractionnement. La décrémentation de la ligne de fractionnement a pour effet de déplacer cette moitié vers le haut, jusqu'au milieu de l'écran.

La procédure Squeeze tirée de MODEXLIB.ASM s'occupe de ces deux opérations :

```
squeeze proc pascal far          ;compose un écran à partir de deux
                                 ;moitiés
  mov si,200*80                  ;première addresse de début
  mov di,199                     ;première valeur de la ligne de
                                 ;fractionnement
sqlp:                            ;boucle principale
  call waitretrace               ;attend le retour de balayage
  call split pascal, di          ;crée la moitié inférieure par
                                 ;fractionnement
  call setstart pascal, si       ;crée la moitié supérieure par
                                 ;défilement
```

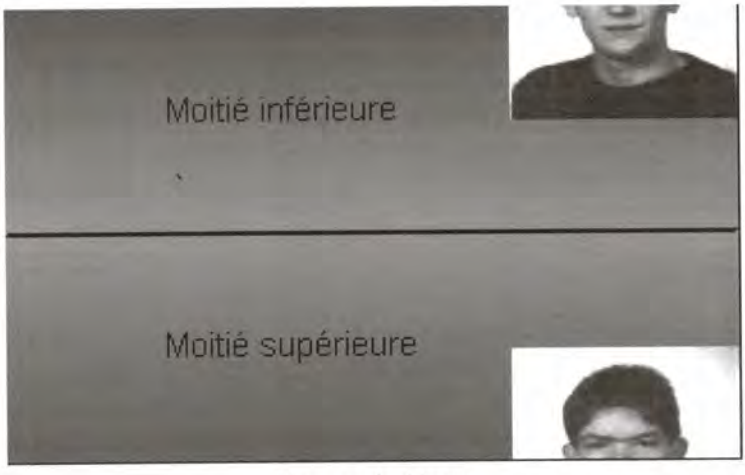
```
    sub si,80                   ;une ligne vers le bas
    dec di                      ;diminue la ligne de fractionnement
    cmp di,99d                  ;ce qui remonte la moitié inférieure
    jae sqlp                    ;fini ?
    ret
squeeze endp
```

Dans cette procédure, les registres SI et DI sont un peu détournés de leur vocation traditionnelle mais ce n'est pas par nécessité : la vitesse n'est pas critique ici car le processeur se tourne les pouces tandis que la carte VGA effectue tout le travail. Mais il n'est jamais mauvais d'exploiter des variables registres qui économisent la mémoire et le temps.

Si reçoit le contenu du registre Linear Starting Address et gère les mouvements de la partie supérieure de l'écran, DI décrit la ligne de fractionnement qui contrôle la position de la partie inférieure. Ces registres sont d'abord chargés avec leurs valeurs initiales pour lesquelles l'image est entièrement en dehors de l'écran.

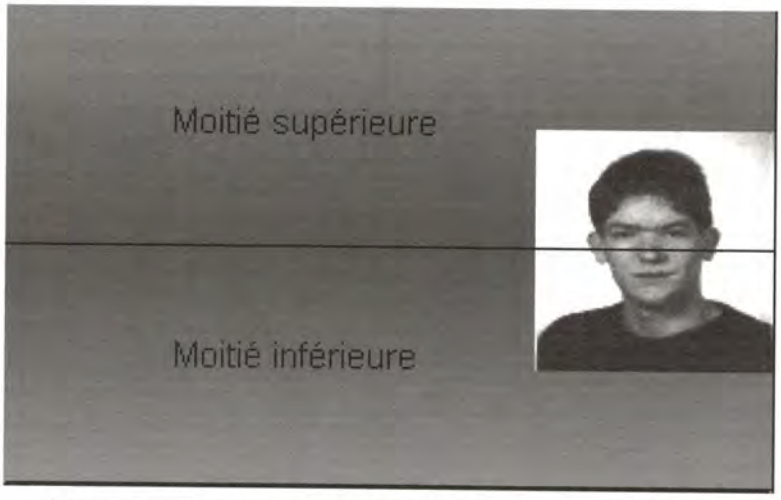
Après la synchronisation nécessaire avec le retour de balayage vertical, la boucle fixe le nouvel état courant à l'aide des registres impliqués. Les instructions qui suivent produisent le déplacement en décrémentant les registres de 80 (Linear Starting Address, 1 ligne) et de 1 (ligne de fractionnement). La partie supérieure descend et la partie inférieure remonte, ce qui conduit à l'assemblage progressif de l'image.

Le test de fin de boucle vérifie si la ligne de fractionnement dépasse la moitié de l'écran qui lui est attribuée.



L'image de départ

Le programme de démonstration Squeeze se contente, à titre d'exemple, de charger une image et d'en assembler les deux moitiés. Le déclenchement se fait par la touche **ENTREE** et l'attente correspondante permet de se rappeler une condition à respecter lors du développement d'une image. Avant frappe de la touche **ENTREE**, l'image apparaît à l'écran sous sa forme originale. Vous constaterez ainsi que les parties inférieure et supérieure sont permutées. Il ne faut pas oublier en effet que sous la ligne de fractionnement apparaît le début de la mémoire d'écran, alors que le défilement gère des données graphiques issues de la deuxième moitié de la mémoire d'écran.



L'image après permutation des moitiés supérieure et inférieure

Pour respecter ce format inversé, il suffit de permuter les deux parties de votre image par la fonction copier-coller de votre éditeur de dessins. Compte tenu de ces précisions, le programme SQUEEZE.PAS s'explique de lui-même :

```
uses ModeXLib,Gif;
Begin
  Init_ModeX;                          {Active le mode X}
  LoadGif('squeeze');                  {charge l'image }
  p13_2_ModeX(vram_pos,rest div 4);
  ReadLn;                              {attend une frappe de touche }
  Squeeze;                             {assemble l'image }
  ReadLn;
TextMode(3);
End.
```

6.6. DÉFILEMENT CONTINU EN MODE TEXTE

En mode texte, le défilement s'effectue de la même façon qu'en mode graphique. L'extrait visible de la mémoire d'écran est déplacé par variation de l'adresse de début (Linear Starting Address). Mais l'affaire présente un inconvénient : ce défilement présente des saccades car il se fait caractère par caractère alors que précédemment il se faisait pixel par pixel. Ceci tient à l'organisation de la mémoire d'écran en mode texte. On ne dispose plus d'informations sur les pixels individuels, la Linear Starting Address se référant toujours à un caractère entier. Le défilement manque donc de finesse.

Le salut vient comme d'habitude des nouveaux registres VGA qui n'attendent que des problèmes de ce type à résoudre. Nous allons exploiter les registres de panning (panoramiques) horizontaux et verticaux. Un panoramique est un déplacement de l'écran d'un pixel, les deux registres s'appliquant également au mode texte.

Pour effectuer un défilement continu, on déplace simplement le contenu de l'écran en déclenchant un panoramique dans la direction souhaitée. Lorsqu'on a fait défiler l'équivalent d'un caractère, le registre de panning est réinitialisé et le registre Linear Starting Address est calé sur la nouvelle situation. Cette dernière mesure est indispensable car le panoramique ne peut pas dépasser une largeur ou une hauteur de caractère. Il s'agit d'un instrument pour la mise au point fine. Le registre Linear Starting Address est quant à lui responsable du défilement grossier.

On déclenche un panoramique vertical à l'aide du registre 8 du CRTC (Initial Row Address) : les bits 4-0 indiquent la ligne graphique à utiliser dans la première ligne de balayage. Si on augmente ce nombre de 1, la ligne 1 est prise en compte dans le jeu de caractères et le contenu de l'écran se déplace d'une ligne vers le haut.

Le panoramique horizontal est géré par le contrôleur d'attributs (Attribute Controller) dont le registre 13h (Horizontal Pixel Panning) déclenche des déplacements dans le sens des abscisses. Les valeurs de ce registre ont une signification inhabituelle : un 0 induit un déplacement de 1 pixel, les nombres de 1 à 7 donnent des déplacements de 2 à 8 pixels, alors que le nombre 8 ne provoque aucun panoramique. Les conversions nécessaires sont effectuées par un calcul simple qui évite toute structure if, facteur de ralentissement :

```
nombre_registre:=(déplacement_panoramique - 1) mod 9
```

Voici le programme complet de la démonstration TSCROLL.PAS :

```
Uses ModeXLib,Crt;

Var x,                          {abscisse en pixels}
    x_dir,                      {direction horizontale }
    y,                          {ordonnée en pixels }
    y_dir:Word;                 {direction verticale }

Procedure Wait_In_Display;assembler;
{symétrique de Wait_In_Retrace, attend la génération de l'image par
 le rayon cathodique }
asm
  mov dx,3dah                   {Input Status 1}
@wait2:
  in al,dx
  test al,8h
  jnz @wait2                    {Image présente ? -> terminé }
End;
Procedure Wait_In_Retrace;assembler;
{attend le retour de balayage vertical, réinitialise
en même temps le flip-flop de l'ATC en effectuant
un accès en lecture au registre Input Status 1 }
asm
  mov dx,3dah                   {Input Status 1}
@wait1:
  in al,dx
  test al,8h
  jz @wait1                     {balayage actif ? -> terminé }
End;

Procedure FillScreen;
{remplit la mémoire d'écran avec une image de test de 160*50
 caractères}
var i:word;
Begin
  For i:=0 to 160*50 do Begin  {boucle des caractères}
    If i mod 10 <> 0 Then{enregistrer le compteur de colonnes ?}
      mem[$b800:i shl 1]:=     {non, alors'-'}
```

```
        Ord('-') Else
      mem[$b800:i shl 1]:=      {oui, numéro de colonne décompté en
                                 dizaines }
      ((i mod 160) div 10) mod 10 + Ord('0');
    If i mod 160 = 0 Then       {colonne 0 ? -> enregistre le compteur
                                 de lignes }
      mem[$b800:i shl 1]:=(i div 160) mod 10 + Ord('0');
  End;
End;

Procedure V_Pan(n:Byte);assembler;
{exécute un panoramique vertical}
asm
  mov dx,3d4h                   {registre 8 du CRTC (Inittial Row
                                 Adress)}
  mov al,8
  mov ah,n                      {fixe le déplacement panoramique }
  out dx,ax
End;

Procedure H_Pan(n:Byte);assembler;
{exécute un panoramique horizontal}
asm
  mov dx,3c0h                   {Index/Data Port de l'ATC}
  mov al,13h or 32d             {sélectionne le registre 13 h
                                 (Horizontal Pixel Paning)}
  out dx,al                     {met à 1 le bit 5 (Palette RAM Address
                                 Source)}
  mov al,n                      { pour ne pas couper l'écran }
  or al,32d                     {fixe le déplacement panoramique }
  out dx,al
End;

Begin
  TextMode(3);                  {mode 3 du BIOS (80*25 caractères,
                                 couleur)}
  FillScreen;                   {construit l'image de test }
```

```
  portw[$3d4]:=$5013;          {double largeur d'écran virtuelle
                                (160 caractères)}
  x:=0;                        {initialise les coordonnées et
                                directions }
  x_dir:=1;
  y:=0;
  y_dir:=1;
  Repeat
    Inc(x,x_dir);              {déplacement horizontal et vertical }
    Inc(y,y_dir);
    If (x<=0) or (x>=80*9)     {changement de direction au bord }
      Then x_dir:=-x_dir;
    if (y<=0) or (y>=25*16)
      Then y_dir:=-y_dir;
    Wait_in_Display;           {attend la génération de l'image }
    SetStart((y div 16 *160)   {recale l'adresse de début
                                (défilement grossier }
      + x div 9);
    Wait_in_Retrace;           {attend le retour de balayage }
    V_Pan(y mod 16);           {panoramique vertical (défilement fin)}
    H_Pan((x-1) mod 9);        {panoramique horizontal (défilement
                                fin)}
  Until KeyPressed;            {attend une frappe de touche }
  TextMode(3);                 {remet le mode vidéo normal }
End.
```

Après appel du mode texte et remplissage des caractères d'une image témoin, on active la largeur double virtuelle en mettant dans le registre 13 h du CRTC (Row Offset) la valeur 320 octets / 4 octets = 80 (4 à cause de l'accès doubleword). Le déplacement lui-même est géré comme en mode graphique, mais avec d'autres valeurs.

Vous remarquerez que le programme WaitRetrace a été subdivisé en deux parties appelées Wait_in_Display et Wait_in_Retrace. L'explication est la suivante. Les registres employés ici sont liés à une logique de timing différente. Le registre Linear Starting Address est chargé pratiquement au début du retour de balayage de sorte que la plupart du temps une commutation à ce moment n'a encore aucun effet sur l'image qui suit. Jusqu'ici, ce n'était pas trop grave car l'ensemble du processus n'était décalé que d'un retour de balayage. Mais maintenant, il faut encore ajuster les registres de panoramique dont l'effet est immédiat lorsqu'ils sont modifiés.

L'adresse de début est ainsi fixée pendant la génération de l'image car elle ne joue plus de rôle à ce moment et sera donc prête pour la prochaine génération. Les registres de panoramique sont fixés pendant le retour de balayage car leur effet est instantané et doit donc être déclenché dans la partie invisible de l'image.

La procédure Wait_in_Retrace remplit également une autre mission. L'accès à l'ATC est un peu bizarre. A chaque écriture sur le port 3c0h, ce dernier change de rôle, devenant tantôt un registre d'index, tantôt un registre de données. Au début du programme l'état du registre n'est pas déterminé. Un accès en lecture au registre 1 de l'Input Status va alors commuter le port 3c0h en mode index, ce qui nous place en territoire connu.

Comme l'accès en écriture à l'ATC (contrôleur d'attributs) suit presque immédiatement l'attente du retour de balayage, on peut se permettre de supposer que le mode index est actif. La seule cause de changement de mode de ce port pourrait être due à un programme, résident qui intercale une interruption à la sauvette, mais cette hypothèse est peu probable et on peut de toute façon l'interdire par une instruction CLI.

Quand on écrit dans l'index de l'ATC, il est très important de toujours mettre à 1 le bit 5. Ce bit contrôle en effet l'accès à la palette interne de l'ATC. S'il est à 0, le processeur obtient les pleins pouvoirs pour l'accès et l'ATC se met en veilleuse, de sorte que dans le meilleur des cas (si on remet immédiatement le bit à 1), on obtient un tressautement, sinon l'ordinateur se plante et il faut le réinitialiser.

Bien entendu, vous pouvez également combiner ce défilement avec un fractionnement d'écran. Nous ne le ferons pas ici car la méthode est exactement la même que dans le mode graphique. Le fractionnement d'écran est complètement indépendant du mode car on y indique directement la ligne de balayage où doit avoir lieu le partage.

6.7. UNE AUTRE MANIÈRE DE NETTOYER L'ÉCRAN : L'IMAGE QUI S'ÉCOULE

Nous avons déjà souvent parlé du mode double scan de la carte VGA. Le dédoublement des lignes (pour afficher 200 lignes graphiques avec une résolution physique de 400 lignes) est obtenu selon le Bios soit par le bit 7 (Double scan enable) du registre 9 du CRTC (Maximum Row Address), soit en remplaçant 0 par 1 dans les bits 4-0. Ces bits contiennent en mode texte le nombre de lignes de balayage par ligne de caractères moins 1 (= 15 en

VGA), ce qui détermine aussi le nombre de lignes de balayage dans lesquelles sont répétées les mêmes informations issues de la mémoire d'écran.

En mode texte, ces lignes ne sont pas tout à fait identiques car elles sont construites à partir d'autant de lignes différentes du jeu de caractères. Mais en mode graphique, il n'y a pas jeu de caractères de sorte que ce sont effectivement les mêmes données qui sont reprises. Une valeur de 1 dans les bits mentionnés provoque donc le dédoublement des lignes, en pratique chaque ligne donne lieu à une copie.

Mais que se passe-t-il maintenant si on augmente cette valeur ? Exactement ce à quoi on peut s'attendre : le nombre de copies créées s'accroît, les pixels s'allongent en hauteur, la résolution verticale diminue.

Le programme ECOULE.PAS exploite très précisément cet effet. Il affiche d'abord à l'écran une image de démo puis l'étire en incrémentant de façon continue les bits 4-0. Il faut veiller à ne pas toucher aux autres bits du registre. On sauve donc l'ancien contenu en old9 et il est recombiné par l'opérateur OR avant l'accès en écriture. L'ensemble est synchronisé comme d'habitude sur le retour de balayage vertical :

```
Uses Crt,Gif,ModeXLib;
Procedure Ecoule;
var i,
    Old9:Byte;
Begin
  Port[$3d4]:=9;                {sélectionne le registre 9 du CRTC
                                 (Maximum Row Adress) }
  Old9:=Port[$3d5] and $80;     {mémorise l'ancien contenu, ...}
  for i:=2 to 31 do begin       {...évite la relecture }
    WaitRetrace;                {synchronisation}
    Port[$3d5]:=old9 or i;      {écriture dans le registre }
  End;
End;

Begin
  asm mov ax,13h; int 10h End;  {active le mode 13h (ou un autre mode
                                 graphique)}
  LoadGif('coin');              {charge le fond d'écran }
  Move(vscreen^,Ptr($a000,0)^,64000); {et l'affiche }
  ReadLn;
  Ecoule;                       {lance l'effet d'écoulement }
```

```
  ReadLn;
  TextMode(3);                    {remet la carte VGA dans l'état
                                   initial }
End.
```

En principe, la méthode s'applique également au mode texte mais elle conduit à des écrans horribles. La raison est la suivante : comme en mode graphique, la carte VGA dédouble, triple, etc le contenu de la mémoire. Mais les nouvelles lignes rajoutées ne disposent pas d'informations sur le jeu de caractères. De ce fait, l'écran est parsemé de résidus aléatoires. Notre effet se limitera donc au mode graphique.

Pour choisir une image appropriée, il faut veiller à ce que le bord supérieur comporte au moins 13 lignes noires ou d'une même couleur. Sinon l'image ne donne pas l'impression de s'écouler mais simplement de s'étirer. Avec 400 lignes de résolution verticale et un maximum de 31 pour le registre Maximum Row Address, chaque ligne est affichée 32 fois, ce qui représente 400/32=12,5 lignes visibles qui doivent être de la même couleur pour que l'image entière soit uniforme.

6.8. Un peu plus de couleurs SVP : les Copper list

Les effets étudiés jusqu'ici pouvaient s'obtenir par simple reprogrammation de certains registres VGA. Le processeur ne prenait en charge que la commande, par exemple la modification perpétuelle de l'adresse de début d'écran pour un défilement.

Dans ce chapitre, le processeur se voit confier une nouvelle tâche qui ne se réduit plus à la commande générale mais consiste à effectuer des contrôles. La copie reste cependant tabou. Le processeur va surveiller sans cesse la carte VGA et, au passage de certaines lignes de balayage, il entreprendra des modifications dans les registres VGA. Il en résulte un effet déjà connu au temps du C64 : les Copper list.

Il s'agit de barres horizontales colorées qui remontent et redescendent sans cesse sur l'écran, pendant que d'autres effets comme des textes défilants se déroulent en avant-plan. Rien qu'en voyant l'image de ces barres, on peut déjà en déduire leur principe de fonctionnement. La mémoire d'écran ne contient aucune donnée aux endroits concernés, elle est par exemple remplie de zéros, qui ne peuvent pas être directement à l'origine de ces apparitions. Mais dans chaque nouvelle ligne de balayage, la signification de ces zéros est

redéfinie : ils sont interprétés ici comme une touche de rouge, là-bas comme un jaune éclatant, et ainsi de suite.

On peut ainsi représenter bien plus de 256 couleurs à l'écran. En effet, chaque ligne de balayage peut contenir 256 couleurs qui sont réinterprétées différemment à la ligne suivante. En pratique, cette variété de couleurs est pourtant très restreinte car à l'intérieur d'un retour de balayage vertical il n'est pas possible de redéfinir complètement la palette mais uniquement quelques entrées dont le nombre dépend de la vitesse de l'ordinateur. Dans l'exemple étudié ici, seule la couleur 0 est reprogrammée, ce qui donne déjà 127 nouvelles couleurs. Si on retouche quelques couleurs de plus, le nombre obtenu s'accroît encore davantage. Mais cette possibilité ne joue plus grand rôle à l'ère des cartes Hicolor et TrueColor.

Si notre couleur variable est redéfinie au rythme des lignes de balayage, on obtient une structure en barres horizontales. Les couleurs autres que la couleur 0 ne sont pas touchées. Ainsi, un texte peut tranquillement continuer son défilement au premier plan s'il exploite par exemple les couleurs 1 à 16. Aux emplacements occupés par un zéro, il deviendra transparent et laissera apparaître les Copper list.

Pendant le défilement du texte, il n'est pas nécessaire de se soucier du fond d'écran, qui sera sans cesse sauvegardé et réaffiché avec des méthodes conventionnelles. On copie simplement sur l'écran des blocs d'octets qui contiennent des zéros aux endroits transparents. La possibilité d'exploiter le mode d'écriture 1 qui est très rapide représente alors un grand avantage.

Ce processus peut se comparer à celui des interfaces GenLock qui aux endroits où le signal vidéo présente une certaine couleur (le plus souvent bleue) incruste une image télé. Là aussi, le mélange est effectué à un niveau très bas de l'électronique et non pas dans la mémoire d'écran qui est trop lente.

Mais comment produire les lignes multicolores ? Les ordinateurs domestiques possèdent généralement une interruption de ligne de balayage. Le contrôleur vidéo peut être programmé de façon à déclencher une interruption au moment où il traite une certaine ligne de balayage. Cette interruption permet de réagir très rapidement pour changer la couleur.

Si certaines cartes VGA possèdent encore une interruption associée au retour de balayage vertical (souvent désactivée par micro-interrupteur), il n'en est pas de même avec le signal de retour horizontal. Nous ne connaissons aucune carte graphique qui dispose de cette interruption. Nous n'avons donc pas le choix : il nous faut surveiller sans cesse l'état de la carte et décompter les lignes de balayage pour changer la couleur au bon moment.

Nous rechercherons d'abord une situation de départ clairement identifiée en attendant le retour de balayage vertical. Ensuite, dès que la ligne Display

Enable sera activée (ce qui se détecte par le bit 0, Display Enable Complement du registre Input Status), on peut être sûr qu'on se trouve en ligne de balayage 0 . La surveillance du bit cité permet de décompter les lignes parcourues.

Comparé à l'affichage d'un écran complet, l'affichage d'une ligne de balayage prend très peu de temps (30 µs ce qui équivaut à 990 cycles à la cadence de 33 Mhz). Pendant le laps d'attente, on ne peut donc guère se permettre de dérouler des interruptions la plupart du temps trop longues.

Mais l'abandon de l'interruption pose cependant un autre problème très sérieux : si on veut émettre un son en même temps, le temps se fait pressant et il faut un excellent timing pour éviter que la carte sonore ne demande des données juste pendant le retour de balayage horizontal.

Mais cette difficulté se révèle négligeable par rapport au temps de calcul nécessaire à l'obtention de l'effet Copper par copie dans la mémoire d'écran.

Si on ne remplit pas l'écran entier avec des lignes de Copper en se limitant par exemple à la moitié supérieure, il restera assez de temps dans la moitié inférieure pour faire défiler des textes et mener à bien des calculs de sons.

Pour maîtriser le timing , il faut également accorder beaucoup d'attention à l'ordre des instructions. Dans la brève période du retour de balayage (environ 6 µs), il est pratiquement impossible de faire des calculs pour plusieurs barres. Il faudra donc les préparer à l'avance, et pendant le retour proprement dit ne procéder qu'à la retouche des couleurs.

Le programme de démonstration Copper présenté ci-dessous se compose d'une partie en Pascal (COPPER.PAS) et d'une partie en assembleur (COPPER.ASM). Il présente la mise en oeuvre des techniques exposées :

```
Uses Crt,ModeXLib;
var y1,                    {ordonnée du copper 1}
    y1_dir,                {direction du copper 1}
    Masque:Word;           {masque de recouvrement des coppers}

Procedure MakeCopper(y_pos1,y_pos2,overlay_Masque:word);external;
{$l copper}

begin
  TextMode(3);             {les coppers fonctionnent dans TOUS les
                            modes vidéos!}
  y1:=Port[$3da];          {active le mode index de l'ATC }
```

```
  Port[$3c0]:=$11 or 32; {sélectionne le registre 11h }
  Port[$3c0]:=255;       {couleur de cadre 255}
  y1:=0;                 {on commence au bord supérieur de l'écran }
  y1_dir:=2;             {déplacement d'abord vers le bas }
  Masque:=$00ff;         {copper 1 (rouge) au premier plan}
  Repeat
    Inc(y1,y1_dir);      {déplacement du copper }
    If (y1<=0) or (y1>=150) {arrivé au bord :}
      then Begin
      y1_dir:=-y1_dir;   {change de direction}
      Masque:=Swap(Masque);{autre copper au premier plan }
    End;
    Write('C e c i  e s t  u n  t e x t e    ');
    MakeCopper(y1,150-y1,Masque);{dessine le copper}
  Until KeyPressed;
End.
```

On commence par activer le mode texte 3 mais tout autre mode texte ou graphique convient également. Le comptage des lignes de balayage est en effet indépendant du mode VGA. Seule la résolution verticale qui diffère d'un mode à l'autre est à prendre en considération, car elle change les intervalles de définition des variables qui expriment la position et la direction des barres. On exploite encore ici la résolution physique qui dans les mode à 200 lignes (par exemple mode 13h) comporte 400 lignes (double scan).

La couleur du cadre de l'écran est fixée à 255 par l'intermédiaire du registre 11h de l'ATC. La couleur 0 est en effet prise pour colorer les barres. Si on veut aussi voir les barres dans le cadre d'écran, on peut laisser la couleur du cadre à 0. Une valeur différente de 0 augmente la durée du retour de balayage horizontal ce qui peut faire disparaître les irrégularités qui apparaissent au bord gauche de l'écran avec les ordinateurs les plus lents.

A l'intérieur de la boucle, le copper est déplacé verticalement et affiché. L'appel de WaitRetrace se cache dans la procédure MakeCopper pour rester à proximité immédiate de Retrace et du comptage de lignes. Un texte quelconque est par ailleurs affiché à des fins de démonstration. Il montre à quel point il est simple de modifier l'écran indépendamment du fond. On observe aussi que lorsque le rayon cathodique se trouve dans le domaine d'où les barres de Copper sont exclues, il reste assez de ressources de calcul pour effectuer d'autres tâches.

La procédure MakeCopper proprement dite se trouve dans le module COPPER.ASM. Au moment de l'appel, on transmet les ordonnées y des deux coppers (rouge et vert) et un masque de priorité qui régit le recouvrement

des deux objets. L'octet faible est associé au copper 1 et l'octet fort au copper 2. Un 0 signifie que le copper associé est à l'arrière-plan. 0ffh le met au premier plan. Si les deux valeurs sont nulles, les couleurs se mélangent en cas de superposition. Un masque de 0ffffh efface les deux coppers de la zone de recouvrement.

Dans notre exemple, le masque est inversé lorsque la barre atteint son ordonnée limite, ce qui donne l'impression d'une sorte de mouvement circulaire. A la montée, le copper rouge est à l'arrière-plan mais il passe à l'avant en descendant l'écran.

Le module COPPER.ASM ne contient que la seule procédure MakeCopper que nous expliquons maintenant plus en détail :

```
extrn waitretrace:far
data segment public
  maxrow dw (?)
data ends

code segment public
public makecopper

assume cs:code,des:data

MakeCopper proc pascal y_pos1,y_pos2,overlay_masque:word
; dessine 2 barres de Copper en ordonnée y_pos1 (rouge)
; et y_pos2  (vert)
; overlay_masque: 0ff00h : copper 2 au premier plan
;                 000ffh : copper 1 au premier plan
;                 00000h : mélange des deux coppers

hauteur equ 88              ;hauteur par copper


  mov ax,y_pos1             ;détermine l'ordonnée maximale
  cmp ax,y_pos2
  ja ax_high
  mov ax,y_pos2
ax_high:
  add ax,hauteur            ;ajoute la hauteur
  mov maxrow,ax             ;dernière ligne à prendre en compte
```

```
  xor cx,cx                ;initialise le compteur à 0
  call waitretrace         ;attend le retour de balayage pour
                           ;synchronisation

next_line:
  inc cx                   ;incrémente le compteur

  mov bx,cx                ;calcule couleur 1
  sub bx,y_pos1            ;position relative au départ du copper
  cmp bx,hauteur/2 -1      ;déjà la deuxième moitié ?
  jle copper1_up
  sub bx,hauteur -1        ;alors bx:=127-bx
  neg bx
copper1_up:
  or bx,bx
  jns copper1_ok           ;positif, alors couleur
  xor bl,bl
copper1_ok:
  mov ax,cx                ;calcule la couleur 2
  sub ax,y_pos2            ;position relative
  cmp ax,hauteur/2 -1      ;2ème moitié
  jle copper2_up
  sub ax,hauteur -1        ;alors ax:=127-ax
  neg ax
copper2_up:
  or ax,ax                 ;positif alors couleur
  jns copper2_ok
  xor al,al
copper2_ok:
  mov bh,al                ;bl contient couleur copper 1 / copper 2

  mov ax,bx                ;calcule le recouvrement
  and ax,overlay_masque    ;masque copper 1 ou 2
  or al,al                 ;copper 1 prioritaire
  je Copper1_derriere
  xor bh,bh                ;efface copper 2
copper1_derriere:
```

```
  or ah,ah                  ;copper 2 prioritaire
  je Copper2_derriere
  xor bl,bl                 ;efface copper 1
copper2_derriere:

  xor al,al                 ;sélectionne la couleur 0 dans le DAC
  mov dx,3c8h
  out dx,al

  or bl,bl                  ;si copper 1 noir -> laisser  je bl_0
  add bl,(128-hauteur) / 2;sinon couleur la plus claire possible
bl_0:
  or bh,bh                  ;idem pour copper 2
  je bh_0
  add bh,(128-hauteur) / 2
bh_0:

;attend le retour de balayage horizontal et active les coppers

  cli     ;pas d'interruption, durée TRES critique
  mov dx,3dah               ;registre 1 de l'Input Status
in_retrace:
  in al,dx                 ;attend l'affichage
  test al,1
  jne in_retrace

in_display:
  in al,dx                 ;attend le retour horizontal
  test al,1
  je in_display

  mov al,bl                ;charge la couleur 1
  mov dx,3c9h              ;et l'applique
  out dx,al                ;fixe la part de rouge pour copper 1
  mov al,bh
  out dx,al                ;fixe la part de vert pour  copper 2
  xor al,al
  out dx,al
```

```
    cmp cx,maxrow          ;dernière ligne ?
    jne next_line

    mov dx,3dah            ;oui -> on termine
wait_hret:                 ;attend le retour avant décrochage
    in al,dx               ;sinon la dernière ligne scintille
    test al,1
    je wait_hret

    xor al,al              ;sélectionne la couleur 0 dans le DAC
    mov dx,3c8h
    out dx,al
    inc dx                 ;tous à 0 : noir
    out dx,al
    out dx,al
    out dx,al

    sti
    ret
makecopper endp
code ends
end
```

On détermine d'abord la plus grande des deux ordonnées y pour savoir à partir de quelle ligne on est sûr de ne plus devoir afficher de copper. La procédure pourra être abandonnée à cet instant, et les ressources accaparées rendues à d'autres tâches.

Le compteur de lignes est initialisé à 0 et on attend le signal de retour vertical pour pouvoir commencer à la ligne 0 du moniteur.

Dans la boucle next_line, on calcule à l'aide des ordonnées la position relative de la ligne de balayage par rapport à la ligne de départ du copper qui donne la couleur. Si le rayon se trouve déjà dans la deuxième moitié, la couleur est de nouveau décrémentée pour assurer la symétrie.

Peu importe que le rayon se trouve au-delà ou en deçà de la moitié du copper, une couleur négative est générée en dehors du copper, reprise par la suite et transformée en 0 (noir).

A la suite de ce calcul, bl reçoit la couleur du copper 1 et bh celle du copper 2. En cas de recouvrement des deux coppers, ces deux valeurs doivent être soumises au masque d'overlay. On y arrive ici sans mettre en oeuvre de CMPS (trop lent), en deux branchements seulement. La couleur est combinée par AND avec le masque correspondant de sorte que seul le Copper masqué avec 0ffh compte. Celui qui est masqué par 0 passe automatiquement à l'arrière-plan. Si les deux coppers sont masqués par 0, aucune prééminence n'est décidée et il y a mélange. Le copper masqué par 0ff dès qu'il est affiché à sa position courante, c'est-à-dire s'il possède une couleur différente de 0, efface l'autre copper en mettant sa part de couleur à 0.

A cet endroit, on sélectionne déjà le registre de couleurs du DAC pour gagner du temps au moment du changement effectif de couleur. Il paraît infiniment peu probable qu'un programme résident agisse dans l'intervalle, pourquoi le ferait-il ?

Pour le cas où le copper n'atteint pas la hauteur maximale de 128 lignes (cette hauteur est mémorisée dans la variable de même nom), on prend en considération le reste pour ajouter encore une couleur. On va couper les parties sombres et laisser bien visible le milieu en éclaircissant au maximum le Copper par addition à la couleur du reste précité.

C'est alors que commence la partie critique, ce que dénote la présence de l'instruction CLI. On attend un retour de balayage horizontal approprié en guettant un Display Enable associé à son pendant vertical. On est sûr alors qu'aucune commutation n'aura lieu pendant le temps d'affichage, même pour les ordinateurs lents où le rayon cathodique est déjà entre-temps dans le prochain retour de balayage.

Il faut définir la couleur 0 par le registre Pixel Color Value du DAC. La proportion de rouge est définie par le copper 1, la proportion de vert par le copper 2. Le bleu reste à 0 pour les deux. La boucle commence, à moins que la dernière ligne (définie par la variable maxrow) soit déjà tracée. On attend un retour de balayage et la couleur 0 est définie comme le noir pour tout le reste de l'écran. Il faut attendre une resynchronisation car sinon la commutation tombe pendant l'affichage de la dernière ligne qui est interrompue. On peut ensuite autoriser de nouveau les interruptions et mettre fin à la procédure.

6.9. Un écran qui tremblote comme du pudding : le wobbler

Avec le signal de retour de balayage horizontal, on peut encore se livrer à d'autres effets que les barres de copper. Que se passerait-il par exemple si au lieu de changer la couleur de chaque ligne, on en changeait la position ? En exploitant une table de sinus, on pourrait ainsi imprimer des oscillations à l'ensemble de l'écran, même en mode texte puisque nous manipulons directement les registres du CRTC.

Le mieux pour contrôler l'ordonnée d'une ligne de balayage est d'utiliser le registre 4 du CRTC. Comme le suggère son nom (Horizontal Sync Start), ce dernier détermine l'endroit du retour de balayage. La fin de la synchronisation étant fixée relativement au début, la modification de ce registre conduit à déplacer simplement le retour de balayage sans toucher à sa longueur, ce qui est préférable (si le retour est trop court, certains moniteurs ne suivent plus et l'image dégénère complètement).

La technique du wobbler consiste à attendre une ligne de balayage donnée puis à introduire à chaque ligne de nouvelles valeurs dans le registre 4. Les valeurs en question seront tirées d'une table trigonométrique, créée par la procédure Sin_Gen de l'unité Tools (voir chapitre 2.3 Fonctions mathématiques personnalisées). Vous pouvez évidemment choisir une autre fonction mais, pour un effet de vague, le sinus est parfait.

Il faudra encore une fois veiller avec beaucoup de rigueur au timing de la procédure. Contrairement à ce qui se passait dans le programme Copper, la modification du registre devra avoir lieu pendant que le rayon cathodique génère l'image de l'écran. Dans le programme Copper, le changement de couleur ne perturbait pas le retour de balayage, mais la génération de l'écran : c'est pourquoi la commutation se faisait pendant le retour de balayage. Dans le cas du wobbler, la position du retour joue un grand rôle pendant le retour ou la période de blanc (blank time), alors que le registre que nous convoitons n'a pas d'influence pendant l'affichage proprement dit des données de l'écran. La commutation du registre doit donc s'effectuer cette fois pendant le temps du Display Enable. Il suffit de permuter l'ordre des deux boucles d'attente.

Cet effet fonctionne tout à fait indépendamment du contenu de l'écran puisqu'on intervient au niveau le plus bas dans le timing de la génération de l'écran. Il faudra simplement se souvenir que la valeur par défaut du registre dépend du mode exploité au moment où on crée la table des sinus. En mode texte 3, cette valeur est de 85, en mode 13h ou X elle est de 84. Si vous ne respectez pas cette valeur, l'écran sera déplacé vers la droite ou la gauche, en correspondance avec le point 0 de la table des sinus. Par exemple, si vous prenez 87 comme point 0 en ajoutant une amplitude de 4, le résultat sera

au-delà du domaine admis. Il n'y aura aucun effet, le sinus est désactivé à cet endroit. En Hi-Fi, cet effet s'appelle le clipping.

Rappelons encore une fois que le registre Horizontal Sync Start, comme tous les registres du timing horizontal, est sous la protection du bit 7 du registre 11h du CRTC (Vertical Sync End). Il faut donc d'abord mettre ce bit à 0, en appelant la procédure CRTC_unprotect (Unité ModeXLib). A la fin du programme, la procédure inverse CRTC_protect remet la protection qui est sûrement justifiée du moment qu'elle existe.

Les deux procédures sont très simples et ne demandent pas de commentaire supplémentaire :

```
Procedure CRTC_UnProtect;
Begin
  Port[$3d4]:=$11;                {registre 11h du CRTC (Vertical Sync
                                   End)}
  Port[$3d5]:=Port[$3d5] and not $80
                                  {met à 0 le bit 7 (Protection Bit) }
End;
Procedure CRTC_Protect;
Begin
  Port[$3d4]:=$11;                 {register 11h du CRTC (Vertical
                                    Sync End)}
  Port[$3d5]:=Port[$3d5] or $80 {met à 1 le bit 7 (Protection Bit) }
End;
```

Notre démonstration sera constituée par le programme wobbler. A l'intérieur du programme principal WOBBLER.PAS, on trouve la liaison avec le module en assembleur WOBBLER.ASM.

```
Uses Crt,Gif,ModeXLib,Tools;
const y=100;                      {ordonnée et hauteur, peuvent aussi }
      hauteur=100;                {se présenter sous forme de variable}

Var Sinus:Array[0..63] of Word; {table de sinus, remplie
                                 ultérieurement }
    i:Word;                       {compteur temporaire }

Procedure Make_Wob(wob_pos,wob_hauteur,wob_offset:word);external;
{$l wobbler}
```

```
begin
  TextMode(3);                    {le wobbler fonctionne dans TOUS
                                   les modes vidéos ! }
  For i:=1 to 40 do               {prépare une image test }
    Write('Ceci est un texte de démo ');

  Sin_Gen(Sinus,64,4,83);         {calcule les sinus }
  CRTC_Unprotect;                 {libère l'accès au timing horizontal}
  ReadKey;                        {attend }
  i:=0;
  Repeat
    inc(i);                       {paramètre de déplacement }
    Make_Wob(y,hauteur,i);        {fait des vagues }
  Until KeyPressed;
  CRTC_Protect;                   {protège à nouveau le CRTC }
End.
```

On élabore d'abord une image de test en mode texte 3, puis on forme la table des sinus et on libère l'accès aux registres 0-7 du CRTC (CRTC_unprotect) pour pouvoir manipuler le registre 4. La boucle qui démarre après une frappe de touche se contente d'incrémenter une variable i qui lors de l'appel de la procédure Make_Wob indique l'offset courant à l'intérieur de la table des sinus et provoque ainsi la déformation ondulatoire.

Pour terminer, on remet à 1 le bit de protection du CRTC, ce qui réactive la sécurité.

La procédure Make_Wob proprement dite se trouve dans le module WOBBLER.ASM :

```
extrn WaitRetrace:far

data segment public
  extrn sinus:dataptr          ;table des sinus
data ends


code segment public
assume cs:code,des:data

public make_wob
```

```
make_wob proc pascal wob_pos,wob_hauteur,wob_offset:word
  xor cx,cx                    ;initialise le compteur de lignes
  call waitretrace             ;synchronisation avec le rayon
                               ;cathodique

next_line:
  inc cx                       ;incrémente le compteur de lignes

  mov bx,cx   ;détermine la position à l'intérieur du wobbler
  sub bx,wob_pos
  mov si,bx                    ;mémorise pour la fin

  add bx,wob_offset            ;ajoute l'offset pour provoquer le
                               ;déplacement
  and bx,63                    ;valeurs de 0..63 uniquement
                               ;(dimension du tableau)
  shl bx,1                     ;accès au tableau par mot
  mov bx,sinus[bx]             ;lit la valeur en bx

  cli                          ;inhibe les interruptions, portion
                               ;TRES critique
  mov dx,3dah                  ;sélectionne le registre 1 de l'Input
                               ;Status

in_display:
  in al,dx                     ;attend le retour de balayage
                               ;horizontal
  test al,1
  je in_display
in_retrace:
  in al,dx                     ;attend l'affichage
  test al,1
  jne in_retrace

  cmp cx,wob_pos               ;ligne souhaitée atteinte?
  jb next_line                 ;non -> fixe valeur standard

  mov dx,3d4h                  ;sélectionne le registre 4
```

```
                                ;(Horizontal Sync Start)
  mov al,4                      ;du CRTC
  mov ah,bl                     ;lit le sinus
  out dx,ax                     ;et le reporte

  cmp si,wob_hauteur            ;fini ?
  jb next_line

  mov dx,3dah
attente:
  in al,dx                      ;attend le retour de balayage
                                ;horizontal
  test al,1
  jne attente
  mov dx,3d4h                   ;réinitialise Sync Start
  mov ax,5504h
  out dx,ax
  sti                           ;rétablit les interruptions
  ret
make_wob endp

code ends
end
```

La procédure reçoit les arguments suivants : d'abord l'ordonnée du wobbler et sa hauteur en lignes de balayage. Le troisième argument est constitué par l'offset qui donne en pratique la transposition de la première ligne qui se répercute sur toutes les autres. Si on augmente ce nombre de 1, la vague sinusoïdale se déplace d'une ligne vers le haut. Ainsi, sa modification constante provoque une sorte de mouvement ondulatoire.

Comme dans le programme Copper, on attend la ligne indiquée dans wob_pos. Pour gagner du temps plus tard, la position verticale à l'intérieur du wobbler est calculée avant les deux boucles d'attente de la synchronisation horizontale. Pour accéder à la table des sinus, on additionne d'abord l'offset transmis à la procédure. Le résultat devant se trouver dans l'intervalle de la table, il est combiné par l'opérateur AND au nombre 63 (voir chapitre 2.5 les tableaux circulaires). L'accès se fait sur des éléments constitués par des mots d'où l'instruction shl bx,1. Le résultat est stocké temporairement en BX.

On attend ensuite le début de la période d'affichage, et non le retour de balayage que justement on ne souhaite pas rencontrer. Si la ligne de début

n'est pas encore atteinte, on continue le comptage (jb next_line), sinon on charge le nouveau sinus dans le registre 4 du CRTC.

Si ayant servi au début de registre de mémorisation temporaire, il contient encore l'ordonnée courante par rapport au début du wobbler. On peut donc faire une comparaison pour savoir si on a atteint la hauteur du wobbler. Avant de rétablir le registre dans son état d'origine, il faut encore attendre un peu, sans quoi on risque de saboter l'affichage de la dernière ligne. Il faut en fait guetter le début d'affichage d'une ligne.

6.10. Animations faciles en temps réel : effets de palette

La palette offre d'excellentes possibilités pour modifier l'écran tout entier en peu d'instructions et peu de temps. Elle permet en effet de changer instantanément tous les pixels d'une même couleur.

Fondu de fermeture

L'effet le plus simple que vous pouvez réaliser de cette manière est le fondu de fermeture. Comme dans un film, l'intensité de l'image est progressivement réduite à zéro, dans un délai assez court. Avec la palette, cet effet est assez facile à obtenir. Il suffit de créer une boucle qui diminue toutes les couleurs de 1 et les remet dans la palette. On attend alors le prochain retour de balayage vertical et on poursuit la décrémentation, jusqu'à ce que tout soit noir.

La procédure fade_out du module MODEXLIB.ASM remplit exactement cette fonction :

```
fade_out proc pascal far        ; Fondu de fermeture , dépend du mode
                                ;vidéo
local maxcoul:word              ;couleur de valeur maximale
  mov maxcoul,63
  mov ax,des                    ;charge le segment de destination
  mov es,ax
main_loop:                      ;boucle principale parcourue une fois
                                ;par image
  lea si,palette                ;Source et destination sur palette
  mov di,si
  mov cx,768                    ;768 octets à modifier
```

```
lp:
  lodsb                    ;lit une couleur
  dec al                   ;la décrémente
  jns fixe                 ;si pas encore négative, la réinstalle
  xor al,al                ;sinon 0
fixe:
  stosb                    ;écrit la couleur
  dec cx                   ;compteur d'itérations
  jne lp

  call waitretrace         ;synchronisation
  call setpal              ;prend en compte la palette calculée
  dec maxcoul              ;décrémente la boucle extérieure
  jne main_loop            ;pas fini ? on continue

  ret
fade_out endp
```

Le test de fin de boucle exploite la variable maxcoul, qui n'est rien d'autre qu'un compteur, décrémenté de 63 à 0. Concrètement, ce compteur indique l'intensité maximale présente sur l'écran. Sa valeur initiale est de 63 (proportion maximale d'une composante rouge, verte ou bleu). A mesure qu'on retouche les couleurs, l'intensité est diminuée, jusqu'à atteindre 0 lorsque l'écran se fait tout noir.

Après chargement des pointeurs de source et de destination, la procédure passe par une boucle secondaire qui diminue toute la palette de 1. Chaque couleur est chargée, décrémentée de 1 et réécrite dans la palette, sans qu'elle puisse devenir négative.

La palette ainsi préparée est activée par le moyen de SetPal, avec synchronisation sur le retour de balayage vertical. Il est évidemment possible de réaliser la palette en même temps qu'elle est calculée : il faut alors préparer à l'écriture le registre DAC (port[$3c8]:=0) et écrire à chaque fois sur le port $3c9. L'inconvénient réside dans la durée du calcul des nouvelles couleurs, qui sur les machines les plus lentes dépasse la durée du retour de balayage vertical. On obtient donc des irrégularités dans la partie supérieure de l'écran. Il vaut donc mieux réaliser la palette en une seule fois par SetPal (instruction outsb), ce qui est plus rapide. La boucle principale tourne jusqu'à ce que maxcoul atteigne la valeur 0, plongeant ainsi l'écran dans le noir.

Le programme de démonstration FADE_OUT.PAS montre que cet effet de palette fonctionne aussi en mode texte :

```
uses crt,modexlib;
var i:word;

Begin
  For i:=1 to 40 do              {prépare l'écran de test }
    Write('Ceci est un texte de démonstration ');
  GetPal;                        {charge la table de palette avec la
                                  palette du DAC}
  ReadLn;
  Fade_out;                      {fondu de fermeture de l'écran }
  ReadLn;
  TextMode(3);                   {rétablit l'écran normal }
End.
```

Ce programme est d'abord obligé de prélever la palette dans les registres du DAC. En mode graphique, la palette se trouve déjà le plus souvent dans la variable Palette car le chargeur de GIF l'y a déposée. L'appel de Fade_Out n'est alors plus qu'une affaire de forme.

On prend conscience ici des avantages de la programmation directe du DAC par rapport à l'exploitation du BIOS. Un accès à la palette du mode texte n'est pas possible en mode texte, tandis qu'en mode graphique, il est insupportablement lent. Il ne reste alors plus qu'à se livrer à la manipulation directe des registres.

La direction opposée : le fondu d'ouverture

Le contraire du fondu de fermeture est le fondu d'ouverture qui fait apparaître progressivement une image d'écran. A partir d'un écran noir, l'intensité est augmentée graduellement jusqu'à obtention de la palette réelle de l'écran.

Le fonctionnement de principe est le même que pour la fermeture : les couleurs sont augmentées de 1 à chaque itération, en synchronisation avec le retour de balayage horizontal, jusqu'à retrouver leur valeur d'origine dans la palette associée à l'écran. La condition de terminaison du processus n'est plus une comparaison avec 0 mais avec la valeur maximale attendue.

Le programme suivant appelé FADE_IN.PAS n'a qu'un rôle de démonstration. Nous verrons en effet dans la section suivante que l'ouverture en fondu peut s'effectuer avec une procédure plus générale qui offre davantage de possibilités.

```
uses crt,ModeXLib;
var i,j:word;
    paldest:Array[0..767] of Byte;

Procedure Fade_in(DPal:Array of Byte);
Begin
  For j:=0 to 63 do Begin {64 itérations pour mener à bien le fondu }
    For i:=0 to 767 do  {calcule 768 couleurs }
      If Palette[i] < DPal[i]  {couleur inférieur à la valeur final ?
        Then Inc(Palette[i]); {on l'augmente }
    WaitRetrace;              {synchronisation}
    SetPal;                   {réalise la palette calculée}
  End;
End;

begin
  ClrScr;                     {efface l'écran }
  GetPal;                     {charge la palette avec les valeurs
                               lues dans le DAC }
  Move(Palette,Paldest,768);  {sauvegarde la palette }
  FillChar(Palette,768,0);    {extinction des feux}
  SetPal;                     {réalise la palette noire }

  For i:=1 to 40 do           {prépare l'écran de démo }
    Write('Ceci est un texte de démonstration ');

  ReadLn;
  fade_in(Paldest);           {ouvre l'écran en fondu sur Paldest
                               (palette originale) }
  ReadLn;
  TextMode(3);                {rétablit l'écran normal }
End.
```

La procédure fade_in constitue le coeur de ce programme. On lui transmet comme argument la palette de destination que l'écran doit adopter à la fin du fondu. On suppose qu'à ce moment une palette noire est en cours.

La boucle principale de la procédure est parcourue 64 fois (après quoi la teinte la plus intense -le blanc brillant- est visible). Elle calcule à chaque fois une nouvelle palette et l'envoie à la carte VGA. Le calcul est très simple : chaque

couleur de la palette est testée par rapport à sa valeur de destination. Si elle est inférieure, elle est incrémentée, sinon elle reste à son maximum qu'elle ne peut dépasser.

Le programme principal de FADE_IN.PAS ne fait rien d'autre que de préparer un écran de test pour le faire apparaître ensuite en fondu d'ouverture. Auparavant, la palette courante est lue dans le registre DAC et sauvegardée dans la variable Destpal, la palette de destination. On met ensuite toutes les couleurs en noir, ce qui constitue le point de départ de l'effet de fondu d'ouverture.

La voie universelle : fondre une palette source dans une palette de destination

Jusqu'ici, nous avons créé des fondus d'une palette de couleurs vers le noir ou vice-versa. Ce qui nous manque, c'est de pouvoir passer d'une palette de couleurs à une autre. A première vue, l'entreprise paraît déraisonnable, puisqu'elle conduit naturellement à une falsification des couleurs de l'écran. Erreur ! Abstraction faite de l'application de cette procédure au chapitre 10, nous allons pouvoir créer un très bel effet.

Que pensez-vous par exemple du projet suivant ? Alors qu'une partie de démo touche à sa fin, le dernier écran passe progressivement en noir et blanc. Au premier plan apparaît alors un texte avec les crédits, c'est-à-dire les remerciements.

L'utilité de notre procédure devient manifeste. Après calcul de la palette noir-et-blanc, il suffit de la réaliser progressivement. Un premier problème se pose tout de suite : comment créer une palette noir-et-blanc ? En fait, le Bios possède déjà une fonction prévue à cette fin. Nous ne l'utiliserons pas car elle est top lente et manque de souplesse. Mais nous pouvons en exploiter les idées.

Pour ramener en noir-et-blanc une couleur définie par ses composantes rouge, verte et bleue, il faut additionner les trois parts et les reporter dans les trois couleurs. Si les trois couleurs sont mélangées avec des parts égales, il en résulte toujours un niveau de gris intermédiaire entre le noir et le blanc.

La question est de savoir comment effectuer la somme. Il ne suffit pas de faire intervenir les trois couleurs de la même façon. L'oeil humain ne prête pas la même luminosité aux différentes couleurs. Un point bleu d'intensité maximale apparaît nettement plus sombre qu'un point vert. Un coup d'oeil dans le Bios nous permet d'obtenir un mélange optimal qui assure une traduction fidèle en niveaux de gris. Il faut prendre 30 % de la proportion de rouge, 59 % de la proportion de vert et 11 % de la proportion de bleu. C'est à ce calcul que se livre la procédure Make_bw dans MODEXLIB.PAS :

```
Procedure Make_bw;      {réduit une palette en niveaux de gris }
Var i,sum:Word;         {pondération: 30% rouge, 59% vert, 11% bleu}
Begin
  For i:=0 to 255 do Begin
    Sum:=Round(WorkPal[i*3]*0.3 + WorkPal[i*3+1]*0.59 +
               WorkPal[i*3+2]*0.11);
    FillChar(WorkPal[i*3],3,Sum);  {reporte les couleurs }
  End;
End;
```

La boucle calcule pour chaque couleur la somme pondérée des pourcentages qui est ensuite écrite dans la palette par FillChar. La moyenne pondérée est effectuée avec les opérateurs virgule flottante de Pascal car la vitesse d'exécution n'est pas importante ici. Le reste est simple affaire de calcul de pourcentage.

Le programme de démonstration FADE_TO.PAS présente une application de la nouvelle procédure de fondu :

```
uses crt,ModeXLib;
var i:word;
    origpal,
    DestPal:Array[0..767] of Byte;

begin
  ClrScr;
  GetPal; {charge la palette avec les couleurs courantes du DAC }
  Move(Palette,OrigPal,768);    {sauvegarde la palette }
  Move(Palette,DestPal,768);    {initialise la palette de
                                 destination}

  For i:=1 to 40 do begin       {prépare l'écran de test }
    TextColor(i mod 15);        {allons-y pour les couleurs !}
    Write('Ceci est un texte de démonstration ');
  End;
  Make_bw(DestPal);             {convertit DestPal en noir-et-blanc }
  readkey;
  fade_to(DestPal,1);           {active la palette n/b }
  ReadKey;
  fade_to(OrigPal,1);           {restaure la palette d'origine }
```

```
  ReadLn;
  TextMode(3);                    {retour à l'état normal }
End.
```

On ne fait rien d'autre ici que de fondre en noir-et-blanc un écran rempli de texte multicolore. Une frappe de touche lance le processus et une autre l'inverse, ramenant les couleurs à l'écran.

La palette courante du mode texte est d'abord chargée par getPal dans la variable palette, qui est sauvegardée en OrigPal pour réutilisation ultérieure et reproduite en DestPal pour l'initialisation du calcul noir-et-blanc.

Après affichage de l'écran, un simple appel à Make_bw transforme les couleurs de DestPal. L'écran n'est pas encore modifié car seul le tableau DestPal a été préparé. Après frappe d'une touche, la procédure fade_to proprement dite est invoquée. On lui communique comme argument la palette de destination qui correspond au résultat recherché. On suppose que la palette actuellement affichée par la carte VGA est bien celle contenue dans la variable palette.

La procédure fade_to reçoit un deuxième argument constitué par le pas du fondu. Ce dernier correspond à la vitesse du processus. A chaque génération d'écran, les couleurs de la palette sont réduites ou augmentées d'autant. Donc, plus ce nombre est grand, plus la transition est rapide.

La procédure fade_to se trouve dans le module MODEXLIB.ASM de l'unité ModeXLib.

```
fade_to proc pascal far DestPal:dword, longueur:word, pas:byte
;transforme progressivement la palette "Palette" en "DestPal",
;transmis par Pascal comme Array of Byte !
local maxcoul:word
  mov ax,63                     ;calcule le nombre d'itérations
                                ;nécessaires
  div pas  ;pour atteindre 63
  xor ah,ah
  mov maxcoul,ax                ;mémorise le nombre d'itérations
next_frame:
  les di,DestPal                ;cherche l'offset, Pascal transmet
                                ;les tableaux en mode far !
  lea si,palette                ;cherche l'offset de "Palette"
  mov cx,768                    ;768 octets à traiter
```

```
 suite:
   mov al,[si]              ;lit une couleur de la palette courante
   mov ah,[di]              ;lit la couleur de destpal

   mov bl,ah
   sub bl,al                ;différence de plus
   cmp bl,pas               ; d'un pas ?
   jg plus                  ;-> décrémente
   neg bl                   ;différence de plus d'un pas
   cmp bl,pas               ;en négatif ?
   jg moins

   mov al,ah                ;destination atteinte, on fixe la
                            ;couleur
 ecrire:
   dec cx                   ;décrémente la boucle des couleurs
   je fini                  ;0 ? -> fini
   mov [si],al              ;écrit la couleur dans la palette
   inc si                   ;couleur suivante
   inc di
   jmp suite                ;on continue
 moins :
   sub al,pas               ;décrémentation
   jmp ecrire
 plus :
   add al,pas               ;incrémentation
   jmp ecrire
 fini :                     ;palette calculée
   call waitretrace         ;synchronisation
   call setpal              ;réalise la palette
   dec maxcoul              ;at-on fait 63 itérations?
   jne next_frame           ;non -> suite

   ret
 fade_to endp
```

La première tâche de cette procédure est de déterminer le nombre d'itérations nécessaires pour amener chaque couleur à sa valeur finale. Au pire, une

couleur devra passer de la valeur 0 (composante absente) à 63 (proportion de composante maximale). Avec un pas de 1, il faudra 63 étapes de transition, mais, avec un pas de 2, la moitié suffira. On initialise ainsi le compteur maxcoul avec la valeur 63/pas.

La procédure se poursuit en chargeant en es:di le pointeur qui référence la palette transmis par Pascal. Seul l'offset nous intéresse car la palette est de toute façon dans le registre de données. Mais ce n'est pas une mauvaise chose de charger également le registre ES (à moins que vous trouviez que mov di,destpal+2 soit plus élégant...).

La boucle suite est parcourue 768 fois (256 couleurs * 3 composants). On met dans al la couleur traitée et en ah la couleur de destination. Il faut alors déterminer le sens du fondu : "plus" pour augmenter l'intensité ou "moins" pour la diminuer. Mais une simple comparaison ne suffit pas ici pour obtenir ce qu'on cherche. Supposons que se produise le cas suivant : valeur initiale de 0, valeur finale de 15, pas de 2. Après 7 itérations, la couleur courante est de 14, elle est trop petite, il faut l'augmenter. Elle passe alors à 16, qui est excessif, et qu'il faut ramener à 14. Ce petit jeu peut continuer longtemps, l'entrée de palette fluctuant indéfiniment entre deux valeurs. Lorsque le fondu est très rapide, avec un pas élevé, l'esthétique est déplorable.

La solution consiste à ne pas rechercher d'égalité parfaite, ni à pratiquer une comparaison simplette, mais à tester si la valeur courante s'est approchée de la destination à une distance de moins d'un pas. On forme cette différence dans BL. Si elle est positivement supérieure au pas (JG, jump if greater), la couleur se trouve encore loin de sa valeur finale et il faut la descendre. Si elle est inférieure à l'opposé du pas (comparaison avec neg bl), il faut la remonter. Si aucune de ces conditions n'est remplie, le but est atteint et les deux valeurs peuvent être rendues égales en compensant la légère différence.

Que l'on passe par l'étiquette "moins" pour soustraire la valeur d'un pas, par "plus" pour l'ajouter à la couleur courante, ou que l'on équilibre les valeurs, tous les chemins mènent à l'étiquette écrire qui reporte la nouvelle couleur dans la palette et avance les pointeurs. Lorsque les 768 entrées ont été traitées, un branchement permet de sortir de la boucle inférieure pour transférer le contrôle à l'étiquette fini. On réalise alors la palette, après synchronisation. Pour le cas où maxcoul ne serait pas nul, la palette suivante est recalculée (next_frame, c'est-à-dire génération de l'image suivante).

Prêt pour le cinéma : d'une image à l'autre

Il est déjà agréable, lorsqu'on passe d'une image à l'autre, de voir la première s'évanouir progressivement puis de voir apparaître lentement la seconde. C'est là ce que nous permettent de faire les procédures de fondu étudiées précédemment. Mais pour avoir des transitions véritablement profession-

nelles, on n'échappe pas à la nécessité du fondu enchaîné qui passe par le mélange de deux écrans.

Il existe Entre-temps de nombreux logiciels de Morphing qui font des choses merveilleuses mais qui présentent l'inconvénient de travailler sur des images précalculées. L'effet réclame un temps de calcul préalable très important pour préparer les images qui, au demeurant, prennent beaucoup de place sur le disque dur et en mémoire centrale et doivent par ailleurs être copiées péniblement dans la mémoire d'écran. C'est donc là une solution pour amateur, le "pro" calculant ses fondus enchaînés en temps réel !

Pour les images à base de palette - et les modes graphiques que nous étudions reposent tous sur l'usage de la palette - le fondu enchaîné se ramène à une transition de palette (avec la procédure fade_to présentée précédemment). Seules les métamorphoses complexes mettant en jeu des mouvements de différentes parties d'image nécessitent le recours aux logiciels de Morphing.

Nous allons donc à nouveau transformer progressivement une palette en une autre. Mais comme les deux images sont superposées et qu'il n'est pas question de modifier les données graphiques (ce qui est du ressort du morphing) pour une raison de vitesse, les même données devront représenter différentes images en fonction de la palette. Au début, c'est la palette source qui est active, et les données représentent l'image source. A la fin, c'est la palette de destination qui est active, et les même données qui représentaient l'image source représentent maintenant l'image de destination. Les étapes intermédiaires sont assurées par la procédure fade_to.

Cette façon de procéder soulève deux questions :

1. Comment manipuler les données de l'image pour qu'elles représentent, selon la palette utilisée, soit l'image source soit l'image de destination?
2. Comment doivent être constituées les palettes pour pouvoir générer des images différentes à partir de données semblables ?

Répondons d'abord à la première question : ce type de fondu est fondamentalement différent de celui que nous avons traité précédemment. Jusqu'ici, en effet, c'était soit la couleur de destination qui était la même pour tous les pixels (le noir pour le fondu de fermeture), soit la couleur de départ (fondu d'ouverture). Dans ce nouveau problème, chaque couleur doit être transformée en une autre. Certains pixels rouges sont appelés à devenir verts, d'autres qui sont rouges doivent évoluer vers le bleu.

Si vous avez de bons souvenirs en statistiques (stochastique, analyse combinatoire), vous vous rappellerez peut-être ce que signifie la variété des couleurs observées. Si on combine un certain nombre de couleurs avec le même nombre d'autres couleurs, le nombre de possibilités est égal au carré du nombre de

couleurs. Chacune de ces combinaisons doit être prise en compte au moment du fondu, et doit donc correspondre à une entrée de la palette utilisée.

Pour transformer une image bicolore en une autre, il faudra donc 4 entrées de palette (couleur 0 en couleur 0, donc sans changement, couleur 0 en couleur 1, couleur 1 en 0 et couleur 1 en 1). Avec des images en quatre couleurs, le nombre d'entrées de palette atteint déjà 16.

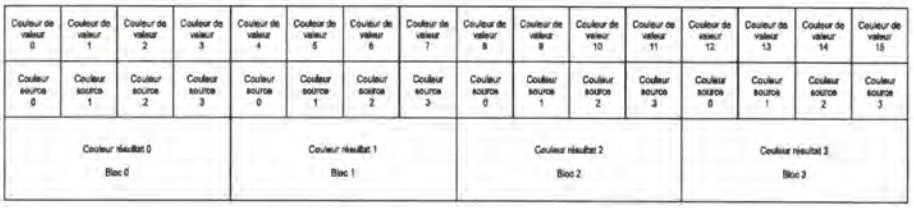
Mais sur la carte VGA, la palette ne comporte que 256 couleurs. Le nombre maximal de couleurs qui peuvent être fondues sans modification des données de l'image est donc de 16. Comme nous le verrons un peu plus loin, lorsqu'on veut effectuer plusieurs fondus successifs sans scintillement, ce nombre se réduit en fait à 15.

Ces réflexions combinatoires nous permettent de définir un mode opératoire pour mélanger les images. Chaque combinaison doit être représentée. On introduit donc N blocs de N entrées, N étant le nombre de couleurs par image. Le numéro de bloc correspond à la couleur de destination, tandis que l'indice à l'intérieur du bloc correspond à la couleur de la source. On pourrait imaginer la construction inverse mais on accroîtrait alors la complexité du Reset dont nous parlerons plus loin. Pour connaître un numéro de couleur, on applique la formule :

```
N° couleur := Couleur de destination * Nombre de couleurs + Couleur de
la source
```

Ce principe vous dit peut-être quelque chose. C'est ainsi qu'on convertit en effet un octet hexadécimal en son équivalent décimal : quartet de poids fort (chiffre supérieur) * 16 + quartet de poids faible. Si le nombre de couleurs est égal à 16, on disposerait là d'un moyen d'accélérer les calculs : il suffirait de mettre la couleur de destination dans le quartet supérieur et la couleur de la source dans le quartet inférieur d'un octet, et on aurait la couleur à utiliser. Mais nous ne pouvons pas exploiter ici 16 couleurs, le nombre maximal étant de 15, de sorte que nous en resterons à une lente et pénible multiplication traditionnelle. Ce n'est pas trop grave car la partie de programme qui contient la multiplication n'est pas critique du point de vue du temps d'exécution, au pire elle ne fait que différer légèrement l'instant où débute réellement le fondu.

Le schéma suivant explique encore une fois la formation des blocs dans le cas où le fondu enchaîné porte sur deux images quadricolores.



Palette de couleurs pour fondu enchaîné

Ce schéma contient la réponse à notre deuxième question sur l'organisation de la palette. La structure des blocs correspond exactement à celle de la palette. La palette source (de taille 1 bloc) est copiée aussi souvent que le demande le nombre de couleurs, il en résulte des blocs sources identiques. De l'autre côté, la palette de destination est expansée : chaque couleur est répétée plusieurs fois. Les blocs de cette palette sont donc homogènes.

L'effet sur lequel nous travaillons ne nécessite pas l'intervention de plusieurs pages d'écran à calculer ou à recopier : les manipulations peuvent se faire sur l'écran lui-même.

Il faut cependant noter que l'image visible à l'écran ne change pas pendant la génération et le mélange de palettes, sinon il se produirait des scintillements. En conséquence, la palette active ne doit être modifiée que dans des zones dont les couleurs ne sont pas présentes sur l'écran. Par ailleurs, si on modifie les données de l'image, la palette doit être déjà prête pour que les pixels modifiés retrouvent leur couleur d'origine. Si un pixel rouge, qui a par exemple la couleur 3, doit recevoir la couleur 23, il faut s'assurer que la couleur 0 contient déjà du rouge pour que rien ne change à l'écran.

Le fondu enchaîné devra donc respecter l'ordre suivant des procédures :

1. Préparer les palettes, ne modifier que les couleurs inactives
2. Mélanger les données de l'image, aucune modification n'est visible
3. Réaliser le fondu enchaîné des palettes

Comme nous ne devons modifier que les zones de palettes inactives, nous allons introduire un bloc de couleurs supplémentaires. Ce bloc contiendra d'abord les couleurs source, issues de l'image source. Comme une image source quadricolore utilise normalement les couleurs 0 à 3, c'est là que se trouve ce nouveau bloc. Pour générer la palette source (donc ici copier quatre fois le bloc), ce bloc sert de palette de départ et est reproduit en quatre exemplaires. Le bloc lui-même n'étant pas modifié, rien ne se remarque à l'écran.

Ce bloc que nous appelons bloc de Reset a encore une autre mission à remplir : à l'issue du fondu, l'image visible a une palette bien plus grande que l'image d'origine, le nombre de ses couleurs ayant été élevé au carré. Si on désire effectuer un fondu enchaîné avec une autre image, le nombre de couleurs présentes est trop élevé, nous avons dit que nous en autorisions 15. Mais les blocs de destination étant homogènes, toutes les couleurs d'un bloc ont le même contenu, ce qui nous permet de les remplacer. Ainsi, dans notre exemple, nous n'avons finalement que 4 couleurs sur l'écran.

Pour réduire ces couleurs à une palette acceptable, on réutilise le bloc de Reset qui doit en fait contenir la palette source. C'est pourquoi on l'appelle bloc de Reset : les données expansées de l'image sont réduites à ce bloc, donc en quelque sorte réinitialisées. Après le fondu, ce bloc contient la palette de destination proprement dite, de sorte qu'il n'y plus qu'à recalculer les données de l'image.

Ce calcul est en réalité très simple car il suffit d'annuler l'expansion pour revenir à la couleur de destination effective. Une division par le nombre de couleurs donne le numéro de bloc et par conséquent la couleur de destination, à condition que l'on soustraie le bloc de Reset comme dans toutes ces opérations.

L'existence du bloc de Reset constitue la raison pour laquelle nous avons été obligé de limiter à 15 le nombre de couleurs. En plus des entrées de palettes nécessitées par le fondu (n^2, n=nombre de couleurs utilisées), il faut prévoir un bloc de Reset de n couleurs, ce qui finalement fixe à n*n+n=n*(n+1) le nombre d'entrées de palette nécessaires. Avec 16 couleurs, on aurait 16*17=272 entrées de palette, ce dont la carte VGA ne dispose pas. Pour finir, nous sommes donc capables de faire un fondu enchaîné avec deux images de 15 couleurs, ce qui demande 15*16=240 entrées de palette.

Il est parfois judicieux de réduire encore davantage ce nombre de couleurs pour libérer des entrées tout en haut de la palette à l'intention de parties d'image statiques. Il s'agira par exemple d'un logo fixe, situé au-dessus d'un texte soumis à fondu. L'image statique devra exploiter la partie supérieure de la palette car l'effet de fondu accapare déjà toute la partie inférieure.

Pour mélanger les couleurs des deux images, on applique la formule expliquée plus haut qui donne la nouvelle entrée de palette et on l'écrit dans la mémoire de la carte VGA. Rappelons qu'il suffit d'écrire un autre numéro de bloc, l'indice à l'intérieur du bloc (qui correspond à la couleur source) ne variant pas. Les blocs ayant à ce stade le même contenu, l'écran ne change pas pour le moment.

Pour terminer, il faut fondre la palette actuelle, issue de la multiplication de la palette source, avec la palette de destination expansée, de façon que les blocs deviennent homogènes et contiennent la palette de destination, ce qui

rend visible l'image de destination. Ce n'est qu'après cette opération que les modifications effectuées deviennent apparentes, de sorte que c'est là la seule partie dont la vitesse d'exécution s'avère critique. Nous utiliserons malgré tout la procédure fade_to qui n'a pas présenté de problème de vitesse. A chaque image générée, il faut recharger les registres de palette ce qui est plus rapide que de modifier les données graphiques elles-mêmes.

Pour concrétiser les idées, voici d'abord une première application du fondu enchaîné. Ce programme appelé FADE_OVE.PAS se sert de l'unité FADE.PAS décrite plus loin.

```
uses CRT, ModeXLib,gif,fade;
Var pic1_pal,                    {palettes des deux images}
    pic2_pal:Array[0..767] of Byte;
    pic1,                        {contient la première image }
    pic2:Pointer;                {deuxième image , égal à vscreen}

Begin
  Init_Mode13;                   {active le mode 13h }
  Screen_off;                    {éteint l'écran pendant le chargement }
  LoadGif('echec');              {charge la première image }
  GetMem(pic1,64000);            {mémoire pour la première image }
  Move(vscreen^,pic1^,64000); {sauve l'image }
  Move(Palette,pic1_pal,768); {et la palette }
  Show_Pic13;                    {affiche cette image }

  LoadGif('caisse');             {charge l'image suivante en vscreen^ }
  pic2:=vscreen; {définit pic2 comme pointeur }
  Move(Palette,pic2_pal,768); {sauvegarde la palette }

  Move(pic1_pal,Palette,768); {active la palette de la première
                               image }
  SetPal;                        {et la réalise }
  Screen_on;                     {rallume l'écran }

  ReadLn;                        {attente}
  FonduEnch(pic2,pic2_pal,0,0,200);
                                 {fondu enchaîné avec l'image 2 }

  fade_ResetPic(0,200);          {prépare un nouveau fondu }
```

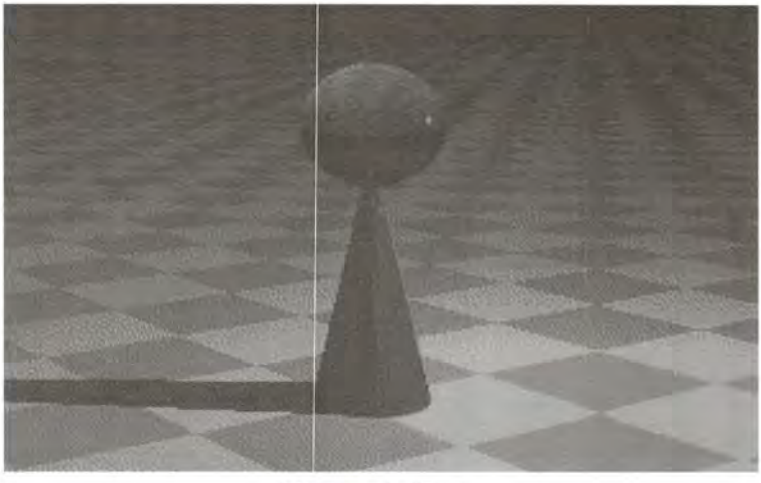
```
  ReadLn;
  FonduEnch(pic1,pic1_pal,0,0,200);
                              {fondu enchaîné avec l'image 1 }
  ReadLn;
TextMode(3)
End.
```

Ce programme réalise un fondu enchaîné entre deux images GIF totalement différentes. La variable pic1 pointe sur les données graphiques de la première image, situées en mémoire centrale (ce pointeur doit d'abord être initialisé). Pic2 pointe sur la deuxième image, mais on tire profit de ce qu'il existe déjà un bloc alloué de la bonne taille en mémoire centrale (vscreen). Pour une plus grande clarté du programme, on a cependant maintenu comme synonyme le pointeur pic2.

Les deux palettes individuelles sont mémorisées dans les variables Pic1_Pal et Pic2_Pal.

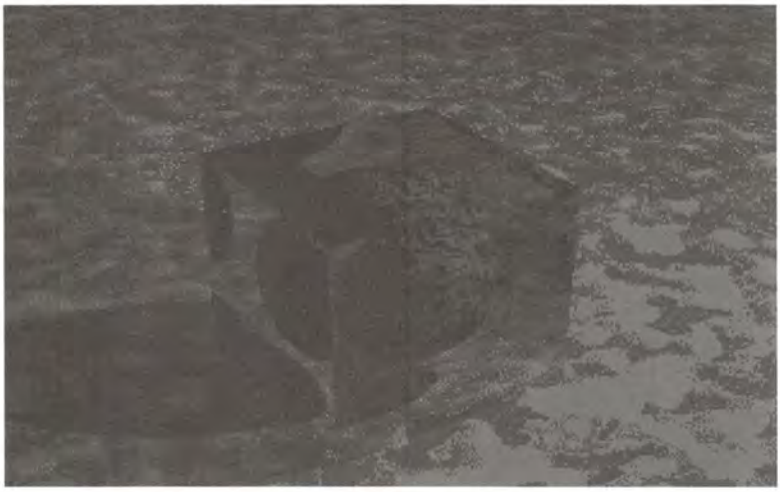
Après mise en service du mode 13h, l'écran est désactivé car la palette va changer au moment du chargement ce qui donne des effets désagréables.

Après chargement et allocation de mémoire, la première image est déplacée à l'adresse Pic 1. Sa palette est sauvegardée en Pic1_Pal.



L'image de départ

Une fois cette première image transférée dans la mémoire vive de la carte VGA, c'est au tour de la deuxième d'être chargée. Elle se trouve à l'adresse vscreen pendant tout le déroulement de la suite du programme. On introduit le pointeur synonyme pic2 qu'une affectation fait pointer sur la même adresse.



L'image d'arrivée

Lorsque la deuxième palette a été sauvegardée dans la variable prévue à cet effet, la première palette est activée (recopiée dans palette) et envoyée à la carte VGA (setpal). L'écran peut alors être rallumé.

Le fondu enchaîné proprement dit est pris en charge par la procédure FonduEnch qui reçoit comme arguments : l'adresse de l'image de destination, sa palette, l'ordonnée à l'intérieur de l'image de destination, l'ordonnée où l'image de destination est à copier et la hauteur de l'image de destination. Les trois derniers arguments ne sont pas utiles pour ce programme et se réfèrent simplement à une image entière. Les deux premiers arguments concernent dans notre exemple la deuxième image, celle qui apparaît progressivement.

Passage entre les deux images par un fondu enchaîné

Après le fondu, l'image est réinitialisée pour permettre le renouvellement de l'effet. On transmet également une ordonnée et une hauteur à la fonction mais ces indications ne jouent aucun rôle dans le cas présent. Pour terminer, on refait un fondu enchaîné vers la première image.

L'instruction centrale de ce programme est l'appel à FonduEnch. Cette procédure rassemble toutes les tâches nécessaires à l'obtention de l'effet et se trouve dans l'unité Fade.

```
Unit fade;
{pour faire un fondu enchaîné assurant la transition d'une image
 (ou partie d'image) déjà affichée à une nouvelle }

Interface
Uses ModeXLib;

Var Colors:Word;                {nombre de couleurs par image }

Procedure fade_ResetPic(y,Hauteur:Word);
Procedure FonduEnch(Pic:Pointer;Pal:Array of Byte; Start,y,
                    Hauteur:Word);

Implementation
Var i,j:Word;                   {compteurs temporaires }
    Dest_Pal:Array[0..768] of Byte;  {palette de destination
                                      temporaire }

Procedure fade_set(Source:Pointer;Start,y,Hauteur:Word);external;
{"mélange" la source avec la mémoire VGA }
{la source est exploitée à partir de la ligne Start
et la mémoire VGA à partir de la ligne y avec une hauteur "hauteur" }

Procedure fade_ResetPic(y,Hauteur:Word);external;
{prépare une image obtenue par fondu enchaîné à
un nouvel effet de fondu enchaîné}
{réduit les couleurs de "Colors^2" à "Colors" }
{ici aussi y=ligne en mémoire VGA, Hauteur=Hauteur de la zone à
 traiter }
{$l fade}
```

```
Procedure fade_CopyPal;
{fabrique Colors^2 exemplaires de la palette (multiplication du
 bloc 0 non homogène) }
Begin
  For i:=1 to Colors do
   Move(Palette[0],Palette[i*3*Colors],Colors*3);

End;

Procedure fade_expans(Var Pal:Array of Byte);
{dilate la palette à Colors^2 (chaque couleur est répétée
 individuellement)}
{forme des blocs homogènes à partir des couleurs 0..Colors-1 }
Begin
  For i:= 0 to Colors-1 do      {traite chaque couleur }
    For j:=0 to Colors -1 do    {et la réécrit Colors fois }
      Move(Pal[i*3],Pal[(i+1)*3*Colors+j*3],3);
End;

Procedure FonduEnch(Pic:Pointer;Pal:Array of Byte; Start,y,
                    Hauteur:Word);
{fait un fondu enchaîné de l'image
actuellement visible vers Pic (de palette Pal),
début à la ligne "Start" de Pic, copie "hauteur" lignes en ordonnée
 y de l'image actuelle}
Begin
  WaitRetrace; {synchronisation}
  fade_CopyPal; {multiplie les blocs de la palette courante }
  SetPal; {réalise cette nouvelle palette }
  Move(Palette,Dest_Pal,768);  {pour conserver des parties de la
                                palette d'origine }
  Move(pal,Dest_Pal,Colors*3); {charge la palette de destination }
  fade_expans(Dest_Pal);       {expanse les blocs de la palette de
                                destination }
  fade_set(pic,start,y,hauteur);
                               {mélange la nouvelle image }
  fade_to(Dest_Pal,1);         {et lance le fondu enchaîné}
End;
```

```
Begin
  Colors:=15;                    {valeur de défaut uniquement !}
End.
```

Cette unité utilise la variable globale Colors qui indique le nombre de couleurs exploitées par chacune des images. Si on réalise un fondu enchaîné avec des images à 15 couleurs, cette variable aura la valeur 15 qui est d'ailleurs la valeur par défaut.

L'ensemble du processus est piloté par la procédure FonduEnch. Pour éviter des scintillements, on commence par attendre le retour de balayage vertical. Puis la palette source, qui est la palette courante (variable palette), est préparée au fondu par démultiplication et réalisée (pas de changement à l'écran, seules des entrées de palette inactives ayant été modifiées). Les deux instructions Move qui suivent construisent la palette de destination. La palette courante est prise comme point de départ pour que les éléments d'image statiques qui utilisent les dernières entrées soient également représentés dans la palette de destination. Ensuite, on insère dans la partie inférieure la palette transmise par pal (entrées Colors à 3 octets). Cette zone inférieure est alors expansée par appel de fade_expans pour préparer le fondu.

La procédure fade_set sert à mélanger les données graphiques proprement dites de façon qu'elles représentent l'image source ou l'image de destination selon la palette associée. Pour terminer, on déclenche réellement le fondu enchaîné en faisant appel à la procédure fade_to : l'image de destination apparaît alors progressivement sur l'écran.

L'unité contient deux procédures Pascal internes appelées fade_copyPal et fade_expans. Leur rôle n'est pas déterminant du point de la vitesse d'exploitation, c'est pourquoi on les a maintenues en Pascal.

Fade_CopyPal prépare la palette source (contenue dans Palette) en fabriquant Colors exemplaires du bloc de Reset (couleurs 0 à colors -1). La boucle ne fait rien d'autre que de copier Colors fois Colors*3 octets aux emplacements prévus.

La procédure fade_expans est un peu plus compliquée. Chaque couleur doit être agrandie sur un bloc. On utilise deux boucles imbriquées. La boucle extérieure (compteur i) prend soin de dilater les couleurs (de 0 à Colors-1) dans la zone prévue. A l'intérieur de cette boucle se trouve la boucle j qui, pour chaque couleur, produit Colors copies qui se trouvent l'une derrière l'autre dans un bloc. Un bloc qui correspond à la couleur en cours de traitement est rempli avec cette couleur. L'instruction Move s'occupe de ce remplissage : elle copie la couleur active (située à la position de palette i*3) dans chaque position (j*3) à l'intérieur du bloc courant ((i+1)*3*Colors).

L'unité fait aussi appel à deux procédures en assembleur tirées du module FADE.ASM : fade_set et fade_ResetPic.

```
data segment public
  extrn colors:word
data ends

code segment public
assume cs:code,des:data
public fade_set,fade_ResetPic


col db 0                        ;segment de code pour Colors


fade_set proc pascal near source:dword, start:word, y:word,
 hauteur:word
  mov ax,colors                 ;reporte Colors dans la variable de
                                ;segment de code
   col
  mov col,al
  push des
  mov ax,word ptr Source + 2    ;des:si doit pointer sur l'image
                                ;source
  mov des,ax
  mov si,word ptr Source

  mov ax,320                    ;ajoute l'adresse de début à
                                ;l'intérieur de l'image
    source
  mul start
  add si,ax

  mov ax,0a000h                 ;es:di doit pointer sur 0a000:0
                                ;(destination)
  mov es,ax
  mov ax,320                    ;ajoute l'adresse de début à
                                ;l'intérieur de l'image de destination
  mul y
```

```
  mov di,ax

  mov ax,320                    ;convertit la hauteur en nombre
                                ;d'octets
  imul hauteur
  mov cx,ax

lp:                             ;boucle principale
  lodsb                         ;valeur de destination en al
  mul col                       ;calcule la nouvelle couleur
  add al,es:[di]                ;ajoute la valeur actuelle
  add al,col
  stosb                         ;et écrit le résultat

  dec cx                       ;tous les pixels copiés ?
  jne lp

  pop des
  ret
fade_set endp

fade_ResetPic proc pascal far y:word, hauteur:word
  mov ax,0a000h                 ;es:di doit pointer sur l'adresse VGA
                                ;0a000:0
  mov es,ax

  mov ax,320                    ;tient compte de l'ordonnée y
  mul y
  mov di,ax

  mov ax,320                    ;calcule le nombre d'octets à traiter
  mul hauteur
  mov cx,ax
res_lp:
  mov al,es:[di]                ;lit une valeur
  xor ah,ah                     ;annule ah pour la division!
  div byte ptr colors           ;calcule le numéro de bloc
  dec al                        ;extrait le bloc de Reset
```

```
    stosb                       ;écrit le résultat

    dec cx                      ;tous les pixels traités?
    jne res_lp                  ;non, on continue

    ret
fade_ResetPic endp

code ends
end
```

Ces procédures recourent aussi à la variable globale Colors, qui se trouve ici dans le segment data. Entre-temps, fade_set ne peut plus y accéder à cause de la variation de des. C'est pourquoi on institue ici une variable de segment de code col.

La procédure fade_set copie d'abord le contenu de Colors dans Col, après quoi des est libéré pour recevoir le pointeur de destination (partie segment). Si est chargé avec l'offset de l'image de destination. Mais il faut tenir compte de l'ordonnée de début Start, qui indique la ligne à partir de laquelle on souhaite lire l'image. L'offset de cette ligne est calculé par multiplication par 320 et ajouté à si.

Le pointeur es:di reçoit alors l'adresse de la mémoire VGA. Le segment est tout simplement le segment où débute la mémoire vive de la carte, l'offset se calculant comme précédemment à l'aide d'une multiplication de l'ordonnée y par 320.

Le registre cs est aussi muni d'une valeur appropriée : on y multiplie la hauteur par 320. La procédure passe ensuite dans la boucle principale lp.

Elle y lit la valeur de destination tirée de l'image de destination (des:di) qui correspond au numéro de bloc à utiliser. Il faut donc la multiplier par le nombre de couleurs. Comme offset à l'intérieur du bloc, on lit la couleur de l'image source tirée de la mémoire d'écran et on l'ajoute.

Ce résultat est réécrit et, après décrémentation et test, la boucle recommence.

La deuxième procédure de cette partie en assembleur, appelée fade_RestPic, sert à réduire à Colors le nombre de couleurs d'une image amenée par fondu enchaîné. A l'aide du numéro de bloc, qui correspond à la couleur de destination, les couleurs sont remises dans le bloc de Reset avec les numéros 0 à Color -1.

On commence par charger en es:di un pointeur sur l'ordonnée y de la mémoire VGA. Cx est initialisé avec la hauteur de façon à indiquer le nombre d'itérations à effectuer.

La boucle principale de cette procédure res_lp lit à chaque fois une valeur dans la mémoire d'écran, calcule son numéro de bloc avec une division par le nombre de couleurs et réécrit le résultat. Il faut évidemment tenir compte du bloc de Reset : les numéros de blocs s'étendent de 1 à Colors, et les couleurs doivent être réduites à un intervalle compris entre 0 et Colors -1. La boucle finie, la procédure est terminée.

Notre exemple porte sur une image complète de 15 couleurs transformée en une autre, le fondu enchaîné impliquant presque la totalité de la palette des couleurs. Si on réduit le nombre de couleurs d'origine à 14, la partie supérieure de la palette contient des couleurs qui ne sont pas touchées par l'opération. Ces couleurs peuvent être exploitées pour afficher des parties d'images statiques qui restent inchangées pendant le fondu.

On peut utiliser cette possibilité pour afficher par exemple les crédits d'une démo. Pour chaque section de démo, on fait un fondu d'ouverture sur une petite image (par exemple une copie d'écran) tandis que dans la moitié inférieure les textes avec les crédits correspondants se suivent en fondu enchaîné.

Le programme FADE_TXT.PAS montre la manière de procéder :

```
Uses Crt,Gif,ModeXLib,Fade;
Var
  Text_Pal:Array[0..767] of Byte;
  i:word;
 Begin
  Init_Mode13;                  {initialise le mode 13h }
  Screen_Off;                   {désactive l'écran pendant le
                                 chargement }
  LoadGif('vflog210');          {charge la partie statique }
  Move(Palette[210*3],          {et enregistre sa participation à la
                                 palette (couleurs 210..255)}
    Text_Pal[210*3],46*3);
  Show_Pic13;                   {copie en mémoire VGA l'image
                                 statique }
   LoadGif('texte');             {charge une image avec des textes }
  Move(Palette,Text_Pal,14*3); {et enregistre sa participation à la
                                 palette (couleurs 0..13)}
```

```
  Move(Text_Pal,Palette,768);  {réalise la palette préparée }
  SetPal;

  Move(vscreen^,               {le premier texte peut être
                                directement copié }
    Ptr($a000,160*320)^,19*320);{sur l'écran }
  Screen_On;                   {on réallume l'écran }
  Colors:=14;                  {dans ce programme les images ont 14
                                couleurs !}

  For i:=1 to 6 do Begin       {ouvre en fondu les 6 autres textes }
    Delay(500);                {temps laissé à la lecture }
    FonduEnch(vscreen,         {fondu enchaîné sur l'image suivante
                               à l'ancienne position (y=160) }
      text_pal,i*20,160,19);
    Fade_ResetPic(160,19);     {réinitialisation }
    If KeyPressed Then Exit;   {possibilité d'interruption }
  End;
  Readln;
TextMode(3);
End.
```

L'image fixe de départ

Ce programme charge d'abord en mémoire d'écran la partie statique et écrit sa contribution à la palette globale dans la variable Text_Pal. L'image avec les textes est ensuite chargée dans vscreen et sa contribution à la palette est également chargée en Text_Pal. La palette ainsi constituée est ensuite chargée et réalisée.

Le texte qui va s'afficher à l'écran

La première partie du texte est ensuite copiée directement dans la mémoire d'écran, car il n'est pas nécessaire ici d'avoir recours à un fondu enchaîné. Une fois l'écran réactivé, la boucle principale commence son travail. Les six autres parties de texte sont parcourues avec la séquence suivante : on attend d'abord une demi-seconde pour que l'utilisateur ait le temps de lire. Puis le texte est soumis à un fondu enchaîné avec indication de coordonnées à l'appel de FonduEnch. La copie de l'image de destination commence à la ligne i*20 car les textes de notre exemple commencent aux ordonnées 0,20,40, etc...Tous ont une hauteur de 19 pixels et apparaissent à la ligne d'écran 160.

Les mêmes coordonnées sont reprises dans l'appel suivant à Fade_ResetPic. Un test de clavier (structurellement contestable) permet d'interrompre le processus.

Résultat du programme Fade_TXT :
le texte apparaît en fondu enchaîné en bas de l'image fixe

Lorsqu'on développe ses propres images pour cet effet, il faut veiller à subdiviser correctement la palette. Les images à enchaîner ne peuvent avoir plus de 15 couleurs. Les images qui présentent davantage de couleurs doivent être réduites. Dans tous les cas, ces couleurs doivent se trouver dans la partie basse de la palette (0.. nombre de couleurs-1).

Le nombre de couleurs de l'image statique se calcule comme suit :

```
Couleurs statiques=256-couleurs dynamiques*(couleurs dynamiques+1)
```

Les couleurs statiques doivent toujours être situées dans la partie supérieure de la palette car c'est la seule zone qui reste inchangée pendant le fondu enchaîné.

Si on souhaite fusionner des images de 15 couleurs, il reste 256-15*16=16 couleurs pour les images statiques en fin de palette (couleurs N°240-255). Avec des images de 13 couleurs il reste 256-13*14=74 couleurs (couleurs N°182-255) .

L'alternative rapide : animation par rotation de palette

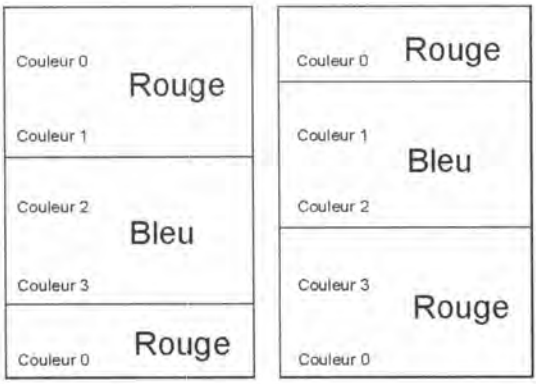
Comme nous l'avons vu, de simples remaniements de palette permettent de modifier d'un seul coup de grandes zones d'écran, sans que la programmation soit importante. Cet avantage des modes graphiques basés sur les palettes peut être exploité pour de vraies animations, où certaines zones d'écran ne présentent pas seulement des successions de fondus, mais de véritables mouvements.

Comment est-ce possible ? Prenons comme exemple très simple un point rouge qui doit être déplacé vers la droite de dix pixels. On trace côté à côte dix points avec des couleurs qui se suivent dans la palette, par exemple les couleurs 0-9. Si on prend du rouge comme couleur 0, et du noir pour les 9 autres couleurs, seul le premier pixel est visible, les autres se confondant avec le fond d'écran. Si on décale la palette d'un élément, de façon que la couleur 1 soit le rouge, et que les couleurs 2-9 ainsi que la couleur 0 soient constituées par du noir, le deuxième pixel devient visible et le premier disparaît.

Si on continue de décaler la palette, le point lumineux se déplace sur l'écran de gauche à droite, sans qu'un seul octet ait été modifié dans la mémoire d'écran (après l'initialisation des dix points). Lorsque l'entrée rouge est parvenue à l'extrémité supérieure (couleur 9), on peut poursuivre le décalage en revenant à la couleur 0, ce qui donne une animation cyclique. Dans ce cas, le point va de gauche à droite puis réapparaît brusquement à gauche.

Pour un point isolé, cette manipulation ne présente pas grand intérêt car, au lieu de modifier deux octets dans la mémoire d'écran (effacer l'ancien point et dessiner le nouveau), on change dix couleurs en accédant à un port particulièrement lent. Mais la rotation de palette est bien plus efficace sur des formes étendues et/ou compliquées. Un grand nombre de points ou une description mathématique compliquée de la zone à déplacer peuvent parfaitement justifier l'intervention d'une rotation de palette.

Mais avant de déplacer de grandes surfaces, il faut évidemment les préparer en conséquence. Les zones en question ne doivent pas être constituées d'une seule couleur de palette. Elles doivent présenter dans le sens du déplacement un gradient de couleurs. Concrètement, le schéma suivant montre à quoi pourrait ressembler une telle structure :



Organisation des couleurs en prévision d'un déplacement

Ici, des cases d'une hauteur de 2 pixels défilent vers le haut. Le principe serait le même pour des cases plus grandes ou des structures plus complexes. L'image de l'exemple contient deux cases et demie, colorées alternativement en rouge et en bleu. Le nombre de cases ne joue aucun rôle, l'image peut être étendue librement vers le haut ou vers le bas, pourvu que l'ordre des couleurs soit respecté. On peut également introduire des fractions de case.

L'organisation de l'ensemble de l'image est périodique, autrement dit la structure se répète par groupe de deux cases remplies avec un gradient de couleurs contiguës allant de la couleur 0 à la couleur 3. Dans la palette, les couleurs 0 et 1 sont remplies avec du rouge, les couleurs 2 et 3 avec du bleu, de façon que les blocs soient reconnaissables comme tels. Il est recommandé de faire ces préparatifs avec un logiciel de dessin, de façon que le programme n'ait plus à s'occuper que du mouvement proprement dit.

Celui-ci est produit par une rotation vers le bas de la palette (partielle) des couleurs de 0 à 3. La couleur 0 est remplacée par la couleur 1, la couleur 1 par la couleur 2, la couleur 2 par la couleur 3, la couleur 3 par la couleur 0, ce qui correspond à une rotation. Comme on le voit dans le schéma, les zones rouges et bleues se déplacent d'une ligne vers le haut, ce qui était bien le but de l'opération.

Ce défilement n'a pas grand sens en lui-même, car il peut aussi être obtenu par modification de l'adresse de début de l'écran (Linear Starting Address). Le tout ne devient vraiment intéressant que lorsque la structure des cases de notre exemple est munie d'effets. Supposons qu'on construise un échiquier avec ces cases et qu'on l'incline vers l'arrière avec un bon logiciel de dessin. La rotation de palette présente une surface inclinée qui avance ou recule. La programmation de cet effet avec des manipulations de pixels conventionnelles conduira probablement à l'échec en raison du temps de calcul nécessaire. Il vaut mieux laisser les calculs au logiciel de dessin et se contenter, dans son propre programme, de déplacer quelques octets en mémoire et de réaliser la palette.

Un autre avantage de la méthode consiste en la possibilité de placer des objets quelconques devant la surface en déplacement. Normalement, ces objets doivent être recopiés à chaque génération d'écran à l'aide d'une routine de sprite (relativement lente). Ici, ils sont intégrés à l'image.

Pour réaliser une telle image, il est impératif d'utiliser un logiciel de dessin basé sur la palette. Malheureusement, la plupart des logiciels Windows qui sont apparus récemment travaillent en TrueColor. Dans ce cas, il n'est pas possible de définir la couleur de palette que prendra ultérieurement un pixel. Pour préparer des gradients de couleurs contiguës, il faut pouvoir contrôler directement la palette.

L'image qui va s'animer à l'écran grâce à la rotation de palette

L'image d'exemple ECHECS.GIF a été créée sous Deluxe Paint II Enhanced, un logiciel qui maîtrise la perspective et autorise des manipulations directes de palette. On a d'abord dessiné un échiquier noir et blanc. Les cases blanches ont été remplies avec les couleurs 16 à 31 (gradient rouge à bleu) et les cases noires avec les couleurs 32 à 47. Cet échiquier peut alors être manipulé librement, la forme graphique n'ayant pas d'influence sur la rotation de palette. Dans notre exemple, nous l'avons incliné vers l'arrière, de sorte qu'il représente un plan en perspective.

Au-dessus de l'échiquier se trouve un logo qui exploite la partie supérieure libre de la palette. A droite au-dessus de l'échiquier, vous pouvez apercevoir une autre application de la rotation de palette, souvent utilisée dans les jeux d'aventure : la fontaine. Elle fait appel aux couleurs 48 à 63 qui contiennent un gradient de couleurs allant du noir au bleu. Par rapport au fond noir, l'ensemble donne une impression d'eau jaillissante.

Comme dernier effet, l'image contient un "radar" rouge. On a généré un gradient concentrique avec les couleurs 64 à 88. Il est constitué de 25 cercles de même centre, qui ont des couleurs consécutives dans la palette. On prend du rouge pour les couleurs 64 à 80, le reste étant noir. Pendant la rotation, les deux anneaux rouges se déplacent par-dessus les anneaux de couleur 64-80 et produisent ainsi des cercles concentriques fuyants.

Pour permettre le contrôle des effets dessinés, Deluxe Paint possède une fonction spéciale qui exécute (très lentement) la rotation. Mais comment obtenir cet effet dans son propre programme ? C'est dans ce but qu'a été écrite la procédure pal_rot (dans MODEXLIB.ASM) qui attend comme argument la zone de palette à soumettre à rotation. Normalement, la zone décrite est décalée d'une position vers le bas, vers les couleurs de numéros inférieurs, la

couleur la plus basse réapparaissant tout en haut. Mais si la première limite transmise est supérieure à la deuxième, la rotation se fait vers le haut.

```
Pal_Rot proc pascal far Debut,Fin:Word
;effectue une rotation sur une partie de la palette de Debut à Fin
 avec décalage de 1
;si Debut < Fin : Rotation vers le bas
;si Debut > Fin : Rotation vers le haut

  mov ax,des        ;es doit pointer sur le segment de données
  mov es,ax

  lea si,palette    ;charge l'offset de la palette
  mov di,si         ;en si et di

  mov ax,3          ;convertit "Debut" en offset de palette
  mul Debut
  add si,ax         ;ajoute à si
  mov ax,3          ;la même chose pour la fin
  mul Fin
  add di,ax         ;ajoute à di

  mov bx,[si]       ;sauvegarde les octets de la couleur du début
  mov dl,[si+2]

  mov cx,di         ;la différence entre Debut et Fin
  sub cx,si         ;donne le nombre d'octets à copier

  mov di,si         ;couleur de début comme offset de destination
  add si,3          ;une couleur plus loin comme offset source
                    ;prêt pour la copie
  cld               ;a priori copie vers l'avant
  or cx,cx          ;si cx négatif (Debut > Fin)
  jns enavant
  std               ;alors copie vers l'arrière
  neg cx            ;on corrige cx
  sub si,4          ;si sur le 2ème octet de l'avant dernière couleur
  add di,2          ;di sur le 2ème octet de la dernière couleur
  add cx,2          ;copie 2 octets supplémentaires
```

```
enavant:            ;pour que ,après la boucle de copie, la position
                    ;soit correcte

  rep movsb         ;copie les couleurs
  mov [di],bx       ;les octets de l'ancienne couleur de Debut
  mov [di+2],dl     ;deviennent la dernière couleur

  cld               ;remet à 0 l'indicateur de direction
  ret
Pal_Rot Endp
```

Après conversion des couleurs transmises en offsets d'octets, la couleur de départ est sauvegardée car les autres couleurs se déplacent vers elles et vont l'écraser.

Le nombre d'octets à décaler se déduit simplement de la différence entre les deux offsets. Pour préparer le décalage proprement dit, on fait pointer di sur l'offset du début, car il constitue la destination. En supposant que le décalage se fasse vers l'avant, on fait pointer si sur la couleur suivante, de sorte qu'à la première itération ce sera la couleur du début qui sera copiée.

Si la valeur du début est supérieure à la valeur de la fin (nombre d'octets à copier négatif), certains registres doivent être ajustés. L'indicateur de direction est mis à 1 car, pour effectuer un décalage de bas en haut lorsque les zones source et destination se recouvrent, il faut faire la copie en arrière, sinon les valeurs copiées écrasent celles qui ne le sont pas encore et on ne fait qu'implémenter un algorithme inefficace de remplissage de mémoire. Comme cx est négatif, il faut inverser son signe. Puis si et di doivent être ajustés car si pointe sur une valeur de couleur qui suit immédiatement la zone de la palette (le pointeur a été déplacé à cet endroit par add si, 3 ce qui est exact pour la copie en avant). Il faut donc placer si sur le dernier octet de l'avant-dernière couleur (le dernier octet, car movsb copie en arrière !). di doit également pointer sur le dernier (deuxième) octet de la dernière couleur pour remplir cette couleur et celles qui se trouvent en dessous de haut en bas.

Du fait de ce positionnement, di ne renvoie pas à la dernière couleur à l'issue de la copie. On pourrait corriger ceci en insérant un test après la boucle et rectifier di en conséquence, mais il est plus simple de copier deux octets de plus qui, après la boucle de copie, sont remplacés par les octets sauvegardés de la couleur de début. La boucle et le traitement de la couleur du début sont identiques quelle que soit la direction de copie, c'est pourquoi la section est introduite par l'étiquette enavant. Pour finir, l'indicateur de direction est remis à 0, pour ne pas susciter de chaos dans d'autres procédures qui reposent sur cette valeur.

Associé à l'image ECHECS.GIF, vous trouverez sur le CD d'accompagnement le programme d'exemple appelé PALROT.PAS :

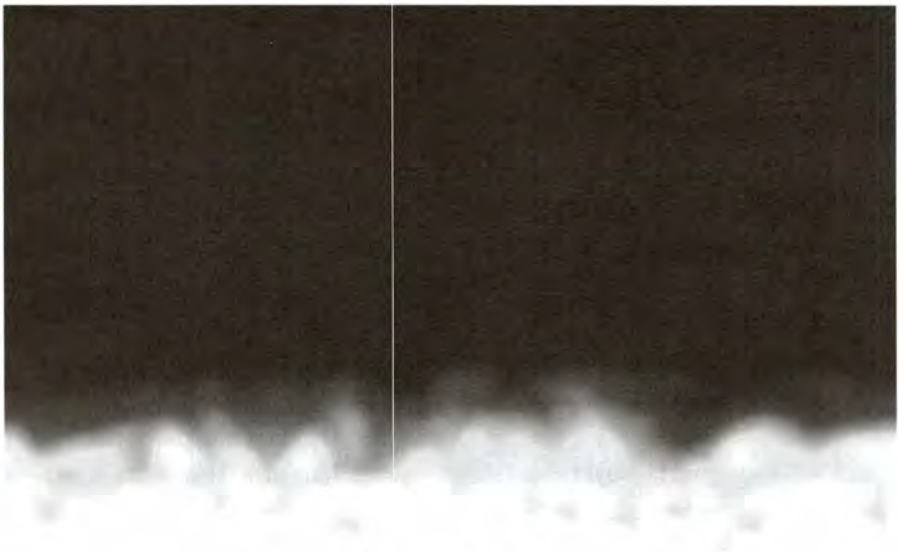
```
Uses Crt,ModeXLib,Gif;

Var slow_flag:Boolean;          {pour ralentir les processus}

Begin
  Init_Mode13;                  {active le mode 13h }
  LoadGif('echecs');            {charge et affiche l'image }
  Show_Pic13;
  Repeat
    Pal_Rot(16,47);             {déplace l'échiquier }
    If slow_flag Then Begin     {toutes les 2 itérations }
      Pal_Rot(63,48);           {anime la fontaine }
      Pal_Rot(88,64);           { et le radar }
    End;
    slow_flag:=not slow_flag;   {permet de n'animer la fontaine et le
                                 radar qu'une fois sur deux }
    WaitRetrace;                {synchronisation}
    SetPal;                     {réalise la palette après rotation }
  Until KeyPressed;             {jusqu'à frappe de touche }
TextMode(3);
End.
```

L'image est chargée dans la mémoire d'écran en mode 13h. Une boucle effectue la rotation des zones de palette qui gèrent les animations. Vous pouvez faire quelques expériences en modifiant les directions communiquées à Pal_Rot.

Tandis que l'échiquier doit défiler à pleine vitesse, le rythme est excessif pour la fontaine et le radar. C'est pourquoi on ne fait tourner les parties de palette associées à ces objets qu'une fois sur deux. Ce ralentissement est piloté par la variable booléenne slow_flag qui prend alternativement les valeurs TRUE et FALSE.

6.11. MONITEUR EN FLAMME : EFFET DE FEU

Après les jeux de rotation de palette, fabriquer un feu qui dévore l'écran n'a rien d'extraordinaire. Les flammes peuvent naître d'images prédessinées qui sont affichées l'une après l'autre. Mais nous allons présenter une autre possibilité employée dans de nombreuses démos en simulant directement la structure des flammes.



L'écran en feu !!!

Il faut faire deux observations à ce sujet. Un feu se trouve toujours en mouvement ascendant. Par ailleurs, il est blanc dans sa partie inférieure et devient de plus en plus rouge vers le haut.

Le programme FLAMES.PAS montre comment procéder.

Il exploite deux buffers qui reflètent la mémoire d'écran (ou sa partie inférieure dans les systèmes rapides).

```
Uses Crt,ModeXLib;
Type Block=Array[0..99,0..319] of Byte;
Var
  Src_Frame, {image précédente }
  Dest_Frame:^Block;            {image courante }
```

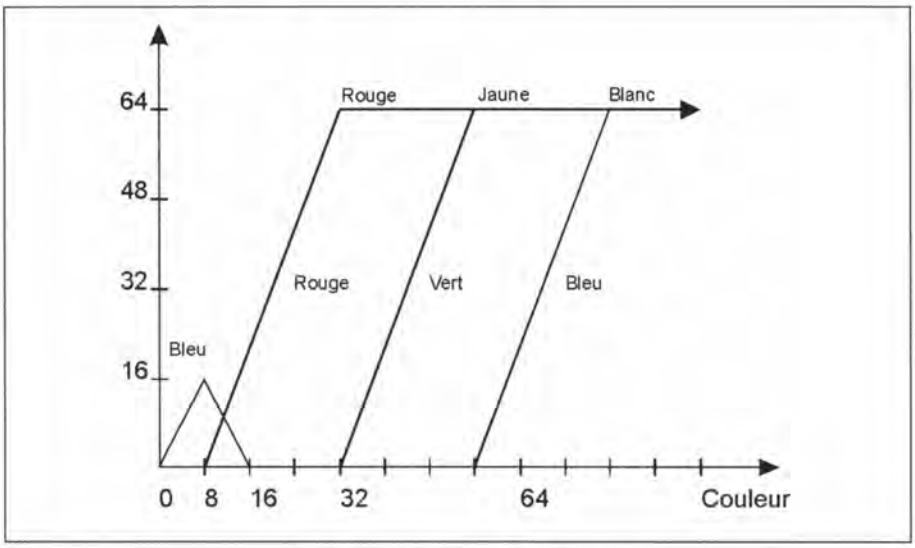
Les procédures Show_Screen et Prep_Pal ainsi que le programme principal sont faciles à comprendre :

```
Procedure Show_Screen;          {copie l'écran dans la carte graphique}
Var temp:Pointer;               {pour permuter les pointeurs }
Begin
asm
  push des
  lds si,Dest_Frame             {image finie comme source }
  mov ax,0a000h                 {VGA comme destination }
  mov es,ax
  mov di,320*100                {à partir de la ligne 100}
  mov cx,320*100/4              {copie 100 lignes comme Dwords}
db 66h                          {Operand Size Prefix (32 bits)}
  rep movsw                     {copie}
  pop des
End;
  temp:=Dest_Frame;             {échange les pointeurs sur source et
                                 destination }
  Dest_Frame:=Src_Frame;
  Src_Frame:=temp;
End;

Procedure Prep_Pal;             {prépare la palette pour Flames }
Var i:Word;
Begin
  FillChar(Palette,80*3,0);     {au début: tout noir }
  For i:=0 to 7 do Begin
    Palette[i*3+2]:=i*2;        {couleurs 0-7: bleu croissant }
    Palette[(i+8)*3+2]:=16-i*2;{couleurs 8-15: bleu décroissant }
  End;
  For i:=8 to 31 do             {couleurs 8 -31: rouge croissant }
    Palette[i*3]:=(i-8)*63 div 23;
  For i:=32 to 55 do Begin      {couleurs 32-55: vert croissant,
                                 rouge constant }
    Palette[i*3]:=63;
    Palette[i*3+1]:=(i-32)*63 div 23;
  End;
```

```
  For i:=56 to 79 do Begin      {couleurs 56-79: bleu croissant,
                                 rouge et vert const.}
    Palette[i*3]:=63;
    Palette[i*3+1]:=63;
    Palette[i*3+2]:=(i-56)*63 div 23;
  End;
  FillChar(Palette[80*3],176*3,63);  {le reste blanc }
  SetPal;                       {réalise la palette }
End;

begin
  Randomize;                    {initialise le générateur de nombres
                                 aléatoires }
  GetMem(Src_Frame,320*100);    {réserve de la mémoire pour l'image
                                 source et la met à 0 }
  FillChar(Src_Frame^,320*100,0);
  GetMem(Dest_Frame,320*100);   {réserve de la mémoire pour l'image
                                 de destination et la met à 0 }
  FillChar(Dest_Frame^,320*100,0);
  Init_Mode13;                  {initialise le mode 13h }
  Prep_Pal;                     {prépare la palette }
  Repeat
    Scroll_Up;                  {flammes montantes }
    New_Line;                   {ajoute une nouvelle ligne en bas }
    Show_Screen;                {affiche l'écran }
  Until KeyPressed;
  TextMode(3);
end.
```

Le programme principal alloue et initialise d'abord les deux buffers. Une fois le mode 13h activé, la palette est remplie pour l'effet de flammes.

Le résultat est représenté par le schéma suivant :



Palette de couleurs pour les flammes

Les pixels dont le numéro de couleur est faible correspondront à des zones froides, ils se trouvent ici à gauche. Si on lit le graphique de droite à gauche, c'est-à-dire dans le sens des températures décroissantes, on observe une succession de couleurs ainsi définies.

D'abord, toutes les proportions de composants sont maximales (=blanc). A partir de la couleur 80, la proportion de bleu est réduite progressivement, de sorte que les couleurs apparaissent plus jaunâtres. A partir de la couleur 56, on abaisse la proportion de vert, ce qui laisse subsister en fin de compte le rouge. Quand cette dernière composante ne participe plus à la couleur, il ne reste théoriquement que du noir. Mais nous avons introduit encore un soupçon de bleu (couleurs 16-0) qui se situe au-dessus des flammes rouges. Si vous n'aimez pas, retirez simplement la première boucle de la procédure.

La boucle principale se compose de trois ordres : Scroll_Up, New_Line et Show_Screen. Après composition de l'écran dans le buffer Dest_Frame, il est affiché par Show_Screen. Une simple boucle qui copie des doubles mots (dwords) suffit à cette tâche. Ensuite, les pointeurs sur les deux buffers sont permutés pour que l'écran terminé serve de source à l'itération suivante.

Les procédures centrales sont Scroll_Up et New_Line

```
Procedure Scroll_Up;assembler;
{fait défiler l'image d'une ligne vers le haut et interpole }
asm
  push des
  les di,Dest_Frame          {pointeur sur image de destination }
  lds si,Src_Frame           {pointeur sur image source }
  add si,320                 {ligne 1 de l'image source }
  mov cx,320*98              {fait défiler 99 lignes }
  xor bl,bl                  {octet de poids fort fantôme }
@lp1:
  xor ax,ax
  xor bx,bx
  mov al,[si-321]            {cherche le premier point }
  mov bl,[si-320]            {ajoute le deuxième }
  add ax,bx
  mov bl,[si-319]            {ajoute le suivant }
  add ax,bx
  mov bl,[si-1]              {etc...}
  add ax,bx
  mov bl,[si+1]
  add ax,bx
  mov bl,[si+319]
  add ax,bx
  mov bl,[si+320]
  adc ax,bx
  mov bl,[si+321]
  adc ax,bx
  shr ax,3

  or ax,ax                   {déjà 0 ?}
  je @null
  dec al                     {si non, décrémenter}
@null:
  stosb                      {valeur en destination }
  inc si                     {point suivant }
  dec cx                     {autres points ?}
  jne @lp1
```

```
  pop des
End;

Procedure New_Line;              {reconstruit les lignes inférieures }
Var i,x:Word;
Begin
  For x:=0 to 319 do Begin       {remplit 3 lignes avec des valeurs
                                  aléatoires }
    Dest_Frame^[97,x]:=Random(15)+64;
    Dest_Frame^[98,x]:=Random(15)+64;
    Dest_Frame^[99,x]:=Random(15)+64;
  End;
  For i:=0 to Random(45) do Begin {ajoute un nombre aléatoires de
                                   foyers brûlants }
    x:=Random(320);              {à des endroits au hasard }
    asm
      les di,Dest_Frame          {adresse l'image de destination }
      add di,98*320              {traite la ligne 98 (deuxième du bas) }
      add di,x                   {ajoute l'abscisse x }
      mov al,0ffh                {couleur la plus claire }
      mov es:[di-321],al         {produit un foyer brûlant de grande
                                  taille (9 pixels)}
      mov es:[di-320],al
      mov es:[di-319],al
      mov es:[di-1],al
      mov es:[di],al
      mov es:[di+1],al
      mov es:[di+319],al
      mov es:[di+320],al
      mov es:[di+321],al
    End;
  End;
End;
```

Scroll_Up déplace la mer de flammes d'une ordonnée vers le haut. Une interpolation simultanée permet de calculer chaque nouveau point à partir de la moyenne de ses voisins immédiats, ce qui donne un aspect un peu flou et délavé aux flammes.

La procédure recherche dans la boucle @lp1 les 8 pixels voisins de chaque pixel source, les additionne et divise la somme par huit. Cette moyenne est diminuée de 1 pour que les flammes ne montent pas trop, puis écrite dans le buffer.

New_Line a pour mission d'insérer de nouvelles lignes à l'extrémité inférieure de l'écran (la braise, en quelque sorte). La première boucle remplit les trois lignes du bas avec des valeurs aléatoires comprises entre 64 et 79. La deuxième boucle introduit les foyers brûlants, qui ont une température très élevée et d'où jaillissent les flammes. Ils sont faits de blocs de 9 pixels auxquels on donne la valeur extrême 255. Lors du défilement vers le haut, ils ne se refroidissent que lentement et apparaissent comme des zones lumineuses en ascension.

6.12. Le secret de Comanche (Tm) - Voxel spacing

Apprécié dans de nombreuses démos et largement intégré dans quelques jeux, le voxel spacing est un paysage imaginaire avec des montagnes et des vallées que survole l'observateur.

Autant cette description est générale, autant il existe de méthodes pour produire cet effet. Le nombre des algorithmes doit être au moins aussi élevé que le nombre de programmeurs qui s'y sont essayés. Nous allons ici montrer une méthode simple qui se passe de représentation tridimensionnelle et se limite à une résolution à deux dimensions.

Le principe est d'incliner une carte géographique de façon que l'observateur puisse l'examiner en perspective. Les altitudes peuvent être représentées par des lignes verticales de différente longueur.

La carte où sont dessinés les hauts et les bas du paysage sera constituée par une image 320x200 avec des transitions douces. La couleur 0 correspondra au point le plus bas, et la couleur 255 au point le plus élevé.

La projection en deux dimensions s'effectue selon un principe relativement simple. On sait que les objets éloignés apparaissent plus petits que les objets proches. Si on regarde un paysage à travers un cadre - notre écran - on constate que les objets lointains qui rentrent dans ce cadre sont plus nombreux que les objets proches, parce qu'ils paraissent plus petits. On voit donc toujours un extrait semblable à celui indiqué par le schéma suivant :

Le cadre rectangulaire représente l'ensemble du paysage existant, le trapèze sa partie visible. On cherche à dessiner le trapèze sur un écran rectangulaire. La partie antérieure est donc étirée horizontalement, de façon que les objets y soient agrandis. Les coordonnées indiquées se rapportent à une projection sur la moitié d'écran inférieure du mode X.

La projection trapézoïdale est obtenue en réduisant de façon continue le nombre de pixels qui entrent dans une ligne. En d'autres termes : le rapport entre la taille d'un élément du paysage et celle d'un pixel de l'écran est constamment décrémenté. Au début, l'échelle est de 1:1, on range 80 pixels dans une ligne (résolution choisie à cause de la vitesse). Pendant le dessin, le rapport est réduit jusqu'à une échelle 1:2, de sorte que 40 pixels seulement trouvent place dans la ligne la plus avancée.

Comme la représentation se fait ligne par ligne, il faut veiller à ce que la distance entre deux lignes successives se réduise en arrière du plan. Le dessin se faisant du haut vers le bas, la distance entre deux lignes devra être augmentée de façon continue.

Il manque encore la prise en compte de l'altitude. On utilise à cet effet la couleur de chaque point qui donne le nombre de pixels utilisés verticalement. Cette structure verticale évite par ailleurs les lacunes qui se produisent en raison de la distance croissante entre les lignes vers l'avant.

Pour rendre la scène encore plus réaliste, il manque des nappes d'eau. On décide ainsi de donner à chaque couleur inférieure à un seuil la valeur de ce seuil. Il en résulte des zones dénuées de structure verticale.

Pour générer les paysages, le plus simple est de se servir d'un générateur de fractales capable de dessiner des nuages de plasma.

Le programme VOXEL.PAS dessine la scène dans une boucle, en fonction du déplacement de la souris (pilotage par souris).

```
Uses Crt,Gif,ModeXLib;
Var
    x,y:Word;                    {coordonnées du trapèze }

Procedure Draw_Voxel;external;
{$l voxel.obj}

Begin
  Init_ModeX;                    {active le mode X }
  LoadGif('landsc3');            {charge le paysage }
  x:=195;                        {fixe les coordonnées de début }
  y:=130;
  Repeat
    ClrX($0f);                   {efface l'écran }
    Draw_Voxel;                  {représente le paysage }
    Switch;                      {active la page d'écran terminée }
    WaitRetrace;                 {attend le retour de balayage
                                  vertical }
    asm
      mov ax,000bh               {fonction 0bh: lit les coordonnées
                                  relatives }
      int 33h
      sar cx,2                   {division par 2}
      sar dx,2
      add x,cx
      add y,dx
    End;
    If x < 0 Then x:=0; If x > 130 Then x:=130;
    If y < 0 Then y:=0; If y > 130 Then y:=130;
  Until KeyPresed;               {sort du programme dès qu'une touche
                                  est activée }
  TextMode(3);
End.
```

La procédure centrale est Draw_Voxel qui se trouve dans le module en assembleur VOXEL.ASM :

```
 data segment
  extrn vscreen:dword          ;pointe sur les données du paysage
  extrn x,y: word              ;coordonnées du trapèze
  extrn vpage:word             ;page d'écran courante
data ends

code segment
assume cs:code,des:data

;Variables avec partie décimale (8 bits inférieurs):

offst dd 0                     ;offset courant
step dd 0                      ;taille du pixel
row_start dd 0                 ;début de la ligne courante
row_step dd 0                  ;distance jusqu'à la ligne suivante

z_count dw 0                   ;compteur pour la profondeur
shrink dw 0                    ;correction au bas de l'écran

Ligne dd 0                     ;numéro de ligne d'écran courante
vpage_cs dw 0                  ;page d'écran dans le segment de code

.386
public Draw_Voxel
Draw_Voxel proc pascal
;représente un paysage dans la page d'écran courante
;lit les données dans vscreen à partir de la position (x,y)
  mov ax,vpage     ;mémorise le numéro de la page d'écran
  mov vpage_cs,ax

  push des
  mov ax,0a000h                ;charge le segment de destination
  mov es,ax

  mov ax,320                   ;calcule l'offset dans le paysage
  imul y
```

```
add ax,x

lds si,vscreen              ;prend les données dans vscreen
add si,ax                   ;tient compte de l'offset
shl esi,8                   ;conversion en virgule fixe

mov offst,esi               ;valeurs initiales pour le pixel ...
mov row_start,esi           ;... et la ligne

mov step,100h               ;facteur d'échelle initial 1
mov ligne,100*256           ;commence en ligne d'écran 100
mov row_step,14040h         ;distance interligne 320,25
mov shrink,0                ;pas de correction au début
mov z_count,160             ;nombre de lignes à traiter
```

La première partie initialise quelques variables importantes. Mentionnons notamment Offst et row_start qui sont en format virgule fixe : la partie entière est dans les bits 8-31, la partie décimale dans les bits 0-7. Le premier de ces nombres représente l'offset du pixel courant et le second l'offset de la ligne courante. Après chaque point, Offst est augmenté de Step pour progresser dans le paysage. Si Step est égal à 080h (=0,5), on lit un nouveau pixel du paysage toutes les deux itérations.

Le fonctionnement est le même pour row_start et row_step. Ainsi, row_step contient 14040 ce qui en notation décimale correspond à la valeur 320,25. Après chaque ligne, row_start indique le début de la ligne suivante, et on tient compte du rétrécissement vers l'avant en faisant débuter la ligne suivante un peu plus à droite.

```
next_y:
 mov eax,Ligne              ;lit le numéro de ligne courant (écran)
 mov ebx,eax                ;le sauvegarde
 shr eax,8                  ;le convertit en entier
 add eax,50                 ;50 pixels vers le bas
 imul eax,80                ;convertit en offset
 mov di,ax                  ;sauvegarde comme pointeur de
                            ;destination
 cmp di,199*80              ;a-t-on dépassé le bord de l'écran?
 jb normal
 mov di,199*80              ;oui, on se positionne en dernière
                            ;ligne
```

```
  mov eax,Ligne              ;différence par rapport au bord
                             ;inférieur de l'écran
  shr eax,8
  sub eax,149
  mov shrink,ax              ;mémorisée comme correction

normal:
  add di,vpage_cs            ;ajoute la page d'écran courante

  imul ebx,16500             ;multiplie le numéro de ligne par
                             ;1.007
  shr ebx,14                 ;le tout * 16500 / 16384
  mov Ligne,ebx              ;on sauvegarde
```

On calcule ici l'offset de chaque ligne. La variable Ligne contient le numéro de la ligne courante qui est à convertir en offset. Il ne faut pas confondre ce numéro de ligne avec row_Offst qui s'occupe de la position dans le paysage, alors que Ligne s'occupe de la position à l'écran. Si l'offset calculé est en dessous de l'écran, il faut quand même procéder à un dessin car les verticales de ce point peuvent quand même apparaître sur l'écran. Pour traiter ce cas, on remet le pointeur de destination di sur l'écran et la hauteur de la barre est diminuée (du nombre indiqué en shrink).

Il faut ensuite tenir compte de la distance croissante entre les lignes vers l'avant de la perspective. L'ordonnée y courante de l'écran se trouve dans la variable Ligne qui est multipliée par le facteur 1,007, de sorte que les lignes s'écartent bien vers l'avant.

```
mov bp,80                    ;nombre de pixels par ligne
next_x:
  mov esi,offst              ;charge l'offset du pixel courant
  shr esi,8                  ;le convertit en entier
  xor eax,eax
  mov al,[si]                ;charge un point du paysage
  mov cx,ax                  ;le sauvegarde

  cmp cx,99                  ;couleur (=hauteur) < 100
  ja fill_bar
  mov ax,99                  ;alors on fixe à 99

fill_bar:
```

```
  shl ax,5                    ;projection à point de fuite :
                              ;hauteur * 32
  xor dx,dx
  push bp
  mov bp,z_count              ;divise par la distance
  add bp,50
  idiv bp
  pop bp

  sub ax,shrink               ;effectue la correction
  jbe suite                   ;si <= 0, ne dessine pas
```

Pour chaque point, on lit d'abord la couleur à l'offset courant. Le niveau de l'eau est fixé au seuil de 99. A l'aide du compteur z_count qui donne la distance de chaque ligne à l'observateur, on calcule la perspective à point de fuite. Cette méthode permet de représenter les objets 3D en 2D et sera présentée au chapitre 8.2. La hauteur (sauvée en al) est diminuée de la correction shrink pour qu'on puisse travailler tout en bas de l'écran.

```
push di
next_fill:
  mov es:[di],cl              ;enregistre la couleur
  sub di,80                   ;ligne suivante vers le haut
  dec al                      ;décrémente le compteur
  jne next_fill               ;on continue ?
  pop di

suite:
  inc di                      ;adresse l'octet suivant à l'écran
  mov esi,step                ;lit le pas
  add esi,offst               ;l'ajoute
  mov offst,esi               ;et réenregistre

  dec bp                      ;point suivant
  jne next_x

  mov esi,row_step            ;déplace le début de la ligne
  add esi,row_start
  mov row_start,esi
  mov offst,esi               ;recharge également l'offset du pixel
```

```
    dec step                    ;décrémente le facteur d'échelle
    dec z_count                 ;et le compteur de ligne
    jne next_y
    pop des
    ret
Draw_Voxel endp

code ends
end
```

La boucle next_Fill trace la barre verticale de la taille calculée. On détermine ensuite l'offset du point suivant et, si la ligne est nouvelle, l'offset de cette dernière. Par ailleurs, le pas step est décrémenté à chaque ligne pour obtenir l'effet d'élargissement vers l'avant.

7. Les sprites : de l'action fulgurante sur l'écran

Vaisseaux spatiaux dans une bataille galactique ou figures de jeu dans Jump'n'Run, les sprites sont partout. Tout ce qui sur l'écran se meut différemment du fond peut être qualifié de sprite. Chacun sait ce qu'est un sprite. Mais la question est de savoir les programmer.

7.1. Les principes

Avant de créer des sprites soi-même, il faut d'abord réfléchir à leur structure interne. En fait, les sprites ne sont rien d'autre que de petites sections graphiques (ou grandes, selon la puissance de calcul disponible) qui peuvent être placées librement sur une image de fond.

Ces graphiques doivent également remplir une autre condition : ils doivent présenter des zones transparentes qui laissent apparaître le fond de l'écran. Il ne s'agit pas seulement des "trous" au milieu du sprite mais également des bords. Tout sprite qui n'est pas complètement rectangulaire doit posséder des zones transparentes sur ses bords. Au moment de l'affichage, on ne peut pas faire appel à des formes de type circulaire, pour des raisons de vitesse de calcul. On est obligé de se restreindre à des zones rectangulaires qui doivent notamment être capables de gérer la transparence, sans quoi tous les sprites seraient des rectangles pleins.

Alors que les ordinateurs domestiques s'occupent pratiquement de tout à la place du programmeur, ne lui laissant que le soin de fixer des positions, la tâche est plus ardue avec un PC. L'ensemble du processus se déroule dans la mémoire d'écran de la carte VGA, il faut donc optimiser soigneusement son programme. Mais la solution logicielle nécessitée par le PC présente des avantages inestimables : la création est entre les mains du programmeur, les sprites peuvent avoir n'importe quelle taille même si la vitesse de calcul nous

ramène assez vite à la raison. Il est également possible de faire des changements d'échelle et des rotations. Il est vraiment possible de se défouler dans ce domaine, à condition de s'y connaître.

Les étapes d'affichage d'un ensemble de sprites sont les suivantes :

1. Effacer les sprites en copiant l'image de fond dans la page d'écran courante
2. Déplacer et dessiner les sprites dans leur nouvelle position
3. Activer la nouvelle page

Avant de pouvoir représenter les sprites dans leur nouvelle position, il faut d'abord faire disparaître les anciens. Il paraît a priori exagéré de recopier une page d'écran entière. Ne pourrait-on pas, avant de dessiner un sprite, sauvegarder le fond qu'il recouvre et le rétablir par la suite ? Cette manière de procéder se justifie pour un ou deux sprites mais au-delà la gestion nécessaire est trop importante.

D'abord, il ne saurait être question que d'une sauvegarde du fond en mémoire VGA, car c'est là que l'accès est rapide. Mais cette mémoire est très limitée, tôt ou tard on manquera de place. Ensuite, le nombre d'instructions nécessaires dépasse celui d'une copie d'écran. Il s'agit en effet de zones au milieu de l'écran qui ne peuvent être transférées que ligne par ligne, un simple rep movsb sur l'ensemble de la zone n'est pas possible.

Nous allons donc recopier à chaque fois une page de fond d'écran complète qui sera naturellement placée dans la mémoire vive de la carte VGA, par exemple en page 3. De là, nous effectuerons un transfert d'un trait sur la page actuelle par une boucle de copie ultra rapide.

Le déplacement des sprites est complètement libre, on peut leur faire parcourir des droites ou des cercles, les promener le long de sinusoïdes ou même les faire réagir à la musique : toute liberté est laissée au programmeur.

Avant de décrire la représentation des sprites, il nous faut revenir sur la notion de double page d'écran. Comme nous l'avons déjà mentionné, il est presque impossible d'effectuer des manipulations d'image de quelque importance pendant un retour de balayage vertical. C'est pourtant la seule manière d'éviter le scintillement et la déformation des sprites. Par ailleurs, dans la démarche expliquée plus haut, il y a un moment (entre les étapes 2 et 3) où aucun sprite ne se trouve sur l'écran, ce qui se traduit par un scintillement gênant.

Il faut donc faire appel à une deuxième page d'écran pour préparer en cachette le nouveau dessin pendant que la page visible demeure inchangée. Ce n'est

qu'à l'issue des modifications que la nouvelle page d'écran est activée, la page escamotée devenant la nouvelle page de travail.

Revenons à la représentation proprement dite. Deux principes peuvent être appliqués. On pourrait laisser les données des sprites dans la mémoire graphique, par exemple en page 2 et de là les recopier systématiquement dans la page courante. Il en résulte deux inconvénients. Le mode d'écriture 1 qui nous intéresse par sa rapidité repose sur des ensembles de quatre octets (un octet issu de chacun des quatre plans) de sorte qu'il n'est pas possible de copier par exemple un sprite d'abscisse 5 (plan 1) en 6 (plan 2). Une copie rapide ne peut se faire que dans un plan donné. La seule possibilité de contourner ce problème est de maintenir quatre fois en mémoire chaque sprite : une fois en abscisse 0, une fois en abscisse 1, etc ce qui limite le nombre de sprites disponibles. Le deuxième inconvénient de ce principe de résolution tient au masquage des points transparents qui nécessite des masques en mémoire centrale, qu'il faut d'abord créer.

7.2. Lire et écrire des sprites

Nous allons donc dans cet ouvrage travailler sur un autre principe de résolution. Les données des sprites restent en mémoire centrale et sont amenées point par point sur l'écran. L'inconvénient de cette démarche est sa lenteur, il faudra donc sérieusement réfléchir à son optimisation.

Cette préoccupation commence déjà au stade de la définition du format des données des sprites en mémoire centrale. Lors de la création d'un sprite, les données graphiques sont réparties sur les quatre plans : les points dont l'adresse est un multiple de 4 vont dans le plan 0, ceux qui donnent un reste de 1 passent dans le plan 1 et ainsi de suite. La façon la plus simple d'amener les données à l'écran est de les disposer dans leur ordre naturel et de changer sans cesse de plan d'écriture.

Mais ce n'est pas la solution la plus rapide, comme nous l'avons vu dans le cadre de la procédure p13_2_ModeX. Cette procédure suit une autre démarche également valable pour les sprites. Elle copie d'abord toutes les données d'un plan, puis change de plan et copie les données suivantes.

Le format des sprites tient compte de cette nécessité. Les données ne se trouvent pas dans le format du mode 13h (succession linéaire de pixels) mais sont rangées en harmonie avec le mode X : d'abord les données du premier plan, puis celles du second, etc. Lorsqu'on écrit les sprites, on peut accéder facilement à ces données. Il suffit de charger l'adresse de début dans le registre

si, puis on copie le premier plan et les autres immédiatement à la suite. Il n'est pas nécessaire de réinitialiser le registre si.

Les données elles-mêmes contiennent directement leur couleur, elles peuvent donc être transférées sans précaution dans la mémoire d'écran. Simplement, la couleur 0 jouera un rôle particulier : elle signifie qu'à cet endroit on voit le fond, le sprite étant transparent.

Le calcul de la largeur du sprite réserve encore un petit problème car il n'est pas le même dans tous les plans. Supposons par exemple qu'un sprite d'une largeur de 5 pixels soit à écrire en abscisse 0. Dans ce cas, on copie deux octets dans le plan 0 (pixels 0 et 4), tous les autres plans ne recevant qu'un seul octet.

Mais dans les algorithmes décrits dans le présent ouvrage, la répartition en question est globale, elle ne dépend pas de l'abscisse de la position de destination. Si on copie le sprite de l'exemple précédent à l'abscisse 1, le bloc de 2 octets se trouve cette fois dans le plan physique 1. Mais comme la boucle de copie commence à cet endroit, du point de vue des données du sprite, il s'agit toujours du plan 0 avec ses deux octets. En d'autres termes : à partir de la seule largeur du sprite, on peut créer un tableau géré par le compteur de la boucle de copie et qui contient les largeurs occupées dans chaque plan. Dans notre exemple, le premier plan des données du sprite a toujours deux octets, les autres un seul octet, indépendamment de la position courante.

Le véritable plan de départ n'est à établir que pour les soins de la transmission à la carte VGA.

7.3. PAR-DELÀ LES FRONTIÈRES : LE CLIPPING

En plus de la prise en compte des 0, le dessin des sprites pose un autre problème de vitesse : le clipping. Quel que soit le programme, il n'est guère possible d'éviter qu'un sprite ne puisse toucher le bord de l'écran et disparaître. Si avec la stratégie étudiée on se contente de recopier aveuglément les données dans la mémoire d'écran, on obtient des résultats très confus. Lorsqu'on dépasse le bord droit ou le bas de l'écran, le reste du sprite est copié dans la ligne suivante ou la page d'écran suivante. Aux bords supérieur ou gauche, la ligne ou la page précédente reçoivent des données. Aucun de ces effets n'est désirable, il faut donc se mettre à la recherche d'un algorithme qui gère le clipping, c'est-à-dire le découpage des données qui dépassent (écrêtement).

Le plus simple est de tester la situation de chaque point avant de le dessiner : mais cette solution est trop lente. Il est plus habile de modifier les conditions

d'encadrement avant la boucle de copie proprement dite de façon que la représentation se fasse de la manière habituelle. On essaye donc de maintenir le plus possible le clipping à l'extérieur de la boucle. Vous savez en effet que plus une instruction se trouve à l'extérieur d'un ensemble de boucles imbriquées, moins elle est utilisée et moins elle consomme de temps.

Ainsi compris, le principe du clipping est simple. La largeur ou la hauteur du sprite sont ajustées avant affichage. Pour les bords droit et inférieur, il faut introduire une variable qui permet de sauter par-dessus les données source correspondant aux zones non dessinées. Si une ligne de sprite n'est copiée qu'à moitié, à l'issue de son tracé le pointeur source doit malgré tout référencer le début de la ligne suivante. La même chose est valable pour le bord inférieur, ici ce sera la distance aux données du plan suivant qui interviendra.

Pour les bords supérieur et gauche, un problème supplémentaire fait son apparition. L'abscisse où commence le dessin doit être remise à 0 pour que ce dernier ait lieu à l'intérieur de l'écran. Au bord gauche, le plan de début et l'offset de début de la première colonne à copier seront modifiés. Si un sprite se déplace par exemple d'un point vers le bord gauche, la première colonne se trouvera à l'abscisse -1 et ne devra pas être dessinée. La colonne suivante du premier plan du sprite sera alors en abscisse 3, valeur qui devra être ajustée.

D'une façon générale, le clipping sur les bords verticaux devra tenir compte de la répartition des points dans les différents plans. On applique un principe semblable à celui mis en oeuvre pour le calcul de la largeur mais en partant cette fois-ci du nombre de pixels restants. Pour ne pas compliquer inutilement les choses, on supposera qu'un clipping à gauche exclut un clipping simultané à droite, sinon on aurait un sprite de plus de 321 pixels de large, ce qui n'est pas très courant dans la pratique. On s'épargnera donc l'étude de la combinaison bord gauche/bord droit.

7.4. L'UNITÉ SPRITES

Tous les points discutés sont appliqués dans les procédures GetSprite et PutSprite de l'unité Sprites :

```
Unit Sprites;
Interface
Type SpriteTyp=Record              {structure d'un bloc de données
                                    décrivant un sprite}
  Adr:Pointer;                     {pointe sur les données
                                    graphiques }
  dtx,dty:Word;                    {largeur et hauteur en pixels}
  px,py,                           {position courante, facultatif *}
  sx,sy:Integer;                   {vitesse courante, facultatif *}
End;
{*: facultatif veut dire que les routines GetSprite et PutSprite
 n'utilisent pas ces variables qui ne servent qu'à faciliter une
 gestion par le programme principal }
Procedure GetSprite(Ofs,dtx,dty:Word;var dSprite:SpriteTyp);
{lit un sprite de largeur dtx, de hauteur dty, à l'offset ofs
 de vscreen, dSprite est l'enregistrement qui va mémoriser le sprite}
Procedure PutSprite(pg_ofs,x,y:Integer;sSprite:spritetyp);
{copie un sprite de la mémoire centrale (situation et taille dans
 sSprite) vers la mémoire d'écran en page pg et coordonnées (x,y}

Implementation
Uses ModeXLib;
Var i:Word;
Procedure GetSprite;
Var ppp:Array[0..3] of Byte;       {table avec le nombre de pixels
                                    à copier par plan }
    Skip:word;                     {nombre d'octets à sauter}
    Plane_Count:Word;              {nombre de plans déjà copiés }
Begin
  GetMem(dSprite.adr,dtx*dty);     {alloue de la mémoire }
  dSprite.dtx:=dtx;                {prend note de la largeur et
                                    de la hauteur dans l'enregis-
                                    trement du sprite }
```

```
  dSprite.dty:=dty;
  i:=dtx shr 2;                    {nombre de groupes de 4 octets}
  ppp[0]:=i;ppp[1]:=i;             {correspond au nombre minimum
                                    d'octets à copier }
  ppp[2]:=i;ppp[3]:=i;
  For i:=1 to dtx and 3 do         {mémorise en ppp les pixels
                                    restants}
    Inc(ppp[(i-1) and 3]);         {ajoute les pixels en commen-
                                    çant par le plan de début}
  Plane_Count:=4;                  {copie 4 plans }
asm
  push ds
  mov di,word ptr dSprite          {charge d'abord un pointeur sur
                                    le bloc de données}
  les di,[di]                      {es:di va pointer sur les données
                                    graphiques}
  lea bx,ppp                       {bx référence le tableau ppp}
  lds si,vscreen                   {charge un pointeur sur l'écran}
   add Ofs,si                      {ajoute l'offset des données
                                    du sprite proprement dites}
@lcopy_plane:                      {boucle des plans}
  mov si,ofs                       {si reçoit l'adresse de début
                                    des données du sprite }
  mov dx,dty                       {le compteur d'ordonnées reçoit
                                    le nombre de lignes}
  xor ah,ah                        {efface ah}
  mov al,ss:[bx]                   {charge en al l'élément courant
                                    de ppp}
  shl ax,2                         {on déplace des groupes de 4 octets}
  sub ax,320                       {calcule la différence par
                                    rapport à 320}
  neg ax                           {convertit ax-320 en 320-ax }
  mov skip,ax                      {sauvegarde le résultat en Skip}
@lcopy_y:                          {boucle des lignes }
  mov cl,ss:[bx]                   {tire la largeur du tableau ppp}
@lcopy_x:                          {boucle des pixels }
  movsb                            {copie un octet }
  add si,3                         {point suivant dans le même plan}
```

```
  dec cl                          {copie tous les points de la ligne}
  jne @lcopy_x
  add si,skip                     {passe à la ligne suivante }
  dec dx                          {copie toutes les lignes }
  jne @lcopy_y
  inc bx                          {pointe sur l'élément suivant
                                   de ppp}
  inc ofs                         {se positionne sur le nouveau début
                                   de plan }
  dec plane_count                 {copie tous les plans }
  jne @lcopy_plane
  pop ds
End;
End;
Procedure PutSprite;
var plane_count,                  {nombre de plans déjà copiés }
    masqueplan:Byte;              {masque Write-Plane dans le
                                   registre 2 du TS }
    Skip,                         {nombre d'octets à sauter }
    ofs,                          {offset courant dans la mémoire
                                   d'écran }
    plane,                        {numéro du plan courant }
    Largeur,                      {largeur en octets à copier
                                   dans une ligne }
    dty:Word;                     {hauteur }
    source:Pointer;               {pointe sur les données
                                   graphiques lorsque ds varie }
    clip_lt, clip_rt:integer;     {nombre de pixels restant à
                                   droite et à gauche }
    clipact_lt,                   {nombre d'octets restants }
    clipact_rt,                   {pour le plan courant }
    clip_dn,clip_up:Word;         {nombre de lignes restant en
                                   haut et en bas }
    ppp:Array[0..3] of Byte;      {nombre de pixels par plan }
    cpp:Array[0..3] of Byte;      {nombre d'octets restants par plan }
Begin
  if (x > 319) or                 {pas de dessin }
  (x+sSprite.dtx < 0) or          {sprite en dehors de l'écran ?}
```

```
  (y > 199) or
  (y+sSprite.dty < 0) then exit;
  clip_rt:=0;                    {normalement pas de clipping }
  clip_lt:=0;                    {-> toutes les variables de
                                  clipping à 0 }
  clip_dn:=0;
  clip_up:=0;
  clipact_rt:=0;
  clipact_lt:=0;
   with sSprite do begin
    if y+dty > 200 then begin    {premier cas de clipping : en bas}
      clip_dn:=(y+dty-200);      {nombre de lignes restantes }
      dty:=200-y;                {réduit la hauteur du sprite }
    End;
    if y<0 then begin            {deuxième cas de clipping : en haut}
      clip_up:=-y;               {nombre de lignes restantes }
      dty:=dty+y;                {réduit la hauteur du spr }
      y:=0;                      {ordonnée de départ nulle, au
                                  bord supérieur de l'écran }
    End;
    if x+dtx > 320 then begin    {troisième cas de clipping:à droite }
      clip_rt:=x+dtx-320;        {pixels restants }
      dtx:=320-x;                {réduit la largeur }
    End;
    if x<0 then begin            {quatrième cas de clipping:à gauche }
      clip_lt:=-x;               {nombre de pixels restants }
      plane:=4-(clip_lt mod 4); {nouveau plan de début pour la
                                 colonne 0 }
      plane:=plane and 3;        {ramène à l'intervalle 0..3 }
      ofs:=pg_ofs+80*y+((x+1) div 4) - 1;
                                 {Ofs sur groupe de 4 octets }
      x:=0;                      {colonne de début }
    End Else Begin               {à droite pas de clipping?}
      plane:=x mod 4;            {alors calcul conventionnel du plan }
      ofs:=pg_ofs+80*y+(x div 4);{et de l'offset }
    End;
  End;
```

```
  Source:=sSprite.adr;          {mémorise pointeur sur les
                                 données graphiques }
  dty:=sSprite.dty;             {et hauteur dans variables locales }
  Largeur:=0;                   {initialise largeur et Skip }
  Skip:=0;
  i:=sSprite.dtx shr 2;         {nombre de groupes de 4 octets
                                 complets}
  ppp[0]:=i;ppp[1]:=i;          {correspond au nombre minimum
                                 d'octets à copier }
  ppp[2]:=i;ppp[3]:=i;
  For i:=1 to sSprite.dtx and 3 do {mémorise en ppp les pixels
                                    restants }
    Inc(ppp[(plane+i - 1) and 3]); {ajoute les pixels en commençant
                                    par le plan de début }
  i:=(clip_lt+clip_rt) shr 2;
  cpp[0]:=i;cpp[1]:=i;          {valeurs par défaut du clipping}
  cpp[2]:=i;cpp[3]:=i;
  For i:=1 to clip_rt and 3 do  {si clipping à droite }
    Inc(cpp[i-1]);              {enregistre le nombre dans les plans}
  For i:=1 to clip_lt and 3 do  {si clipping à gauche }
    Inc(cpp[4-i]);              {enregistre le nombre dans les plans}
asm
  mov dx,3ceh                   {met le registre 5 du GDC (GDC Mode)}
  mov ax,4005h                  {en mode Write 0 }
  out dx,ax
  push ds                       {sauve ds}
  mov ax,0a000h                 {charge le segment de destination
                                 (VGA) }
  mov es,ax
  lds si,source                 {ds:si va pointer sur la source
                                 (données graphiques ) }
  mov cx,plane                  {masque de plan de départ }
  mov ax,1                      {décale le bit 0 vers la gauche }
  shl ax,cl
  mov masqueplan,al             {sauve le masque }
  shl al,4                      {le reporte dans le quartet
                                 supérieur }
  or masqueplan,al
```

```
  mov plane_count,4              {4 plans à copier }
 @lplane:                        {boucle des plans }
  mov cl,byte ptr plane          {charge le plan courant }
  mov di,cx                      {en di}
  mov cl,byte ptr ppp[di]        {charge le nombre ppp
                                  corrrespondant en cx }
  mov byte ptr Largeur,cl        {recalcule Skip }
  mov ax,80                      {forme la différence 80-largeur}
  sub al,cl
  mov byte ptr skip,al           {et la reporte dans skip }
  mov al,byte ptr cpp[di]        {charge la largeur de clipping
                                  spécifique au plan }
  cmp clip_lt,0                  {si pas de clipping à gauche,
                                  voyons à droite }
  je @adroite                    {sauve en clip_act_lt clip.
                                  plan + spécif.}
  mov clipact_lt,ax
  sub Largeur,ax                 {réduit le nombre d'octets à copier}
  jmp @clip_rdy                  {à droite pas de clipping}
@adroite:                        {si pas de clipping à gauche}
  mov clipact_rt,ax              {clipping pour tous les plans
                                  en clip_act}
@clip_rdy:
  mov ax,Largeur                 {calcule la largeur totale en octets}
  add ax,clipact_rt
  add ax,clipact_lt
  mul clip_up                    {multiplie par  le nombre de
                                  lignes du clipping supérieur}
  add si,ax                      {octets non représentés }
  mov cx,Largeur                 {charge la largeur en cx }
  or cl,cl                       {largeur 0, plan terminé }
  je @planfini
  mov di,ofs                     {offset de destination dans
                                  l'écran en di}
  mov ah,masqueplan              {réduit le masque de plan aux
                                  bits [0..3] }
  and ah,0fh
```

```
  mov al,02h                {active le registre 2 du TS
                             (Write Plane Mask)}
  mov dx,3c4h
  out dx,ax
  mov bx,dty                {initialise le compteur de lignes y}
@lcopy_y:                   {boucle des lignes y }
  add si,clipact_lt         {pointeur source au-delà
                             clipping gauche }
  add di,clipact_lt         {idem pour le pointeur de
                             destination }
@lcopy:                     {boucle des pixels }
  lodsb                     {lit un octet }
  or al,al                  {si 0, pas de traitement }
  je @Valeur0
  stosb                     {sinon copie }
@entry:
  loop @lcopy               {suite de la boucle }
  add si,clipact_rt         {après ligne complète, clipping
                             droit }
  dec bx                    {incrémente le compteur de lignes }
  je @planfini              {compteur de lignes  = 0,
                             plan suivant }
  add di,skip               {sinon passe en début de ligne
                             suivante }
  mov cx,Largeur            {réinitialise le compteur de
                             colonnes }
  jmp @lcopy_y              {retourne dans la boucle des lignes}
@valeur0:                   {couleur du sprite 0 }
  inc di                    {saute l'octet de destination}
  jmp @entry                {retourne dans la boucle }
@planfini:                  {fin de la boucle des lignes}
  mov ax,Largeur            {calcul la largeur globale en octets}
  add ax,clipact_rt
  add ax,clipact_lt
  mul clip_dn               {multiplie par le nombre de
                             lignes du clipping inf. }
  add si,ax                 {octets non réprésentés }
  rol masqueplan,1          {masque le plan suivant }
```

```
  mov cl,masqueplan          {a-t-on sélectionné le plan 0?}
  and cx,1                   {(bit 1 à 1), alors}
  add ofs,cx                 {incrémente de 1 l'offset de
                              destination (cx bit 1 !)}
  inc plane                  {incrémente le numéro de plan
                              (indice dans ppp) }
  and plane,3                {réduit de 0 à 3 }
  dec plane_count            {déjà 4 plans copiés, on arrête}
  jne @lplane
  pop ds                     {restaure ds, ciao}
End;                         {asm}
End;
Begin
End.
```

En plus des deux procédures, l'unité contient la définition d'un enregistrement de sprite. Les données caractéristiques d'un sprite y sont reportées, on crée une variable de ce type par sprite.

Le premier élément de cet enregistrement est un pointeur sur les données graphiques proprement dites situées en mémoire centrale (adr). Cet élément est suivi de la largeur du sprite (dtx), de sa hauteur (dty). Toutes ces variables sont remplies par GetSprite et exploitées par PutSprite, elles ne doivent pas être modifiées par d'autres procédures.

Le bloc en question peut être librement étendu, selon les besoins de l'application. Pour la gestion de déplacements linéaires, on a prévu en plus quatre variables qui ne sont pas utilisées par les procédures de l'unité : px et py donnent les coordonnées courantes du sprite, et sx et sy sa vitesse selon les deux axes (déplacement à chaque génération d'écran). C'est le programme principal qui décide d'exploiter ou non ces variables, l'unité ne fait que les mettre à sa disposition. Vous pouvez très bien les négliger.

Avant que des sprites ne puissent apparaître à l'écran, il faut déjà qu'ils existent. Nous allons à nouveau faire appel au format GIF, en raison de sa simplicité et de sa compacité. Quand l'image qui contient les sprites a été chargée, il faut les disposer correctement en mémoire centrale : c'est là le but de la procédure GetSprite de la présente unité.

Elle suppose que les sprites se trouvent dans l'écran vscreen et que vous en connaissiez les coordonnées. Ces dernières sont en effet transmises comme argument à l'appel de la procédure, non pas sous forme de couple de nombres mais comme offset unique, ce qui ne présente aucune difficulté en raison de

la vieille formule : offset:=y*320+x (vscreen contient l'image dans le format du mode 13h).

C'est à cet offset relatif à vscreen que GetSprite commence à lire les données graphiques. La taille de l'image est définie, ainsi que la largeur dtx et la hauteur dty. Le dernier argument de la procédure GetSprite est une variable du type SpriteTyp qui mémorise les données du sprite.

GetSprite prend soin de mettre à jour les éléments de l'enregistrement avec les valeurs appropriées. Les indications de taille transmises sont ainsi mémorisées et seront utilisées pour les allocations de mémoire.

La procédure règle alors le problème des largeurs qui ne sont pas les mêmes dans les différents plans. Comme nous l'avons vu, un sprite dont la largeur n'est pas un multiple exact de 4 se répartit inégalement dans les différents plans. La procédure calcule de ce fait un tableau qui contient les différentes largeurs des plans. Au début, tous les plans ont la largeur dtx/4 puis une boucle répartit le solde, en partant du plan 0.

La partie en assembleur qui suit contient la routine de copie proprement dite. Elle se compose de trois boucles imbriquées. On charge d'abord en es:di (respectivement ds:si) le pointeur de destination (zsprite.adr) et le pointeur source (vscreen+Ofs). Par ailleurs, bx référence le tableau ppp dans le segment de pile (variable locale !).

La boucle extérieure (lcopy_plane) est parcourue une fois par plan, soit quatre fois. Elle fournit à si la nouvelle valeur de départ, met en dx la hauteur du sprite et en skip la nouvelle valeur qui exprime la distance entre les lignes dans ce plan.

La boucle lcopy_y est traitée une fois par ligne de sprite. Elle remplit cx avec la largeur de ligne courante avant le processus de copie. Après copie d'une ligne, le saut skip est rajouté à si pour arriver au début de la ligne suivante du sprite. Cette boucle est itérée dx fois, dx ayant stocké la hauteur du sprite dans la boucle des plans.

La boucle la plus interne ne fait rien d'autre que de copier un octet. Comme les données d'un plan sont rangées selon le format du mode 13h utilisé par vscreen, la distance d'un point au suivant est de 4 pixels : on augmente donc si de 3 puisque movsb effectue une incrémentation de 1.

La deuxième procédure de cette unité (PutSprite) ressemble dans son principe à GetSprite mais elle est plus compliquée. On lui transmet quatre arguments : d'abord l'offset de la page d'écran courante, en général vpage, ensuite les coordonnées x et y du sprite, relatives à son coin supérieur gauche. Le dernier argument est l'enregistrement Sprite que nous avons déjà vu dans GetSprite et d'où PutSprite tire les informations nécessaires au dessin du sprite.

Les coordonnées montrent déjà une particularité de PutSprite par rapport à GetSprite. Ces deux variables sont en effet définies comme integer, elles peuvent donc être négatives. Dans ce cas, le coin supérieur gauche du sprite est en dehors de l'écran, il faut donc écrêter le sprite par un clipping en haut ou à gauche. Il en sera de même lorsque les coordonnées seront suffisamment importantes pour faire disparaître le sprite en bas ou à droite.

La procédure commence en fait par traiter le clipping. Dans le cas le plus simple à prendre en compte, le sprite n'est pas visible sur l'écran. Il ne s'agit pas à proprement parler de clipping, le premier test If qui s'en occupe arrête le dessin par un ordre exit et le contrôle est rendu au programme appelant.

Si le sprite est au moins visible en partie, la procédure entre dans sa deuxième phase qui calcule réellement le clipping. Toutes les variables impliquées dans l'opération sont initialisées à 0, ce qui équivaut à un affichage complet. On évalue ensuite pour chaque bord l'écrêtement nécessaire.

On traite d'abord le cas élémentaire ou le clipping a lieu au bas de l'écran (if y+dy>200). Il suffit alors de réduire la hauteur au domaine visible. Mais le sprite est parcouru quatre fois en tout, il faudra donc veiller à ce que, compte tenu des circonstances à la sortie de chaque plan, le pointeur source indique bien le début du plan suivant. On utilise à cet effet une variable clip_dn qui contient le nombre de lignes à sauter à la fin de chaque plan parce qu'elles ne sont pas affichées.

Le clipping en haut de l'écran est également assez simple à traiter. Le nombre de lignes à sauter est également mémorisé (en clip_up) et la hauteur est diminuée. Par ailleurs, pour indiquer la nouvelle ligne de début, y doit être initialisé à 0.

Du côté du bord droit, l'affaire se complique quelque peu. En cas de clipping horizontal, il faut tenir compte de la présence des plans qui nécessitent des traitements différents comme nous l'avons déjà vu pour la largeur du sprite. La boucle suivante traite ces particularités. On commence comme dans le clipping vertical par prendre note du nombre de pixels (dans clip_rt) et on diminue la largeur.

A gauche, cependant, la largeur est conservée et ajustée à l'intérieur de la routine de copie, pour assurer la collaboration avec le clipping vertical. Mais le plan de début doit être recalculé. Prenons par exemple un sprite qui dépasse de trois pixels à gauche de l'écran. La représentation commence toujours par le premier bloc de données du plan à l'intérieur du sprite dans lequel se trouve ici la colonne -3. Mais comme la colonne -3 n'apparaît pas, c'est la première colonne visible de ce plan qui est utilisée, c'est-à-dire dans notre exemple -3+4=1. Le calcul de plan tient compte de ces considérations. L'offset lui aussi doit être recalculé car, avec une abscisse négative, il pointerait sur la ligne

précédente. Comme au bord supérieur pour l'ordonnée y, on utilise une initialisation à 0 pour l'abscisse x.

Les modifications de plan et d'offset ne sont valables que pour le clipping à gauche, dans les autres cas le calcul reste classique, conforme aux habitudes du mode X.

Après initialisation de quelques variables, la procédure GetSprite calcule le tableau ppp des différentes largeurs en fonction du plan. On détermine de la même façon le tableau des cpp qui mémorise les valeurs de clipping pour le bord droit ou le bord gauche. Il n'est pas possible ici de pratiquer un clipping des deux côtés en même temps, ce perfectionnement sans grand intérêt ne ferait que compliquer inutilement la programmation.

Pour déterminer dans chaque plan le nombre de groupes de 4 octets, on part de la largeur de clipping (clip_lt+clip_rt) divisée par 4. A ce résultat, il faut ajouter dans le cadre d'une boucle le nombre spécifique au plan. Il faut bien respecter la différence entre le bord droit et le bord gauche. A droite, le reste se répartit à partir du plan 0 (cpp[i-1]), tandis qu'à gauche il commence au plan 3 (cpp[4-i]).

La partie en assembleur débute par l'initialisation du mode d'écriture 0 qui permet un adressage individuel de chaque point. Après chargement du segment de destination et du pointeur source, il faut déterminer le masque de sélection du plan courant. Ce masque est à envoyer sur le TS et l'un de ses bits doit correspondre au plan concerné. On convertit un nombre compris entre 0 et 3 en un bit : 0 en 0001b, 1 en 0010b, 2 en 0100b et 3 en 1000b. Ce masque est sauvegardé en masqueplan pour être ultérieurement soumis à une rotation vers la gauche à chaque changement de plan.

La variable plane_count remplit une toute autre tâche. Il s'agit d'un simple compteur qui veille à ce que la boucle des plans soit parcourue quatre fois. Cette boucle appelée @lplane commence par charger la largeur courante dans la variable de même nom à partir du tableau ppp. Puis elle calcule la longueur à sauter dans la variable skip.

Le tableau cpp est mis à contribution pour préparer le clipping. S'il se fait à droite, la variable clipact_rt est chargée avec la valeur de cpp et est disponible à chaque itération de ligne pour indiquer le nombre d'octets à sauter. A gauche, la variable clipact_lt sert à une fin identique, mais la largeur doit être réduite en plus, ce qui ne peut se faire dans la partie Pascal à cause du changement de plan de départ. Il faut donc rattraper cet oubli pour chaque plan individuellement.

Après tous ces préparatifs, c'est le moment de résoudre le premier cas de clipping. La largeur totale d'une ligne, y compris le clipping droit et gauche,

est multipliée par le nombre de lignes à écrêter en haut. Le nombre d'octets correspondants est ajouté à si pour n'être pas pris en compte.

A cet endroit, on teste la largeur du sprite dans le plan : si elle est nulle, pas la peine de se fatiguer à faire une copie. Si ce test ne concernait que les sprites de largeur nulle, il serait inutile. Mais à cause du mode X, les plans ne présentent pas la même largeur. Pour cette raison, les sprites de moins de quatre pixels ne sont pas représentés dans tous les plans.

Après détermination du plan courant par le registre 2 du séquenceur de timing, la boucle imbriquée suivante @lcopy_y peut commencer. Elle gère d'abord le clipping gauche de chaque ligne en avançant les pointeurs, puis copie la ligne et finit par traiter le clipping droit (add si, clipact_rt). Ensuite, le pointeur de destination est positionné sur le début de la ligne suivante et le compteur cx rechargé avec la largeur courante. La boucle de copie proprement dite (@lcopy) déplace simplement un octet de l'adresse source à l'adresse de destination, à moins qu'il ne s'agisse d'un 0 qui caractérise un pixel du fond d'écran et n'est pas pris en compte.

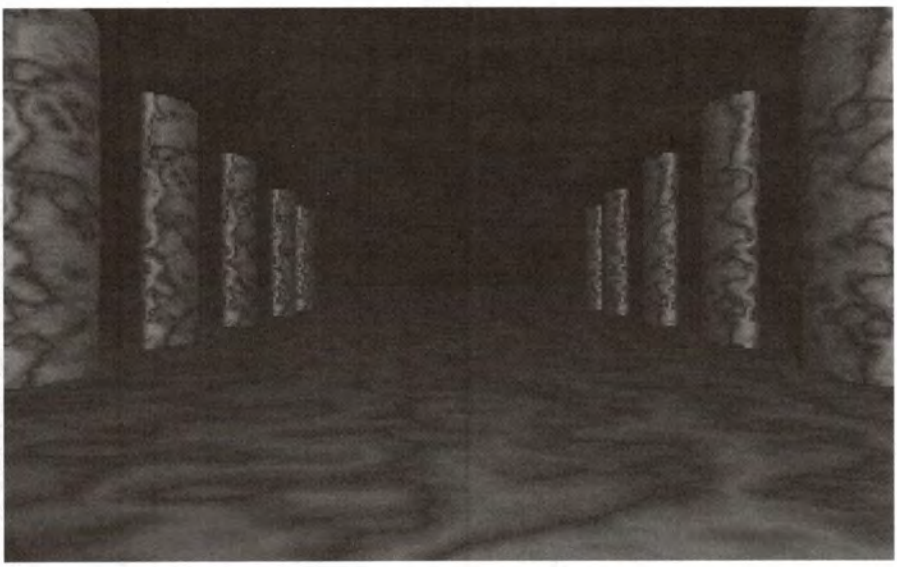
Une fois le traitement d'un plan achevé (@plan_fini), on calcule le clipping inférieur comme le supérieur, puis le masque de plan est soumis à rotation. La plupart du temps, cette rotation donne lieu, une fois par sprite, à un débordement : le plan 0 est copié à la suite du plan 3. Dans ce cas, l'offset doit être également augmenté pour éviter qu'une colonne ne soit copiée à gauche du sprite. On peut intercepter cette situation par une combinaison CMP-JNE inefficace et lente mais ici, on a choisi une voie plus rapide. En cas de débordement, le masque de plan (masqueplan) saute de 1000b à 0001b : on extrait le bit 0 et on l'ajoute à l'offset, ce qui ne donne quelque chose que si le masque est 0001b, tous les autres cas s'éliminant d'eux-mêmes par l'opérateur AND. La boucle des plans s'achève après mise à jour des deux compteurs de plans plane et plane_count.

Encore un mot à propos du code inline de ces routines. Il est généralement plus judicieux de gérer à l'extérieur les modules en assembleur et de les assembler séparément car on profite alors des énormes avantages d'un assembleur professionnel (macros, structures, instructions REP, etc). Mais ici, les parties en assembleur ont besoin d'un certain nombre de variables locales issues de la partie en Pascal : de ce fait, on a préféré mettre en service des instructions ASM.

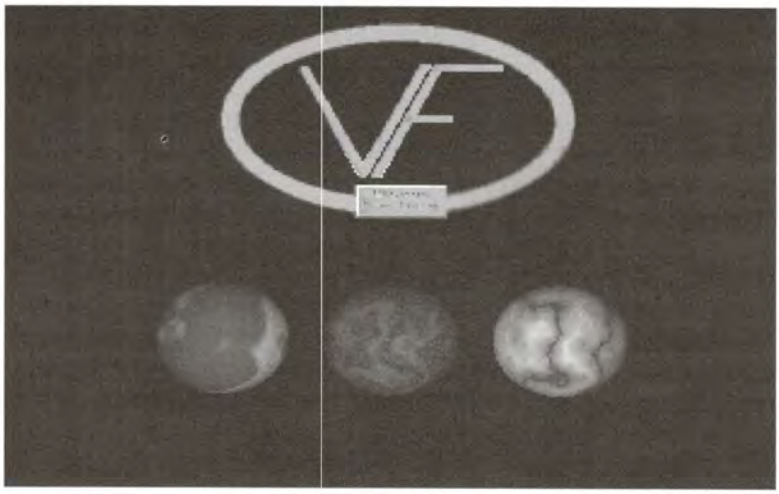
Le programme SPRT_TS.PAS donne un exemple d'application de ces routines :

```
Uses Crt,Gif,ModeXLib,Sprites;
Const NbSprites=3;                {nombre de sprites utilisés
                                   dans le programme }
Var Sprite:Array[1..NbSprites] of SpriteTyp;
                                  {données représentant les sprites }
    i:Word;                       {compteur}

Begin
  Init_ModeX;                     {enclenche le mode X }
  LoadGif('sprites');              {charge image avec trois sprites }
  GetSprite(62 +114*320,58,48,Sprite[1]);
                                  {coordonnées (62,114), dimensions
                                   58*48}
  GetSprite(133+114*320,58,48,Sprite[2]);
                                  {(133,114), 58*48}
  GetSprite(203+114*320,58,48,Sprite[3]);
                                  {(203,114), 58*48}
                                  {charge les trois sprites }
  LoadGif('phint');               {charge l'arrière-plan }
  p13_2_ModeX(48000,16000);       {et le copie dans la page de fond }
  With Sprite[1] do Begin         {coordonnées et vitesse }
    px:=160;py:=100;              {des trois sprites (valeurs
                                   arbitraires)}

    sx:=1;sy:=2;
  End;
  With Sprite[2] do Begin
    px:=0;py:=0;
    sx:=1;sy:=-1;
  End;
  With Sprite[3] do Begin
    px:=250;py:=150;
    sx:=-2;sy:=-1;
  End;
  Repeat
    CopyScreen(vpage,48000);      {fond d'écran dans page courante}
```

```
    For i:=1 to NbSprites do      {boucle parcourue pour tous les
                                   3 sprites }
      With Sprite[i] do Begin
        Inc(px,sx); Inc(py,sy);   {déplacement}
        If (px < -dtx div 2)      {arrivé à gauche ou à droite ?
                                   -> demi-tour}
          or (px > 320-dtx div 2) Then sx:=-sx;
        If (py < -dty div 2)     {arrivé en bas ou en haut :
                                   -> demi-tour }
          or (py > 200-dty div 2) Then sy:=-sy;
        PutSprite(vpage,px,py,Sprite[i]);{dessine le sprite }
      End;
    switch;                       {active la page calculée }
    WaitRetrace;                  {attend le retour de balayage
                                   vertical }
  Until KeyPressed;               {avant de procéder à une autre
                                   modification }
  ReadLn;
TextMode(3);
 End.
```

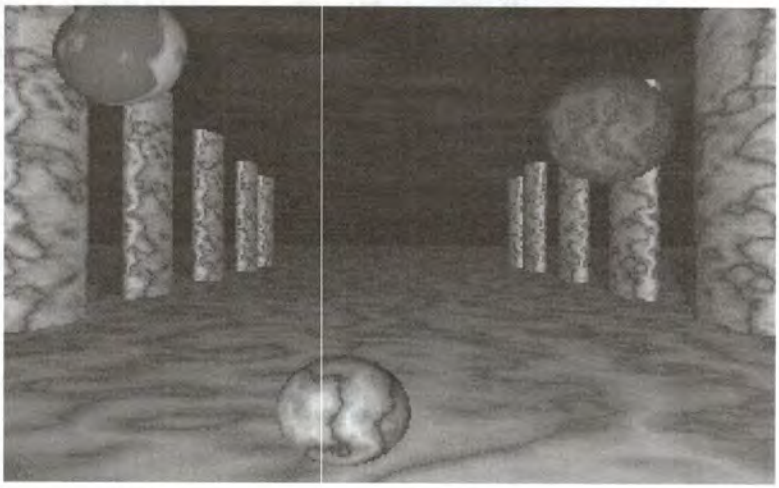


L'image qui va nous servir de fond...



...et les sprites qui vont bientôt s'animer

Une fois chargés, les sprites sont placés à des endroits arbitraires et on leur assigne des vitesses quelconques. La boucle Repeat concrétise le principe fondamental de la représentation des sprites : on restaure le fond (copyscreen), on déplace les sprites et on les affiche. L'appel à WaitRetrace se trouve cette fois après l'instruction switch car l'activation de la nouvelle page a lieu pendant le délai de synchronisation. On suit donc une logique un peu différente de ce qui se faisait jusqu'ici. L'instruction de commutation de page est d'abord envoyée à la carte VGA. Si on attend le retour de balayage vertical, on peut être sûr que la commutation a lieu à ce moment et que l'on peut préparer la page suivante.



Nos trois globes se déplacent librement dans notre image de départ

Malheureusement, les routines de sprite demandent énormément de ressources de calcul, c'est pourquoi les ordinateurs les plus lents seront débordés par la gestion de trois sprites. Vous serez alors amené à diminuer ce nombre.

7.5. RÉDUCTIONS D'ÉCHELLE POUR SIMULER DES MOUVEMENTS RÉALISTES

Lorsqu'une figure de jeu d'aventure court vers l'arrière de l'écran, elle diminue de taille pour donner une impression de fuite tridimensionnelle. Avec les moyens étudiés jusqu'ici, il n'y aurait qu'une seule possibilité d'obtenir cet effet : on définit pour chaque étape de la course un sprite de la bonne taille, en faisant appel à un logiciel de dessin. C'est la méthode la plus rapide mais aussi la plus coûteuse en mémoire. On préférera souvent procéder autrement, en réduisant le sprite à l'aide d'un programme à partir du seul modèle original.

Pour résoudre ce problème, il existe des algorithmes mathématiques généraux qui projettent chaque point sur son nouvel emplacement. Mais ces calculs sont trop longs, point n'est besoin d'insister. Nous allons donc explorer une autre méthode plus simple qui repose sur une idée différente mais en accord avec les principes mathématiques de base.

A titre d'exemple, nous allons expliquer le changement d'échelle selon l'axe des ordonnées, ce qui nous permettra de réaliser un logo tournant. Il revient au même de travailler dans l'axe des abscisses mais, pour simplifier l'exposé, nous nous limiterons à la perspective verticale.

Pour réduire un sprite selon un facteur égal à une fraction de base, par ex 1/2, il suffit de représenter une ligne sur deux. Une réduction au tiers se fera en affichant une ligne sur trois. Cette méthode est applicable à toute fraction et même à une réduction exprimée en pourcentage.

Il faudra ainsi rechercher au bout de combien de lignes il faut en laisser passer une. On introduit dans l'ordonnée y une partie décimale. Au moment de l'affichage, on procède à un arrondi entier. En effet, si on considère l'écran d'un point de vue mathématique, il est composé non seulement de lignes d'ordonnée entière mais aussi de lignes intermédiaires en nombre infini, qui sont utilisées dans le processus de réduction. Mais comme l'écran réel, lui, n'a qu'un nombre limité de lignes, les lignes intermédiaires doivent être affichées à l'aide des lignes disponibles.

Voilà pour la théorie mais que donne la mise en pratique ? Nous allons faire intervenir des calculs en virgule fixe (addition et soustraction) en raison de

leur efficacité. L'ordonnée relative y à l'intérieur d'un sprite présentera une partie entière (qui n'est pas exprimée explicitement dans une variable) et une partie décimale (rel_y). A chaque ligne, on additionne une constante (add_y) à la partie décimale. Si le résultat dépasse 100, ce qui correspond à un débordement dans la partie entière, on en tire les conséquences : la partie entière est incrémentée, ce qui se concrétise par la non prise en compte de la ligne suivante. Par ailleurs, la partie décimale est diminuée de 100.

La constante suit la devise : plus le facteur d'échelle est grand (proche de l'original), plus je me fais petite car plus les sauts de ligne deviennent rares. Elle est simplement déterminée par la différence 100-facteur d'échelle.

Exemple : si un sprite est réduit à 60 %, la constante additionnée est égale à 40. Après la première ligne, la partie décimale est de 40, à la seconde elle passe à 80. Il ne se passe rien pour l'instant. Mais au-delà de la troisième ligne, la partie décimale atteint la valeur de 120 et dépasse le seuil autorisé de 100. Elle est donc ramenée à 20 et la ligne suivante n'est pas prise en compte. Deux lignes plus loin, le phénomène se reproduit avec une partie décimale de 100 et le petit jeu recommence à 0.

Le fait de ne pas prendre en compte la ligne peut se faire de deux manières : on peut survoler la ligne dans les données source en avançant le pointeur, ou laisser le pointeur de destination en place et écrire la ligne suivante par-dessus l'ancienne. Cette dernière alternative aura notre préférence : toutes les lignes seront représentées même si elles ne sont pas toutes visibles. La vitesse de génération sera toujours la même indépendamment de la réduction d'échelle. Par ailleurs, cette méthode facilite la création de sprites réfléchis, comme le montre le programme SCAL_TST.PAS.

```
Uses Crt,Sprites,ModeXLib,Gif,Tools;
Procedure PutScalSprt(pg_ofs,x,y,scale_y:Integer;sSprite:
                                                spritetyp);
var planecount,                    {nombre de plans déjà copiés}
    masqueplan:Byte;               {masque le plan Write dans le
                                    registre 2 du TS }
    Skip,                          {nombre d'octets à sauter }
    ofs,                           {offset courant dans la mémoire
                                    d'écran }
    plane,                         {numéro du plan courant }
    Largeur,                       {largeur en nombre d'octets à
                                    copier dans une ligne }
    dty:Word;                      {hauteur }
    source:Pointer;                {pointe sur les données
                                    graphiques quand ds varie}
```

```
    ppp:Array[0..3] of Byte;      {nombre de pixels par plan }
    rel_y,                        {partie décimale de l'ordonnée
                                   relative }
    add_y:Word;                   {partie décimale à additionner}
    direction:Integer;            {direction de déplacement
                                   (+/- 80)}
    i:Word;                       {compteur d'itérations local}
Begin
  if (x + sSprite.dtx > 319)      {Clipping ? on arrête }
  or (x < 0)
  or (y + sSprite.dty*scale_y div 100 > 199) or (y < 0) then exit;
  add_y:=100-abs(scale_y);        {calcule le nombre à additionner}
  if scale_y < 0 then direction:=-80 else direction:=80;
                                  {fixe la direction }
  Source:=sSprite.adr;            {pointe sur les données
                                   graphiques }
  dty:=sSprite.dty;               {charge la hauteur en local }
   plane:=x mod 4;                {plan et }
  ofs:=pg_ofs+80*y+(x div 4);     {offset de début }
  Largeur:=0;                     {initialise la largeur et le saut }
  Skip:=0;
  i:=sSprite.dtx shr 2;           {nombre de groupes de 4 octets
                                   complets}
  ppp[0]:=i;ppp[1]:=i;            {nombre minimum d'octets à copier}
  ppp[2]:=i;ppp[3]:=i;
  For i:=1 to sSprite.dtx and 3 do
                                  {mémorise en ppp les pixels
                                   restants }
    Inc(ppp[(plane+i - 1) and 3]);
                                  {ajoute les pixels en
                                   commençant par le plan de début }
asm
  push ds                         {sauve ds}
  mov ax,0a000h                   {charge le segment de destination
                                    (VGA)}
  mov es,ax
  lds si,source                   {ds:di va pointer sur la source
                                   (données graphiques) }
```

```
  mov cx,plane              {crée le masque de plan de début}
  mov ax,1                  {décale le bit 0 vers la gauche}
  shl ax,cl
  mov masqueplan,al         {sauve le masque }
  shl al,4                  {le recopie dans le quartet
                             supérieur }
  or masqueplan,al
  mov planecount,4          {4 plans à copier }
@lplane:                    {boucle des plans }
  mov cl,byte ptr plane     {charge le plan courant }
  mov di,cx                 {en di}
  mov cl,byte ptr ppp[di]   {charge le nombre ppp
                             correspondant en cx }
  mov byte ptr Largeur,cl   {recalcule skip }
  mov ax,direction          {forme la différence direction
                             -largeur }
  sub ax,cx
  mov skip,ax               {et écrit le résultat dans skip}
  mov rel_y,0               {on recommence en y=0}
  mov cx,Largeur            {charge la largeur en cx }
  or cl,cl                  {largeur 0, plan terminé }
  je @planfini
  mov di,ofs                {charge en di l'offset de
                             destination dans la mémoire d'écran}
  mov ah,masqueplan         {réduit le masque de plan aux
                             bits [0..3] }
  and ah,0fh
  mov al,02h                {active le registre 2 du TS
                             (Write Plane Mask)}
  mov dx,3c4h
  out dx,ax
  mov bx,dty                {initialise le compteur de lignes y}
@lcopy_y:                   {boucle des lignes }
@lcopy_x:                   {boucle des pixels }
  lodsb                     {lit un octet }
  or al,al                  {si 0, passe}
  je @Valeur0
  stosb                     {sinon copie }
```

```
@entry:
  loop @lcopy_x                 {suite de la boucle }
  mov ax,rel_y                  {ajoute la partie décimale de
                                 la constante additionnelle }
  add ax,add_y
  cmp ax,100                    {partie entière à augmenter ?}
  jb @noaddovfl                 {non, on continue }
  sub ax,100                    {sinon met à jour la partie
                                 décimale }
  sub di,direction              {ligne suivante/précédente }
@noaddovfl:
  mov rel_y,ax                  {réécrit la partie décimale }
  dec bx                        {suite du décomptage de lignes}
  je @planfini                  {compteur  = 0, alors plan suivant }
  add di,skip                   {sinon saute au début de ligne
                                 suivant }
  mov cx,Largeur                {réinitialise le compteur de
                                 colonnes }
  jmp @lcopy_y                  {retourne dans la boucle des lignes}
@valeur0:                       {couleur du sprite 0 }
  inc di                        {saute l'octet de destination}
  jmp @entry                    {retourne dans la boucle }
@planfini:                      {fin de la boucle des lignes }
  rol masqueplan,1              {masque le plan suivant }
  mov cl,masqueplan             {plan 0 sélectionné ?}
  and cx,1                      {(bit 1 à 1), alors}
  add ofs,cx                    {incrémente de 1 l'offset de
                                 destination (bit 1 !)}
  inc plane                     {incrémente le numéro de plan
                                 (indice dans ppp) }
  and plane,3                   {réduit de 0 à 3 }
  dec planecount                {déjà 4 plans copiés, on arrête}
  jne @lplane
  pop ds                        {restaure ds, ciao }
End;                            {asm}
End;
Var Logo:SpriteTyp;
    Sinus:Array[0..99] of Word;
```

```
    Hauteur:Integer;
    i:Word;
Begin
  Init_ModeX;                    {active le mode X }
  LoadGif('sprites');            {charge une image avec un logo}
  GetSprite(88+ 6*320,150,82,Logo);
                                 {initialise le logo }
  LoadGif('phint');              {charge l'image d'arrière-plan}
  p13_2_ModeX(48000,16000);      {et la copie dans la page de fond }
  Sin_Gen(Sinus,100,100,0);      {calcule une table de sinus}
  I:=0;                          {indice à 0 dans la table des sinus}
  repeat
    Inc(i);                      {incrémente l'index }
    Hauteur:=Integer(Sinus[i mod 100]);
                                 {fixe la hauteur à partir du sinus }
    CopyScreen(vpage,48000);     {efface le fond }
    PutScalSprt(vpage,85,100-Hauteur *84 div 200,Hauteur,Logo);
                                 {copie dans la page courante
                                  le sprite en réduction }
    Switch;                      {active la page }
    WaitRetrace;                 {attend le retour de balayage
                                  vertical }
  Until KeyPressed;
  ReadLn;
  TextMode(3);                   {revient au mode texte normal}
End.
```

Ici, un logo se met à tourner devant un fond qui apparaît au travers. La hauteur du logo est périodiquement changée à l'aide d'une table des sinus : la projection verticale d'un mouvement circulaire est en effet un mouvement sinusoïdal. Le cadre du programme correspond à l'ancienne démo de sprite, la gestion du sprite est donc déjà connue : chaque sprite peut être représenté par PutSprite et affiché à l'écran sous forme réduite par PutScalSprt.

Le logo subit une rotation suivant son axe horizontal

La procédure PutScalSprt ne sert qu'à des fins de démonstration, c'est pourquoi elle n'a pas été stockée dans une unité à part. Pour conserver une vitesse acceptable, elle renonce à tout clipping, ce qui d'ailleurs ne s'impose guère pour un logo tournant. Elle reçoit comme argument supplémentaire le nombre scale_y qui donne le pourcentage de réduction à appliquer. Une valeur négative signifie ici une symétrie par rapport à l'axe horizontal, ce qui facilite bien les rotations.

La variable Add_Y est initialisée avec la valeur 100-scale_y qui fait appel au facteur de réduction. La symétrie est commandée par la variable direction qui prend la valeur 80 lorsque le facteur de réduction est positif, et -80 dans le cas contraire, ce qui correspond à la largeur de la ligne. Cette dernière est prise en compte dans le calcul de la variable skip, la largeur du sprite n'étant pas retranchée de 80 mais de la variable direction. Si la direction est négative, la valeur de skip le devient aussi, ce qui correspond à un déplacement vers la ligne précédente. La partie qui traite à proprement parler la réduction se trouve directement après la boucle @lcopy_x, elle est donc exécutée pour chaque ligne affichée. On additionne la partie décimale, ce qui donne éventuellement un débordement. Dans ce cas, la partie décimale est diminuée de 100 et on repasse sur la ligne déjà traitée (en avant ou en arrière selon la valeur de direction).

8. La troisième dimension : programmation de graphiques 3D

Tous les effets montrés jusqu'ici ont une caractéristique commune : ils ne fonctionnent qu'en deux dimensions. Pour disposer d'images plus réalistes, l'introduction de la troisième dimension - la profondeur - est inévitable. Comme le moniteur sur lequel se fait l'affichage est à deux dimensions, il faudra trouver des astuces pour donner une impression de profondeur, par exemple en exploitant une perspective à points de fuite ou en jouant sur des effets d'ombre.

Ces astuces nécessitent des connaissances mathématiques assez étendues, notamment en géométrie et dans le domaine des transformations, des projections et du traitement de la lumière. Les calculs afférents sont très complexes. Ils doivent être optimisés tant sur le plan purement mathématique que sur celui de la programmation informatique, sans quoi la vitesse d'exécution est inacceptable. Le langage requis est l'assembleur car les langages évolués engendrent des structures trop complexes dont le traitement est excessivement long.

Dans le cadre de ce chapitre, nous verrons à la fois des aspects théoriques et leur mise en application dans de brefs programmes. Ces derniers reposent sur les mêmes modules en assembleur (3DASM.ASM, POLY.ASM, BRES.ASM, TEXTURE.INC) qui contiennent à la fois les fonctions générales de représentation (transformations, etc) et les parties spécialisées (effets de lumière) qui sont activées par des variables globales. S'il est vrai que le pilotage par les variables globales n'est pas très élégant, il économise du code et surtout du temps de calcul.

8.1. Mathématiques pour l'amateur de graphiques

Si vous vous y connaissez en géométrie analytique, vous pouvez laisser tomber cette section. Si vous avez quelques lacunes dans ce domaine, nous allons faire connaissance avec les vecteurs, déterminants et produit scalaire.

Le vecteur

Le vecteur en géométrie est l'équivalent du nombre en algèbre. Un vecteur peut être assimilé à une sorte de flèche (les mathématiciens nous pardonneront) qui pointe dans une certaine direction et possède une certaine longueur. Un vecteur peut être librement déplacé dans l'espace, il est par nature relatif. Plus précisément, il est défini par ses composantes ou projections selon les trois axes x, y et z qui donnent sa longueur et sa direction.

Cette définition ne précise pas à quel endroit se trouve exactement le vecteur. Les vecteurs localisés à l'origine ont pour point de départ l'origine du système de coordonnées (0,0) et définissent ainsi un point précis dans l'espace.

Dans la notation classique, on écrit l'un au-dessus de l'autre les composants d'un vecteur selon les axes x, y et z.

$$\vec{a} = \begin{pmatrix} a_x \\ a_y \\ a_z \end{pmatrix} \quad \text{ou} \quad \vec{a} = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix}$$

$$\text{Ex. : } \vec{a} = \begin{pmatrix} 2 \\ 3 \\ -5 \end{pmatrix}$$

Représentation mathématique d'un vecteur

Opérations sur les vecteurs

Comme nous l'avons dit, un vecteur contient à la fois une information de direction et une information de longueur. La dernière nommée est souvent utilisée et se déduit aisément des composantes par application du théorème de Pythagore à l'espace tridimensionnel :

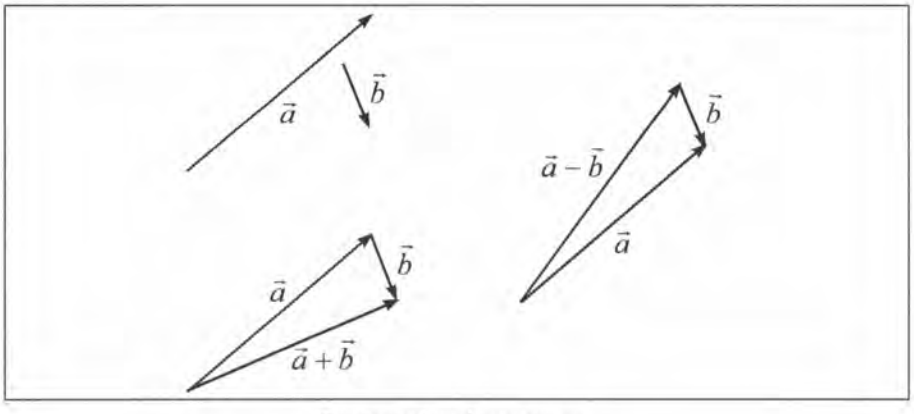
$$a = |\vec{a}| = \left\| \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \right\| = a_1^2 + a_2^2 + a_3^2$$

$$\text{Ex. : } \left\| \begin{pmatrix} 3 \\ -2 \\ 1 \end{pmatrix} \right\| = 3^2 + (-2)^2 + 1^2 = 14$$

Longueur d'un vecteur

Pour faire la somme de deux vecteurs, on additionne leurs composantes respectives. Du point de vue géométrique, additionner des vecteurs revient à les mettre bout à bout.

La soustraction fonctionne de manière analogue. Les composantes sont soustraites. Du point de vue géométrique, l'opposé du deuxième vecteur est mis à la suite du premier.



Soustraction de vecteurs

La multiplication existe sous plusieurs formes

Dans la multiplication scalaire, le vecteur est multiplié par un nombre qui ne modifie que sa longueur. Si ce nombre est négatif, la direction du vecteur s'inverse.

$$(-3)\begin{pmatrix}1\\-2\\3\end{pmatrix}=\begin{pmatrix}-3\\6\\-9\end{pmatrix}$$

Multiplication scalaire

La deuxième variante de multiplication est le produit scalaire qui donne pour résultat un nombre (scalaire). Il en existe deux définitions :

1. $\vec{a}\ \vec{b}=a\ b\ \cos\alpha$

α: angle entre les vecteurs

2. $\vec{a}\ \vec{b}=\begin{pmatrix}a_1\\a_2\\a_3\end{pmatrix}\begin{pmatrix}b_1\\b_2\\b_3\end{pmatrix}=a_1\ b_1+a_2\ b_2+a_3\ b_3$

Ex.: $\begin{pmatrix}1\\2\\3\end{pmatrix}\begin{pmatrix}3\\5\\-1\end{pmatrix}=1\ 3+2\ 5+3\ (-1)=10$

Produit scalaire

On applique ce calcul lorsqu'on a besoin de déterminer l'angle des deux vecteurs, il suffit d'introduire la première égalité dans la seconde et de résoudre le tout en cosinus :

$$\cos\alpha = \frac{\vec{a}\ \vec{b}}{a\ b}$$

$$= \frac{a_1\ b_1 + a_2\ b_2 + a_3\ b_3}{a\ b}$$

Angle de deux vecteurs

La dernière forme de multiplication est appelée produit vectoriel. Elle associe à deux vecteurs un troisième vecteur perpendiculaire aux deux premiers, ce qui montre que cette opération n'a de sens que dans un espace à trois dimensions. La définition exacte est la suivante :

$$\vec{a}\quad \vec{b} = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} a_2\ b_3 - a_3\ b_2 \\ a_3\ b_1 - a_1\ b_3 \\ a_1\ b_2 - a_2\ b_1 \end{pmatrix}$$

$$\text{Ex.}: \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \begin{pmatrix} 3 \\ 5 \\ -1 \end{pmatrix} = \begin{pmatrix} 2\ (-1) - 3\ 5 \\ 3\ 3 - 1\ (-1) \\ 1\ 5 - 2\ 3 \end{pmatrix} = \begin{pmatrix} -17 \\ 10 \\ -1 \end{pmatrix}$$

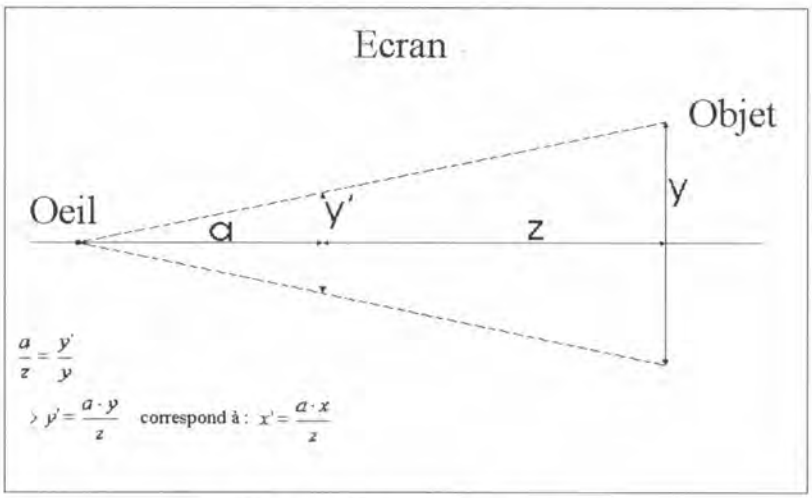
Produit vectoriel

8.2. Représentation d'objets 3D en 2D

Tant qu'il n'existe pas encore de périphérique de sortie tridimensionnel, tout calcul en 3D doit être représenté sur un écran plan. Il existe diverses méthodes de résolution de ce problème allant de la simple projection parallèle aux algorithmes complexes de raytracing. Cette dernière méthode est réservée aux "rendeurs" qui prennent tranquillement plusieurs minutes pour calculer une image.

Le moyen le plus simple de traiter les informations de profondeur impossibles à représenter sur l'écran consiste à les ignorer complètement. Les coordonnées bidimensionnelles des sommets de chaque forme sont tirées des coordonnées x et y de leur définition tridimensionnelle. Ainsi, les parallèles de l'espace apparaissent également parallèles sur l'écran : d'où le nom de projection parallèle.

Ce procédé présente certains avantages. Il est d'ailleurs utilisé en architecture. Mais il ne correspond pas à la perception de la réalité. Les objets éloignés produisent en effet une image plus petite sur la rétine de nos yeux. Cette propriété doit être simulée dans la reproduction de la réalité de façon que la profondeur ne soit plus simplement ignorée mais provoque le rapetissement de l'image. L'algorithme nécessaire peut être visualisé à l'aide du théorème des rayons lumineux :



Théorème des rayons lumineux

On contemple l'univers à trois dimensions comme s'il s'étendait derrière l'écran. De cet univers, on ne voit d'abord rien, seule sa projection sur l'écran

est visible. Tous les rayons lumineux qui partent de l'objet doivent finalement parvenir jusqu'à l'oeil de l'observateur pour que l'image soit perçue. Toutes les coordonnées de l'objet (ici par exemple les ordonnées des coins supérieurs et inférieurs) doivent être converties en coordonnées bidimensionnelles (ici y'). On utilise à cet effet le théorème des rayons lumineux qui relie de façon unique y, y', a (distance oeil-écran) et z (profondeur ou cote de l'objet).

On peut imaginer assez facilement que si la profondeur z augmente, l'angle des rayons diminue ainsi que la taille de l'image, de sorte qu'on obtient la perspective à point de fuite souhaitée. Dans cette méthode, le point de fuite où convergent toutes les parallèles à l'infini se trouve toujours sur l'axe z. Il n'est pas librement déplaçable, mais ce n'est pas nécessaire pour la plupart des applications.

L'exploitation de la formule mentionnée peut se faire en économisant beaucoup de temps de calcul si on prend comme distance a (par ailleurs arbitraire) une puissance de deux (par exemple 128). Dans ce cas, la multiplication peut se faire par décalage (instructions SHL/SHR).

8.3. Pour déformer les objets : les transformations

La programmation en trois dimensions porte rarement sur des objets immobiles. C'est le mouvement qui va justement donner l'illusion de la réalité. Un dé qui tourne avec des images sur ses faces est sûrement plus intéressant qu'une vue statique, au demeurant plus facile à mettre au point avec un logiciel de dessin.

Un mouvement se décompose toujours en une translation et une rotation. On peut également y inclure le cas échéant un changement d'échelle (ou homothétie).

On appelle translation un simple déplacement dans une direction fixe, par exemple lorsqu'on parcourt un long couloir. Mathématiquement parlant, la translation se traduit par une addition vectorielle. C'est le vecteur dit de translation qui la définit. Il est ajouté à tous les vecteurs allant de l'origine à l'objet à déplacer (coordonnées des points de l'objet).

On gère de la même façon une translation de l'observateur mais le point de vue est inversé. Pour se déplacer d'une unité dans la direction de l'axe z, il suffit de translater dans sa totalité l'univers 3D d'une unité dans le sens négatif.

La rotation est une opération un peu plus compliquée, surtout si on veut à tout prix impliquer des calculs matriciels comme c'est un peu la mode. Les matrices rassemblent dans une sorte de tableau les éléments de calcul nécessaires à une transformation : translation, rotation, homothétie, toutes ces opérations ont leur matrice associée. On peut ainsi regrouper plusieurs matrices et économiser du temps de calcul lorsqu'on connaît d'avance les transformations à effectuer et leur ordre de succession. Mais comme c'est rarement le cas, les matrices de rotation ne sont citées ici que par souci d'exhaustivité.

Dans une rotation, il faut bien distinguer lequel des trois axes est concerné, les équations ne sont pas les mêmes. Pour une rotation autour de l'axe x, on utilise les formules :

```
x'= x
y'= y*cos(a)-z*sin(a)
z'= y*sin(a)+z*cos(a)
```

Voici la matrice correspondante pour les intéressés :

```
1  0  0
0  cos(a)  -sin(a)
0  sin(a)   cos(a)
```

Autour de l'axe y, on écrira :

```
x'= x*cos(a)+z*sin(a)
 y'= y
 z'= -x*sin(a)+z*cos(a)
```

ce qui correspond à la matrice :

```
cos(a)  0   sin(a)
0   1   0
-sin(a) 0  cos(a)
```

Autour de l'axe z, les formules deviennent :

```
 x'= x*cos(a)-y*sin(a)
y'= x*sin(a)+y*cos(a)
z'= z
```

et la matrice :

```
cos(a) sin(a)  0
sin(a) cos(a)  0
0   0  1
```

Ces formules s'appliquent à des rotations autour des axes de coordonnées. Mais en combinant plusieurs de ces rotations de base, on peut obtenir toute rotation autour d'une droite passant par l'origine. Pour éliminer la contrainte de la droite passant par l'origine, on ajoute encore une translation.

Avant la rotation, on translate l'espace des objets de façon que sa nouvelle origine se confonde avec le centre de rotation. Après la rotation, on refait éventuellement la translation inverse, le vecteur de translation ayant lui-même subi la rotation.

Lorsque plusieurs rotations se succèdent autour de plusieurs axes, les coordonnées résultant de chaque rotation doivent servir de coordonnées sources pour la suivante. Si on raisonne toujours dans le même système de coordonnées de départ, on trouve des résultats aberrants qui ne correspondent à rien de réel.

Avec leurs cortèges de sinus et de cosinus, les calculs de rotation constituent un domaine d'application rêvé pour les consultations de tables. Si on utilisait les fonctions Pascal ordinaires pour les programmer, on n'arriverait jamais à représenter un mouvement fluide. Les sinus et cosinus seront donc tirés d'une table commune et multipliés par les coordonnées en fonction des exigences du calcul.

En principe, à l'exception des pures translations, les transformations citées ne sont pas commutatives. Leur ordre de succession n'est pas indifférent. Si on soumet par exemple le point (1,0,0) qui se trouve sur l'axe des x à une première rotation de 90° autour de l'axe des x, puis à une seconde rotation identique autour de l'axe des z, le résultat se situe sur l'axe des y. Si on suit l'ordre inverse, il se situe sur l'axe des z. Il est donc recommandé de fixer une fois pour toute l'ordre de définition des rotations, par exemple d'abord autour de l'axe des x, puis autour de l'axe des y et enfin autour de l'axe des z.

Lorsque les transformations sont mélangées, il faut aussi respecter l'ordre dans lequel elles se présentent. Une translation suivie d'une rotation ne donne pas le même résultat que la succession inverse, il est donc également recommandé de se fixer des règles. Le plus judicieux est ici de privilégier la succession translation-rotation, car la translation est alors définie par rapport aux coordonnées de base de l'espace et non pas par rapport à leur image après rotation.

En résumé on peut considérer que l'ordre suivant est le plus indiqué :

1. translation
2. rotation (x, y puis z)
3. projection sur l'écran.

8.4. Objets filiformes : modèles en fil de fer.

La façon la plus simple d'obtenir à l'écran des objets tridimensionnels est d'utiliser des modèles en fil de fer. Seules les arêtes des objets sont dessinées et non leurs faces. L'objet est alors transparent et il n'est pas nécessaire de se livrer à des calculs de faces cachées. Ce modèle exploite le fait que la projection parallèle et la perspective à point de fuite conservent les droites et ne les transforment pas en courbes. Grâce à cette constatation, on peut se contenter de transformer les sommets des objets sans être obligé de traiter les points qui forment les arêtes. Si on relie les points calculés, on obtient une image réaliste de l'objet sous forme de modèle en fil de fer. Dans cette résolution, l'algorithme de tracé de segment joue un rôle prépondérant. C'est de lui que dépend la vitesse d'affichage car les transformations en elles-mêmes nécessitent peu de temps de calcul. Pour tracer un segment, l'algorithme de Bresenham est actuellement l'un des plus efficaces, c'est pourquoi nous l'appliquerons dans le présent chapitre.

La justification mathématique de cet algorithme se trouve dans les ouvrages consacrés aux graphiques ou à la microinformatique en général et parfois aussi dans les revues spécialisées. Nous ne l'exposerons donc pas ici, nous contentant d'en appliquer le principe.

L'algorithme en question se limite d'abord aux segments de pente comprise entre 0 et 1 (inclinaison de 0 à 45 °). Pendant le tracé de la ligne, il faut décider pour chaque point si on le place immédiatement à droite du précédent ou à droite au-dessus. Tout autre placement est interdit. Pour choisir le point le plus proche, on applique une méthode proche des calculs en virgule fixe. Une variable Dist (stockée en BP) est incrémentée de Add_1 (en SI) ou de Add_2 (en DI) en fonction du dernier placement (immédiatement à droite ou en haut à droite). Au point suivant, la décision est fonction du signe de Dist.

La limite imposée par l'intervalle de la pente qui doit être comprise entre 0 et 1 peut être facilement levée. Les pentes entre 1 et l'infini (45 à 90 °) s'obtiennent par permutation de x et y et les pentes négatives par inversion de la direction de traitement.

Le détail des opérations apparaît dans le listing ci-dessous de BRES.ASM :

```
b equ byte ptr
w equ word ptr
data segment
  extrn vpage:word
data ends
putpixel macro
  pusha
  xchg ax,bx                    ;échange x et y
  push ax                       ;sauve y pour usage ultérieur
  mov cx,bx                     ;lit x
  and cx,3                      ;masque le plan
  mov ax,1                      ;et met à 1 le bit correspondant
  shl ax,cl
  mov ah,2                      ;registre 2 du TS
  xchg ah,al
  mov dx,3c4h
  out dx,ax
  pop cx                        ;lit y
  mov ax,80d                    ;calcule l'offset de ligne
  mul cx
  shr bx,2                      ;ajoute l'offset de colonne
  add bx,ax
  add bx,vpage                  ;écrit dans la page courante
  mov b es:[bx],3               ;et fixe la couleur
  popa
endm
code segment public
assume cs:code,ds:data
public bline
bline proc    near
  push bp
  push ax                       ;sauvegarde x0
  push bx                       ;et y0
  mov bx,4340h                  ;prépare l'automodification
  sub cx,ax                     ;évalue deltax
  jns deltax_ok                 ;négatif ?
  neg cx                        ;oui, change le signe de deltax
```

```
  mov bl,48h                ;et dec ax au lieu de inc ax
deltax_ok:
  mov bp,sp                 ;adressage de y1 sur la pile
  sub dx,ss:[bp]            ;calcule deltay
  jns deltay_ok             ;négatif ?
  neg dx                    ;oui, change le signe de deltay
  mov bh,4bh                ;et dec bx au lieu de inc bx
deltay_ok:
  mov si,dx                 ;deltay et
  or si,cx                  ;deltax = 0 ?
  jne ok
  add sp,6                  ;alors enlève ax, bx et bp
                            ;de la pile et fin
  ret
ok:
  mov w cs:dist_pos,bx      ;écrit dec/inc ax/bx
  cmp cx,dx                 ;deltax >= deltay ?
  jge deltax_grand
  xchg cx,dx                ;non, alors échange deltax et deltay
  mov bl,90h                ;et change inc ax en nop
  jmp constantes
deltax_grand:
  mov bh,90h                ;sinon change inc bx en nop
constantes:
  mov w cs:dist_neg,bx      ;écrit dec/inc ax/bx
  shl dx,1                  ;définit Add_2
  mov di,dx                 ;sauve en di
  sub dx,cx                 ;définit Start-Dist
  mov bp,dx                 ;et sauve en bp
  mov si,bp                 ;définit Add_1
  sub si,cx                 ;et sauve en si
  mov ax,0a000h             ;charge le segment VGA
  mov es,ax
  pop bx                    ;reprend les valeurs de x0 et y0
  pop ax
loop_p:
  putpixel                  ;dessine un point
  or bp,bp                  ;Dist positif ?
```

```
  jns dist_pos
dist_neg:
  inc ax                          ;x continue  (automodification
                                  ;éventuelle)
  inc bx                          ;y continue (automodification
                                  ;éventuelle)
  add bp,di                       ;met à jour Dist
  loop loop_p                     ;point suivant
  jmp fini                        ;c'est terminé
dist_pos:
  inc ax                          ;x continue (automodification
                                  ;éventuelle)
  inc bx                          ;y continue (automodification
                                  ;éventuelle)
  add bp,si                       ;met à jour Dist
  loop loop_p                     ;point suivant
fini:
  pop bp
  ret
bline endp
code ends
end
```

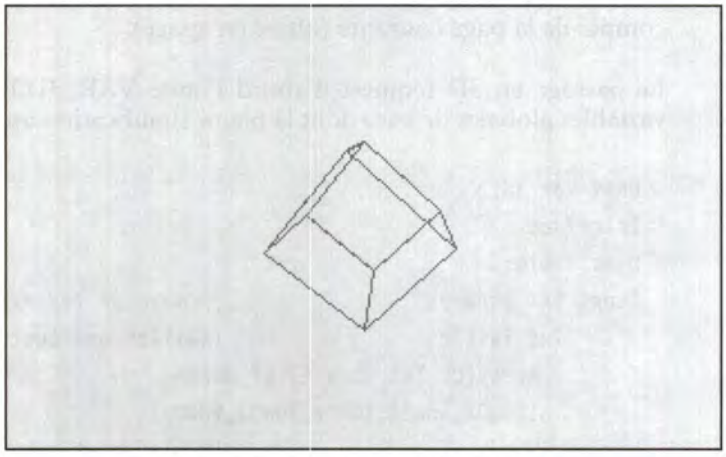
Il est important de bien prendre note des arguments de cette procédure. Elle trace en mode X un segment allant du point (ax,bx) au point (cx,dx) en tenant compte de la page courante (offset en vpage).

Le passage en 3D requiert d'abord l'unité VAR_3D.PAS qui contient des variables globales de base dont la pleine signification apparaîtra par la suite :

```
Unit Var_3d;
Interface
Uses Tools;
Const Txt_Nbre=5;                 {nombre de textures utilisées }
      Txt_Taille:                 {tailles des textures }
        Array[0..Txt_Nbre-1] of Word=
        ($0a0a,$0a0a,$0a0a,$0a0a,$0a0a);
Var  vz:Word;                     {déplacement dans l'écran }
    rotx,                         {angle de rotation }
    roty,
```

```
    rotz:word;                    {rotation en unités de 3 degrés}
    su_sort:Boolean;              {tri des surfaces ?}
    Remplir:Boolean;              {true: remplir / false: lignes}
    su_cacher:Boolean;            {traitement des faces cachées ?}
    Texture:Boolean;              {utiliser des textures ?}
    lightsrc:Boolean;             {utiliser une source de lumière ?}
    Verre:Boolean;                {surfaces en verre ?}
    Txt_Data:Array[0..Txt_Nbre-1] of Pointer;
                                  {situation des textures en mémoire }
    Txt_Offs:Array[0..Txt_Nbre-1] of Word;
                                  {offset à l'intérieur de l'image de
                                   la texture }
    Txt_Pic:Pointer;              {pointeur sur l'image de la texture}
    Sinus:Array[0..149] of Word; {table de sinus pour les rotations }
Implementation
Begin
  Sin_Gen(Sinus,120,16384,0);
  Move(Sinus[0],Sinus[120],60);
End
```

Cette unité est exploitée par tous les programmes 3D. Elle initialise d'abord la table des sinus utilisée par les rotations. Ensuite, le premier quart de la table (30 entrées = 60 octets) est ajouté à sa suite : elle pourra donc servir à lire tant les sinus (origine en sin[0]) que les cosinus (valeur nulle en sin[30]).



Rotation d'un graphique 3D en fil de fer

Le programme 3D_FIL.PAS utilise l'algorithme de Bresenham. En mettant à zéro la variable globale Remplir, il signale au module en assembleur qu'il faut tracer des modèles en fil de fer.

```
Uses Crt,ModeXLib,var_3d;
Const
    worldlen=8*3;                {tableau des points }
    Worldconst:Array[0..worldlen-1] of Integer =
    (-200,-200,-200,
    -200,-200,200,
    -200,200,-200,
    -200,200,200,
    200,-200,-200,
    200,-200,200,
    200,200,-200,
    200,200,200);
    surfclen=38;                 {tableau des faces }
    surfcconst:Array[0..surfclen-1] of Word=
    (0,4, 0,2,6,4,
     0,4, 0,1,3,2,
     0,4, 4,6,7,5,
     0,4, 1,5,7,3,
     0,4, 2,3,7,6,
     0,4, 0,4,5,1,0,0);
Var
    i,j:Word;

procedure drawworld;external;    {dessine l'espace des objets dans
                                  la page d'écran courante}
{$l 3dasm.obj}
{$l poly.obj}
{$l bres.obj}
{$l \bp\myunit\racine.obj}
Begin
  vz:=1000;                      {profondeur de l'objet }
  vpage:=0;                      {commence en page 0}
  init_modex;                    {active le mode X }
  rotx:=0;                       {rotation initiale }
  roty:=0;
```

```
  rotz:=0;
  remplir:=false;                {pas de remplissage }
  su_sort:=false;                {pas de tri des faces}
  su_cacher:=false;              {pas de traitement des faces cachées}
  verre:=false;                  {pas de face en verre }
  repeat
    clrx($0f);                   {efface l'écran }
    DrawWorld;                   {dessine l'espace des objets }
    switch;                      {active l'écran }
    WaitRetrace;                 {attend le prochain retour de
                                  balayage }
    Inc(rotx);                   {poursuit la rotation ... }
    If rotx=120 Then rotx:=0;
    Inc(rotz);
    If rotz=120 Then rotz:=0;
    inc(roty);
    if roty=120 Then roty:=0;
  Until KeyPressed;              { ...jusqu'à frappe de touche}
  TextMode(3);
End
```

Ce programme commence comme les suivants par la définition de l'espace de référence et des surfaces. L'espace de référence ne contient que les coordonnées des sommets qui le limitent, dans l'ordre x, y, z pour chaque point. Ces points ne sont reliés entre eux que par la définition des faces qui se présente sous forme de tableau.

Le premier mot de chaque définition de face contient la structure superficielle qui sera examinée plus loin et qui est initialisée à 0 ici. On indique ensuite le nombre de sommets de la face et leurs numéros en commençant le comptage à 0. Le premier carré est ainsi formé des points 0,2,6,4 qui correspondent aux coordonnées (-200,-200,-200), (-200,200,-200), (200,200,-200), (200,-200, 200). La liste de ces définitions se termine par deux zéros pour que la procédure d'affichage puisse y reconnaître.

Le programme principal fixe d'abord la profondeur totale vz de l'objet. Diminuer cette valeur revient à effectuer un zoom avant. Les variables rotx, roty et rotz donnent l'angle de rotation de l'objet en pas de 3 degrés. Ceci veut dire qu'une rotation de 15 degrés correspond à une valeur de 5.

Le mode graphique retenu est le mode X. S'il se montre lent dans l'adressage d'un pixel isolé, il présente l'avantage de gérer deux écrans. C'est au moment de l'exploitation des textures que cette caractéristique se révèlera importante

car, à ce moment-là, la mise à jour d'un écran sera plus longue qu'un retour de balayage vertical.

Le programme initialise ensuite les variables de commande globale. Il désactive ainsi le remplissage des faces (Remplir), le tri (su_sort) et le traitement des faces cachées (su_cacher) ainsi que l'effet de verre (verre). Toutes ces variables ne seront exploitées que dans les chapitres suivants.

La boucle qui vient ensuite est reprise de façon similaire dans tous les programmes. Tant qu'aucune touche n'a été frappée, l'écran s'efface, l'espace se dessine, une nouvelle page apparaît et la rotation se poursuit.

L'essentiel de ce programme est pris en charge par le module en assembleur 3DASM.ASM qui existe en tant que tel pour des impératifs de vitesse. Ne vous laissez pas impressionner par le code source qui paraît long et complexe. Les chapitres qui suivent en expliquent la teneur point par point. Mais comme il ne peut se découper en plusieurs parties, il est donné ci-après dans son intégralité :

```
w equ word ptr
b equ byte ptr
surfclen equ 200                ;longueur maximale de la
                                ;définition des faces
PointsT equ 4*100               ;taille du tableau des points
nb_su equ 30                    ;nombre maximal de faces
nb_som equ 10                   ;nombre maximal de sommets
data segment                    ;variables externes de la
                                ;partie Pascal
  extrn vz:word                 ;profondeur général
  extrn rotx:Word               ;angles de rotation
  extrn roty:Word
  extrn rotz:word
  extrn worldconst:dataptr      ;tableau des points
  extrn surfcconst:dataptr      ;tableau des définitions des faces
   extrn lightsrc:word          ;indicateur pour effet de lumière
   extrn su_sort:word           ;indicateur pour tri des faces
  extrn su_cacher:word          ;indicateur pour traitement
                                ;des faces cachées
  extrn Texture:Byte            ;indicateur pour textures
  extrn Remplir:Byte            ;indicateur pour remplissage/
                                ;fil de fer
crotx dw 0                      ;angle x, y et z comme offset
```

```
croty dw 0                          ;sur la valeur du sinus
crotz dw 0
rotx_x dw 0                         ;x,y,z après rotation x
rotx_y dw 0
rotx_z dw 0
roty_x dw 0                         ;après rotation y
roty_y dw 0
roty_z dw 0
rotz_x dw 0                         ;après rotation z, définitif
rotz_y dw 0
rotz_z dw 0
startpoly dw 0                      ;début de la déf de la face courante
Points      dw PointsT dup (0)      ;mémorise les coordonnées calculées
Pointsptr   dw 0                    ;pointe sur le tableau de points
Points3d    dw PointsT dup (0)      ;mémorise les coordonnées 3D
                                    ;(texture)
moyen       dw nb_su*2 dup (0)      ;répertoire des valeurs moyennes de z
moyenptr    dw 0                    ;pointeur dans le tableau "moyen"
n           dw 0,0,0,0,0,0          ;vecteur normal 32 bits
n_long      dw 0                    ;longueur du vecteur normal
extrn sinus:dataptr
data ends
extrn drawpol:near                  ;dessine une face en fil de fer
extrn fillpol:near                  ;remplit une face
extrn racine:near                   ;calcule la racine de ax
getdelta macro                      ;calcule les deux vecteurs
                                    ;compris dans le plan de la face
  mov ax,poly3d[0]                  ;x: sommet de départ
  mov delta2[0],ax                  ;sauvegarde temporaire en delta2
  sub ax,poly3d[8]                  ;différence avec le premier point
  mov delta1[0],ax                  ;delta1 terminé
  mov ax,poly3d[2]                  ;y: sommet de départ
  mov delta2[2],ax                  ;sauvegarde temporaire en delta2
  sub ax,poly3d[10d]                ;différence avec le premier point
  mov delta1[2],ax                  ;delta1 terminé
  mov ax,poly3d[4]                  ;z: sommet de départ
  mov delta2[4],ax                  ;sauvegarde temporaire en delta2
  sub ax,poly3d[12d]                ;différence avec le premier point
```

```
  mov delta1[4],ax              ;delta1 terminé
  mov bp,polyn                  ;sélectionne le dernier point
  dec bp
  shl bp,3                      ;8 octets à chaque fois
  mov ax,poly3d[bp]             ;lit x
  sub delta2[0],ax              ;forme la différence
  mov ax,poly3d[bp+2]           ;lit y
  sub delta2[2],ax              ;forme la différence
  mov ax,poly3d[bp+4]           ;lit z
  sub delta2[4],ax              ;forme la différence
endm
setcoord macro source,offst     ;fixe les coord d'écran calculées
.386
  mov ax,source                 ;projette les coordonnées
  cwd
  shld dx,ax,7
  shl ax,7
  idiv cx
   add ax,offst                 ;centre de l'écran = 0,0,0
  mov bx,Pointsptr              ;report dans le tableau des points
  mov Points[bx],ax
  add Pointsptr,2               ;déplace le pointeur sur le tableau
endm
z2cx macro tabofs               ;lit en cx la coordonnée z
  mov cx,tabofs + 4
  add cx,vz                     ;ajoute la translation z
  mov bx,moyenptr               ;report dans le tableau moyen
  add moyen[bx],cx
endm
xrot macro dcoord,scoord        ;rotation de scoord autour de x,
                                ;résultat en dcoord
.386
  mov bp,crotx                  ;charge l'angle
  mov bx,[scoord]
  shl bx,3                      ;x8, pour alignemt sur entrée
                                ;de points
  mov Pointsptr,bx
  sub bx,[scoord]               ;en tout x6, pour alignement
```

```
                                    ;sur entrées espace
  sub bx,[scoord]
  add bx,offset worldconst
  mov ax,[bx]                       ;lit x
  mov dcoord,ax                     ;qui reste inchangé
  mov ax,[bx+2]                     ;lit y
  imul w ds:[bp+60d]                ;*cos rotx
  shrd ax,dx,14d
  mov cx,ax                         ;mémorise en cx
  mov ax,[bx+4]                     ;lit z
  imul w ds:[bp]                    ;*-sin rotx
  shrd ax,dx,14d
  sub cx,ax
  mov dcoord+2,cx                   ;y terminé et stocké
  mov ax,[bx+2]                     ;lit y
  imul w ds:[bp]                    ;*sin rotx
  shrd ax,dx,14d
  mov cx,ax                         ;mémorise en cx
  mov ax,[bx+4]                     ;lit z
  imul w ds:[bp+60d]                ;*cos rotx
  shrd ax,dx,14d
  add cx,ax
  mov dcoord+4,cx
endm
yrot macro dcoord,scoord            ;rotation de scoord autour de y,
                                    ;résultat en dcoord
  mov bp,croty                      ;lit l'angle
  mov ax,scoord+2                   ;et y
  mov dcoord+2,ax                   ;y reste inchangé
  mov ax,scoord                     ;lit x
  imul w ds:[bp+60d]                ;*cos roty
  shrd ax,dx,14d
  mov cx,ax                         ;mémorise en cx
  mov ax,scoord+4                   ;lit z
  imul w ds:[bp]                    ;*sin roty
  shrd ax,dx,14d
  add cx,ax
  mov dcoord,cx                     ;x terminé et stocké
```

```
  mov ax,scoord                ;lit x
  imul w ds:[bp]               ;*-sin roty
  shrd ax,dx,14d
  mov cx,ax                    ;mémorise en cx
  mov ax,scoord+4              ;lit z
  imul w ds:[bp+60d]           ;*cos roty
  shrd ax,dx,14d
  sub ax,cx
  mov dcoord+4,ax
endm
zrot macro dcoord,scoord       ;rotation de scoord autour de z,
                               ;résultat en dcoord
  mov bx,Pointsptr             ;prépare le report dans le tableau
                               ;de points 3D
  mov bp,crotz                 ;lit l'angle
  mov ax,scoord+4              ;lit z
  mov dcoord+4,ax              ;et le laisse inchangé
  mov Points3d[bx+4],ax        ;report égalmt dans tableau 3D
  mov ax,scoord                ;lit x
  imul w ds:[bp+60d]           ;*cos rotz
  shrd ax,dx,14d
  mov cx,ax                    ;mémorise en cx
  mov ax,scoord+2              ;lit y
  imul w ds:[bp]               ;*-sin rotz
  shrd ax,dx,14d
  sub cx,ax
  mov dcoord,cx                ;x terminé et stocké
  mov Points3d[bx],cx
  mov ax,scoord                ;lit x
  imul w ds:[bp]               ;*sin rotz
  shrd ax,dx,14d
  mov cx,ax                    ;mémorise en cx
  mov ax,scoord+2              ;lit y
  imul w ds:[bp+60d]           ;*cos rotz
  shrd ax,dx,14d
  add cx,ax
  mov dcoord+2,cx
  mov Points3d[bx+2],cx
```

```
endm
get_normal macro                ;calcule le vecteur normal d'une face
  mov ax,delta1[2]              ;a2*b3
  imul delta2[4]
  shrd ax,dx,4
  mov n[0],ax
  mov ax,delta1[4]              ;a3*b2
  imul delta2[2]
  shrd ax,dx,4
  sub n[0],ax
  mov ax,delta1[4]              ;a3*b1
  imul delta2[0]
  shrd ax,dx,4
  mov n[2],ax
  mov ax,delta1[0]              ;a1*b3
  imul delta2[4]
  shrd ax,dx,4
  sub n[2],ax
  mov ax,delta1[0]              ;a1*b2
  imul delta2[2]
  shrd ax,dx,4
  mov n[4],ax
  mov ax,delta1[2]
  imul delta2[0]
  shrd ax,dx,4
  sub n[4],ax                   ;produit vectoriel
                                ;(=vecteur normal) calculé
  mov ax,n[0]                   ;x1 ^ 2
  imul ax
  mov bx,ax
  mov cx,dx
  mov ax,n[2]                   ;+x2 ^ 2
  imul ax
  add bx,ax
  adc cx,dx
  mov ax,n[4]                   ;+x3 ^ 2
  imul ax
  add ax,bx
```

```
  adc dx,cx                    ;somme en dx:ax
  push si
  call racine                  ;racine en ax
  pop si
  mov n_long,ax                ;longueur du vecteur normal calculée
endm
light macro                    ;détermine la luminosité d'une face
  mov ax,n[0]
  imul l[0]                    ;vecteur de la lumière*vecteur normal
  mov bx,ax                    ;somme en cx:bx
  mov cx,dx
  mov ax,n[2]
  imul l[2]
  add bx,ax
  adc cx,dx
  mov ax,n[4]
  imul l[4]
  add ax,bx                    ;produit scalaire en dx:ax
  adc dx,cx
  idiv l_long                  ;division par l_long
  mov bx,n_long                ;et par n_long
  cwd
  shld dx,ax,5                 ;valeurs de -32 à +32
  shl ax,5d
  mov bp,startpoly             ;prépare l'adressage de la
                               ;couleur de surface
  idiv bx                      ;division par numérateur
  inc ax
  or ax,ax
  js danslumiere               ;si cos A positif -> lumière partante
  xor ax,ax                    ;donc pas d'éclairage
danslumiere:
  sub b polycol,al             ;cos<0 -> addition à la
                               ;couleur de base
endm
code segment
assume cs:code,ds:data
public drawworld
```

```
public linecount
public polycol
public polyn
public poly2d
public poly3d
linecount dw 0
polycol dw 3                    ;couleur de face courante
polyn   dw 0                    ;nombre de sommets effectivement
                                ;présents
poly2d dw nb_som*4 dup (0)      ;sommets du polygone à desssiner
poly3d dw nb_som*4 dup (0)      ;sommets 3D
public Txt_Nr
Txt_Nr dw 0                     ;numéro de la texture courante
public delta1,delta2
delta1       dw 0,0,0           ;vecteurs de plans
delta2      dw 0,0,0
l            dw 11d,11d,11d     ;vecteur lumière
l_long      dw 19d              ;longueur du vecteur lumière
drawworld proc pascal           ;dessine un espace tridimensionnel
  push ds
  push es
  push bp
  lea si,surfcconst             ;faces adressées par si
  mov moyenptr,0                ;on commence par 0 dans le tableau
                                ;moyen
  mov ax,ds:[rotx]              ;lit l'angle,
  shl ax,1                      ;convertit en offset mémoire
  add ax,offset sinus
  mov crotx,ax                  ;et stocke dans variables auxiliaires
  mov ax,ds:[roty]              ;idem pour y
  shl ax,1
  add ax,offset sinus
  mov croty,ax
  mov ax,ds:[rotz]              ;et z
  shl ax,1
  add ax,offset sinus
  mov crotz,ax
npoly:                          ;boucle des polygones
```

```
  mov startpoly,si              ;sauve pour usage ultérieur
  add si,2                      ;passe par-dessus la couleur
  mov cx,[si]                   ;lit le nombre de sommets
  mov linecount,cx              ;charge le compteur
  inc cx                        ;à cause des faces fermées
  mov w polyn,cx                ;report dans le tableau des points
  add si,2                      ;on continue par les coordonnées
                                ;proprement dites
nline:
  xrot rotx_x,si                ;rotation des coordonnées autour de x
  yrot roty_x,rotx_x            ;autour de y
  zrot rotz_x,roty_x            ;et de z
  z2cx rotz_x                   ;lit début z
  setcoord rotz_x,160           ;enregistre les coordonnées
  setcoord rotz_y,100
  add si,2                      ;sommet suivant
  dec linecount                 ;diminue le compteur de ligne
  je polyok                     ;tout dessiné ? -> fini
  jmp nline                     ;sinon ligne suivante
polyok:
  mov bx,moyenptr               ;calcule la valeur moyenne:
  mov ax,moyen[bx]              ;lit la somme
  mov cx,polyn
  dec cx
  cwd
  div cx                        ;et la divise par le nombre
                                ;de sommets
  mov moyen[bx],ax              ;puis enregistre le résultat
  mov ax,startpoly              ;avec le "numéro" de la face
  mov moyen[bx+2],ax
  add moyenptr,4                ;on continue
  cmp w [si+2],0                ;tous les polygones traités ?
  je fini
  jmp npoly
fini:
  cmp b su_sort,0               ;trier les faces ?
  je no_quicksort
  call quicksort pascal,0,bx    ;trie le champ de 0 à la
```

```
                                ;position actuelle
no_quicksort:
  mov moyen[bx+4],0             ;terminaison
  mov ax,cs                     ;segment de destination
  mov es,ax
  xor bx,bx                     ;on commence par la première face
npoly_draw:
  lea di,poly2d                 ;destination: tableau Poly
  mov bp,moyen[bx+2]            ;lit pointeur sur couleur et points
                                ;de la face
  mov ax,ds:[bp]                ;lit et fixe la couleur
  mov polycol,ax
  mov texture,0                 ;a priori: pas de texture
  cmp ah,0ffh                   ;texture ?
  jne no_textur
  mov texture,1                 ;oui, on la fixe
  mov b txt_nr,al               ;prend note du n°
no_textur:
  mov b lightsrc,0              ;a priori: pas d'ombre
  cmp ah,0feh                   ;ombre ?
  jne no_sourcelum
  mov b lightsrc,1              ;oui, on la fixe
no_sourcelum:
  add bp,2                      ;se positionne sur le nombre
  mov cx,ds:[bp]                ;lit le nombre de sommets
  mov polyn,cx                  ;l'enregistre dans le tableau Poly
npoint:
  add bp,2
  mov si,ds:[bp]                ;pointe sur les points effectifs
  shl si,3                      ;3 entrées Word !
  add si,offset Points          ;et x,y du tableau des points
                                ;en coord Poly
  mov ax,[si+Points3d-Points]   ;lit 3d-x
  mov es:[di+poly3d-poly2d],ax  ;fixe 3d-x
  mov ax,[si+Points3d-Points+2] ;lit 3d-y
  mov es:[di+poly3d-poly2d+2],ax ;fixe 3d-y
  mov ax,[si+Points3d-Points+4] ;lit 3d-z
  mov es:[di+poly3d-poly2d+4],ax ;fixe 3d-z
```

```
  movsw                          ;fixe les coordonnées 2D
  movsw
  add di,4                       ;entrée suivante Poly2d
  dec cx                         ;tous les sommets ?
  jne npoint
  mov bp,polyn                   ;copie le premier sommet sur
                                 ;le dernier
  shl bp,3                       ;se positionne sur le premier point
  neg bp
  mov ax,es:[di+bp]              ;et effectue la copie
  mov es:[di],ax
  mov ax,es:[di+bp+2]
  mov es:[di+2],ax
  add di,poly3d-poly2d           ;idem pour les coordonnées 3d
  mov ax,es:[di+bp]              ;effectue la copie
  mov es:[di],ax
  mov ax,es:[di+bp+2]
  mov es:[di+2],ax
  mov ax,es:[di+bp+4]
  mov es:[di+4],ax
  cmp Remplir,1                  ;remplir la face ?
  jne lines
  getdelta                       ;oui alors on calcule Delta1 et 2
  cmp b lightsrc,0               ;source de lumière ?
  jne delombre
  jmp no_lumiere
delombre:                        ;oui
  push bx
  get_normal                     ;alos calcule le vecteur normal
  light                          ;et la luminosité
  pop bx
no_lumiere:
  inc polyn                      ;incrémente le nombre de sommets
  call fillpol                   ;dessine la face
next:
  add bx,4                       ;cap sur la face suivante
  cmp moyen[bx],0                ;est-ce la dernière ?
  je _npoly_draw                 ;non on continue
```

```
  jmp npoly_draw
lines:
  push bx
  call drawpol                    ;dessine le polygone
  pop bx
  jmp next
_npoly_draw:
  pop bp                          ;c'est terminé
  pop es
  pop ds
  ret
drawworld endp
public quicksort
quicksort proc pascal bas,haut:word
;trie le tableau moyen avec l'algorithme Quicksort
local cle:word
local gauche:word
  push bx
  mov bx,bas                      ;trouve le milieu
  add bx,haut
  shr bx,1
  and bx,not 3                    ;positionnement sur groupe de 4
  mov dx,moyen[bx]                ;lit la clé
  mov cle,dx
  mov  ax,bas                     ;initialise droite et gauche
                                  ;avec valeurs de base
  mov si,ax
  mov gauche,ax
  mov ax,haut
  mov di,ax
  mov dx,cle
gauche_pluspres:
  cmp moyen[si],dx                ;plus grand que clé -> on
                                  ;continue la recherche
  jbe a_gauche
  add si,4                        ;position suivante
  jmp gauche_pluspres             ;on la teste
a_gauche:
```

```
  cmp moyen[di],dx              ;plus petit que la clé->
                                ;on continue la recherche
  jae a_droite
  sub di,4                      ;position suivante
  jmp a_gauche                  ;on la teste
a_droite:
  cmp si,di                     ;gauche <= droite ?
  jg finbou                     ;non -> partie triée
  mov eax,dword ptr moyen[si]   ;échange les valeurs moyennes
                                ;et les positions
  xchg eax,dword ptr moyen[di]
  mov dword ptr moyen[si],eax
  add si,4                      ;avance le pointeur
  sub di,4
finbou:
  cmp si,di                     ;gauche > droite , alors on continue
  jle gauche_pluspres
  mov gauche,si                 ;sauve gauche à cause récursion
  cmp bas,di                    ;bas < droite -> partie gauche triée
  jge droite_fini
  call quicksort pascal,bas,di  ;diviser par deux récursivt,
                                ;poursuivre le tri
droite_fini:
  mov si,gauche                 ;haut > gauche -> partie droite triée
  cmp haut,si
  jle gauche_fini
  call quicksort pascal,si,haut ;diviser par deux récursivt,
                                ;poursuivre le tri
gauche_fini:
  pop bx
  ret
quicksort endp
code ends
end
```

La signification centrale de la procédure Drawworld se devine déjà à sa longueur. Il s'agit en fait de dessiner l'espace tridimensionnel en fonction des valeurs courantes des transformations opérées. Aucun argument n'est transmis : la commande se fait par les variables globales.

Tout à fait au début, les angles de rotation courants (rotx, roty, rotz) sont convertis en offset par rapport au segment de données (crotx, croty, crotz), de façon que l'accès y soit accéléré ultérieurement. Dans la boucle npoly qui est parcourue pour chaque polygone, le registre si sert à effectuer l'adressage à l'intérieur du tableau des surfaces (surfcconst).

Les informations de couleur sont provisoirement ignorées, et le nombre de sommets du polygone est directement lu dans le tableau. Ce nombre est stocké dans le compteur LineCount qui donne par la suite le nombre de sommets restant à calculer.

La procédure entre alors dans la boucle des points nline qui effectue la rotation des points et les projette sur l'écran. Chaque rotation (macros xrot, yrot, zrot) transmet son résultat à son successeur, de sorte que les coordonnées d'origine tournent successivement autour des axes x y et z.

Les structures des trois macros de rotation sont très similaires : bp contient l'angle sous forme d'offset, le sinus correspondant peut alors être lu directement dans la table avec l'indice bp. Le cosinus est évalué par addition de 60 (octets, donc 30 éléments) à bp.

Pour pouvoir accéder plus tard au tableau tridimensionnel de points (avec des éléments à 8 octets), le numéro du sommet courant doit être d'abord multiplié par 8. Cet indice est mémorisé dans la variable PointsPtr. Après une double soustraction, rien ne s'oppose plus à un adressage des 6 octets de chaque sommet.

La suite est pratiquement la même pour toutes les macros : pour chacune des trois coordonnées, on calcule une en fonction de la matrice de rotation correspondante. Ainsi, dans la rotation x, l'abscisse x peut être reprise telle quelle, mais l'ordonnée y et la cote z seront déterminées par la formule bien connue : y'=y*cosA - z*sinA et z'=y*sinA + z*cosA.

Les deux autres macros sont les mêmes avec une autre matrice, seule la rotation z reporte en plus les coordonnées dans le tableau, qui ne joue encore aucun rôle à cet endroit.

La mission de la macro z2cx est suggérée par son nom : elle charge en cx la cote z en fin de rotation, après prise en compte de la profondeur totale vz.

Les coordonnées du sommet traité sont alors disponibles sous leur forme tridimensionnelle définitive : le but est de les ramener à leur projection bidimensionnelle sur l'écran. C'est la macro SetCoord qui s'en charge. Elle projette les coordonnées sources (une fois rotz_x et une fois rotz_y) et déplace en même temps l'image terminée au milieu de l'écran, en ajoutant l'offset transmis comme deuxième argument.

La multiplication par la distance de l'observateur A est obtenue par décalage à gauche de 7 bits, ce qui équivaut à une multiplication par 128. A cet endroit, les 16 bits du registre ax se font déjà trop étroits pour les coordonnées importantes, de sorte qu'on fait appel à l'association dx:ax. Après division par la cote (mémorisée en cx grâce à z2cx), l'offset est additionné et la coordonnée finale 2D stockée dans le tableau des points à l'emplacement correspondant au numéro du sommet. Après chaque enregistrement, le pointeur associé à ce tableau est incrémenté de deux octets, pour pouvoir avec la même macro se charger de l'ordonnée y après l'abscisse x.

Après ces macros, la boucle nline se termine par l'adressage du sommet suivant (incrémentation de si) et la décrémentation du compteur de lignes.

Une fois tous les sommets calculés, l'exécution se poursuit à l'étiquette polyok par l'évaluation de la profondeur moyenne du polygone. Nous reparlerons de ce point plus tard. Pour l'instant, il est important de noter que dans le tableau moyen un mot sur deux permet d'identifier le polygone associé grâce à son adresse de début (stockée en Startpoly). Pour les modèles en fil de fer, les choses redeviennent intéressantes à la fin de la boucle npoly du fait de l'instruction cmp. Si le nombre de sommets est nul, la boucle est interrompue et le contrôle passe à l'étiquette fini.

Plus tard, lorsque nous aborderons des modèles plus évolués, nous effectuerons à cet endroit un tri des surfaces. Pour le moment, cette partie de programme est court-circuitée en raison de la valeur de la variable su_sort. L'exécution se poursuit à l'étiquette no_quicksort. Un zéro de clôture est ajouté au tableau moyen puis chaque polygone est dessiné individuellement dans la boucle npoly_draw. Un autre tableau nommé Poly2d sert à transmettre les coordonnées à la routine DrawPoly qui prend en charge le tracé proprement dit.

A chaque fois, l'identification du polygone est tirée du tableau moyen, puis la couleur est lue dans le champ surfcconst et sauvegardée en polycol.

Les deux blocs d'instruction qui suivent servent à activer les routines de texture et de lumière, laissées de côté pour l'instant. Pour nous, l'étiquette no_lumiere conduit à lire dans le tableau surfcconst le nombre de sommets sauvegardé en polyn. La boucle npoint qui est parcourue pour l'ensemble des sommets lit le numéro de chacun d'eux et retrouve ainsi leurs coordonnées bidimensionnelles dans le tableau des points. Les deux instructions movsw copient ces coordonnées dans le champ Poly2D, dont le pointeur di est augmenté avant la fin de la boucle.

Il faut encore copier le premier sommet dans le dernier pour obtenir un tracé fermé. On utilise à cet effet une astuce : si on multiplie le nombre de sommets (polyn) par 8 (nombre d'octets par sommet), on obtient la longueur totale du

tableau Poly2D en octets. Cette taille est retranchée de la position courante (de là neg bp) et le mot y est recopié à l'intérieur.

Comme ce programme exploite la modélisation en fil de fer (remplir=false=0), l'exécution passe à l'étiquette lines qui appelle simplement la procédure DrawPoly du module POLY.ASM et dessine à l'écran le polygone sous forme de fil de fer. Ensuite, l'étiquette next se préoccupe du polygone suivant et nous sommes ramenés au début de la boucle npoly_Draw.

Le dessin du polygone est exécuté dans le module POLY.ASM, dans notre cas par la procédure DrawPoly :

```
public drawpol
;dessine une représentation en fil de fer de la surface en Poly2d
drawpol proc near
  push es
  pusha
  xor si,si                    ;indice sur la première entrée
  mov bp,polyn                 ;lit le nombre de sommets
@nline:
  mov ax,poly2d[si]            ;lit les coordonnées dans la table
  mov bx,poly2d[si+2]
  mov cx,poly2d[si+8]
  mov dx,poly2d[si+10d]
  push bp
  push si
  call bline                   ;trace une arête
  pop si
  pop bp
  add si,8                     ;arête suivante
  dec bp                       ;décrémente le nombre d'arêtes
  jne @nline
  popa
  pop es
  ret
drawpol endp
```

Ici, la représentation est très simple. On lit les coordonnées de début et de fin de chaque arête dans le tableau Poly2D et on appelle la procédure bline. Le registre si sert d'indice dans le tableau et bp, détourné de sa vocation, devient un compteur du nombre de segments à tracer.

8.5. Pour plus de transparence : les objets en verre

Les objets présentés jusqu'ici ne se composent que d'arêtes : ils ont peu de rapport avec la réalité observée. L'étape suivante va donc consister à munir les objets de faces solides. Ceci va nous entraîner dans l'un des domaines les plus difficiles de la géométrie 3D : la gestion des faces cachées.

Si on dessine les faces dans l'ordre où elles sont définies, certaines d'entre elles deviennent apparentes alors qu'elles ne le devraient pas. Nous allons nous occuper dans un instant de leur dissimulation, pour le moment nous nous livrerons à une autre réflexion.

Au lieu d'adapter l'image à la réalité, nous allons d'abord adapter la réalité à sa représentation. Un objet en verre n'a que des faces visibles forcément colorées, sinon on ne les verrait pas.

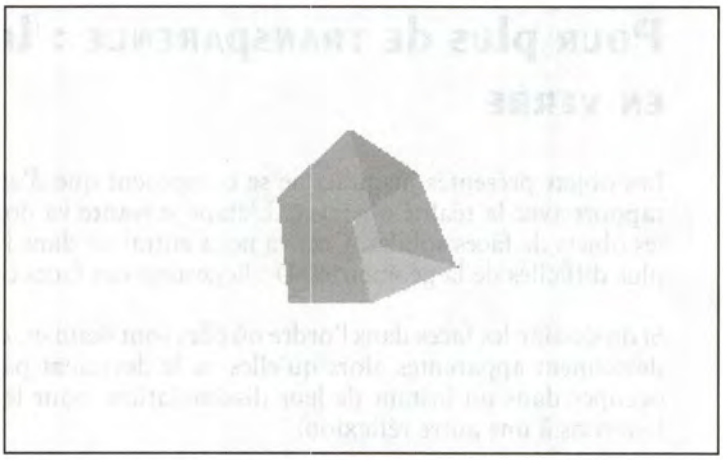
Si deux faces sont superposées, leurs couleurs se mélangent et donnent un résultat plus sombre (deux faces filtrent davantage la lumière qu'une seule).

Il faut en principe tenir compte de toutes les combinaisons possibles de faces : si une face A est susceptible de recouvrir une face B sous un angle quelconque, il faut prévoir le mélange des couleurs correspondantes. La seule possibilité d'y parvenir est de réserver pour chaque face un bit dans l'information de couleur. Seules les faces qui ne peuvent en aucun cas se recouvrir peuvent utiliser le même bit, sinon le mélange n'est pas possible.

Si, dans la représentation de la face, l'ancienne couleur n'est pas écrasée mais combinée à l'autre par l'opérateur OR, une nouvelle couleur apparaît. Par exemple, si la face A a la couleur 2 (bit 1 à 1) et la face B la couleur 16 (bit 4), le résultat de la combinaison est la couleur 18 (bits 1 et 4 à 1).

Bien entendu, la palette doit être en harmonie avec cette structure particulière. Elle doit d'une part contenir les couleurs pures, d'autre part prévoir un mélange pour chaque combinaison de bits. La couleur 18 évoquée plus haut doit représenter un mélange des couleurs 2 et 16.

Au début du programme, la palette doit être initialisée en conséquence : au remplissage du polygone, on exploite l'unité arithmétique intégrée dans la carte VGA. On peut en effet réaliser des opérations OR avec le registre 3 du GDC : les données en provenance du processeur sont combinées avec celles qui se trouvent dans les bascules avant d'être écrites dans la mémoire d'écran. Avant l'accès en écriture, il suffit de mettre dans les bascules les valeurs présentes dans la mémoire d'écran en exécutant un accès en lecture.



Le cube a maintenant des surfaces de couleurs transparentes

Le programme suivant appelé 3D_VERRE.PAS repose aussi sur le module 3DASM.ASM mais, par rapport au chapitre précédent, les variables sont différentes.

```
Uses Crt,ModeXLib,var_3d;
Const
    worldlen=8*3;                    {tableau des points }
    Worldconst:Array[0..worldlen-1] of Integer =
    (-200,-200,-200,
    -200,-200,200,
    -200,200,-200,
    -200,200,200,
    200,-200,-200,
    200,-200,200,
    200,200,-200,
    200,200,200);
    surfclen=38;                     {tableau des faces }
    surfcconst:Array[0..surfclen-1] of Word=
    (01,4, 0,2,6,4,
     02,4, 0,1,3,2,
     04,4, 4,6,7,5,
     08,4, 1,5,7,3,
     16,4, 2,3,7,6,
     32,4, 0,4,5,1,0,0);
Var
```

```
    i,j:Word;

Procedure Verre_Pal;
{prépare la palette pour la représentation d'objets en verre }
Begin
  FillChar(Palette[3],765,63);   {d'abord tout en blanc }
  For i:=1 to 255 do Begin       {définit 255 mélanges}
    If i and 1 = 1 Then Dec(Palette[i*3],16);
    If i and 2 = 2 Then Dec(Palette[i*3+1],16);
    If i and 4 = 4 Then Dec(Palette[i*3+2],16);
    If i and 8 = 8 Then Begin
                          Dec(Palette[i*3],16);
                          Dec(Palette[i*3+1],16);
                        End;
    If i and 16 = 16 Then Begin
                          Dec(Palette[i*3],16);
                          Dec(Palette[i*3+2],16);
                        End;
    If i and 32 = 32 Then Begin
                          Dec(Palette[i*3+1],16);
                          Dec(Palette[i*3+2],16);
                        End;
  End;
  SetPal;
End;

procedure drawworld;external;
{représente l'espace des objets dans la page d'écran courante }
{$l 3dasm.obj}
{$l poly.obj}
{$l bres.obj}
{$l racine.obj}
Begin
  vz:=1000;                      {profondeur de l'objet }
  vpage:=0;                      {commence par la page 0}
  init_modex;                    {active le mode X }
  Verre_Pal;
```

```
  rotx:=0;                          {valeurs initiales pour la
                                     rotation }
  roty:=0;
  rotz:=0;
  remplir:=true;                    {active le remplissage des faces }
  su_sort:=false;                   {pas de tri des surfaces }
  su_cacher:=false;                 {pas de traitement des faces cachées}
  verre:=true;                      {objet en verre }
  repeat
    clrx($0f);                      {efface l'écran }
    DrawWorld;                      {dessine l'espace }
    switch;                         {active l'écran préparé }
    WaitRetrace;                    {attend le prochain retour de
                                     balayage }
    Inc(rotx);                      {poursuit la rotation ... }
    If rotx=120 Then rotx:=0;
    Inc(rotz);
    If rotz=120 Then rotz:=0;
    inc(roty);
    if roty=120 Then roty:=0;
  Until KeyPressed;                 {... jusqu'à ce qu'on frappe
                                     une touche }
  TextMode(3);
End.
```

L'algorithme de remplissage des surfaces est activé par la variable Remplir. Par ailleurs, on donne à Verre la valeur TRUE, ce qui enclenche le mode OR du GDC. Avant le programme principal, qui correspond grosso modo au précédent, on appelle la procédure Verre_Pal qui prépare la palette à la représentation du verre.

Chaque couleur dont le bit 0 est à 1 voit sa proportion de rouge diminuée de 16. Ainsi, les mélanges de couleurs auxquelles participe la couleur 1 sont assombries dans le domaine du rouge, ce qui correspond à une filtration partielle du rouge, effectivement observable dans la réalité. On procède de même avec les 5 autres bits, pour filtrer d'autres couleurs.

Dans le module 3DASM.ASM, l'exécution suit pratiquement le même cours que pour la représentation en fil de fer. Ce n'est que tout à la fin que la procédure DrawWorld, à cause de la variable Remplir égale à true, se rend à l'étiquette no_lumiere. A cet endroit, la procédure FillPol se substitue à DrawPol et la boucle décrite précédemment achevée.

Le reste du module POLY.ASM entre alors en action. Après un algorithme très rapide de remplissage de polygone, une face colorée est générée avec les coordonnées indiquées en Poly2D.

Il existe fondamentalement deux catégories d'algorithmes de remplissage : les uns remplissent des surfaces dessinées d'avance et se rencontrent souvent dans les logiciels de dessin, les autres s'appliquent à des polygones définis par leurs coordonnées. Ce dernier est indiscutablement le meilleur pour nous en raison de sa rapidité.

La méthode employée ici repose sur le tracé de segments. En partant du point d'ordonnée minimale, on teste si on a atteint le bord droit ou gauche du polygone et on remonte jusqu'au point d'ordonnée maximale. Les lignes de délimitation du polygone sont calculées sans être dessinées. Si, des deux côtés, on a avancé d'une ligne, on peut tracer un segment horizontal entre les points calculés à gauche et à droite. On exploite ainsi l'excellente vitesse de tracé horizontal du mode X.

Ainsi comprise, une routine de remplissage doit constamment calculer des segments formant les bords droit et gauche du polygone. Lorsqu'un segment est en fin d'évaluation, on passe au segment suivant dont le point de départ correspond au dernier sommet.

La procédure FillPol montre comment mettre en application cette présentation théorique. En plus de la procédure DrawPol déjà évoquée, le code complet du module est également utilisé par l'algorithme de remplissage.

```
w equ word ptr
b equ byte ptr
include texture.inc              ;inclut les macros de texture
setnewlinel macro                ;n'utiliser ici que ax et bx!
local dylpos,dxlpos,dxlgrand,macro_fini
  mov bx,4043h                   ;code pour inc ax (en bh) et
                                 ;inc bx (en bl)
  mov bp,gauche
  mov ax,poly2d[bp+8]            ;sauvegarde les coordonnées de
                                 ;destination
  mov xl1,ax
  mov ax,poly2d[bp+10d]
  mov yl1,ax
  mov ax,poly2d[bp]              ;x,y début gauche dans var glob
  mov xl0,ax
  sub ax,xl1                     ;forme delta x
```

```
  inc xl1                    ;pour la condition d'arrêt
  neg ax                     ;xl1-xl0
  jns dxlpos                 ;dxl négatif ?
  neg ax                     ;alors valeur absolue
  mov bh,48h                 ;code pour dec ax (dec xl0)
  sub xl1,2                  ;extension négative des
                             ;coordonnées de destination
dxlpos:
  mov dxl,ax                 ;sauvegarde glob.
  mov incflagl,ax            ;mémorise dans l'indicateur
                             ;d'incrémentation
  mov ax,poly2d[bp+2]
  mov yl0,ax
  sub ax,yl1                 ;forme |delta y|
  inc yl1                    ;pour la condition d'arrêt
  neg ax
  jns dylpos                 ;négatif ?
  neg ax                     ;alors valeur absolue
  mov bl,4bh                 ;code pour dec bx (dec yl1)
  sub yl1,2                  ;extension négative des
                             ;coordonnées de destination
dylpos:
  mov dyl,ax                 ;sauvegarde glob.
  cmp dxl,ax                 ;dx < dy
  jae dxlgrand
  neg incflagl               ;changement de signe pour
                             ;l'indicateur d'incrémentation
dxlgrand:
  mov cs:byte ptr incxl,bh   ;effectuer l'automodification
  mov cs:byte ptr incyl,bl
  cmp texture,1              ;besoin de texture  ?
  jne macro_fini             ;non, on passe
  txt_makevarl               ;sinon calcule les variables
                             ;de texture
macro_fini:
  mov ax,xl0
  mov bx,yl0
  mov si,incflagl
```

```
endm
setnewliner macro                ;n'utiliser ici que cx et dx!
local dyrpos,dxrpos,dxrgrand,macro_fini
  mov cx,4142h                   ;code pour inc cx (en ch) et
                                 ;inc dx (en cl)
  mov bp,droite
  mov dx,poly2d[bp]              ;lit les coordonnées de destination
  mov xr1,dx
  mov dx,poly2d[bp+2]
  mov yr1,dx
  mov dx,poly2d[bp+8]            ;x,y à droite dans var glob.
  mov xr0,dx
  sub dx,xr1                     ;forme |delta x|
  inc xr1                        ;pour la condition d'arrêt
  neg dx
  jns dxrpos                     ;négatif ?
  neg dx                         ;opposé
  mov ch,49h                     ;code pour dec cx
  sub xr1,2                      ;extension négative des
                                 ;coordonnées de destination
dxrpos:
  mov dxr,dx                     ;sauvegarde en var glob
  mov incflagr,dx
  mov dx,poly2d[bp+10d]          ;forme |delta y|
  mov yr0,dx
  sub dx,yr1
  inc yr1                        ;pour la condition d'arrêt
  neg dx
  jns dyrpos                     ;négatif ?
  neg dx                         ;opposé
  mov cl,4ah                     ;code pour dec dx
  sub yr1,2                      ;extension négative des
                                 ;coordonnnées de destination
dyrpos:
  mov dyr,dx                     ;sauvegarde en var glob
  cmp dxr,dx                     ;dx < dy ?
  jae dxrgrand
```

```
  neg incflagr                ;alors changement de signe pour
                              ;l'indicateur d'incrémentation
dxrgrand:
  mov cs:byte ptr incxr,ch    ;automodification
  mov cs:byte ptr incyr,cl
  cmp texture,1               ;besoin de texture ?
  jne macro_fini              ;non, on passe
  txt_makevarr                ;sinon calcule variables de texture
macro_fini:
  mov cx,xr0                  ;charge les registres
  mov dx,yr0
  mov di,incflagr
endm
data segment public
  extrn vpage:word            ;page d'écran courante
  extrn su_cacher             ;indicateur pour dessin des
                              ;faces cachées
  extrn verre:Byte;           ;indicateur pour surfaces en verre
;variables de texture :
  extrn Texture:Byte          ;besoin de texture ?
  extrn Txt_Data:DataPtr      ;tableau de pointeurs sur
                              ;données graphiques
  extrn Txt_Offs:DataPtr      ;tableau d'offsets à
                              ;l'intérieur de l'image de texture
  extrn Txt_Taille:DataPtr    ;tableau avec indications de
                              ;dimensions
d_x  dd 0                     ;abscisse relative
d_y dd 0                      ;ordonnée relative
D    dd 0                     ;déterminant principal
Colonne1 dd 0                 ;composants du déterminant principal
       dd 0
Colonne2 dd 0
       dd 0
ligne_sup dw 0                ;quelles sont les coordonnées
                              ;utilisées?
Ligne_inf dw 0
xl_3d dd 0                    ;valeurs courantes des coordonnées
                              ;3d pendant le remplissage
```

```
yl_3d dd 0
zl_3d dd 0
xr_3d dd 0
yr_3d dd 0
zr_3d dd 0
inc_xl dd 0                    ;valeurs à ajouter aux
                               ;coordonnées courantes
inc_yl dd 0
inc_zl dd 0
inc_xr dd 0
inc_yr dd 0
inc_zr dd 0                    ;variables pour l'algorithme
                               ;de remplissage
point_haut   dw 0              ;maintenu en dx pendant la recherche
y_haut      dw 0               ;maintenu en bx
gauche       dw 0              ;point du côté gauche
droite      dw 0               ;point du côté droit
xl0  dw 0                      ;valeurs courantes des points
                               ;de début et de fin à gauche
yl0 dw 0
xl1 dw 0
yl1 dw 0
xr0 dw 0                       ;idem à droite
yr0 dw 0
xr1 dw 0
yr1 dw 0
dxl dw 0                       ;Delta X , Y pour les deux côtés
dyl dw 0
dxr dw 0
dyr dw 0
incflagl dw 0                  ;indicateurs pour savoir quand
                               ;y doit être incrémenté
incflagr dw 0                  ;sorte de "pente"
data ends
code segment public
assume cs:code,ds:data
extrn polycol:word             ;couleur de surface
extrn polyn:word               ;nombre de sommets
```

```
extrn poly2d:word              ;tableau de coordonnées 2D
extrn poly3d:word              ;tableau de coordonnées 3-D
extrn delta1,delta2:word       ;vecteurs de plan
extrn bline:near               ;dessine un segment
lambda1 dd 0                   ;coordonnées affines
lambda2 dd 0
inc_lambda1 dd 0               ;pas d'incrémentation
inc_lambda2 dd 0
plane dw 0002h                 ;plan courant à fixer
x0 dw 0                        ;coordonnées de segment
y0 dw 0
x1 dw 0
zz dw 0                        ;points restant à tracer
extrn Txt_Nr:Word              ;numéro de la texture à dessiner
public drawpol                 ;dessine un modèle en fil de
                               ;fer des faces en Poly2d
drawpol proc near
  push es
  pusha
  xor si,si                    ;indice sur première entrée
  mov bp,polyn                 ;lit le nombre de sommets
@nline:
  mov ax,poly2d[si]            ;lit les coordonnées dans la table
  mov bx,poly2d[si+2]
  mov cx,poly2d[si+8]
  mov dx,poly2d[si+10d]
  push bp
  push si
  call bline                   ;dessine un segment
  pop si
  pop bp
  add si,8                     ;segment suivant
  dec bp                       ;décrémente le nombre
  jne @nline
  popa
  pop es
  ret
drawpol endp
```

```
hline proc near                 ;trace un segment horizontal
                                ;ax,bx -> cx,bx
  pusha
  push es
  mov x0,ax                     ;sauve les coordonnées pour
                                ;usage ultérieur
  mov y0,bx
  mov x1,cx
  sub cx,ax                     ;calcule le nombre de points
                                ;à dessiner
  jne zzok
  inc cx
zzok:
  mov zz,cx
  cmp verre,1                   ;surface de verre?
  jne Solid1
  push ax                       ;oui alors mode GDC: OR
  mov dx,3ceh
  mov ax,1003h                  ;registre 3: Function Select
  out dx,ax
  pop ax
Solid1:
  mov dx,3c4h                   ;port du Timing Sequencer
  mov di,0a000h
  mov es,di                     ;sélectionne le segment VGA
  mov di,ax                     ;calcule l'offset
  shr di,2                      ;(x div 4) + y*80
  add di,vpage                  ;ajoute la page courante
  mov bx,y0
  imul bx,80d
  add di,bx                     ;le tout dans di
  cmp zz,4
  jl no_moyen                   ;<4 points à dessiner ->
                                ;pas de groupe de 4
  and ax,11b                    ;les deux bits inférieurs sont
                                ;importants
  je milieu                     ;si 0 fixe immédiatement des
                                ;groupes de 4
```

```
no_moyen:
  mov bx,0f02h              ;si no_shift, utiliser ce masque
  mov cx,zz                 ;fixe le nombre de points dans
                            ;masque
  cmp cx,20h                ;à partir de 20h le 386 dérape
                            ;au décalage !
  jae no_shift
  mov bx,0102h              ;prépare le masque
  shl bh,cl                 ;nombre de points = nombre de
                            ;bits à mettre à 1
  dec bh
  and bh,0fh
no_shift:
  mov cx,ax                 ;le bon décalage selon le plan
                            ;de début
  and cl,3
  shl bh,cl
  mov ax,bx                 ;le masque est fini
  sub zz,4                  ;décrémente le nombre de
                            ;points à dessiner
  add zz,cx
start:
  out dx,ax                 ;installe le masque d'écriture
                            ;calculé
  mov al,b polycol          ;lit la couleur
  mov ah,es:[di]            ;charge les bascules, uniqt objets
                            ;en verre
  stosb                     ;installe un octet
milieu:
  cmp zz,4
  jl termine                ;plus de groupe de 4 -> terminus
  mov ax,0f02h              ;prend tous les plans
  out dx,ax                 ;(zz div 4) calcule les
                            ;groupes de 4
  mov cx,zz
  shr cx,2
  mov al,b polycol
  cmp verre,1               ;face en verre ?
```

```
  jne Solid
@lp:
  mov ah,es:[di]              ;charge les bascules, uniqmt
                              ;objets en verre
  stosb                       ;et réécrit un octet
  dec cx
  jne @lp
  jmp termine
Solid:
  rep stosb                   ;dessine la partie du milieu
termine:
  mov cx,x1                   ;dessine les pixels restants
  and cx,3h
  dec zz
  js hline_fini               ;si plus rien -> c'est fini
  mov ax,0102h
  shl ah,cl                   ;crée le masque
  dec ah
  out dx,ax
  mov al,b polycol            ;lit la couleur
  mov ah,es:[di]              ;charge les bascules, uniqmt
                              ;objets en verre
  stosb                       ;et dessine les points
hline_fini:
  mov dx,3ceh                 ;GDC Mode en position MOVE
  mov ax,0003h
  out dx,ax
  pop es
  popa
  ret
hline endp
txt_hline                     ;macro contenant la procédure
                              ;"hline_texture"
public fillpol
fillpol proc near             ;remplit un polygone en mode X
  push bp
  pusha
  cmp texture,1               ;utilise-t-on des textures ?
```

```
  jne Remplir                    ;non, remplissage simple
  txt_Detprinc                   ;sinon calcule le déterminant
                                 ;principal
Remplir:
  xor si,si                      ;recherche le point le plus
                                 ;haut, sél première entrée
  mov cx,polyn                   ;nombre de sommets
  sub cx,2
  mov bx,0ffffh                  ;valeur extrême, interdite
npoint:
  mov ax,poly2d[si+2]            ;lit y
  cmp ax,bx                      ;compare au minimum courant
  ja  no_min
  mov bx,ax                      ;fixe un nouveau minimum
  mov dx,si
no_min:
  add si,8
  dec cx                         ;sommet suivant, si pas 0ffffh
  jns npoint
  mov point_haut,dx              ;sauve en variable globale
  mov y_haut,bx                  ;recherche du point le plus
                                 ;haut achevée
  or dx,dx                       ;gauche = 0 ?
  jne dec_valid
  mov bx,polyn                   ;oui: positionner à droite
  sub bx,2
  shl bx,3
  jmp lr_fini                    ;à l'autre extrémité
dec_valid:
  mov bx,dx                      ;sinon un de moins
  sub bx,8
lr_fini:
  mov gauche,dx                  ;sauve en variable globale
  mov droite,bx
; ax,bx : coordonnées de départ gauche (xl0,yl0)
; cx/dx : coordonnées de départ droite (xr0,yr0)
; si    : indicateur de débordement gauche
; di    : indicateur de débordement droite
```

```
; bp    : pointeur sur point courant
  setnewlinel                    ;charges les variables des segments
  setnewliner
boucleg:
  cmp ax,xl1
  je nouv_ligneg                 ;si fini -> nouveau segment
  cmp bx,yl1
  je nouv_ligneg                 ;sinon poursuite du dessin
  or si,si                       ;indicateur d'incrémentation <= 0
  jg flaglgrand
incyl:                           ;ce passage va être modifié !
  inc bx                         ;y suivant
  add si,dxl                     ;incrémente l'indicateur IncFlag
  txt_incl                       ;coordonnées 3D suivantes
  cmp bx,yl1                     ;destination atteinte ?
  je nouv_ligneg                 ;alors nouveau segment
  jmp gauche_plus                ;y augmenté à gauche ->
                                 ;maintenant à droite
flaglgrand:
  sub si,dyl                     ;décrémente Incflag
incxl:                           ;ce passage va être modifié !
  inc ax                         ;x suivant
  jmp boucleg
fini__:
  jmp fini
nouv_ligneg:
  mov bx,gauche                  ;prépare l'augmentation
  cmp bx,droite
  je fini__                      ;identique, alors fini
  add bx,8                       ;gauche suivant
  mov ax,polyn                   ;gauche en fin de liste ?
  shl ax,3
  sub ax,8                       ;détermine la fin
  cmp bx,ax                      ;comparaison
  jb fixer_gauche
  xor bx,bx                      ;si oui, alors mise à 0
fixer_gauche:
  mov gauche,bx
```

```
  setnewlinel                ;recharge les variables
  jmp boucleg
fini_:
  jmp fini
gauche_plus:
boucled:
  cmp cx,xr1
  je nouv_ligned             ;si fin -> fixe un nouveau segment
  cmp dx,yr1
  je nouv_ligned             ;sinon poursuit
  or di,di                   ;indicateur d'incrémentation <= 0
  jg flagrgrand
incyr:                       ;ce passage va être modifié !
  inc dx                     ;y suivant
  add di,dxr                 ;incrémente IncFlag
  txt_incr
  cmp dx,yr1                 ;destination atteinte ?
  je nouv_ligned             ;alors nouveau segment
  jmp droite_plus            ;à droite y a été augmenté ->
                             ;trace le segment horiz
flagrgrand:
  sub di,dyr                 ;décrémente Incflag
incxr:
  inc cx                     ;ce passage va être modifié !
  jmp boucled
nouv_ligned:
  mov dx,droite              ;prépare la diminution
  cmp dx,gauche
  je fini_                   ;si égal, alors fini
  sub dx,8                   ;si précédmt à 0->passe à
                             ;l'autre extrémité
  jns fixer_droite
  mov dx,polyn
  sub dx,2
  shl dx,3                   ;se positionne à l'extrémité
fixer_droite:
  mov droite,dx
  setnewliner                ;recharge les variables
```

```
  jmp boucled
droite_plus:
  push ax
  push cx
  cmp cx,ax                    ;ordre correct ?
  jae direct_ok                ;alors ok, sinon:
  cmp w su_cacher,0            ;dissimuler les faces cachées?
  je dessiner                  ;non, alors on dessine quand même
  pop cx
  pop ax
  jmp fini                     ;polygone non dessiné
dessiner:
  xchg ax,cx                   ;coordonnées dans le bon ordre
direct_ok:
  cmp texture,1                ;a-t-on une texture ?
  jne remp_norm                ;non, alors remplissage normal
  call hline_texture           ;dessine segment de texture
                               ;horizontal
  pop cx
  pop ax
  jmp boucleg                  ;et on continue
remp_norm:
  call hline                   ;dessine segment horizontal
  pop cx
  pop ax
  jmp boucleg                  ;et on continue
fini:
  popa
  pop bp
  ret
fillpol endp
code ends
end
```

D'abord, la boucle npoint recherche le point le plus haut (ordonnée y minimale). BX contient alors le minimum courant et SI le numéro du point correspondant. Les variables gauche et droite indiquent l'arête à gauche ou à droite en cours de traitement. Par exemple, si gauche pointe sur le point 1, un segment d'arête est en cours d'évaluation à gauche du point 1 vers le

point 2. Du côté droit, la variable est définie à l'opposé : un n° de point égal à 3 signifie qu'un segment est évalué du point 4 vers le point 3.

La variable "gauche" reçoit le point haut calculé et "droite" désigne un point situé avant dans la table de coordonnées, ou en fin de définition si "gauche" représente le point 0. Les deux macros setnewlinel et setnewliner sont alors invoquées qui chargent les variables nécessaires pour les arêtes droite et gauche.

Comme dans l'algorithme de Bresenham, on calcule delta x et delta y et en cas de signe négatif, on prend leur valeur absolue et on modifie l'emplacement du code approprié pour inverser le sens du tracé. On fait appel à une automodification qui n'a rien à voir avec la programmation structurée et qui est rejetée par la plupart des informaticiens, mais elle représente ici une alternative efficace et rapide en évitant de commander une boucle avec des variables.

L'indicateur Incflag signale à quel moment dans le déroulement de la boucle il faut augmenter l'ordonnée y. Il est d'abord chargé avec delta x et, en cas de pente supérieure à 1 (delta x < delta y), change de signe. Ceci, tant que les coordonnées courantes sont maintenues dans des registres pour atteindre une vitesse élevée de traitement. Du côté gauche, ce sont les registres AX/BX pour les coordonnées et SI pour Incflag, du côté droit respectivement CX/DX et DI.

L'algorithme de tracé de segment proprement dit utilise une sorte de pente définie par delta x et delta y. A chaque déplacement dans la direction x, l'indicateur Incflag (en SI et DI) est diminué de delta y. A chaque déplacement dans la direction y, l'indicateur est augmenté de delta x. Le changement de signe de Incflag permet de décider de la direction de l'étape suivante : si l'indicateur est positif, la direction sera l'axe des x, s'il est négatif, ce sera l'axe des y.

Soit par exemple delta x = 100 et delta y = 50. Au début, Incflag est chargé avec la valeur 100. Le premier déplacement se fait vers la droite (Incflag=100, donc >0, on soustrait delta y, ce qui donne 50). Le deuxième déplacement est semblable (Incflag devient 0). Le déplacement suivant est effectué dans la direction y car Incflag est <=0. Incflag est alors augmenté de delta x de sorte qu'il reprend la valeur 100. On effectue ainsi deux pas vers la droite puis un vers le bas, ce qui correspond bien à la pente souhaitée de 0.5.

Ces calculs sont pris en charge par les boucles boucleg et boucled pour les côtés gauche et droit. On commence par tester si le point de destination n'est pas déjà atteint. A cet effet, on a déplacé dans setNewLine le point de destination d'un point dans la direction x et dans la direction y. Il suffit donc à présent de tester si l'une des deux coordonnées ne renvoie pas au point de destination étendu. Dans ce cas, le point de destination d'origine a déjà été dépassé.

La suite du déroulement jusqu'à l'étiquette fini correspond à l'algorithme décrit avec cependant une extension : si l'ordonnée y est augmentée, on se branche sur l'étiquette gauche_plus ou droite_plus. Le premier cas implique une entrée dans la boucle droite boucled et le second un appel du tracé de ligne horizontal. En clair, le segment gauche est "tracé" jusqu'à ce que l'ordonnée y soit incrémentée, puis on procède de même à droite. Lorsque les deux côtés ont progressé d'un point dans le sens des ordonnées, la ligne correspondante peut être remplie.

Dès qu'une ligne est arrivée à son terme, il faut rechercher la suivante dans la liste et initialiser les variables correspondantes. Ces actions sont effectuées à partir des étiquettes nouv_ligneg et nouv_ligned.

On regarde d'abord si gauche et droite sont identiques. Dans ce cas, le dessin du polygone est terminé et la procédure peut être abandonnée par fini_ ou fini__. Sinon, on avance du côté gauche le pointeur (gauche) d'une position (8 octets) et on recule du côté droit le pointeur droit. Il faut évidemment tenir compte des bords : après le dernier point vient le premier et vice-versa.

Une fois les pointeurs chargés, les variables de segment doivent aussi être recalculées. On appelle setnewlinel et setnewliner et on se branche à nouveau sur la boucle.

Après double incrémentation de l'ordonnée y, la ligne de remplissage est tracée. Cette action se produit à l'étiquette droite_plus. La procédure hline est invoquée (nous expliquerons les variantes plus loin) : elle trace une droite allant du point ax,bx au point cx,bx. On retourne ensuite dans la boucle gauche.

Cette procédure commence par sauvegarder les variables qui sont encore nécessaires et calculent la longueur du trajet. Si les faces sont en verre, comme c'est le cas dans ce programme, on fait appel au GDC pour exécuter une combinaison OR entre l'ancien contenu des bascules et les nouvelles données.

La façon la plus simple de tracer un segment horizontal est de mettre dans la même couleur les points successifs qui la forment. Mais en mode X, cette approche n'est pas très fine car l'adressage individuel des pixels est très lent. On essayera plutôt de dessiner quatre points d'un coup, ou tout au moins le maximum possible à l'adresse envisagée.

A l'étiquette Solid1, on calcule l'adresse de destination du premier octet. A partir de l'étiquette no_moyen, on dessine le premier bloc incomplet. On arrive directement à cet endroit lorsque le nombre total de points à dessiner est inférieur à 4, ce qui est nécessaire lorsque le plan de début (trouvé par combinaison AND entre l'abscisse x et la constante 11b) est différent de 0. Pour pouvoir dessiner ce bloc, il faut d'abord masquer le nombre de pixels à tracer en bh. Avec un très grand nombre de pixels, le masque est d'abord mis

à 0f, ce qui masque tous les pixels du bloc, sinon on se limite au nombre de bits nécessaires. Ce masque est déplacé en fonction du plan de début et après réduction de zz envoyé au Timing Sequencer.

On exécute alors un accès en lecture à l'offset souhaité, non pas pour procurer au processeur des informations sur le contenu de l'écran mais simplement pour charger les bascules pour que l'accès en écriture suivant puisse combiner la nouvelle couleur avec les données correctes.

S'il y a moins de quatre pixels à dessiner, on se branche à l'étiquette termine. Sinon tous les différents plans doivent être activés et le nombre correspondant de blocs de 4 pixels colorés en conséquence. Cette coloration se fait pour les objets opaques par une instruction rep stosb. Avec les objets en verre, la chose est un peu plus compliquée. Pour chaque octet il faut faire un accès en lecture de sorte qu'une boucle devient indispensable.

A la fin, tous les pixels du dernier bloc sont dessinés jusqu'aux coordonnées de destination. On forme un masque en fonction du plan de destination et la couleur est appliquée. L'étiquette hline_fini remet le mode d'écriture du GDC en position MOVE, après quoi la procédure est terminée.

8.6. Arêtes cachées : hidden line

Les formes en verre peuvent avoir leur charme propre mais elles conviennent rarement pour représenter des objets réels. La plupart des objets sont opaques, il faut donc que leur reproduction sur ordinateur ne montre pas leurs faces cachées.

Le problème de la dissimulation des faces cachées est l'un des plus complexes que pose la représentation tridimensionnelle. Il ne s'agit pas seulement de dessiner certaines faces et de ne pas dessiner les autres. Deux surfaces peuvent se recouvrir partiellement de sorte qu'il faut dessiner les deux mais dans le bon ordre. Nous allons présenter les deux types de méthodes en usage : inhibition et tri.

Pour inhiber les faces cachées, on prend généralement en considération l'angle que fait la face avec la direction du regard (droite reliant l'oeil de l'observateur à la face considérée). Cet angle permet de savoir si l'observateur regarde l'avant ou l'arrière de la face. Dans ce dernier cas, il faut empêcher sa représentation.

La méthode présentée ici s'inspire de ce schéma mais en introduisant une simplification de taille. Nous supposerons que toutes les faces sont définies

dans le sens inverse des aiguilles d'une montre (positives dans le sens mathématique). Ainsi toute face, indépendamment de sa situation dans l'espace, est toujours dessinée d'un geste cursif vers la gauche. Mais si elle s'est tournée de façon que l'observateur la regarde par l'arrière, elle apparaît transformée par symétrie et tracée par la droite. Lorsqu'on dessine un segment horizontal par hline, on teste si le point final correspondant à la définition de l'algorithme de surface se trouve à droite du point de départ. Si tel n'est pas le cas, c'est qu'on observe une face arrière, qui doit être dissimulée. Dans le programme, le test se fait dans la procédure Fillpol du module POLY.ASM après l'étiquette droite_plus :

```
droite_plus:
  push ax
  push cx
  cmp cx,ax                    ;ordre correct ?
  jae direct_ok                ;alors ok, sinon:
  cmp w su_cacher,0            ;dissimuler les faces cachées ?
  je dessiner                  ;non, alors on dessine quand même
  pop ax
  jmp fini                     ;polygone non dessiné
dessiner:
  xchg ax,cx                   ;coordonnées dans le bon ordre
direct_ok:
```

Les registres AX et CX contiennent les abscisses du point initial et du point final. Pour une surface définie positivement en termes mathématiques, on a CX supérieur à AX. Si tel est le cas, on passe directement au dessin (étiquette direct_ok) sinon on arrête le remplissage (si su_cacher est TRUE, sinon les circonstances sont ignorées et après permutation des coordonnées on procède quand même au dessin).

Cette méthode est suffisante pour les corps convexes, c'est-à-dire ceux qui n'ont pas de "creux" comme les dés. Mais que se passe-t-il pour un objet concave qui aurait par exemple la forme d'un U ? Les faces invisibles sont sélectionnées mais leur ordre n'est pas correct : certaines faces sont visibles en totalité, alors qu'elles sont en fait recouvertes par d'autres.

Si on trie les faces de façon que soit dessinée en premier celle qui possède la plus grande cote z (qui se trouve le plus loin vers l'arrière), les nouvelles faces apparaissant plus en avant recouvrent en partie des faces déjà dessinées. Cette méthode est largement exploitée par les logiciels de dessin, on l'appelle ainsi "Painter's algorithm".

Trier des faces, c'est bien mais comment faire ? Les sommets d'une face ont en général des cotes z complètement dissemblables. Les algorithmes complexes et lents essayent d'en tenir compte et de trouver des relations entre les sommets qui permettent une affectation sans ambiguïté. Nous préférerons pour notre part le tri par profondeur moyenne qui est beaucoup plus rapide.

On calcule à cet effet la valeur moyenne de l'information de profondeur de chaque face qui devient le critère de tri. Cette méthode est très imprécise, surtout lorsque les faces s'étendent loin dans la direction z. Mais pour la dissimulation des faces cachée elle donne des résultats intéressants et surtout rapides.

Le tableau moyen déjà mentionné mémorise ces valeurs moyennes ainsi que les pointeurs de face associés? A cet effet, la macro z2cx ajoute des valeurs de z toutes prêtes à la position courante du tableau. La moyenne est alors formée à l'étiquette polyok, par division de la somme par le nombre de sommets.

```
polyok:
  mov bx,moyenptr              ;calcule la valeur moyenne :
  mov ax,moyen[bx]             ;lit la somme
  mov cx,polyn
  dec cx
  cwd
  div cx                       ;et la divise par le nombre de sommets
  mov moyen[bx],ax             ;puis enregistre le résultat
```

Si la variable su_sort est à TRUE, on fait suivre le calcul de toutes les faces par la procédure Quicksort qui trie le tableau "moyen" avec le célèbre algorithme Quicksort. L'information de profondeur (à chaque fois dans la partie inférieure d'un double mot) sert de critère de tri et l'identification de la face est évidemment triée en même temps.

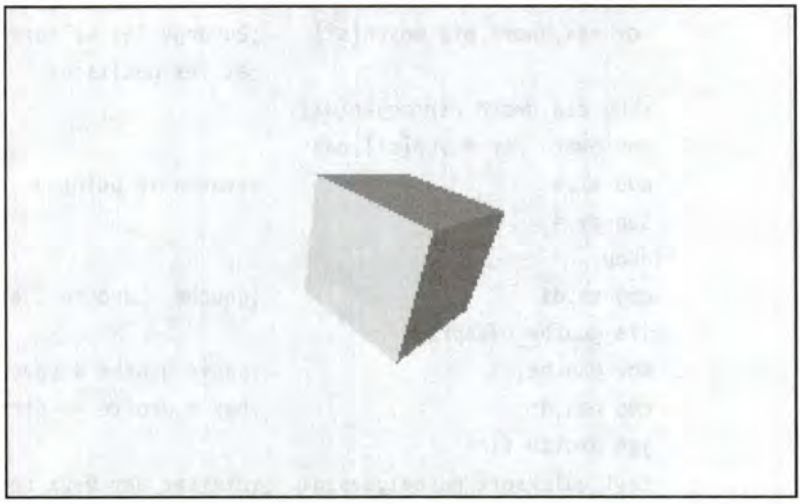
```
public quicksort
quicksort proc pascal bas,haut:word
;trie le tableau moyen avec l'algorithme Quicksort
local cle:word
local gauche:word
  push bx
  mov bx,bas                   ;trouve le milieu
  add bx,haut
  shr bx,1
  and bx,not 3                 ;positionnement sur groupe de 4
```

```
  mov dx,moyen[bx]              ;lit la clé
  mov cle,dx
  mov  ax,bas                   ;initialise droite et gauche
                                ;avec valeurs de base
  mov si,ax
  mov gauche,ax
  mov ax,haut
  mov di,ax
  mov dx,cle
gauche_pluspres:
  cmp moyen[si],dx              ;plus grand que clé -> on
                                ;continue la recherche
  jbe a_gauche
  add si,4                      ;position suivante
  jmp gauche_pluspres           ;on la teste
a_gauche:
  cmp moyen[di],dx              ;plus petit que la clé-> on
                                ;continue la recherche
  jae a_droite
  sub di,4                      ;position suivante
  jmp a_gauche                  ;on la teste
a_droite:
  cmp si,di                     ;gauche <= droite ?
  jg finbou                     ;non -> partie triée
  mov eax,dword ptr moyen[si]   ;échange les valeurs moyennes
                                ;et les positions
  xchg eax,dword ptr moyen[di]
  mov dword ptr moyen[si],eax
  add si,4                      ;avance le pointeur
  sub di,4
finbou:
  cmp si,di                     ;gauche > droite , alors on continue
  jle gauche_pluspres
  mov gauche,si                 ;sauve gauche à cause récursion
  cmp bas,di                    ;bas < droite -> partie gauche triée
  jge droite_fini
  call quicksort pascal,bas,di  ;diviser par deux récursivt,
                                ;poursuivre le tri
```

```
droite_fini:
  mov si,gauche                    ;haut > gauche -> partie droite triée
  cmp haut,si
  jle gauche_fini
  call quicksort pascal,si,haut    ;diviser par deux récursivt,
                                   ;poursuivre le tri
gauche_fini:
  pop bx
  ret
quicksort endp
code ends
end
```

L'algorithme Quicksort divise l'ensemble à trier en deux parties et permute les éléments jusqu'à ce que la moitié gauche ne contienne plus que des éléments supérieurs à l'élément moyen (la partie droite ne contenant plus que des éléments inférieurs). Une fois cet objectif atteint, les deux sous-ensembles sont à nouveau soumis récursivement à la même procédure. Le détail de la méthode se trouve dans tous les livres d'algorithmique.

Lors du remplissage des faces, on renonce à commuter le GDC de sorte que les données arrivent directement dans la mémoire d'écran et produisent une surface opaque. On peut exploiter ici une boucle rapide rep stosb car le chargement des bascules préalablement à tout accès en écriture n'est plus nécessaire.



La rotation du cube avec des surfaces pleines opaques

A l'inverse de 3D_VERRE.PAS, le programme 3D_SOLID.PAS renonce à générer une palette (d'autres couleurs sont définies pour les faces) et il initialise différemment les variables globales : verre est à FALSE puisque les faces ne sont pas en verre et les deux variables de traitement des surfaces cachées su_cacher et su_sort sont à TRUE. Pour le reste, les programmes sont identiques.

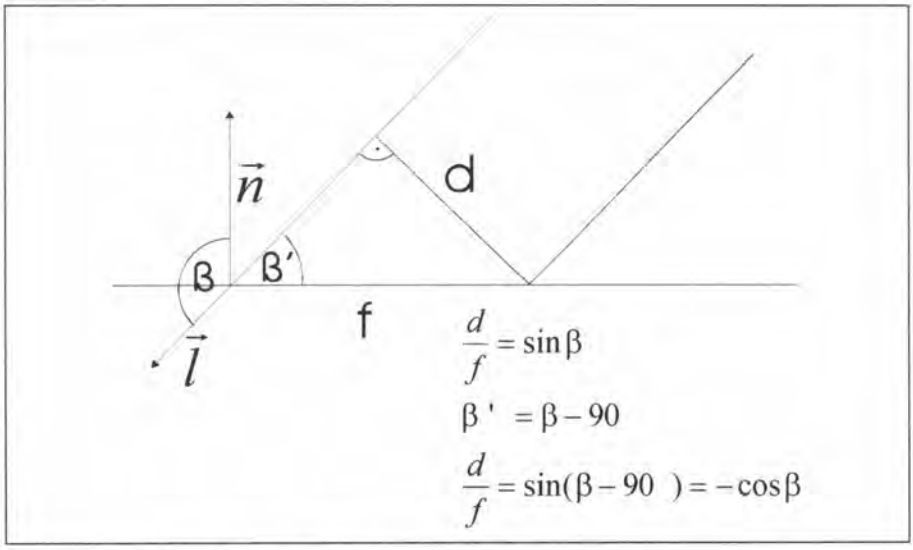
8.7. Jeux d'ombre et de lumière

Nous disposons maintenant d'objets solides et opaques, capables de se mouvoir librement dans l'espace. Nous n'avons pas encore pris en considération l'éclairage. Lorsque l'on introduit une source de lumière, la représentation des espaces tridimensionnels devient encore plus impressionnante.

Sur les PC actuels, il n'est pas encore possible de faire du Ray Tracing en temps réel. La mise au point des images demande au minimum plusieurs minutes. Nous avons besoin d'une méthode plus rapide qui n'étudie pas chaque rayon de lumière mais permet d'obtenir des effets intéressants au prix de quelques simplifications.

Si la source de lumière est éloignée à l'infini, tous les rayons tombent parallèlement sur les faces des objets. Un seul vecteur de lumière suffit à faire les calculs au lieu d'un vecteur pour chaque point. Cette illumination homogène permet de calculer pour chaque face une luminosité à laquelle on adapte sa coloration.

Il se pose la question de savoir comment calculer la luminosité d'une face. Nous allons nous servir d'un modèle simplifié. Plus la lumière est rasante, plus la face s'assombrit. Par contre, si la lumière est perpendiculaire à la face, la luminosité de cette dernière est maximale. Une même quantité d'énergie est en effet dissipée sur une surface plus grande lorsque l'angle d'incidence se fait plus faible.



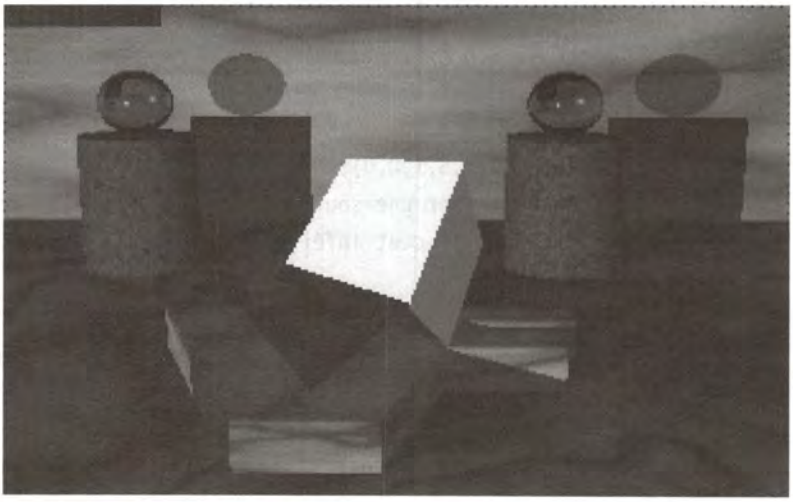
Intensité de la lumière éclairant une surface

Le rapport d/f est ici proportionnel à la luminosité de la face. Il est égal au cosinus négatif de l'angle que fait le vecteur de lumière avec le vecteur normal de la face. Le vecteur normal est un vecteur perpendiculaire à la face. Il s'obtient facilement par produit vectoriel de deux vecteurs situés dans le plan de la face.

Le fait que ce soit le cosinus qui intervienne et non l'angle lui-même simplifie énormément les calculs. En effet, le cosinus de l'angle apparaît comme résultat de sa détermination (par le produit scalaire).

Le processus de détermination de la luminosité sera donc le suivant :

- trouver deux vecteurs situés dans le plan de la face, par exemple deux vecteurs du bord (du premier au deuxième et au dernier point)
- former un vecteur normal (produit vectoriel des deux vecteurs de surface)
- calculer un produit scalaire pour déterminer le cosinus de l'angle entre le vecteur de lumière constant et le vecteur normal
- ajouter le résultat à la couleur



Le jeu d'ombres sur le cube donne l'impression que la lumière arrive par le coin supérieur gauche

Dans l'exemple dessiné, le résultat du calcul d'angle est négatif. Mais si la lumière s'éloigne de la surface (<90°), alors le résultat est positif, et seule la couleur de base interviendra car elle se trouve dans l'ombre et n'apparaît qu'en raison de la dispersion de la lumière.

Le programme 3D_LIGHT.PAS démontre les capacités de la routine de tracé des ombres :

```
Uses Crt,ModeXLib,Gif,var_3d;
Const
    worldlen=8*3;                {tableau des points }
    Worldconst:Array[0..worldlen-1] of Integer =
    (-200,-200,-200,
    -200,-200,200,
    -200,200,-200,
    -200,200,200,
    200,-200,-200,
    200,-200,200,
    200,200,-200,
    200,200,200);
    surfclen=38;                 {tableau des faces }
    surfcconst:Array[0..surfclen-1] of Word=
    ($fee0,4, 0,2,6,4,
```

```
    $fec0,4, 0,1,3,2,
    $fec0,4, 4,6,7,5,
    $fee0,4, 1,5,7,3,
    $fec0,4, 2,3,7,6,
    $fec0,4, 0,4,5,1,0,0);
   { $fe = utiliser une source de lumière, couelur de base
           dans l'octet inférieur }
Var
   i,j:Word;
Procedure Ombre_Pal;              {prépare la palette au dessin
                                   des ombres }
Begin
  For j:=192 to 223 do Begin      {traite les couleurs 192 -
                                   223 et 224 - 255 }
    i:=trunc((j/32)*43);          {définit la luminosité }
    Fillchar(Palette[j*3],3,i+20);
                                  {couleurs 192-223 en niveaux de gris}
    Palette[(j+32)*3]:=i+20;      {couleurs 224-255 en rouge }
    Palette[(j+32)*3+1]:=0;
    Palette[(j+32)*3+2]:=0;
  End;
  Setpal;                         {réalise la palette }
End;
procedure drawworld;external;     {dessine l'espace des objets
                                   dans la page d'écran courante }
{$l 3dasm.obj}
{$l poly.obj}
{$l bres.obj}
{$l \bp\myunit\racine.obj}
Begin
  vz:=1000;                       {profondeur de l'objet }
  vpage:=0;                       {commence en page 0 }
  LoadGif('logor.gif');           {charge l'image de fond }
  init_modex;                     {enclenche le mode X }
  Ombre_Pal;                      {calcule la palette des ombres}
  rotx:=0;                        {valeurs initiales de rotation}
  roty:=0;
  rotz:=0;
```

```
  remplir :=true;                 {remplissage actif }
  su_sort:=true;                  {tri des faces }
  su_cacher:=true;                {traitement des faces cachées}
  verre:=false;                   {pas d'objet en verre }
  p13_2_modex(16000*2,16000);     {fond d'écran en page 2 de la
                                   carte VGA }
  repeat
    CopyScreen(vpage,16000*2);    {fond d'écran en page courante}
    DrawWorld;                    {dessine l'espace des objets }
    switch;                       {montre l'image achevée }
    WaitRetrace;                  {attend le prochain retour de
                                   balayage }
    Inc(rotx);                    {poursuit la rotation ... }
    If rotx=120 Then rotx:=0;
    Inc(rotz);
    If rotz=120 Then rotz:=0;
    inc(roty);
    if roty=120 Then roty:=0;
  Until KeyPressed;               { ...jusqu'à ce qu'on frappe
                                      une touche }
  TextMode(3);
End.
```

Avant que l'on puisse représenter la luminosité, il faut préparer la palette. On remplit les éléments 192 à 223 et 224 à 255 avec deux gradients de couleurs. Le premier va du gris au blanc, et le second du rouge sombre au rouge. Au remplissage, il suffira d'additionner le cosinus à la couleur de base pour obtenir la bonne luminosité.

La couleur de base est codée dans l'octet inférieur de l'information de couleur (ici 0c0h et Oe0h), alors que l'octet supérieur contient le premier octet de commande : 0feh a pour effet d'activer l'ombrage d'une face. Il sera possible de mélanger des faces ombrées (octet de commande 0feh) avec des faces à valeur constante (octet de commande < 0feh) ou plus tard avec des textures (octet de commande 0ffh).

L'image de fond d'écran constitue également une différence avec le programme 3D_SOLID. Au début, on charge une image calculée par Ray Tracing et qui possède un vecteur de lumière semblable à celui des objets tournants, ce qui accroît l'impression de présence d'une source lumineuse. Au lieu d'effacer la mémoire d'écran à chaque génération d'image, on se contente de recopier l'image de fond dans la page correspondante.

La procédure Drawworld (dans 3DASM.ASM) commence par décoder l'octet de commande et initialise en conséquence les variables globales Lightsrc et Texture (seule la variable Lightsrc est ici mise à TRUE).

Par ailleurs, les tableaux Points3D et Poly3D entrent maintenant en scène. Le premier mémorise les coordonnées tridimensionnelles en fin de rotation, telles qu'elles ont été calculées par la macro zrot. Celles-ci sont alors transférées par la boucle npoint dans le tableau Poly3D, avec fermeture de la figure, le premier sommet étant recopié dans le dernier. Ce processus correspond exactement à celui exécuté avec les coordonnées à deux dimensions.

```
getdelta macro                  ;calcule les deux vecteurs
                                ;compris dans le plan de la surface
  mov ax,poly3d[0]              ;x: sommet de départ
  mov delta2[0],ax              ;sauvegarde temporaire en delta2
  sub ax,poly3d[8]              ;différence avec le premier point
  mov delta1[0],ax              ;delta1 terminé
  mov ax,poly3d[2]              ;y: sommet de départ
  mov delta2[2],ax              ;sauvegarde temporaire en delta2
  sub ax,poly3d[10d]            ;différence avec le premier point
  mov delta1[2],ax              ;delta1 terminé
  mov ax,poly3d[4]              ;z: sommet de départ
  mov delta2[4],ax              ;sauvegarde temporaire en delta2
  sub ax,poly3d[12d]            ;différence avec le premier point
  mov delta1[4],ax              ;delta1 terminé
  mov bp,polyn                  ;sélectionne le dernier point
  dec bp
  shl bp,3                      ;8 octets à chaque fois
  mov ax,poly3d[bp]             ;lit x
  sub delta2[0],ax              ;forme la différence
  mov ax,poly3d[bp+2]           ;lit y
  sub delta2[2],ax              ;forme la différence
  mov ax,poly3d[bp+4]           ;lit z
  sub delta2[4],ax              ;forme la différence
endm
```

Delta1 reçoit ici la différence entre le premier et le deuxième sommet du polygone, delta2 la différence entre le premier et le dernier sommet. Tant qu'on ne définit pas des surfaces farfelues, ces deux vecteurs ne sont pas colinéaires et conviennent donc à l'opération projetée. Sinon le programme s'interrompt avec un message de division par zéro.

Si, comme c'est le cas ici, la variable globale lightsrc est TRUE, les deux macros get_normal et light sont appelées. La première définit le vecteur normal de la face à laquelle appartiennent les deux vecteurs delta1 et delta2. La deuxième macro définit la luminosité à partir de l'angle d'incidence.

```
get_normal macro                ;calcule le vecteur normal d'une face
  mov ax,delta1[2]              ;a2*b3
  imul delta2[4]
  shrd ax,dx,4
  mov n[0],ax
  mov ax,delta1[4]              ;a3*b2
  imul delta2[2]
  shrd ax,dx,4
  sub n[0],ax
  mov ax,delta1[4]              ;a3*b1
  imul delta2[0]
  shrd ax,dx,4
  mov n[2],ax
  mov ax,delta1[0]              ;a1*b3
  imul delta2[4]
  shrd ax,dx,4
  sub n[2],ax
  mov ax,delta1[0]              ;a1*b2
  imul delta2[2]
  shrd ax,dx,4
  mov n[4],ax
  mov ax,delta1[2]
  imul delta2[0]
  shrd ax,dx,4
  sub n[4],ax                   ;produit vectoriel
                                ;(=vecteur normal) calculé
  mov ax,n[0]                   ;x1 ^ 2
  imul ax
  mov bx,ax
  mov cx,dx
  mov ax,n[2]                   ;+x2 ^ 2
  imul ax
  add bx,ax
  adc cx,dx
```

```
  mov ax,n[4]                    ;+x3 ^ 2
  imul ax
  add ax,bx
  adc dx,cx                      ;somme en dx:ax
  push si
  call racine                    ;racine en ax
  pop si
  mov n_long,ax                  ;longueur du vecteur normal calculée
endm
```

La première partie de cette macro évalue le vecteur normal proprement dit. Conformément à la définition du produit vectoriel, les composantes résultent d'une différence de deux produits. Elles sont stockées dans le tableau n. La deuxième partie calcule la longueur du vecteur en élevant au carré les composants et en les additionnant. De la somme obtenue, on extrait la racine carrée par une méthode itérative : le résultat est mémorisé en n_long.

Les dernières étapes sont effectuées par la macro light :

```
light macro                      ;détermine la luminosité d'une face
  mov ax,n[0]
  imul l[0]                      ;vecteur de la lumière *
                                 ;vecteur normal
  mov bx,ax                      ;somme en cx:bx
  mov cx,dx
  mov ax,n[2]
  imul l[2]
  add bx,ax
  adc cx,dx
  mov ax,n[4]
  imul l[4]
  add ax,bx                      ;produit scalaire en dx:ax
  adc dx,cx
  idiv l_long                    ;division par l_long
  mov bx,n_long                  ;et par n_long
  cwd
  shld dx,ax,5                   ;valeurs de -32 à +32
  shl ax,5d
  mov bp,startpoly               ;prépare l'adressage de la
                                 ;couleur de surface
```

```
  idiv bx                    ;division par numérateur
  inc ax
  or ax,ax
  js danslumiere             ;si cos A positif -> lumière partante
  xor ax,ax                  ;donc pas d'éclairage
danslumiere:
  sub b polycol,al           ;cos<0 -> addition à la couleur de
                             ; base
endm
```

Après avoir formé en l le produit scalaire du vecteur normal par le vecteur de la lumière (constant), on enchaîne avec une division par la longueur du vecteur de lumière et par la longueur du vecteur normal. On obtient ainsi l'angle entre les deux vecteurs (par son cosinus). Normalement, le résultat de cette opération doit être compris entre -1 et +1. Mais comme on travaille ici sur des nombres entiers, on multiplie par 32 le résultat intermédiaire de la première division, ce qui donne finalement un intervalle compris entre -32 et +32.

Si le résultat est positif, Ax est mis à 0 et la couleur de base est maintenue. S'il est négatif, il est retranché de la couleur de base ce qui revient à ajouter la valeur absolue. La luminosité calculée se trouve finalement dans la variable PolyCol qui est utilisée par FillPol pour le remplissage .

8.8. Des faces impressionnantes : les textures

Le dernier perfectionnement présenté par les routines 3D de cet ouvrage est l'inclusion de textures qui prêtent une structure à des surfaces autrement lisses et monochromes. A cet effet, on projette des graphiques en mode point (bitmaps) sur les faces d'un objet. Ces graphiques se déplacent en même temps que l'objet à chaque rotation.

Concrètement, les bitmaps paraissent collés sur les surfaces et peuvent simuler certaines apparences comme celle du bois ou du métal. Cette technique est souvent utilisée dans les jeux de rôle comme Ultima Underworld où les murs sont tantôt en pierre, tantôt en bois ou en d'autres matériaux.

Si on réfléchit à la programmation nécessaire, la plus simple des solutions paraît s'imposer de prime abord. On compose une face avec de multiples petites facettes qui correspondent à un point du bitmap. Cette technique n'est pas la plus rapide, et même dans la plupart des cas elle ne fonctionne pas

correctement. Car si une face ainsi construite est agrandie ou tournée, des lacunes apparaissent aussitôt, certains points se chevauchant et créant un vide à côté d'eux.

La solution vraiment praticable consiste à inverser la démarche. Lors de la représentation de la face, on procède comme auparavant en dessinant chaque point. Mais ce point est rétroprojeté sur sa surface d'origine pour y définir sa situation exacte dans l'espace. En conclusion, on peut alors déterminer la couleur du point à partir du bitmap de la texture.

Comme il est impossible de déduire la situation tridimensionnelle d'un point à partir de ses coordonnées bidimensionnelles, on tient compte des coordonnées 3D au stade du remplissage. De chaque point, on connaît donc les coordonnées et la situation exacte.

Le cube avec une texture sur chacune de ses faces

A titre de démonstration des routines de texture, voici le programme 3D_TEXTU.PAS. Il se distingue des précédents uniquement par la présence d'autres couleurs dans la définition de l'espace des objets et par l'appel à la procédure Prep_Textures qui survient dès le début du programme.

```
Uses Crt,ModeXLib,Gif,var_3d;
Const
    worldlen=8*3;                {tableau des points }
    Worldconst:Array[0..worldlen-1] of Integer =
    (-200,-200,-200,
    -200,-200,200,
    -200,200,-200,
```

```
    -200,200,200,
    200,-200,-200,
    200,-200,200,
    200,200,-200,
    200,200,200);
    surfclen=38;                    {tableau des faces }
    surfcconst:Array[0..surfclen-1] of Word=
    ($ff00,4, 0,2,6,4,
     $ff01,4, 0,1,3,2,
     $ff02,4, 4,6,7,5,
     $ff00,4, 1,5,7,3,
     $ff03,4, 2,3,7,6,
     $ff04,4, 0,4,5,1,0,0);
   { $ff = utiliser des textures, n° dans l'octet inférieur}
Var
    i,j:Word;

Procedure Prep_Textures;
{charge les variables définissant les textures }
Begin
  LoadGif('Textures');              {charge l'image avec les textures }
  GetMem(Txt_Pic,64000);            {prépare de la mémoire }
  Move(VScreen^,Txt_Pic^,64000); {et y effectue la copie }
  For i:=0 to Txt_Nbre-1 do Begin
    Txt_Data[i]:=Txt_Pic;           {pointeur sur données }
    Txt_Offs[i]:=i*64;              {définit l'offset }
  End;
End;
procedure drawworld;external;
{dessine l'espace des objets dans la page d'écran courante }
{$l 3dasm.obj}
{$l poly.obj}
{$l bres.obj}
{$l racine.obj}
Begin
  vz:=1000;                         {profondeur de l'objet }
  vpage:=0;                         {commence en page 0 }
  init_modex;                       {active le mode X }
```

```
  Prep_Textures;
  LoadGif('logor.gif');          {charge l'image de fond }
  rotx:=0;                       {rotation initiale }
  roty:=0;
  rotz:=0;
  remplir:=true;                 {demande de remplissage }
  su_sort:=true;                 {tri des surfaces activé }
  su_cacher:=true;               {dissimule les surfaces cachées}
  verre:=false;                  {pas de surface en verre }
  p13_2_modex(16000*2,16000);    {fond en page 2 VGA }
  repeat
    CopyScreen(vpage,16000*2);   {...recopié dans la page courante }
    DrawWorld;                   {dessine l'espace }
    switch;                      {active l'écran préparé }
    WaitRetrace;                 {attend le prochain retour de
                                  balayage }
    Inc(rotx);                   {poursuit la rotation ... }
    If rotx=120 Then rotx:=0;
    Inc(rotz);
    If rotz=120 Then rotz:=0;
    inc(roty);
    if roty=120 Then roty:=0;
  Until KeyPressed;              { ... jusqu'à frappe de touche}
  TextMode(3);
End.
```

L'octet de commande est 0ffh pour toutes les faces. Il active pour chaque face la texture dont le numéro est indiqué dans l'octet inférieur. La procédure Prep_Textures charge d'abord l'image GIF avec les textures nécessaires et la déplace dans une autre zone de mémoire (TxtPic^). C'est alors qu'interviennent les tableaux Txt_Data et Txt_Offs. Le premier tableau associe à chaque texture un pointeur sur la position de l'écran de texture, il est donc possible de charger plusieurs écrans indépendants avec des textures. Le tableau Txt_Offs contient l'offset de la texture correspondante à l'intérieur de cet écran.

Dans notre cas, Txt_Data contient pour toutes les textures un pointeur sur Txt_Pic^ et Txt_Offs est rempli avec des multiples de 64, ce qui donne 5 textures contiguës de 64 pixels de large.

La taille des textures est déterminée par le tableau de constantes Txt_Taille où les octets supérieurs donnent la hauteur (y) et les octets inférieurs la

largeur (x) de chaque texture. On respecte cependant une convention particulière pour la définition de ces tailles qui sont toujours de la forme 256 shr (n-8) pixels. Le nombre 10 (0ah) indique par exemple une taille de 256 shr 2 = 64 pixels (dans la direction x ou y).

Certaines parties de programme déjà étudiées, comme la formation des vecteurs de surface Delta1 et Delta2, trouvent également leur application dans le calcul des textures. Mais la plupart des algorithmes utilisés se présentent sous forme de macros dans le fichier d'inclusion TEXTURE.INC, qui joue un rôle de complément pour POLY.ASM.

Lorsqu'on traite des textures, la variable globale Texture est égale à TRUE. Au début de la procédure FillPol, on calcule donc le déterminant principal avec la macro Txt_Detprinc. Elle permettra plus tard de déterminer les coordonnées relatives d'un point à l'intérieur de sa surface. Il est important à cet égard de retenir que chaque point appartenant à une face constitue une solution du système d'équations suivant :

```
x1= lambda1 * a1 + lambda2 * b1
x2= lambda1 * a2 + lambda2 * b2
x3= lambda1 * a3 + lambda2 * b3
```

Dans ces expressions, x1-x3 sont les coordonnées du point, a1-a3 les composantes du premier vecteur de surface (Delta1), b1-b3 celles du deuxième (Delta2).

lambda1 et lambda2 donnent les coordonnées affines relativement aux deux vecteurs de surface et peuvent être exploités directement pour accéder aux textures. Pour calculer lambda1 et lambda2, il suffit de prendre en considération deux équations, la troisième étant alors forcément vérifiée puisque le point est dans le plan.

Si on prend par exemple les deux premières équations, la solution s'obtient facilement avec les déterminants. Le déterminant principal est D=a1*b2-a2*b1, le premier déterminant auxiliaire D1=x1*b2-x2*b1 et le deuxième déterminant auxiliaire D2=a1*x2-a2*x1. Les deux inconnues se calculent par lambda1=D1/D et lambda2=D2/D. Le déterminant principal est le même sur toute la surface de sorte qu'il est utile de l'évaluer directement ici.

```
txt_Detprinc macro              ;calcul de déterminant principal
  xor si,si                     ;premier essai: lignes  0 et 1
  mov di,2
suivante:
  mov ax,w delta1[si]           ;calcule le déterminant principal
  imul w delta2[di]
```

```
    mov bx,ax                   ;sauvegarde résultat intermédiaire
    mov cx,dx
    mov ax,w delta2[si]
    imul w delta1[di]
    sub bx,ax                   ;sauvegarde la différence
    sbb cx,dx
    mov w D,bx
    mov w D+2,cx
    or bx,cx                    ;déterminant principal = 0 ?
    jne D_fini
    add si,2                    ;alors nouvelles composantes
    add di,2
    cmp di,4                    ;dans la ligne disponible?
    jbe suivante
    xor di,di                   ;non, on recommence en haut
    jmp suivante
  D_fini:
    movsx eax,delta1[si]        ;sauvegarde les valeurs de
                                ;colonnes utilisées
    mov colonne1[0],eax
    movsx eax,delta1[di]
    mov colonne1[4],eax
    movsx eax,delta2[si]
    mov colonne2[0],eax
    movsx eax,delta2[di]
    mov colonne2[4],eax
    shl si,1                    ;prend note des colonnes utilisées
    shl di,1
    mov ligne_sup,si
    mov ligne_inf,di
  endm
```

Il se pose encore un problème que nous n'avons pas encore évoqué : dans certains cas, le choix des deux premières équations peut être malheureux, ce qui se traduit par un déterminant principal nul. Il faut alors sélectionner une autre combinaison d'équations.

Cette décision est prise avant l'étiquette D_fini. On se positionne simplement sur la ligne suivante à l'intérieur des vecteurs de surface (d'abord 0,2 puis 2,4 enfin 4,0) et un nouvel essai est tenté. Le déterminant principal est stocké

dans les tableaux colonne1 et colonne2 et les numéros des lignes utilisées sont sauvegardées dans les variables ligne_sup et ligne_inf. Comme vous le devinez, ces indications seront réutilisées par la suite.

L'algorithme de remplissage doit de son côté s'appliquer constamment à suivre les coordonnées du point courant. En même temps que sont testés les bords du polygone, on procède à des calculs de segments en trois dimensions dont l'ensemble décrit le polygone. Les macros SetNewLineL et SetNewLineR doivent être complétées par une initialisation des segments tridimensionnels associés. Cette tâche est effectuée par les macros Txt_MakeVarL et Txt_MakeVarR :

```
txt_makevarl macro              ;recharge les variables 3D du
                                ;côté gauche
.386
  movsx ebx,dyl                 ;nombre de pas
  inc ebx
  push ecx
  push edx
  movsx eax,poly3d[bp]          ;lit 3d-x
  shl eax,8                     ;les 8 bits inf sont la partie
                                ;décimale
  mov xl_3d,eax                 ;enregistre
  movsx ecx,poly3d[bp+8]        ;forme la différence
  shl ecx,8
  sub eax,ecx
  neg eax
  cdq
  idiv ebx                      ;définit le pas
  mov inc_xl,eax
  movsx eax,poly3d[bp+2]        ;lit 3d-y
  shl eax,8
  mov yl_3d,eax
  movsx ecx,poly3d[bp+10d]      ;forme la différence
  shl ecx,8
  sub eax,ecx
  neg eax
  cdq
  idiv ebx                      ;définit le pas
  mov inc_yl,eax
```

```
  movsx eax,poly3d[bp+4]        ;lit 3d-z
  shl eax,8
  mov zl_3d,eax
  movsx ecx,poly3d[bp+12d]      ;forme la différence
  shl ecx,8
  sub eax,ecx
  neg eax
  cdq
  idiv ebx                      ;définit le pas
  mov inc_zl,eax
  pop edx
  pop ecx
endm
```

Dans le tableau Poly3D se trouvent les coordonnées tridimensionnelles des sommets de la face en cours de traitement. Le registre bp pointe comme avant sur la position des coordonnées courantes du tableau Poly2D (on se trouve encore dans SetNewLineL ou SetNewLineR). Comme ces tableaux sont synchrones, bp permet aussi d'adresser les coordonnées 3D des sommets. La macro charge à chaque fois les variables xl_3D, yl_3D etc avec les valeurs de départ du premier sommet et forme la différence par rapport au deuxième sommet, ce qui donne la longueur du segment dans les trois axes.

Pour ces segments, on applique un principe simple. Pendant le tracé, chaque modification de l'ordonnée y nous fait avancer d'un pas sur le segment. Les différences doivent donc être divisées par le nombre de pas (c'est-à-dire la "hauteur" du segment bidimensionnel en dyl ou dyr). Les valeurs doivent être décalée de 8 bits vers la gauche pour maintenir une précision acceptable. Ainsi, si le segment tridimensionnel a une longueur de sept unités (disons dans la direction z) et si le nombre de pas est de quatre, il faut à chaque fois additionner 1,75, ce qui n'est possible qu'à condition de réserver l'octet inférieur à la partie décimale (dans ce cas inc_zl=01c0h).

A chaque pas, les variables doivent évidemment continuer de progresser dans la ligne d'écran suivante.

```
txt_incl macro                  ;poursuite du comptage à gauche
  push eax
  mov eax,inc_xl                ;avance l'abscisse 3d x
  add xl_3d,eax
  mov eax,inc_yl                ;avance l'ordonnée 3d y
  add yl_3d,eax
  mov eax,inc_zl                ;avance la profondeur 3d z
```

```
  add zl_3d,eax
  pop eax
endm
txt_incr macro                    ;poursuite du comptage à droite
  push eax
  mov eax,inc_xr                  ;avance l'abscisse 3d x
  add xr_3d,eax
  mov eax,inc_yr                  ;avance l'ordonnée 3d y
  add yr_3d,eax
  mov eax,inc_zr                  ;avance la profondeur 3d z
  add zr_3d,eax
  pop eax
endm
```

Ces macros ne font rien d'autre que ce que révèle leur nom : elles accroissent les variables courantes. Inc_xl est ajouté à xl_3D, inc_yl à yl_3D etc. Le bloc de variables xl_3D-yl_3D-zl_3D contient alors toujours les dernières coordonnées 3D du point de départ gauche du segment de remplissage horizontal. La même chose pour le côté droit et pour le point d'arrivée du segment de remplissage.

La procédure hline_texture se trouve dans la macro txt_hline, cette disposition permettant de l'héberger dans un fichier d'inclusion. Elle est appelée à la place de hline dans le cas de surfaces munies de textures et elle remplit fondamentalement la même tâche. Elle trace un segment horizontal de (ax,bx) à (cx,bx) mais cette fois-ci la couleur n'est pas la même pour chaque point, elle est tirée de la texture.

Après sauvegarde des coordonnées, on calcule les déterminants auxiliaires pour le côté droit et le côté gauche ce qui donne lambda1 et lambda2. Le côté droit fournit la valeur initiale car c'est de ce côté que commence le dessin. Le côté gauche livre la valeur finale qui n'est pas mémorisée mais passe dans les variables inc_lambda1 et inc_lambda2. Ces deux variables fonctionnent à l'instar des segments tridimensionnels : à chaque pas vers la droite, lambda1 est augmenté de inc_lambda1 et lambda2 de inc_labda2.

```
txt_hline macro
hline_texture proc near           ;remplace la procedure "hline"
                                  ;avec des textures
.386
  push es
  pusha
```

```
  mov x0,ax                    ;sauvegarde des coordonnées
                               ;pour usage ultérieur
  mov y0,bx
  mov x1,cx
  sub cx,ax                   ;calcule le nombre de points à tracer
   jne zzok2
  inc cx
zzok2:
  mov zz,cx
  mov bp,ligne_sup
  mov bx,ligne_inf
  mov eax,xr_3d[bx]            ;détermine l'abscisse relative x
  movsx ecx,poly3d[2]
  shl ecx,8                    ;ramène au format "virgule fixe"
  sub eax,ecx
  mov d_y,eax
  movsx ecx,w colonne2[0]
  imul ecx                     ;multiplie par Delta2 x
  mov esi,eax                  ;sauvegarde intermédiaire
  mov eax,xr_3d[bp]            ;détermine l'ordonnée relative y
  movsx ecx,poly3d[0]
  shl ecx,8                    ;ramène au format "virgule fixe"
  sub eax,ecx
  mov d_x,eax
  movsx ecx,w colonne2[4]
  imul ecx                     ;multiplie par Delta2 y
  sub eax,esi                  ;forme la différence (D1)
  cdq                          ;prépare la division
  idiv dword ptr D             ;divise par le déterminant principal
  shl eax,8
  neg eax
  mov inc_lambda1,eax          ;sauvegarde pour soustraction
  mov eax,d_x                  ;lit l'abscisse relative x
  movsx ecx,w colonne1[4]
  imul ecx                     ;multiplie par Delta1 y
  mov esi,eax                  ;sauvegarde intermédiaire
  mov eax,d_y                  ;lit l'ordonnée relative y
  movsx ecx,w colonne1[0]
```

```
imul ecx                    ;multiplie par Delta1 x
sub eax,esi                 ;forme la différence (D2)
cdq                         ;prépare la division
idiv dword ptr D            ;divise par le déterminant principal
shl eax,8
neg eax
mov inc_lambda2,eax         ;sauvegarde pour soustraction
```

A l'aide des variables ligne_sup et ligne_inf, on lit les coordonnées nécessaires (en fonction du déterminant principal utilisé) d'abord du côté droit. On forme la différence avec l'origine de la face (point 0). La multiplication par la partie correspondante du déterminant principal (en colonne1 et colonne2) suivie de la soustraction des deux produits permet de définir les déterminants auxiliaires. Après division par le déterminant principal, les résultats sont stockés en inc_lambda1 et inc_lambda2. Les valeurs de lambda à gauche sont ensuite calculées de la même façon :

```
mov eax,xl_3d[bx]           ;définit l'abscisse relative x
  movsx ecx,poly3d[2]
  shl ecx,8                 ;convertit au format virgule fixe
   sub eax,ecx
  mov d_y,eax
  movsx ecx,w colonne2[0]
  imul ecx                  ;multiplie par Delta2 x
  mov esi,eax               ;sauvegarde intermédiaire
  mov eax,xl_3d[bp]         ;définit l'ordonnée relative y
  movsx ecx,poly3d[0]
  shl ecx,8                 ;convertit au format virgule fixe
  sub eax,ecx
  mov d_x,eax
  movsx ecx,w colonne2[4]
  imul ecx                  ;multiplie par Delta2 y
  sub eax,esi               ;forme la différence (D1)
  cdq                       ;prépare la division
  idiv dword ptr D          ;divise par le déterminant principal
  shl eax,8
  neg eax
  mov lambda1,eax           ;lambda1 déterminé
  sub inc_lambda1,eax
  mov eax,d_x               ;lit l'abscisse relative x
```

```
movsx ecx,w colonne1[4]
imul ecx                    ;multiplie par Delta1 y
mov esi,eax                 ;sauvegarde intermédiaire
mov eax,d_y                 ;lit l'oronnée relative y
movsx ecx,w colonne1[0]
imul ecx                    ;multiplie par Delta1 x
sub eax,esi                 ;forme la différence (D2)
cdq                         ;prépare la division
idiv dword ptr D            ;divise par le déterminant principal
neg eax
shl eax,8
mov lambda2,eax             ;lambda2 déterminé
sub inc_lambda2,eax
```

Le résultat de ces calculs de lambda est reporté dans les variables lambda1 et lambda2 et la différence avec les valeurs précédentes est stockée en inc_lambda1 et inc_lambda2. Ces deux valeurs sont alors divisées par la longueur du segment horizontal, c'est-à-dire le nombre de pas :

```
movsx ecx,zz                ;calcule les accroissements des lambda
  mov eax,inc_lambda1       ;lit la longueur totale
  cdq
  idiv ecx                  ;divise par le nombre de pas
  mov inc_lambda1,eax
  mov eax,inc_lambda2       ;lit la longueur totale
  cdq
  idiv ecx                  ;divise par le nombre de pas
  mov inc_lambda2,eax
```

Il suffit désormais d'ajouter à chaque pas inc_lambda1 à lambda et inc_lambda2 à lambda2 pour mettre à jour les valeurs lambda1 et lambda2 de chaque point.

A cet endroit, on prépare également l'adressage des points dans la mémoire d'écran.

```
mov ax,80d                  ;définit l'offset
  mov bx,y0
  mul bx
  mov bx,x0                 ;(x div 4) + y*80
  shr bx,2
```

```
add ax,bx
add ax,vpage
mov di,ax
mov ax,0a000h                    ;charge le segment VGA
mov es,ax
mov cx,x0                        ;masque le plan de début
and cx,3
mov ax,1
shl ax,cl                        ;met à un le bit correspondant
mov b plane+1,al
shl al,4                         ;extension au quartet supérieur
or b plane+1,al
```

Après calcul de l'offset du premier point, le plan de début est déterminé et déposé dans l'octet supérieur de la variable Plane. L'octet inférieur contient la constante 2, ainsi on peut envoyer directement l'ensemble du mot au Timing Sequencer pour sélectionner le plan courant.

Le plan est par ailleurs préparé à une rotation du fait de la présence du masque aussi bien dans le quartet supérieur que dans le quartet inférieur (le plan 3 devient 88h, après rotation 11h qui correspond au plan 0).

```
mov bp,Txt_Nr                    ;lit le numéro de la texture courante
  shl bp,1                       ;2 octets par élément
  mov bx,Txt_Taille[bp]          ;lit l'indication de taille courante
  mov b cs:Taille_Patch+3,bl     ;et modifie le code en conséquence
  mov b cs:Taille_Patch+7,bh
 mov ax,word ptr Txt_Offs[bp]    ;lit l'offset de cette texture
  push ds
  shl bp,1                        ;4 octets par élément
  lds si,dword ptr Txt_Data[bp]  ;pointe sur les données
                                  ;proprement dites
  add si,ax
  mov w cs:Ofs_Patch+2,si        ;et modifie le code en conséquence
  mov dx,3c4h                    ;timing sequencer
  mov ebp,lambda1                ;variables dans registres
  mov esi,lambda2
```

Le numéro de la texture courante (txt_nr) permet de charger la taille de la texture et de modifier le code en conséquence. A cet effet, les opérandes de deux instructions SAR sont écrasés pour limiter l'intervalle des lambdas.

L'offset est également lu dans le tableau correspondant et mis dans le code avec la partie offset du pointeur txt_data.

La boucle de tracé des points est désormais assez brève, ce qui est important du point de vue de la vitesse.

```
lp:                              ;boucle parcourue pour chaque point
  add ebp,inc_lambda1            ;mise à jour lambda1 et 2
  add esi,inc_lambda2
  mov ax,plane                   ;lit le plan
  out dx,ax                      ;et le sélectionne
  mov eax,ebp                    ;détermine l'offset à
                                 ;l'intérieur du graphique de texture
  mov ebx,esi
Taille_Patch:
  sar eax,11d                    ;fixe la taille (modifié)
  sar ebx,11d
   imul eax,320d
  add ebx,eax
Ofs_Patch:
  mov al,ds:[bx+1111h]           ;lit la couleur de la texture
                                 ;(modifié)
  mov es:[di],al                 ;reporte la couleur
  dec cx                         ;diminue le nombre de points
  je hlt_fini                    ;tous les points traités ?
  rol b plane+1,1                ;plan suivant
  cmp b plane+1,11h              ;débordement de plan de 3 à 0 ?
  jne lp                         ;non, on continue
  inc di                         ;sinon on augmente l'offset
  jmp lp                         ;et on continue
hlt_fini:
  pop ds
  popa
  pop es
  ret
hline_texture endp
endm
```

Dans la boucle, le registre EBP contient lambda1 et ESI lambda2. Ces deux valeurs doivent d'abord subir leur incrémentation. Puis le plan courant est

sélectionné et l'offset à l'intérieur de la texture est déterminé par l'intermédiaire de EBP et SI. Auparavant, les deux valeurs de lambda sont décalées à droite par leur facteur de taille pour couvrir l'intervalle de valeurs souhaité.

C'est à l'étiquette Ofs_Patch que se passent les événements décisifs. La couleur est tirée de la texture. L'offset patché dans le code est ajouté à celui qui est calculé et la texture est lue à cet endroit précis. Le tout est écrit dans la mémoire d'écran et la boucle se conclut par une poursuite de la rotation du masque de plan. Si on passe du plan 3 au plan 0, l'adresse de destination est augmentée de 1.

9. Une forme moderne de la protection contre la copie : le contrôle par mot de passe

L'une des curiosités du monde informatique est qu'il existe plus d'ordinateurs en service que de programmes vendus. On en déduit que le nombre de copies pirates est très élevé car tous les utilisateurs ne se contentent pas d'exploiter des programmes du domaine public. Les éditeurs de logiciels essaient en conséquence d'empêcher la diffusion illégale de leurs productions par toutes sortes de méthodes de protection contre la copie. Ils estiment à plusieurs centaines de millions de Francs la perte annuelle due à la copie illicite.

La forme la plus répandue de la protection contre la copie est le contrôle par mot de passe. Cette méthode est très ancienne et reste très simple à implémenter. Parfois, les ordinateurs récents ne démarrent qu'à l'introduction d'un mot de passe gravé en mémoire morte. Si vous avez déjà travaillé sur un ordinateur en réseau, un mot de passe est souvent nécessaire pour accéder à certains ensembles de programmes. Si vous êtes propriétaire d'un modem, vous connaissez les demandes de mots de passe imposées par de nombreuses messageries électroniques.

Dans les sections suivantes, nous allons nous préoccuper des différentes possibilités offertes pr les mots de passe et étudier la meilleure formule pour vous.

9.1. POUR PROTÉGER VOS PROGRAMMES : LE CONTRÔLE PAR MOT DE PASSE

Il existe différentes manières de protéger votre programme. D'abord, il faudra réfléchir à la finalité exacte du contrôle. On peut à cet égard distinguer trois types de préoccupations :

1. Le mot de passe comme justificatif de droit d'accès

Le mot de passe sert à reconnaître si l'utilisateur d'un programme est une personne autorisée ou un intrus. Ce type de mot de passe se trouve par exemple dans les messageries électroniques (n° d'utilisateur), dans les réseaux ou dans les logiciels de gestion exploités par les banques ou les sociétés commerciales. Il est vrai par exemple que les informations comptables ne concernent pas tous les employés d'une société.

2. Le mot de passe comme moyen d'enregistrement

Ce type de mot de passe est souvent utilisé dans le domaine du shareware. Si vous vous enregistrez régulièrement, vous recevez un mot de passe qui vous permet de débrider votre programme en lui donnant les pleines fonctionnalités. Ce procédé est moins courant dans le domaine des jeux mais très fréquent dans celui des applications.

3. Le mot de passe comme protection contre la copie

Ce type de mot de passe se trouve souvent dans les jeux. La méthode se fonde sur le fait que les jeux sont faciles à copier, mais les manuels beaucoup moins. Souvent, la demande du mot de passe n'apparaît pas directement au démarrage du programme mais lorsqu'on a atteint un certain niveau dans le jeu ou lorsqu'on cherche à recharger une situation sauvegardée antérieurement. La méthode est très répandue pour protéger un secteur très sensible à la copie illicite.

D'autres moyens sont devenus plus ou moins désuets, par exemple la disquette de démarrage avec sectorisation spéciale. Il est vrai qu'à une époque où le moindre disque dur fait 340 mégaoctets, il est un peu contraignant d'imposer l'usage d'une disquette.

Les logiciels très coûteux sont parfois protégés par un dongle. Il s'agit d'une petite fiche que l'on installe dans le connecteur série ou parallèle de l'ordinateur et qui contient un mot de passe ou une somme de contrôle. Certaines versions évoluées renferment même des routines d'identification complète. L'inconvénient de ce dispositif apparaît clairement lorsqu'on est conduit à enchaîner plusieurs dongles à la fois qui forment une queue de rat de 30 cm

dans le prolongement de votre sortie parallèle. Non seulement c'est disgracieux mais il se pose alors de véritables problèmes d'encombrement dans les bureaux.

Mais ici nous ne nous préoccuperons pas de dispositifs matériels de contrôle des mots de passe. Seul l'aspect purement logiciel sera étudié dans les pages qui suivent.

Pour les trois types de mots de passe recensés, il existe un principe de base : la sécurité est d'autant mieux assurée que le mot de passe est mieux dissimulé et mieux encodé. Il faut bien réfléchir à l'emplacement de stockage du mot de passe. Choisissez si possible un fichier externe et non pas le fichier EXE. Si vous disposez de plusieurs fichiers, enregistrez le mot de passe dans l'un des plus volumineux. Ne le mettez jamais en début ou en fin de fichier : c'est là qu'il sera recherché en premier par les pirates. Si vous ne travaillez qu'avec un seul fichier de données, prenez bien soin de déposer votre mot de passe vers le milieu de ce fichier. Il sera plus difficile à détecter par un intrus.

Vous veillerez si possible à ne pas enregistrer le mot de passe en clair sous forme de chaîne de caractères lisible. Il faudra mettre à la place une version codée de ladite chaîne. Ajoutez un degré de sécurité supplémentaire en instituant parallèlement une somme de contrôle.

La plupart des contrôles de mots de passe fonctionnent selon le schéma suivant :

- affichage de l'écran de saisie
- lecture du mot de passe tapé par l'utilisateur
- comparaison avec le mot de passe témoin et réaction appropriée

Ce schéma fonctionne très bien et reste simple à programmer : vous le trouverez appliqué dans 95% des cas. Mais si vous avez l'intention de protéger votre propre production logicielle, il faudra procéder autrement. Les différentes étapes devront être disjointes pour rendre le code plus difficile à identifier. Commencez par exemple à afficher l'écran de saisie puis exécutez aussitôt quelques tâches préliminaires : ouverture des fichiers, initialisation des variables, détournement de vecteurs d'interruption, vérification de sommes de contrôle, etc. Ce n'est que lorsque vous aurez procédé à toutes ces tâches que vous lirez le mot de passe. Une fois que vous avez pris connaissance du mot de passe tapé, ne faites pas tout de suite la comparaison avec le mot de passe correct. Exécutez encore si possible quelques actions du type de celles évoquées plus haut. Ce n'est qu'ensuite que vous tirerez de son fichier le mot de passe témoin, que vous procéderez à son décodage et que vous ferez la comparaison.

La méthode présentée a l'avantage de rendre la tâche plus difficile aux pirates éventuels. Si vous enfermez le contrôle du mot de passe dans une procédure isolée, elle sera facile à repérer en raison de l'affichage associé et un déplombeur aura vite fait de transformer le branchement conditionnel produit par la comparaison en un branchement inconditionnel. Si vous appliquez la méthode préconisée, vous n'aurez évidemment pas la certitude d'une protection absolue mais vous aurez établi des obstacles de taille au contournement du contrôle en raison de la difficulté de sa localisation.

Selon le type de mot de passe dont vous avez besoin, vous devrez inclure dans votre programme des routines d'encodage et de décodage ainsi que les séquences de chargement et d'enregistrement associées. Commençons par traiter le type 2. Dans ce cas, le mot de passe est unique. Il est créé par le développeur du programme et mémorisé dans l'un de ses fichiers. Le programme contient en fait une routine qui demande le mot de passe, le lit et teste s'il est bon.

Vous trouverez dans GEnMT.PAS un programme de création d'un mot de passe codé. Le mot de passe souhaité y est d'abord lu. A chaque caractère est ajoutée une clé unique à introduire librement, puis une opération XOR 255 en forme le complément. On stocke alors en premier la clé (1 char) puis le mot de passe (256 char) puis une somme de contrôle (1 word) qui empêche les manipulations dans le fichier. Le programme produit le fichier MOTPASSE.DAT qui contient le mot de passe codé et qui est à incorporer de préférence dans un autre fichier.

```
program cree_fichiermotdepasse;
{Ce programme encode un mot de passe tapé au clavier et le
 stocke dans le fichier MOTPASSE.DAT .
 (c)'94 by MICRO APPLICATION                    Auteur: Boris Bertelsons}

uses crt;

VAR Motdepasse : string;
    fmp : file;
    check : word;
    key : char;

Function encode(mdp : string;add : char) : string;
var li : integer;
begin;
  for li := 1 to 255 do begin;
    mdp[li] := char(255 xor (ord(mdp[li]) + ord(add)));
```

```
  end;

  encode := mdp;
end;
Function Gen_Checksum(mdp : string) : word;
VAR somme : word;
    li : integer;
begin;
  somme := 0;
  for li := 1 to ord(mdp[0]) do begin;
    somme := somme + ord(mdp[li]);
  end;
  Gen_Checksum := somme;
end;

Function decode(mdp : string;add : char) : string;
var li : integer;
begin;
  for li := 1 to 255 do begin;
    mdp[li] := char((255 xor ord(mdp[li])) - ord(add));
  end;
  decode := mdp;
end;

Function Checksum_Ok(mdp : string; Key: char; somme : word) :
boolean;
Var tsomme : word;
    li : integer;
    h : char;
begin;
  tsomme := 0;
  for li := 1 to ord(mdp[0]) do begin;
    tsomme := tsomme + ord(mdp[li]);
  end;
  if somme = tsomme then
    Checksum_Ok := true
  else
    Checksum_Ok := false;
```

```
end;

begin;
  clrscr;
  writeln('Tapez svp le mot de passe à encoder !');
  write('Mot de passe: ');
  readln(Motdepasse);
  writeln('Indiquez la clé (caractère ASCII de votre choix)');
  write('Clé: ');
  readln(Key);
  writeln('enregistrement de l''information dans un fichier ...');
  writeln;
  Check := gen_Checksum(Motdepasse);
  Motdepasse := encode(Motdepasse,Key);
  assign(fmp,'motpasse.dat');
  rewrite(fmp,1);
  blockwrite(fmp,key,1);
  blockwrite(fmp,Motdepasse,256);
  blockwrite(fmp,check,2);
  close(fmp);
  reset(fmp,1);
  blockread(fmp,key,1);
  blockread(fmp,Motdepasse,256);
  blockread(fmp,check,2);
  close(fmp);
  writeln('Relecture de l''information pour contrôle ...');
  writeln('Mot encodé  : ',Motdepasse);
  writeln('Clé         : ',Key);
  Motdepasse := decode(Motdepasse,Key);
  writeln('Vérification : ',Motdepasse);
  If Checksum_Ok(Motdepasse,Key,check) then
    writeln('Checksum O.K.')
  else
    writeln('ATTENTION ! Erreur dans la somme de contrôle !');
  repeat until keypressed;
{  readkey  ;}
end.
```

Pour pouvoir exploiter le mot de passe dans votre programme, vous devrez d'abord vous rendre par un SEEK à l'endroit du fichier où il est entreposé. Chargez ensuite la clé, puis le mot de passe et pour finir la somme de contrôle, comme le montre le programme imprimé ci-dessus.

Utilisez la procédure decode pour décoder le mot de passe et testez par Checksum_Ok s'il a été modifié.

Si vous avez besoin du premier type de mot de passe, vous devrez donner à l'utilisateur la possibilité de choisir librement son mot de passe et de l'introduire. La méthode est semblable à ce que nous avons déjà vu. Vous enregistrerez un mot de passe blanc dans le fichier approprié et l'utilisateur pourra y substituer un autre. Dans ce cas, il importe surtout d'empêcher la découverte du mot de passe et de ne pas l'afficher à la frappe. Il ne faut pas oublier en effet que la protection s'exerce ici davantage à l'encontre des collaborateurs indiscrets qu'à celle des copieurs illicites.

C'est dans cet esprit que le programme MTPASS1.PAS suivant donne un bref exemple de contrôle par mot de passe. Le mot de passe par défaut est DATA BECKER.

```
program motdepasse_type1;

uses crt,design;

VAR motdepasse : string;
    motdepassecheck : boolean;

procedure secret_readln(var s : string);
var c : char;
   li : integer;
begin;
  repeat
    c := readkey;
    if c <> #8 then begin;
      s := s + c;
      gotoxy(28,12);
      for li := 1 to length(s) do write('*');
    end else begin;
      s := copy(s,1,length(s)-1);
      gotoxy(28,12);
      for li := 1 to length(s) do write('*');
```

```
      write(' ');
      gotoxy(wherex-1,wherey);
    end;
  until c = #13;
end;

procedure motdepasse_modifier;
begin;
  save_screen;
  fenetre(10,8,60,7,' Changement de mot de passe ',black,7);
  writexy(13,10,'Veuillez taper SVP votre nouveau mot de passe ');
  writexy(13,12,'Mot de passe: ');
  motdepasse := '';
  secret_readln(motdepasse);
  restore_screen;
  textcolor(7);
  textbackground(black);
end;

Function Gen_Checksum(mdp : string) : word;
VAR somme : word;
    li : integer;
begin;
  somme := 0;
  for li := 1 to ord(mdp[0]) do begin;
    somme := somme + ord(mdp[li]);
  end;
  Gen_Checksum := somme;
end;

Function encode(mdp : string;add : char) : string;
var li : integer;
begin;
  for li := 1 to 255 do begin;
    mdp[li] := char(255 xor (ord(mdp[li]) + ord(add)));
  end;
  encode := Mdp;
end;
```

```
procedure motdepasse_enregistrer;
var fmp : file;
    key : char;
    check : word;
begin;
  Check := gen_Checksum(motdepasse);
  motdepasse := encode(motdepasse,Key);
  assign(fmp,'motdepasse.dat');
  rewrite(fmp,1);
  blockwrite(fmp,key,1);
  blockwrite(fmp,motdepasse,256);
  blockwrite(fmp,check,2);
  close(fmp);
end;

procedure motdepasse_saisir;
begin;
  save_screen;
  fenetre(10,8,60,7,' Contrôle du mot de passe ',black,7);
  writexy(13,10,'Veuillez taper votre mot de passe ');
  writexy(13,12,'mot de passe: ');
  motdepasse := '';
  secret_readln(motdepasse);
  restore_screen;
  textcolor(7);
  textbackground(black);
end;

Function decode(mdp : string;add : char) : string;
var li : integer;
begin;
  for li := 1 to 255 do begin;
    mdp[li] := char((255 xor ord(mdp[li])) - ord(add));
  end;
  decode := Mdp;
end;

Function Checksum_Ok(mdp : string; Key: char; somme : word) :
```

```
boolean;
Var tsomme : word;
    li : integer;
    h : char;
begin;
  tsomme := 0;
  for li := 1 to ord(mdp[0]) do begin;
    tsomme := tsomme + ord(mdp[li]);
  end;
  if somme = tsomme then
    Checksum_Ok := true
  else
    Checksum_Ok := false;
end;

procedure motdepasse_controler;
var fmp : file;
    key : char;
    check : word;
    mottemoin : string;
begin;
  assign(fmp,'motdepasse.dat');
  reset(fmp,1);
  blockread(fmp,key,1);
  blockread(fmp,mottemoin,256);
  blockread(fmp,check,2);
  close(fmp);
  mottemoin := decode(mottemoin,Key);
  if Checksum_Ok(mottemoin,Key,check) and (mottemoin = motdepasse)
      then
    motdepassecheck := true
  else
    motdepassecheck := false;
end;

procedure motdepasse_reagir;
begin;
  save_screen;
```

```
  fenetre(10,8,50,7,'',black,7);
  If motdepassecheck then begin;
    writexy(13,11,'mot de passe correct - Accès autorisé ');
  end else begin;
    writexy(13,11,'Mot de passe ERRONE ! - Accès interdit !');
  end;
  repeat until keypressed; readkey;
  restore_screen;
  textcolor(7);
  textbackground(black);
end;

procedure menu;
var choix : byte;
begin;
  repeat
    clrscr;
    writexy(10,1,'Exemple de programme pour mot de passe de type
          1   (c) ''94 by MICRO APPLICATION ');
    writexy(20,4,'M E N U');
    writexy(20,5,'~~~~~~~');
    writexy(15,6,'1  Modifier le mot de passe ');
    writexy(15,8,'2  Contrôler le mot de passe');
    writexy(15,10,'3  Fin');
    writexy(15,13,'Votre choix: ');
    readln(choix);
    if choix = 1 then begin;
      motdepasse_modifier;
      motdepasse_enregistrer;
    end;
    if choix = 2 then begin;
      motdepasse_saisir;
      motdepasse_controler;
      motdepasse_reagir;
    end;
  until choix = 3;
end;
```

```
 begin;
  menu;
end.
```

L'option 1 du menu vous permet d'introduire un nouveau mot de passe. L'option 2 simule un contrôle du mot de passe. Notez bien la répartition des tâches en trois procédures. Dans votre programme, veillez à ne pas les appeler directement à la queue-leu-leu mais à différents endroits bien disséminés.

Si vous avez l'intention de mettre en service un contrôle de type 3, vous devez disposer non pas d'un mot de passe mais de toute une palette de mots, le contrôle s'effectuant chaque fois sur l'un d'eux. Les jeux appliquent souvent cette méthode. On y demande le nième mot d'une certaine page. Parfois, l'interrogation porte sur un symbole, une couleur. Mais le principe reste le même, seules les procédures de saisie et d'affichage varient.

Dans l'exemple suivant, vous trouverez ce type de contrôle. L'option 1 du menu vous permet d'entrer une liste de 10 mots de passe en association avec des numéros de page fictifs. L'option 2 du menu sert à simuler un contrôle du mot de passe. Le programme sélectionne alors au hasard l'un des dix mots enregistrés dans le fichier. Pour des raisons de place, les mots de passe ne sont pas encryptés ici. Mais il faudrait compléter le programme sinon les mots sont faciles à trouver et à altérer. Vous avez vu précédemment un exemple de routine d'encodage qu'il suffit de reprendre.

```
program motdepasse_type3;

uses crt,design;

Type motdepassetyp = record
       page : byte;
       text  : string[20];
     end;

VAR lmots : array[1..10] of motdepassetyp;
    motp    : motdepassetyp;
    motdepasse : string;
    motdepassecheck : boolean;
    mp_Index : byte;

procedure liste_saisir;
var li : integer;
```

```
begin;
 for li := 1 to 10 do begin;
  save_screen;
  fenetre(10,8,60,9,' Changement de mot de passe ',black,7);
  writexy(13,10,'Veuillez taper le mot de passe n° ');
  write(li,' / 10 de la liste ');
  writexy(13,12,'Page du mot de passe dans le manuel: ');
  readln(lmots[li].page);
  writexy(13,14,'Mot de passe :');
  readln(lmots[li].text);
  restore_screen;
  textcolor(7);
  textbackground(black);
 end;
end;

Procedure liste_enregistrer;
var fmp : file;
begin;
  assign(fmp,'Passtyp3.dat');
  rewrite(fmp,10*sizeof(motdepassetyp));
  blockwrite(fmp,lmots,1);
  close(fmp);
end;

procedure motdepasse_charger(Idx : byte);
var fmp : file;
begin;
  assign(fmp,'Passtyp3.dat');
  reset(fmp,1);
  seek(fmp,Idx*sizeof(motdepassetyp));
  blockread(fmp,motp,sizeof(motdepassetyp));
  close(fmp);
  save_screen;
  fenetre(10,8,50,7,'',black,7);
  writexy(12,10,'Veuillez saisir le mot de passe de la page ');
  write(motp.page);
  writexy(12,12,'mot de passe: ');
```

```
  readln(motdepasse);
  restore_screen;
end;

procedure motdepasse_controler;
begin;
  if motdepasse = motp.text then
    motdepassecheck := true
  else
    motdepassecheck := false;
end;

procedure motdepasse_reagir;
begin;
  save_screen;
  fenetre(10,8,48,7,'',black,7);
  If motdepassecheck then begin;
    writexy(13,11,'Mot de passe correct - Accès autorisé ');
  end else begin;
    writexy(13,11,'Mot de passe ERRONE ! - Accès interdit !');
  end;
  repeat until keypressed; readkey;
  restore_screen;
  textcolor(7);
  textbackground(black);
end;

procedure menu;
var choix : byte;
begin;
  repeat
    clrscr;
    writexy(10,1,'Exemple de programme pour mot de passe de type
           3 (c) ''94 by MICRO APPLICATION ');
    writexy(20,4,'M E N U');
    writexy(20,5,'~~~~~~~');
    writexy(15,6,'1  Saisir une liste de mots de passe');
    writexy(15,8,'2  Contrôler un mot de passe ');
```

```
    writexy(15,10,'3  Fin');
    writexy(15,13,'Votre choix: ');
    readln(choix);
    if choix = 1 then begin;
      liste_saisir;
      liste_enregistrer;
    end;
    if choix = 2 then begin;
      mp_Index := random(10)+1;
      motdepasse_charger(mp_Index);
      motdepasse_controler;
      motdepasse_reagir;
    end;
  until choix = 3;
end;

begin;
  textcolor(7);
  textbackground(black);
  menu;
end.
```

9.2. Programmes en langage évolué au niveau machine

Maintenant que vous avez vu les possibilités de base en matière de contrôle par mot de passe programmé en Pascal, nous allons nous tourner vers la programmation orientée machine. Ce n'est en effet qu'au niveau machine que l'on peut effectuer un contrôle réellement sûr : un programme écrit en Pascal est toujours plus ou moins analysable. Cette constatation ne découle pas de votre style de programmation mais de la façon dont fonctionne le compilateur Pascal.

La structure d'un programme Pascal

Examinons d'abord ce simple programme :

```
program Little_Program;
uses crt;
VAR i : integer;
begin;
  clrscr;
  for i := 5 downto 1 do begin;
    gotoxy(10,6);
    writeln('Veuillez attendre ',i,' secondes');
    delay(1000);
  end;
end.
```

Si nous le désassemblons avec Turbo Debugger par exemple, nous reconnaîtrons sans peine les différentes parties d'un programme en Pascal. Au début, le programme appelle les procédures d'initialisation des unités incluses. Dans ce cas précis, il s'agit de SYSTEM.TPU et CRT.TPU.

```
LITTLE_PROGRAM.7: begin;
  cs:001C 9A0000D862      call   62D8:0000
  cs:0021 9A0D007662      call   6276:000D
  cs:0026 55              push   bp
  cs:0027 89E5            mov    bp,sp
```

Ensuite, c'est au tour de la procédure CLRSCR de l'unité CRT d'être invoquée. Si vous avez compilé le programme sans mettre les informations pour un debugger externe, vous trouvez à la place du symbole CRT.CLRSCR une adresse Far comme dans les initialisations :

```
LITTLE_PROGRAM.8:  clrscr;
 cs:0029 9ACC017662      call   far CRT.CLRSCR
```

On passe ensuite à l'initialisation de la boucle. La variable i se retrouve dans le segment de données à l'adresse [0052h]. Elle reçoit la valeur 5. A la première itération, cette valeur convient et le programme se branche à la ligne source n°10. A l'occasion des itérations suivantes, la mémoire [0052h], c'est-à-dire i, est décrémentée de 1.

```
LITTLE_PROGRAM.9:  for i := 5 downto 1 do begin;
  cs:002E C70652000500   mov    word ptr [0052],0005
```

```
cs:0034 EB04            jmp     LITTLE_PROGRAM.10 (003A)
cs:0036 FF0E5200        dec     word ptr [0052]
```

Dans la boucle se trouve d'abord l'appel de la procédure gotoxy. Elle reçoit comme arguments les coordonnées x et y concernées par l'affichage. Ainsi, les valeurs 000Ah (pour x) et 0006h (pour y) sont déposées sur la pile. La procédure est ensuite invoquée par un appel far :

```
LITTLE_PROGRAM.10:  gotoxy(10,6);
  cs:003A 6A0A          push    000A
  cs:003C 6A06          push    0006
  cs:003E 9A1F027662    call    far CRT.GOTOXY
```

La ligne suivante traduit l'affichage du message à l'aide de la procédure writeln. Cette dernière accepte des arguments en nombre variable : on peut lui communiquer une chaîne unique, ou une combinaison de chaînes et de variables numériques. Dans notre exemple, on voit bien la manière dont le compilateur Pascal effectue la traduction : il appelle une procédure pour chaque argument transmis. On distingue ainsi trois types de traitement. L'affichage de chaînes de texte (strings) est prise en charge par la procédure 0615h. On ne lui communique pas les chaînes proprement dites mais leur adresse qui est empilée. On reconnaît ensuite l'affichage du nombre i [0052h]. Il est copié dans les registres ax:dx et transmis par la pile à la procédure 06D9h de SYSTEM.TPU. A la fin, on trouve encore la procédure de génération de l'avancement d'une ligne. Son incorporation constitue la seule différence avec la procédure write. Elle est située à l'adresse 0582h de SYSTEM.TPU.

```
LITTLE_PROGRAM.11:  writeln('Veuillez attendre ',i,' secondes');
  cs:0043 BF6801        mov     di,0168
  cs:0046 1E            push    ds
  cs:0047 57            push    di
  cs:0048 BF0000        mov     di,0000
  cs:004B 0E            push    cs
  cs:004C 57            push    di
  cs:004D 6A00          push    0000
  cs:004F 9A1506D862    call    62D8:0615
  cs:0054 A15200        mov     ax,[0052]
  cs:0057 99            cwd
  cs:0058 52            push    dx
  cs:0059 50            push    ax
  cs:005A 6A00          push    0000
  cs:005C 9A9D06D862    call    62D8:069D
  cs:0061 BF1200        mov     di,0012
  cs:0064 0E            push    cs
  cs:0065 57            push    di
  cs:0066 6A00          push    0000
```

```
cs:0068 9A1506D862      call   62D8:0615
cs:006D 9A8205D862      call   62D8:0582
```

A l'affichage du message succède l'appel de la procédure Delay. Avant qu'elle ne soit lancée, la pile reçoit le nombre de ms du délai d'attente. Dans notre cas, il s'agit de 03E8h (=1000d) ms.

```
LITTLE_PROGRAM.12:  delay(1000);
  cs:0072 68E803        push   03E8
  cs:0075 9AA8027662    call   far CRT.DELAY
```

A la fin de la boucle signalée par end, le programme teste si [0052h] (i) correspond à la valeur d'abandon (1). Si tel n'est pas le cas, on revient au début de la boucle et on poursuit en décrémentant la variable [0052h].

```
LITTLE_PROGRAM.13:  end;
  cs:007A 833E520001    cmp    word ptr [0052],0001
  cs:007F 75B5          jne    0036
```

Si la valeur finale est atteinte, le programme est arrivé à la dernière ligne, c'est-à-dire à l'instruction end. Le cadre de pile est alors enlevé. Puis le registre ax qui contient le "résultat" du programme est effacé et la procédure de clôture O116h de SYSTEM.TPU est appelée.

```
LITTLE_PROGRAM.14: end.
  cs:0081 C9            leave
  cs:0082 31C0          xor    ax,ax
  cs:0084 9A1601D862    call   62D8:0116
```

Vous avez vu à travers ce petit programme comment on peut repérer de nombreuses procédures significatives de Pascal. Un fichier EXE trahit ainsi son origine Turbo Pascal. A côté des procédures d'initialisation des unités, nous allons en pratique nous intéresser à une procédure du type Delay de l'unité CRT. Examinons-la plus attentivement :

```
CRT.DELAY
  cs:02A8_8BDC          mov    bx,sp
  cs:02AA 368B4F04      mov    cx,ss:[bx+04]
  cs:02AE E313          jcxz   02C3
  cs:02B0 8E064400      mov  es,[SYSTEM.SEG0040]
  cs:02B4 33FF          xor    di,di
  cs:02B6 268A1D        mov    bl,es:[di]
  cs:02B9 A16000        mov    ax,[0060]
  cs:02BC 33D2          xor    dx,dx
```

```
cs:02BE E80500          call    02C6
cs:02C1 E2F6            loop    02B9
cs:02C3 CA0200          retf    0002
cs:02C6 2D0100          sub     ax,0001
cs:02C9 83DA00          sbb     dx,0000
cs:02CC 7205            jb      02D3
cs:02CE 263A1D          cmp     bl,es:[di]
cs:02D1 74F3            je      02C6
cs:02D3 C3              ret
```

Le compteur de ms est d'abord chargé en cx. Autrement dit, si à cet endroit on met :

```
cs:02AA  B90100         mov     cx,0001
cs:02AD  90             nop
```

la boucle ne serait plus parcourue qu'une seule fois. Sinon le nombre d'itérations correspond à ms.

La transparence de la procédure DELAY nous montre l'enjeu véritable de nos efforts. Les procédures aussi importantes que les boucles d'attente ou les contrôles de mot de passe ne devraient pas reposer sur les procédures mises à la disposition par Pascal. Il est recommandé de développer ses propres solutions personnalisées. Mais ceci ne signifie pas forcément qu'il faut réinventer la roue. Il est tout à fait possible de se servir des procédures disponibles. Mais en changeant l'ordre des instructions et en effectuant des modifications bien ciblées, on peut embarrasser bien des programmes d'analyse.

Pour notre procédure d'attente, nous allons ainsi nous détacher complètement de la procédure DELAY et inventer notre propre routine. Elle sera équivalente à la procédure DELAY mais ne sera pas détectée par les scanners de programmes.

La solution proposée ici détourne l'interruption du timer vers une routine personnelle. Pour pouvoir travailler correctement, nous allons reprogrammer l'interruption du timer à 1000 appels par seconde. Pour plus de clarté, nous ferons appel aux routines de l'unité DOS qui traitent des interruptions. Mais au chapitre 10, nous apprendrons à nous en passer pour une meilleure protection contre le débogage.

```
program attente;

uses crt,dos;
```

```
var oldtimer : pointer;
    mscount  : longint;
    msready  : boolean;
    i        : integer;

procedure attenteint; interrupt;
begin;
  dec(mscount);
  if mscount = 0 then msready := true;
  port [$20] := $20;
end;

procedure att(ms : longint);
begin;
 Getintvec(8,oldtimer);
 SetIntVec(8,@attenteint);
 asm
   cli
   mov dx,43h
   mov al,36h
   out dx,al
   sub dx,3
   mov al,169
   out dx,al
   mov al,4
   out dx,al
   sti
 end;
 msready := false;
 mscount := ms;
 repeat until msready;
 asm
   cli
   mov dx,43h
   mov al,36h
   out dx,al
   sub dx,3
   xor ax,ax
```

```
    out dx,al
    out dx,al
    sti
  end;
  SetIntVec(8,oldtimer);
end;

begin;
  clrscr;
  for i := 5 downto 1 do begin;
    gotoxy(10,6);
    writeln('Veuillez attendre ',i,' secondes');
    att(1000);
  end;
end.
```

9.3. UN CONTRÔLE PAR MOT DE PASSE TRÈS PROTÉGÉ

La méthode expliquée ici pour gérer les mots de passe en sécurité ne présente pas toutes les garanties. Mais elle empêchera les pirates en herbe de déplomber la routine. Certes les mots de passe et les indications de page y sont stockés sous forme de constantes. A vous cependant de compléter vos propres programmes en vous inspirant des méthodes présentées plus haut. Pour préserver la clarté des sources, ces extensions ne figurent pas dans les exemples ci-dessous.

Notre nouveau programme ne contient pas d'écueil insurmontable mais toute une série de croche-pieds à l'intention des petits curieux. Le tout commence par l'instruction Mem au début du programme. La mémoire réclamée ne sert à rien mais elle empêche l'exploration du logiciel par la version ordinaire de Turbo Debugger.

Ce piège peut être déjoué en modifiant à l'intérieur de l'en-tête EXE l'offset 0ah qui contient la valeur programmée, par exemple en la diminuant à 0d430h pour 200000 octets. Le programme peut alors être chargé par la version commune de Turbo Debugger mais d'autres infamies attendent les petits curieux. La boucle principale qui demande le mot de passe est maintenue en assembleur. Elle commence par désactiver le clavier en intervenant sur le matériel. On déclenche ensuite une interruption int 3h. Cette interruption

est sans effet si aucun debogueur n'est installé. Mais si TD fonctionne en tâche de fond, le programme est interrompu à cet endroit. Et comme le clavier ne marche plus, ce sera suffisant pour l'utilisateur du TD standard. Il est vrai que la souris peut encore fonctionner mais il suffit de détourner convenablement l'interruption 33h.

Si cet obstacle devait être franchi, il reste à faire le coup de la PIQ que je vais vous présenter un peu plus loin. Une batterie de branchements conditionnels arbitraires contribue à égarer le pirate en herbe. S'il parvenait à déjouer toutes ces embûches, il s'apercevrait alors que tout son parcours serait à recommencer plusieurs fois. S'il ne recule pas devant un tel défi, c'est qu'il ne s'agit pas d'un amateur mais d'un véritable professionnel de la piraterie. Et là, il faut bien avouer que nos armes sont inefficaces.

Mais revenons à notre point de départ : l'instruction d'allocation de mémoire. Si le pirate amateur ne sait pas comment la modifier, il va se servir de TD 386. Ce logiciel ne consomme pas beaucoup de mémoire et présente l'avantage de virtualiser complètement le PC. Ainsi, les astuces du genre détournement d'interruption et désactivation du clavier n'opèrent que sur le PC virtuel et n'ont pas d'influence sur TD. Mais il existe malgré tout une mesquinerie que ne supporte pas la version TD 386 : le mode protégé. C'est pourquoi nous activerons et désactiverons à plusieurs endroits le mode protégé. Cette manipulation n'aura aucune influence sur notre programme mais TD 386 abandonnera la tâche en émettant une erreur d'exception.

Pour parfaire la protection, nous ajouterons encore des appels de procédure indirects. Les procédures du programme Pascal ne sont pas invoquées sous leur nom d'origine mais par l'intermédiaire d'un pointeur qui contient leur adresse. La lisibilité du programme s'en trouve nettement diminuée.

Voici la source d'un programme qui gère les mots de passe avec toutes les protections étudiées :

```
{$F+}
{$M $4000,550000,650000}

program controlemotdepasse;

uses crt,design;

const motsdepasse : array[1..10] of string =
        ('Micro Application','Inspire','PC Interdit','Soundblaster',
         'Démos','Super','Vengeance','Dynamite','Bière','Maison');
      mdpPages : array[1..10] of word =
```

```
        (17,3,29,43,12,21,4,9,13,30);

Var mdp_nr : byte;
    iterations_restantes : byte;
    motdepasse_correct : word;
    New_Pass : string;
    Pchoisir_nouveau_motdepasse : pointer;
    Pdessiner_cadre_de_saisie : pointer;
    Pdemander_motdepasse : pointer;
    Parreter_systeme : pointer;
    variable_bidon1 : word;
    variable_bidon2 : word;

{$L Pwmodul}
procedure boucle_de_saisie; far; external;
procedure choisir_nouveau_motdepasse;
begin;
  mdp_nr := random(10)+1;
  variable_bidon1 := 1;
  variable_bidon2 := 2;
end;

procedure dessiner_cadre_de_saisie;
var mdps : string;
begin;
  str(mdpPages[mdp_nr]:2,mdps);
  asm int 3; end;
  Fenster(20,10,50,4,'Veuillez taper le mot de passe de la page
  '+mdps,black,7);
  variable_bidon1 := 1;
  variable_bidon2 := 2;
  gotoxy(23,12);
end;

procedure demander_motdepasse;
begin;
  readln(New_Pass);
  variable_bidon1 := 1;
```

```
  variable_bidon2 := 2;
  if New_Pass = motsdepasse[mdp_nr] then
    motdepasse_correct := 1
  else
    motdepasse_correct := 0;
end;

procedure arreter_systeme;
begin;
  textbackground(black);
  textcolor(7);
  clrscr;
  writeln('Il aurait mieux valu acheter la version originale...');
  halt(0);
end;

procedure Main_Programm;
begin;
  textbackground(black);
  textcolor(7);
  clrscr;
  gotoxy(20,12);
  writeln('Bienvenue dans le programme principal !');
  gotoxy(20,22);
  write('ENTREE pour terminer ... ');
  readln;
  halt(0);
end;

begin;
  textbackground(black);
  textcolor(7);
  clrscr;
  iterations_restantes := 57;
  Pchoisir_nouveau_motdepasse := @choisir_nouveau_motdepasse;
  Pdessiner_cadre_de_saisie   := @dessiner_cadre_de_saisie;
  Pdemander_motdepasse        := @demander_motdepasse;
  Parreter_systeme            := @arreter_systeme;
```

```
  randomize;
  boucle_de_saisie;
end.
```

Le programme MOTPASSE.PAS fait appel au module assembleur PWMODUL.OBJ suivant :

```
.386p
.MODEL TPASCAL
keyb_off macro
  push ax
  in  al,21h
  or  al,02
  out 21h,al
  pop ax
endm
keyb_on macro
  push ax
  in  al,21h
  and al,0Fdh
  out 21h,al
  pop ax
endm
.DATA
 extrn iterations_restantes
 extrn Pchoisir_nouveau_motdepasse : dword
 extrn Pdessiner_cadre_de_saisie   : dword
 extrn Pdemander_motdepasse        : dword
 extrn Parreter_systeme            : dword
 extrn motdepasse_correct          : byte
 extrn variable_bidon1             : word
 extrn variable_bidon2             : word
.CODE
extrn Main_Programm          : far
public boucle_de_saisie
boucle_de_saisie proc pascal
  keyb_off
 ;coup de la PIQ
  int 3
```

```
  mov  cs:word ptr [@int_21_fonct1],4CB4h
                                  ; Fonction arrêter programme
@int_21_fonct1:
  mov  ah,30h                     ; fonction version de DOS
  int 21h
@saisie_loop:
  keyb_off
  call dword ptr Pchoisir_nouveau_motdepasse
  cmp  variable_bidon1,5
  jbe  @destination_bidon1a
 ;coup de la PIQ
  int 3
  mov  cs:word ptr [@int_21_fonct2],4CB4h
                                  ; fonction arrêter le programme
@int_21_fonct2:
  mov  ah,30h                     ; fonction version de DOS
  int 21h
  mov  cs:word ptr [@int_21_fonct2],30B4h
                                  ; fonction arrêter le programme
  call dword ptr Pdessiner_cadre_de_saisie
  jmp  @destination_bidon1b
@destination_bidon1a:
 ;coup de la PIQ
  int 3
  mov  cs:word ptr [@int_21_fonct2],4CB4h
                                  ; fonction arrêter le programme
@int_21_fonct2a:
  mov  ah,30h                     ; fonction version de DOS
  int 21h
  mov  cs:word ptr [@int_21_fonct2a],30B4h
                                  ; Fonction arrêter le programme
  call dword ptr Pdessiner_cadre_de_saisie
@destination_bidon1b:
  keyb_on
  cmp  variable_bidon2,10
  jbe  @destination_bidon2a
  dec  byte ptr iterations_restantes
                                  ; coup du mode protégé
```

```
    pusha
    cli                          ; désactive les interruptions
    mov eax,cr0                  ; passe dans le mode protégé
    or eax,1
    mov cr0,eax
    jmp PROTECTION_ENABLED       ; efface le canal d'exécution
PROTECTION_ENABLED:
    and al,0FEh                  ; revient au mode réel
    mov cr0,eax                  ; pas de raz du processeur
    jmp PROTECTION_DISABLED      ; efface le canal d'exécution
PROTECTION_DISABLED:
    sti                          ; rétablit les interruptions
popa
  call dword ptr Pdemander_motdepasse
  jmp  @destination_bidon2b
@destination_bidon2a:
  dec  byte ptr iterations_restantes
                                 ; coup du mode protégé
    pusha
    cli                          ; désactive les interruptions
    mov eax,cr0                  ; passe dans le mode protégé
    or eax,1
    mov cr0,eax
    jmp PROTECTION_ENABLED2a     ; efface le canal d'exécution
PROTECTION_ENABLED2a:
    and al,0FEh                  ; rétablit le mode réel
    mov cr0,eax                  ; pas de raz du processeur
    jmp PROTECTION_DISABLED2a    ; efface le canal d'exécution
PROTECTION_DISABLED2a:
    sti                          ; restaure les interruptions
    popa
  call dword ptr Pdemander_motdepasse
@destination_bidon2b:
  cmp  byte ptr motdepasse_correct,1
  je   @demande_OK
  jmp  @demande_pas_OK
@demande_OK:
  call Main_Programm
```

```
@demande_pas_OK:
  cmp byte ptr iterations_restantes,54
  ja  @saisie_loop
  call dword ptr Parreter_systeme
  ret
boucle_de_saisie endp
END
```

10. Protégez votre savoir-faire : les astuces anti-debugging

La méthode de protection la plus sûre contre le piratage est la mise en service d'un dongle. Mais il en est du dongle comme de tout autre bien : son prix est d'autant plus élevé que sa qualité est meilleure. Pour protéger un jeu, cette voie est inabordable : le dongle coûterait plusieurs fois le prix du jeu. Et un dongle à 10 DM n'apporte qu'une sécurité apparente.

Une autre solution consiste à utiliser un CD-ROM. Peu importe que le CD ne puisse pas recevoir de données en écriture, la quantité d'information qu'on peut y mettre est colossale. Mis à part les fraudes de professionnels, le CD présente donc une certaine sécurité. Beaucoup d'utilisateurs y réfléchiront à deux fois avant d'immobiliser 150 mégaoctets sur leur disque dur (entre-temps les compilateurs de Borland devraient atteindre progressivement cet ordre de grandeur...). Si vous émettez le son directement à partir du CD, les copieurs en seront pour leurs frais. Mais il s'agit d'une arme à double tranchant. Le lecteur de CD-ROM est un périphérique qui n'est pas encore universellement répandu. Vous ne pourrez donc pas viser les utilisateurs qui n'en possèdent pas encore.

Si ces deux voies sont exclues pour vous, il ne vous reste plus qu'une seule issue : incorporer une protection contre la copie dans vos programmes. Mais sachez d'avance qu'il n'existe jamais dans ce domaine de garantie absolue. Plutôt que de laisser la porte ouverte à tous les abus, il est cependant préférable de compliquer la vie des voleurs de données.

A une époque où la criminalité économique est en pleine croissance, les astuces anti-debugging gagnent à être connues. Personne ne souhaite se faire voler son savoir. Souvenez-vous de Stack Electronics dont le code a été piqué par Microsoft pour son DOS 6.0. Les suites juridiques en sont bien connues.

Les astuces présentées ci-après ne vous protégeront pas à long terme contre les pirates professionnels mais elles constituent une bonne intimidation à l'encontre des déplombeurs en herbe. Elles sont d'ailleurs utilisées par nombre de logiciels commerciaux.

10.1. Analyse des programmes

Pour analyser des programmes il existe toute une gamme d'outils. Nous allons vous présenter quelques-uns des meilleurs d'entre eux.

Si on vous parle d'outil, vous penserez peut-être à des programmes universels du type PC Tools ou Norton Utilities. Mais ce n'est pas de ceux-là que nous allons parler. Nous allons plutôt vous présenter des programmes destinés à vous faciliter la vie avec votre ordinateur. Nous évoquerons ainsi un représentant de chacune des catégories suivantes : programme de décompactage, débogueur et éditeur hexadécimal.

Enlevez le camouflage - Les programmes de décompactage

Les programmes comme PKLITE (tm) permettent de compacter des fichiers EXE et COM qui restent cependant exécutables. Ils prennent alors moins de place sur le disque dur. Le temps de décompactage est négligeable. Mais cette méthode présente un grand danger. Car un programme peut être infecté par un virus. Et une fois qu'il est compacté, aucun scanner de virus ne peut plus détecter l'infection!

Si vous voulez inspecter le fonctionnement d'un programme (bonjour le droit européen), il vaut mieux disposer du programme sous sa forme directement exécutable. Pour annuler les effets du compactage, il existe des programmes de décompactage. Issus la plupart du temps du domaine du shareware, ils présentent des performances souvent étonnantes. Aucun programme n'est à l'abri de leur pouvoir. Nous allons vous présenter le meilleur décompacteur du moment : il s'agit du programme UNP (Unpack) disponible dans sa version 3.20 a au moment de la mise sous presse de cet ouvrage.

UNP est capable de décompacter tous les programmes compactés avec l'un des logiciels de compactage courants. On l'invoque par

```
UNP commande [options] fichier-source [fichier-destination]
```

Le paramètre "commande" peut avoir les valeurs suivantes :

e - décompacter un fichier (expand compressed file)

Cette commande lance le décompactage du fichier indiqué. Si aucun nom de fichier n'est précisé, tous les fichiers du répertoire courant sont décompactés. Si vous ne précisez pas de paramètre, UNP réagit comme si vous aviez tapé e.

c - convertir en fichier COM (convert to com)

Certains fichiers EXE peuvent être transformés en fichiers COM. Dans ce cas ils ne doivent pas contenir de segment de données ! Si tel est le cas, le paramètre c déclenche la conversion. Avantage : les fichiers deviennent plus petits.

i - afficher des informations (info only)

Si vous souhaitez simplement avoir des informations sur le programme à examiner ("tel fichier est-il compacté ?") alors appelez UNP avec le paramètre i. Vous obtenez alors les informations que vous auriez eues avec le paramètre e mais sans déclencher de décompactage.

l - charger et sauvegarder (load and save)

Cette option est utile si vous voulez retirer du programme des données d'en-tête inutiles. Le programme compacté est chargé en mémoire puis réenregistré sous forme compactée. A utiliser conjointement avec les options -h et -k-.

s - rechercher les fichiers compactés (search for compressed files)

Cette option affiche une liste de tous les fichiers compactés dans un répertoire donné. Tous les fichiers conformes au masque indiqué sont analysés. Avec la liste des fichiers compactés UNP indique le programme de compactage qui a servi à les compacter.

x - convertir en fichier EXE (convert to EXE)

Cette option permet de transformer les fichiers COM en fichiers EXE. L'opération se justifie parce que certains programmes de compactage n'opèrent que sur des fichiers EXE. Si vous voulez compacter un programme COM, convertissez-le au préalable en le soumettant à UNP avec cette option, puis procédez au compactage.

Il existe encore d'autres options pour faire fonctionner UNP. Les options suivantes sont disponibles dans la version 3.12 :

-? - afficher une aide (help)

UNP affiche une liste de tous les paramètres de commande et options. ***Attention :*** tous les autres paramètres ou options indiqués sont alors ignorés.

-a - faire automatiquement une nouvelle tentative (automatic retry)

Il existe des fichiers compactés plus d'une fois ou "protégés" par d'autres programmes par exemple le Central Point Anti Virus Packet. Avec cette option, le fichier est automatiquement testé à l'issue du décompactage pour voir s'il est suffisamment décompacté. S'il ne l'est pas, le processus est recommencé.

-b - créer un fichier de backup (make .BAK file)

Je vous recommande vivement l'usage constant de cette option si vous avez assez d'espace sur votre disque. Il pourrait arriver qu'un fichier décompacté ne fonctionne plus ! Il est alors réconfortant de disposer d'une copie de sécurité avec l'extension .BAK.

-c - demander confirmation avant décompactage (ask for confirmation before decompressing)

Si vous choisissez cette option (en association avec des jokers), UNP vous demande confirmation avant de procéder au décompactage d'un fichier

-h - enlever les données d'en-tête (remove irrelevant header date)

Option à utiliser après compactage d'un programme. Elle permet de retirer les avis de copyright laissés par le programme de compactage et qui ne servent à rien.

-i - ne pas intercepter l'interruption 21h (do not intercept 21h calls)

Normalement l'interruption 21h de DOS est exploitée par UNP pour tester si le programme traité fonctionne correctement. Mais il existe quelques programmes résidents (par exemple Turbo Debugger) qui peuvent poser problème et conduire au plantage du système. L'option citée permet de désactiver cette vérification interne.

-k - signature PKLite (pklite signature)

Option permettant d'ajouter ou de retirer la signature PKLite. Si vous compactez un programme, il convient évidemment de l'enlever (encore des octets inutiles !).

Si vous écrivez -k- ou -k, la signature est retirée. Avec -k+ la signature PKLite est ajoutée, ce qui ne présente d'intérêt que si vous voulez être sûr qu'on puisse décompacter plus tard votre programme. Cette option est active par défaut : vous pouvez donc ne pas la mentionner.

Si vous utilisez le paramètre -k?, UNP vous avertit chaque fois qu'il trouve la signature.

-l - utiliser systématiquement Loadfix (always use loadfix)

Normalement Unpack ne remplit pas les 64 premiers Ko de la mémoire centrale pour pouvoir traiter des fichiers plus grands. Mais en cas de nécessité (pour les fichiers compactés par EXEPACK), UNP recharge le programme en utilisant la zone en question. Si vous souhaitez maintenant décompacter plusieurs fichiers, utilisez cette option pour que le programme ne soit pas rechargé sans cesse.

-O - écraser le fichier de sortie (overwrite output file)

Si vous souhaitez effacer systématiquement le fichier de destination au cas où il existerait déjà, indiquez ce paramètre. Sinon UNP demande à chaque fois la permission de procéder à l'écrasement.

-p - aligner sur une page les données d'en-tête (align header data on a page)

Sert à augmenter la taille de l'en-tête EXE jusqu'à ce qu'il occupe exactement une page de 512 octets. Normalement le chargement du fichier EXE est alors plus rapide.

-r - enlever les données de recouvrement (remove overlay data)

Certains programmes contiennent des données d'overlay. La présente option permet de les enlever. Mais la plupart du temps il faut éviter cette manoeuvre car les fichiers en question contiennent des données importantes.

-u - mettre à jour l'heure et la date du fichier (update file date / time)

Normalement lorsque UNP crée un fichier décompacté, il lui associe la date et l'heure du fichier compacté. Cette option permet au contraire d'associer au nouveau fichier la date et l'heure courantes.

-v - autres informations (verbose)

Cette option sert à obtenir des informations supplémentaires sur le processus de décompactage. Mais ces informations ne sont généralement pas très utiles.

Si vous voulez par exemple décompacter le programme HELLO.EXE, il faut écrire :

```
UNP e -a -o HELLO.EXE
```

Pour retirer d'un fichier compacté par vos soins toutes les informations inutiles, tapez :

```
UNP 1 -k- -h HELLO.EXE
```

Pas à pas - Turbo Debbuger

Le programme Turbo Debugger fourni avec Borland C(++) et Pascal est l'un des meilleurs débogueurs disponibles à l'heure actuelle. Avec lui vous pouvez examiner pas à pas à la fois vos codes sources et des fichiers EXE ou COM tout constitués. Vous pouvez également poser des points d'arrêt : le programme s'exécute jusqu'aux endroits ainsi désignés puis vous pouvez poursuivre son exploration pas à pas.

Nous allons expliquer dans les pages qui suivent comment analyser un programme avec ce puissant outil et comment surveiller certaines variables. Les informations très spécialisées comme celles qui concernent par exemple le débogage orienté objet se trouvent expliquées dans l'excellent manuel de Borland.

Appel de Turbo Debugger

Vous devez d'abord décider quelle est la version de Turbo Debugger que vous comptez mettre en oeuvre. La version ordinaire fonctionne avec n'importe quel gestionnaire de mémoire mais elle consomme beaucoup de mémoire. L'alternative consiste à utiliser TD386. Vous devez alors inclure la ligne

```
Device=TD386.sys
```

dans votre fichier Config.sys. Le gestionnaire en question ne supporte malheureusement pas la compagnie de EMM386 ou QEMM. Si votre programme exploite de la mémoire EMS, vous ne pourrez pas donc pas faire fonctionner ce débogueur. Sinon il présente l'avantage d'autoriser le débogage de programme très volumineux et gourmands en mémoire.

Les paramètres de la ligne de commande

Vous pouvez configurer Turbo Debugger à chaque appel en tapant des paramètres appropriés en ligne de commande. Mais certaines options sont systématiquement nécessaires : vous pouvez alors les enregistrer dans un fichier de configuration. Ce fichier est créé sous Turbo Debugger par la commande **Option - Save Options**. Donnez lui un nom personnalisé facile à retrouver par ex maconfig.td. Le paramètre -c de la ligne de commande

permet de charger directement le fichier de configuration désigné à chaque démarrage de Turbo Debugger. Vous taperez ainsi :

```
TD386 -c MACONFIG.TD LeProg
```

"LeProg" indiquant le programme à déboguer.

Les options contenues dans le fichier de configuration peuvent être complétées ou modifiées par d'autres paramètres en ligne de commande. Les possibilités suivantes s'offrent à vous :

-d - gestion de l'écran - do - utilisation de deux écrans simultanés

Si vous disposez encore d'une vieille carte Hercules avec son moniteur vous pouvez demander au débogueur de représenter le programme traité sur le moniteur Hercules et les affichages graphiques sur le moniteur VGA. Vous avez alors l'avantage d'avoir sans cesse devant vos yeux les sorties de votre programme.

-dp - commutation de page (surtout en mode texte)

Avec cette option Turbo Debugger exploite virtuellement deux pages d'écran. Ceci ne fonctionne qu'en mode texte et uniquement lorsque le programme n'utilise pas lui-même plusieurs pages d'écran ou jongle avec l'adresse de début de la mémoire vidéo. Si vous effectuez un débogage en mode graphique, vous obtiendrez de légères erreurs lorsque vous passerez d'une page à l'autre. La plupart du temps ces erreurs sont supportables. Comme ce mode est le plus rapide, il est à privilégier a priori.

-ds - permutation d'écran

Si vous choisissez ce mode, Turbo Debugger stocke dans un buffer le contenu de l'écran du programme à déboguer avant de rétablir l'écran du débogueur. Il est restauré en cas de saisie utilisateur. Cette permutation nécessite quelque délai et devient agaçante en mode pas à pas. Ne sélectionnez donc ce mode que si vous en avez absolument besoin.

-k - enregistrement du clavier

Voici une option relativement utile. Elle permet d'enregistrer toutes les frappes de l'utilisateur, qu'elles s'adressent au programme ou au débogueur. Vous pouvez ainsi revenir à un endroit antérieur du programme si vous avez poussé trop loin la session de débogage. Les saisies effectuées sont enregistrées dans un fichier.

-l - démarrage en mode assembleur

Si vous lancez le débogueur en mode assembleur, la fenêtre affichée n'est pas celle du code source mais celle du processeur. Les processus normalement effectués au démarrage ne sont donc pas exécutés directement, ce qui veut dire que vous pouvez suivre le chargement du programme.

-m - fixation de la taille du tas

Pour ses besoins internes Turbo Debugger utilise un tas qui s'étend normalement sur 18 Ko. Si vous tombez en manque de mémoire, vous pouvez abaisser la taille du tas jusqu'à 7 Ko. Vous indiquerez la taille souhaitée (en Ko) directement à la suite de la lettre m.

-v - gestion de la mémoire d'écran
-vg - sauvegarde de l'intégralité de la mémoire d'écran

Cette option permet de sauvegarder entièrement les 8 Ko de la mémoire d'écran. Elle consomme certes 8 Ko supplémentaires de la mémoire d'écran mais elle permet de déboguer des programmes qui exploitent cette mémoire et dont l'affichage serait détruit autrement.

-vn - pas de mode 50 lignes

Si vous êtes sûr de ne pas avoir besoin du mode 50 lignes, vous pouvez mettre en service ce paramètre pour gagner un peu de mémoire.

-vp - sauvegarde de la palette de couleurs

Certains programmes modifient la palette VGA : il est recommandé alors d'activer cette option. La palette sera sauvegardée et en examinant pas à pas le programme vous aurez toujours les bonnes couleurs.

-y - fixation de la taille de l'overlay de Turbo Debugger

Turbo Debugger stocke les parties de programmes dont il n'a pas besoin dans un overlay. Vous pouvez adapter la taille de cet overlay en fonction de la mémoire demandée par le programme à déboguer. Si votre programme est très vorace en mémoire, sélectionnez le minimum par -y20. Vous disposez alors d'un grand espace de mémoire mais en contrepartie les délais de chargement s'accroissent. Si le programme à déboguer a besoin de fort peu de mémoire, vous pouvez à l'aide du paramètre -y200 sélectionner le maximum d'espace possible (200 Ko). Le débogueur est alors entièrement maintenu en mémoire vive, et le travail s'en trouve accéléré d'autant.

Recherche ciblée avec Turbo Debugger

Comme le montrera l'exemple de l'entraîneur, il n'est pas très intéressant pour nous d'exploiter Turbo Debugger pour déboguer un programme dont le code source est disponible. Ce qui nous intéresse, c'est de trouver certaines variables ou certaines instructions à l'intérieur du programme analysé. Le droit européen vous y autorise mais il vous interdit de faire des modifications sur les fichiers COM ou EXE ou les autres fichiers associés au programme.

A partir du programme suivant écrit en Pascal nous allons voir comment procéder pour trouver par exemple les variables Vie ou Points dans un fichier EXE.

```
{$A+,B-,D-,E+,F+,G+,I+,L-,N-,O-,P-,Q-,R-,S+,T-,V+,X+,Y-}
{$M 16384,0,655360}
program show_how_to_debug;

uses crt;

Var inputvar : word;
    vie  : byte;
    facteur : real;

Procedure init_variables;
begin;
  vie  := 4;
  facteur := 0.27
end;

Procedure saisie_utilisateur;
begin;
  textcolor(14);
  gotoxy(10,3);
  write('                                                    ');
  gotoxy(10,3);
  write('Tapez la nouvelle valeur de test: ');
  readln(inputvar);
end;

Procedure calcul_arbitraire;
begin;
```

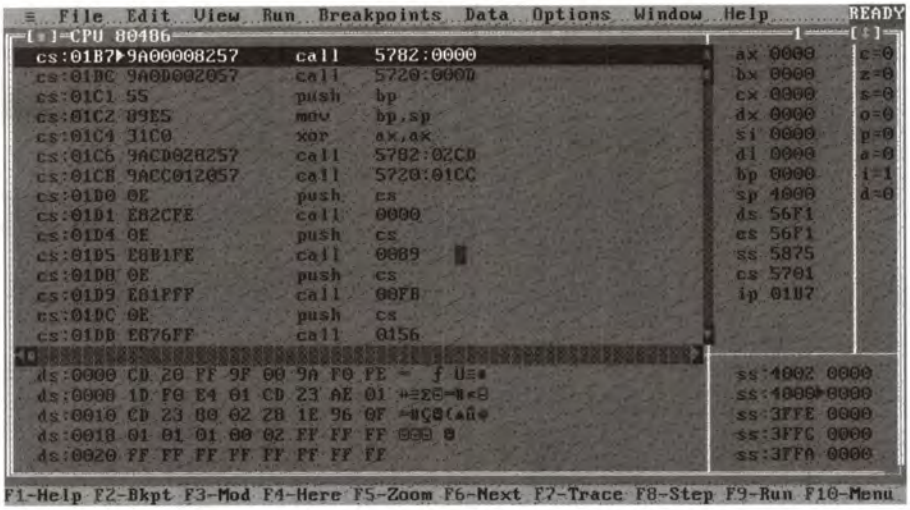
```
  if (inputvar * facteur) < 10 then
    dec(vie);
end;

Procedure affiche_etat;
begin;
  textcolor(15);
  gotoxy(10,10);
  write('Saisie : ',inputvar:5,'      ==>    Vie : ',vie:3);
end;
begin;
  clrscr;
  init_variables;
  repeat
    saisie_utilisateur;
    calcul_arbitraire;
    affiche_etat;
  until vie = 0;
end.
```

Le programme lit un nombre tapé par l'utilisateur. S'il est inférieur à 38, une "vie" est retirée. Le processus se répète aussi longtemps qu'il reste des vies.

Après avoir compilé le programme (qui se trouve sur le CD d'accompagnement), vous pouvez lancer Turbo Debugger. Tapez :

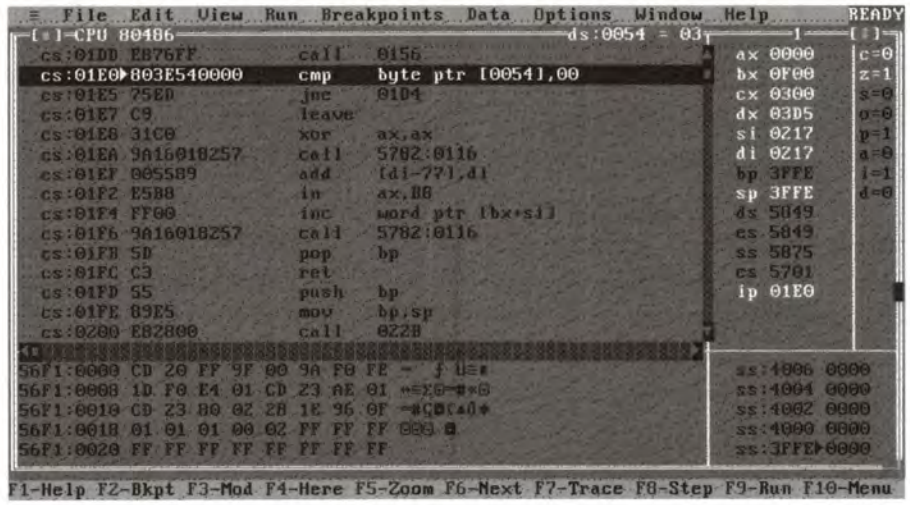
```
TD td_test
```

et vous verrez apparaître l'écran suivant :



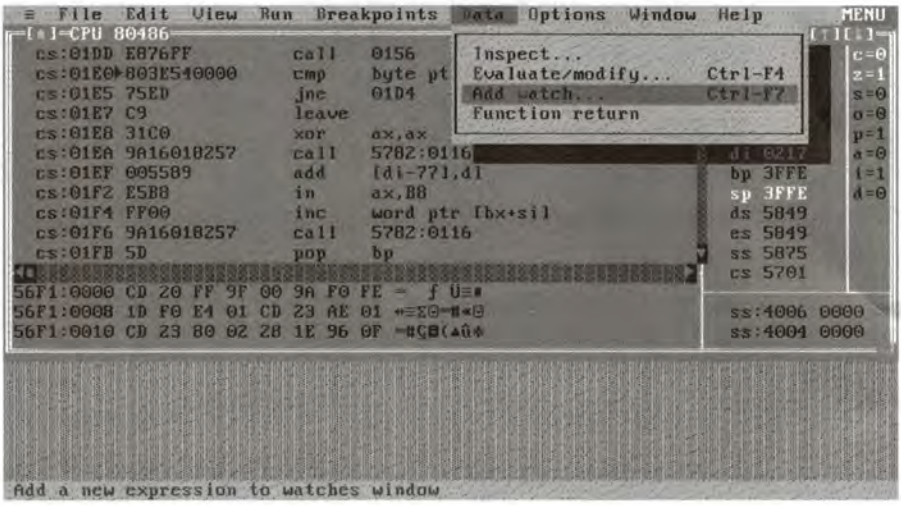
Programme chargé

Au début vous observez les procédures d'initialisation des unités. Parcourez ensuite le programme pas à pas en appuyant sur **F8**. Vous tomberez sur l'appel de différentes sous-procédures. Pour le moment ne les approfondissez pas, il s'agit simplement de deviner ce qu'elles font. Vous constatez que vous vous trouvez dans une boucle qui commence par lire des données tapées. La signification de la procédure qui suit n'est pas reconnaissable de prime abord. La troisième affiche le nombre tapé et le nombre de vies restantes. On relève ensuite une comparaison très instructive. Le test vérifie si la variable [0054] est nulle. Si tel n'est pas le cas, la boucle est recommencée une nouvelle fois.



Examen d'une variable

Examinons de plus près cette variable puisqu'elle paraît jouer un rôle important. Nous la reporterons donc dans la fenêtre de suivi qui sert à surveiller des zones de mémoire désignées à l'avance.

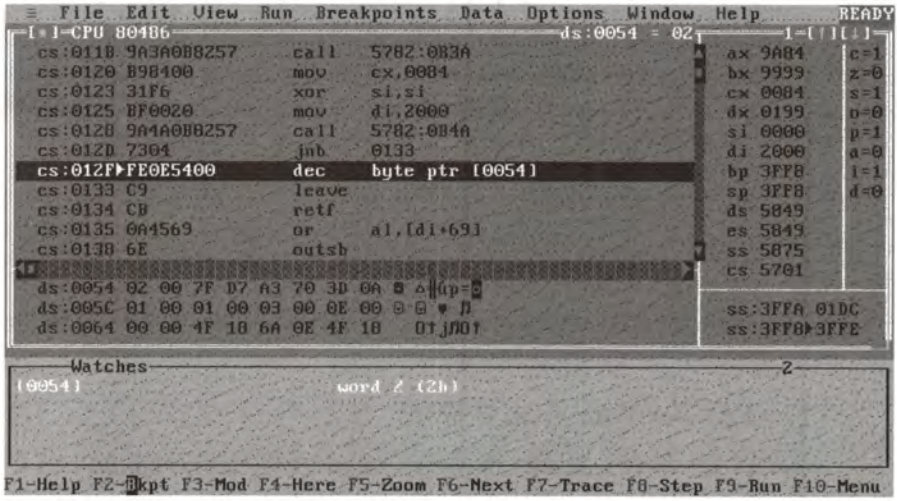


Création d'une variable watch

Nous poursuivons à présent l'exécution du programme. Vous constaterez que la variable suivie possède la même valeur que la variable vie du programme. On suspecte donc qu'elles sont identiques. La signification de la deuxième procédure, non interprétée jusqu'ici, devient claire. Si une valeur inférieure à 38 est introduite, la variable pistée est décrémentée.

Nous savons à présent où se trouve le mauvais génie qui diminue nos vies. La prochaine fois que vous tomberez sur la deuxième procédure, ne la survolez pas mais examinez-la en détail pas à pas. A cet effet appuyez sur la touche **F7** au moment où vous vous trouvez sur l'appel. Vous passez alors dans la sous-procédure. Reprenez la touche **F8** pour l'exécuter pas à pas. Vers la fin vous découvrirez l'instruction qui décrémente les vies :

```
dec byte ptr[0054]
```



Chaîne d'instruction détectée

Pour vérifier que les vies sont réellement décrémentées à cet endroit, vous pouvez remplacer les 4 ocets de l'instruction dec par 5 NOP (non operates). Pour entrer une autre instruction, appuyez sur la barre d'espacement et tapez l'instruction désirée. Veillez bien à ce que la longueur de la nouvelle instruction soit égale à celle de l'ancienne qu'elle recouvre.

Saisie de nouvelles instructions

Attention : la substitution d'instructions NOP à des fins de test est à la limite de la légalité.

Vous constaterez d'abord que la variable [0054] est bien effectivement la variable vie recherchée. Si vous rédigez un "entraîneur", vous devez soit modifier l'emplacement mémoire où est stocké la variable soit l'endroit dans la RAM où la vie est décrémentée. Attention ! Ne touchez pas au fichier EXE ! Ne recherchez pas le code programme de DEC avec un éditeur hexadécimal pour y susbtituer des NOP : c'est strictement interdit et vous risquez des poursuites juridiques.

A côté des témoins de suivi et de l'exécution pas à pas, Turbo Debugger vous offre encore toute une série d'autres fonctions fort utiles. Vous pouvez par exemple arrêter l'exécution du programme à certains endroits et effectuer des recherches et des branchements.

F2 - Poser un point d'arrêt

La touche **F2** permet de poser un point d'arrêt (breakpoint). L'exécution du programme est interrompue et Turbo Debugger reprend le contrôle dès qu'un certain point du programme est atteint. Placez-vous à l'endroit du programme où vous comptez installer un point d'arrêt et appuyez sur la touche **F2**.

ALT-F2 - Poser un point d'arrêt à un certain emplacement mémoire

La commande **ALT-F2** sert à poser un point d'arrêt à un endroit précis de la mémoire. Vous devez indiquer le segment et l'offset du point souhaité. Il n'est pas obligé de se trouver à l'intérieur du programme mais peut être situé n'importe où en mémoire. Cette option est très utile pour déboguer des interruptions ou des gestionnaires. Il faut alors bien maîtriser ce que l'on fait.

F9 - Exécuter le programme

Pour lancer le programme chargé, appuyez simplement sur la touche **F9**. La plupart du temps vous pouvez interrompre le programme débogué à un endroit critique en appuyant sur **CTRL-PAUSE** puis en opérant pas à pas.

F4 - Exécuter jusqu'à la position courante

Si vous vous trouvez à un certain endroit du programme et si vous voulez le réexécuter depuis le début jusqu'à cet endroit, il n'est pas nécessaire de relancer un pas à pas. Il est recommandé dans ce cas de faire une réinitialisation par **CTRL-F2** et de lancer l'exécution jusqu'à l'emplacement courant en tapant **F4**.

ALT-F9 - Exécuter le programme jusqu'à un emplacement mémoire donné

Cette combinaison de touches à un effet similaire à celui de **F4**. Vous pouvez entrer directement un emplacement du programme (segment et offset) qui marquera le terme de l'exécution (à partir de la position courante).

ALT-F5 - Afficher l'écran utilisateur

Pendant le débogage, il est toujours utile de connaître les derniers affichages de l'écran utilisateur ou les saisies qui ont suscité l'interruption à l'emplacement d'un point d'arrêt. Turbo Debugger vous permet précisément d'afficher l'écran utilisateur en appuyant sur la touche **ALT-F5**. Si l'application a fonctionné en mode graphique, ce mode est activé automatiquement. Mais la reprogrammation de certains registres VGA n'est pas prise en compte. Si votre programme exploite le mode X de la carte VGA, vous observerez quelques bizarreries sur l'écran.

CTRL-S - Rechercher une expression

Si vous recherchez une certaine expression dans le programme en cours d'examen, par exemple la décrémentation de telle variable, vous pouvez procéder en appuyant sur **CTRL-S**. Tapez simplement l'expression concernée dans la fenêtre de saisie qui s'ouvre à l'écran, par exemple :

```
dec byte ptr [0054]
```

Turbo Debugger commence la recherche à la position courante. Attention aux expressions trop courtes. Il se peut qu'une partie d'instruction soit interprétée comme l'expression cherchée. Dans ce cas faites défiler l'écran pour vérifier qu'il s'agit bien d'une instruction entière et non partielle.

CTRL-G - Aller à un emplacement donné

Turbo Debugger indique toujours la position courante à gauche dans la fenêtre CPU . Si vous avez repéré un endroit intéressant dans le programme, prenez-en note. Par la suite en appuyant sur **CTRL-G** vous pourrez vous y rendre directement. La vérification du programme pourra s'en trouver facilitée.

CTRL-C - Retourner à la procédure appelante

Si vous êtes entré dans une procédure pour en traquer le code et s'il s'avère finalement que l'objet de votre recherche ne s'y trouve pas, il n'est pas nécessaire de poursuivre l'exécution pas à pas jusqu'à ce que la procédure en question soit terminée. Avec **CTRL-C** vous pouvez retourner à l'endroit où a été appelée la procédure et par **F8** en sauter le détail.

Maintenant que vous connaissez les fonctions de base de Turbo Debugger, vous devriez être capable d'analyser un programme quelconque. Faites par exemple un essai avec le mini-jeu RAIDER qui se trouve sur le CD et sert d'exemple à la programmation d'un "trainer". Ce programme peut être librement exploré et modifié par vos soins.

Les éditeurs hexadécimaux

Un éditeur est dit hexadécimal lorsqu'il présente les fichiers non pas sous la forme ordinaire des caractères ASCII mais sous la forme de codes internes au système. Ces derniers sont des nombres hexadécimaux, d'où le nom d'éditeur hexadécimal.

A quoi sert un éditeur hexadécimal ?

A quoi peut bien servir un éditeur de ce type ? C'est qu'il existe des fichiers qui ne nous intéressent pas en raison du texte qu'ils contiennent mais en raison des nombres qui y sont stockés. Le meilleur exemple est celui des situations de jeu sauvegardées. Si dans une simulation commerciale vous manquez d'argent ou si dans un simulateur de vol les armes vous font défaut, vous pouvez sauver l'état courant du jeu et l'inspecter avec un éditeur hexadécimal. Notez auparavant la somme dont vous disposiez. Cette somme est à convertir en hexadécimal. Supposons par exemple que vous possédiez 1000 pièces d'or, ce qui correspond à 03E8 en hexadécimal. Le processeur Intel mémorise les nombres en partant de l'octet inférieur vers les octets supérieurs. On obtient E803h. Avec l'éditeur hexadécimal vous allez vous mettre à la recherche précise de ce nombre. Dès que vous l'avez trouvé, remplacez-le à titre d'exemple par 9999h et relancez le jeu. Après avoir rechargé la situation sauvegardée et être retourné à l'endroit du jeu que vous avez quitté, vous serez l'heureux propriétaire de 39321 pièces d'or.

Si cette méthode ne fonctionne pas, c'est peut-être que la somme en question apparaît encore ailleurs. Vous pouvez de la même façon modifier à votre avantage les traits de caractère, le nombre de vies, de munitions, d'armes supplémentaires, etc.

Le procédé s'applique à tous les fichiers d'un ordinateur. Vous pouvez rechercher ainsi toute combinaison d'octets et y substituer une autre. Ce dont vous aurez besoin, c'est d'un bon éditeur hexadécimal.

L'éditeur hexadécimal Hexcalibur

L'un des meilleurs éditeurs actuels s'appelle Hexcalibur. Il ne vous permet pas seulement d'écrire par-dessus des données existantes mais aussi d'ajouter des données, d'en retirer ou d'en faire des copies. Pour lancer l'éditeur, indiquez dans la ligne de commande le nom du fichier à éditer, par exemple :

```
HC SAVEGAME.01
```

Appuyez sur une touche quelconque et vous vous retrouverez dans l'éditeur. Le mode insertion est activé spontanément. Pour éditer la sauvegarde d'une situation de jeu, il est préférable d'activer le mode surfrappe en appuyant sur la touche **Ins**. Vous pouvez vous déplacer dans l'éditeur avec les touches de

direction. Pour passer de la fenêtre hexadécimale à gauche à la fenêtre ACII à droite et vice-versa, il suffit d'actionner la touche **F2.**

Voici un résumé de l'action des touches :

Touches	Signification
Touches de direction	permettent de se déplacer dans la fenêtre active
PGPREC **PGSUIV**	sert à monter où à descendre d'une page d'écran. Une page d'écran contient exactement 256 caractères.
POS1	permet de revenir au début de la ligne courante
FIN	pour aller à la fin de la ligne courante.
ALT-B ou **ALT-0**	L'une ou l'autre de ces combinaisons de touches vous renvoie au début du fichier chargé dans l'éditeur.
ALT-1 à **ALT-9**	Ces touches accélèrent le déplacement dans le fichier. Avec **ALT-1** vous êtes renvoyé à la position qui correspond à 10 % du fichier, avec **ALT-2** à 20 % etc.
ALT-E	provoque un saut à la fin du fichier
ALT-G	sert à aller à un emplacement donné du fichier. Indiquez le secteur et le numéro du premier octet puis tapez **ENTREE**.
F1	affiche un écran qui rappelle les commandes de base de l'éditeur
F2	est un commutateur qui permet de passer de la fenêtre hexadécimale à la fenêtre ASCII et vice-versa.
INS	active le mode insertion ou le mode surfrappe
SUPPR	efface le caractère au-dessus du curseur courant. Ce caractère est effacé aussi bien dans la fenêtre ASCII que dans la fenêtre hexadécimale. Attention : la perte d'un seul caractère peut avoir des conséquences fatales sur le fonctionnement du programme associé.
ALT-A	sert à marquer le début d'un bloc de sélection, pour faire une des opérations suivantes
ALT-C	copie le bloc sélectionné
ALT-M	permet de déplacer le bloc sélectionné
ALT-D	provoque l'effacement du bloc sélectionné. Soyez prudents : quand c'est parti, c'est vraiment parti !
ALT-P	permet d'insérer le bloc sélectionné

Touches	Signification
ALT-F	C'est l'une des commandes les plus intéressantes de l'éditeur. Elle lance la recherche d'une chaîne de caractères. A la suite de **ALT-F** tapez le motif à rechercher : le fichier est scruté depuis la position courante jusqu'à l'endroit où se trouve la chaîne.
ALT-R	Une autre commande fort utile. Elle permet de remplacer une chaîne de caractères par une autre, sans que cette dernière ait nécessairement la même longueur.
ALT-S	sauvegarde le fichier édité.
ALT-X, ALT-Q	sert à sortir du programme. Si vous avez effectué des modifications, vous serez invité à les sauvegarder si vous le désirez.
ALT-U	permet d'annuler toutes les modifications effectuées depuis la dernière sauvegarde.

10.2. Les interruptions de débogage

Pour vous défendre contre le débogage de vos programmes, vous devez savoir comment fonctionne un débogueur. Au chapitre 10.1 nous avons fait la connaissance de Turbo Debugger. Mais comment fonctionne un débogueur d'une manière générale ? Selon son degré de raffinement, il utilise la même machine virtuelle que le programme analysé ou il émule un processeur virtuel. Mais tous les débogueurs ont en commun d'exploiter l'interruption 3 ou interruption de débogage. Si cette interruption est invoquée, elle déclenche en arrière-plan le débogueur présent. Si ce dernier n'existe pas, le programme continue.

Ceci nous donne une première idée de chausse-trappe à installer dans nos programmes : quelques appels à l'int 3 joliment disséminés à droite et à gauche peuvent mettre au désespoir les pirates en herbe.

Masquage des interruptions

Ne serait-il pas plus efficace de masquer matériellement l'interruption 3 ? La méthode est valable pour le clavier. Ce dernier est relié au bit 1 du contrôleur d'interruption (port 020h) c'est-à-dire à l'interruption 09h. Si vous barrez le canal de cette interruption, il peut être invoqué autant de fois que souhaité sans déclencher la routine associée. Mais l'interruption de débogage a un numéro inférieur à 8 et constitue une interruption non masquable (NMI, non maskable interrupt). Il n'est donc pas possible d'intervenir par masquage. Il

reste que la méthode est valable pour paralyser par exemple les ports série. Et en l'absence de souris certains programmes ne peuvent plus être pilotés.

Le détournement de l'interruption de débogage

Une autre possibilité de plier les interruptions à sa volonté consiste à les détourner vers des routines personnalisées. Vous pouvez ainsi stocker dans une procédure déclarée comme interruption une routine importante qui n'est pas appelée souvent mais qui est indispensable à l'exécution du programme. Il suffit de détourner l'interruption 3 sur votre routine et de l'invoquer hardiment. Mais cette méthode ne met en difficulté que les débogueurs qui ne disposent pas d'un processeur virtualisé avec sa propre table d'interruptions. Turbo Debugger ne se laisse pas troubler par ce genre de fantaisie.

Mémorisation de données à l'intérieur du vecteur

Une autre méthode très appréciée est la mémorisation de données dans le vecteur d'interruption. Le vecteur contient de la place pour un double mot (DWORD) et tant qu'il n'est pas appelé, vous pouvez y placer n'importe quelle valeur. Vous pouvez y écrire une somme de contrôle, à tester pendant l'exécution, ou espérer que l'interruption 3 sera appelée ce qui expédie le programme au nirvana. La position de l'interruption dans la table est donnée par [0:4*N° d'interruption]. C'est là que se trouve normalement l'adresse du gestionnaire d'interruption. Ce que vous y mettez pendant l'exécution de votre programme ne dépend que de vous. Mais n'oubliez pas de rétablir le contenu original avant de terminer le programme.

Le fait d'accéder à la table des vecteurs vous ouvre également une possibilité exploitée dans de nombreux logiciels du commerce. Les vecteurs des interruptions nécessitées par le programme, par exemple l'interruption Sound Blaster, ne sont pas détournés par appel des fonctions 25h et 35h de l'interruption 21h mais par accès direct à la table des vecteurs. Malgré tout, les manipulations de vecteurs ne constituent elles aussi une protection contre le débogage que si le débogueur n'utilise pas une machine virtuelle.

Les procédures en assembleur présentées ci-dessous donnent un exemple de mise en service des méthodes étudiées. Vous les trouverez en NODEB.ASM. La procédure Check_vector permet de mémoriser la valeur Longint transmise en la mettant à la position de l'interruption 3 de la table des vecteurs.

```
public Check_vector
Check_vector proc pascal check : dword;
  mov bx,0
  mov es,bx
  mov bx,18
  mov eax,es:[bx]
```

```
  mov oldint3,eax
  mov eax,check
  mov es:[bx],eax
  ret
Check_vector endp
```

Avec la procédure Vector_ok vous pouvez tester si un nombre donné LongInt se trouve à la position de l'interruption 3 dans la table des vecteurs d'interruption. La valeur retournée est 1 ou TRUE s'il y a concordance, sinon on obtient 0 ou FALSE.

```
public Vector_ok
Vector_ok proc pascal check : dword;
  mov bx,0
  mov es,bx
  mov bx,18
  mov eax,es:[bx]
  cmp eax,check
  je  @check_ok
  mov al,0
  jmp @check_fin
@check_ok:
  mov al,1
@check_fin:
  ret
Vector_ok endp
```

La procédure Restore_Checkvector sert à remettre dans son état d'origine le vecteur de l'interruption 3. Elle est exécutée systématiquement avant la fin du programme si vous avez procédé à un détournement de l'interruption

```
public restore_Checkvector
restore_Checkvector proc pascal
  mov bx,0
  mov es,bx
  mov bx,18
  mov eax,oldint3
  mov es:[bx],eax
  ret
restore_Checkvector endp
```

Remplacer des interruptions

Dans le registre des turpitudes appliquées aux interruptions, la substitution d'une interruption à une autre est également un bon tour à jouer. En pratique vous n'appellerez plus l'interruption 21h mais l'interruption pas à pas (01h) ou l'interruption de débogage (03h). Cette méthode se justifie surtout si vous programmez en assembleur (rappelons que les procédures de contrôle des mots de passe doivent être programmées en assembleur, au moins pour ce qui est de leur noyau). Commencez par prendre connaissance de l'interruption 21h et mémorisez-la dans le vecteur souhaité. Chaque fois que vous aurez par la suite besoin de l'interruption 21 h, remplacez 21h par le n° du nouveau vecteur. la méthode n'est pas garantie, mais elle introduit une esthétique horrible, la lisibilité du code s'en trouvant considérablement réduite. En tant que développeur, vous n'aurez pas de problème car vous pouvez opérer la substitution à la fin lorsque le programme est complètement terminé.

Dans la procédure en assembleur Copy_int21_int3 on copie l'interruption 21h dans l'interruption 3h. l'ancienne interruption est sauvegardée dans la variable anc_interrupt3 et doit être restaurée à la fin du programme :

```
 public Copy_int21_int3
Copy_int21_int3 proc pascal
  mov bx,0
  mov es,bx
  mov bx,18
  mov eax,es:[bx]
  mov anc_interrupt3,eax          ; sauvegarde l'ancienne int. 3
  mov bx,84                       ; charge l'int 21
  mov eax,es:[bx]
  mov bx,18                       ; et la mémorise en int3
  mov es:[bx],eax
  ret
Copy_int21_int3 endp
```

10.3. Comment tromper le débogueur

Au lieu de disposer des astuces dans le programme on peut aussi chercher à désactiver sciemment le débogueur. Dans un premier temps nous ne traiterons pas un débogueur particulier mais nous aurons toujours en tête l'exemple concret de Turbo Debugger comme débogueur de référence.

Le coup de la mémoire

Le coup de la mémoire consiste à réserver pour votre programme, non pas la mémoire strictement nécessaire, mais la totalité de la mémoire disponible. En Turbo-Pascal on écrira la directive :

```
{$M $800,550000,655000}
```

Si vous cherchez ensuite à déboguer le programme avec la version ordinaire de TD, vous obtiendrez un message d'erreur qui indiquera un manque de mémoire. SI vous placez la barre suffisamment haut, vous pourrez même faire trébucher TD386 avec le même message d'erreur. Mais il ne faudra pas jeter le bouchon trop loin. Tous les ordinateurs ne sont pas configurés de manière à laisser 630 Ko libres. Et n'est-il pas vrai que votre programme doit finalement tourner sur ces machines ?

Instructions ambiguës

Une autre façon de tromper directement le débogueur est de mettre en service des instructions ambiguës. Vous vous brancherez par exemple en plein milieu d'une instruction. La plupart des débogueurs ne comprendront pas ce qui leur arrive et sauteront à la première instruction qui suit et qui leur paraît claire. Le truc consiste à mettre dans cette instruction un ordre d'arrêt du programme. Si le programme est débogué pas à pas, le débogueur rend la main à cet endroit. En exploitation normale, un branchement s'effectue à l'endroit indiqué où se trouve un autre branchement qui saute par-dessus l'instruction d'arrêt.

Voici un exemple de réalisation pratique : (ds=cs!) :

```
No_Stepping proc near
  push ax
  jmp @Nostep+2
@Nostep:
  mov ds:byte ptr [06EBh],00
  mov ax,4C01h
  int 21h
```

```
    pop ax
    ret
No_Stepping endp
```

Stopper TD386

Jusqu'ici les méthodes présentées s'occupaient avant tout de la version ordinaire de TD mais dans les coups bas de trucage du matériel nous avons vu que TD386 faisait de la résistance. Une attitude inadmissible à laquelle il nous faut mettre un terme. En fait ce débogueur moderne présente un talon d'Achille : le mode protégé qu'il ne supporte absolument pas. Bon me direz-vous et alors ? Rassurez-vous : l'astuce à mettre en oeuvre ne prend que quelques lignes. Il n'est pas nécessaire de se soucier de la gestion générale du mode protégé. Il suffit de l'activer et de le désactiver aussitôt. On peut le faire avec le registre cr0. Le tout ne fonctionne pas sur les ordinateurs antérieurs au 286 mais ce genre de machine est (fort heureusement) voué à une disparition prochaine.

La procédure suivante active le mode protégé er revient aussitôt au mode réel. Elle fonctionne également si un driver EMS est installé :

```
public protected_stopping
protected_stopping proc pascal
    pusha
    cli                         ; inhibe les interruptions
    mov eax,cr0                 ; active le mode protégé
    or eax,1
    mov cr0,eax
    jmp PROTECTION_ENABLED      ; efface le canal d'exécution
PROTECTION_ENABLED:
    and al,0FEh                 ; rétablit le mode réel
    mov cr0,eax                 ; ne remet pas à zéro le processeur
    jmp PROTECTION_DISABLED     ; efface le canal d'exécution
PROTECTION_DISABLED:
    sti                         ; restaure les interruptions
    popa
  ret
protected_stopping endp
```

10.4. Automodifications

L'automodification est une variante de protection intéressante et assez répandue. Le PC exécute alors un programme différent de celui qui a été chargé. L'utilisation de sommes de contrôle répond à cette préoccupation.

Mise en service de sommes de contrôles

Si vous êtes l'heureux propriétaire d'un modem, vous connaissez la notion de somme de contrôle. Ces totaux servent à vérifier si un fichier a la bonne taille ou contient les bonnes données. Quand on transmet des données par modem, il est indispensable de contrôler tout le fichier transmi. Avec des programmes on peut procéder de même mais la méthode n'est pas forcément la plus intelligente. Ce n'est pas l'ensemble du programme qui est intéressant, mais certaines parties, et notamment celle qui contient le contrôle du mot de passe, ou la liste des mots de passe autorisés. Normalement ces emplacements ne sont pas faciles à localiser mais on peut s'aider d'une astuce. Au début de la procédure vous déclarez une constante locale facilement identifiable, par exemple la constante Longint 12345678h. Vous la repérerez sans peine dans l'exécutable. Effectuez ensuite la somme de contrôle des 100 ou 500 octets qui suivent. Vous pourrez ainsi vous assurer que personne n'a patché votre programme.

La même méthode s'applique aussi à un programme chargé en mémoire. L'adresse de la procédure intéressante est facile à trouver. A partir de là, vous vérifiez les 100 ou 500 octets qui suivent pour vous assurer qu'aucun programme résident de haute volée ne vous a modifié votre programme. Votre procédure de Checksum est en tout cas à couvrir par une ou plusieurs autres procédures du même type. Vous les appellerez à des moments arbitraires de votre programme. Le plus odieux est de faire ces tests au bout d'un certain temps, ou mieux encore lorsque l'utilisateur est sur le point de sauvegarder fièrement sa session ou sa situation de jeu.

Algorithmes de codage

Les algorithmes de codage constituent un domaine très vaste qui remplirait un ouvrage à part entière. Nous n'allons évoquer ici que quelques principes de base qui établissent deux catégories d'algorithmes.

La première méthode consiste à transformer le caractère d'origine en un autre en lui appliquant une clé. Le plus simple est d'ajouter la clé au caractère. Par bonheur le processeur maîtrise les débordements et après dépassement de la valeur maximale il recommence à compter à 0. Autrement dit, aucune donnée ne se perd jamais même si la clé a une valeur très élevée.

La combinaison par l'opérateur XOR est un peu plus raffinée. Le caractère d'origine est combiné avec la clé, tant à l'encodage qu'au décodage. La méthode devient plus efficace si on lui associe un nombre tiré au hasard qui n'est pas mémorisé. En réalité, il ne s'agit pas d'un nombre aléatoire car si vous partez de la même initialisation, vous obtiendrez toujours la même séquence de nombres. Vous pouvez donc l'exploiter sans hésiter.

D'autres méthodes nécessitent la mise en place d'un alphabet personnalisé. Tous les caractères sont remplacés par un caractère de votre alphabet. Le décodage est un peu plus difficile que précédemment car il demande non pas une clé unique mais tout un alphabet. Par ailleurs le temps de traitement est un peu plus long. Mais la méthode se justifie si vous ne codez que certaines zones d'un fichier, conformément aux indications du chapitre 10.4.

Le coup de la PIQ

Le coup de la PIQ n'est pas une théorie fumeuse mais un moyen sûr d'arrêter tout débogueur qui examinerait votre programme. On exploite la Prefetch-Queue du processeur. Pour pouvoir travailler plus vite, le processeur ne lit pas seulement l'instruction courante mais également les suivantes.

Le truc consiste à modifier un emplacement mémoire immédiatement contigu. Un test de la version de DOS se transforme ainsi en instruction d'arrêt du programme. Si le programme est exécuté normalement, cette dernière instruction n'est pas prise en compte car elle a déjà été chargée dans la PIQ sous sa forme non altérée. Certes la mémoire a été modifiée mais cette modification n'aura d'effet que si nous y retournons. Mais si un débogueur s'active à ramper pas à pas à travers le code, la PIQ ne sera pas active. L'instruction altérée sera donc exécutée dès la première fois. Et le programme se termine illico. Voici le code correspondant :

```
PIQ_Stop_System proc near
  push ds
  push ax
  push bx
  push cs
  pop  ds                          ;CS en DS
  mov  cs:word ptr [@int_21_fonct],4CB4h
                                   ; fonction arrêter le programme
@int_21_fonct:
  mov  ah,30h                      ; fonction lire la version de DOS
  int  21h
  pop  bx
  pop  ax
```

```
    pop  ds
    ret
PIQ_Stop_System endp
```

Notez que pour des raisons de clarté nous avons présenté toutes ces astuces dans des procédures individuelles. Dans vos programmes, il ne s'agit pas de les reproduire en tant que telles mais de les intégrer directement dans le code. Sinon n'importe quel amateur les désactivera en remplaçant leurs appels par des nops (non operates, 90h).

La mise en service de programmes de compactage

Dans le cadre de nos préoccupations il est tout à fait recommandé de se servir des programmes de compactage. D'abord le fichier EXE se fait plus petit, ensuite il offre un bon niveau de codage. Comme l'ensemble du fichier EXE se présente sous forme compactée, il n'est plus possible de le patcher. certes nous avons vu en 10.1 comment décompacter des programmes. Mais si on les recompacte ensuite, ils prennent généralement une autre taille différente de la taille d'origine. Il suffit alors d'incorporer dans le programme un test qui mesure la taille du fichier EXE et qui émet un message d'erreur lorsqu'elle a été modifiée.

Prises isolément les méthodes présentées ici ne sont pas plus que des barrières franchissables. Mais combinées entre elles elles assurent une protection de bonne qualité. Il est recommandé d'embrasser les saisies dans de grandes boucles. N'hésitez pas dans ces boucles à utiliser un grand nombre de variables globales, qui rendent le code plus difficile à interpréter. Quelques interruptions 3h bien dispersés sont toujours une bonne chose. Et pensez à ne jamais rassembler les astuces dans des procédures : intégrez-les directement dans votre code. De nombreux logiciels commerciaux utilisent ces trucs. Ils décourageront un grand nombre de pirates de la nouvelle génération.

10.5. ENTRAÎNEZ-VOUS : "TRAINERS" ET DÉBOGUEUR

Les "trainers" sont un des domaines les plus fascinants de l'informatique. On appelle ainsi un programme qui se présente la plupart du temps comme un programme résident ou un chargeur, qui se déclenche par une combinaison de touches et qui permet à l'utilisateur d'exercer une influence sur le comportement d'un autre programme. Ce mécanisme est malheureusement exploité par les craqueurs pour déjouer les protections installées dans les programmes. Nous n'évoquerons pas ici ce sombre aspect de la réalité.

Nous allons plutôt nous intéresser à la catégorie des programmes d'entraînement. Il vous est arrivé à tous de progresser à grand-peine dans un jeu d'action pour vous trouver enfin confronté à l'ennemi final... et constater qu'il ne vous reste plus d'arme et que vous n'avez plus qu'une seule misérable vie. Le programme d'entraînement est le sauveur qui vous tire d'affaire. Appuyez sur la bonne combinaison de touches et vous voilà armé jusqu'aux dents et muni de toute une collection de vies pour aborder la phase finale.

Malheureusement ce type d'entraîneur n'existe pas toujours dans tous les jeux. La solution : mettre la main à la pâte soi-même. Ce chapitre vous dévoile tout ce que vous devez connaître pour programmer vous même un entraîneur. Il faut d'abord savoir reconnaître les programmes faciles à entraîner et ceux qui ne s'y prêtent pas. Les jeux de rôle et d'aventures de Sierra (tm) ne conviennent pas. Ce genre de jeu utilise une sorte de langage de script traduit par un interpréteur. Dans ce cas il vaut mieux sauvegarder la situation du jeu et la traiter avec un éditeur hexadécimal. Dans le pire des cas vous pourriez écrire vous-même un éditeur de sauvegarde.

Les jeux d'action par contre se prêtent bien à l'entraînement. En général le nombre de vies y est décrémenté, des armes supplémentaires sont distribuées et des points sont collectés. Tout cela est géré directement dans le jeu par des variables BYTE, WORD ou DWORD. Pour fabriquer un entraîneur il suffit de repérer les positions de ces variables en mémoire et d'écrire un programme résident qui leur donne de nouvelles valeurs lorsqu'on appuie sur une certaine combinaison de touches.

A la recherche des variables

Si tout cela était aussi simple que nous l'avons présenté, il y aurait pour chaque jeu des douzaines d'entraîneurs. Mais la réalité est plus complexe. Le plus gros problème consiste à détecter les variables utilisées. Vous devrez disposer d'un débogueur, Turbo Debugger faisant parfaitement l'affaire. Son mode de fonctionnement est décrit en 10.1. La recherche va d'abord porter sur les structures usuelles aptes à incrémenter ou à décrémenter des variables. Supposons que vous vous trouviez à un endroit du programme où vous perdez une vie. Si vous détectez l'emplacement correspondant, vous aurez aussi trouvé où se trouve la variable qui gère les vies. Il existe deux méthodes pour parvenir à ce résultat :

Méthode A : Recherche de structures fréquemment employées

Essayez d'abord la présente méthode car si elle réussit elle vous conduira plus rapidement au but. Dans notre exemple nous recherchons une séquence d'octets qui décrémente le contenu d'une variable-octet. Il s'agit probablement d'une variable-octet car le nombre de vies est vraisemblablement inférieur à 256.

La structure la plus fréquemment employée à cet effet est :

```
dec byte ptr [xxyy]
```

xxyy étant l'offset de la variable dans le segment de données actuel. L'assembleur traduit cette instruction par :

```
FE 0E yy xx
```

Concrètement vous devrez rechercher la combinaison d'octets FE 0E puisque vous ne connaissez pas yy xx. Si le programme est gros, ne vous effrayez pas du nombre de combinaisons que vous allez trouver. Certaines ne jouent aucun rôle car elles font partie d'une instruction plus longue. Ensuite de nombreuses décrementations se rapporteront probablement aux mêmes variables. Notez les variables trouvées et testez leur contenu pendant l'exécution du programme. Vous allez découvrir la bonne.

Si la recherche de l'instruction de décrementation ne donne rien, testez d'autres combinaisons équivalentes. Le tableau suivant indique les structures les plus fréquentes :

Instruction assembleur	Traduction
dec word ptr [xxyy]	FF 0E yy xx
dec byte ptr [xxyy]	FE 0E yy xx
sub word ptr [xxyy],zz	83 2E yy xx zzzz
sub byte ptr [xxyy],zz	80 2E yy xx zz
sub [xxyy],ax	29 06 yy xx
sub [xxyy],dx	29 16 yy xx
inc word ptr [xxyy]	FF 06 yy xx
inc byte ptr [xxyy]	FE 06 yy xx
add word ptr [xxyy],zz	83 06 yy xx zzzz
add byte ptr [xxyy],zz	80 06 yy xx zz

Rappelez-vous que certains programmes exploitent plusieurs segments de données ou mémorisent des données dans le segment de code. Si vous ne trouvez rien dans le segment courant, essayez les autres segments utilisés par le programme.

Si tous ces essais ne donnent pas de résultat ou si au contraire le nombre de candidats à la solution est trop important, il faudra suivre l'autre voie et utiliser la méthode B.

Méthode B : Suivi de l'exécution du programme

Suivre à la trace l'exécution d'un programme est devenu un sujet encore plus explosif depuis le procès entre Microsoft et Stack Inc. Si la méthode A a échoué, c'est la seule méthode pour détecter les variables. Vous devrez appliquer la stratégie suivante : d'abord localiser une procédure qui redessine l'écran. Explorez pas à pas cette procédure et analysez les éléments de l'affichage. Vous tomberez très vite sur des variables candidates. Si vous ne trouvez pas la variable qui gère les vies ou l'arrêt du programme, suivez la boucle principale jusqu'à parvenir à une comparaison d'une certaine variable avec 0. Il pourrait s'agir (mais pas forcément) de la variable qui gère les vies ou la fin du programme.

Traitement des données de base

Après avoir repéré les variables employées par le programme, vous pouvez déjà commencer à écrire l'entraîneur. Il vous faut connaître quelques paramètres de base du programme. Prenez d'abord note du segment de données de chaque variable. Ensuite vous devez savoir si le programme utilise ou non une routine propre comme gestionnaire du clavier. Recherchez l'instruction

```
in al,60h
```

Posez-y un point d'arrêt. Exécutez le programme et parvenu à cet endroit appuyez sur une touche. Si le programme s'interrompt, vous avez découvert un gestionnaire. Notez le segment de code (CS) et l'offset de l'instruction. Cette dernière valeur se trouve dans le pointeur d'instruction IP.

Si le programme n'est pas interrompu, vous devrez poursuivre votre recherche de l'instruction. Si elle n'apparaît pas, vérifiez si le programme n'a pas plusieurs segments de code. Il faut alors inspecter tous les segments de code si le segment de base ne contient rien. Souvent il est recommandé d'analyser aussi les segments ES et DS.

Si malgré tous vos efforts, vous ne découvrez aucune instruction suspecte, c'est que sans doute votre programme ne contient pas de gestionnaire personnalisé mais utilise l'interruption 9h ou 16h de DOS.

Comment programmer un entraîneur

Comment organiser son programme d'entraînement ? En principe un tel programme se compose de deux parties. La première partie est l'installation, ce qui ne veut pas dire chargement en tant que programme résident mais intégration dans le programme à entraîner. La seconde partie est l'interface qui teste si une certaine touche a été enfoncée et réagit en conséquence.

L'installation

Pour l'installation il faut distinguer si le programme utilise sa propre routine de gestion du clavier ou s'il fait appel aux routines de DOS. En cas d'utilisation des routines de DOS, il suffit de détourner à votre profit l'interruption 9h. Vous testez alors si l'une de vos touches a été pressée et réagissez le cas échéant avant d'activer l'interruption d'origine.

Si le programme utilise son propre gestionnaire, les choses se compliquent un peu. Il nous faut d'abord une interruption libre. Prenez par exemple l'interruption 65h. Puis accrochez-vous à une interruption fréquemment appelée, par exemple l'interruption 21h. Cette dernière est toujours appelée par le programme à entraîner. Elle va vous servir à effectuer une petite modification du code. Cette altération reste parfaitement légale car il ne s'agit pas du programme enregistré sur le disque mais d'une zone en mémoire centrale.

La zone à modifier est celle où un caractère est lu par l'instruction :

```
in al,60h
```

En code machine, la séquence s'écrit : E4 60.

Lorsqu'une interruption se produit, le processeur sauvegarde sur la pile les indicateurs, le segment de code d'où provient l'appel et le pointeur d'instruction courant (IP). Sauvegardez donc sur la pile les valeurs courantes de BP, BX et DS. Puis déplacez le registre de pile SP en BP. Vous pouvez alors lire le segment de code par [BP+10] et la position de l'IP par [BP+8]. Mettez le segment de code en DS et IP en BX. Vous pouvez alors tester si à l'emplacement référencé par DS:[BX] se trouve la séquence E4 60. Si tel est le cas, l'interruption a été déclenchée par le programme à entraîner et vous avez trouvé la zone à modifier. Remplacez les deux octets par CD 65. Au lieu de lire un caractère par in al,60h le programme va déclencher l'interruption 65h.

A l'intérieur de l'interruption 65h vous ne devez modifier aucun registre autre que al. Par ailleurs vous devez y lire un caractère par in al,60h. Testez si le caractère est une touche employée par votre entraîneur. Si c'est oui, le code d'entraînement sera activé, sinon l'interruption est à clôturer.

L'interface

C'est l'interface qui est en fait la partie la plus intéressante de l'entraîneur. Il nous faut à nouveau faire une distinction entre les programmes qui n'utilisent qu'un seul segment de données et ceux qui en ont plusieurs.

Commençons par étudier le premier cas. Il s'agit la plupart du temps de programmes obtenus avec un langage du type Borland Pascal qui n'autorisent qu'un seul segment de données. Vous n'avez plus à vous faire de souci : le segment de données est forcément le bon. Lorsque l'interruption détournée par vos soins est appelée, testez d'abord la touche frappée. Si c'est une touche de l'entraîneur, la routine se branche d'abord à l'endroit où la variable est modifiée et ensuite à l'endroit où l'interruption se termine en bonne et due forme. Ce sera soit un IRET soit un appel à l'interruption originale. Si la touche enfoncée n'est pas une touche de l'entraîneur, cette partie est aussitôt exécutée.

Dans la partie qui modifie la variable, écrivez sa nouvelle valeur à la position repérée par vos soins. Il n'est pas toujours intelligent de prendre la valeur maximale autorisée par le type. Il vaut mieux trouver la valeur qui recharge complètement l'énergie ou restaure complètement les armes. Une instruction typique pourrait s'écrire :

```
Full_Energy:
mov ds:[1234],80
jmp finir_routine
```

Si le programme exploite plusieurs segments de données, l'affaire se complique un peu. Vous avez déjà noté le segment de données de la variable à modifier. Vous connaissez aussi le segment de code où se produit la lecture du clavier. Vous pouvez en déduire le segment de données. Dans l'interruption de l'entraîneur lisez le segment de code à l'appel de l'interruption. A ce nombre, ajoutez ou retranchez la différence entre le segment de données de la variable et le segment de code à l'appel de l'interruption. Vous avez alors le segment de données et vous pouvez procéder à la modification de la variable. Voici un exemple de routine de ce type :

```
Full_weapons:
  pusha
  mov bx,[bp+8]
  sub bx,3E43h
  mov ds,bx
  mov ds:[1234],255
  popa
  jmp finir_interruption
```

Ce sont là tous les secrets à connaître pour rédiger un entraîneur. C'est donc le moment de faire un essai avec le mini-jeu RAIDER qui se trouve sur le disque d'accompagnement.

Un entraîneur pour le jeu RAIDER

Nous espérons d'abord que vous avez déjà tenté votre chance à ce jeu. Il n'est pas difficile à entraîner et constitue une bonne entrée en matière.

S'il s'agit de votre premier entraîneur, vous rencontrerez probablement quelques difficultés. Mais sur le CD d'accompagnement se trouve un exemple de code source complet. Il a été conçu comme un kit de base pour vos propres projets. Nous allons examiner ici la partie intéressante qui contient l'interruption.

L'entraîneur commence par définir quelques variables et une signature de reconnaissance pour tester plus tard s'il est déjà résident.

```
;          ************************************************************
;          ***                                                      ***
;          ***         Entraîneur pour R A I D E R                  ***
;          ***                                                      ***
;          ***        (c) 1994 by Micro Application                 ***
;          ***                                                      ***
;          ***           Auteur: Boris Bertelsons                   ***
;          ***                                                      ***
;          ************************************************************
 ;
.286
jumps
  w equ word ptr
  b equ byte ptr
code segment public
public insthand
public handler9
public reslim
public oldint9
public signature
public siglong
public check_inst
assume cs:code,ds:code
signature:  db 'Entraîneur pour MICRO APPLICATION par InspirE/TC'
```

```
oldint9: dd 0
procedure: dd ?
siglong equ offset oldint9 - offset signature
;  ********************************************************************
;  ***                                                              ***
;  ***            Ici commencent les routines proprement             ***
;  ***                      dites de l'entraîneur                   ***
;  ***                                                              ***
;  ********************************************************************
```

L'étape suivante est l'installation du nouveau gestionnaire de l'interruption 9h. Cette routine va lire un caractère du clavier pour tester s'il s'agit d'une touche de l'entraîneur. Si tel est le cas, la variable visée est ajustée. Les différentes variables ont été repérées dans le cadre d'une petite session de Turbo Debugger.

```
;  ********************************************************************
;  ***                                                              ***
;  ***     Nouvelle interruption 9h. La procédure teste si une      ***
;  ***     touche a été tapée et modifie des variables le cas       ***
;  ***     échéant                                                  ***
;  ***                                                              ***
;  ********************************************************************
handler9 proc pascal
  pushf
  push bp
  push ds
  push bx
  in  al,60h        ; lit un caractère
  cmp al,59         ; touche F1
  je  Game_over
  cmp al,63         ; touche F5
  je  Longue_vie
  cmp al,64         ; touche F7
  je  Add_Points
  jmp fin_keyb
Fin_reguliere:
Fin_keyb:
  pop  bx
  pop  ds
```

```
    pop  bp
    popf
    jmp dword ptr cs:oldint9        ; appelle l'ancienne int 9h
  Game_over:
    pushf
    PUSHA
    ; A adapter à vos besoins!!
    mov ds:byte ptr [0648h],1
    POPA
    popf
    jmp fin_reguliere
  Longue_vie:
    pushf
    PUSHA
    ; A adapter à vos besoins !!
    mov ds:byte ptr [0648h],255
    POPA
    popf
    jmp fin_reguliere
  Add_Points:
    pushf
    PUSHA
    ; A adapter à vos besoins!!
    mov bx,ds:word ptr [064Ah]
    add bx,1000d
    mov ds:word ptr [064Ah],bx
    POPA
    popf
    jmp fin_reguliere
  handler9 endp
```

Le code qui suit sert à installer ou désinstaller l'entraîneur.

```
insthand proc pascal
  reslim label byte
      push ds
    pop ds
      mov ax,3509h              ; sauve l'ancienne INT 21
    int 21h
```

```
  mov w oldint9,bx
  mov w oldint9 + 2,es
  mov ax,2509h                ; détourne l'INT 21h
  lea dx,handler9
  int 21h
     ret
insthand endp
check_inst proc near
  mov ax,3509h                ; lit le vecteur d'interruption
  int 21h
  mov di,bx
  mov si,offset signature
  mov di,si
  mov cx,siglong
  repe cmpsb                  ; teste la signature
    ret
check_inst endp
code ends
end
```

Le programme RAIDER n'a qu'un seul segment et n'utilise pas de routine de clavier propre. C'est pourquoi le présent entraîneur constitue juste une entrée en matière.

L'entraîneur ci-dessous comporte un raffinement supplémentaire en ce sens qu'il appelle une procédure de mise à jour de l'affichage de la variable altérée. Il peut aussi servir de modèle pour voir comment sont traités les programmes qui gèrent directement le clavier.

```
;  ********************************************************************
;  ***                                                              ***
;  ***                Entraîneur pour *******                       ***
;  ***                                                              ***
;  ***             (c) 1994 by Micro Application                    ***
;  ***                                                              ***
;  ***              Auteur: Boris Bertelsons                        ***
;  ***                                                              ***
;  ********************************************************************
;
;
```

```
.286
  w equ word ptr
  b equ byte ptr
code segment public
public insthand
public handler21
public reslim
public oldint21
public oldint65
public signature
public siglong
public check_inst
assume cs:code,ds:code
signature:  db 'MICRO APPLICATION'
oldint21: dd 0
oldint65: dd 0
proced: dd ?
siglong equ offset oldint21 - offset signature
;  ********************************************************************
;  ***                                                              ***
;  ***        Ici commencent les routines proprement                ***
;  ***                      dites de l'entraîneur                   ***
;  ***                                                              ***
;  ********************************************************************
```

Voici d'abord la nouvelle procédure de l'interruption 21h. Elle recherche l'instruction in al,60h dans le programme et la remplace le cas échéant.

```
;  ********************************************************************
;  ***                                                              ***
;  ***      Nouvelle interruption 21h. La procédure teste si à      ***
;  ***      l'endroit indiqué de la mémoire se trouve               ***
;  ***      l'instruction "in al,60h", et la remplace               ***
;  ***      le cas échéant par "int 65h" !                          ***
;  ***                                                              ***
;  ********************************************************************
handler21 proc pascal
  pushf
  push bp
```

```
  push ds
  push bx
  mov  bp,sp
  mov  bx,[bp+10]                ;cs au moment de l'interruption en
                                 ;BX, DOS !!!
                                 ; ATTENTION ! Dans TD [bp+16] !!!
  add  bx,0366h                  ; CS de la 1ère INT 21h +
                                 ;2136h = CS de la routine du clavier
  mov  ds,bx                     ; cs de la routine du clavier en ds
   mov  bx,568Bh                 ; 8B56h = mov dx,[bp+06]
  cmp  ds:word ptr [0005h],bx    ; motif présent dans la routine ?
  jne  pas_changer
  mov  ds:word ptr [0005h],9090h ; écrire Int 65h !
  mov  ds:word ptr [0007h],65CDh ; écrire Int 65h!
pas_changer:
  pop  bx
  pop  ds
  pop  bp
  popf
  jmp dword ptr cs:oldint21      ; appelle l'ancienne Int 21h
handler21 endp
```

Voici maintenant les routines d'entraînement proprement dites. Le gestionnaire prend d'abord connaissance du segment de code au moment du déclenchement de l'interruption. Comme le segment de code du gestionnaire de clavier est le même que celui de la procédure de mise à jour de l'écran, on peut le reprendre directement. Sinon il faudrait procéder à un ajustement comme indiqué plus haut. Lorsqu'une variable doit être entraînée, le pointeur est d'abord initialisé sur la procédure de mise à jour de l'affichage associé. La variable est ensuite appelée et la procédure de mise à jour de l'affichage déclenchée. Puis la procédure se branche sur le gestionnaire original.

```
;  ********************************************************************
;  ***                                                              ***
;  ***      L'interruption 65h - Procédure, lit un caractère        ***
;  ***      par "in al,60h" et teste si la touche correspon-        ***
;  ***      dante a été définie dans l'entraîneur. Si oui,          ***
;  ***      les modifications de mémoire et les appels de           ***
;  ***      procédure indiqués sont effectués. Vous devrez          ***
;  ***      reporter ici vos variables d'entraînement               ***
```

```
;  ***                                                          ***
;  ***************************************************************
handler65 proc far
  pushf
  push bp
  push ds
  push bx
  mov  bp,sp
  mov  bx,[bp+10]              ;cs au moment de l'interruption en BX
   in  al,60h                  ; lit un caractère
  cmp al,63                    ; touche F5
  je  Full_Shoots_j
  cmp al,64                    ; touche F6
  je  Full_Lives_J
  cmp al,65                    ; touche F7
  je Weapon_new_j              ;
  cmp al,66                    ; touche F6
  je Weapon_new_j              ;
  cmp al,67                    ; touche F9
  je Weapon_new_j              ;
  cmp al,68                    ; touche F10
  je  More_Points_J
Fin_keyb:
  pop bx
  pop ds
  pop bp
  popf
  iret
Full_Shoots_j:
  jmp Full_Shoots
Full_Lives_j:
  jmp Full_Lives
More_Points_j:
  jmp More_Points
Weapon_new_j:
  jmp Weapon_new
Full_Shoots:
  pushf
```

```
  PUSHA
  sub  bx,0                    ; CS déjà OK
  mov word ptr proced+2,bx
  mov  bx,1401h                ; es:[bx]  =  14EF:1401
  mov word ptr proced,bx
;--------
  mov ds:byte ptr [0DA3h],20h
  mov ax,20h
  push ax
  call dword ptr [proced]
  POPA
  popf
  jmp Fin_keyb
Full_Lives:
  pushf
  pusha
  sub  bx,0                    ;
  mov word ptr proced+2,bx
  mov  bx,1317h                ; es:[bx]  =  14EF:1317
  mov word ptr proced,bx
;-----------
  mov ds:byte ptr [0DA3h],0009
  mov ax,9
  push ax
  call dword ptr [proced]
  popa
  popf
  jmp Fin_keyb
Weapon_new:
  pushf
  pusha
  sub  bx,0                    ;
  mov word ptr proced+2,bx
  mov  bx,1454h                ; es:[bx]  =  14EF:1454
  mov word ptr proced,bx
;-----------
  sub al,65
  mov ah,0
```

```
  mov ds:byte ptr [0DA2h],al
  push ax
  call dword ptr [proced]
  popa
  popf
  jmp Fin_keyb
More_Points:
  pushf
  pusha
  sub  bx,0                        ;
  mov word ptr proced+2,bx
  mov  bx,1BD0h                    ; es:[bx]  =  14EF:1BD0
  mov word ptr proced,bx
;-----------
  mov ax,1000
  push ax
  call dword ptr [proced]
  popa
  popf
  jmp Fin_keyb
handler65 endp
insthand proc pascal
reslim label byte
    push ds
  pop ds
    mov ax,3521h                   ; sauve l'ancienne INT 21
  int 21h
  mov w oldint21,bx
  mov w oldint21 + 2,es
    mov ax,3565h                   ; sauve l'ancienne INT 65h
  int 21h
  mov w oldint65,bx
  mov w oldint65 + 2,es
  mov ax,2521h                     ; détournement de l'INT 21h
  lea dx,handler21
  int 21h
  mov ax,2565h                     ; INT 65h sur routine de
                                   ; clavier personnelle
```

```
   lea dx,handler65
   int 21h
      ret
insthand endp
check_inst proc near
   mov ax,3521h                    ; lit le vecteur d'interruption
   int 21h
   mov di,bx
   mov si,offset signature
   mov di,si
   mov cx,siglong
   repe cmpsb                      ; teste la signature
     ret
check_inst endp
code ends
end
```

Ces deux exemples devraient vous permettre de rédiger un entraîneur pour la plupart des jeux qui le supportent. Là encore c'est la pratique qui fera de vous un maître de l'art.

11. La gestion de la mémoire

N'importe quel PC moderne est aujourd'hui équipé de 4 Mo de mémoire principale en moyenne, le plus souvent avec un temps d'accès de 70 ns ou moins. Sous DOS, toutefois, seuls 640 Ko peuvent être utilisés directement. Ce chapitre va vous apprendre comment faire intervenir le reste de la mémoire. Nous allons d'abord nous occuper ici des structures de base.

11.1. La mémoire conventionnelle

La mémoire principale libre conventionnelle sous DOS comprend normalement 640 Ko. C'est souvent insuffisant. Pour pallier cette déficience, vous allez apprendre dans ce paragraphe à utiliser le modèle de mémoire FLAT des 80286, 80386 et 80486, de façon à pouvoir gérer au total 4 Go de mémoire. Cette quantité est amplement suffisante pour la plupart des logiciels.

La mémoire DOS conventionnelle, comme nous l'avons déjà mentionné, comprend au maximum 640 Ko. Malheureusement, elle ne peut pas être adressée comme un bloc entier. Elle est partagée en plusieurs segments. L'un des segments a une taille de 64 Ko et commence à une adresse de paragraphe. Il s'agit d'une adresse physique divisible par 16. Quand vous voulez adresser la mémoire, l'ordinateur doit toujours savoir à quel segment il doit accéder. L'adressage à l'intérieur d'un segment s'effectue par l'intermédiaire d'un offset. La taille d'un offset est d'un mot (word), puisqu'un mot permet d'adresser exactement 64 Ko. L'accès à la mémoire se fait donc sous la forme suivante :

```
Dest:=SEGMENT:OFFSET
```

Si vous programmez dans un langage de programmation évolué comme Pascal et si vous vous contentez des types de données prédéfinis, vous n'avez pas besoin de vous occuper de l'adressage de la mémoire. Pascal le fait automatiquement pour vous. En revanche, si vous allez plus avant dans la

programmation du système, vous ne pouvez pas vous passer d'une connaissance plus approfondie des différents accès à la mémoire.

Jusqu'au 286, le PC possédait quatre registres de segments. Il s'agissait en premier lieu du segment de code, destiné à servir d'index au segment dans lequel était situé le code que l'on voulait traiter. Le suivant était le segment de données. Il pointait sur le secteur de mémoire dans lequel les données du programme étaient déposées. Le segment "extra" servait à effectuer les opérations sur la mémoire et les opérations de copie. Enfin le segment "pile" désignait la pile elle-même. A partir du 386, il faut ajouter les registres FS et GS.

Il existe différentes possibilités d'accéder à la mémoire. Si vous programmez en Pascal, vous savez certainement ce que signifie la commande MEM. Par l'intermédiaire de :

```
Effet:=MEM[Segment:Offset];
```

vous pouvez copier un octet de la mémoire dans une variable. Si vous voulez lire par exemple le 50e pixel de l'écran en mode MCGA, vous écrivez :

```
Octet_50:=MEM[$A000:0050];
```

Si vous souhaitez réaliser la même action en assembleur, vous disposez de différentes méthodes. La plus courante, destinée surtout à lire des blocs de données d'une certaine taille, consiste à se servir des commandes lodsb/stosb/movsb, ou de leurs correspondants WORD et DWORD. Cette méthode est un peu longue, car il faut sauvegarder le segment de données, afin de pouvoir accéder ultérieurement aux données du programme.

```
push des
mov  si,0a0000h
mov  des,si
mov  si,50
lodsb
mov  byte ptr byte_50,al
pop  des
```

Dans la plupart des cas, vous aurez à lire non pas des données en provenance d'un périphérique comme la carte VGA, mais des données déjà chargées en mémoire principale. Or il n'est pas possible d'utiliser n'importe quel secteur de mémoire. La mémoire nécessaire doit d'abord être affectée à cet usage. Cette tâche est prise en charge par Pascal ou par DOS. Cette dernière méthode est recommandée quand d'autres programmes fonctionnent en parallèle avec le vôtre dans la mémoire.

Les accès en mémoire à l'un de ces secteurs réservés se font au moyen d'un pointeur. Un pointeur est un type de données contenant l'offset et le segment d'un segment de mémoire. On initialise normalement un pointeur par l'intermédiaire de la procédure Getmem, mais on peut aussi le construire. Au moyen de Freemem, vous pouvez libérer ensuite le secteur ainsi occupé. La mémoire mise en œuvre par Pascal n'est toutefois accessible qu'au programme Pascal. Pascal réserve au début de l'exécution du programme la mémoire minimale spécifiée et la libère ensuite seulement quand le programme prend fin.

Si vous ne voulez pas vous servir de la mémoire conventionnelle DOS mise à votre disposition par Pascal, vous devez faire appel aux fonctions 48h et 49h de l'interruption DOS 21h. Par l'intermédiaire de la fonction 48h, vous pouvez réserver un bloc entier de mémoire. Par l'intermédiaire de 49h, vous libérez à nouveau ce dernier. Il importe de savoir que les blocs DOS ne peuvent être réservés que sous la taille d'un paragraphe. Cela veut dire que vous devez arrondir toutes les indications de taille aux 16 octets supérieurs.

Voici un exemple qui vous montrera quel sera l'aspect de vos routines pour l'accès conventionnel en mémoire. Il utilise les procédures dos_getmem et dos_mem en assembleur.

```
dos_getmem proc pascal pointeur:dword,ensemble:word
; ********************************************************************
; *** Alloue un secteur (max. 64 Ko) dans la mémoire RAM de DOS   ***
; ********************************************************************

  push des
  mov bx,ensemble
  shr bx,4
  inc bx
  mov ah,48h
  int 21h
  mov bx,w [pointeur+2]
  mov des,bx
  mov bx,w [pointeur]
  mov w [bx],0
  mov w [bx+2],ax
  pop des
  ret
dos_getmem endp

dos_freemem proc pascal pointeur:dword
```

```
; ********************************************************************
; ***        Libère un secteur alloué par dos_getmem               ***
; ********************************************************************
  mov ax,word ptr [pointeur+2]
  mov es,ax
  mov ah,49h
  int 21h
  ret
  dos_freemem endp
```

11.2. EMS - une étape vers plus de mémoire

Puisque le secteur de mémoire normal du DOS est limité à 640 Ko et que certaines applications demandent de plus en plus de mémoire, les sociétés Microsoft, Lotus et Intel se sont mises d'accord pour créer un nouveau standard en matière de mémoire. Cela a donné EMS, abréviation de "Expanded Memory System". Il s'agissait à l'origine d'une carte que l'on pouvait connecter dans un PC/XT et qui procurait à l'ordinateur jusqu'à 8 Mo supplémentaires. Cette mémoire était reproduite par blocs de 64 Ko dans la mémoire conventionnelle. Le secteur reproduit était désigné sous le cadre de "fenêtre EMS" ou "EMS frame". Les blocs ainsi ouverts sont divisés en quatre pages de 16 Ko chacun et peuvent être adressés comme l'est la mémoire principale. Pour cette raison, on appelle cette mémoire la "mémoire paginée".

L'accès à ces "pages" est réglé par un Expanded Memory Manager (en abrégé : EMM). Vous n'avez donc pas besoin de vous préoccuper des détails techniques. Avec l'unité décrite dans ce livre, vous avez dans les mains un outil qui vous permettra d'accéder sans aucun problème au EMS.

Les fonctions de la mémoire paginée

EMM propose ses fonctions par l'intermédiaire de l'interruption 67h. Si une erreur se produit dans l'une de ses fonctions, EMM renvoie un code d'erreur, dont on déduira la signification d'après le tableau suivant :

Code	Signification
80h	Interface logiciel erronée
81h	Matériel EMS défectueux
82h	EMM est occupé

Code	Signification
83h	Handle invalide
84h	Numéro de fonction invalide
85h	Pas de handle disponible
86h	Erreur dans la sauvegarde ou la restauration du rapport entre les pages logiques et les pages physiques
87h	Trop peu de pages libres
88h	Nombre de pages invalide
89h	Vous avez essayé d'ouvrir 0 page
8Ah	Numéro de page invalide
8Bh	Numéro de page physique invalide
8Ch	Le mapping ne peut être assuré
8Dh	Le mapping est déjà assuré
8Eh	Le mapping n'a pas été assuré
8Fh	Mauvais numéro de sous-fonction

Nous allons maintenant passer en revue les fonctions de gestionnaire de mémoire paginée EMM. Nous n'envisagerons que les fonctions de la version 3.0. Les versions ultérieures n'existent pas toujours ou alors elles n'ont pas pu s'implanter.

Fonction 40h - Lire le statut

Entrée	AH = 40h
Sortie	AH = Etat du EMM

Quand la fonction retourne la valeur 0, cela veut dire que le gestionnaire EMM fonctionne sans erreur. Sinon, l'erreur peut être déduite de son code, en fonction du tableau ci-dessus.

Fonction 41h - Obtenir l'adresse de segment de la fenêtre (frame)

Entrée	AH = 41h
Sortie	AH = Etat du EMM BX = Adresse de segment de la fenêtre

L'adresse de segment n'est valide que si le code renvoyé contient la valeur 0. Sinon, il s'agit d'une erreur, que l'on détermine en fonction du tableau des

erreurs. L'adresse de segment est celle où l'on trouvera la fenêtre (frame) dans la mémoire principale.

Fonction 42h - Obtenir le nombre de pages EMS

Entrée	AH = 42h
Sortie	AH = Etat du EMM BX = Nombre de pages EMS libres DX = Nombre total de pages EMS

Une page EMS fait 16 Ko. L'ensemble de la mémoire paginée s'obtient donc en effectuant une multiplication par le nombre 16*1024. Les valeurs ne sont valides que si le code de l'état porte la valeur 0.

Fonction 43h - Occuper la mémoire paginée

Entrée	AH = 43h
Sortie	AH = Etat du EMM BX = Nombre de pages (16 Ko) à réserver DX = Numéro de handle

Les pages allouées sont adressées par le numéro de handle retourné par la fonction, valide uniquement quand le code d'état contient la valeur 0. Il est important que vous ne perdiez pas ce numéro et que vous rendiez les pages allouées au terme du programme. Sinon, elles demeurent bloquées pour tous les programmes qui viennent ensuite. Il est possible d'occuper au plus 512 pages.

Fonction 44h - Définir le mapping

Entrée	AH = 44h
Sortie	AH = Etat du EMM AL = Numéro de page dans la fenêtre EMS BX = Numéro de page dans le bloc réservé DX = Handle du bloc réservé

Le numéro de page dans la fenêtre EMS peut prendre une valeur entre 0 et 3. Le numéro de page dans le bloc réservé est compté relativement au début du bloc en question (c'est-à-dire la sixième page du handle 1Ch). Le secteur indiqué s'ouvre aussitôt dans le cadre DOS-EMS. Si le code d'état porte la valeur 0, il n'y a pas d'erreur.

Fonction 45h - Libérer les pages

Entrée AH = 45h

Sortie AH = Etat du EMM
DX = Handle

Cette fonction libère les pages réservées de la mémoire paginée. Toutes les pages du handle sont ainsi libérées. Vous devez sans faute appeler cette fonction au terme du programme pour tous les handles occupés, car sinon la mémoire paginée en question ne serait pas accessible aux autres programmes. Le code d'état 0 indique un fonctionnement sans erreur.

Fonction 46h - Obtenir la version EMS

Entrée AH = 46h

Sortie AH = Etat du EMM
AL = Numéro de version

Le numéro de version est conservé en format BCD. Le numéro principal se trouve dans les quatre bits supérieurs, la sous-version se trouve dans les quatre bits inférieurs. Si le code d'état n'est pas égal à 0, une erreur est survenue.

Fonction 47h - Conserver le mapping

Entrée AH = 47h

Sortie AH = Etat du EMM
DX = Handle

Cette fonction permet de conserver l'affectation du mapping à l'intérieur du gestionnaire de mémoire paginée en tant que spécification par défaut. La fonction ne comporte pas d'erreur si le code d'état est égal à 0.

Fonction 48h - Effacer le mapping

Entrée AH = 48h

Sortie AH = Etat du EMM
DX = Numéro handle

Avec cette fonction, vous pouvez effacer la spécification conservée par l'intermédiaire de la fonction 47h. Si le code d'état n'est pas égal à 0, une erreur s'est produite.

Fonction 4Bh - Obtenir le nombre de handles

Entrée AH = 4Bh

Sortie AH = Etat du EMM
BX = Nombre de handles occupés

La fonction fournit le nombre de handles occupés actuellement par le gestionnaire de mémoire paginée EMM. Un code d'état différent de 0 indique une erreur.

Fonction 4Ch - Obtenir le nombre de pages occupées

Entrée AH = 4Ch

Sortie AH = Etat du EMM
DX = Numéro handle
BX = Nombre de pages occupées

Cette fonction permet d'obtenir le nombre de pages (16 Ko) de mémoire paginée occupées par le handle transmis.

Fonction 4Dh - Obtenir les handles transmis

Entrée AH = 4Dh

Sortie AH = Etat du EMM
ES = Segment d'adresse de la structure de données
DI = Offset d'adresse de la structure de données
BX = Nombre de pages occupées au total

La fonction charge les numéros de tous les handles actifs avec le nombre de toutes les pages occupées dans une structure de données. La longueur de la structure est donnée par le nombre de handles multiplié par 4. Les entrées de la table comprennent chacune deux mots (word). Dans le premier mot, on trouve le numéro du handle. Dans le second, on trouve le nombre de pages.

L'utilisation de la mémoire paginée

Passons maintenant à l'utilisation de la mémoire paginée EMS. En premier lieu, il faut vérifier si la mémoire EMS est installée. On peut le faire par l'intermédiaire de l'interruption 67h. Si l'on y trouve la chaîne 'EMMXXXX', cela veut dire que le gestionnaire de mémoire paginée est installé et l'on continuera alors par la détermination de la version. Cette détermination est importante surtout quand on veut faire intervenir certaines fonctions qui n'étaient pas encore implémentées dans la version 3.0. Voici un petit exemple concernant la procédure d'initialisation :

```
procedure Check_for_EMS;
var emsseg : word;
    emsptr : pointer;
    emshead : EMS_Header;
begin;
  asm
    mov ax,3567h
    int 21h
    mov emsseg,es
  end;
  move(ptr(emsseg,0)^,emshead,17);
  if emshead.Identification = 'EMMXXXX' then begin;
    EMS_Existe := true;
    asm
      mov ah,40h                {Donne l'état du gestionnaire EMS}
      int 67h
      cmp ah,0
      jne @EMS_Vers_Erreur
      mov ah,46h                {Donne la version EMS}
      int 67h
      cmp ah,0
      jne @EMS_Vers_Erreur
      mov bl,al
      shr al,4
      mov bh,al                 { bh = Vers.maj }
      or  bl,0Fh                { bl = Vers.min }
      mov EMS_Version,bx
      jmp @EMS_Vers_Fin
@EMS_Vers_Erreur:
      mov EMS_Existe,1
@EMS_Vers_Fin:
    end;
  end else begin;
    EMS_Existe := false;
  end;
  end;
```

On s'efforcera ensuite d'obtenir l'adresse à laquelle s'ouvre la fenêtre EMS dans la mémoire principale. C'est bien sûr important, puisque tous les accès en mémoire se feront à cette adresse. Servez-vous pour cela de la fonction 41h du gestionnaire de mémoire paginée :

```
Function EMS_Segment_obtenir(VAR Segment : word) : byte;
VAR hseg : word;
    fresultat : byte;
begin;
  asm
    mov ah,41h
    int 67h
    cmp ah,0
    jne @EMS_Segerm_Erreur
    mov hseg,bx
    mov fresultat,0
    jmp @EMS_Segerm_Fin
@EMS_Segerm_Erreur:
    mov fresultat,ah
@EMS_Segerm_Fin:
  end;
  Segment := hseg;
  EMS_Segment_obtenir := fresultat;
  end;
```

Pour définir la quantité de mémoire disponible, utilisez la fonction servant à obtenir les pages EMS, c'est-à-dire la fonction 42h. Celle-ci fournit le nombre total de pages disponibles, mais aussi le nombre de pages encore libres parmi elles. Pour nous, la valeur intéressante sera surtout cette dernière. Elle est déposée par la fonction que nous vous présentons à l'intérieur de la variable globale EMS_Pages_libres. Cette fonction servira de fonction auxiliaire pour EMS_Free.

```
Function EMS_Obtenir_Nombre_Pages : byte;
var fresultat : byte;
begin;
  asm
    mov ah,42h
    int 67h
    cmp ah,0
    jne @EMS_ObtPages_Erreur
```

```
    mov EMS_Pages_libres,bx
    mov EMS_Pages_Insg,dx
    mov fresultat,0
    jmp @EMS_ObtPages_Fin
@EMS_ObtPages_Erreur:
    mov fresultat,ah
@EMS_ObtPages_Fin:
  end;
  EMS_Obtenir_Nombre_Pages := fresultat;
  end;
```

Si vous avez besoin dans votre programme de connaître la quantité de mémoire paginée encore libre, vous vous servirez de la fonction EMS_Free. Elle renvoie le nombre d'octets encore libres dans la mémoire paginée et elle se sert de la fonction auxiliaire EMS_Obtenir_Nombre_Pages :

```
function EMS_free  : longint;
var    aide : longint;
begin;
 EMS_Obtenir_Nombre_Pages;
 aide := EMS_Pages_Libres;
 EMS_free := aide SHL 14;
end;
```

Pour allouer de la mémoire EMS à partir de votre programme, vous pouvez programmer directement le gestionnaire de mémoire paginée EMM. Pourtant, il est plus simple de faire appel à la fonction Getmem_EMS. Vous lui transmettrez comme premier paramètre le handle de destination, pour lequel le bloc sera adressé. Comme seconde valeur, vous lui transmettrez la taille en octets du bloc à occuper. La fonction adapte ses exigences au nombre de pages correspondant et réalise l'occupation de la mémoire par l'intermédiaire de la fonction 43h.

```
Function Getmem_EMS(VAR H : EMSHandle; Size : longint) : byte;
var Fresultat : byte;
    EPages : word;
    Hhandle : word;
begin;
  EPages := (Size DIV 16384) + 1;
  asm
    mov ah,43h
```

```
    mov bx,EPages
    int 67h
    cmp ah,0
    jne @Getmem_Ems_Erreur
    mov Hhandle,dx
    mov fresultat,0
    jmp @Getmem_Ems_Fin
@Getmem_Ems_Erreur:
    mov Fresultat,ah
@Getmem_Ems_Fin:
  end;
  H := Hhandle;
  Getmem_EMS := Fresultat;
  end;
```

Si vous voulez libérer de nouveau la mémoire EMS occupée (et il faut absolument le faire à la fin du programme), utilisez simplement la fonction Freemem_EMS. En faisant intervenir la fonction EMM 45h, cette fonction libère le secteur de mémoire géré par le handle transmis.

```
Function Freemem_EMS(H : EMSHandle) : byte;
var Fresultat : byte;
begin;
  asm
    mov ah,45h
    mov dx,H
    int 67h
    mov Fresultat,ah
  end;
  Freemem_EMS := Fresultat;
  end;
```

La fonction EMS_Affecter est très importante. Elle permet en effet de savoir combien de pages de mémoire paginée ont été reproduites dans la fenêtre EMS. Comme premier paramètre, vous transmettrez à cette fonction le handle sous lequel le bloc de pages est adressé dans la mémoire paginée. Comme second paramètre, vous lui fournirez les numéros des pages à occuper dans la fenêtre EMS (0 à 3). Le numéro de page dans la mémoire paginée elle-même sera transmis à l'aide du dernier paramètre.

```
Function EMS_Affecter(H : EMSHandle;NumPage,PageEMS : word) : byte;
VAR Fresultat : byte;
begin;
  asm
    mov ah,44h
    mov al,byte ptr NumPage
    mov bx,PageEMS
    mov dx,H
    int 67h
    mov Fresultat,ah
  end;
  EMS_Affecter := Fresultat;
  end;
```

Si vous souhaitez conserver l'affectation des pages EMS pour un handle déterminé, servez-vous de la fonction EMS_Conserver_Affectation. Notez qu'une affectation conservée ne peut pas être modifiée tant que vous ne l'avez pas dûment effacée.

```
Function EMS_Conserver_Affectation(H : EMSHandle) : byte;
VAR Fresultat : byte;
begin;
  asm
    mov ah,47h
    mov dx,H
    int 67h
    mov Fresultat,ah
  end;
  EMS_Conserver_Affectation := Fresultat;
  end;
```

Par l'intermédiaire de la fonction EMS_Effacer_Affectation, vous pouvez effacer à nouveau l'affectation des pages EMS. La fonction se sert pour cela de la fonction EMM 48h. Les affectations conservées doivent être d'abord effacées pour que l'on puisse les modifier.

```
Function EMS_Effacer_Affectation(H : EMSHandle) : byte;
VAR Fresultat : byte;
begin;
  asm
    mov ah,48h
```

```
    mov dx,H
    int 67h
    mov Fresultat,ah
  end;
  EMS_Effacer_Affectation := Fresultat;
  end;
```

La fonction RAM_2_EMS fournit un exemple de dépôt des données dans la mémoire paginée. Elle permet de copier un bloc de la mémoire DOS dans la mémoire paginée. Prenez cette fonction modèle pour vos propres routines et utilisez-la telle quelle dans la pratique. Si le temps vous est précieux, il vaut mieux reformuler la routine spécialement pour chaque utilisation, car la méthode ici présentée sert uniquement à déposer les données et non pas à accéder directement aux pages concernées.

```
Function RAM_2_EMS(q : pointer; H : EMSHandle; Size : longint) :
                   byte;
VAR fresultat : byte;
    EMSseg    : word;
    hp        : ^byte;
    li        : word;
begin;
  EMS_Segment_Obtenir_(EMSseg);
  hp := q;
  if Size > 16384 then begin;
    {Il faut plus d'une page}
    for li := 0 to (Size SHR 14)-1 do begin;
      EMS_Affecter(H,0,li);
      move(hp^,ptr(EMSseg,0)^,16384);
      dec(Size,16384);
      inc(hp,16384);
    end;
    EMS_Affecter(H,0,li+1);
    move(hp^,ptr(EMSseg,0)^,16384);
    dec(Size,16384);
    inc(hp,16384);
  end else begin;
    EMS_Affecter(H,0,0);
    move(hp^,ptr(EMSseg,0)^,16384);
    dec(Size,16384);
```

```
    inc(hp,16384);
  end;
  end;
```

En vous servant de la fonction EMS_2_RAM, vous pouvez également copier en retour un bloc de la mémoire paginée dans la mémoire DOS.

```
Function EMS_2_RAM(q : pointer; H : EMSHandle; Size : longint) :
                   byte;
VAR Fresultat : byte;
    EMSseg    : word;
    hp        : ^byte;
    li        : word;
begin;
  EMS_Segment_Obtenir(EMSseg);
  hp := q;
  if Size > 16384 then begin;
    { Plus d'une page nécessaire }
    for li := 0 to (Size SHR 14)-1 do begin;
      EMS_Affecter(H,0,li);
      move(ptr(EMSseg,0)^,hp^,16384);
      dec(Size,16384);
      inc(hp,16384);
    end;
    EMS_Affecter(H,0,li+1);
    move(ptr(EMSseg,0)^,hp^,16384);
    dec(Size,16384);
    inc(hp,16384);
  end else begin;
    EMS_Affecter(H,0,0);
    move(ptr(EMSseg,0)^,hp^,16384);
    dec(Size,16384);
    inc(hp,16384);
  end;
  end;
```

Si vous avez besoin de savoir dans votre programme combien de pages EMS sont occupées par un handle déterminé, vous pouvez vous servir de la fonction 4Ch du gestionnaire de mémoire paginée EMM. Transmettez à la fonction EMS_pages_occupees comme premier paramètre le numéro du handle à

analyser et comme second paramètre la variable dans laquelle doit être déposé le nombre de pages.

```
Function EMS_Pages_occupees(H : EMSHandle;var Pages : word) : byte;
var fresultat : byte;
    Hs : word;
begin;
  asm
    mov ah,4Ch
    mov dx,H
    int 67h
    mov HS,bx
    mov fresultat,ah
  end;
  Pages := Hs;
  EMS_Pages_occupees := Fresultat;
  end;
```

Une dernière fonction qui vous sera utile est celle qui transmet les handles EMS occupés. Malheureusement, les handles EMS ne sont pas disponibles en nombre illimité. Le nombre se limite le plus souvent à 32. C'est pourquoi vous devez vous efforcer de réserver des blocs de la plus grande taille possible, pour l'adresse au moyen d'un handle unique.

```
Function EMS_Handles_disponibles(Var Nombre : word) : byte;
Var Fresultat : byte;
    Han       : word;
begin;
  asm
    mov ah,4Bh
    int 67h
    mov Han,bx
    mov Fresultat,ah
  end;
  Nombre := Han;
  EMS_Handles_disponibles:= Fresultat;
  end;
```

11.3. XMS - Toute la mémoire sera à toi...

Le standard XMS a été lui aussi porté sur les fonds baptismaux par les firmes Microsoft, Lotus et Intel. La société AST a également participé à son développement. Contrairement au standard EMS, XMS n'est pas basé sur une carte d'extension matérielle. Les fonctions du standard XMS sont fournies par un gestionnaire XMS, qui passe par l'interruption 2Fh. Le gestionnaire XMS le plus connu est sans doute HIMEM.SYS, intégré à CONFIG.SYS et fourni en même temps que DOS ou Windows.

Les codes d'erreur XMS

Les fonctions du gestionnaire ne sont pas adressées au moyen d'une interruption, mais en tant que Far-Call. Pour obtenir l'adresse du gestionnaire XMS, vous devez d'abord vérifier s'il y a bien un gestionnaire XMS installé. Vous le ferez en appelant l'interruption 2Fh avec la valeur 4300h dans le registre AX. Si la valeur 80h est renvoyée dans le registre AL, cela veut dire qu'il y a un gestionnaire installé. Après cela, il faut appeler à nouveau l'interruption, cette fois avec la valeur 4310h dans le registre AX. Vous obtenez l'adresse segment du gestionnaire dans le registre ES et l'adresse offset dans le registre BX.

Quand vous avez l'adresse du gestionnaire XMM (Extended Memory Manager), vous pouvez appeler ses fonctions par l'intermédiaire d'un Far-Call. Lorsque la fonction a été traitée correctement, elle renvoie le plus souvent dans AX le code d'état 0001h. Si en revanche une erreur est survenue dans l'exécution, AC contient la valeur 0000h. Dans ce cas, le registre BL contient un code d'erreur. Les codes d'erreur possèdent les significations suivantes :

BL	Signification
80h	La fonction est inconnue
81h	Disque RAM VDISK trouvé
82h	Erreur sur le conduit d'adresse A20
8Eh	Erreur générale du gestionnaire
8Fh	Erreur irréparable
90h	HMA introuvable
91h	HMA déjà affecté
92h	La mémoire indiquée dans DX est trop faible
93h	HMA non alloué
94h	Le conduit d'adresse A20 est encore en service

BL	Signification
A0h	Plus de XMS disponible
A1h	Tous les handles XMS sont occupés
A2h	Handle invalide
A3h	Handle source invalide
A4h	Offset source invalide
A5h	Handle cible invalide
A6h	Offset cible invalide
47h	Longueur invalide pour la fonction Move
A8h	Recouvrement non autorisé pour la fonction Move
A9h	Parity-Error
AAh	UMB non bloqué
ABh	UMB bloqué
ACh	Dépassement du compteur de blocage UMB
ADh	UMB impossible à bloquer
B0h	UMB plus petit disponible
B1h	Plus d'UMB disponible
B2h	Adresse segment UMB invalide

La programmation XMS

Les numéros des fonctions XMS sont toujours transmis dans le registre AH. Nous allons passer en revue les fonctions essentielles du gestionnaire de mémoire étendue XMM. Si les informations fournies vous semblent insuffisantes ou encore si vous avez besoin d'une liste complète des fonctions XMS, nous vous recommandons par exemple de lire "La Bible du PC", parue chez Micro Application.

Fonction 00h - Obtenir le numéro de version

Entrée	AH = 00h
Sortie	AX = Numéro de version XMS BX = numéro interne de révision DX = Etat de la mémoire HMA

Cette fonction fournit le numéro de version du gestionnaire. Veillez à travailler avec un gestionnaire ultérieur à la version 2.0. Si l'état HMA prend la valeur 1, la mémoire HMA est disponible. Sinon, il n'est pas possible d'y accéder.

Fonction 03h - Activer A20 globalement

Entrée	AH = 03h
Sortie	AX = Code d'erreur

Quand vous voulez utiliser la mémoire HMA également en mode réel, vous devez appeler cette fonction avant qu'elle soit exigée par cette mémoire. N'oubliez pas de refermer le conduit d'adresse avant la conclusion du programme, afin d'éviter les dépassements de segment avec les programmes qui viendront ensuite.

Fonction 04h - Fermer A20 globalement

Entrée	AH = 04h
Sortie	AX = Code d'erreur

Cette fonction permet de refermer le conduit d'adresse A20.

Fonction 05h - Activer A20 localement

Entrée	AH = 06h
Sortie	AX = Code d'erreur

Cette fonction permet d'activer le conduit d'adresse localement. Cela veut dire qu'il est activé uniquement lorsque la fonction n'a pas déjà été appelée auparavant. La vérification se fait par l'intermédiaire d'un compteur d'appels, incrémenté par la fonction 05h et décrémenté par la fonction 06h.

Fonction 06h - Désactiver A20 localement

Entrée	AH = 06h
Sortie	AX = Code d'erreur

C'est l'inverse de la fonction 05h. Elle désactive localement le conduit d'adresse A20.

Fonction 07h - Interroger l'état de A20

Entrée	AH = 07h
Sortie	AX = Etat du conduit d'adresse

Lorsque le conduit d'adresse A20 a été libéré, vous obtenez dans AX la valeur 001h, sinon vous obtenez la valeur 000h.

Fonction 08h - Obtenir une mémoire XMS libre

Entrée	AH = 08h
Sortie	AX = Longueur du plus grand bloc libre DX = Taille totale de la mémoire XMS en Ko

Cette fonction fournit la taille de la mémoire XMS libre. Mais faites attention ! Les valeurs retournées ont toujours 64 Ko de trop, puisqu'elles comprennent la mémoire HMA, qui fait précisément 64 Ko.

Fonction 09h - Allouer un bloc de mémoire étendue

Entrée	AH = 09h DX = Taille du bloc en Ko
Sortie	AX = Code d'erreur DX = handle pour l'accès ultérieur au bloc EMB

La fonction réserve un bloc de mémoire étendue (Extended Memory Block, EMB) dans la mémoire XMS. L'allocation réussit uniquement dans le cas où il existe un bloc de taille suffisante dans la mémoire étendue. Le bloc occupé peut être adressé par l'intermédiaire du handle retourné par la fonction. N'oubliez pas de libérer la mémoire occupée avant de conclure le programme. Sinon, elle ne sera pas accessible aux programmes qui suivront.

Fonction 0Ah - Libérer un bloc de mémoire étendue

Entrée	AH = 0A DX = Handle
Sortie	AX = Code d'erreur

Par l'intermédiaire de cette fonction, vous libérez un bloc EMB alloué. Après appel de cette fonction, le handle devient invalide et les données sont définitivement supprimées.

Fonction 0Bh - Copier la mémoire

Entrée	AH = 0Bh DS = Segment de la structure de copie SI = Offset de la structure de copie
Sortie	AX = Code d'erreur

Cette fonction permet de copier la mémoire dans XMS ou à partir de XMS. Puisque les registres ne suffiraient pas à emmagasiner toutes les données, la fonction se sert d'une structure de copie qui a la forme suivante :

Offset	Fonction	Type de données
00h	Longueur du bloc à copier	DWord
04h	Handle du bloc source	Word
06h	Offset dans le bloc source, copié à partir d'ici	DWord
0Ah	Handle du bloc cible	Word
0Ch	Offset dans le bloc cible, copié à partir d'ici	DWord

S'il se produit un chevauchement des secteurs, le secteur source a la priorité sur le secteur cible. Sinon, la fonction ne fonctionnera pas correctement. Pour que la copie soit plus rapide, il est préférable de faire commencer les secteurs à une adresse divisible par 4.

Fonction 0Ch - Protéger EMB contre le déplacement

Entrée	AH = 0Ch DX = Handle
Sortie	AX = Code d'erreur DX:BX = Adresse linéaire 32 bits du bloc EMB en mémoire

Pour qu'il n'apparaisse pas des trous dans la mémoire étendue, le gestionnaire de mémoire étendue XMM déplace au besoin certains blocs de mémoire. Pour protéger un bloc contre le déplacement (ce qui peut être utile en cas d'accès direct à la mémoire), on utilise cette fonction. Elle renvoie dans DX:BX l'adresse linéaire sous laquelle vous pouvez adresser la mémoire.

Fonction 0Dh - Débloquer un EMB protégé

Entrée	AH = 0Dh DX = Handle
Sortie	AX = Code d'erreur

Un bloc protégé par l'intermédiaire de la fonction 0Ch sera débloqué à l'aide de cette fonction.

Fonction 0Eh - Obtenir des informations sur un bloc EMB

Entrée	AH = 0Eh DX = Handle
Sortie	AX = Code d'erreur DX = Longueur du bloc

Par l'intermédiaire de cette fonction, vous pouvez obtenir des informations sur un bloc de mémoire, mais aussi connaître le nombre de handles XMS encore libres. Quand la fonction a rempli sa tâche sans erreur, vous trouverez dans AX la valeur 0001h. Dans ce cas, BH contient le compteur de blocage du bloc de mémoire étendue. Le compteur de blocage est incrémenté à chaque appel de la fonction 0Ch et à nouveau décrémenté à chaque appel de la fonction 0Dh. Lorsque la valeur de BH est donc supérieure à 0, vous devez appeler la fonction 0Dh "BH" fois pour libérer le bloc. Dans BL se trouve le nombre de handles libres. Veillez à réserver des blocs de mémoire aussi grands que possible, car le nombre de handles XMS est limité. Quand AX contient la valeur 0000h, c'est qu'une erreur s'est produite. La cause de l'erreur sera déterminée au moyen du tableau des erreurs. Quand la fonction a été exécutée correctement, DX contient la longueur du bloc, exprimée en Ko.

Fonction 0Fh - modifier la taille du bloc de mémoire étendue

Entrée	AH = 0Fh BX = Nouvelle taille en Ko DX = Handle
Sortie	AX = Code d'erreur

Quand vous voudrez modifier après coup la taille d'un bloc de mémoire étendue, vous pourrez le faire au moyen de cette fonction. Afin de pouvoir modifier le bloc, il faut que celui-ci ne soit pas protégé. Lorsque vous réduisez le bloc, les données qui se trouvent en haut sont irrémédiablement perdues.

Maintenant que vous avez fait connaissance avec les fonctions les plus importantes du gestionnaire de mémoire étendue, passons à la pratique. Nous développerons des routines permettant d'occuper et de libérer la mémoire XMS, de copier des données dans la mémoire ou de les chercher dans la mémoire.

Commençons avec une fonction qui vous permettra de vérifier si le système XMS est installé. La procédure vérifie d'abord si XMM existe et vous transmet le cas échéant l'adresse de début du gestionnaire. Elle vérifie ensuite s'il s'agit d'un ancien gestionnaire (numéro de version inférieur à 2.0). Une valeur en conséquence est alors transmise à la variable XMS_Existe.

```
Procedure Check_for_XMS; assembler;
  asm
    mov ax,4300h          {Vérifie s'il y a un gestionnaire installé}
    int 2Fh
    cmp al,80h
    jne @Pas_de_gestXMS
```

```
    mov ax,4310h          {Donne l'adresse d'entrée du gestionnaire}
    int 2Fh
    mov word ptr XMST + 2,es
    mov word ptr XMST + 0,bx
    xor ax,ax             {Donne le numéro de version}
    call dword ptr [XMST]
    cmp  ax,0200h
    jb   @Pas_de_gestXMS  { Si Version < 2.0, arrêt ! }
    mov  XMS_Version,ax
    mov  XMS_Existe,0
@Pas_de_gestXMS:
    mov XMS_Existe,1
@Fin_XMS_Check:
  end;
```

Avant de vous servir de la mémoire XMS, vous devez vérifier la quantité de mémoire disponible. Vous pouvez le faire à l'aide de cette fonction XMS_free.

```
function XMS_free  : longint;
var xms_en_kb : word;
    xms_long: longint;
begin;
  asm
    mov ax,0800h          { 8 = donne la mémoire libre}
    call dword ptr [XMST]
    mov xms_en_kb,dx
  end;
  xms_long := xms_en_kb;
  XMS_free := xms_long * 1024;
  end;
```

Maintenant que vous connaissez la quantité de mémoire disponible, vous pouvez réserver celle-ci. Utilisez pour cela la fonction suivante, nommée Getmem_XMS. Elle suit dans son principe la fonction Pascal appelée Getmem. Le premier paramètre transmis sera la variable contenant le handle qui sert à accéder à la mémoire allouée. Le second paramètre désigne la taille du bloc à réserver, exprimée en octets.

```
Function Getmem_XMS(VAR H : XMSHandle; Size : longint) : byte;
var bsize : word;
    Fresult : byte;
    xmsh : word;
begin;
  bsize := (size DIV 1024) + 1;
  asm
    mov ax,0900h                { 9 = allouer l'espace en mémoire }
    mov dx,bsize
    call dword ptr [XMST]
    cmp ax,1
    jne @Erreur_GetmemXms
    mov xmsh,dx
    mov Fresult,0
    jmp @Fin_GetmemXms
@Erreur_GetmemXMS:
    mov Fresult,bl
@Fin_GetmemXms:
  end;
  h := xmsh;
  Getmem_Xms := Fresult;
  end;
```

Au plus tard à la fin du programme, vous devrez libérer toute la mémoire XMS que vous avez réservée. Vous pouvez le faire en vous servant de la fonction Freemem_XMS. La fonction a besoin simplement du numéro de handle propre au bloc de mémoire étendue.

```
Function Freemem_XMS(H : XMSHandle) : byte;
var fresult : byte;
begin;
  asm                       { A = libère l'espace réservé en mémoire }
    mov ax,0a00h
    call dword ptr [XMST]
    cmp ax,1
    jne @Erreur_FreememXms
    mov Fresult,0
    jmp @Fin_FreememXms
@Erreur_FreememXms:
```

```
    mov Fresult,bl
@Fin_FreememXms:
  end;
  end;
```

Toute la mémoire allouée ne nous servira à rien si nous ne pouvons copier les données qu'elle contient ou copier des données en elle. C'est pourquoi nous vous présentons les deux fonctions Ram_2_XMS et XMS_2_Ram. La première permet de copier des données de la mémoire principale dans XMS. La seconde fait l'opération inverse. Notez que vous devez tout d'abord allouer un bloc de mémoire étendue par l'intermédiaire de la fonction Getmem_XMS, avant de pouvoir copier des données dans la mémoire !

```
Function RAM_2_XMS(q : pointer; h : XMSHandle; Size : Word) : byte;
VAR fresult : byte;
begin;
  XC.Size     := Size;
  XC.Q_Handle := 0;                          { 0 = RAM }
  XC.Q_Offset := q;
  XC.Z_Handle := h;
  XC.Z_Offset := nil;
  asm
    mov si,offset XC
    mov ax,0B00h
    call dword ptr [XMST]
    cmp ax,1
    jne @Erreur_RAM2XMS
    mov fresult,0
    jmp @Fin_Ram2XMS
@Erreur_Ram2XMS:
    mov fresult,bl
@Fin_Ram2XMS:
  end;
end;
Function XMS_2_Ram(d : pointer; h : XMSHandle; Size : Word) : byte;
VAR fresult : byte;
begin;
  XC.Size     := Size;
  XC.Q_Handle := h;
  XC.Q_Offset := nil;
```

```
  XC.Z_Handle := 0;                          { 0 = RAM }
  XC.Z_Offset := d;
  asm
    mov si,offset XC
    mov ax,0B00h
    call dword ptr [XMST]
    cmp ax,1
    jne @Erreur_XMS2RAM
    mov fresult,0
    jmp @Fin_XMS2Ram
@Erreur_XMS2Ram:
    mov fresult,bl
@Fin_XMS2Ram:
  end;
  end;
```

La dernière fonction que nous présenterons ici s'appelle XMS_2_XMS. Elle offre la possibilité de copier des données à l'intérieur de XMS. Cette opération sera utile par exemple quand vous voudrez charger une image à partir d'un disque pour la dupliquer. La fonction attend comme premier paramètre le handle du bloc source et comme second paramètre le handle du bloc cible. Le dernier paramètre transmis sera la taille en octets.

```
Function XMS_2_XMS(h1,h2 : XMSHandle; Size : Word) : byte;
VAR fresult : byte;
begin;
  XC.Size     := Size;          { Taille du bloc en octets }
  XC.Q_Handle := h1;            { Handle source }
  XC.Q_Offset := nil;           { Offset source, 0 = Début du bloc }
  XC.Z_Handle := h2;            { Handle cible }
  XC.Z_Offset := nil;           { Offset cible }
  asm
    mov si,offset XC
    mov ax,0B00h
    call dword ptr [XMST]
    cmp ax,1
    jne @Erreur_RAM2XMS
    mov fresult,0
    jmp @Fin_Ram2XMS
@Erreur_Ram2XMS:
```

```
    mov fresult,bl
@Fin_Ram2XMS:
  end;
  end;
```

11.4. Le modèle de mémoire Flat - la solution de vos problèmes de mémoire

L'utilisation de XMS constitue dans beaucoup de cas une solution acceptable pour les problèmes de mémoire. Elle présente pourtant un petit inconvénient : on ne peut pas accéder directement à la mémoire XMS. On doit toujours effectuer des copies par bloc dans la mémoire XMS ou à partir d'elle. C'est non seulement pénible, mais cela demande aussi du temps !

Si vous voulez donc accéder directement à la mémoire ou si vous voulez mettre l'accent sur la rapidité, vous devez vous servir du modèle Flat. Celui-ci vous offre la possibilité d'accéder linéairement à la mémoire du PC en sa totalité. Il est supporté par le nouveau Pascal 8.0 de Borland. Si vous travaillez avec une version antérieure, vous serez malheureusement obligé de prévoir une solution avec les moyens du bord.

L'idée qui se trouve derrière le modèle Flat repose sur une astuce : on fait d'abord passer le processeur (il faut que ce soit au moins un 386) en mode protégé. Les limites des segments pour les registres FS et GS passent alors à 4 Go. On ramène ensuite le processeur en mode réel sans reset. On dispose ainsi théoriquement de 4 Go. C'est suffisant pour commencer.

Les fondements techniques du modèle Flat

Quels sont les présupposées techniques de cette méthode ? Tout d'abord, nous devons nous occuper un peu de la gestion de la mémoire en mode protégé. Dans ce mode, la mémoire n'est pas adressée directement. Elle l'est par l'intermédiaire de ce qu'on appelle des "sélecteurs". Ceux-ci sont des pointeurs sur un "descripteur". Dans les descripteurs, on trouve toutes les informations nécessaires concernant un segment quelconque : sa taille, son adresse physique et les droits d'accès dont il jouit. Quand on allume l'ordinateur, la taille d'un segment est définie par défaut sur 0FFFFh. Ce sont les 65536 octets magiques auxquels vous pouvez accéder normalement sous DOS. En mode protégé, vous n'êtes pas soumis à cette limite. Vous pouvez adresser au maximum 4 Go. Vous devez alors adapter le descripteur pour chaque segment. Cette adaptation se fait par l'intermédiaire de la table

globale des descripteurs (Global Descriptor Table, soit en abrégé : GDT). Les informations concernant les segments se trouvent dans cette table.

Si vous voulez maintenant passer dans le modèle Flat, vous devez d'abord construire une table GDT. Il faut comprendre celle-ci comme un tableau (array) de descripteurs. La structure d'un descripteur est la suivante :

Nombre de bits	Signification
16 bits	Longueur du segment
24 bits	Adresse physique
8 bits	Droits d'accès
16 bits	Bits supplémentaires pour la longueur du segment

La GDT doit d'abord contenir un segment nul (toutes les valeurs égales à 0) pour nos besoins. Ensuite vient le segment modifié pour l'accès à 4 Go de mémoire. Les 16 bits inférieurs seront définis sur 0FFFFh. Les 16 derniers bits pour la longueur du segment contiennent la valeur 0FFCFh. L'adresse physique sera placée sur 0. De cette façon, le segment commencera au début de la mémoire. Les droits d'accès seront définis avec la valeur 92h, ce qui correspond à la priorité la plus élevée pour un segment de données.

Une fois que nous aurons ainsi construit une table GDT, il ne restera plus qu'à la charger à l'aide de la commande "lgdi". Après avoir "démasqué" les interruptions par l'intermédiaire de "cli" (pour qu'il n'y ait pas d'interférence), nous pouvons alors passer en mode protégé. Cela fait, vous pouvez effacer l'"execution pipe" du processeur au moyen d'un saut. Ceci est important, car sans cela le système se bloquerait, du fait qu'il possède encore des commandes dans la "Prefetch-Queue". Adaptez maintenant les segments aux 4 Go, en envoyant aux registres de segment les sélecteurs qui leur correspondent. Une fois cela réglé, vous pouvez revenir au mode réel, sans effectuer de reset. Ici aussi, ce retour doit être immédiatement suivi d'un saut. Pour finir, n'oubliez pas d'activer à nouveau les interruptions.

Comment on programme le modèle Flat

Ces explications devront s'avérer suffisantes. Voici maintenant une procédure en assembleur, à l'aide de laquelle vous pourrez initialiser le modèle Flat. La GDT est déclarée de manière externe.

```
;*************************************************************************
***                                                                    ***
;***          Fait passer le processeur en modèle flat                  ***
;***                                                                    ***
;*************************************************************************
Enable_4Giga proc near
    mov GDT_Off[0],16
    mov eax,seg GDT
    shl eax,4
    mov bx,offset GDT
    movzx ebx,bx
    add eax,ebx
    mov dword ptr GDT_Off[2],eax
    lgdt pword ptr GDT_Off      ; charge GDT
    mov bx,08h                  ; bx pointe la première entrée de la
                                  table GDT
    push des
    cli                         ; Désactive les interruptions
    mov eax,cr0                 ; Passe en Protected mode
    or eax,1
    mov cr0,eax
    jmp En_Protectedmode        ; Efface Executionpipe
    En_Protectedmode:
    mov gs,bx                   ; Adapte les segments à 4 GB
    mov fs,bx
    mov es,bx
    mov des,bx
    and al,0FEh                 ; Retour en Real-mode sans
    mov cr0,eax                 ; reset du processeur
    jmp En_Realmode             ; Efface Executionpipe
    En_Realmode:
    sti                         ; Active à nouveau les interruptions
    pop des
  ret
Enable_4Giga endp
code ends
  END
```

Nous avons ensuite besoin d'une procédure à l'aide de laquelle nous puissions accéder à la mémoire. Malheureusement, la RMEM (Real Memory) ne peut pas être adressée par l'intermédiaire des commandes lodsb/stosb/movsb. Il faut adresser la RMEM en passant par son adresse physique. On conserve pour cela l'adresse souhaitée dans une variable longint, au moyen de laquelle on peut accéder à la mémoire. Un accès se présentera par exemple ainsi :

```
mov ebx,pos_variable
mov al,gs:[ebx]
```

Avant d'accéder à la mémoire, vous devez encore être sûr que le 20e bit d'adresse est libre. Normalement, sa fonction est de faire revenir au début de la mémoire. Quand on accède à la mémoire étendue par exemple à l'aide d'un gestionnaire XMS, le gestionnaire prend ce retour à sa charge. Vous devez cependant libérer le bit explicitement. Une programmation directe demande quelques efforts, car il faut se servir du contrôleur de clavier. Mais puisque nous faisons intervenir de toute façon le gestionnaire HIMEM, nous pouvons utiliser simplement la fonction qu'elle met à notre disposition.

Voici maintenant un exemple qui montrera comment accéder en pratique à la RMEM. La procédure s'appelle mem_write. Elle permet de copier un bloc de mémoire principale dans la RMEM. Elle attend comme premier paramètre la destination dans la RMEM. Le second paramètre doit être l'offset du pointeur sur la section source dans la mémoire principale, suivi du segment de ce même pointeur. Le dernier paramètre contiendra la longueur du bloc à copier.

```
;*********************************************************************
;***                                                               ***
;***      Copie un bloc de mémoire principale dans la RMEM         ***
;***                                                               ***
;*********************************************************************
mem_Write proc near
        push bp
        mov  bp,sp
source  equ dword ptr ss:[bp+10]
                          ; Déclaration des "variables"
ofscible  equ word ptr ss:[bp+8]
segcible  equ word ptr ss:[bp+6]
longueur   equ word ptr ss:[bp+4]
        call xms_Enable_A20
        mov  ax,segcible     ; addy de mémoire principale vers ES:SI
        mov  es,ax
```

```
        mov  di,ofscible
        xor  ax,ax              ; Adresse source RMEM vers GS:EAX
        mov  gs,ax
        mov  eax,source
        mov  cx,longueur
nloop:
        mov  bl,es:[di]         ; copie les octets
        mov  byte ptr gs:[eax],bl
        inc  eax
        inc  di
        loop nloop
        pop  bp
        ret  10
 mem_Write endp
```

La procédure inverse s'appellera mem_lire. Elle permettra de copier un bloc de la RMEM dans la mémoire principale. Vous devrez lui transmettre comme premier paramètre la position source dans la RMEM, comme second paramètre l'offset puis le segment du pointeur sur le secteur de destination dans cette mémoire principale. Le dernier paramètre indiquera la longueur à copier, exprimée en octets. Le secteur de destination peut aussi être la mémoire vidéo d'une carte VGA !

```
;*************************************************************************
;***                                                                   ***
;***       Copie un bloc de RMEM dans la mémoire principale            ***
;***                                                                   ***
;*************************************************************************
mem_Lire proc near
        push bp
        mov  bp,sp
source  equ dword ptr ss:[bp+10]
                              ; "Déclarer les variables"
ofscible  equ word ptr ss:[bp+8]
segcible  equ word ptr ss:[bp+6]
longueur   equ word ptr ss:[bp+4]
        call xms_Enable_A20
        mov  ax,segcible      ; addy mémoire principale vers ES:SI
        mov  es,ax
        mov  di,ofscible
```

```
        xor  ax,ax              ; Adresse source RMEM vers GS:EAX
        mov  gs,ax
        mov  eax,source
        mov  cx,longueur
lloop:  mov  bl,byte ptr gs:[eax]
                                ; copie les octets
        mov  es:[di],bl
        inc  eax
        inc  di
        loop lloop
        pop  bp
        ret  10
  mem_Lire endp
```

Il est parfois pénible d'appeler en Pascal à chaque fois l'assembleur Inline. De plus, il s'agit d'une simple procédure de copie sans vérification du secteur. C'est pourquoi nous vous présentons maintenant deux petites procédures en Pascal, grâce auxquelles l'accès à la RMEM devient un jeu d'enfant :

```
{*************************************************************************
 ***                                                                   ***
 ***         Copie un bloc de RMEM dans la mémoire principale          ***
 ***                                                                   ***
 *************************************************************************}
begin
 if source + longueur < Rmem_Max then begin
   Segm:=seg(cible^);
   Offs:=ofs(cible^);
   inc(Segm,Offs div 16);
   Offs:=Offs mod 16;
   inc(source,MByte1);
   mem_lire(source,Offs,Segm,longueur);
  end else begin;
    asm mov ax,0003; int 10h; end;
    writeln('Error reading back XMS Realmemory !');
    writeln('System halted');
    halt(0);
  end;
end;
```

```
procedure Rmem_write(source:pointer;cible:longint;longueur:word);
{
 ************************************************************************
 ***                                                                  ***
 ***         Copie un bloc de mémoire principale dans la RMEM         ***
 ***                                                                  ***
 ************************************************************************
}
begin
 if cible+longueur < Rmem_Max then begin
   Segm := seg(source^);
   Offs := ofs(source^);
   inc(Segm,Offs div 16);
   Offs := Offs mod 16;
   inc(cible,MByte1);
   mem_write(cible, Offs,Segm,longueur);
 end else begin;
    asm mov ax,0003; int 10h; end;
    writeln('XMS allocation error ! Not enough memory ?');
    writeln('System halted');
    halt(0);
 end;
 end;
```

Avant de mettre en œuvre les procédures RMEM, vous devez en tout cas vous assurer que vous avez suffisamment de mémoire disponible. L'erreur la plus fatale qui pourrait arriver à l'unité serait le fait qu'un gestionnaire de mémoire comme EMM386 ou QEMM ait déjà été installé. L'unité n'est pas compatible avec ceux-ci, car ils fonctionnent suivant un principe semblable. Avec la fonction Multitache_actif, vous pouvez facilement vérifier si l'un de ces programmes est actif.

```
public Multitache_actif
;************************************************************************
;***                                                                  ***
;***      Vérifie s'il y a un Multitache actif QEMM ou EMM386         ***
;***                                                                  ***
;************************************************************************
Multitache_actif proc near
   mov eax,cr0
```

```
  and ax,1
  ret
Multitache_actif endp
```

Pour vérifier s'il y a assez de mémoire disponible, vous pouvez vous servir de la procédure Memory_Checks. Elle vérifie la quantité de mémoire principale libre et la quantité de mémoire XMS. Elle utilise pour cela le gestionnaire HIMEM.

```
procedure memory_checks(minmain,minxms : word);
{
 ************************************************************************
 ***                                                                  ***
 ***     Vérifie s'il y a suffisamment de mémoire disponible          ***
 ***                                                                  ***
 ************************************************************************
}
var xmsfree,mainfree : word;
begin;
  { Fournit la quantité de mémoire EMS libre }
  xmsfree := xms_free;
  { Fournit la quantité de mémoire principale libre }
  mainfree := memavail div 1024;
  { Message s'il n'y a pas assez de mémoire libre }
  if (xmsfree < minxms) or (mainfree < minmain) then begin;
    asm mov ax,0003; int 10h; end;
    writeln('Sorry, not enough memory available !');
    writeln('           You need       Available');
    writeln('XMS :     ',minxms :6,' KB         ',xmsfree:4,' KO');
    writeln('Main:     ',minmain:6,' KB         ',mainfree:4,' KO');
    halt(0);
  end;
  end;
```

Pour gérer la mémoire RMEM, il est recommandé de construire une unité de gestion destinée à cet usage. La solution que nous vous proposons ici ne sera réellement utile que si vous savez exactement ce que vous voulez faire de cette mémoire. Ce sera par exemple le cas quand vous voudrez coder une démo. Au contraire, si vous laissez à d'autres utilisateurs, moins expérimentés que vous, le soin de manipuler la RMEM, souciez-vous de Freemem et des routines de tri.

```
function Rgetmem(Var rpos : longint;rsize : longint) : boolean;
{
 ************************************************************************
 ***                                                                  ***
 ***      Une procédure Getmem simplifiée pour RMEM                   ***
 ***                                                                  ***
 ************************************************************************
}
begin;
  if Rmemposi+rsize > Rmem_max then begin;
    Rgetmem := false;
  end else begin;
    rpos := Rmemposi;
    inc(Rmemposi,rsize);
    Rgetmem := true;
  end;
  end;
```

Pour finir, nous allons vous présenter encore deux procédures, qui vous permettront de faire passer votre système sans difficulté en mode RMEM et retour, à partir de Pascal. Il vous suffira de transmettre à la procédure enable_Realmem la quantité de mémoire RMEM que vous souhaitez, exprimée en Ko. La procédure s'occupe du reste. Elle vérifie si un gestionnaire HIMEM est installé, si la mémoire exigée est disponible et si éventuellement il y a un gestionnaire multitâche actif. Enfin, elle réserve la mémoire nécessaire et fait passer en mode RMEM.

```
procedure enable_Realmem(Min : word);
{
 ************************************************************************
 ***                                                                  ***
 ***              Fait passer en mode RMEM                            ***
 ***       Il doit y avoir "MIN" Ko de mémoire XMS libre !            ***
 ***                                                                  ***
 ************************************************************************
}
begin
 { Vérifier le Multitasker ... }
 if multitasker_actif then begin;
   asm mov ax,0003; int 10h; end;
```

```
  writeln('Processor already in V86 mode !');
  writeln('Please reboot without any EMS-drivers such as EMM386,
          QEMM etc.');
  writeln('HIMEM.SYS is required ! ');
  halt(0);
end;
{ Gestionnaire XMS installé ?  }
if not XMS_Existe then begin;
  asm mov ax,0003; int 10h; end;
  writeln('No XMS or Himem-driver available');
  writeln('Please reboot your System using HIMEM.SYS !!!');
  halt(0);
end;
{ Réserver la mémoire nécessaire }
error := Getmem_XMS(My_XmsHandle,min*1024);
if error <> 0 then begin;
  asm mov ax,0003; int 10h; end;
  writeln('Error during memory-allocation !');
  writeln('We need at least ',Min,' KB of free XMS Memory !!!');
  writeln('Please reboot your System using HIMEM.SYS');
  writeln;
  halt(0);
end;
{ Obtenir la position physique de départ }
Rmemposi := XMS_lock(My_XmsHandle);
if rmemposi < 1000000 then begin;
  asm mov ax,0003; int 10h; end;
  writeln('Error during memory-fixing !');
  writeln('We need at least ',Min,' KB of free XMS Memory !!!');
  writeln('Please reboot your System using HIMEM.SYS');
  writeln;
  halt(0);
end;
{ Libérer }
Enable_4Giga;
 end;
```

Bien entendu, vous devrez encore faire le ménage derrière vous. Cela veut dire que vous devrez libérer la mémoire XMS réservée, car sinon elle ne pourra pas être utilisée par d'autres programmes. Vous y parviendrez par l'intermédiaire de la procédure Exit_Rmem. Celle-ci doit en tout état de cause être appelée à la fin du programme quand vous avez utilisé la RMEM.

```
procedure Exit_Rmem;
{
 ************************************************************************
 ***                                                                  ***
 ***      Procédure de sortie du RMEM ; vous DEVEZ l'appeler !        ***
 ***                                                                  ***
 ************************************************************************
}
begin;
  { Libère un bloc }
  XMS_unlock(My_XmsHandle);
  { Libère la mémoire }
  Freemem_XMS(My_XmsHandle);
  end;
```

12. Dopez votre ordinateur - Programmation des éléments supplémentaires

Dans ce chapitre, vous allez apprendre tout ce qu'il est possible de savoir sur différents composants annexes, comme le Timer-Chip (PIT), le contrôleur d'interruption, le Real-Time-Clock-Chip (RTC), ainsi que le contrôleur DMA.

Vous serez ainsi en mesure d'intervenir dans le hardware de votre ordinateur, jusqu'au dernier boulon.

12.1. Le concept d'interruption

Une caractéristique des PC est fournie par ce qu'on appelle en anglais des "interrupts". Ce sont effectivement, comme leur nom l'indique, des interruptions du système. Elles peuvent être déclenchées par les circonstances les plus variées. Il existe des interruptions déclenchées par le matériel intégré. D'autres le seront par la partie logicielle ou par le système d'exploitation. D'autres enfin pourront l'être par les périphériques. Nous devons différencier les interruptions "masquables" et celles qui ne le sont pas (ces dernières sont nommées en abrégé NMI). Vous trouverez des informations plus détaillées là-dessus par la suite. Ce paragraphe va s'occuper des principes de base concernant la programmation des interruptions et vous indiquer diverses possibilités de mise en œuvre.

Quand vous appelez une interruption, le traitement du programme en cours est interrompu pour un instant. Un segment de code correspondant à cette interruption est alors exécuté. Ce segment de code se termine par la com-

mande en assembleur nommée "iret", qui est simplement une abréviation de "Interrupt Return". A l'appel d'une interruption, les registres FLAG, CS et IP sont déposés sur la pile. De cette façon, vous pouvez savoir simplement à partir de quelle position dans le programme principal l'interruption a été déclenchée. Si vous travaillez uniquement en Pascal, vous n'avez pas besoin de vous occuper de ces subtilités. Pascal vous propose quelques procédures permettant de traiter très simplement les interruptions.

Dans la mémoire principale, il existe une table dans laquelle sont reportées les adresses des procédures appelées quand l'interruption correspondante est déclenchée. Ces adresses ne sont rien d'autre que des pointeurs. Les pointeurs en question ne possèdent sans doute pas de secret pour vous si vous travaillez quotidiennement en Pascal. Si vous voulez par exemple affecter à un pointeur l'adresse de votre procédure, vous utilisez l'instruction :

```
My_Pointer:=ADR(La_Procedure)
```

ou plus brièvement :

```
My_Pointer:=@La_Procedure
```

Cette manipulation simple est supportée par Pascal dans la gestion des interruptions. Pour obtenir l'adresse d'une interruption déterminée, par exemple de l'interruption Timer, il vous suffit d'entrer la commande :

```
VAR   Pointer_sur_Timer : pointer;
GetIntvec(8,Pointer_sur_Timer)
```

Pour détourner une interruption vers une routine de votre choix, entrez ce qui suit :

```
SetIntvec(8,@Ma_Procedure_Timer)
```

Voici maintenant quelque chose de très important. Derrière la déclaration de la procédure pour la routine d'interruption, vous devez dans tous les cas écrire le mot clé "Interrupt". Ceci est nécessaire pour que Pascal exécute à la fin de la routine d'interruption la commande "iret" et non pas la commande normale "ret". La déclaration pourra se présenter comme suit :

```
Procedure Ma_Procedure_Timer; Interrupt;
begin;
    writeln('Hello ! Ceci est une interruption !');
    port[$20] := $20;
end;
```

Notez que l'interruption 8 est une interruption matérielle. A la fin d'une telle interruption, vous devez faire savoir à un contrôleur que l'interruption a été traitée jusqu'au bout. Envoyez pour cela au contrôleur situé au port 20h la commande EOI (End of Interrupt, = 20h).

Vous vous demanderez sans doute à quoi tout cela peut bien servir. Nous allons le préciser en prenant l'exemple de l'interruption Timer. Vous voulez écrire un programme qui affiche toutes les demi-seconde le caractère ASCII #1 et toutes les demi-secondes le caractère ASCII #2. Vous pouvez évidemment placer dans une boucle du programme principal un examen de l'heure système et demander l'affichage d'un caractère toutes les demi-secondes. Cela rendrait toutefois impossible le traitement d'autres parties du programme, par exemple le tri simultané d'une liste à 10.000 entrées.

L'autre moyen est de détourner sur une routine de votre composition l'interruption Timer, qui est appelée 18,2 fois par seconde. Dans cette routine, vous pouvez alors faire fonctionner un compteur, qui compte précisément combien de fois l'interruption a été appelée. Chaque fois que le compteur arrive à neuf, vous pouvez demander l'affichage d'un caractère et remettre le compteur à 0. De cette façon, il vous est possible de trier simultanément vos 10.000 entrées, de sortir le COMMAND.COM en sens inverse ou de monter et descendre l'échelle des sons par l'intermédiaire de la commande Sound. Les caractères s'affichent avec l'exactitude d'une horloge suisse. N'oubliez pas simplement de rendre l'interruption à sa routine d'origine à la fin du programme. Cette routine vous sera transmise par GetIntvec. Sachez aussi que la tentative de déboguer pas à pas un programme contrôlé par une interruption se termine en général piteusement !

Maintenant que vous savez comment programmer vos propres interruptions en Pascal, pénétrons un peu plus avant dans cette matière. La question est la suivante : que font réellement les procédures GetIntvec et SetIntvec ? Réponse : elles se servent des fonctions DOS 25h et 35h de l'interruption DOS 21h. Nous nous trouvons aussitôt devant un phénomène nouveau : une interruption ne possède pas seulement une seule routine. Elle peut avoir plusieurs sous-fonctions. L'interruption 21h est mise à votre disposition par DOS pour toutes les fonctions importantes du système d'exploitation. Pour faire savoir à l'interruption quelle sous-fonction vous voulez exécuter, vous ne pouvez pas éviter l'assembleur (Inline). Vous êtes en effet obligé d'écrire le numéro de la fonction d'interruption souhaitée dans le registre AH et d'écrire éventuellement des paramètres dans les autres registres, qui dépendent de la sous-fonction.

Pour la fonction 25h, vous devez ainsi initialiser les registres suivants :

Registre	Contenu/Signification
AH	25h
AL	Numéro de l'interruption à définir
DS	Adresse segment de votre procédure d'interruption
DX	Adresse offset de votre procédure d'interruption

Avec la fonction 35h, vous devez bien sûr entrer des valeurs mais vous recevez aussi des valeurs en retour. La fonction sert en effet à obtenir l'adresse de la routine actuelle d'interruption. Il faut donc bien que vous puissiez faire venir le résultat quelque part. Voici comment les registres sont occupés :

Entrée	AH = 35h AL = Numéro de l'interruption à obtenir
Sortie	ES = Adresse segment de la routine BX = Adresse de la routine

Comment se présente alors une routine en assembleur destinée à obtenir une interruption ?

```
Model TPascal
.data
int_seg dw ?
int_ofs dw ?
.code
get_interrupt proc pascal int_nr : BYTE
  mov ah,35h
  mov al,int_nr
  int 21h
  mov int_ofs,bx
  mov bx,es
  mov int_seg,es
  ret
get_interrupt endp
```

La chose devient un peu plus compliquée quand vous ne remplacez pas entièrement l'ancienne interruption. Supposons que vous vouliez ainsi exécuter d'abord votre propre routine et que vous vouliez appeler ensuite l'interruption originale. Dans ce cas, vous êtes obligé de faire venir d'abord l'adresse de l'interruption originale et de la déposer dans une variable DWord (Pointeur). Vous pouvez alors détourner l'interruption vers votre propre routine.

Dans celle-ci, tous les registres modifiés qui présentent une signification pour l'interruption finale doivent tout d'abord être sauvegardés ! Ensuite, vous pourrez faire intervenir votre routine. Vous pouvez par exemple vérifier si une touche a été pressée ou si une valeur déterminée se trouve quelque part en mémoire. Une fois que votre code a été exécuté, vous devez penser à restaurer les registres. Passez ensuite directement dans l'interruption originale. Vous n'êtes pas obligé ici d'appeler "iret". La routine originale s'en charge pour vous. La structure de l'ensemble peut se présenter ainsi :

```
.data
  oldint dd ?
.code
mon_interrupt proc far
  pusha
  mov ax,Variable_quelconque
  [...]
  popa
  jmp dword ptr oldint
mon_interrupt endp
```

Avec ces techniques, vous ne pouvez pas éviter les solutions programmées purement en assembleur, puisque Pascal sauvegarde tous les registres quand on saute dans une procédure d'interruption et, ce qui est bien pire, restaure ces registres quand on quitte à nouveau la procédure. Cela ne permet pas d'obtenir des valeurs en provenance de la procédure d'interruption.

Voici une dernière astuce dans cet ordre d'idées. Elle permettra à la nouvelle génération de mordus dans ce domaine de la programmation d'épater tout le monde. Les adresses des interprétations sont déposées dans la table des interruptions. Cette table se trouve dans la mémoire principale à l'adresse $0000:0000. Chaque entrée fait 4 octets. Cela veut dire que la valeur de l'adresse d'interruption correspond à la valeur $0000 : (Numéro_d'interruption)*4. Vous n'êtes cependant pas obligé de vous servir à chaque fois des fonctions DOS 25h et 35h lorsque vous voulez obtenir ou modifier l'adresse d'une interruption. Vous pouvez aussi la lire (ou l'écrire) directement dans la table à la position voulue. Vous avez sans doute souvent lu dans la littérature spécialisée que vous ne devez jamais écrire vous-même dans la table. Mais on vous dira aussi qu'il est recommandé de poser un pixel en mode 13h par l'intermédiaire de l'interruption 10h ! Moralité : oubliez les conseils de prudence ! On peut écrire directement dans la table sans provoquer aucune catastrophe. Vous pouvez d'ailleurs créer de cette façon une copie de sauvegarde de la table des interruptions. Au début de votre programme, copiez simplement ses 30 lignes dans un tampon et recréez-les au terme de votre programme. Vous êtes ainsi sûr que toutes les interruptions modifiées pointeront vers leurs routines originales.

12.2. LE PROGRAMMABLE INTERVAL TIMER (PIT)

Le PIT nous intéresse en relation avec la programmation des démos, par exemple pour le cas où il s'agit de synchroniser un processus déterminé ou de l'appeler périodiquement. Nous utiliserons pour cela le compteur 0 du PIT. Celui-ci contrôle l'interruption Timer du PC, avec ses 18,2 pulsations à la seconde.

Avant de voir comment on synchronise l'écran avec exactitude, nous allons dire quelques mots sur les éléments matériels qui conditionnent le Timer.

Le hardware du PIT

Le Timer-Chip utilisé est un 8254 PIT. Il contient trois compteurs 16 bits indépendants. Chacun de ces trois compteurs peut fonctionner selon l'une de ces six opérations logiques :

Mode 0	Génère l'interruption
Mode 1	Monoflop programmable
Mode 2	Générateur du battement
Mode 3	Générateur de signaux périodiques rectangulaires
Mode 4	Déclencheur de la sortie par logiciel
Mode 5	Déclencheur de la sortie par matériel

Ce sont surtout les modes de fonctionnement 1 à 3 qui nous intéresseront ici. Ils permettent de mettre en place une adaptation à certaines fréquences, un appel périodique et des événements synchronisés.

Chacun de ces trois canaux est contrôlé par un compteur qui lui est propre. Celui-ci peut fonctionner en mode binaire ou BCD. Le canal 0 contrôle le System-Timer, responsable de l'heure système, de l'heure courante ou du "Disk timeout". Le hardware du système se sert du canal 1. Il contrôle le "DRAM Refresh". Le canal 2, enfin, est utilisé pour générer le son par l'intermédiaire du speaker PC.

Le canal comprend une série de registres. Nous nous intéresserons surtout au registre compteur, aux deux registres compteurs de saisie, ZEL et ZEH, et au registre de contrôle. Il existe aussi un registre d'état et deux registres compteurs de sortie ZAL et ZAH.

La table suivante donne une vue d'ensemble sur les compteurs existants, ainsi que sur les gates et signaux utilisés.

Compteur 0	Gate 0Ch	System Timer
	Clk 0Ch	génère IRQ0
	Out 0Ch	
Compteur 1	Gate 1Ch	DRAM Refresh
	CLK 1Ch	REFTIME
	OUT 1Ch	
Compteur 2	Gate 2Ch	Fréquence du haut-parleur
	CLK 2Ch	Haut-parleur activé
	OUT 2Ch	

Le registre de contrôle

Le registre de contrôle règle l'activité des canaux. Il peut être adressé par l'intermédiaire du port 43h. Vous lui transmettrez le numéro du canal souhaité, le type de commandes, le mode de fonctionnement et le format du compteur. Par l'intermédiaire du registre d'état, vous pouvez lire les spécifications qui se trouvent actuellement dans le registre de contrôle. Voici, sous forme de tableau, la signification des différents bits de ce registre :

Bit 7-6	Sélection du compteur
00	Compteur 0
01	Compteur 1
10	Compteur 2
11	Mode de lecture inverse
Bit 5-4	Type de commande
00	Compteur latch
01	Lecture/écriture dans l'octet Lo du compteur
10	Lecture/écriture dans l'octet Hi du compteur
11	Lecture/écriture d'abord dans l'octet Lo, puis dans l'octet Hi du compteur
Bit 3-1	Choix du mode de fonctionnement
000	Mode 0, Génère l'interruption
001	Mode 1, Monoflop programmable
010	Mode 2, Générateur du battement
011	Mode 3, Générateur de signaux périodiques rectangulaires
100	Mode 4, Déclencheur de sortie par logiciel
101	Mode 5, Déclencheur de sortie par matériel

Bit 0	Choix du format de compteur
0	16 bits, Binaire (standard)
1	Format

Le registre compteur

Le registre compteur est un registre 16 bits et compte de 0 à 0ffffh. Un accès en écriture à l'un des registres compteurs d'entrée lance le compteur. Une initialisation du compte se présenterait ainsi :

```
port[43] := $36;
port[40] := 12;
port[40] := 34;
```

Lecture inverse des registres

Les données des compteurs peuvent être ramenées dans les registres compteurs. Il faut pour cela spécifier le mode de lecture inverse dans le registre des commandes en tant que sélection du compteur. On peut alors indiquer si l'on veut lire la valeur du compteur ou l'octet d'état. Le format exact de l'octet de commandes sera retiré du tableau suivant :

Bits 7-6	Indiquer commande Read Back
11	Doit avoir cette valeur
Bits 5-4	Sélection Read Back
01	Lire la valeur actuelle du compteur
10	Lire l'octet de l'état
11	Rien à lire
Bits 3-0	Sélectionner le compteur
1000	Choisir compteur 2
0100	Choisir compteur 1
0100	Choisir compteur 0

La valeur d'un octet d'état lu en retour devra être interprétée comme suit :

Bit 7	Etat du signal OUT
0	OUT Signal 0 (low)
1	OUT 1 (high)
Bit 6	Flag compteur NIL-OUT posé ?
0	Le compteur a été posé par l'intermédiaire du registre d'entrée et on peut le lire
1	Le registre de contrôle a déjà reçu des valeurs, mais ce ne sont pas les valeurs attendues
Bits 5-4	Mode de la commande de transfert
00	Réservé
01	Lire/écrire octet Lo
10	Lire/écrire octet Hi
11	Lire/écrire d'abord octet Lo, puis octet Hi
Bits 3-1	Choix du mode de fonctionnement
000	Mode 0
001	Mode 1
010	Mode 2
011	Mode 3
100	Mode 4
101	Mode 5
Bit 0	Choix du format pour le compteur
0	16 bits en binaire (standard)
1	Format BCD

Utilisation du PIT

Maintenant que vous avez fait connaissance avec le hardware du PIT, vous allez sans doute vous demander : "bon, mais à quoi cela sert-il ?". Cela sert en particulier à synchroniser certains événements. Dans la pratique, afin de programmer des démos, on a besoin d'appeler périodiquement une routine. C'est le cas pour la programmation du son ou pour la synchronisation de celui-ci avec l'écran.

Normalement, il suffit d'élaborer l'écran, puis d'attendre le balayage vertical. Pourtant, il est parfois nécessaire de changer de mode à chaque balayage, ou tous les deux ou trois balayages.

Comment pouvez-vous donc réaliser ce changement de mode ? Tout d'abord, vous devez connaître la fréquence avec laquelle l'écran est mis à jour. Dans un mode normal 320x200, cette fréquence est de 70 Hz pour presque toutes les cartes. Nous devons donc générer une interruption très peu de temps avant que l'affichage d'une image ne prenne fin, pour réaliser la synchronisation avec le balayage et effectuer le changement de mode. Cela signifie, si l'on veut que l'interruption soit générée un peu avant, que sa fréquence doit être légèrement supérieure à ces 70 Hz. Tout dépend de l'intensité des calculs dans votre routine.

Le petit programme Test_timer que nous vous présentons met en œuvre les principes que nous venons d'exposer. Il fournit d'abord la fréquence de recomposition de l'écran, en vérifiant par l'intermédiaire de l'interruption Timer de combien il peut modifier une variable en attendant le balayage. On fait ensuite passer le Timer du mode rectangulaire au mode Monoflop. La routine d'interruption doit alors reprogrammer à chaque fois le Timer. Pour visualiser l'endroit où s'effectue l'interruption, l'écran passe d'abord au rouge, puis au vert par l'intermédiaire de l'interruption. De cette façon, on voit bien comment les périodes sont distribuées. Voici le listing du programme TIMER.PAS.

```
program test_timer;

uses crt,dos;

Var OTimerInt  : pointer;
    Timerfreq    : word;
    Orig_freq    : word;
    Sync_counter : word;
    Ticompteur    : word;

PROCEDURE SetColor (Nr, R, G, B : BYTE);
begin;
  asm
    mov   al,Nr
    mov   dx,03C8h
    out   dx,al
    mov   dx,03C9h
    mov   al,r
```

```
    out   dx,al
    mov   al,g
    out   dx,al
    mov   al,b
    out   dx,al
  end;
end;

procedure waitretrace;
begin;
  asm
    MOV   DX,03dAh
@WD_R:
    IN    AL,DX
    TEST  AL,8d
    JZ    @WD_R
@WD_D:
    IN    AL,DX
    TEST  AL,8d
    JNZ   @WD_D
  end;
end;

procedure RegleTimer(Proc : pointer; Freq : word);
var icompteur : word;
    oldv : pointer;
begin;
 asm cli end;
 icompteur := 1193180 DIV Freq;
 Port[$43] := $36;
 Port[$40] := Lo(Icompteur);
 Port[$40] := Hi(Icompteur);
 Getintvec(8,OTimerInt);
 SetIntVec(8,Proc);
 asm sti end;
end;
```

```
procedure Nouvelle_Timerfreq(Freq : word);
var icompteur : word;
begin;
 asm cli end;
 icompteur := 1193180 DIV Freq;
 Port[$43] := $36;
 Port[$40] := Lo(Icompteur);
 Port[$40] := Hi(Icompteur);
 asm sti end;
end;

procedure RegleTimerOff;
var oldv : pointer;
begin;
  asm cli end;
  port[$43] := $36;
  Port[$40] := 0;
  Port[$40] := 0;
  SetIntVec(8,OTimerInt);
  asm sti end;
end;

procedure Syncro_interrupt; interrupt;
begin;
  inc(Sync_counter);
  port[$20] := $20;
end;

procedure Syncronize_timer;
begin;
  Timerfreq := 120;
  RegleTimer(@Syncro_interrupt,Timerfreq);
  Repeat
    dec(Timerfreq,2);
    waitretrace;
    Nouvelle_timerfreq(Timerfreq);
    Sync_counter := 0;
    waitretrace;
```

```
  until (Sync_counter = 0);
end;

Procedure Timer_Handling;
begin;
  setcolor(0,0,63,0);
end;

Procedure Timer_Proc; interrupt;
begin;
  Timer_Handling;
  waitretrace;
  Port[$43] := $34;                    { Mode Monoflop }
  Port[$40] := Lo(Ticompteur);
  Port[$40] := Hi(Ticompteur);
  setcolor(0,63,0,0);
  port[$20] := $20;
end;

Procedure Start_Syncrotimer(Proc : pointer);
var calcl : longint;
begin;
 asm cli end;
  port[$43] := $36;
  Port[$40] := 0;
  Port[$40] := 0;
  Ticompteur := 1193180 DIV (Timerfreq+5);
  setintvec(8,Proc);
  waitretrace;
  Port[$43] := $34;                    { Mode monoflop }
  Port[$40] := Lo(Ticompteur);
  Port[$40] := Hi(Ticompteur);
 asm sti end;
end;

begin;
  clrscr;
  Syncronize_Timer;
```

```
  writeln('La fréquence du timer est : ',Timerfreq);
  Start_Syncrotimer(@Timer_Proc);
  repeat until keypressed;
  while keypressed do readkey;
  RegleTimerOff;
  setcolor(0,0,0,0);
end.
```

12.3. LE "PROGRAMMABLE INTERRUPT CONTROLLER" (PIC)

Le PIC est également un élément important. Il contrôle les interruptions du PC. Plus précisément, le PC possède deux contrôleurs d'interruption. Le premier se trouve à l'adresse 10h, le second à l'adresse A0h. Ce second contrôleur est toutefois lié au premier par l'intermédiaire d'une liaison en cascade. Celle-ci est réalisée par l'intermédiaire du canal du contrôleur en 20h.

Le PIC fonctionne selon une logique des priorités. Cela veut dire que l'interruption ayant la valeur la plus faible, donc l'interruption 0, possède la priorité la plus élevée. La priorité la plus faible est toutefois dévolue à l'interruption 7 et non pas, comme l'on pourrait s'y attendre, à l'interruption 15. Cela tient au fait que les interruptions 8 à 15 sont intégrées par l'intermédiaire de l'interruption 2 et sont donc pourvues d'une priorité plus élevée que l'interruption 3.

Voici un tableau mettant en évidence cette logique des priorités :

Priorité	PIC n°	Int. n°	Signification
1		NMI	Toutes les interruptions non masquées
2	1	IRQ 0	Interval Timer
3	1	IRQ 1	Timer Clavier
4	2	IRQ 8	RTC
5	2	IRQ 9	Emulateur de logiciel IRQ 2
6	2	IRQ 10	Non utilisé
7	2	IRQ 11	Non utilisé

Priorité	PIC n°	Int. n°	Signification
8	2	IRQ 12	Souris ou autre entrée semblable
9	2	IRQ 13	Interruption du coprocesseur
10	2	IRQ 14	Contrôleur de disque dur
11	2	IRQ 15	Non utilisé
12	1	IRQ 3	2ème interface série
13	1	IRQ 4	1ère interface série
14	1	RIQ 5	2ème port parallèle
15	1	IRQ 6	Contrôleur de disquette
16	1	IRQ 7	1er port parallèle

Grâce à cette logique des priorités, le PC facilite le fonctionnement d'un système d'exploitation contrôlé par les interruptions et contribue à réduire l'inflation des opérations de gestion.

Nous n'entrerons pas ici dans tous les détails du PIC. Si vous avez besoin d'autres informations, vous les trouverez dans les ouvrages consacrés à la programmation du système. Nous allons plutôt nous occuper de son utilisation dans la programmation quotidienne.

Le PIC distingue quatre types d'interruptions :

- Les interruptions masquables (MI)
- Les interruptions non masquables (NMI)
- Les interruptions matérielles
- Les interruptions logicielles

Les interruptions masquables

Les interruptions masquables sont déclenchées par le hardware du PC. Elles peuvent être supprimées par utilisation de la commande en assembleur

```
cli   ;Supprimer toutes les interruptions masquables
```

Pour les activer à nouveau, vous ferez usage de la commande :

```
sti   ;Activer à nouveau toutes les interruptions
```

Vous avez en outre la possibilité de désactiver les interruptions de manière sélective. Cela se fait en les démasquant au port concerné. Pour les interrup-

tions IRQ 0 à IRQ 7, cela se fait au port 020h. Pour les autres interruptions, cela se fait au port 0A0h.

Interruptions non masquables

Les interruptions non masquables sont plus difficiles à manipuler. Elles sont appelées quand il s'est produit par exemple une erreur de parité ou quand le Bios a déclenché ces interruptions. C'est intéressant en particulier quand on exécute des opérations critiques en matière de temps et quand on ne veut pas qu'elles soient interrompues. Par l'intermédiaire de l'instruction en assembleur

```
mov al,80h
out 20h,al
```

vous pouvez désactiver les NMI. Vous les réactivez par l'intermédiaire de :

```
mov al,00h
out 70h, al
```

Interruptions matérielles

On entend par là les interruptions déclenchées par les éléments matériels périphériques. Ils génèrent le plus souvent une interruption sur l'un des canaux d'interruption libres. Leur traitement est équivalent à celui des interruptions masquables.

Interruptions logicielles

Les interruptions logicielles sont celles qui sont déclenchées par des éléments de programme. On trouve parmi celles-ci les interruptions système déclenchées par le système d'exploitation, par exemple int 21h. Il peut aussi s'agir des interruptions installées par l'utilisateur pour contrôler le hardware ou pour un traitement universel. Elles peuvent être désactivées par l'intermédiaire de la commande "cli" déjà mentionnée et réactivées par l'intermédiaire de "sti".

12.4. LE CONTRÔLEUR DMA

Le contrôleur DMA permet de transférer rapidement de grands blocs de données d'un appareil périphérique jusque dans la mémoire de l'ordinateur et inversement. La technique utilisée à cet effet est plutôt radicale : le contrôleur DMA déconnecte le processeur, puis transfère les données.

Le PC propose deux contrôleurs DMA. Le premier contrôleur est responsable des transferts 8 bits. Le second règle les transferts 16 bits. Ce dernier propose une liaison en cascade avec le premier contrôleur par l'intermédiaire du canal 0.

Voici un tableau des différents canaux :

Canal	Contrôleur	Description
Canal 0	Contrôleur 1	Ram-Refresh
Canal 1	Contrôleur 1	libre, souvent SB, GUS, etc.
Canal 2	Contrôleur 1	Lecteur de disquettes
Canal 3	Contrôleur 1	Lecteur de disque dur
Canal 4	Contrôleur 2	Cascade Contr.1 => Contr.2
Canal 5	Contrôleur 2	libre
Canal 6	Contrôleur 2	libre
Canal 7	Contrôleur 2	libre

La question qui nous intéressera dans la pratique est la suivante : comment programmer le contrôleur DMA par exemple pour qu'il puisse fournir des données a une carte Sound Blaster ?

Le procédé général est le suivant :

1. Bloquer le canal DMA
2. Spécifier le mode de transfert
3. Supprimer le flip-flop
4. Ecrire l'adresse du bloc de données
5. Ecrire la page du bloc de données
6. Ecrire la longueur du transfert
7. Libérer à nouveau le canal DMA

Masquer un canal DMA

Nous devons indiquer maintenant comment mettre en œuvre ces différentes étapes dans la pratique. Occupons-nous d'abord de savoir comment on "masque" et "démasque" un canal DMA. Pour masquer un canal DMA, il existe deux possibilités :

1. Masquer par l'intermédiaire du registre Single-Mask

Par l'intermédiaire de ce registre, on peut toujours masquer et démasquer un canal. Le registre se trouve au port 0Ah pour les canaux 0-3 et au port 0D4h pour les canaux 4-7. Les bits ont la signification suivante :

Bits 7-3	non utilisé
Bit 2	Poser ou supprimer un masque
1	Poser le masque
0	Supprimer le masque
Bits 1-0	Choix du canal
00	Canal 0 du contrôleur en cours
01	Canal 1 du contrôleur en cours
10	Canal 2 du contrôleur en cours
11	Canal 3 du contrôleur en cours

Cela veut dire que si vous voulez démasquer le canal 1 du contrôleur 1 (c'est nécessaire pour une Sound Blaster standard), vous devez écrire la valeur 5 dans le port 0Ah. Pour remettre le canal en activité après programmation du contrôleur DMA, écrivez la valeur 1.

2. Masquer par l'intermédiaire du registre All-Mask

Le contrôleur DMA régit différents modes de transfert. Au début d'un transfert DMA, vous devez spécifier le mode nécessaire. Pour cela, il faut sortir le bit du mode pour le premier contrôleur sur le port 0Bh et pour le second contrôleur sur le port 0D6h. Voici d'abord un tableau des bits qui interviennent :

Bits 7-6	Choix du mode
00	Mode Exigence
01	Mode Unique
10	Mode Bloc
11	Mode Cascade
Bits 5	Incrémenter/Décrémenter les adresses
1	Adresse décrémentée
0	Adresse incrémentée
Bit 4	Activer/désactiver l'auto-initialisation
1	Auto-initialisation activée
0	Auto-initialisation désactivée
Bits 3-2	Choix du transfert
00	Transfert Verify
01	Transfert par écriture
10	Transfert par lecture
11	invalide
Bits 1-0	Choix du canal
00	Utiliser le canal 0
01	Utiliser le canal 1
10	Utiliser le canal 2
11	Utiliser le canal 3

Voici maintenant de plus près la signification des différents bits.

Choix du mode

Par l'intermédiaire des deux bits 7-6, vous sélectionnez le mode à utiliser pour le transfert. Les quatre modes possibles sont les suivants :

- Mode Demande
- Mode Unique
- Mode Bloc
- Mode Cascade

Le mode qui nous intéresse est le mode "Unique". Dans ce mode, on transfère toujours un octet unique et on incrémente ou décrémente ensuite le compteur de l'adresse, en fonction du mode programmé. Quand le compteur atteint sa valeur cible, une auto-initialisation est exécutée si la fonction est spécifiée.

Incrémenter/décrémenter les adresses

Le bit 5 permet de spécifier si le bloc devra être transféré vers l'avant (incrémenter) ou vers l'arrière (décrémenter).

Activer/désactiver l'auto-initialisation

Avec le bit 4, vous indiquez si une auto-initialisation doit se faire au terme d'un transfert. L'effet de l'auto-initialisation est de restaurer les valeurs programmées dans le registre d'adresse et le registre du compteur. Grâce à l'action de ce bit, on peut se servir d'une astuce très utile : quand vous programmez un Player MOD pour la carte SB, vous pouvez travailler avec un "double-buffer" mais aussi avec un "ring-buffer". Si vous choisissez ce dernier, vous pouvez faire savoir à la carte SB que vous souhaitez transférer un très grand bloc (jusqu'à 0FFFFh). Vous programmerez cependant le contrôleur DMA à la taille réelle du bloc. En conséquence de quoi le bloc reprend à partir du début une fois qu'il est parvenu à son terme (principe du ring-buffer). Vous pourrez de cette manière éviter les appels à la routine d'initialisation DMA et vous appellerez l'interruption SB plus rarement. Cela procure un bénéfice certain en matière de vitesse d'exécution et réduit éventuellement les ratés qui pourraient surgir.

Choix du transfert

Par l'intermédiaire des deux bits 3-2, vous indiquerez la direction du transfert à exécuter. Un transfert en écriture signifie un transfert qui va de la source à la mémoire principale adressée. Un transfert en lecture permet au contraire de transférer des données de la mémoire principale adressée à l'emplacement cible. Un transfert "Verify" vérifie le transfert exécuté.

Choix du canal

Vous devez spécifier ici quel canal vous voulez utiliser pour l'opération de transfert. Notez que le canal DMA 4 correspond au canal 0 du second contrôleur. Les différences entre le premier et le second contrôleur tiennent ainsi uniquement aux différentes adresses de port.

La suppression du flip-flop DMA

Avant certaines actions pendant la programmation du contrôleur DMA, il est nécessaire de supprimer le flipflop. Vous le ferez au moyen du registre Clear-Flipflop. Pour le premier contrôleur, il se trouve au port 0Ch. Pour le

second contrôleur, il est situé au port 0D8h. Vous supprimez ce registre en écrivant la valeur 0 sur le port correspondant.

Spécifier l'adresse d'un bloc de données

Le contrôleur DMA a besoin de connaître la page physique du bloc de données à transférer. Celle-ci sera calculée en fonction de la formule :

```
adr:=16*longint(Seg(Block^))+ofs(Block^);
```

A partir du nombre 32 bits ainsi calculé, vous obtenez la page (16 bits supérieurs) et l'offset (16 bits inférieurs). Pour définir l'adresse, il faut ensuite écrire l'offset dans le registre "Adresse DMA". Celui-ci pourra se trouver aux ports suivants en fonction du canal.

Canal 0	Port 00h
Canal 1	Port 02h
Canal 2	Port 04h
Canal 3	Port 06h
Canal 4	Port C0h
Canal 5	Port C4h
Canal 6	Port C8h
Canal 7	Port CCh

Vous écrirez d'abord l'octet Low au port indiqué, puis l'octet High correspondant à l'offset de l'adresse.

Vous pouvez alors transférer la page dans le contrôleur. Il existe pour cela deux registres différents, un registre Lower-Page pour les 8 bits inférieurs de l'adresse et un registre Upper-Page pour les 4 bits supérieurs de l'adresse. De cette façon, on peut adresser jusqu'à 256 Ko par l'intermédiaire des pages, ce qui devrait se révéler amplement suffisant. Le registre Lower-Page remplace le contenu du registre Upper-Page par 0, quand il est programmé. C'est pourquoi il doit être programmé avec ce dernier. Le tableau suivant donne une vue d'ensemble sur les ports de ces registres.

Canal	Lower Page	Upper Page
0	87h	487h
1	83h	483h
2	81h	481h

Canal	Lower Page	Upper Page
3	82h	482h
4	0	0
5	8Bh	48Bh
6	89h	489h
7	8Ah	48Ah

Spécifier la taille d'un transfert DMA

Avant de pouvoir écrire la taille du bloc à transférer au contrôleur DMA, vous devez supprimer le flipflop. Vous pourrez alors transférer l'octet Low, puis l'octet High au registre DMA-Count. Vous spécifiez de cette façon le nombre de données à transférer. Voici les ports du registre compteur.

Canal 0	Port 01
Canal 1	Port 03
Canal 2	Port 05
Canal 3	Port 07h
Canal 4	Port C0h
Canal 5	Port C4h
Canal 6	Port C8h
Canal 7	Port CCh

Voici maintenant une petite unité servant à diriger le contrôleur DMA. Elle met à votre disposition toutes les procédures nécessaires. Vous pouvez programmer toutes les étapes isolément ou alors réaliser une initialisation complète par l'intermédiaire de la procédure DMA_Init_Transfer. Vous devez transmettre comme premier paramètre à la procédure le numéro du canal à utiliser, puis comme second paramètre le mode souhaité. Le troisième paramètre sera un pointeur sur le bloc de données situé dans la mémoire principale. Enfin le quatrième paramètre sera la taille de ce bloc de données.

```
unit DMA;
interface

TYPE DMAarray = array[0..7] of byte;

CONST
```

```
{ Adresses des contrôleurs DMA }
DMA_Adress           : DMAarray = ($00,$02,$04,$06,$C0,$C4,$C8,$CC);
DMA_Count            : DMAarray = ($01,$03,$05,$07,$C2,$C6,$CA,$CE);
DMARead_status_Reg   : DMAarray = ($08,$08,$08,$08,$D0,$D0,$D0,$D0);
DMAWrite_status_Reg  : DMAarray = ($08,$08,$08,$08,$D0,$D0,$D0,$D0);
DMAWrite_requ_Reg    : DMAarray = ($09,$09,$09,$09,$D2,$D2,$D2,$D2);
DMAWr_single_mask_Reg: DMAarray = ($0A,$0A,$0A,$0A,$D4,$D4,$D4,$D4);
DMAWr_mode_Reg       : DMAarray = ($0B,$0B,$0B,$0B,$D6,$D6,$D6,$D6);
DMAClear_Flipflop    : DMAarray = ($0C,$0C,$0C,$0C,$D8,$D8,$D8,$D8);
DMARead_Temp_Reg     : DMAarray = ($0D,$0D,$0D,$0D,$DA,$DA,$DA,$DA);
DMAMaster_Clear      : DMAarray = ($0D,$0D,$0D,$0D,$DA,$DA,$DA,$DA);
DMA_Clear_Mask_Reg   : DMAarray = ($0E,$0E,$0E,$0E,$DC,$DC,$DC,$DC);
DMA_Wr_All_Mask_Reg  : DMAarray = ($0F,$0F,$0F,$0F,$DE,$DE,$DE,$DE);
DMA_Lower_Page       : DMAarray = ($87,$83,$81,$82,$00,$8B,$89,$8A);
DMA_Higher_Page      : Array[0..7] of word
                               = ($487,$483,$481,$482,$0,$48B,$489,
                                  $48A);
{ Modes des registres DMA DMA_Wr_mode_Reg }
Mode_Demande                   = $00;
Mode_Unique                    = $40;
Mode_Bloc                      = $80;
Mode_Cascade                   = $C0;
Adresses_Decrement             = $20;
Adresses_Increment             = $00;
Autoinit_Enable                = $10;
Autoinit_Disable               = $00;
Verif_transfer                 = $00;
Ecrit_Transfer                 = $04;
Lect_Transfer                  = $08;
Set_Request_Bit                = $04;
Clear_Request_Bit              = $00;
Set_Mask_Bit                   = $04;
Clear_Mask_Bit                 = $00;

procedure DMA_mode_def(canal,mode : byte);

procedure DMA_Normmode_def(canal,mode : byte);
procedure DMA_Clear_Flipflop(canal : byte);
```

```
procedure DMA_Startadresse(canal : byte; Start : pointer);
procedure DMA_Taille_bloc(canal : byte; size : word);
procedure DMA_canal_Masquer(canal : byte);
procedure DMA_canal_Demasquer(canal : byte);
procedure DMA_Init_Transfer(canal,mode : byte; p : pointer; s :word);

implementation

TYPE
  pt = record                    { Permet de traiter }
    ofs,sgm  :  word;            { simplement les pointeurs }
  end;

procedure DMA_mode_def(canal,mode : byte);
begin;
  port[DMAWr_Mode_Reg[canal]] := mode;
end;

procedure DMA_Normmode_def(canal,mode : byte);
begin;
  port[DMAWr_Mode_Reg[canal]] := mode+Adresses_Increment+Lect_
                                 Transfer+
                                Autoinit_Disable+canal;
end;

procedure DMA_Clear_Flipflop(canal : byte);
begin;
  port[DMAClear_Flipflop[canal]] := 0;
end;

procedure DMA_Startadresse(canal : byte; Start : pointer);
var l : longint;
    pn,offs : word;
begin;
  l := 16*longint(pt(Start).sgm)+pt(Start).ofs;
  pn := pt(l).sgm;
  offs := pt(l).ofs;
  port[DMA_Adress[canal]] := lo(offs);
```

```
  port[DMA_Adress[canal]] := hi(offs);
  port[DMA_Lower_Page[canal]] := lo(pn);
  port[DMA_Higher_Page[canal]] := hi(pn);
end;

procedure DMA_Taille_bloc(canal : byte; size : word);
begin;
  DMA_Clear_Flipflop(canal);
  port[DMA_Count[canal]]:= lo(size);
  port[DMA_Count[canal]] := hi(size);
end;

procedure DMA_canal_Masquer(canal : byte);
begin;
 port[DMAWr_single_mask_Reg[canal]] := canal + Set_Mask_Bit;
end;

procedure DMA_canal_Demasquer(canal : byte);
begin;
 port[DMAWr_single_mask_Reg[canal]] := canal + Clear_Mask_Bit;
end;

procedure DMA_Init_Transfer(canal,mode : byte; p : pointer; s :
                            word);
begin;
  DMA_canal_Masquer(canal);
  DMA_Startadresse(canal,p);
  DMA_Taille_bloc(canal,s);
  DMA_Normmode_Def(canal,mode+canal);
  DMA_canal_Demasquer(canal);
end;
begin;
end.
```

12.5. L'horloge temps réel (Real-Time-Clock, RTC)

L'horloge RTC constitue une puce importante. Cette puce est une Dallas 1287 ou un élément analogue. Elle fonctionne avec une pile, en mode Lower-Power, qui la protège des pertes de données lorsque vous allumez et éteignez l'appareil. Dans la mémoire RTC de 64 octets, désignée souvent sous le nom de CMOS-RAM, vous avez la configuration du système emmagasinée. De plus, la puce vous donne l'heure, un calendrier, ainsi que la possibilité d'une interruption périodique programmable.

La mise à jour de l'heure et du calendrier se fait de manière cyclique. Tant que la puce RTC fonctionne normalement, le compteur des secondes est incrémenté chaque seconde. Lorsqu'il se produit un dépassement, les autres registres concernés sont incrémentés à leur tour. Pendant ce dépassement, les octets 0 à 9 de la RAM RTC ne sont pas accessibles à l'unité centrale. La vitesse du cycle de mise à jour est spécifiée par les bits de partage 2-0 du registre d'état A, ainsi que le bit 7 (SET) du registre d'état B.

L'accès à la RAM RTC est aussi simple que possible. Il faut d'abord sortir au port 70h le numéro d'index du registre souhaité. On peut alors lire ou écrire des données sur le port 71h. Tous les registres peuvent être ici lus ou écrits, sauf les suivants, qui ne sont accessibles qu'à la lecture :

- registres d'état C et D
- Bit 7 du registre d'état 1
- Bit 7 de l'octet des secondes (Index 00h)

Les 14 premiers octets de la RAM RTC sont employés pour l'horloge et les quatre registres d'état. Les 50 octets restants servent à la configuration du système. La définition exacte des différents octets est donnée par le tableau suivant :

Secondes	00h
Secondes Alarme	01h
Minutes	02h
Minutes Alarme	03h
Heures	04h
Heures Alarme	05h
Jour de la semaine	06h

Jour du mois	07h
Mois	08h
Année	09h
Etat A	0Ah
Etat B	0Bh
Etat C	0Ch
Etat D	0Dh
Etat Diagnostic	0Eh
Etat Shutdown	0Fh
Type Floppy	10h
Réservé	11h
Type HD	12h
Réservé	13h
Equipement	14h
Low Base Memory	15h
High Base Memory	16h
Low Extended Memory	17h
High Extended Memory	18h
HD 1 extended Type Byte	19h
HD 2 extended Type Byte	1Ah
Réservé	1Bh
Réservé	1Ch
Réservé	1Dh
Réservé	1Eh
Features installés	1Fh
HD 1 Low Cylinder, nombre	20h
HD 1 High Cylinder, nombre	21h
HD 1 têtes	22h
HD 1 Low Pre-Compensation Start	23h
HD 1 High Pre-Compensation Start	24h
HD 1 Low Landing zone	25h

HD 1 High Landing zone	26h
HD 1 secteurs	27h
Options 1	28h
Réservé	29h
Réservé	2Ah
Options 2	2Bh
Options 3	2Ch
Réservé	2Dh
Low CMOS Ram Checksum	2Eh
High CMOS Ram Checksum	2Fh
Low Extended Memory Byte	30h
High Extended Memory Byte	31h
Byte siècle	32h
Setup Information	33h
Vitesse CPU	34h
HD 2 Low Cylinder, nombre	35h
HD 2 High Cylinder, nombre	36h
HD 2 Têtes	37h
HD 2 Low Pre-Compensation Start	38h
HD 2 High Pre-Compensation Start	39h
HD 2 Low Landing zone	3Ah
HD 2 High Landing zone	3Bh
HD 2 secteurs	3Ch
Réservé	3Dh
Réservé	3Eh
Réservé	3Fh

Les fonctions de l'horloge

L'unité centrale reçoit l'heure et la date en lisant les octets correspondants de la mémoire RTC. Lorsque ceux-ci sont incrémentés, la lecture n'est pas possible. Quand on écrit une valeur dans les octets, on les initialise. Si vous voulez écrire des données dans ces octets, vous devez d'abord mettre hors service les RTC-Updates par l'intermédiaire du bit SET dans le registre d'état

B. Les données de l'horloge sont conservées en format BCD. Cela veut dire que les quatre bits supérieurs contiennent les dizaines, tandis que les bits inférieurs contiennent les unités du nombre. Les différents octets contiennent les valeurs suivantes :

Fonction	Adresse	Données BCD
Secondes	00	00 à 59
Alarme Secondes	01	00 à 59
Minutes	02	00 à 59
Alarme minutes	03	00 à 59
Heures	04	(Mode 12 heures) 01 à 12 (AM) 81 à 92 (PM) (mode 24 heures) 00 à 23
Alarme heures	05	(mode 12 heures) 01 à 12 (AM) 81 à 92 (PM) (mode 24 heures) 00 à 23
Jour de la semaine	06	01 à 07
Jour du mois	07	01 à 31
Mois	08	01 à 12
Année	09	00 à 99

Les registres d'état

La mémoire RTC dispose de quatre registres d'état. Ils servent à contrôler la puce et à indiquer son état.

Registre A

Le registre A est intéressant surtout parce qu'il offre la possibilité de déterminer la vitesse d'appel de l'interruption périodique. Dans le bit 7 se trouve le bit UIP (Update in progress). S'il a la valeur 1, cela veut dire qu'une mise à jour est en cours. Les bits 6 à 4 contiennent la base de l'heure. La valeur standard est ici 101b, ce qui correspond à une valeur de 32768d. Faites particulièrement attention aux bits RS 3 à 0. Ils sont responsables de la sélection des vitesses. Celles-ci se calculent à l'aide de la formule

```
Vitesse=65536/2^RS
```

La valeur par défaut est de 1024 Hz. Elle s'obtient par l'opération 65536/2^6 (1001b).

Registre B

Le registre B est un registre possédant beaucoup de fonctions, chacun des bits ayant la sienne. Commencez avec le bit SET, qui est le bit 7. S'il est sur 1, cela veut dire que le cycle actuel de mise à jour est interrompu. Ce sera nécessaire en particulier pour l'initialisation. Quand la valeur est 0, une mise à jour normale s'effectue toutes les secondes. Le bit 6 (PIE, soit Periodic Interrupt Enable) indique si l'interruption périodique a été appelée avec la fréquence spécifiée dans le registre A. Si le bit est sur 1, l'appel est en cours. Sinon, aucune interruption n'a été déclenchée. Dans le bit 5 (AIE, Alarm interrupt Enable) se trouve l'information indiquant si l'interruption de l'alarme a été déclenchée au moment voulu. Si le bit a la valeur 1, l'interruption est activée. L'interruption de mise est déclenchée si le bit 4 (UIE, Update Ended Interrupt Enable) contient la valeur 1. Avec le bit 3 (SQWE, Square Wave Enable), vous pouvez indiquer si la fréquence des ondes rectangulaires spécifiée dans le registre A doit être activée (bit = 1) ou non. Le bit 2 (DM, Date Mode) est important si vous accédez à la date. Quand le bit contient la valeur 1, les nombres sont en format binaire. Par défaut, on trouve toutefois ici la valeur 0, qui indique le format BCD. Par l'intermédiaire du bit 1 (24/12 hour), on choisit entre l'affichage de l'heure sur 24 heures ou sur 12 heures. Par défaut, c'est la valeur 1 que l'on trouve ici, c'est-à-dire la valeur qui correspond au mode 24 heures. Dans le bit 0 (DSE, Daylight Savings Enabled), on trouve l'information indiquant si l'heure doit se conformer au changement en heure d'été ou non. De manière standard, c'est exclu par la valeur 0.

Registre C

Le registre 0Ch est un registre flag. Le bit 7 est celui du flag Interrupt-Request. Il contient la valeur 1 quand l'une des situations déclenchant une interruption est en place et donc quand le flag d'interruption correspondant est sur "enable". Le bit 6 est un flag indiquant une interruption périodique. Le bit 5 est posé sur 1 quand l'heure sur laquelle est réglée l'alarme et l'heure actuelle viennent à coïncider. Le flag est également posé quand l'interruption d'alarme n'est pas activée. Le bit 4, enfin, est appelé le flag "Update Ended Interrupt". Il indique (comme son nom le laisse présager) qu'un cycle de mise à jour de la mémoire RTC a pris fin. Les bits 3 à 0 sont réservés. Notez que le contenu de ce registre s'efface après chaque accès en lecture.

Registre D

Le registre D sert à surveiller la pile. Quand le bit 7 est posé, la pile est en état de fonctionnement sans problème. Sinon, ce bit contient la valeur 0, ce qui indique que la pile est défectueuse. Les bits 6 à 0 n'ont aucune fonction.

Les octets de configuration

Dans les octets à partir de 0Eh, on trouve la configuration du système. Nous allons nous occuper ici brièvement des différents octets. Vous trouverez d'autres informations plus approfondies dans la littérature sur le sujet, par exemple dans la Bible du PC de Micro Application.

L'octet pour le diagnostic 0Eh

Cet octet sert à vérifier au démarrage si la configuration du système est correcte. La signification des différents bits est la suivante :

Bit 7	Etat de la pile RTC
1	Pas de courant
0	Le courant passe
Bit 6	Sommes de vérification
1	La somme de vérification n'est pas valide
0	La somme de vérification est valide
Bit 5	Information correcte sur la configuration ?
1	La configuration n'est pas valide
0	La configuration est valide
Bit 4	Vérification de la taille de la mémoire
1	La taille de la mémoire s'est modifiée
0	La taille de la mémoire ne s'est pas modifiée
Bit 3	Vérification du disque dur
1	Erreur du disque dur ou du contrôleur
0	Pas d'erreur
Bit 2	Etat de l'heure
1	Heure incorrecte
0	Heure correcte
Bits 1-0	Réservé

L'octet d'état Shutdown - 0Fh

Cet octet est posé lors d'un reset de l'unité centrale. Le code de reset sert d'indicateur pour le type du reset. Les valeurs possibles sont les suivantes :

00h	Reset système normal
09h	Reset logiciel (retour à partir du mode protégé)

Type des lecteurs de disquettes - 10h

Cet octet permet d'obtenir des informations sur les lecteurs de disquettes connectés.

Bits 7-4	Type du premier lecteur de disquettes
0000	Pas de lecteur de disquettes installé
0001	Lecteur 360 Ko, 5"25
0010	Lecteur 1,2 Mo, 5"25
0011	Lecteur 720 Ko, 3"5
0100	Lecteur 1,44 Mo, 3"5
0101	Lecteur 2,88 Mo, 3"5
Bits 3-0	Type du second lecteur de disquettes
0000	Pas de lecteur de disquettes installé
0001	Lecteur 360 Ko, 5"25
0010	Lecteur 1,2 Mo, 5"25
0011	Lecteur 720 Ko, 3"5
0100	Lecteur 1,44 Mo, 3"5
0101	Lecteur 2,88 Mo, 3"5

Type des disques durs installés - 12h

Vous pouvez déterminer le type des disques durs installés par l'intermédiaire de cet octet. Il faut également aller quérir les informations fournies par les octets 19h et 1Ah. Cet octet a la signification suivante :

Bits 7-4	Type du premier disque dur
0000	Pas de disque dur installé
0001-1110	Disque dur de type 1 à 14
1111	Information sur le type dans l'octet 19h

Bits 3-0	Type du second disque dur
0000	Pas de disque dur installé
0001-1110	Disque dur de type 1 à 14
1111	Information sur le type dans l'octet 1Ah

Définition de l'équipement - 14h

Cet octet est utilisé pour l'auto-test du hardware. Voici sa signification :

Bits 7-6	Nombre de lecteurs de disquettes installés
00	Un lecteur de disquettes
01	Deux lecteurs de disquettes
10	Réservé
11	Réservé
Bits 5-4	Type de la carte graphique
00	Extended Functionality Controller
01	Affichage des couleurs sur 40 colonnes
10	Affichage des couleurs sur 80 colonnes
11	Affichage monochrome
Bits 3-2	Non utilisés
Bit 1	Coprocesseur
1	Coprocesseur installé
0	Pas de coprocesseur installé
Bit 0	Lecteur de disquettes installé ?
1	Le lecteur est installé
0	Pas de lecteur

Low/High Base Memory - 15h/16h

Les deux octets Low-Base-Memory et High-Base-Memory permettent de déterminer la taille de la mémoire principale installée. Une valeur de 0200h représente ainsi 512 Ko de RAM. 0280h représente 640 Ko.

Low/High Extended Memory - 17h/18h

Les octets Low-Extended-Memory et High-Extended-Memory permettent de déterminer la taille de la mémoire étendue installée (au-dessus de 1 Mo). La valeur 0400h représente ainsi 1024 Ko de mémoire étendue, tandis que 0C00h représente 3072 Ko.

Octet Extended-Type pour le disque dur 1 - 19h

Dans cet octet, vous trouverez le type du premier disque dur, lorsque les bits 7-4 de l'octet 12h contiennent la valeur 0Fh.

Octet Extended-Type pour le disque dur 2 - 1Ah

Dans cet octet, vous trouverez le type du second disque dur, lorsque les bits 3-0 de l'octet 12h contiennent la valeur 0Fh.

Périphériques - 1Fh

Cet octet sert à annoncer les messages d'erreur.

Bits 7-3	Réservés
Bit 2	Annonce une erreur du lecteur de disquettes
Bit 1	Annonce une erreur de l'affichage vidéo
Bit 0	Annonce une erreur du clavier

Vitesse de l'unité centrale - 34h

Par l'intermédiaire de cet octet, vous pouvez déterminer la vitesse de l'unité centrale. Les bits 7 à 1 sont réservés. Lorsque le bit 0 a la valeur 0h, l'unité centrale fonctionne en mode lent. Au contraire, lorsque le bit est posé, l'unité centrale fonctionne en mode turbo.

La mémoire RTC dans la pratique

Le programme que nous allons maintenant présenter montre comment on accède à la mémoire RTC et comment on prend en compte ses registres. Il vous servira de base pour créer une unité de programmation destinée à contrôler cette mémoire.

```
program rtc_unit;
uses crt,dos;
const
  Rtc_Secondes          = $00;
```

```
Rtc_Secondes_alarm  = $01;
Rtc_Minutes         = $02;
Rtc_Minutes_alarm   = $03;
Rtc_Heures          = $04;
Rtc_Heures_alarm    = $05;
Rtc_Jour_semaine    = $06;
Rtc_Jour_mois       = $07;
Rtc_Mois            = $08;
Rtc_Annee           = $09;
Rtc_Etat_A          = $0A;
Rtc_Etat_B          = $0B;
Rtc_Etat_C          = $0C;
Rtc_Etat_D          = $0D;
Rtc_Diagnost_Etat   = $0E;
Rtc_Shutdown_Etat   = $0F;
Rtc_Floppy_Type     = $10;
Rtc_HD_Type         = $12;
Rtc_Equipement     = $14;
Rtc_Lo_Basememory   = $15;
Rtc_Hi_Basememory   = $16;
Rtc_Lo_Extendedmem  = $17;
Rtc_Hi_Extendedmem  = $18;
Rtc_HD1_extended    = $19;
Rtc_HD2_extended    = $1A;
Rtc_Features        = $1F;
Rtc_HD1_Lo_Cylinder = $20;
Rtc_HD1_Hi_Cylinder = $21;
Rtc_HD1_Tetes       = $22;
Rtc_HD1_Lo_Precom   = $23;
Rtc_HD1_Hi_Precom   = $24;
Rtc_HD1_Lo_Landing  = $25;
Rtc_HD1_Hi_Landing  = $26;
Rtc_HD1_Secteurs    = $27;
Rtc_Options1        = $28;
Rtc_Options2        = $2B;
Rtc_Options3        = $2C;
Rtc_Lo_Checksum     = $2E;
Rtc_Hi_Checksum     = $2F;
```

```
  Rtc_Extendedmem_Lo  = $30;
  Rtc_Extendedmem_Hi  = $31;
  Rtc_Siecle          = $32;
  Rtc_Setup_Info      = $33;
  Rtc_CPU_speed       = $34;
  Rtc_HD2_Lo_Cylinder = $35;
  Rtc_HD2_Hi_Cylinder = $36;
  Rtc_HD2_Tetes      = $37;
  Rtc_HD2_Lo_Precom   = $38;
  Rtc_HD2_Hi_Precom   = $39;
  Rtc_HD2_Lo_Landing  = $3A;
  Rtc_HD2_Hi_Landing  = $3B;
  Rtc_HD2_Secteurs    = $3C;

function wrhexb(b : byte) : string;
const hexcar : array[0..15] of char =
 ('0','1','2','3','4','5','6','7','8','9','A','B','C','D','E','F');
begin;
  wrhexb := hexcar[(b shr 4)] + hexcar[(b AND $0F)];
end;

function wrhexw(w : word) : string;
begin;
  wrhexw := '$'+wrhexb(hi(w))+wrhexb(lo(w));
end;

procedure write_rtc(Reg,val : byte);
{
 Ecrit une valeur dans le registre RTC indiqué dans Reg
}
begin;
  port[$70] := Reg;
  port[$71] := val;
end;

function read_rtc(Reg : byte) : byte;
{Lit une valeur dans le registre RTC indiqué dans Reg}
begin;
```

```
  port[$70] := Reg;
  read_rtc := port[$71];
end;
Procedure Write_Floppy;
{
Affiche des informations sur les lecteurs de disquettes installés
}
Var Fl : byte;
    Fls : array[1..2] of byte;
begin;
  Fl     := Read_Rtc(Rtc_Floppy_Type);
  Fls[2] := Fl AND $OF;
  Fls[1] := Fl SHR 4;
  for Fl := 1 to 2 do begin;
    write('Floppy ',Fl,': ');
    case Fls[Fl] of
      0 : begin;
            writeln('No  Floppy        ');
          end;
      1 : begin;
            writeln('5" Floppy, 360 KO');
          end;
      2 : begin;
            writeln('5" Floppy, 1.2 MO');
          end;
      3 : begin;
            writeln('3" Floppy, 720 KO');
          end;
      4 : begin;
            writeln('3" Floppy, 1.44 MO');
          end;
    end;
  end;
end;

Procedure Write_Hd;
{
 Affiche le type du disque dur installé
```

```
}
Var Hd : byte;
    Hds : array[1..2] of byte;
begin;
  Hd     := Read_Rtc(Rtc_HD_Type);
  Hds[2] := Hd AND $OF;
  Hds[1] := Hd SHR 4;
  If HDs[1] = $F then HDs[1] := Read_Rtc(Rtc_HD1_extended);
  If HDs[2] = $F then HDs[2] := Read_Rtc(Rtc_HD2_extended);
  writeln('HD 1 : Type ',Hds[1]);
  writeln('HD 2 : Type ',Hds[2]);
end;

procedure Write_Memory;
{
 Affiche la mémoire disponible
}
var base,extended : word;
begin;
  Base     := 256 * Read_Rtc(Rtc_Hi_Basememory) +
              Read_Rtc(Rtc_Lo_Basememory);
  extended := 256 * Read_Rtc(Rtc_Hi_Extendedmem) +
              Read_Rtc(Rtc_Lo_Extendedmem);
  writeln('Base memory: ',Base,' KO');
  writeln('Extended memory: ',extended,' KO');
end;

procedure Write_Display;
{
 Affiche le type de la carte graphique et indique si un
 coprocesseur est installé
}
var dType : byte;
    Copro : byte;
begin;
  dType  := Read_Rtc(Rtc_Equipement);
  Copro := (dType AND 3) SHR 1;
  dType  := (dType AND 63) SHR 4;
```

```
  case dType of
    0 : begin;
          writeln('Extended functionality GFX-Controller');
        end;
    1 : begin;
          writeln('Color Display sur 40 colonnes');
        end;
    2 : begin;
          writeln('Color Display sur 80 colonnes');
        end;
    3 : begin;
          writeln('Monochrome Display Controller');
        end;
  end;
  if Copro = 1 then
    writeln('Coprocesseur trouvé')
  else
    writeln('Pas de coprocesseur trouvé');
end;

procedure write_shadow;
{
 Indique les secteurs supportés par la shadow-Ram
}
var shadow : byte;
begin;
  shadow := read_rtc(Rtc_Options1);
  shadow := shadow AND 3;
  case shadow of
    0 : begin;
          writeln('Shadow System AND Video Bios');
        end;
    1 : begin;
          writeln('Shadow System Bios');
        end;
    2 : begin;
          writeln('Shadow disabled');
        end;
```

```
  end;
end;

procedure write_cpuspeed;
{
 Indique si l'unité centrale se trouve en mode Turbo
}
var speed : byte;
begin;
  speed := read_rtc(Rtc_CPU_speed);
  if speed = 1 then
    writeln('CPU in Turbo-Mode')
  else
    writeln('CPU in Deturbo-Mode');
end;
begin;
 clrscr;
 Write_Floppy;
 Write_Hd;
 Write_Memory;
 Write_Display;
 Write_Shadow;
 Write_CPUSpeed;
 readln;
end.
```

13. Vivre dans un monde sonore - La carte Sound Blaster

La carte Sound Blaster constitue sans doute aujourd'hui le standard en matière de son. Cette carte, sortie en 1989 par Creative Labs, a subi beaucoup de modifications depuis cette date. Après SB Pro et SB 16, elle en est à la version AWE 32. Toutes ces cartes sont plus ou moins compatibles en amont. C'est également vrai pour la puce DSP (Digital Signal Processor), mais avec certaines restrictions.

Malheureusement, la compatibilité cesse avec la puce OPL et la puce de mixage. C'est pourquoi vous devrez intégrer un traitement à part des cartes son dans vos programmes. Ce chapitre va vous apporter toutes les connaissances indispensables pour que vous puissiez travailler avec Sound Blaster. Vous y trouverez des références détaillées pour la partie matérielle, mais aussi beaucoup d'éléments pratiques.

13.1. Composants des cartes Sound Blaster

Occupons-nous d'abord de la partie matérielle de la carte Sound Blaster. Sans ces connaissances, une programmation correcte de la carte serait impossible. Le cœur de la carte Sound Blaster est constitué par le DSP, Digital Signal Processor. Cette puce possède les fonctionnalités d'un convertisseur AD/DA et est prévue pour l'entrée et la sortie de données numériques. La seconde puce intégrée à la carte est une puce OPL. Elle est responsable de la synthèse FM. Puisque cette technique a pris entre-temps un coup de vieux, nous ne la présenterons pas ici en détail. A partir de SB Pro, on trouve également une

puce de mixage sur la carte. Celle-ci possède des fonctionnalités utiles, comme le réglage du volume ou de la balance.

Programmer le processeur de signaux - DSP

La puce DSP intégrée dans la série Sound Blaster porte la désignation CT-DSP-1321. Elle se trouve au port d'adresse 2x0h, où le x représente l'adresse de base spécifiée. On la programme en écrivant une valeur dans l'un de ses registres. L'adresse de base peut être choisie par étapes de 10h et peut se situer entre 210h et 280h. Voici quels sont les registres disponibles :

Port	Nom	Etat
B+00h	Registre FM gauche, sélection et état	R/W
B+01h	Registre FM gauche, données	W
B+02h	Registre FM droite, sélection et état	R/W
B+03h	Registre FM droite, données	W
B+04h	Puce Mixer, registre d'index	W
B+05h	Puce Mixer, registre de données	L/W
B+06h	Reset DSP	W
B+08h	Les deux registres FM, sélection et état	R/W
B+09h	Les deux registres FM, données	W
B+0Ah	Données DSP (Read Data)	R
B+0Ch	Données DSP ou commande et état	R/W
B+0Dh	DSP Timer_Interrupt Clear Port	R
B+0Eh	DSP état des données existantes	R
B+0Fh	DSP 16-bit Voice-Int. Clear Port	R
B+10h	Données CD-ROM	R
B+11h	CD-ROM Command	W
B+12h	CD-ROM Reset	W
B+13h	Mise en service du CD-ROM	W
338h	FM Register Status Port	R
339h	Registre de sélection FM	W
38Bh	Sélection Advanced FM-Register	W

La forme générale d'un accès aux registres est la suivante : on écrit dans SB d'abord le numéro de la commande, puis les paramètres nécessaires par l'intermédiaire du port de commande 2xCh. L'accès en lecture se fait par le port de données 2xAh.

Avant d'envoyer des commandes au Sound Blaster, il faut effectuer un reset. Vous le ferez par l'intermédiaire du port de reset 2x6h. Il faut écrire pour cela d'abord la valeur 1 sur le port de reset, attendre un bref instant, puis écrire la valeur 0 sur le même port de reset. Vous pouvez voir immédiatement si le reset s'est fait correctement. C'est le cas si le port de données affiche la valeur $AA après environ 100 ms. Si vous rencontrez bien cette valeur, le reset a fonctionné correctement. Sinon, il n'a pas réussi.

Voici sur un exemple comment pourrait se présenter une fonction de reset pour SB. La fonction s'appelle Reset_SBCard.

```
FUNCTION Reset_SBCard : BOOLEAN;
{
 La fonction effectue un reset du DSP. Si le reset réussit,
 elle retourne TRUE, sinon FALSE
}
CONST  ready = $AA;
VAR ct, stat : BYTE;
BEGIN
  PORT[dsp_adr+$6] := 1;          { dsp_adr+$6 = fonction de reset }
  FOR ct := 1 TO 100 DO;
  PORT[dsp_adr+$6] := 0;
  stat := 0;
  ct   := 0;                      { Comparaison  ct < 100, car }
  WHILE (stat <> ready)           { l'initialisation dure    }
  AND   (ct < 100)      DO BEGIN  { environ 100ms             }
    stat := PORT[dsp_adr+$E];
    stat := PORT[dsp_adr+$a];
    INC(ct);
  END;
  Reset_SBCard := (stat = ready);
END;
```

On peut se servir du fait que la valeur 0AAh se trouve dans le registre de données après un reset réussi, pour réaliser une reconnaissance du port de base. On placera pour cela dans une boucle tous les ports possibles situés entre 210h et 280h par pas de 10h, comme des candidats potentiels au titre

de port de base SB. Lorsque le reset réussit, nous avons trouvé ainsi une carte Sound Blaster. La fonction Detect_SBReg montre comment se présente une routine de détection. La fonction utilise le principe que nous venons d'exposer et renvoie la valeur TRUE quand une carte Sound Blaster a été trouvée.

```
FUNCTION Detect_SBReg : BOOLEAN;
{
 La fonction retourne TRUE, quand une carte Soundblaster a pu être
 initialisée, sinon FALSE. La variable dsp_adr reçoit l'adresse de
 base de la SB.
}
VAR
  Port, Lst : WORD;
BEGIN
 Detect_SBReg := SbRegDetected;
 IF SbRegDetected THEN EXIT;       { Exit, lorsque l'initialisation
                                     est faite  }
 Port := Startport;                { Adresse SB possibles }
 Lst  := Endport;                  { entre $210 et $280 !      }
 WHILE (NOT SbRegDetected)
 AND   (Port <= Lst)  DO BEGIN
   dsp_adr := Port;
   SbRegDetected := Reset_SBCard;
   IF NOT SbRegDetected THEN
     INC(Port, $10);
 END;
 Detect_SBReg := SbRegDetected;
END;
```

Vous venez donc de trouver votre carte Sound Blaster et vous savez à quel port vous pouvez l'adresser. Vous vous trouvez alors confronté au problème suivant : à quelle version de la carte SB a-t-on affaire ? On sait qu'il en existe de nombreuses versions et que le contrôle de ces différentes cartes possibles n'est pas toujours le même.

Pour répondre à cette question, nous allons nous servir d'une fonction offerte par la carte Sound Blaster elle-même : l'obtention de son numéro de version. Nous devons à cette fin envoyer la commande à la carte et attendre en retour le numéro promis dans le registre de données. Mais comment écrit-on une commande adressée à la carte SB ?

Il suffit en fait de savoir que les commandes sont toujours envoyées par l'intermédiaire du port de commande 2xCh. Il faut toutefois noter que l'on ne peut pas écrire directement dans ce port tant que le bit 7 du port est posé. Il faut donc attendre dans une boucle jusqu'à ce que le port soit en mesure de recevoir votre valeur. Voici comment la procédure en question pourrait se présenter :

```
procedure Wr_dsp(v : byte);
{
 Attend jusqu'à ce que la DSP soit prête à l'écriture, et écrit
 alors dans la DSP l'octet transmis dans "v"
}
begin;
  while port[dsp_adr+$c] >= 128 do ;
  port[dsp_adr+$c] := v;
end;
```

Puisque nous n'allons pas seulement envoyer des valeurs à SB, mais aussi lire des valeurs, nous avons besoin d'une fonction capable de lire sur la carte SB. Pour lire une valeur à partir du port de données, il faut attendre que celui-ci ne contienne plus la valeur 0AAh. Voyons comment ce processus très simple se présentera dans la pratique : la fonction SBReadSB fonctionne selon le procédé que nous venons de décrire et elle fournit l'octet actuel à partir du port de données.

```
FUNCTION SbReadByte : BYTE;
{
 La fonction attend, jusqu'à ce que la DSP puisse être lue et
 retourne alors la valeur lue.
}
begin;
  while port[dsp_adr+$a] = $AA do ;  { attend que DSP soit prête }
  SbReadByte := port[dsp_adr+$a];    { lit une valeur }
end;
```

Revenons maintenant à notre problème : nous voulions connaître le numéro de version de SB. Nous utiliserons pour cela la commande $E1, qui sert à reconnaître la version. Après envoi de la commande, on pourra lire le numéro de version dans le port. Ce numéro comprend deux octets. En premier lieu, on obtient en retour le numéro principal de la version. En lisant à nouveau dans le port, on obtient cette fois le numéro de sous-version. SBGetDSPVersion fournit un moyen d'obtenir ainsi le numéro de version.

```
PROCEDURE SbGetDSPVersion;
{
 Fournit le numéro de version de la DSP et le dépose dans les
 variablesglobales SBVERSMAJ et SBVERSMIN, ainsi que SBVERSSTR.
}
VAR i : WORD;
    t : WORD;
    s : STRING[2];
BEGIN
  Wr_dsp($E1);                    {$E1 = interroge le numéro de version}
  SbVersMaj := SbReadByte;
  SbVersMin := SbReadByte;
  str(SbVersMaj, SbVersStr);
  SbVersStr := SbVersStr + '.';
  str(SbVersMin, s);
  if SbVersMin > 9 then
    SbVersStr := SbVersStr +       s
  else
    SbVersStr := SbVersStr + '0' + s;
END;
```

Outre la commande servant à reconnaître la version, la carte SB possède aussi une série d'autres commandes. Voici d'abord un tableau des commandes DSP sur la carte Sound Blaster :

Commande	Fonction
10h	Sortie directe 8 bits
14h	Sortie 8 bits par DMA
16h	Sortie des samples comp. 2 bits via DMA
17h	Sortie des samples comp. 2 bits avec octet de référence via DMA
20h	Enregistrement direct
24h	Enregistrement de samples 8 bits via DMA
30h	Entrée MIDI directe
31h	Entrée MIDI via interruption
32h	Entrée MIDI directe avec timestamp
33h	Entrée MIDI via interruption avec timestamp

Commande	Fonction
34h	Mode MIDI-UART, direct
35h	Mode MIDI-UART via interruption
37h	MIDI-UART via interruption avec timestamp
38h	Envoyer un code MIDI
40h	Définir une vitesse de sample
45h	DMA-Multiblock Continue SB16
48h	Spécifier la taille du bloc
74h	Sortie de samples comp. 4 bits via DMA
75h	Sortie de samples comp. 4 bits avec référence via DMA
76h	Sortie de samples comp. 2, 6 bits via DMA
77h	Sortie de samples comp. 2, 6 bits avec octet de référence via DMA
80h	Définir un bloc de silence
91h	Sortie 8 bits via DMA en High-Speed
99h	Entrée 8 bits via DMA en High-Speed
B6h	Sortir des données 16 bits sur SB16 via DMA
B9h	Enregistrer des données 16 bits sur SB16 via DMA
C6h	Sortir des données 8 bits sur SB16 via DMA
C9h	Enregistrer des données 8 bits sur SB16 via DMA
D0h	Arrêter DMA
D1h	Mettre le haut-parleur en service
D3h	Mettre le haut-parleur hors service
D4h	Continuer DMA
D8h	Interroger le réglage du haut-parleur
E1h	Interroger la version

Voyons maintenant ces commandes en détail.

10h - Sortie directe 8 bits

Pour sortir des données périodiquement, par exemple par l'intermédiaire de l'interruption Timer, vous utiliserez la fonction 10h. Les deux étapes doivent être répétées à vitesse constante !

- Envoyez la commande 10h
- Envoyez l'octet de données.

14h - Sortie 8 bits via DMA

La sortie via DMA est assez complexe, surtout pour quelqu'un qui ne s'est jamais penché auparavant sur la question. Vous devez être en mesure de programmer correctement le contrôleur DMA. Sinon, vous obtiendrez certes beaucoup d'effets intéressants, mais pas la sortie des données sample. Vous trouverez d'autres informations sur le contrôleur DMA et sa programmation au paragraphe 12.4.

Si vous voulez sortir des données via DMA, vous devez d'abord détourner l'interruption SB sur une routine qui vous est propre, puis programmer correctement le contrôleur DMA. Vous devez ensuite déterminer et indiquer la vitesse de sampling. Après quoi vous devrez effectuer les deux opérations suivantes :

- Envoyez la commande 14h
- Envoyez l'octet Low de la longueur sample -1.
- Envoyez l'octet high de la longueur sample -1.

La sortie démarre dès que vous en avez fini avec la troisième étape. Notez que vous ne pouvez pas sortir de bloc supérieur à 64 Ko. Si vous avez affaire à un bloc de cette taille, il vous faut le diviser en plusieurs sous-blocs.

Dans la pratique, vous devrez procéder comme suit : en premier lieu, vous calculerez l'adresse du bloc à sortir. Vous l'obtiendrez à l'aide de la formule:

```
Physpos:=16*sgm(Block^)+ofs(Block^).
```

Le mot (word) supérieur de l'adresse physique est la page que vous devez alors transmettre au contrôleur DMA. Le mot inférieur contient l'offset que vous devez définir. Bloquez d'abord le canal DMA pour empêcher tout accès. Supprimez ensuite le flip-flop et sortez le mode d'écriture du contrôleur DMA. Pour la sortie des données, vous avez besoin du mode $48+numéro du canal DMA, donc en fait $49 si vous utilisez le canal DMA n° 1. L'étape suivante

consiste à poser l'octet Low, puis l'octet High de l'offset de l'adresse physique. A la suite de quoi, vous sortirez d'abord l'octet Low, puis l'octet High correspondant à la taille du bloc à transférer, avant d'envoyer la page de l'adresse physique. De cette façon, le contrôleur DMA est prêt au transfert et vous pouvez passer à la programmation de la carte Sound Blaster. Envoyez d'abord la commande $14 pour la sortie 8 bits via DMA. Envoyez ensuite l'octet Low et l'octet High correspondant à la taille du bloc à transférer. De cette façon, vous avez également programmé la carte Sound Blaster. Libérez alors le canal DMA et la sortie du son commence.

Vous trouverez dans la procédure suivante, nommée Jouer_SB, un exemple de programmation correcte de l'ensemble. Vous devez transmettre à cette procédure comme premier paramètre la page de l'adresse physique du bloc sample. Le second paramètre contiendra la partie offset de l'adresse physique du bloc sample. Le troisième paramètre, enfin, contiendra sa longueur.

```
procedure Jouer_Sb(Segm,Offs,dsize : word);
{
 Cette procédure exécute le bloc adressé par Segm:Offs
 de taille dsize. Il faut noter que le contrôleur DMA ne peut
 PAS travailler en accédant aux pages...
}
var li : word;
begin;
  port[$0A] := dma_ch+4;                    { Bloque le canal DMA }
  Port[$0c] := 0;                           { Adresse du tampon (blk)  }
  Port[$0B] := $48+dma_ch;                  { Pour la sortie du son }
  Port[dma_adr[dma_ch]] := Lo(offs);        { au DMA-Controller }
  Port[dma_adr[dma_ch]] := Hi(offs);
  Port[dma_wc[dma_ch]] := Lo(dsize-1);      { Taille du bloc }
  Port[dma_wc[dma_ch]] := Hi(dsize-1);      { au DMA-Controller }
  Port[dma_page[dma_ch]] := Segm;
  Wr_dsp($14);
  Wr_dsp(Lo(dsize-1));                      { Taille du bloc }
  Wr_dsp(Hi(dsize-1));                      { à la DSP }
  Port[$0A] := dma_ch;                      { Libère le canal DMA }
end;
```

16h - Sortie des samples comp. 2 bits via DMA

La sortie des samples 2 bits comprimés se fait de la même façon que pour les samples 8 bits. Vous devez simplement remplacer la commande $14 par la commande $16 pour la programmation de la carte Sound Blaster.

17h - Sortie des samples comp. 2 bits avec octet de référence via DMA

Pour la commande 17h, nous pouvons répéter ce qui a été dit à propos de la commande 16h. La seule différence tient dans le fait que le premier octet est interprété comme octet de référence. C'est pour ainsi dire la valeur de départ à partir de laquelle sont formées les différences. Le plus souvent, il faut d'abord sortir un bloc avec octet de référence par l'intermédiaire de la commande 17h, puis des blocs sans octet de référence.

20h - Enregistrement direct

La commande 20h est utilisée pour l'enregistrement direct des données sample 8 bits. Si vous voulez enregistrer au moyen de cette commande, vous devrez suivre les étapes suivantes :

- Envoyer la commande 20h
- Lire l'octet sample.

Notez ici encore que vous devez répéter les deux étapes périodiquement à intervalles constants. Cette méthode ne convient que pour l'enregistrement à vitesse réduite des samples. Pour l'enregistrement de haute qualité, il faut utiliser la méthode DMA.

24h - Enregistrement de samples 8 bits via DMA

Quand vous voulez enregistrer des données via DMA, vous devez d'abord détourner l'interruption SB sur une routine qui vous est propre et programmer ensuite correctement le contrôleur DMA. L'étape suivante consiste à déterminer et à poser la vitesse du sample. Procédez comme suit :

- Envoyez la commande 24h,
- Envoyez l'octet Low de la longueur du sample -1
- Envoyez l'octet High de la longueur du sample -1

L'enregistrement démarre dès que vous avez terminé la troisième étape. Notez que vous ne pouvez pas enregistrer des blocs supérieurs à 64 Ko. Les blocs de cette taille doivent être partagés en plusieurs sous-blocs.

Dans la pratique, vous devrez procéder comme suit : en premier lieu, vous calculerez l'adresse physique du bloc à enregistrer, selon la formule :

```
Physpos:=16*sgm(Block^)+ofs(Block^).
```

Le mot (word) supérieur de l'adresse physique est la page que vous devez transmettre au contrôleur DMA. Le mot inférieur contient l'offset que vous devez définir. Bloquez d'abord le canal DMA pour empêcher tout accès. Supprimez ensuite le flip-flop et sortez la valeur correspondant au mode d'écriture du contrôleur DMA. Pour la sortie des données, vous avez besoin du mode $44+numéro du canal DMA, donc en fait $45 si vous utilisez le canal DMA n° 1. L'étape suivante consiste à poser l'octet Low, puis l'octet High de l'offset de l'adresse physique. A la suite de quoi vous enverrez d'abord l'octet Low, puis l'octet High correspondant à la taille du bloc à transférer, avant d'envoyer la page de l'adresse physique. De cette façon, le contrôleur DMA est prêt au transfert et vous pouvez passer à la programmation de la carte Sound Blaster. Envoyez d'abord la commande $24 pour l'enregistrement 8 bits via DMA. Envoyez ensuite l'octet Low et l'octet High correspondant à la taille du bloc à transférer. De cette façon, vous avez également programmé la carte Sound Blaster. Libérez alors le canal DMA et l'enregistrement commence.

Vous trouverez dans la procédure suivante, nommée Record_SB, un exemple de programmation correcte de l'ensemble. Vous devez transmettre à cette procédure comme premier paramètre la page de l'adresse physique du bloc sample. Le second paramètre contiendra la partie offset de l'adresse physique du bloc sample. Le troisième paramètre, enfin, contiendra sa longueur.

```
procedure Record_Sb(Segm,Offs,dsize : word);
{
 Cette procédure reçoit "dsize" fichiers dans le bloc adressé
 par l'intermédiaire de Segm:offs.
 Notez que le contrôleur DMA ne fonctionne PAS en accédant aux
 pages...
}
var li : word;
begin;
  port[$0A] := dma_ch+4;                { Bloque le canal DMA }
  Port[$0c] := 0;                       { Adresse du tampon (blk)  }
  Port[$0B] := $44+dma_ch;              { Pour recevoir }
  Port[dma_adr[dma_ch]] := Lo(offs);    { au contrôleur DMA }
  Port[dma_adr[dma_ch]] := Hi(offs);
  Port[dma_wc[dma_ch]] := Lo(dsize-1);  { Taille du bloc }
  Port[dma_wc[dma_ch]] := Hi(dsize-1);  { au contrôleur DMA }
```

```
  Port[dma_page[dma_ch]] := Segm;
  Wr_dsp($14);
  Wr_dsp(Lo(dsize-1));                    { Taille du bloc }
  Wr_dsp(Hi(dsize-1));                    { à la DSP }
  Port[$0A] := dma_ch;                    { Libérer le canal DMA }
end;
```

30h - Entrée MIDI directe

La commande 30h sert à enregistrer directement des données MIDI. Le procédé est le suivant :

- Envoyer 30h à la DSP,
- Lire le port d'état jusqu'à ce que le bit 7 soit posé,
- Lire les données dans le port de données.

Lorsque le bit 7 est posé dans l'octet de données, il s'agit d'un octet de contrôle. Les étapes 2 et 3 doivent être ensuite répétées pour parvenir à la note. Sinon, celle-ci se présente directement.

31h - Entrée MIDI via interruption

Pour que vous ne soyez pas sans cesse obligé dans votre programme de vérifier si le port MIDI contient des données, la carte SB vous propose la commande 31h. Celle-ci permet de lire des données MIDI contrôlées par une interruption. Pour cela, vous devez d'abord détourner l'interruption SB sur votre propre routine. Envoyez ensuite la commande 31h. S'il y a des données MIDI dans le port, l'interruption SB est déclenchée et vous pouvez les recevoir. Procédez donc comme suit :

- Lire et emmagasiner l'octet qui se trouve dans le port de données.
- Lire le port d'état.
- Mettre fin à l'interruption.

32h - Entrée MIDI directe avec timestamp

Pour respecter le timing des données MIDI, vous disposez du "timestamp". Celui-ci est une valeur sur 24 bits, qui contient le nombre de millisecondes écoulées depuis l'envoi de la commande 32h. Procédez comme suit :

- Ecrire la commande 32h.
- Tester si le bit 7 du port d'état est posé (a la valeur 1).
- Lire l'octet inférieur du timestamp.
- Lire l'octet du milieu dans le timestamp.

- Lire l'octet supérieur dans le timestamp.
- Lire le code MIDI.

Les étapes 2 à 6 doivent être répétées continuellement.

33h - Entrée MIDI avec timestamp via une interruption

La commande fonctionne exactement comme la précédente, à la différence près que vous devez ici lire d'abord les trois octets du timestamp avant de lire le code MIDI.

34h - Mode MIDI-UART, direct

Une fois passé dans ce mode, vous pouvez entrer et sortir directement des données MIDI. Vous n'avez plus besoin d'envoyer la commande 38h. Vous pouvez au contraire écrire directement dans la puce DSP.

35h - Mode MIDI-UART via une interruption

Cette commande a en principe les mêmes effets que la commande 34h. La différence est qu'elle est contrôlée par l'intermédiaire d'une interruption. Vous trouverez un exemple de son traitement sous la commande 32h.

37h - MIDI-UART via interruption avec timestamp

Le traitement des données se fait ici aussi par l'intermédiaire de l'interruption. La structure de la routine correspond à celle de la commande 35h, mais vous devez d'abord lire les trois octets du timestamp avant de saisir le code MIDI.

38h - Envoyer un code MIDI

Pour envoyer un code MIDI pour le contrôle d'un des appareils connectés, vous devrez procéder comme suit :

- Envoyez la commande 38h.
- Lisez le port de commande jusqu'à ce que le bit 7 ne soit plus posé (=0).
- Ecrivez votre code MIDI.

40h - Définir une vitesse de sample

Cette commande est très importante. Elle permet de définir la vitesse de sampling de la carte Sound Blaster. Vous en avez donc besoin dans tous les programmes où vous utilisez une sortie numérique. Procédez comme suit :

- Envoyez la commande 40h
- Envoyez la constante TC calculée.

Cette constante TC sera calculée pour les modes Normal-Speed et les modes high Speed (à partir de SB Pro). Pour les modes Normal-Speed, employez la formule suivante :

```
TC:=256-(1.000.000 fréquence DIV)
```

Pour le mode High-Speed, la formule est la suivante :

```
TC:=65536-(256.000.000 fréquence DIV)
```

Attention ! Seul l'octet supérieur de la constante TC calculée sera envoyé. L'octet inférieur sera simplement ignoré.

45h - DMA-Multiblock Continue SB16

Sur une SB16, vous avez la possibilité de poursuivre le transfert d'un bloc au moyen de cette commande, au lieu de réaliser la programmation par l'intermédiaire des commandes B6h, B9h, C6h, C9h. Dans ce cas, vous n'êtes pas obligé d'envoyer à nouveau la longueur et le mode de transfert. Vous pouvez le faire directement par l'intermédiaire de cette commande 45h. Vous envoyez ainsi un bloc de même taille que le bloc précédemment transféré.

48h - Spécifier la taille du bloc

Par l'intermédiaire de cette commande, vous spécifiez la taille du bloc de sampling transféré en mode High-Speed. Pour cela, vous devez

- envoyer d'abord la commande 48h,
- envoyer l'octet inférieur correspondant à la taille du bloc -1,
- envoyer l'octet supérieur correspondant à la taille du bloc -1.

74h - Sortie de samples comp. 4 bits via DMA

Pour la sortie des samples comprimés 4 bits, reportez-vous à ce qui a été dit pour la commande 14h.

75h - Sortie de samples comp. 4 bits avec référence via DMA

Reportez-vous ici aux explications données pour la commande 17h.

76h - Sortie de samples comp. 2, 6 bits via DMA

Reportez-vous à la commande 14h.

77h - Sortie de samples comp. 2, 6 bits avec octet de référence via DMA

Reportez-vous à la commande 17h.

80h - Définir un bloc de silence

Par l'intermédiaire de cette commande, vous pouvez définir et sortir un bloc de silence. Procédez comme suit :

- Détournez l'interruption SB sur votre propre routine.
- Définissez la vitesse souhaitée pour le sampling (commande 40h).
- Envoyez la commande 80h.
- Envoyez l'octet Low de la taille du bloc -1.
- Envoyez l'octet High de la taille du bloc -1.

Quand le bloc de silence a été envoyé en entier, SB génère une interruption.

91h - Sortie 8 bits via DMA en High-Speed

Cette commande permet d'envoyer des données de sampling avec une qualité allant jusqu'à 44 KHz. Il faut procéder pour cela comme pour une sortie normale.

- Détournez l'interruption SB sur votre propre routine.
- Initialisez le contrôleur DMA.
- Définissez la vitesse souhaitée pour le sampling par l'intermédiaire de la commande 40h.
- Envoyez la commande 48h, pour définir la taille du bloc.
- Envoyez l'octet Low de la taille du bloc -1.
- Envoyez l'octet High de la taille du bloc -1.
- Envoyez la commande 91h.

La sortie des données commence immédiatement après l'envoi de la commande 91h. A la fin du transfert, la carte Sound Blaster génère une interruption.

99h - Entrée 8 bits via DMA en High-Speed

L'enregistrement de données en mode High-Speed se fait exactement comme pour la sortie des données. La seule différence est que vous devez envoyer ici la commande 99h au lieu de 91h.

B6h - Sortir des données 16 bits sur SB16 via DMA

A partir de la version SB16, vous avez aussi la possibilité de transformer des données 16 bits en samples et de les reproduire. Le procédé est le même que pour la sortie traditionnelle des données via DMA. Pourtant, vous n'êtes plus obligés de préciser au SB16 par l'intermédiaire de la puce de mixage si les données doivent être transférées en mode Mono ou Stéréo. Vous pouvez le faire savoir à la puce DSP au moment de l'initialisation du transfert. Procédez comme suit :

- Détournez l'interruption SB sur votre propre routine.
- Initialisez le contrôleur DMA.
- Envoyez la commande B6h.
- Envoyez 00h pour la sortie des données Mono ou 20h pour les données Stéréo.
- Envoyez l'octet Low de la taille du bloc -1.
- Envoyez l'octet High de la taille du bloc -1.

La sortie commence dès que vous avez envoyé la longueur du bloc. Lorsque la sortie des données prend fin, le SB génère (théoriquement) une interruption.

B9h - Enregistrer des données 16 bits sur SB16 via DMA

L'enregistrement des données 16 bits se fait exactement comme la reproduction des données. La seule différence est que vous devez envoyer la commande B9h au lieu de B6h.

C6h - Sortir des données 8 bits sur SB16 via DMA

La sortie des données 8 bits sur la carte SB16 se fait exactement comme la sortie des données 16 bits, à cette différence près que vous devez envoyer ici la commande C6h au lieu de B6h. Reportez-vous à cette dernière.

C9h - Enregistrer des données 8 bits sur SB16 via DMA

On peut répéter ici ce qui a été dit pour la commande B9h. Vous ferez intervenir la commande C9h au lieu de B9h.

D0h - Arrêter DMA

Cette commande permet d'interrompre le transfert des données via DMA. Par l'intermédiaire de la commande D4h, vous pouvez reprendre la sortie ainsi interrompue.

D1h - Mettre le haut-parleur en service

Pour entendre les données que vous sortez, vous devez d'abord mettre en service le haut-parleur de la carte Sound Blaster. Vous le ferez à l'aide de cette commande D1h.

D3h - Mettre le haut-parleur hors service

Cette commande permet de mettre hors service le haut-parleur de la carte Sound Blaster.

D4h - Continuer DMA

Grâce à cette commande, vous pouvez poursuivre la sortie de données via DMA, quand elle a été interrompue par la commande D0h.

D8h - Interroger le réglage du haut-parleur

Pour savoir si le haut-parleur de la carte SB a été mis en service, vous disposez de cette fonction à partir de SB Pro. Elle permet de connaître l'état du haut-parleur. Envoyez pour cela la commande D8h et lisez ensuite un octet en provenance de la DSP. Si elle contient la valeur 0, cela veut dire que le haut-parleur est en service. Sinon, il est hors service.

E1h - Interroger la version

Le numéro de version de la carte Sound Blaster sera obtenu par l'intermédiaire de cette commande. Vous devez pour cela :

- envoyer la commande E1h,
- lire le numéro principal de la version,
- lire le numéro de sous-version.

Reconnaître la carte Sound Blaster

Quand vous installez la carte Sound Blaster, le programme d'installation inscrit la variable d'environnement BLASTER dans le fichier AUTOEXEC.BAT. A l'aide de cette variable, vous pouvez reconnaître de façon simple et sans risque les paramètres de la carte SB dans les cas habituels. Pourtant, que ferez-vous quand l'utilisateur veut entendre le son alors que la variable d'environnement n'est pas définie ?

La réponse est très simple. Nous devons essayer de reconnaître la carte par des moyens matériels. Cette méthode a évidemment ses limites. Tout d'abord, l'exécution de ce test de reconnaissance ne va pas sans "danger", car des cartes de réseau ou des puces S3 n'aiment pas qu'on écrive de manière sauvage sur leurs registres. C'est pourquoi vous devez vous montrer prudent. Les cartes ne seront sans doute pas abîmées, mais il se peut que le système se bloque.

Le port de base la carte SB est facilement reconnaissable. Avec l'interruption, il faut faire preuve d'un peu d'astuce. Devant le canal DMA, nous ne pouvons que capituler et laisser l'utilisateur indiquer le numéro de son canal, lorsque la carte SB ne se trouve pas sur 1 (ce qui est pourtant le cas la plupart du temps).

Comment reconnaît-on donc le port de base de la carte Sound Blaster ? Très simplement. Les adresses possibles de la SB sont situées entre 200h et 280h et elles peuvent varier de 10h en 10h. Il suffit donc de tester toutes les adresses et de vérifier si la SB se trouve ou non à cette adresse. Si c'est le cas, tout va bien dans le meilleur des mondes. Si au contraire vous ne trouvez pas de Sound Blaster même en 280h, vous devez commencer à vous poser des questions. Il se peut qu'il n'y ait pas du tout de carte Sound Blaster installée dans votre ordinateur (peut-être s'agit-il plutôt d'une GRAVIS ULTRASOUND avec SBOS). Il se peut aussi que nos routines de reconnaissance n'aient pas accompli le travail qu'on attendait d'elles. Le second cas doit normalement être exclu si vous avez indiqué le bon canal DMA. Si ce canal est erroné, la reconnaissance ne pourra évidemment pas aboutir.

A quoi voit-on qu'une carte Sound Blaster se trouve à l'adresse en cours ? Il suffit d'exécuter pour cela un reset. S'il réussit, cela veut dire qu'une SB se trouve à cette adresse. Sinon, vous devez poursuivre votre investigation.

Les routines nécessaires à cet effet sont Reset_SBCard et Detect_SBReg, que nous avons déjà rencontré.

Une fois que le port de base de la carte Sound Blaster est connu, nous pouvons passer à la reconnaissance de l'interruption. Le principe est ici le suivant : si nous transférons un bloc par l'intermédiaire de DMA, une interruption est toujours déclenchée à la fin de ce transfert. Nous concevrons donc notre routine de test pour l'interruption de telle façon qu'un très court bloc soit envoyé via DMA. Nous attendrons ensuite un peu pour donner à la SB le temps de déclencher une interruption. Cette opération sera répétée dans une boucle. Dans celle-ci, l'un des candidats potentiels à l'interruption sera détourné sur notre propre routine. Quand on passera à cette routine, il faudra poser un flag pour indiquer que l'interruption a été trouvée. Dans la boucle, on peut aussi vérifier si ce flag est posé. Si c'est le cas, l'interruption en question a été trouvée et on peut sortir de la boucle. Sinon, il faut continuer.

La routine Detect_SBIRQ montre comment traduire ce principe dans les faits. Veillez à sauvegarder les anciennes valeurs de l'interruption et à les restaurer après traitement de la routine. Vous devez aussi désactiver le haut-parleur avant de transférer le bloc. Les bruits qu'on entend pendant le test ne sont pas particulièrement plaisants.

```
procedure detect_sbIRQ;
{
 Cette routine reconnaît l'interruption de la carte Sound
 Blaster. Pour cela on teste toutes les interruptions possibles.
 On envoie des blocs courts via DMA. Si la sortie se conclut
 par le saut à l'interruption indiquée, celle-ci a été trouvée.
}
const irqs_possibles : array[1..5] of byte = ($2,$3,$5,$7,$10);
var i : integer;
    h : byte;
begin;
 getintvec($8+dsp_irq,intback);          { Sauvegarde les valeurs ! }
 port21 := port[$21];
 getmem(blk,1200);
 fillchar(blk^,1200,127);
 set_Timeconst_sb16(211);
 Wr_dsp($D3);                            { Eteint le haut-parleur }
 i := 1;
 interrupt_check := true;
 while (i <= 5) and (not IRQDetected) do
   begin;
     dsp_irq := irqs_possibles[i];       { IRQ à tester }
     getintvec($8+dsp_irq,oldint);       { Interruption détournée }
     setintvec($8+dsp_irq,@Dsp_Int_sb16);
     irqmsk := 1 shl dsp_irq;
     port[$21] := port[$21] and not irqmsk;
     Sampling_Rate := 211;
     taille_bloc := 1200;                { Sortie pour test }
     Jouer_Block_Dsp(taille_bloc,blk,true,false);
     delay(150);
     setintvec($8+dsp_irq,oldint);       { Rétablit l'interrupt }

     port[$21] := Port[$21] or irqmsk;
```

```
      h := port[dsp_adr+$E];
      Port[$20] := $20;
      inc(i);
    end;
  interrupt_check := false;
  Wr_dsp($D1);                          { Rétablit le haut-parleur }
  freemem(blk,1200);
  setintvec($8+dsp_irq,intback);        { Rétablit les valeurs !!! }
  port[$21] := port21;
  dsp_rdy_sb16 := true;
end;
```

Mixer les données du son - puce de mixage

La carte Sound Blaster possède depuis la version SB Pro non seulement une puce DSP, mais aussi une puce de mixage. Celle-ci est prévue pour le réglage des entrées et des sorties de la carte Sound Blaster. En outre, elle permet de régler les volumes et la balance. La puce de mixage de la SB16 est à moitié compatible en amont (une fois de plus, les concepteurs ont pris leur tâche à la légère). Si l'on veut utiliser ses fonctions étendues, il faut la programmer par l'intermédiaire d'autres registres.

La puce sera adressée par l'intermédiaire du couple de registres situés aux adresses 2x4h et 2x5h. 2x4h est le port de sélection du mixer et 2x5h est le port des données.

Quand vous désirez programmer le mixer, vous devez sélectionner le registre à modifier par l'intermédiaire du port de sélection et vous pouvez alors lire et écrire les données dans le port de données.

Passons en revue les registres de la puce de mixage qui équipe la SB Pro :

Registre 00h - Reset

Ce registre sert à ramener la puce de mixage à son état par défaut. Procédez pour cela comme suit :

- Ecrivez la valeur 0 dans le registre d'index.
- Attendez environ 100 ms.
- Ecrivez la valeur 0 dans le registre de données.

Registre 02h - haut-parleur DSP

Ce registre sert à régler le volume. Les bits 7-5 sont responsable du volume pour le canal de gauche. Le volume peut ainsi prendre une valeur allant de 0

à 7. De même, les bits 3-1 sont responsables du volume pour le canal de droite. Pour régler le volume, procédez comme suit :

- Ecrivez la valeur 02h dans le registre d'index.
- Calculer le volume pour la SB Pro selon la formule :

```
Volume:=(left shl 5)+(right shl 1);
```

- Ecrivez le volume dans le registre de données.

Registre 0Ah - Volume du microphone

Ce registre sert à régler le volume du microphone. Les valeurs possibles vont de 0 à 3. Le volume sera défini comme suit :

- Ecrivez la valeur 0Ah dans le registre d'index.
- Calculez le volume grâce à la formule :

```
Volume:=(valeur shl 1);
```

- Ecrivez le volume dans le registre de données.

Registre 0Ch - Réglage des filtres d'entrée

Ce registre permet d'indiquer les réglages pour l'enregistrement. Vous pouvez choisir un filtre et la source d'entrée. Si le bit 5 n'est pas posé, le filtre est en service. Sinon, il est hors service. Le bit 3 permet de régler le degré de perméabilité du filtre. Si le bit 3 est posé, il est faible. S'il n'est pas posé, il est au contraire élevé. Les bits 2-1 règlent le choix de la source d'entrée. 0 correspond à l'entrée par microphone, 1 à l'entrée par CD-ROM et 3 au line-input. Pour définir le registre, procédez comme suit :

- Ecrivez la valeur $0C dans le registre d'index.
- Calculez la valeur du réglage :
  Quand le filtre est en service,

```
Ew:=(1 shl 5)
```

  Quand la perméabilité est faible,

```
Ew:=Ew+(1 shl 3)
Ew:=Ew+(Source shl 1)
```

- Ecrivez la valeur du réglage dans le registre de données.

Registre 0Eh - Filtre de sortie et sélection stéréo

Ce registre a deux fonctions. D'une part, il sert à indiquer si l'on veut ou non choisir un filtre de sortie. D'autre part, il permet de passer de la sortie Mono à la sortie Stéréo.

Lorsque le bit 5 n'est pas posé, le filtre est en service. Sinon, il est hors service. Si le bit 1 est posé, la sortie se fait en stéréo, sinon en mono. Le registre sera réglé de la façon suivante :

- Ecrivez la valeur $0Eh dans le registre d'index.
- Si le filtre est en service, valeur-registre:=(1 shl 5)
- En Stéréo, valeur-registre:=valeur-registre+(1 shl 1)
- Ecrivez la valeur du registre dans le registre de données.

Registre 22h - Haut-parleur maître

Ce registre sert à régler le haut-parleur général. Les bits 7-5 servent à régler le volume du canal de gauche. Le volume peut prendre sur la SB Pro une valeur allant de 0 à 7. De même, les bits 3-1 règlent le volume du canal de droite. Pour définir le volume, procédez comme suit :

- Ecrivez la valeur 22h dans le registre d'index.
- Calculez le volume pour la SB Pro :

```
Volume=(left shl 5)+(right shl 1)
```

- Ecrivez le volume dans le registre de données.

Registre 26h - Volume FM

Ce registre sert à régler le volume du canal FM. Pour le reste, ce que nous avons dit pour le registre 22h vaut ici aussi. Le volume pourra être réglé de la façon suivante :

- Ecrivez la valeur 26h dans le registre d'index.
- Calculez le volume en fonction de :

```
Volume=(left shl 5)+(right shl 1)
```

- Ecrivez le volume dans le registre de données.

Registre 28h - Volume CD

Ce registre sert à régler le volume du canal CD. Pour le reste, ce que nous avons dit pour le registre 22h vaut ici aussi. Le volume pourra être réglé de la façon suivante :

- Ecrivez la valeur 28h dans le registre d'index.
- Calculez le volume en fonction de :

```
Volume=(left shl 5)+(right shl 1)
```

- Ecrivez le volume dans le registre de données.

Registre 2Eh - Volume Line

Ce registre sert à régler le volume du canal Line. Pour le reste, ce que nous avons dit pour le registre 22h vaut ici aussi. Le volume pourra être réglé de la façon suivante :

- Ecrivez la valeur 2Eh dans le registre d'index.
- Calculez le volume en fonction de :

```
Volume=(left shl 5)+(right shl 1)
```

- Ecrivez le volume dans le registre de données.

Maintenant que vous avez fait connaissance avec la puce de mixage de la SB Pro, la puce améliorée de la SB 16 ne vous posera aucun problème. Voici une vue d'ensemble de ses registres :

Les registres de la puce de mixage sur la SB16 (ASP)

Register 48: Left Master Volume

Bit numéro	Fonction	Valeur par défaut
Bit 0:	- - -	0
Bit 1:	- - -	0
Bit 2:	- - -	0
Bit 3:	Master Volume	0
Bit 4:	Master Volume	0
Bit 5:	Master Volume	0
Bit 6:	Master Volume	1
Bit 7:	Master Volume	1

Register 49: Right Master Volume

Bit numéro	Fonction	Valeur par défaut
Bit 0:	- - -	0
Bit 1:	- - -	0
Bit 2:	- - -	0
Bit 3:	Master Volume	0
Bit 4:	Master Volume	0
Bit 5:	Master Volume	0
Bit 6:	Master Volume	1
Bit 7:	Master Volume	1

Register 50: Left Voice Volume

Bit numéro	Fonction	Valeur par défaut
Bit 0:	- - -	0
Bit 1:	- - -	0
Bit 2:	- - -	0
Bit 3:	- - -	0
Bit 4:	Volume	0
Bit 5:	Volume	0
Bit 6:	Volume	1
Bit 7:	Volume	1

Register 51: Right Voice Volume

Bit numéro	Fonction	Valeur par défaut
Bit 0:	- - -	0
Bit 1:	- - -	0
Bit 2:	- - -	0
Bit 3:	- - -	0
Bit 4:	Volume	0
Bit 5:	Volume	0
Bit 6:	Volume	1
Bit 7:	Volume	1

Register 52: Left MIDI Volume

Bit numéro	Fonction	Valeur par défaut
Bit 0:	- - -	0
Bit 1:	- - -	0
Bit 2:	- - -	0
Bit 3:	- - -	0
Bit 4:	MIDI Volume	0
Bit 5:	MIDI Volume	0
Bit 6:	MIDI Volume	1
Bit 7:	MIDI Volume	1

Register 53: Right MIDI Volume

Bit numéro	Fonction	Valeur par défaut
Bit 0:	- - -	0
Bit 1:	- - -	0
Bit 2:	- - -	0
Bit 3:	- - -	0
Bit 4:	MIDI Volume	0
Bit 5:	MIDI Volume	0
Bit 6:	MIDI Volume	1
Bit 7:	MIDI Volume	1

Register 54: Left CD Volume

Bit numéro	Fonction	Valeur par défaut
Bit 0:	- - -	0
Bit 1:	- - -	0
Bit 2:	- - -	0
Bit 3:	- - -	0
Bit 4:	Volume	0
Bit 5:	Volume	0
Bit 6:	Volume	0
Bit 7:	Volume	0

Register 55: Right CD Volume

Bit numéro	Fonction	Valeur par défaut
Bit 0:	- - -	0
Bit 1:	- - -	0
Bit 2:	- - -	0
Bit 3:	- - -	0
Bit 4:	Volume	0
Bit 5:	Volume	0
Bit 6:	Volume	0
Bit 7:	Volume	0

Register 56: Left Line Volume

Bit numéro	Fonction	Valeur par défaut
Bit 0:	- - -	0
Bit 1:	- - -	0
Bit 2:	- - -	0
Bit 3:	- - -	0
Bit 4:	Volume	0
Bit 5:	Volume	0
Bit 6:	Volume	0
Bit 7:	Volume	0

Register 57: Right Line Volume

Bit numéro	Fonction	Valeur par défaut
Bit 0:	- - -	0
Bit 1:	- - -	0
Bit 2:	- - -	0
Bit 3:	- - -	0
Bit 4:	Volume	0
Bit 5:	Volume	0
Bit 6:	Volume	0
Bit 7:	Volume	0

Register 58: Mic Volume		
Bit numéro	**Fonction**	**Valeur par défaut**
Bit 0:	- - -	0
Bit 1:	- - -	0
Bit 2:	- - -	0
Bit 3:	- - -	0
Bit 4:	Volume	0
Bit 5:	Volume	0
Bit 6:	Volume	0
Bit 7:	Volume	0

Register 59: Pc Speaker Volume		
Bit numéro	**Fonction**	**Valeur par défaut**
Bit 0:	- - -	0
Bit 1:	- - -	0
Bit 2:	- - -	0
Bit 3:	- - -	0
Bit 4:	- - -	0
Bit 5:	- - -	0
Bit 6:	Volume	0
Bit 7:	Volume	0

Register 60: Output Controll Register		
Bit numéro	**Fonction**	**Valeur par défaut**
Bit 0:	Mic	1
Bit 1:	CD droite	1
Bit 2:	CD gauche	1
Bit 3:	Line droite	1
Bit 4:	Line gauche	1
Bit 5:	- - -	0
Bit 6:	- - -	0
Bit 7:	- - -	0

Register 61: Left In Controll Register

Bit numéro	Fonction	Valeur par défaut
Bit 0:	Mic	1
Bit 1:	CD droite	0
Bit 2:	CD gauche	1
Bit 3:	Line droite	0
Bit 4:	Line gauche	1
Bit 5:	MIDI droite	0
Bit 6:	MIDI gauche	0
Bit 7:	- - -	0

Register 62: Right In Controll Register

Bit numéro	Fonction	Valeur par défaut
Bit 0:	Mic	1
Bit 1:	CD droite	1
Bit 2:	CD gauche	0
Bit 3:	Line droite	1
Bit 4:	Line gauche	0
Bit 5:	MIDI droite	0
Bit 6:	MIDI gauche	0
Bit 7:	- - -	0

Register 63: Left Input Gain Factor

Bit numéro	Fonction	Valeur par défaut
Bit 0-5:	- - -	0
Bit 6:	Gain Factor	
Bit 7:	Gain Factor	

La valeur codée dans les deux octets est celui qui dénote le déplacement (shift) à gauche de la valeur d'entrée. ***Exemple :***
Bit 6: 0; Bit7: 1 => Gain Factor = 4 (1 shl 2)

Register 64: Right Input Gain Factor

Bit numéro	Fonction	Valeur par défaut
Bit 0-5:	- - -	0
Bit 6:	Gain Factor	
Bit 7:	Gain Factor	

Register 65: Left-Output Gain Factor

Bit numéro	Fonction	Valeur par défaut
Bit 0-5:	- - -	0
Bit 6:	Gain Factor	
Bit 7:	Gain Factor	

Register 66: Right Output Gain Factor

Bit numéro	Fonction	Valeur par défaut
Bit 0-5:	- - -	0
Bit 6:	Gain Factor	
Bit 7:	Gain Factor	

Register 67: AGC ON/OFF (AGC = Automatic Gain Controll)

Bit numéro	Fonction	Valeur par défaut
Bit 0:	AGC en/hors service	0
Bit 1-7:	- - -	0

Attention ! Quand Bit 0 posé (!), AGC = OFF.

Register 68: Left Treble

Bit numéro	Fonction	Valeur par défaut
Bit 0:	- - -	0
Bit 1:	- - -	0
Bit 2:	- - -	0
Bit 3:	- - -	0
Bit 4:	Treble Controll	0
Bit 5:	Treble Controll	0
Bit 6:	Treble Controll	1
Bit 7:	Treble Controll	1

Register 69: Right Treble

Bit numéro	Fonction	Valeur par défaut
Bit 0:	- - -	0
Bit 1:	- - -	0
Bit 2:	- - -	0
Bit 3:	- - -	0
Bit 4:	Treble Controll	0
Bit 5:	Treble Controll	0
Bit 6:	Treble Controll	1
Bit 7:	Treble Controll	1

Register 70: Left Bass

Bit numéro	Fonction	Valeur par défaut
Bit 0:	- - -	0
Bit 1:	- - -	0
Bit 2:	- - -	0
Bit 3:	- - -	0
Bit 4:	Bass Controll	0
Bit 5:	Bass Controll	0
Bit 6:	Bass Controll	0
Bit 7:	Bass Controll	1

Register 71: Right Bass

Bit numéro	Fonction	Valeur par défaut
Bit 0:	- - -	0
Bit 1:	- - -	0
Bit 2:	- - -	0
Bit 3:	- - -	0
Bit 4:	Bass Controll	0
Bit 5:	Bass Controll	0
Bit 6:	Bass Controll	0
Bit 7:	Bass Controll	1

13.2. Sortie des fichiers VOC par programmation directe

Si vous voulez sortir un fichier VOC par l'intermédiaire de la carte SB, vous pouvez vous servir des gestionnaires de Creative Labs, qui suivent ce principe : ils produisent leurs informations non par l'intermédiaire d'une programmation directe de la carte Sound Blaster, mais par les gestionnaires eux-mêmes. Il est cependant intéressant de ne pas être entièrement dépendant d'un gestionnaire et de pouvoir programmer la carte directement. C'est pourquoi nous vous proposons ici un petit VOC-Player, qui adresse directement la carte.

Les routines présentées dans ce paragraphe constituent en même temps la base pour le MOD-Player avec la carte SB, telle que nous allons la développer au paragraphe 15.2.

Le Player utilise les routines présentées ici. Vous trouverez son code source complet sur le CD qui accompagne cet ouvrage. Nous nous limiterons donc ici aux routines essentielles pour la sortie VOC.

Commençons par la routine Init_Voc. Transmettez-lui le nom du fichier à exécuter, avec le chemin d'accès si nécessaire. La routine examine d'abord si le fichier existe. Elle vérifie ensuite le header et interprète le premier bloc du fichier VOC. A partir de ce bloc, la routine se procure les informations nécessaires sur la vitesse de sampling et le mode de sortie (Mono/Stéréo). On peut alors remplir un double tampon de données provenant du fichier et la sortie commence par envoi direct des données du tampon actuel via DMA. Les tampons sont ensuite alimentés par l'intermédiaire de l'interruption SB.

Vous voyez qu'il n'est pas si difficile que cela d'écrire un VOC-Player. Vous pouvez intégrer également dans l'interruption un traitement pour des blocs ultérieurs. Normalement, les VOC sont toutefois en un seul morceau et ils peuvent aussi être exécutés de cette façon.

```
procedure Init_Voc(filename : string);
const VOCid : string = 'Creative Voice File'+#$1A;
var ch : char;
    idstr : string;
    ct : byte;
    h : byte;
    error : integer;
    srlo,srhi : byte;
    SR : word;
```

```
    Samplingr : word;
    stereoreg : byte;
begin;
  Transfer_Testing := false;
  VOC_READY   := false;
  vocsstereo := stereo;
  stereo := false;
  assign(vocf,filename);
  reset(vocf,1);
  if filesize(vocf) < 5000 then begin;
    VOC_READY   := true;
    exit;
  end;
  blockread(vocf,voch,$19);
  idstr := voch.Idstr;
  if idstr <> VOCid then begin;
    { Identification fausse ! }
    VOC_READY   := true;
    exit;
  end;
  Blockread(vocf,inread,20);
  vblock.Identif := inread[2];
  if vblock.Identif = 1 then begin;
    vblock.SR := inread[6];
  end;
  if vblock.Identif = 8 then begin;
    SR := inread[6]+(inread[7]*256);
    Samplingr := 256000000 div (65536 - SR);
    if inread[9] = 1 then begin; {stereo}
      if sb16detected then samplingr := samplingr shr 1;
      stereo := true;
    end;
    vblock.SR := 256 - longint(1000000 DIV samplingr);
  end;
  if vblock.Identif = 9 then begin;
    Samplingr := inread[6]+(inread[7]*256);
    if inread[11] = 2 then begin;{stereo}
      stereo := true;
```

```
      if sbprodetected then samplingr := samplingr * 2;
      vblock.SR := 256 - longint(1000000 DIV (samplingr));
    end else begin;
      vblock.SR := 256 - longint(1000000 DIV samplingr);
    end;
  end;
  if vblock.SR < 130 then vblock.SR := 166;
  set_timeconst_sb16(vblock.SR);
  t_bloc := filesize(vocf) - 31;
  if t_bloc > 2500 then t_bloc := 2500;
  blockread(vocf,blk1^,t_bloc);
  ch := #0;
  fgr := filesize(vocf) - 32;
  fgr := fgr - t_bloc;
  Block_actif := 1;
  if fgr > 1 then begin;
    blockread(vocf,blk2^,t_bloc);
    fgr := fgr - t_bloc;
  end;
 Wr_dsp($D1);
 lastone := false;
 if not sb16Detected then begin;
   if Stereo then begin;
     stereoreg := Read_Mixer($0E);
     stereoreg := stereoreg OR 2;
     Write_Mixer($0E,stereoreg);
   end else begin;
     stereoreg := Read_Mixer($0E);
     stereoreg := stereoreg AND $FD;
     Write_Mixer($0E,stereoreg);
   end;
 end;
 Jouer_Block_Dsp(t_bloc,blk1,false,true);
end;

procedure voc_done;
var h : byte;
begin;
```

```
 lastone := true;
 repeat until dsp_rdy_sb16;
 close(vocf);
 Reset_SBCard;
 stereo := vocsstereo;
end;

procedure dsp_int_sb16; interrupt;
{
 On passe à cette procédure avec l'interruption générée à la fin
 d'un transfert de bloc. Si le flag dern_sortie n'a pas été posé,
 une nouvelle sortie démarre.
}
var h : byte;
begin;
  if interrupt_check then begin;
    IRQDetected := true;
  end else begin;
    if Transfer_Testing then begin;
      h := port[dsp_adr+$E];
      dsp_rdy_sb16 := true;
      if not dern_sortie then begin;
        Jouer_Block_Dsp(taille_bloc,blk,true,false);
      end;
    end else begin;
      h := port[dsp_adr+$E];
      if (fgr > t_bloc) and not lastone then begin
        lastone := false;
        if block_actif = 1 then begin
          Jouer_Block_Dsp(t_bloc,blk2,false,true);
          blockread(vocf,blk1^,t_bloc);
          fgr := fgr - t_bloc;
          block_actif := 2;
        end else begin;
          Jouer_Block_Dsp(t_bloc,blk1,false,true);
          blockread(vocf,blk2^,t_bloc);
          fgr := fgr - t_bloc;
          block_actif := 1;
```

```
        end;
      end else begin;
        if not lastone then begin;
          if block_actif = 1 then begin
            Jouer_Block_Dsp(t_bloc,blk2,false,true);
            lastone := true;
          end else begin;
            Jouer_Block_Dsp(t_bloc,blk1,false,true);
            lastone := true;
          end;
        end else begin;
          dsp_rdy_sb16 := true;
          Wr_dsp($D0);
          VOC_READY := true;
        end;
      end;
    end;
  end;
  Port[$20] := $20;
end;
```

14. Gravis Ultrasound - la carte son des mordus

La carte Gravis Ultrasound, que les connaisseurs nomment volontiers GUS, est une carte son utilisant la nouvelle technologie Wavetable. Elle se distingue par le fait que les samples ne sont pas établis à demeure dans la ROM, contrairement à ce qui se passe pour le Waveblaster de Creative Labs. Ils sont chargés dans la RAM de l'Ultrasound. Celle-ci possède normalement une taille de 256 Ko, mais peut être étendue par étapes de 256 Ko jusqu'à 1 Mo. Cet accroissement de mémoire est fortement conseillé et il peut se faire pour un prix modique. On place des DRAMS, comme on en trouve par exemple aussi sur une carte VGA ou sur une carte mère de 286. Ces DRAMS peuvent être remplacés sans problème. Il faut juste faire attention au sens dans lequel on insère les puces.

La Gravis Ultrasound est intéressante surtout par le fait qu'elle peut exécuter 32 canaux en même temps. Un canal désigne ici un secteur déterminé dans la DRAM de la Gravis. La puce son de l'Ultrasound exécute à vitesse "constante" de 44 KHz les différents samples. Les 44 KHz ne valent toutefois que pour 14 voix parmi les 32 possibles. Pour chaque nouvelle voix exécutée, la qualité de la reproduction diminue jusqu'à 22 KHz pour 28 voix. Si vous exécutez un sample à 11 KHz, il se produit des trous avec une vitesse de sortie effective de 44 KHz. Ces trous sont remplis par la puce au moyen d'une interpolation des valeurs rencontrées, ce qui augmente en partie la qualité.

La raison principale qui rend la Gravis Ultrasound intéressante pour le programmeur tient au fonctionnement indépendant de la puce. Si vous avez chargé les données de sampling dans la DRAM de la carte, vous pouvez lancer les samples au moyen d'une simple modification des paramètres concernant les différentes voix. C'est pourquoi cette carte convient particulièrement bien à l'exécution de fichiers MOD. A cause du faible laps de temps nécessaire à la modification des paramètres sur la carte, celle-ci est très intéressante pour les programmeurs de démos et de jeux. Elle permet de créer des animations qui demandent beaucoup de calculs et de les équiper d'un son très satisfaisant.

14.1. LA STRUCTURE DE LA GRAVIS ULTRASOUND

La Gravis Ultrasound comprend trois unités logiques différentes. En premier lieu, on a le connecteur MIDI de la carte. L'interface MIDI 101 est intégrée dans la puce GF1 de la carte. Pour toutes les données qui entrent et qui sortent, la carte génère une interruption. Celle-ci peut être activée ou désactivée par l'intermédiaire d'un registre de contrôle. Un registre d'état sert à déterminer la source de l'interruption.

L'unité suivante est une "Eliminator Joystick Interface", développée par Gravis. L'interface de la manette de jeu comprend un registre 8 bits. Quand une valeur a été écrite dans le registre, quatre flip-flops sont posés sur 0. Quatre compteurs enregistrent le mouvement de la manette relativement à la position au moment de l'accès en écriture. Si vous lisez alors le registre de l'interface, vous obtenez l'un après l'autre les états des quatre compteurs et l'état des flip-flops. L'interface est activée ou désactivée par l'intermédiaire d'un cavalier sur la carte.

La troisième unité, qui est aussi la plus importante de la carte Gravis Ultrasound, est la puce GF1 développée par Gravis. Cette puce s'occupe de tâches très différentes. Vous pouvez avec elle sortir des données 8 bits et 16 bits, "signed" ou "unsigned", en mono ou en stéréo. Les données peuvent être transférées soit directement, soit au moyen d'un registre DMA 8 bits ou 16 bits, dans le DRAM de la carte ou à partir du DRAM.

Le principe de fonctionnement de la carte est celui de la technologie nommée Wavetable. Celle-ci permet de restituer des sons naturels (numérisés sous forme de samples) fidèles à l'original. Au moyen d'une modification de la vitesse d'exécution, on obtient des sons plus ou moins graves ou aigus. D'autres effets peuvent être générés au moyen d'une modulation d'amplitude ou de fréquence. La carte Gravis Ultrasound offre la possibilité de régler pour chaque canal à part les paramètres concernant la fréquence de l'exécution, la modulation de l'amplitude et le "panning" (balance de la voix).

14.2. Mode de fonctionnement de la Gravis Ultrasound

Ce paragraphe est intéressant pour tous ceux qui veulent savoir comment fonctionne la carte Ultrasound au niveau le plus bas.

La GF1 est ce qu'on appelle un processeur pipeline. Il parcourt constamment une boucle allant de la voix 0 à la fin de la voix qui a été définie. Toutes les 1,6 microsecondes, la puce exécute une série d'opérations avec les différentes voix. Plus vous avez défini de voix, plus la GF1 a donc besoin de temps pour traiter l'ensemble des voix définies. Cela conduit à une baisse de qualité quand le nombre de voix augmente de façon conséquente. Vous pouvez déduire la fréquence de sortie d'après la formule

```
Freq:=1.000.000 DIV (1.619695497*voix)
```

Vous obtenez par exemple une fréquence de sortie de 30870 Hz pour 20 voix. Si vous en avez besoin dans vos programmes, il est préférable de faire le calcul des fréquences à l'avance et de les déposer dans une table. Cela vous épargnera un travail supplémentaire avec des virgules flottantes.

L'interpolation de la GF1 est l'une des raisons qui expliquent la bonne qualité du son produit par la carte. Elle est réalisée même pour les données 8 bits avec des données sur 16 bits. Voyons concrètement comment se fait l'interpolation sur un ***exemple :*** avec 20 voix actives (=30870 Hz), nous voulons sortir un sample en 10 KHz. Cela veut dire que la GF1 sera obligée d'accomplir trois parcours jusqu'à ce qu'elle parvienne à l'octet de données suivant dans le sample (car 3*10=30). La GF1 calcule alors la valeur du second parcours selon la formule

```
Valeur de sortie:=(2/3*Valeur1+1/3*Valeur2) DIV 2
```

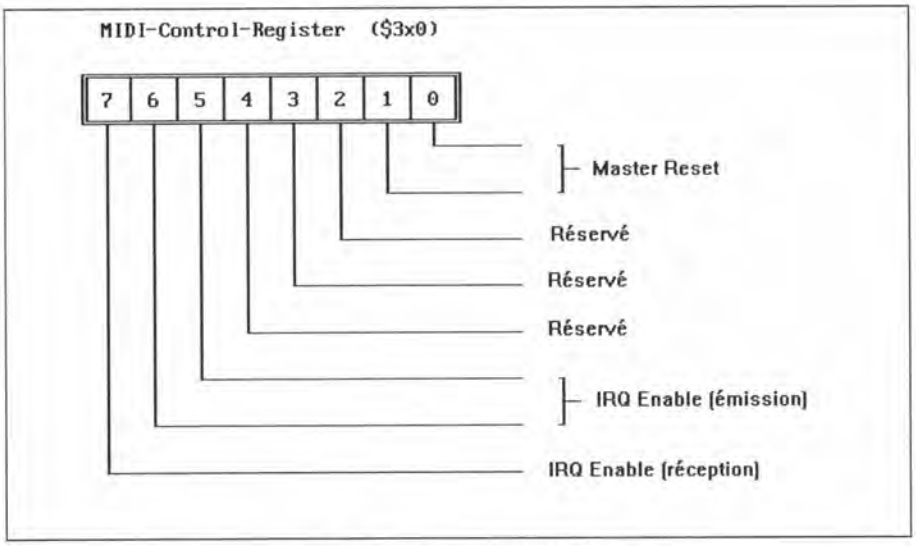
De même, la troisième valeur de sortie fait intervenir la valeur 1 au niveau de 1/3 et la valeur 2 au niveau de 2/3.

14.3. REGISTRES DE LA CARTE GUS

La carte Ultrasound peut être adressée à travers un grand nombre de registres, qui indexent en partie d'autres sous-fonctions. Le tableau suivant fournit un aperçu général de ces registres. Le x dans les indications d'adresse sera remplacé par la valeur de base spécifiée à chaque fois. Si vous avez par exemple installé la GUS avec le port de base $240, le registre de sélection des voix se trouve au port d'adresse $342.

Adresse	Mode	Description
Interface MIDI :		
$3x0	W	Registre de contrôle
$3x0	R	Registre d'état
$3x1	W	Envoyer des données
$3x1	R	Recevoir des données
Joystick Interface :		
$201	W	Lancer le timer
$201	R	Registre de données
GF1 Synthesizer :		
$2x6	R	Registre d'état IRQ
$2x8	R/W	Registre Timer Control
$2x9	W	Registres données timer
$3x2	R/W	Registre choix des voix
$3x3	R/W	Registre choix de la fonction
$3x4	R/W	Registre des données Lo Byte
$3x5	R/W	Registre des données Hi Byte
$3x7	R/W	GUS DRAM
On Board:		
$2x0	W	Registre contrôle Mixer
$2xB	W	Registre contrôle IRQ
$2xB	W	Registre contrôle DMA

Comme nous l'avons mentionné plus haut, la carte GUS comprend diverses unités servant d'interfaces. L'unité la plus intéressante pour un musicien est celle qui constitue l'interface MIDI. Elle se trouve aux adresses $3x0 et $3x1. Considérons d'abord la première d'entre elles :

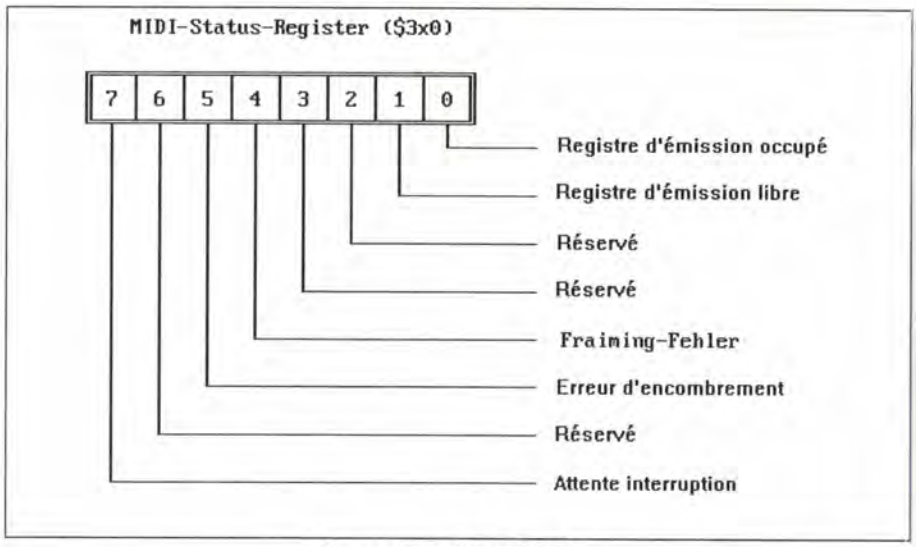


Le registre de contrôle MIDI

En écrivant à cette adresse, vous adressez le registre de contrôle MIDI. Pour exécuter un reset master, vous devez d'abord poser les bits 0 et 1 sur 1, puis sur 0 après un bref délai. Vous devrez rester ensuite à la valeur 0, pour assurer à l'ensemble un fonctionnement sans erreur.

Afin d'activer l'interruption permettant d'envoyer des données, vous devez poser le bit 5 sur 1 et le bit 6 sur 0. L'interruption permettant de recevoir les données sera activée par la position du bit 7 sur 1.

Quand vous lisez à l'adresse $3x0, vous obtenez les données du registre d'état MIDI.



Le registre d'état MIDI

La position des bits dans ce registre est expliquée par la figure. Le registre $3x0 se comporte de la même façon avec un 6850 UART, qui constitue l'interface MIDI standard.

A l'adresse $3x1 se trouve le registre MIDI des données. Il est large de 8 bits et vous pouvez y accéder en lecture aussi bien qu'en écriture. Vous y trouverez les données qui arrivent à partir d'un appareil périphérique MIDI. Si vous voulez envoyer des données, vous devez les envoyer à cette adresse de port.

Pour les programmeurs de jeux, l'interface de la manette de jeux sur la carte Advanced Gravis est particulièrement intéressante. Elle se trouve à l'adresse $201. Si vous écrivez une valeur à cette adresse, les quatre compteurs de l'interface sont posés sur 0 et ils enregistrent les mouvements de la manette. En lisant dans ce port, vous pouvez interroger l'un après l'autre les quatre compteurs et interroger l'état des flip-flops correspondants.

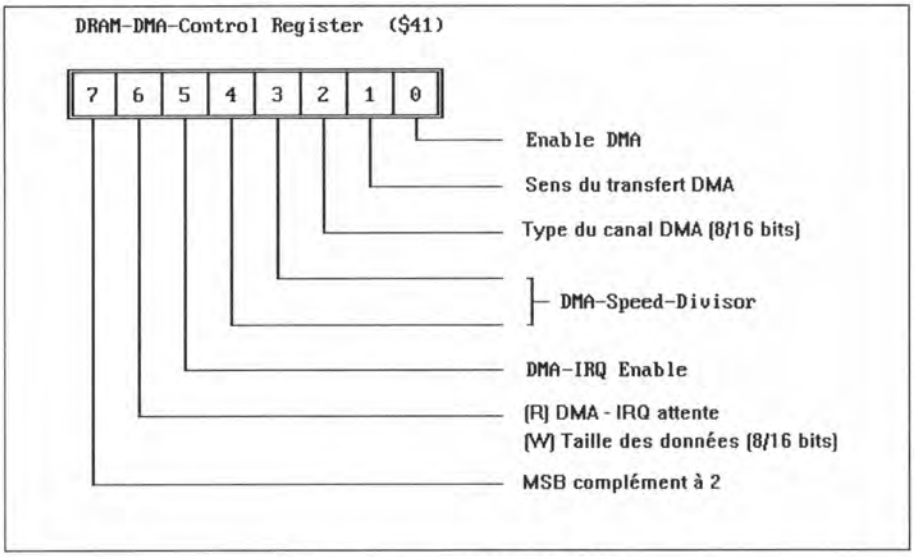
L'interface la plus importante est néanmoins l'interface GF1. Pour ne pas être obligé d'indiquer à chaque opération le numéro du canal à modifier, vous disposez d'un registre de sélection des voix. Celui-ci se situe à l'adresse $3x2 et il est large de 8 octets. On peut y accéder en lecture aussi bien qu'en écriture. Les valeurs admises vont de 0 à 31. Une valeur plus élevée risque de conduire à une déficience de la carte.

Voici un autre registre important : celui qui permet de sélectionner les fonctions. Il fait 8 bits de large et propose toute une série de sous-fonctions. Choisissez d'abord la fonction voulue au moyen du registre situé à l'adresse $3x3, puis envoyez ou lisez les données dans le port de données correspondant.

Considérez maintenant les fonctions indépendantes de la voix dans le registre de sélection des fonctions :

Fonction	Mode	Largeur	Description
$41	R/W	8	DRAM DMA Control
$42	W	16	Adresse de départ DMA
$43	W	16	DRAM I/O Adresse LO
$44	W	8	DRAM I/O Adresse Hi
$45	R/W	8	Registre de contrôle Timer
$46	W	8	Compteur Timer 1
$47	W	8	Compteur Timer 2
$48	W	8	Fréquence de sampling
$49	R/W	8	Contrôle du sampling
$4B	W	8	Réglage du joystick
$4C	R/W	8	Reset

La première fonction se trouve sous le numéro d'index $41. Elle désigne le registre de contrôle DRAM-DMA. Le registre se présente ainsi :



Registre de contrôle DRAM-DMA

Le registre contrôle le transfert des données entre la mémoire principale et la carte GUS via DMA.

Pour lancer un transfert, vous devez poser le bit 0 sur 1. Avec le bit 1, vous définissez la direction du transfert. Si ce bit prend la valeur 0, les données seront transférées de la mémoire principale vers la mémoire DRAM de la carte. Pour une valeur de 1, le transfert s'effectue en sens inverse. Le type du canal utilisé sera défini par l'intermédiaire du bit 2. Si celui-ci contient un 0, c'est un canal 8 bits (0-3) que l'on utilise. La valeur 1 désigne au contraire un canal 16 bits (4-7). Dans les bits 3 et 4, on trouvera la valeur du DMA-Speed-Divisor.

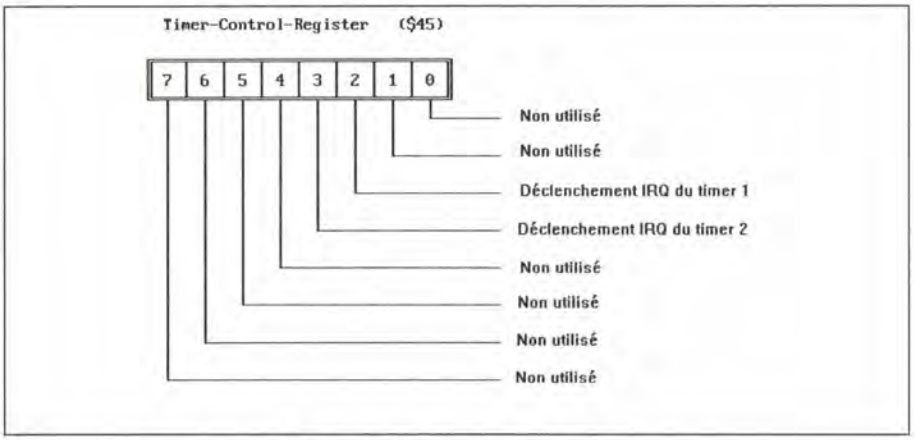
La vitesse maximale se situe aux environs de 650 KHz. Elle est divisée par le diviseur des vitesses, qui peut prendre une valeur entre 1 et 4. Le bit 5 permet d'indiquer si une interruption doit être déclenchée ou non quand le registre du bloc prend fin. La valeur 1 active l'interruption, tandis que la valeur 0 la désactive. Le bit 6 a une signification qui dépend de la direction de transfert des données.

S'il s'agit d'un accès en lecture, le bit 6 indique si l'interruption DMA doit être placée en position d'attente. Pour un transfert vers la carte, on spécifie ici la taille des données envoyées. La valeur 0 indique des données sur 8 bits. La valeur 1 indique des données sur 16 bits. La taille des données est indépendante du type que l'on a spécifié pour le canal DMA. Le bit 7, enfin, indique si le bit High doit être interverti pour amener les données dans la forme "complément à 2". Le bit High est le bit 7 pour les données sur 8 bits et le bit 15 pour les données sur 16 bits.

La fonction $42 (Adresse de départ DMA) permet de définir l'adresse de départ pour un transfert DMA dans la RAM de la carte GUS, dans les deux directions. Si vous voulez transférer des données sur 8 bits, celles-ci doivent commencer à une adresse divisible par 16. Si les données à transférer sont sur 16 bits, il faut que les données se situent à une adresse divisible par 32. La raison de cette limitation est qu'on ne peut en fait utiliser que 16 bits d'adresse, alors que 19 bits d'adresse seraient nécessaires pour adresser 1 Mo. C'est pourquoi les bits 15 à 0 correspondent aux lignes d'adresse 19 à 4.

Par l'intermédiaire des registres d'adresse DRAM-I/O $43 et $44, vous pouvez spécifier dans la DRAM l'adresse à laquelle vous pouvez écrire ou lire directement des données. Dans le registre $43, vous entrerez 16 lignes d'adresse. Dans $44, vous écrirez les 4 lignes d'adresse supérieures (bits 0 à 3). Lorsque vous avez spécifié une adresse, vous pouvez lire ou entrer l'emplacement en mémoire par l'intermédiaire du registre GUS-DRAM ($3x7).

Le registre de contrôle des timers ($45) est nécessaire pour activer ou désactiver les timers intégrés. Par l'intermédiaire du bit 2, vous activerez l'interruption du timer 1. Le bit 3 est responsable du timer 2.



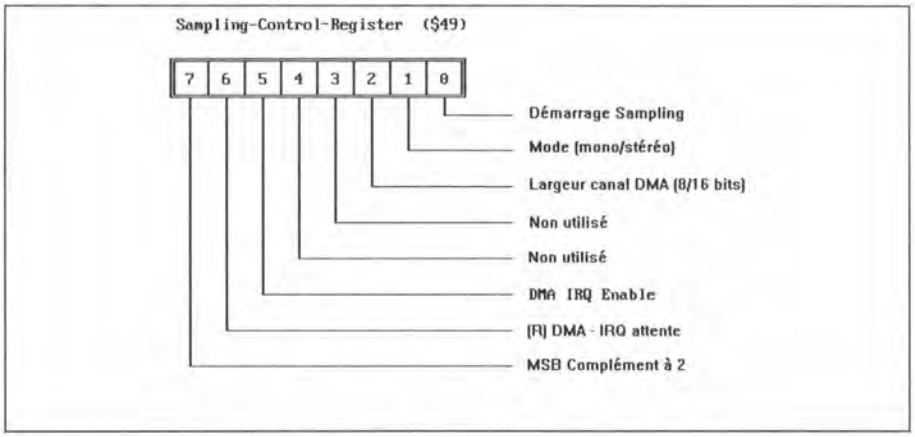
Registre de contrôle du Timer

Les registres 1 et 2 servant de compteurs pour les timers correspondent aux fonction $46 et $47. Vous devez fournir vous-même une valeur de départ pour les compteurs. Ils sont ensuite incrémentés jusqu'à la valeur $FFh. En atteignant cette valeur, ils génèrent une interruption. Le timer 1 est incrémenté toutes les 80 microsecondes, le timer 2 toutes les 320 microsecondes.

Avec le registre préposé à la fréquence de sampling ($48), vous pouvez spécifier la vitesse d'enregistrement de la carte. La valeur sur 8 bits se calcule d'après la formule :

```
SRQ:=9878400 DIV ((fréquence+2) SHL 4)
```

Le registre de contrôle du sampling ($49) joue un rôle important. Il permet de lancer l'enregistrement via le DMA.

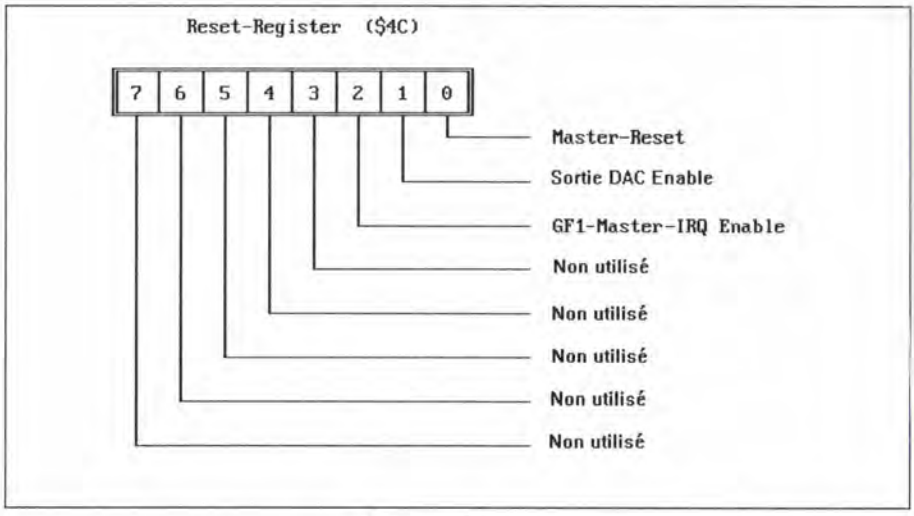


Registre de contrôle du sampling

Avec le bit 0, vous démarrez le sampling. Si vous avez programmé auparavant le contrôleur DMA, l'enregistrement commence directement une fois que vous avez posé ce bit sur 1. Par l'intermédiaire du bit de mode 1, vous indiquerez si l'enregistrement doit se faire en mono (=0) ou en stéréo (=1). Pour un enregistrement en stéréo, comme pour la carte SB, l'octet est d'abord déposé dans le canal de gauche, puis dans le canal de droite. Avec le bit 2, vous spécifiez la largeur du canal DMA. La valeur 0 représente un canal large de 8 bits. La valeur 1 indique un canal 16 bits. Si vous enregistrez en stéréo et si vous utilisez un canal 16 bits, notez que l'octet pour le canal de gauche est transféré en Low Word et celui pour le canal de droite en High Word. Si une interruption doit être déclenchée, vous pouvez l'activer par l'intermédiaire du bit 5. Lorsque vous lisez dans le registre, la valeur 1 dans le bit 6 indique qu'il y a une interruption en attente. Par l'intermédiaire du bit 7, enfin, vous pouvez activer l'inversion du bit Hi, déjà décrite précédemment.

La fonction $4B sert à régler la manette de jeu. Il est recommandé d'apporter le moins de changements possibles aux réglages disponibles. Nous ne mentionnons ceux-ci que par souci d'exhaustivité. Par l'intermédiaire de la valeur $29, le registre est réglé avec une valeur de 4,3 volt. Une valeur plus forte ou plus faible n'est nécessaire que si vous voulez utiliser un ordinateur très lent ou très rapide. Pour régler le registre, il est préférable d'utiliser l'utilitaire Ultrajoy fourni par Gravis.

Le registre de reset $4C figure lui aussi parmi les registres très importants.



Registre de reset

Le registre contient normalement la valeur $07. Par l'intermédiaire du bit 0, vous déclenchez un reset master de GF1. La valeur 1 représente ici le fonctionnement normal. La valeur 0 signifie un reset. L'état de reset reste actif

tant que la valeur n'est pas revenue à 1. Par l'intermédiaire du bit 1, vous activez la sortie du DAC. Si vous voulez donc arrêter toutes les voix dans votre programme, sans interrompre pour autant la sortie, il vous suffit de poser le bit 1 sur 0. Le bit 2 doit être posé sur 1 lorsque vous souhaitez utiliser les interruptions de la carte GUS.

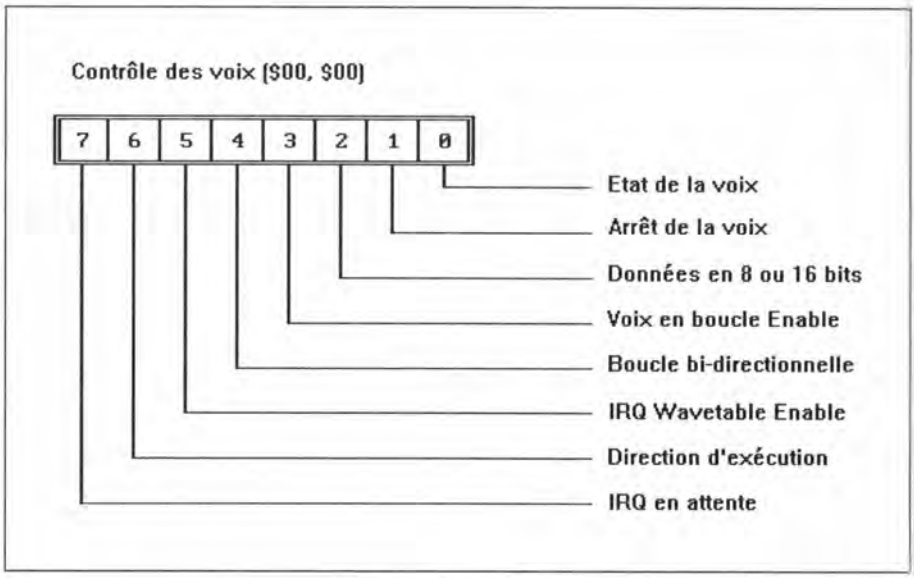
Maintenant que vous avez fait connaissance avec les fonctions indépendantes des voix, nous allons envisager les fonctions qui en dépendent. Voici d'abord un tableau des fonctions disponibles :

Ecrire	Lire	Largeur	Description
$00	$80	8	Contrôle des voix
$01	$81	16	Contrôle de la fréquence
$02	$82	16	Adresse de départ Hi
$03	$83	16	Adresse de départ Lo
$04	$84	16	Adresse finale Hi
$05	$85	16	Adresse finale Lo
$06	$86	8	Facteur Volume Ramp
$07	$87	8	Début Volume Ramp
$08	$88	8	Fin Volume Ramp
$09	$89	16	Volume actuel
$0A	$8A	16	Adresse actuelle Hi
$0B	$8B	16	Adresse actuelle Lo
$0C	$8C	8	Position Pan
$0D	$8D	8	Contrôle du volume
$0E	$8E	8	Voix actives (ne dépend pas des voix)
$0F	$8F	8	Source IRQ (ne dépend pas des voix)

Quelques-uns des registres mentionnés ici sont modifiés par GF1. L'architecture en pipeline de la puce GF1 a pour effet qu'une valeur entrée par vos soins peut fort bien être remplacée par GF1. Cela se produit toujours quand vous avez modifié votre registre entre le processus de lecture et celui d'écriture. Vous pouvez résoudre le problème en écrivant toutes les commandes à deux reprises et en attendant au moins trois cycles GF1 (4,8 microsecondes) entre les deux envois.

Une autre propriété de cette puce est que les registres de lecture des fonctions dépendant des voix se trouvent exactement à $80 octets au-delà des registres d'écriture. Si vous voulez donc lire un registre, il suffit d'additionner $80 à l'adresse d'écriture et vous pouvez lire le contenu.

Considérons d'abord la fonction $0/$80. Elle sert à contrôler les différentes voix. Les bits ont la signification suivante :



Le contrôle des voix

Les bits 0, 6, 7 sont modifiés par GF1. S'ils ont pour vous de l'importance, vous devrez donc réaliser un double accès. Le bit 0 indique si une voix est en cours d'exécution ou si elle a atteint sa position finale. La valeur 0 signifie que la voix est active. Avec le bit 1, vous pouvez arrêter la voix. C'est une propriété très importante, par exemple pour un MOD-Player. Le bit 2 indique si les données utilisées sont en 8 bits ou en 16 bits. La valeur 1 signale des données en 16 bits, la valeur 0 des données en 8 bits. Avec le bit 3, vous activez et désactivez le looping de la voix. La valeur 1 indique que le looping est actif. C'est très utile, parce que vous n'êtes plus obligés ainsi de vous occuper vous-mêmes d'un looping correct dans les routines d'exécution. Si vous posez le bit 4 sur 1, vous obtenez un looping bi-directionnel. Cela veut dire que la voix est exécutée d'abord vers l'avant, puis vers l'arrière, puis encore vers l'avant, et ainsi de suite. Dans un looping normal, la voix recommence au début quand elle est arrivée à la fin et elle est exécutée dans une seule direction. Par l'intermédiaire du bit 5, vous pouvez activer l'IRQ Wavetable. Lorsque la voix atteint sa position finale, une interruption est générée. L'interruption est d'ailleurs générée également quand le looping est actif. Le bit 6 indique la direction de l'exécution. La valeur 0 désigne la direction par adresses crois-

santes (exécution vers l'avant). La valeur 1 représente les adresses décroissantes. Enfin, le bit 7 indique qu'une IRQ est en attente. Quand la fin de la voix est atteinte, une interruption est générée en mode non-looping jusqu'à ce que la voix ait été arrêtée.

Avec le registre de contrôle des fréquences ($01/$81), vous indiquez à quelle vitesse la voix doit être exécutée. Dans les bits 15 à 10 se trouve la valeur qui précède la virgule pour le facteur d'avancement de la voix. Dans les bits 9 à 1 se trouvent les décimales. Le bit 0 n'est pas utilisé. Quand vous voudrez sortir une voix par exemple en 22 KHz, vous vous conformerez à la méthode suivante :

Déterminez le diviseur à partir du tableau suivant :

Voix	Diviseur	Voix	Diviseur
14	43	24	25
15	40	25	24
16	37	26	23
17	35	27	22
18	33	28	21
19	31	29	20
20	30	30	20
21	28	31	19
22	27	32	18
23	26		

Divisez la fréquence (22000) par le diviseur (par exemple 30 pour 20 voix). Sortez alors le résultat sur le port $3x5.

Avec les couples de registres ($02/$82) et ($03/$83), vous pouvez définir l'adresse de départ de la voix. Les bits 12 à 0 du registre $02/$82 représentent les 13 bits supérieurs de l'adresse dans la mémoire DRAM de la carte GUS (lignes d'adresse 19-7). Les bits 15 à 13 de ce registre ne sont pas utilisés. Les sept bits inférieurs de l'adresse de départ sont spécifiés par l'intermédiaire des bits 15 à 9 du couple de registres $03/$83. Ils correspondent donc aux lignes d'adresse 6 à 0. Les bits 8 à 5 du registre indiquent la position des chiffres décimaux dans l'adresse de départ. Les bits 4 à 0 ne sont pas utilisés.

L'adresse finale d'une voix sera spécifiée par l'intermédiaire des couples de registres $04/$84 et $05/$85. La disposition des bits est analogue à celle de l'adresse de départ.

La carte GUS supporte le sliding du volume par voie matérielle. A l'aide du couple de registres $06/$86, vous pouvez définir la vitesse de ramping du volume. Les six bits inférieurs désignent la taille de l'intervalle de réduction ou d'accroissement. La valeur 1 représente l'accroissement minimal du volume. La valeur 63 est la valeur maximale d'accroissement. Les deux bits supérieurs désignent la vitesse de ramping. Ses valeurs ont la signification suivante :

Bit 7 et 6	Facteur
00	1 DIV (1,6 * Voix actives)
01	8 DIV (1,6 * Voix actives)
10	64 DIV (1,6 * Voix actives)
11	512 DIV (1,6* Voix actives)

Le ramping le plus rapide sera obtenu avec une vitesse de 00b et un intervalle de 63. Le ramping le plus lent correspond à une vitesse de 11b et à un intervalle de 1.

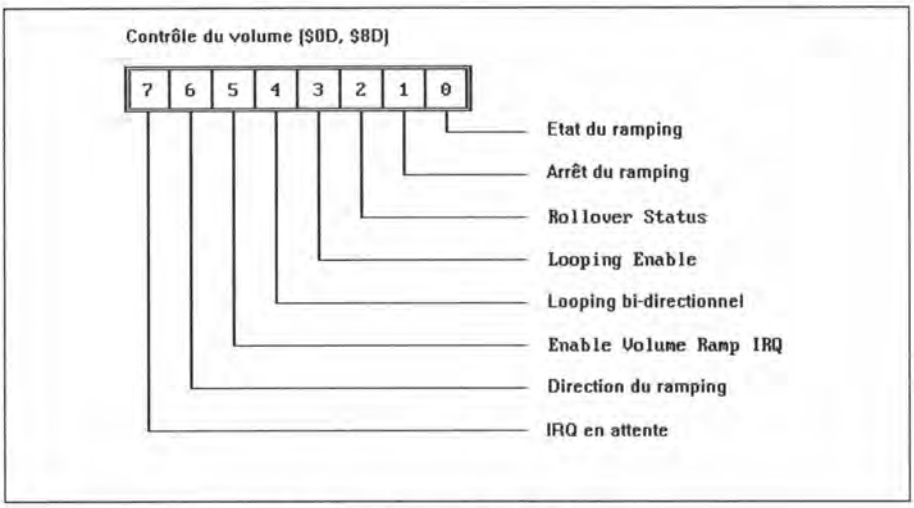
Par l'intermédiaire des couples de registres $07/$87 et $08/$88, vous pouvez spécifier le volume de départ et le volume de fin du ramping. Les bits 7 à 4 représentent les exposants de la valeur indiquant le volume. La base est constituée par les bits 3 à 0. Il faut barrer les quatre bits inférieurs du volume original pour parvenir à la valeur de ramping. Notez que le volume de départ doit toujours être inférieur au volume d'arrivée. Pour un "ramping down", vous devez poser le bit de direction sur le volume décroissant dans le registre de contrôle des volumes.

Le volume actuel sera spécifié par l'intermédiaire du couple de registre $09/$89. Les bits 15 à 12 contiennent ici l'exposant du volume, tandis que les bits 11 à 4 contiennent la base. La valeur étendue pour la base est nécessaire pour une exactitude plus grande lors du ramping. Les bits 3 à 0 du registre ne sont pas utilisés. Ce registre doit être posé de la même façon que le volume de départ du ramping.

Avec les couples de registres $0A/$8A et $0B/$8B, vous pouvez accéder à la position actuelle de la voix choisie. La disposition des bits correspond à celle du registre $02/$82, celui de l'adresse de départ.

La position "Pan" sera spécifiée par l'intermédiaire du registre $0C/$8C. Les bits 7 à 4 ne sont pas utilisés. Les bits 3 à 0 contiennent la position. La valeur nulle correspond à la position la plus éloignée vers la gauche. La valeur 15 correspond à la position la plus éloignée vers la droite.

Le couple de registres $0D/$8D a une grande importance pour le ramping du volume. C'est lui qui contrôle le volume.



Registre de contrôle du volume

Les bits 0, 6 et 7 sont modifiés par la puce GF1. Ici encore, vous devez donc utiliser un double accès. Le bit 0 indique que le ramping est terminé. Avec le bit 1, vous pouvez arrêter le ramping. Si vous le posez sur 1, le ramping cesse aussitôt. Le bit 2 sert à reconnaître le "rollover". Vous verrez un peu plus bas de quoi il s'agit. Par l'intermédiaire du bit 3, vous pouvez activer (=1) ou désactiver (=0) la boucle du ramping. Le bit 4 indique si la boucle doit être parcourue dans une seule direction (=0) ou dans les deux directions. En posant le bit 5, vous déclenchez une interruption lorsque le ramping atteint son terme. Le bit 6 indique la direction du ramping. La valeur 0 signifie une augmentation de volume, la valeur 1 une diminution de volume. Le bit 7 indique si une IRQ de ramping est en attente.

Le registre $0E/$8E dépend des voix. Il permet de définir le nombre de voix actives. Les bits 7 et 5 doivent toujours contenir la valeur 1. Les bits 4 à 0 contiennent le nombre de voix utilisées moins 1. Notez que le nombre minimal de voix est égal à 14.

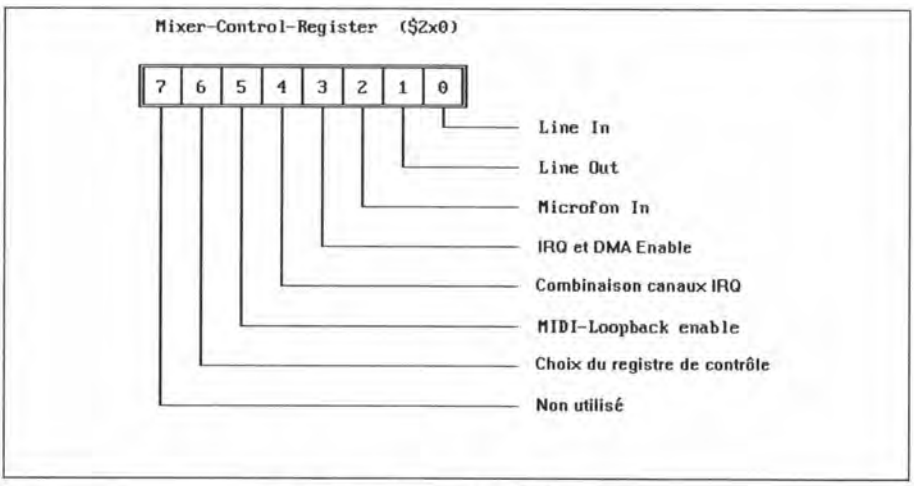
Par l'intermédiaire du registre source IRQ, vous pouvez indiquer par quoi une IRQ Wavetable a été déclenchée. Les bits 4 à 0 contiennent le numéro de la voix qui déclenche l'interruption moins 1. Le bit 5 porte toujours la valeur 1. Lorsque le bit 6 a la valeur 0, cela veut dire qu'une IRQ Volume-Ramp est en attente. Une valeur 0 dans le bit 7 indique qu'une IRQ Wavetable est en attente. Il faut que ce registre perde ses valeurs une fois qu'il a été lu.

Vous devez traiter dans votre handler d'interruption toutes les IRQ indiquées, donc le volume-ramp et le wavetable des voix concernées. Puisqu'il est possible théoriquement que deux voix déclenchent une interruption en même temps, vous devez traiter le registre jusqu'à ce que les bits 6 et 7 portent la valeur 1. Puisqu'une voix pourrait à nouveau déclencher une interruption pendant que vous en êtes encore au traitement du handler d'interruption, vous devez ignorer toutes les interruptions suivantes d'une voix déjà traitée.

Après ces longues explications sur le registre $3x3, nous allons en venir aux registres moins complexes de la carte GUS. Le registre de données de la Gravis se trouve aux ports $3x4 et $3x5. Le port $3x4 est large de 16 bits. Il sert à recevoir une valeur sur 16 bits, pour l'indication des positions dans les voix. Mais vous pouvez aussi sortir la partie inférieure d'une valeur sur 16 bits quand vous utilisez une instruction en 8 bits. Par l'intermédiaire du port $3x5, vous exécuterez tous les transferts 8 bits (les transferts de données vers un registre 8 bits de la carte GUS). De plus, le registre sert à accéder en écriture ou en lecture à l'octet High d'une valeur 16 en bits envoyée par l'intermédiaire des ports $3x4/$3x5.

Le registre $3x7 de l'Ultrasound est le registre d'entrée/sortie de la DRAM. Par l'intermédiaire de ce registre, vous pouvez écrire ou lire directement dans la DRAM de la carte. Notez toutefois que vous devez auparavant spécifier l'adresse de transfert dans la DRAM, par l'intermédiaire des fonctions $43 et $44.

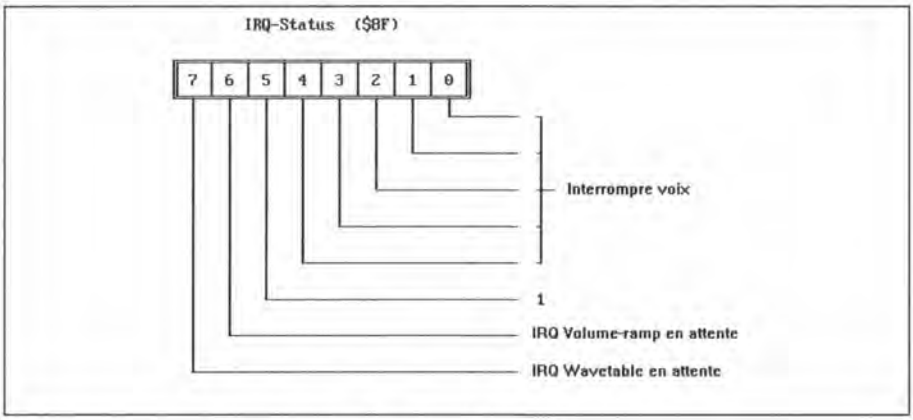
Vous venez de faire connaissance avec les registres 3x? de la carte GUS. Nous allons maintenant passer aux registres 2x?. Le plus élémentaire se trouve à l'adresse $2x0, adresse de base de la carte. C'est le registre de contrôle du mixer. Sa structure est la suivante :



Registre de contrôle du Mixer

En posant le bit 0 sur 0, vous spécifiez le signal line-IN de la carte. De même, le fait de poser le bit 1 sur 0 active le signal line-OUT. Si vous voulez activer en plus le microphone de la carte GUS, vous devez poser le bit 2 sur 1. Le bit 3 est très important. Par l'intermédiaire de ce bit, vous activerez les interruptions de la carte et les canaux DMA. Si vous les avez activés une fois, vous ne devez plus les désactiver, sans quoi vous risquez d'obtenir des interruptions arbitraires. Le bit 4 indique si le canal d'interruption 1, responsable de la GF1, doit être combiné avec le canal d'interruption 2 (MIDI). Par l'intermédiaire du bit 5, vous activerez le loopback MIDI. Cela veut dire que toutes les données MIDI sortantes seront envoyées encore une fois au port d'entrée MIDI. Le bit 6 représente une sorte de protection contre les applications qui testent les cartes de manière sauvage. Lorsque le bit est sur 0, l'accès suivant au port $2xB modifie les latches DMA. Lorsque le bit est sur 1, ce sont les latches IRQ qui sont modifiés. L'accès suivant doit être un accès au port $2xB. Sinon, l'accès aux latches est ignoré. Cela empêche que les latches puissent être modifiés involontairement par les valeurs de test d'autres programmes.

Le registre d'état IRQ $3x6 est en étroite relation avec le registre source IRQ. Par l'intermédiaire de ce registre, vous pouvez déterminer en effet quelle est l'unité de l'Ultrasound qui a déclenché l'interruption.

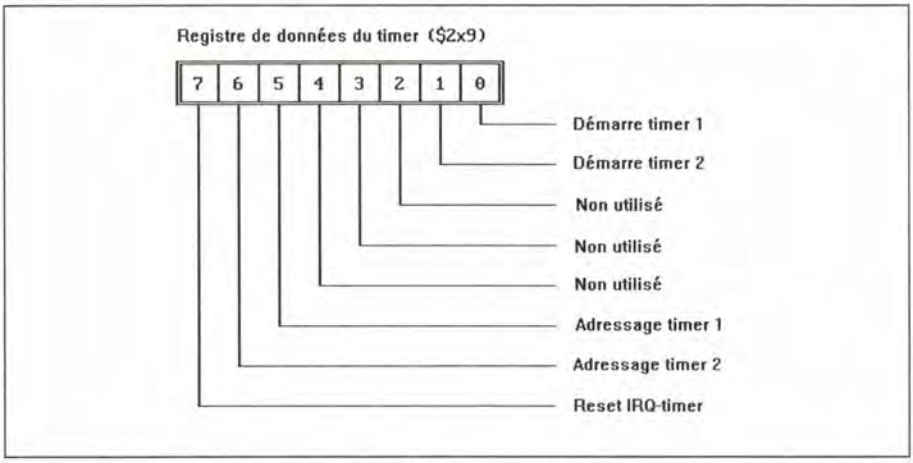


Le Registre d'état IRQ

L'unité qui a causé l'interruption sur l'Ultrasound sera déterminée en fonction du bit posé dans le registre. Le bit 0 représente l'envoi de données MIDI, le bit 1 la réception de données MIDI. Si le bit 2 est posé, l'interruption a été déclenchée par le timer 1. Si c'est le bit 3, cela veut dire que la cause de l'interruption est le timer 2. Le bit 4 n'est pas utilisé. Le bit 5 signale une interruption Wavetable et le bit 6 une interruption Volume-Ramp. Notez que vous devez déterminer dans les deux cas quelle est la voix qui a déclenché l'interruption. Le bit 7 est posé lorsque l'interruption a été déclenchée par la fin d'un transfert DMA.

Le registre $2x8 est le registre de contrôle du timer compatible avec Adlib. En envoyant la valeur 4, vous pouvez activer le timer. Le bit 6 est posé lorsque le timer 1 a terminé son décompte. Si c'est le timer 2, le bit posé est le 5.

Le registre $2x9 est le registre de données du timer. Il permet de contrôler le timer propre à la carte GUS.



Le registre de données du timer

Par l'intermédiaire des bits 0 et 1, vous lancez les timers 1 et 2. Si vous voulez "démasquer" un timer, vous devez poser le registre de masque qui lui correspond. Le bit 5 permet d'adresser le timer 2, le bit 6 d'adresser le timer 1. Vous aboutissez à un reset du timer en posant le bit 7.

Le dernier registre est le registre de contrôle IRQ/DMA à l'adresse $2xB. Lorsque le bit 6 du registre 2x0 est posé sur 1, on adresse le registre IRQ. Dans les bits 0 à 2 se trouve l'IRQ pour la puce GF1 (canal 1). Les valeurs ont les significations suivantes :

0	Pas d'interrupt
1	IRQ 2
2	IRQ 5
3	IRQ 3
4	IRQ 7
5	IRQ 11
6	IRQ 12
7	IRQ 15

Par l'intermédiaire des bits 3 à 5, vous pouvez choisir l'IRQ pour le canal MIDI (canal 2). La signification des valeurs est la suivante :

0	Pas d'interrupt
1	IRQ 2
2	IRQ 5
3	IRQ 3
4	IRQ 7
5	IRQ 11
6	IRQ 12
7	IRQ 15

Quand le bit 6 est posé sur 1, cela signifie que les deux IRQ sont combinés, donc utilisent le même canal. Dans ce cas, c'est l'IRQ de la puce GF1 (canal 1) qui est utilisée. Le second canal doit alors être posé sur 1, car sinon il se produirait un conflit de bus. Le bit 7 n'est pas utilisé. Si vous modifiez les spécifications pour les IRQ, cela conduit normalement à une interruption sur l'ancien canal. Cela est dû au fait que les interruptions des latches ne sont plus contrôlées. Ce phénomène ne devrait cependant poser aucun problème.

Lorsque le bit 6 du port 2x0 présente la valeur 0, le registre de contrôle DMA est en cours d'adressage. Par l'intermédiaire des bits 0 à 2 de ce registre, vous déterminez le premier canal DMA. Les valeurs possibles sont les suivantes :

0	Pas de canal DMA
1	Canal DMA 1
2	Canal DMA 3
3	Canal DMA 5
4	Canal DMA 6
5	Canal DMA 7
6,7	non autorisés

Par l'intermédiaire des bits 3 à 5, vous choisissez le second canal DMA. Ici aussi, voici les valeurs possibles :

0	Pas de canal DMA
1	Canal DMA 1
2	Canal DMA 3
3	Canal DMA 5
4	Canal DMA 6
5	Canal DMA 7
6,7	non autorisés

Un bit 6 posé indique que GF1 et MIDI utilisent le même canal. Lorsque les deux passent par le même canal DMA, il porte la valeur 0 dans le second canal DMA. Sinon, on voit surgir des conflits de bus. Le bit 7 n'est pas utilisé.

14.4. PROGRAMMATION DE LA CARTE GUS

Vous avez appris tout ce qu'il faut savoir sur la partie matérielle de la carte GUS. Nous allons maintenant passer à la programmation de la carte. Bien que les fonctions de base de la carte soient très faciles à programmer, la carte possède cependant ses particularités. Ainsi ne vous étonnez pas en voyant que vos routines timer ne fonctionnent pas lorsque vous n'avez pas activé les interruptions globales. Mais n'ayez pas peur. Nous allons étudier ensemble les routines essentielles. Plus loin, au paragraphe 15.3, vous trouverez un autre exemple d'application pour votre Gravis : un Player MOD, que vous pourrez intégrer sans problème dans vos programmes.

Comment on initialise la carte GUS

Les routines de base présentées ici pour la GUS sont toutes écrites en assembleur. Cela ne devrait donc présenter aucun problème pour vous de les faire fonctionner avec n'importe quel langage courant. Une réalisation des routines en assembleur est chose raisonnable, car vous n'avez un contrôle total de la machine qu'au niveau de l'assembleur.

Commençons par la routine la plus fondamentale, qui ne peut être absente d'aucun programme GUS : l'initialisation de la carte. Comment élaborer une procédure d'initialisation pour la Gravis ?

Nous nous servirons d'abord pour cela de la fonction reset de la carte. Mais ce n'est pas tout. Le registre de contrôle DMA, le registre de contrôle du timer et le registre de contrôle du sampling doivent subir un reset "à la main". De plus, vous devez définir le nombre de voix utilisées, poser les paramètres des différents canaux et régler le volume des voix dans la procédure d'initialisation. La procédure s'allonge certes un peu, mais vous évitez ainsi les effets secondaires indésirables, comme un crissement soudain et indéfinissable venant d'un côté ou de l'autre.

Une initialisation de la carte au niveau du hardware sera obtenue en choisissant la fonction d'initialisation par l'intermédiaire du registre de commandes et en lui donnant la valeur 0. Attendez quelques cycles GF1 et replacez le registre sur 1, sans quoi la carte reste en état de reset et ne peut pas être adressée correctement.

L'opération suivante consiste à effectuer le reset des registres de contrôle DMA, timer et sampling. En pratique, les trois registres possèdent un état par défaut, défini par la valeur 0. Il suffit donc de les appeler par l'intermédiaire des registres de commandes et d'écrire la valeur 0 en eux. Par l'intermédiaire de la fonction 0Eh du registre de commandes, choisissez alors le registre des voix. Notez bien que vous devez poser les deux bits supérieurs de l'octet avec le nombre des voix sur 1 ! La façon la plus simple d'y parvenir est de charger l'octet dans l'un des registres et de lui donner alors la valeur nécessaire avec un OR 0C0h.

Si vous avez déjà installé votre handler d'interruption (ce n'est pas nécessaire pour le fonctionnement correct de la carte GUS), vous devez aussi vous occuper des interruptions pendant le reset. Videz éventuellement les interruptions DMA concernées en lisant d'abord le registre d'état, puis le registre de contrôle DMA. A l'aide d'un accès en lecture au registre de contrôle du sampling, supprimez ensuite les interruptions Wavetable rencontrées. Ces interruptions peuvent simplement être sautées, car elles n'ont de toute façon aucune signification pour nous. En dernier lieu, effectuez un reset de la carte !

L'étape suivante consiste à effectuer dans une boucle le reset de tous les canaux. Il est utile, afin d'être complet, d'effectuer non seulement le reset des canaux concernés, mais en fait celui des 32 canaux en leur totalité, ne serait-ce que pour éviter les effets de bord toujours possibles. Choisissez pour cela dans la boucle tout d'abord la voix à modifier et posez alors sur 3 le mode de la voix. Vous arrêtez ainsi les éventuelles sorties des voix. Ensuite, placez le volume sur 0 à l'aide du registre des volumes. Vous pouvez également initialiser les données loop de la voix, même si ce n'est pas indispensable. En effet, vous devez de toute façon les définir avant de faire entrer la voix en action.

Une fois que les voix ont été initialisées dans la boucle, il est conseillé de modifier les interruptions qui pourraient avoir surgi. La dernière étape

consistera alors à exécuter un reset au niveau matériel par l'intermédiaire du registre de reset et à activer dans le registre master GF1 les interruptions en même temps que les sorties des samples.

Vous trouverez un exemple de la fonction de reset dans la procédure en assembleur intitulée U_Initialize.

```
u_Initialize proc near
; ********************************************************************
; ***              Initialise la carte Ultrasound                  ***
; ********************************************************************
  mov bx,w u_Command
  mov cx,w u_datahi
  mov dx,bx
  mov al,4ch                    ; Choisir le registre d'initialisation
  out dx,al
  mov dx,cx
  mov al,0                      ; Exécuter l'initialisation
  out dx,al
  call u_delay                  ; Attendre
  call u_delay
  mov dx,bx
  mov al,4ch
  out dx,al
  mov dx,cx
  mov al,1                      ; Terminer l'initialisation
  out dx,al
  call u_delay
  call u_delay
  mov dx,bx                     ; Reset du DMA Control Register
  mov al,41h
  out dx,al
  mov dx,cx
  mov al,0
  out dx,al
  mov dx,bx                     ; Reset du Timer Control Register
  mov al,45h
  out dx,al
  mov dx,cx
  mov al,0
```

```
  out dx,al
  mov dx,bx                    ; Reset du Sampling Control Register
  mov al,49h
  out dx,al
  mov dx,cx
  mov al,0
  out dx,al
  mov dx,bx                    ; Poser le nombre de voix
  mov al,0Eh
  out dx,al
  add dx,2
  mov al,Num_Voices
  or  al,0C0h
  out dx,al
  mov dx,w u_status            ; Vider éventuellement les interrupts
                               ; DMA
  in al,dx
  mov dx,bx
  mov al,41h
  out dx,al
  mov dx,cx
  in al,dx
  mov dx,bx                    ; Vider éventuellement les interrupts
                               ; de sampling
  mov al,49h
  out dx,al
  mov dx,cx
  in al,dx
  mov dx,bx                    ; Lire le registre d'état de l'IRQ
  mov al,8Fh                   ; ==> Il n'y a plus d'interruptions
   out dx,al                   ;     non traitées
  mov dx,cx
  in al,dx
  push bx                      ; Désactive les voix dans la boucle
  push cx
  mov cx,0
@VoiceClearLoop:
  mov dx,w u_Voice             ; Choisir les voix
```

```
mov al,cl
out dx,al
inc dx
mov al,0                    ; Poser le mode Voice
out dx,al
add dx,2
mov al,3                    ; Arrêter les voix
out dx,al
sub dx,2                    ; Poser le volume sur 0
mov al,0dh
out dx,al
add dx,2
mov al,3
out dx,al
inc cx
cmp cx,32                   ; Répéter pour toutes les voix
jnz @VoiceClearLoop
pop cx
pop bx
mov dx,bx                   ; Traiter les interruptions
mov al,41h                  ; éventuelles
out dx,al
mov dx,cx
in al,dx
mov dx,bx
mov al,49h
out dx,al
mov dx,cx
in al,dx
mov dx,bx
mov al,8fh
out dx,al
mov dx,cx
in al,dx
mov dx,bx                   ; Exécuter un reset
mov al,4ch
out dx,al
mov dx,cx                   ; Activer l'IRQ master GF1
```

```
    mov al,7
    out dx,al
    ret
u_Initialize endp
```

Avant de pouvoir exécuter un reset, vous devez cependant savoir à quel port se trouve la Gravis. Il est possible de demander à l'utilisateur qu'il entre lui-même les paramètres pour la carte GUS. Cette possibilité est la plus intéressante pour le programmeur, puisque c'est la solution la plus simple. Pourtant, cette méthode n'est pas des plus conviviales et elle ne ferait pas preuve d'un grand savoir-faire. Il faut plutôt essayer de mettre en place une routine de test facile à réaliser pour reconnaître le port de base de la Gravis.

Si vous voulez créer une routine de test de ce genre, vous devrez procéder comme suit : testez dans une boucle par étapes de 10h tous les ports possibles entre 200h et 280h. Initialisez la carte, écrivez les données dans la RAM de celle-ci et lisez à nouveau ces données. Si elles concordent, vous avez trouvé le port de base de la carte GUS. Sinon, vous devez essayer le port suivant, à moins qu'il n'y ait tout simplement pas de carte GUS connectée.

Vous trouverez un exemple de routine de détection dans la procédure Detect_Gus. La procédure initialise en premier la carte par l'intermédiaire du registre Init. Un octet est alors écrit dans la mémoire DRAM de la carte GUS au moyen de la fonction Poke. Il n'est évidemment conservé dans la DRAM de la carte que si l'on a choisi le port de base correct. Sinon, la commande "out" fonctionne en vain pour accéder à la mémoire. Attendez ensuite un certain temps pour être tout à fait sûr que la puce GF1 n'est pas en activité. La dernière étape consistera alors à lire en retour l'octet dans la RAM de la carte au moyen de la fonction Peek.

Lorsque la valeur lue en retour coïncide avec la valeur écrite, cela veut dire que vous avez trouvé le port de base de la carte GUS et vous pouvez quitter la routine Detect_loop. Sinon, vous devez répéter l'ensemble pour le port suivant. N'oubliez pas de quitter la routine de même après le test pour le port $280. Dans ce cas, cela signifie que vous n'aviez pas installé de GUS dans votre ordinateur et vous n'avez plus besoin dans ce cas d'appliquer le test à d'autres ports.

Voici maintenant le code de la routine que nous venons de décrire.

```
detect_gus proc near
; ********************************************************************
; ***     La routine sert à reconnaître la Gravis Ultrasound.      ***
; ***     Le port de base                                          ***
; ***     est reconnu. La procédure retourne 0 quand la carte      ***
; ***     a été trouvée,sinon la valeur 1                          ***
; ********************************************************************
  mov di,1F0h
@detect_loop:                   ; Test des ports possibles dans une
                                ; boucle
  add di,10h
  mov dx,di
  add dx,103h                   ; Tente d'initialiser
  mov al,4Ch
  out dx,al
  mov dx,di
  add dx,105h
  mov al,0
  call u_delay
  call u_delay
  mov dx,di
  add dx,103h
  mov al,4Ch
  out dx,al
  mov dx,di
  add dx,105h
  mov al,0
  mov dx,di                     ; Tentative d'écrire des données
  add dx,103h                   ; dans la RAM de la carte
  mov al,43h
  out dx,al
  mov dx,di
  add dx,105h
  mov al,0h
  out dx,al
  mov dx,di
  add dx,103h
```

```
  mov al,44h
  out dx,al
  mov dx,di
  add dx,105h
  mov al,0h
  out dx,al
  mov dx,di
  add dx,107h
  mov al,0AAh
  out dx,al
  call u_delay              ; Attente, pour que la GF1
  call u_delay              ; ne puisse pas nous échapper
  call u_delay
  call u_delay
  call u_delay
  call u_delay
  xor ax,ax                 ; Lecture en retour dans la RAM de la
                            ; carte
  mov dx,di
  add dx,107h
  in  al,dx
  cmp al,0AAh               ; Valeur écrite = Valeur lue ?
  je  @Carte_trouvee        ; Carte trouvée !
  cmp di,280h
  jae @Carte_non_trouvee    ; Pas de carte à ce port :
  jmp @detect_loop          ; Tentative avec un nouveau port
@Carte_trouvee:
  mov w u_base,di           ; Initialiser le registre de base
                            ; de la carte
  mov ax,di
  add ax,6
  mov w u_status,ax
  mov ax,di
  add ax,102h
  mov w u_voice,ax
  mov ax,di
  add ax,103h
  mov w u_command,ax
```

```
  mov ax,di
  add ax,104h
  mov w u_Datalo,ax
  mov ax,di
  add ax,105h
  mov w u_Datahi,ax
  mov ax,di
  add ax,107h
  mov w u_DramIO,ax
  mov ax,0
  jmp @Fin_Recherche
@Carte_non_trouvee:
  mov ax,1
@Fin_Recherche:
  ret
detect_gus endp
```

La sous-procédure d'attente se présentera ainsi :

```
u_delay proc pascal
; ************************************************************************
; ***      Attendre le temps qu'il faut pour un Double-write      ***
; ************************************************************************
  mov dx,300h
  in al,dx
  in al,dx
  in al,dx
  in al,dx
  in al,dx
  in al,dx
  in al,dx
  ret
u_delay endp
```

Charger des données son

Le chargement de données dans la RAM de la carte GUS se fait très simplement. On copie au moyen d'une boucle les données de la mémoire principale dans la DRAM de la carte. Pour arrêter rapidement la routine de chargement, il faut essayer de l'écrire avec aussi peu de commandes "out" que possible dans chaque parcours de la boucle.

Vous trouverez un exemple de cette routine de chargement dans Ultra_Mem2Gus. Vous devrez transmettre à cette routine un pointeur sur le secteur source dans la RAM, la position de départ dans la RAM de la carte et enfin la longueur du bloc de données à copier.

La routine charge d'abord le pointeur sur le secteur source et définit la position de départ High dans la RAM de la carte GUS. Cette position se modifie non pas à chaque parcours, mais toutes les 65536 fois. C'est pourquoi il est important de ne pas redéfinir cette valeur à chaque parcours. Vous devez ensuite charger la longueur, puis parcourir la boucle de copie proprement dite (Copy-Loop) "long" fois. Dans la boucle, vous définissez en premier lieu la position de départ Low dans la DRAM de la carte GUS, vous chargez alors l'octet provenant du secteur source et vous écrivez cet octet dans la mémoire de la carte par l'intermédiaire de la fonction Poke. Puisque l'adresse de départ High n'est pas posée à chaque parcours, vous devez encore vérifier si la position Low a atteint la valeur 0FFFFh. Dans ce cas, il se produit un dépassement pour l'octet suivant et vous devez à nouveau définir la position High.

Voici comment se présentera en pratique une routine de copie :

```
Ultra_Mem2Gus proc pascal sampp:dword,start:dword,long:word
; ********************************************************************
; ***       Copie un secteur de RAM dans la RAM de la carte        ***
; ********************************************************************
    push ds
    push si
    mov si,[bp+12]              ;Segment
    mov ds,si
    mov si,[bp+10]              ; Offset
    mov dx,w u_Command          ; Poser l'octet Hi de l'adresse de
                                ; GUS-DRAM
    mov al,44h
    out dx,al
    mov dx,w u_DataHi
    mov ax,[bp+08]              ; hstart
    out dx,al
    mov cx,[bp+4]               ; Charger la longueur
@Copy_loop:
    mov dx,w u_Command          ; Poser l'octet Lo de l'adresse de
                                ; GUS-DRAM
    mov al,43h
```

```
    out dx,al
    mov dx,w u_DataLo
    mov ax,[bp+06]              ; lstart
    out dx,ax
    mov dx,w u_DramIo           ; Charger et envoyer l'octet
    lodsb
    out dx,al
    cmp word ptr [bp+06],0ffffh  ; lstart = 0ffffh ?
    je @depassement
    inc word ptr [bp+06]        ; lstart++
    jmp @continuer
@depassement:
    inc word ptr [bp+08]        ; hstart ++
    mov word ptr [bp+06],0      ; lstart sur 0
    mov dx,w u_Command          ; Poser l'adresse hi de l'adresse de
                                ; la RAM GUS
    mov al,44h
    out dx,al
    mov dx,w u_DataHi
    mov ax,[bp+08]              ; hstart
    out dx,al
@continuer:
    loop @copy_loop
    pop si
    pop ds
    ret
Ultra_Mem2Gus endp
```

Exécuter les samples

Nous venons de charger notre sample dans la mémoire DRAM de la carte GUS. Nous voulons maintenant l'exécuter. Avant de lancer la sortie sur un canal, vous devez définir les paramètres spécifiques au canal en question. Il faut tout d'abord choisir le volume. Vous le ferez avec la commande de volume dans le registre de commandes. Voici comment pourra se présenter une procédure servant à régler le volume. La procédure s'appelle U_Voice-Volume.

```
u_VoiceVolume proc pascal Nr:byte,Vol:word
; ********************************************************************
; ***          Règle le volume pour un canal (0 - 63)              ***
; ********************************************************************
    mov dx,w u_Voice          ; Choisir la voix
    mov al,Nr
    out dx,al
    mov dx,w u_Command        ; Commande de définition du volume
    mov al,9
    out dx,al
    mov dx,w u_DataLo         ; Charge le volume GUS dans la table
    mov di,vol                ; et le définit
    shl di,1
    mov ax,word ptr [offset uVolumes + di]
    out dx,ax
    ret
u_VoiceVolume  endp
```

L'étape suivante consiste à régler les paramètres généraux pour chacune des voix. Ces paramètres sont les suivants : le début de la voix dans la DRAM de la carte GUS, la position de départ de la boucle et la position finale de la voix. Vous définirez le début de la voix par l'intermédiaire de la combinaison des commandes 0Ah/0Bh dans le registre de commandes. Si vous ne voulez pas faire entrer la voix dans une boucle, le mieux est de définir le départ de la boucle (loop) sur la position du début de la voix. Sinon, vous reporterez ici la position correspondante (et non un offset) dans la mémoire de la carte GUS. Cette position sera comptée à partir du début de la voix. La combinaison servant à définir le registre de départ du loop est 02h/03h. Pour définir la position finale, vous utiliserez 04h/05h. La procédure U_Voicedata vous indique comment initialiser correctement les registres.

```
u_Voicedata proc pascal start,lsta,llong:dword,Nr:word
; ********************************************************************
; ***            Définit les paramètre pour un canal               ***
; ********************************************************************
    mov dx,w u_Voice          ; Choisir la voix
    mov ax,Nr
    out dx,al
    mov dx,w u_command        ; Définir le début de la voix
    mov al,0ah
    out dx,al
```

```
mov ax, word ptr [start+2]
mov cx, word ptr [start]
mov bx,cx
shr ax,7
shr cx,7
shl bx,9
or  ax,bx
mov dx,w u_DataLo
out dx,ax
mov dx,w u_Command
mov al,0bh
out dx,al
mov dx,w u_datalo
mov ax,word ptr [start]
shl ax,9
out dx,ax
mov dx,w u_command          ; Définir le début d'une boucle
mov al,2
out dx,al
mov ax, word ptr [lsta]
mov cx, word ptr [lsta+2]
mov bx,cx
shr ax,7
shr cx,7
shl bx,9
or  ax,bx
mov dx,w u_DataLo
out dx,ax
mov dx,w u_Command
mov al,3
out dx,al
mov dx,w u_datalo
mov ax,word ptr [lsta]
shl ax,9
out dx,ax
mov dx,w u_command          ; Définir la fin de la boucle
mov al,4
out dx,al
```

```
    mov ax, word ptr [llong]
    mov cx, word ptr [llong+2]
    mov bx,cx
    shr ax,7
    shr cx,7
    shl bx,9
    or  ax,bx
    mov dx,w u_DataLo
    out dx,ax
    mov dx,w u_Command
    mov al,5
    out dx,al
    mov dx,w u_datalo
    mov ax,word ptr [llong]
    shl ax,9
    out dx,ax
   ret
u_Voicedata endp
```

La carte GUS sait déjà avec quel volume la voix doit être exécutée et où se trouvent les données du sample dans la DRAM de la carte GUS. Il vous manque toutefois encore l'information la plus importante : la fréquence avec laquelle la voix doit être exécutée. Vous pouvez spécifier celle-ci au moyen de la commande 1 dans le registre de commandes. Notez toutefois que l'on ne transmet pas la fréquence de sortie, mais la valeur calculée selon la formule :

```
Freq:=Freq DIV Voice_Divisor[num_voices]
```

Pourquoi en va-t-il ainsi ? Dieu seul le sait, et peut-être aussi Forte et Gravis... La procédure U_Voicefreq vous montrera comment spécifier correctement la voix. La procédure fournit d'abord le numéro de la voix à modifier. Elle fait savoir ensuite à la puce GF1 que la fréquence de la voix spécifiée doit être modifiée. L'opération suivante consiste à déterminer la position à l'intérieur de la table des diviseurs de voix. Cette position dépend du nombre de canaux utilisés. La dernière étape charge le diviseur voulu, divise la fréquence en conséquence et sort le résultat.

```
u_Voicefreq proc pascal Nr:byte,Freq:word
; ********************************************************************
; ***     Règle la fréquence à laquelle le canal sera exécuté      ***
; ********************************************************************
  mov dx,w u_Voice              ; Adresser la voix
  mov al,Nr
  out dx,al
  mov dx,w u_Command            ; Ecrire la commande pour la fréquence
  mov al,1
  out dx,al                     ; Freq := Fréquence DIV
  xor bx,bx                     ; Voice_Divisor [num_voices-13]
  mov bl,num_voices
  mov ax,Freq
  mov di,bx
  sub di,14
  xor bx,bx
  xor dx,dx
  mov bl,byte ptr [voice_Divisor+di]
  div bx
  mov dx,w u_DataLo
  out dx,ax
  ret
u_Voicefreq endp
```

Une option que vous n'êtes pas obligé d'utiliser, mais qui nous paraît cependant digne d'être mentionnée, est la suivante : vous pouvez spécifier la balance pour chaque voix à part. Vous le ferez à l'aide de la commande 0Ch du registre de commandes. Les valeurs qui interviennent pour la sortie se situent entre 0 et 15, où 0 correspond à la position tout à fait à gauche et 15 tout à fait à droite.

```
u_VoiceBalance proc pascal Nr,balance : byte
; ********************************************************************
; ***        Règle la position Pan pour un canal (0 - 15)          ***
; ********************************************************************
  mov dx,w u_Voice              ; Choisir la voix
  mov al,byte ptr Nr
  out dx,al
  mov dx,w u_Command            ; Commande Set Pan-Position
  mov al,0Ch
```

```
  out dx,al
  mov dx,w u_dataHi          ; Ecrire la position
  mov al,balance
  out dx,al
  ret
u_VoiceBalance endp
```

La dernière étape consiste à lancer la sortie de la voix. Vous le ferez à l'aide du registre de contrôle des voix dans le registre de commandes. La signification des différents bits a été donnée dans le paragraphe qui portait sur le registre de contrôle des voix. La procédure U_StartVoice prend en charge le démarrage de la voix. Notez que la sortie démarre dès que vous avez modifié le registre. Si vous voulez arrêter une voix manuellement, vous devez poser le bit 1 de ce registre sur 1.

```
U_StartVoice proc pascal Nr,Mode : byte
; ********************************************************************
; ***              Lance la sortie sur un canal GUS                ***
; ********************************************************************
  mov dx,w u_voice           ; Choisir la voix
  mov al,byte ptr Nr
  out dx,al
  mov dx,w u_command
  mov al,0                   ; Mode de la voix
  out dx,al
  mov dx,w u_DataHi
  mov al,Mode                ; Poser l'octet MODE
  out dx,al
  ret
U_StartVoice endp
```

Déformer des voix (Effets Ramping)

Pour obtenir une sortie propre du son, il est préférable de régler le volume d'une voix non pas au moyen de la procédure U_VoiceVolume, mais avec la procédure que nous allons vous présenter ici sous le nom de Voice_Rampin. La procédure ne place pas la voix immédiatement au volume défini : elle commence avec un volume égal à 0 et augmente peu à peu, jusqu'à ce que le volume cible soit atteint. L'avantage de cette méthode est d'empêcher tout grincement ou craquement, tels qu'on les entend parfois au début d'une voix à cause du changement soudain de la fréquence.

Dans cette méthode, la procédure choisit tout d'abord la voix à modifier. Elle définit ensuite le facteur de ramping par l'intermédiaire de la commande "volume ramp rate" du registre de commandes, en la plaçant sur le "ramping" le plus rapide possible. Elle s'occupe ensuite du volume en cours, ce qui veut dire qu'elle le place sur la valeur de départ 0. Le volume de départ du ramping est lui aussi placé sur 0. Le volume cible est déterminé comme dans la procédure U_VoiceVolume et initialisé par l'intermédiaire de la commande "volume ramp end" dans le registre de commandes. La dernière opération importante consiste à spécifier la direction du ramping dans le registre de contrôle des volumes. Cela fait démarrer le ramping.

```
voice_rampin proc pascal voix:byte,vol : word;
; ********************************************************************
; ***    Un moyen de définir directement le volume d'une voix.     ***
; ***    Le player perd un peu de son agressivité, mais on réduit ***
; ***    ainsi les craquements éventuels.                          ***
; ********************************************************************
  mov dx,w u_voice              ; Choisir la voix
  mov al,byte ptr voix
  out dx,al
  mov dx,w u_command            ; Définit le facteur de Ramping
  mov al,6
  out dx,al
  mov dx,w u_datahi
  mov al,00111111b
  out dx,al
  mov dx,w u_Command            ; Modifie le volume actuel
  mov al,9
  out dx,al
  mov dx,w u_datahi
  mov al,00010010b
  out dx,al
  mov dx,w u_command            ; Définit le volume de départ
                                ; du ramping
  mov al,7
  out dx,al
  mov dx,w u_datahi
  mov al,00010010b
  out dx,al
  mov dx,w u_command            ; Définit le volume final du Ramping
```

```
  mov al,8
  out dx,al
  mov dx,w u_datahi
  mov di,word ptr vol
  shl di,1
  mov ax,word ptr [offset uVolumes + di]
  shr ax,8
  out dx,al
  mov dx,w u_command          ; Direction du ramping dans le
                              ; contrôle du volume
  mov al,0dh                  ; Définit le registre
  out dx,al
  mov dx,w u_datahi
  mov al,0
  out dx,al
  ret
voice_rampin endp
```

15. Méthodes de traitement du son dans vos programmes

Toutes les démos conçues par les programmeurs modernes comportent un traitement du son, comme aussi tous les jeux vidéo. Une sortie réussie du son est un élément essentiel d'un bon logiciel. C'est pourquoi nous allons nous occuper dans ce chapitre de tous les aspects importants de la programmation du son et des différents formats de fichiers son.

Pour les deux cartes son les plus utilisées, la carte Sound Blaster et la carte son des branchés, la Gravis Ultrasound, vous trouverez dans le CD qui accompagne ce livre des players MOD rendant possibles des essais directs.

15.1. Le format de fichier MOD

Le format de fichier MOD provient en droite ligne de Comodore AMIGA, mais s'est frayé aussi la voie vers le PC avec l'expansion de la carte Sound Blaster. Cette carte permet d'exécuter de la musique numérisée sans grand investissement en matière de mémoire.

On parvient à ce résultat en réduisant les "sons naturels" en samples, puis en utilisant ces samples comme des instruments. Ceux-ci fournissent bien sûr un son beaucoup plus réaliste que la synthèse FM du Sound Blaster. La technologie "Sample Wave" est aussi supportée par le hardware, par exemple dans le Wave Blaster de Creative Labs ou dans le nouveau Gravis Ultrasound. Nous nous contenterons toutefois ici du seul canal de sample mis à notre disposition par la carte Sound Blaster.

Le format MOD

Il existe différentes versions du format MOD, puisque chaque programmeur concoctait au début sa propre potion magique. Le format MOD, que nous allons décrire dans ce qui suit, doit être considéré comme le format standard. Un fichier standard MOD peut être divisé en trois parties :

1. Le header MOD

Le header fait 1084 octets de long et contient toutes les informations importantes pour le morceau :

Octets 0-19 - Song Name

Dans les 20 premiers octets de la chanson se trouve le nom de cette dernière sous forme de chaîne C(NULL-String). Le dernier octet doit donc être égal à O.

Octets 20-41 - Nom du 1er instrument

Dans ces 22 octets se trouve le nom du premier instrument sous forme de chaîne C en format ASCII. Le dernier caractère doit être un 0.

Octets 42-43 - Longueur de l'instrument

Dans ces deux octets, on trouve la longueur du premier instrument. L'indication se rapporte aux mots. Pour obtenir le nombre d'octets, il faut multiplier la valeur par 2. De plus, notez qu'il s'agit ici d'un fichier qui provient à l'origine d'Amiga. Le processeur 68000 conserve une valeur dans l'ordre octet High/octet Low, et non pas dans l'ordre octet Low/octet High, comme les processeurs de la série 80x86. C'est pourquoi vous devez échanger les octets High et Low pour parvenir à la valeur voulue.

Octet 44 - Fine Tune

Seuls les quatre bits inférieurs sont utilisés. Ils donnent une valeur située entre -8 et +8, qui permet de réaliser un réglage fin de l'instrument. Cette option n'est toutefois utilisée que très rarement.

Octet 45 - Volume

Cet octet indique le volume par défaut de l'instrument. Les valeurs admises sont situées entre 0 et 64.

Octets 46-47 - Départ du loop

Le format MOD offre la possibilité de mettre un sample en boucle. Les deux octets désignent le début de la boucle (loop). Ici aussi, il faut se rappeler qu'il s'agit d'une indication de mot dans le style Amiga, c'est-à-dire que les octets High et Low sont intervertis et que le résultat doit être multiplié par 2.

Octets 48-49 - Longueur du loop

Ces deux octets contiennent la longueur du loop. Il s'agit d'une indication de mot dans le style Amiga. Les informations des octets 20 à 49 se répètent ensuite pour les 30 autres instruments dans le nouveau format MOD ou les 14 instruments pour l'ancien format MOD. Nous en resterons ici au nouveau format MOD, largement répandu.

Octet 950 - Longueur de la chanson

Cet octet désigne la longueur de la chanson en "patterns". Il désigne la longueur réelle de la chanson et non pas le nombre de patterns définis.

Octet 951 - Vitesse CIAA

Cet octet est spécifique à Amiga et n'a donc pas de signification particulière pour le PC.

Octets 952-1079 - Arrangement de la chanson

Chacun des 128 octets peut prendre une valeur entre 0 et 163, qui désigne le pattern à exécuter. Ces 128 octets et la longueur de la chanson déterminent l'arrangement de la chanson.

Octets 1080-1083 - Caractérisation

Ces quatre octets contiennent l'identité du fichier MOD. Si l'on y trouve la valeur 'M.K.' ou 'FLT4', il s'agit d'un fichier MOD à quatre voix avec 31 instruments. Sur le PC, on trouve désormais de nouveaux formats, comme '6CHN' pour un fichier à six voix et '8CHN' pour un fichier MOD à huit voix.

2. Les patterns MOD

Octets 1084-2107

Ces octets contiennent le premier pattern. Une note d'un pattern comprend quatre octets, avec quatre notes définies par ligne. Chaque pattern contient 64 lignes. On en déduit la longueur totale d'un pattern, soit 1024 octets. Le nombre de patterns est difficile à déterminer, il exige un calcul. Il faut d'abord prendre la longueur totale du fichier MOD et en retirer la longueur du header. Il faut ensuite additionner les longueurs des samples et les retirer également de la longueur du fichier. On divise le reste par la taille d'un pattern (1024) et l'on obtient le nombre de samples. Les patterns peuvent alors être chargés sans problème.

3. Les données des samples MOD

Octets n1-nx

Accolés aux patterns, on trouve les données sample en format Raw-Data signé. Cela veut dire qu'il s'agit purement et simplement de données sample 8 bits, qui prennent des valeurs entre -127 et 128. Sur le PC, les données sont définies au contraire entre 0 et 255. C'est pourquoi les données sample doivent encore être converties avant la sortie.

Le format d'une note MOD

Le format d'une note MOD est quelque peu compliqué, mais on peut en fournir une explication claire.

Octet 1

Les quatre bits supérieurs (7-4) doivent être interprétés comme les bits High du numéro d'instrument. Les quatre bits inférieurs font partie de l'indication de la hauteur du son.

Octet 2

L'octet contient avec les bits 3-0 du premier octet l'indication de la hauteur du son, mesurée en samples par minute. Les facteurs standard des samples sont les suivants :

$036 H-4	$071 H-3	$0E2 H-2	$1C5 H-1	$386 H-0
$039 A#4	$078 A#3	$0F0 A#2	$1E0 A#1	$3C1 A#0
$03C A-4	$07F A-3	$0FE A-2	$1FC A-1	$3FA A-0
$040 G#4	$087 G#3	$10d G#2	$21A G#1	$436 G#0
$043 G-4	$08F G-3	$11d G-2	$23A G-1	$477 G-0
$047 F#4	$097 F#3	$12E F#2	$25C F#1	$4BB F#0
$04C F-4	$0A0 F-3	$140 F-2	$280 F-1	$503 F-0
$055 E-4	$0AA E-3	$153 E-2	$2A6 E-1	$54F E-0
$05A D#4	$0B4 D#3	$168 D#2	$2D0 D#1	$5A0 D#0
$05F D-4	$0BE D-3	$17d D-2	$2FA D-1	$5F5 D-0
$065 C#4	$0CA C#3	$194 C#2	$328 C#1	$650 C#0
$06B C-4	$0D6 C-3	$1AC C-2	$358 C-1	$6B0 C-0

Octet 3

Dans les quatre bits supérieurs (7-4), on trouve les bits Low du numéro d'instrument. Dans les quatre bits inférieurs, c'est la commande d'effet qui est conservé.

Octet 4

L'octet 4 contient l'opérande de la commande d'effet.

Effets

Effet 00 - Arpeggio (deux opérandes)

L'effet arpège est tel qu'une note n'est pas jouée de manière continue sur le même ton, elle passe entre trois tons différents. Le premier paramètre désigne la différence entre les deux premiers tons, le second paramètre la différence entre les deux autres, ces différences étant mesurées en demi-tons.

Effet 01 - Portamento Up (un opérande)

Par cet effet, le ton est augmenté de façon constante. La vitesse d'accroissement est déterminée par l'opérande.

Effet 02 - Portamento Down (un opérande)

Par cet effet, le ton est diminué de façon constante. La vitesse de cette décroissance est déterminée par l'opérande.

Effet 03 - Portamento to Note (un opérande)

Contrairement à ce qui se passe dans les effets 01 et 02, c'est la hauteur visée qui est indiquée ici. L'opérande détermine à nouveau la vitesse.

Effet 04 - Vibrato (deux opérandes)

Cet effet permet d'obtenir un vibrato. Le premier opérande fournit la vitesse et le second la profondeur du vibrato.

Effet 05 - Portamento to Note + Volume Sliding (deux opérandes)

Avec cet effet, vous pouvez exécuter un portamento to Note et en même temps faire diminuer le volume. La partie supérieure de l'opérande (Xx) indique la vitesse du portamento et la seconde partie (xX) la vitesse avec laquelle le volume doit être diminué.

Effet 06 - Vibrato + Volume Sliding (deux opérandes)

Vous pouvez exécuter simultanément un vibrato et une décroissance du volume. Le premier opérande (Xx) indique la vitesse avec laquelle le vibrato aura lieu, tandis que le second opérande (xX) fournit la vitesse avec laquelle le volume sera diminuée.

Effet 07 - Tremolo (deux opérandes)

L'effet Tremolo est très proche de l'effet Vibrato. Il désigne une vibration du volume. Puisqu'on obtient aussi de cette manière un effet de vibrato, les routines d'exécution pour les effets Vibrato et Tremolo sont souvent identiques. Comme premier paramètre (Xx), on transmet la vitesse du tremolo. Le second opérande (xX) est la profondeur du tremolo.

Effet 08 - Non (encore) défini

C'est la seule commande non définie. Elle peut donc être éventuellement utilisée pour vos propres routines.

Effet 09 - Définir l'offset du sample (un opérande)

Cet effet consiste à faire démarrer une sortie de sample non à partir du début, mais à partir de la position offset transmise à l'intérieur des données du sample. La valeur transmise désigne les deux premiers nombres indiquant la longueur du sample.

Effet 10 - Volume Sliding (un opérande)

A l'aide de cet effet, vous pouvez réduire ou augmenter le volume d'un instrument. Pour un opérande de type n0, le volume est augmenté à la vitesse n. Pour un opérande de type 0n, le volume est réduit à la vitesse n.

Effet 11 - Position Jump (un opérande)

La sortie du pattern actuel est interrompue et poursuivie dans le pattern indiqué par l'opérande. L'opérande transmis désigne la ligne dans laquelle vous voulez continuer la sortie.

Effet 12 - Set Note Volume (un opérande)

Cet effet permet de choisir le volume d'une note. La valeur transmise dans l'opérande peut se situer entre 0 et 63.

Effet 13 - Pattern break (pas d'opérande)

Avec cet effet, vous interrompez la sortie du pattern actuel. La sortie se poursuit alors avec le pattern suivant.

Effet 14 - Commandes diverses (deux opérandes)

L'effet est très diversifié. Il a plusieurs sous-fonctions. Le premier opérande concerne la sélection du sous-effet souhaité. Le second opérande sert de paramètre au sous-effet.

Effet 14.1 - Fineslide Up

Cet effet se comporte exactement comme le "portamento up" normal. Elle ne provoque toutefois qu'une modification unique du son, contrairement au "portamento", avec lequel la hauteur du son se modifie pendant tout le temps d'exécution de la note.

Effet 14.2 - Fineslide Down

Cet effet est analogue au précédent, à cette différence qu'il fait descendre la note au lieu de la monter.

Effet 14.3 - Glissando Control

Le "glissando" agit en relation avec les commandes "portamento". Si vous l'activez, la modification du son se fait non pas de manière constante, mais par demi-tons. Un "1" dans le paramètre active le glissando. Au contraire, un 0 le désactive.

Effet 14.4 - Vibrato sinusoïdal

Cet effet permet de sélectionner une forme sinusoïdale pour le vibrato. La valeur par défaut 0 représente réellement une sinusoïde. Le paramètre 1 représente une diminution constante, 2 une sinusoïde rectangulaire et 3 une forme aléatoire.

Effet 14.5 - Définir un "finetune"

Vous pouvez modifier ici la valeur du "finetune". Vous utilisez le tableau suivant pour trouver la valeur souhaitée :

Finetune :	+7	+6	+5	+4	+3	+2	+1	0	-1	-2	-3	-4	-5	-6	-7	-8
Valeur :	7	6	5	4	3	2	1	0	15	14	13	12	11	10	09	08

Effet 14.6 - Pattern Loop

Cet effet permet de mettre en "boucle" une partie déterminée dans un pattern. Le paramètre 0 désigne ici le départ du loop. Chaque autre paramètre (n) conduit à une n-ième exécution du secteur, de la position de départ jusqu'à la position actuelle. Le morceau continue ensuite normalement.

Effet 14.7 - Tremolo sinusoïdal

Cet effet permet de sélectionner une forme sinusoïdale pour le tremolo. La valeur par défaut 0 représente réellement une sinusoïde. Le paramètre 1 représente une diminution constante, 2 une sinusoïde rectangulaire et 3 une forme aléatoire.

Effet 14.10 - Fine Volumesliding Up

Cet effet provoque un glissement de volume unique (volumesliding). Il fonctionne exactement comme le glissement de volume normal, à cette différence près que la modification est unique au lieu d'être continue.

Effet 14.12 - Sortie brève des notes

Cet effet provoque des sorties abrégées pour les notes. Le paramètre transmis désigne le nombre de battements après lesquels la sortie doit être interrompue. Les battements dont il est question ici concernent la vitesse définie par "Set Speed". Avec F06, vous auriez par exemple six battements. Si vous transmettez la valeur 4, seuls les 2/3 de la note seront exécutés.

Effet 14.13 - Délai pour une note

Cet effet provoque un délai avant la sortie d'une note. Le délai est transmis dans le paramètre sous la forme de "battements". On peut utiliser l'effet ainsi défini pour faire entendre un écho.

Effet 14.14 - Délai pour le pattern

Cet effet arrête la sortie du pattern pendant le temps indiqué dans le paramètre.

Effet 14.15 - Set Speed (un opérande)

Cet effet définit la vitesse avec laquelle le fichier MOD sera exécuté. Les valeurs autorisées pour l'opérande se situent entre 0 et 31.

Le format 669

Outre le format MOD standard, il existe d'autres formats, établis souvent par les concepteurs les plus en vue de démos. Le format 669 trouve ainsi son origine dans l'équipe de Renaissance. Il est semblable au format MOD, mais un peu plus compact. On peut le diviser lui aussi en trois parties :

Le header du format 669

Contrairement au header MOD, la longueur du header 669 n'est pas statique. Elle dépend du nombre de samples définis.

Octets 0-1 - Identité

Ces deux octets contiennent l'identité du fichier 669. La valeur de cette caractérisation à l'aide d'un mot (Word) est $6669.

Octets 2-110 - Song Message

Dans ces 108 octets, on trouve 3 lignes à 36 caractères chacun, comme une sorte de texte d'information sur la chanson. Ce message est propre au format 669.

Octet 111 - Nombre de samples

Cet octet contient le nombre de samples emmagasinés. Les valeurs autorisées se situent entre 0 et 64. Le nombre maximal de samples est donc ici le double de ce qu'il était pour un fichier standard MOD.

Octet 112 - Nombre de patterns

Cet octet contient le nombre de patterns emmagasinés. Cette caractérisation est utile, car ainsi on n'est pas obligé de calculer le nombre de patterns soi-même.

Octet 113 - Repeat Start

Cet octet désigne le pattern par lequel commence la répétition.

Octets 114-242 - Arrangement de la chanson

Dans ces 128 octets, on trouve l'arrangement de la chanson. Chaque entrée fournit le numéro du pattern exécuté. De cette façon, les valeurs autorisées se situent entre 0 et 128.

Octets 243-371 - Liste des tempos

Dans cet array long de 128 octets se trouvent les vitesses dans lesquelles on doit exécuter les patterns correspondants.

Octets 372-500 - Liste des position de break

Dans cet array long de 128 octets, on trouve les positions de break à l'intérieur des différents patterns. Chaque entrée dans la liste désigne la ligne dans laquelle l'exécution du pattern doit prendre fin. Les valeurs autorisées se situent entre 1 et 64.

Octet 501 - 501+(nombre de samples$19) - Informations sur les samples*

Dans le secteur qui suit, vous trouverez les informations sur les samples. La longueur du secteur dépend du nombre de samples déclarés. La longueur des

enregistrements déclarés pour chaque sample est de $19 octets. Elle a la structure suivante :

13 octets pour le nom de fichier de l'instrument sous forme de chaîne ASCII

1 dword (=longint)	Longueur de l'instrument
1 dword	Position de départ de la boucle de l'instrument
1 dword	Fin de la boucle pour l'instrument

Le pattern en format 669

Octets 501+(nombre de samples*$19) - Octets 501-501+(nombre de samples*$19)+(nombre de patterns*$600) - Patterns

Dans la partie qui suit, on trouve la définition des différents patterns. Un pattern est constitué de 64 lignes de 8 notes chacune. Chaque note est longue de 3 octets. On obtient ainsi une longueur totale de 1536 octets par pattern. Les informations occupent les octets tout entiers. Les bits ont les significations suivantes :

Définition des désignations

Octet 0	Octet 1	Octet 2
76543210	76543210	76543210

Octet 0, bit 7 à octet 0, bit 2 - Valeur des notes

Dans ces six bits, on trouve la valeur des notes à exécuter. Celle-ci est calculée en fonction de la formule VALEUR:=(12*octave)+(Note). La note Ré1 aurait ainsi la valeur 12*1+2=14. De cette manière, vous pouvez adresser tout de même cinq octaves.

Octet 0, bit 1 à octet 1, bit 4 - Numéro de l'instrument

Ces six bits contiennent le numéro de l'instrument à exécuter.

Octet 1, bit 3 à octet 1, bit 0 - Volume

Ces quatre bits contiennent le volume de la note à exécuter. Il n'y a certes que 16 volumes possibles, mais c'est amplement suffisant pour l'usage habituel, car une distinction plus fine serait pratiquement inaudible.

Octet 2, bit 7 à octet 2, bit 4 - Commande

Dans ces quatre bits est déposée la commande à exécuter. Les valeurs doivent être interprétées comme suit :

Valeur	Description
0	Portamento Up
1	Portamento Down
2	Portamento to Note
3	Pattern Break
4	Vibrato
5	Set Speed

Il faut noter ici que, pour le vibrato, l'opérande de la commande est déterminé au moyen de la formule "Valeur du vibrato":=Opérande SHL 4 + 1.

Octet 2, bit 3 à Octet 2, bit 0 - Opérande de la commande

Dans les quatre derniers bits, on trouve la valeur de l'opérande pour l'effet prévu.

Valeurs particulières

Les combinaisons suivantes entre octets possèdent une signification particulière :

- Octet 0 = $FE : Pas de note, seulement une modification de volume
- Octet 0 = $FF : Pas de note et pas de modification de volume
- Octet 2 = $FF : Pas de commande.

Le format de fichier Scream Tracker

Un autre format très populaire est celui du Scream Tracker de chez Future Crew. Le problème avec ce format est que les informations qui le concernent sont très difficiles à obtenir. Scream Tracker fait une distinction entre "Songs" et "Modules". La différence tient dans le fait que des instruments sont mémorisés (on a alors affaire à un module) ou non. Mais puisque seul le module a connu une véritable expansion, nous nous occuperons en priorité de celui-ci.

Le header STM

Le header des fichiers STM, que l'on peut créer avec le Scream Tracker V2.24, possède la structure suivante :

Octets 0-19 - Nom de la chanson

Dans les vingt premiers octets, on trouve le nom de la chanson en format ASCII. Pour modifier ce nom, saisissez-le sous la forme d'un tableau de caractères ("Array of char").

Octets 20-27 - Nom du tracker

Dans ces 8 octets, on trouve le nom du tracker avec lequel la chanson a été créée. La chaîne ne se termine pas par un 0.

Octet 28 - Version

Le numéro de version du fichier a été déposé dans cet octet. Il est actuellement de $1A.

Octet 29 - Type de fichier

Cet octet indique de quel type de fichier il s'agit. La valeur 1 représente une chanson (sans samples), la valeur 2 représente un module (avec samples).

Octets 30-31 - Numéro de version

Ces deux octets contiennent le numéro de version du fichier. L'octet 30 contient le numéro principal et l'octet 31 le numéro de la sous-version.

Octet 32 - Tempo

Le contenu de cet octet indique le tempo dans lequel le fichier devra être exécuté.

Octet 33 - Nombre de patterns

Cet octet contient le nombre de patterns déposés. Il a une grande importance, car on en a besoin ultérieurement lorsque l'on charge les patterns. C'est une disposition heureuse, car ainsi on n'est pas obligé de calculer le nombre de patterns, comme dans les fichiers MOD.

Octet 34 - Volume global

Vous trouverez ici le volume avec lequel le fichier doit être exécuté. Les valeurs autorisées sont situées entre 0 et 63.

Octets 36-47 - Reserved

Les octets 36 à 47 sont réservés pour les besoins internes du Scream Tracker.

Octets 48-80 - Instrument n° 1

Dans les 32 octets suivants, on trouve les données du premier instrument. Les informations concernant les instruments sont disposées de la manière suivante :

Octets 48-59 - Nom de l'instrument

Les octets 48 à 59 contiennent le nom du premier instrument sous forme de texte ASCII.

Octet 60 - Nom d'instrument 0

L'octet 60 contient toujours la valeur 0 et termine le nom de l'instrument (ASCIIZ).

Octet 61 - Lecteur des instruments

Cet octet indique le numéro de lecteur sur lequel l'instrument est déposé. Ce numéro est important pour les "songs" (sans samples).

Octets 62-63 - Réservés

Ces deux octets sont réservés. Ils sont en fait utilisés dans le fonctionnement interne en tant que segment d'adresse.

Octets 64-65 - Longueur de l'instrument

Ces deux octets contiennent la longueur de l'instrument. L'indication est donnée en nombre d'octets.

Octets 66-67 - Loop Start

Les octets 66 et 67 contiennent la position de départ à l'intérieur des données du sample pour la boucle dans laquelle l'instrument pourra être exécuté.

Octets 68-69 - Loop End

Ces deux octets contiennent la position finale de la boucle à l'intérieur des données du sample.

Octet 70 - Volume

On a ici le volume par défaut de l'instrument. Il peut prendre des valeurs situées entre 0 et 63. On peut remplacer ces valeurs pour le temps de l'exécution.

Octet 71 - Réservé

Cet octet est réservé, sans autre précision.

Octets 72-73 - Vitesse pour C1

Dans ces deux octets devrait se trouver la vitesse pour C1. En réalité, la valeur qui s'y trouve est en règle générale égale à 0.

Octets 74-77 - Réservés

Ces quatre octets sont également réservés pour les besoins internes.

Octets 78-79 - Adresse segment/ Longueur en paragraphes

Vous trouverez ici l'adresse segment interne dans une chanson. Dans les modules les plus courants, on trouve ici la longueur exprimée en paragraphes (longueur en octets divisée par 16).

Ces informations se répètent ensuite pour les trente autres instruments. Ensuite la disposition des octets continue de la façon suivante :

Octets 1028-1091 - Arrangement

Dans cet array long de 64 octets, vous trouverez l'arrangement du morceau. Une entrée dans cet array représente le numéro du pattern correspondant.

Octets 1092-2115 - Pattern 1

Les octets 1092 à 2115 contiennent le premier pattern. Il comprend 64 lignes avec 4 notes chacune. Une note est constituée par 4 octets. Il en résulte que la taille d'un pattern est de 1024 octets (64*4*4). La structure des notes est la suivante :

Octet 0, bits 7 à 4 - Octave

Ces quatre octets contiennent le numéro d'octave de la note à exécuter.

Octet 0, bits 3 à 0 - Note

On trouve ici la note à exécuter. 0= Do, 1 = Do dièse, 2 = Ré, etc.

Octet 1, bits 7 à 3 - Instrument

Ces cinq bits contiennent le numéro de l'instrument qui doit jouer.

Octet 1, bit 2 à Octet 2, bit 4 - Volume

Le volume de l'instrument est défini dans ces bits.

Octet 2 bits 3 à 0 - Effet

Ces bits contiennent le numéro de l'effet en cours. Les numéros sont les mêmes que pour les fichiers MOD.

Octet 3 - Opérandes

Cet octet contient les opérandes de l'effet.

Les autres patterns définis suivent ce modèle. Leur nombre est déposé dans le header. S'il s'agit d'un module, les données des samples viennent après les patterns. Leur longueur est définie également dans le header.

Le format S3M

Le format S3M est le format son utilisé actuellement par Future Crew. Il provient de Scream Tracker 3.0 et c'est sans doute l'un des formats son les plus flexibles. Il peut aussi bien emmagasiner des modules de samples que des morceaux Adlib. Il distingue les modules contenant des samples et les chansons contenant uniquement l'arrangement.

Le header S3M

Octets $00-$1B - Nom de la chanson

Au début du header, on trouve le nom du morceau. Le nom est terminé ici par un 0.

Octets $1C - Identité $1A

Cet octet contient toujours la valeur $1A. Il sert simplement à l'identification et n'a pas d'autre fonction.

Octet $1Q - Type de fichier

Pour le type de fichier, on distingue entre les fichiers modules et les fichiers chansons. La valeur 16 représente un fichier module. La valeur 17 représente un fichier chanson.

Octets $1E-$1F - Non utilisés
Octets $20-$21 - Longueur de l'arrangement

Dans cette variable Word, on trouve la longueur variable de l'arrangement. Cette longueur doit toujours être un nombre pair.

Octets $22-$23 - Nombre des instruments

Ce "mot" (word) contient le nombre des instruments utilisés dans le morceau.

Octets $24-$25 - Nombre des patterns

Vous trouverez ici le nombre de patterns définis. La valeur est d'ailleurs nécessaire pour connaître la longueur des "parapointeurs" sur les patterns.

Octets $26-$27 - Flags

Les flags influencent l'initialisation des différents effets. Les bits ont les significations suivantes :

Bit 1	Vibrato ST2
Bit 2	Temp ST2
Bit 3	Utilisation du sliding Amiga
Bit 4	Optimisation pour volumes 0
Bit 5	Conserver les limites Amiga
Bit 6	Rendre possibles les effets filtre et son

Octets $28-$29 - Information sur la version

Ces deux octets contiennent l'information sur la version du fichier. Le numéro du tracker se trouve dans les quatre bits supérieurs. Le numéro de version se trouve quant à lui dans les douze bits inférieurs. La valeur pour le Scream Tracker 3.0 est $1300.

Octets $2A-$2B - Version du format de fichier

Il existe actuellement deux versions différentes pour le format de fichier. La valeur 1 signale qu'il s'agit d'un fichier original. La valeur 2 signale une version originale dans laquelle les samples sont déposés en format "non signé", comme nous y sommes habitués dans le PC.

Octets $2C-$2F - Identification

Ces octets contiennent la caractérisation 'SCRM', qui permet d'identifier le fichier Scream Tracker.

Octet $30 - Volume master

Cet octet contient le volume master utilisé au départ.

Octet $31 - Vitesse de départ

La vitesse utilisée au départ se retrouve dans cet octet (commande A).

Octet $32 - Tempo de départ

Cet octet contient le tempo utilisé au départ (commande T).

Octet $33 - Multiplicateur master

La valeur du multiplicateur maître (Master Multiplier) se trouve dans les quatre bits inférieurs. Vous l'obtenez par l'intermédiaire d'un AND $0F. Si

les quatre bits supérieurs comportent une valeur non nulle, vous êtes en mode stéréo.

Octets $34-$3F - Non utilisés
Octets $40-$5F - Spécifications du canal

Dans ces 32 bits se trouvent les spécifications pour les 32 canaux (nombre maximal de canaux possibles). La signification des valeurs est donnée par le tableau suivant :

Valeur	Signification
0	Sample sur le canal de gauche (S1)
1	Sample sur le canal de gauche (S2)
2	Sample sur le canal de gauche (S3)
3	Sample sur le canal de gauche (S4)
4	Sample sur le canal de droite (S5)
5	Sample sur le canal de droite (S6)
6	Sample sur le canal de droite (S7)
7	Sample sur le canal de droite (S8)
8	Voix Adlib Melody, gauche (A1)
9	Voix Adlib Melody, gauche (A2)
10	Voix Adlib Melody, gauche (A3)
11	Voix Adlib Melody, gauche (A4)
12	Voix Adlib Melody, gauche (A5)
13	Voix Adlib Melody, gauche (A6)
14	Voix Adlib Melody, gauche (A7)
15	Voix Adlib Melody, gauche (A8)
16	Voix Adlib Melody, gauche (A9)
17	Voix Adlib Melody, droite (B1)
18	Voix Adlib Melody, droite (B2)
19	Voix Adlib Melody, droite (B3)
20	Voix Adlib Melody, droite (B4)
21	Voix Adlib Melody, droite (B5)
22	Voix Adlib Melody, droite (B6)

Valeur	Signification
23	Voix Adlib Melody, droite (B7)
24	Voix Adlib Melody, droite (B8)
25	Voix Adlib Melody, droite (B9)
26	Adlib Basedrum, gauche (AB)
27	Adlib Snare, gauche (AS)
28	Adlib Tom, gauche (AT)
29	Adlib Cymbal, gauche (AC)
30	Adlib Hihat, gauche (AH)
31	Adlib Basedrum, droite (BB)
32	Adlib Snare, droite (BS)
33	Adlib Tom, droite (BT)
34	Adlib Cymbal, droite (BC)
35	Adlib Hihat, droite (BH)
128	inactif
255	inutilisé

Octets $60-($60+Longueur de l'arrangement)-Arrangement

On retrouve ici l'arrangement du morceau. Les valeurs reportées représentent les numéros des patterns à exécuter.

Octet xxxx - Parapointeurs sur les instruments

Tout de suite après l'arrangement, on trouve les parapointeurs sur les différents instruments. Ce sont des pointeurs de paragraphe. Le mot désigne l'offset compté à partir du début du header, divisé par 16. Un parapointeur a toujours une taille d'un mot. La longueur totale des parapointeurs sur les instruments est donc de (Nombre d'instruments)*2.

Octet xxx - Parapointeurs sur les patterns

Après les parapointeurs sur les instruments viennent les parapointeurs sur les patterns. Ils désignent le début de chaque pattern à l'intérieur du fichier. La longueur de ces parapointeurs est de (Nombre de patterns)*2.

Le pattern S3M

La longueur d'un pattern se calcule à partir du nombre de canaux utilisés, multiplié par 320 octets pour chaque canal. Les notes des différentes lignes

sont déposées l'une à la suite de l'autre. Chaque note comprend cinq octets. Les octets ont la signification suivante :

Octet 0	Note, bits 7-4 = octave
	bits 4-0 = note
	255 = vide
	254 = Key off (pour Adlib)
Octet 1	Numéro d'instrument, 255 = vide
Octet 2	Volume, 255 = vide
Octet 3	Commande, 255 = vide
Octet 4	Paramètre de commande

Les instruments en S3M

Le format S3M fait la distinction entre deux types d'instruments. Il utilise d'une part les instruments sous forme de samples, connus déjà dans les fichiers MOD. Mais il supporte aussi les instruments Adlib. Les deux types d'instruments comprennent un header long de $40 octets, suivi par les données des samples dans le cas du premier type. Considérons d'abord la structure du header pour les instruments sous forme de samples.

Octet $0 - Type de l'instrument

Si vous trouvez la valeur 1 dans cet octet, il s'agit d'un instrument sous forme de sample. Sinon, il s'agit d'un instrument Adlib.

Octets $1-$C - Nom de fichier DOS

Le nom du fichier de sample est déposé dans ces octets. Il ne s'agit pas ici toutefois d'une chaîne terminée par 0 !

Octets $D-$F - Segment de mémoire

Le segment de mémoire contient la position exprimée en nombre de paragraphes (Offset divisé par 16) des données sample.

Octets $10-$13 - Longueur du sample

Dans ce double-mot est déposée la longueur des données pour les samples. Il est recommandé de vérifier si la longueur dépasse le secteur du mot et de travailler avec des variables de format word.

Octets $14-$17 - Début du loop

La position de départ de la boucle à l'intérieur des données pour les samples est déposée dans ce double-mot.

Octets $18-$1B - Fin du loop

La position finale de la boucle à l'intérieur des données pour les samples est déposée dans ce double-mot.

Octet $1C - Volume

Le volume par défaut de l'instrument est déposé dans cet octet.

Octet $1D - Disque

Cet octet contient l'indication du disque sur lequel se trouve éventuellement un instrument donné.

Octet $1E - Type de pack

L'octet indique si les données sample sont compactées ou non. La valeur 0 indique des données 8 bits non compactées, la valeur 1 indique un algorithme de compactage DP30ADPCM1.

Octet $1F - Flags

Dans les flags, vous trouverez des informations plus précises sur le type des données sample. Si le bit 1 est posé, cela veut dire que le looping est activé. Le bit 2 indexe les données stéréo, tandis que le bit 3 représente les samples 16 bits en format Intel.

Octets $20-$23 - C2 Speed

La valeur est utilisée pour accorder l'instrument. Il désigne la vitesse avec laquelle le sample doit être exécuté pour que l'on obtienne le son C2 (do de la seconde octave).

Octets $24-$27 - Non utilisés
Octets $28-$29 - Position dans la mémoire GRAVIS

Ce "mot" (word) n'est utilisé que par le Scream Tracker 3.0. Il désigne la position du sample dans la RAM de la Gravis Ultrasound DIV 32.

Octets $2A-$2F - Non connu exactement
Octets $30-$4B - Nom de l'instrument

Le nom de l'instrument est déposé ici sous la forme d'une chaîne ASCII possédant au maximum 28 caractères.

Octets $4C-$4F - Identification

C'est ici que se trouve la caractérisation 'SCRS' pour Scream Tracker Sample.

Nous en venons maintenant aux instruments Adlib. La structure du header est semblable dans ses grandes lignes à celle de l'instrument sample :

Octet 0 - Type de l'instrument

Cet octet contient le type de l'instrument. En voici la signification :

Valeur	Signification
2	Adlib Melody
3	Adlib Basedrum
4	Adlib Snare
5	Adlib Tom
6	Adlib Cymbal
7	Adlib Hihat

Octets $1-$C - Nom de fichier DOS

Le nom de fichier DOS de l'instrument Adlib se trouve ici.

Octets $D-$F - Octets vides

Les octets $D à $F contiennent la valeur 0.

Octets $10-$1A - FM Synthese Table

Les octets $10 à $1A contiennent tour à tour le modulateur et le support pour la synthèse FM. Les informations fournies doivent toutefois être munies d'un point d'interrogation.

Les valeurs des octets seront calculées d'après les formules suivantes :

- Octets $10 & $11 = (Multiplicateur de fréquence)+(sustain*32)+(pitch vibrato*64)+(volume vibrato*128)
- Octets $12 & $13 = (63 - volume)+((levelscale AND 1)*128)+((levelscale AND)*64)
- Octets $14 & $15 = (Attack * 16) + decay
- Octets $16 & $17 = ((15 - sustain) * 16) + release

- Octets $18 & $19 = wave select
- Octets $1A = (modulation feedback * 2) + additive synthesis

Octet $1B = non utilisé
Octet $1C - Volume

Le volume par défaut de l'instrument est déposé dans cet octet.

Octet $1D - Disque

Cet octet définit sur quel disque se trouve l'instrument.

Octets $1E-$1F - non utilisés
Octets $20-$23 - C2 Speed

La valeur est utilisée pour accorder l'instrument. Il désigne la vitesse avec laquelle le sample doit être exécuté pour que l'on obtienne le son C2 (do de la seconde octave).

Octets $24-$2F - non utilisés
Octets $30-$4B - Nom de l'instrument

Le nom de l'instrument est déposé ici sous la forme d'une chaîne ASCII possédant au maximum 28 caractères.

Octets $4C-$4F - Identification

C'est ici que se trouve la caractérisation 'SCRS' pour Scream Tracker Sample.

15.2. Un MOD-Player pour la carte Sound Blaster

Vous avez appris dans ce livre comment vous pouvez programmer des démos graphiques. Mais la démo la plus extraordinaire ne serait encore rien s'il n'y avait pas le son en arrière-plan. Il faut donc un Soundplayer. La méthode la plus simple serait de se servir d'un VOC-Player. Pourtant, quand le flux de données est d'environ 16.000 octets par seconde, vous auriez besoin déjà de 2,3 Mo rien que pour les données du son, avec une démo ne dépassant pas 2,5 minutes. Même avec des loopings et un compactage des données, cette quantité ne se laisserait pas ramener à une valeur raisonnable (environ 400 Ko). La méthode VOC ne convient par conséquent que pour des présentations commerciales ou pour des applications CD-ROM, là où quelques Mo en plus ne changent rien à l'affaire.

Il faut donc trouver une solution qui soit capable de fournir un son de bonne qualité tout en exigeant peu de place en mémoire. C'est pourquoi nous devons prendre en compte les fichiers MOD. Un Player pour ces fichiers est malheureusement toujours étroitement lié au matériel dont on dispose. Dans ce paragraphe, nous allons construire un Player pour la carte Sound Blaster. Dans le paragraphe suivant, nous présenterons un Player pour la carte GUS, bien mieux appropriée à ce genre de tâches.

Passons d'abord à la version Sound Blaster. La carte Sound Blaster est sans doute la carte la plus répandue, ce qui est dû en grande partie aux qualités publicitaires de Creative Labs. La carte est l'ancien standard pour les démos. Même si elle pose certains problèmes, vous êtes obligés de la prendre en compte, dans la mesure où le temps de calcul et la mémoire restante le permettent.

Vous pourriez répliquer : "En réalité, la sortie des MOD sur la carte SB ne devrait pas poser de problèmes insurmontables. Il s'agit finalement d'une simple sortie de sample via DMA et elle n'a quand même pas besoin de si longs calculs !" Ce n'est malheureusement vrai qu'en partie. Envisageons donc la bonne vieille Amiga, d'où provient le format MOD. Elle avait quatre DAC pour la sortie du son. Avec ces quatre canaux, on pouvait sortir les MOD réellement sans rencontrer aucun problème, puisque les manipulations pouvaient se limiter aux modifications de volume et de fréquence. Avec la Sound Blaster, les choses sont différentes. Cette carte ne possède en effet qu'un seul canal DAC (deux sur les modèles stéréo). La conséquence en est qu'il n'est plus possible de sortir les données sans s'en préoccuper. On est obligé de calculer toutes les données jusqu'au bout. Pour une fréquence de sortie de 22 KHz, cela représente tout de même 22.000 données par seconde.

Il faut donc que nous disposions d'un Player très rapide, si nous voulons que nos histoires de son soient audibles derrière les graphiques sans baisse conséquente de la qualité. D'où la restriction suivante : puisque nous devons utiliser une arithmétique sans virgule flottante pour des raisons de rapidité et que nous avons besoin malgré tout d'une résolution de 24 bits, le 286 ne peut pas entrer en ligne de compte avec ses registres qui ne font que 16 bits de large. Il est bien sûr possible de réaliser quand même cette résolution sur un 286 en se servant d'astuces. Il n'en reste pas moins qu'il est un peu faible de constitution et qu'il représente de toute façon une espèce en voie d'extinction. La dépense d'énergie nécessaire à cette programmation et la baisse considérable de qualité ne parlent pas en la faveur de cette solution.

Si vous voulez écrire le Player en Turbo-Pascal, le langage dans lequel sont écrites la plupart des démos, vous devrez ensuite vous demander sous quelle forme vous allez faire intervenir les 32 bits. Dans les versions de Pascal inférieures à 8, il n'est malheureusement pas possible d'écrire un code sur 32 bits et une version qui en resterait purement au Pascal ferait preuve de toute manière d'une lenteur désespérante. Il ne nous reste donc qu'une seule chose

à faire : à intégrer un module assembleur externe. Nous pourrons alors utiliser tous les moyens mis à notre disposition et faire intervenir tout l'arsenal des commandes du processeur 386.

Principes de base d'un Player MOD

Avant d'en venir à la programmation d'un player MOD, nous devons nous demander comment il fonctionne. Ce sera la première question : comment réaliser des sons différents ?

Les données déposées dans un fichier MOD sous forme de samples représentent l'amplitude de l'oscillation. Sur Amiga, vous pouvez choisir des valeurs entre -128 et 127. Sur le PC, ces valeurs sont situées entre 0 et 255. Cela veut dire que la valeur pour laquelle il n'y a rien à entendre (pas d'oscillation) est sur l'Amiga la valeur 0, tandis que c'est la valeur 127 sur le PC. Pour convertir un sample de format Amiga en un sample de format PC, il faut donc augmenter chaque valeur d'un coefficient 127. Le fait que seules les amplitudes sont mémorisées rend la chose facile. Nous nous servons d'un effet que vous connaissez sans doute pour l'avoir expérimenté au bon vieux temps du microsillon : si on le fait tourner trop vite, le son devient trop aigu ; si on le fait tourner au contraire trop lentement, une soprano reçoit une voix de basse. C'est exactement ce que nous ferons avec nos samples. Nous les exécuterons rapidement ou lentement.

Pourtant, dans la pratique, nous ne pouvons définir à chaque fois la vitesse de sampling pour la carte Sound Blaster. Cela fonctionne au plus pour une voix. Quand on a quatre voix ou plus, les problèmes surgissent. C'est pourquoi nous devons nous servir de petites astuces. L'idée de base est la suivante : nous sortons notre fichier MOD par exemple à une vitesse de 16 KHz. Cela signifie qu'avec un sample à 16 KHz, nous entendons le son original. Puisque l'on a affaire à 16.000 octets, il faut revenir à chaque fois en arrière d'un octet par sample pour déterminer l'octet à sortir. Si nous voulons maintenant sortir le sample à 32 KHz, nous devons l'exécuter à une vitesse double. Cela veut dire que nous ne pourrons plus sortir tous les octets dans le temps disponible. Nous ne pourrons sortir qu'un octet sur deux. C'est précisément ainsi que l'on procède pour une sortie à 8 KHz. Nous avons cette fois du temps pour sortir deux fois chaque octet.

Sur le plan musical, vous devez savoir que la différence entre les demi-tons est toujours la racine douzième de 2. Cela veut dire en pratique qu'un son situé à un demi-ton au-dessus du précédent doit être exécuté 1,059463094 fois plus vite. De même, si l'on a affaire à un son situé un demi-ton en dessous, le facteur est cette fois 0,943874312. C'est ce dont nous avons besoin dans notre programme : il serait absurde d'exécuter tous les calculs pendant l'exécution à l'intérieur des samples. Nous utiliserons plutôt une table calculée à l'avance avec les facteurs correspondants à chaque note. La différence entre deux demi-tons sera toujours la racine douzième de 2.

Le point suivant auquel nous devons consacrer notre attention concerne les volumes. Rappelons-nous que les valeurs des samples sont les amplitudes de l'oscillation. Avec une amplitude de 0 (ou de 127, c'est selon), le silence règne. Cela permet de comprendre le système selon lequel fonctionne le volume. Plus les amplitudes sont grandes, plus le sample résonne haut et fort. Si nous voulons alors calculer dans le fichier MOD les volumes des différentes voix, nous devons utiliser une variable word. Les volumes dans un fichier MOD sont en effet toujours situés entre 0 et 63. Il suffit donc de multiplier l'octet du sample par le volume, puis de diviser par le volume maximal, c'est-à-dire 64. De façon pratique, 64 est égal à 2 puissance 6, ce qui permet de remplacer la division par un SHR 6. Cela épargne 13 cycles par parcours.

Par l'intermédiaire du volume, on peut aussi réaliser simplement un effet comme le vibrato. Il suffit de monter et de baisser le volume d'une valeur qu'on lira dans une table, pour donner l'impression du vibrato. Bien entendu, vous pouvez réaliser aussi un vibrato au moyen de la vitesse du sample. Si vous l'augmentez et la diminuez de manière périodique, vous obtenez également un effet de vibrato. C'est à vous de choisir la méthode qui vous convient. La méthode des volumes exige un peu moins de temps de calcul (vous travaillez uniquement avec des octets), mais elle est un peu moins pure que l'autre et elle pose des problèmes quand on fait agir en même temps d'autres effets, comme le volume-sliding.

Mélanger des sons - Structure d'une procédure de mixage

Maintenant que vous avez acquis les connaissances élémentaires sur la question, nous allons passer à la réalisation du player. Le rôle central dans le player est dévolu à la procédure de mixage. Pour l'octet de sortie, elle doit être parcourue par chacune des voix actives. Lorsqu'une voix devient inactive, il n'est pas nécessaire de repasser l'ensemble de la procédure et de charger un octet vide. Il suffit de sauter la voix en question, c'est-à-dire d'appeler une procédure vide. Voici tout d'abord la structure schématique d'une procédure de mixage.

<table>
<tr><td colspan="3">Position_voix < fin_position de la voix ?</td></tr>
<tr><td>Oui</td><td colspan="2">Non</td></tr>
<tr><td rowspan="3"></td><td colspan="2">Voix en boucle ?</td></tr>
<tr><td>Oui</td><td>Non</td></tr>
<tr><td>Position initiale</td><td>Procédure de mixage chargée sur proc_vide et octet_vide</td></tr>
<tr><td colspan="2">Charger l'octet du Sample</td><td rowspan="2"></td></tr>
<tr><td colspan="2">Adapter au volume</td></tr>
<tr><td colspan="3">Ajouter au Mixbuffer</td></tr>
<tr><td colspan="3">Position de la voix augmentée du facteur d'incrémentation</td></tr>
</table>

Au début de la procédure, nous devons vérifier si la fin de la voix est atteinte ou non. Il est préférable de comparer non pas avec la fin de la voix, mais avec une valeur proche de cette fin. Les quelques octets qui manquent dans le sample n'attireront pas l'attention. Si vous voulez cependant comparer avec la fin proprement dite, vous obtenez presque toujours un dépassement. Celui-ci conduit à ce que des données aléatoires se mélangent à vos données de sortie. C'est une source de discordances sonores facilement évitables.

Lorsque la position finale n'a pas encore été atteinte, vous pouvez poursuivre le traitement normalement. Sinon, vous devez vérifier si la voix est ou non soumise à un looping. Si c'est le cas, vous définirez la position à l'intérieur de la voix sur la valeur de départ du loop. Sinon, le sample est parvenu à son terme. Dans ce cas, il faut charger un octet vide (amplitude = 0) et affecter à la variable d'appel de la procédure de mixage non plus la voix active, mais la procédure vide mentionnée ci-dessus.

Une fois que vous avez réalisé cette interrogation concernant la position, vous devez charger l'octet-sample à partir du sample. Vous changerez la valeur de l'octet en suivant la méthode décrite plus haut pour la conformer au volume de la voix. Vous pouvez alors ajouter l'octet dans le tampon de mixage. Ce tampon sera ultérieurement divisé par le nombre de voix, ce qui permettra d'atteindre une interpolation satisfaisante entre les voix.

Finalement, vous devez encore définir la position à l'intérieur du sample. Vous le ferez par une simple addition du facteur "Inc" à la position actuelle. La position et le facteur "Inc" devraient être des valeurs sur 32 bits, afin de pouvoir calculer avec une exactitude de 16,16 bits.

Player MOD pour la carte Sound Blaster

Le projet de développement d'un player MOD demande un certain temps. Les programmeurs qui manquent d'expérience risquent d'être effrayés par la durée du travail exigé ou de faire des fautes en travaillant trop vite. Nous n'échappons d'ailleurs pas à la règle. Mais ne craignez rien, on ne vous demande pas d'inventer la roue une seconde fois. Dans la suite de ce paragraphe, vous allez faire connaissance avec l'unité MOD_SB. Cette unité est facilement intégrable dans vos programmes et elle est assez souple. D'une part, vous pourrez la mettre en oeuvre par l'intermédiaire de l'interruption Timer. Elle sera appelée 50 fois par seconde et calculera une fraction des données à sortir. La sortie sera contrôlée par l'interruption SB. C'est la solution la plus satisfaisante et elle fonctionne sur toutes les cartes qui exécutent correctement le transfert SB-DMA. L'autre méthode disponible est la "méthode de Polling". Dans ce cas, le son n'est pas calculé périodiquement, mais seulement lorsque le moment est venu de le faire. C'est normalement le cas quand on attend le balayage. Cette méthode n'est cependant pas tout à fait sûre, car il peut arriver que l'interruption SB soit exécutée à un moment où toutes les données ne sont pas encore calculées. Pourtant, il y a dans une démo des parties qui exigent cette méthode. C'est pourquoi nous l'avons implémentée dans l'unité, même si elle correspond à une diminution de la qualité.

L'unité est assez rapide. Pour une qualité d'émission de 22 KHz lors de la sortie d'un fichier MOD à huit voix sur une machine 486dx-33, elle exige moins de 15% du temps de calcul. Avec un MOD à quatre voix, vous êtes aux alentours de 13%. En fonction des exigences de votre propre programme, vous pouvez mettre en oeuvre un fichier à quatre ou à huit voix. Si vous n'occupez pas tous les canaux du fichier à huit voix, la sortie sera d'autant plus rapide.

L'unité contient même un petit "bonus", puisqu'elle comporte une routine servant à exécuter les fichiers VOC. De cette façon, elle propose un outil universel de sortie du son pour la SB.

Le programme MOD386 montre comment on utilise cette unité de programmation. MOD386 est un petit player MOD et VOC. Vous devez lui transmettre le nom du fichier à exécuter ou l'appeler dans un menu qui apparaîtra à l'écran. Si vous entrez le paramètre de ligne de commande -r au moment du démarrage, vous vous trouverez en mode Repeat. Vous pourrez alors exécuter l'un après l'autre différents morceaux de musique (MOD ou VOC). Pour savoir quel temps de calcul il vous reste, appuyez dans le module MOD sur la touche <P> comme "Performance". Le player calcule le temps qui lui reste (cela dure environ trois secondes) et affiche la performance dont est encore capable le système. Celle-ci s'obtient en vérifiant combien de temps il reste au système avant un balayage vertical. Si le système ne fait rien d'autre qu'attendre le balayage et incrémenter la caractère de test, il vous reste 100%

de temps de calcul. Ce test sera exécuté à chaque démarrage du programme. Si vous appelez ce test de performance, la boucle ne fait en principe rien d'autre qu'attendre le balayage. La différence tient néanmoins dans le fait que le fichier MOD est en cours d'exécution à l'arrière-plan par l'intermédiaire du Timer. Le calcul des données exige entre 10% et 30% des capacités du système, selon l'ordinateur utilisé et la vitesse spécifiée pour le sample. La valeur incrémentée dans la boucle sera d'autant plus faible. C'est d'elle que l'on déduit directement la performance.

Variables nécessaires du player MOD

Maintenant que vous savez ce qui vous attend dans l'unité MOD_SB, nous allons entrer plus profondément dans les détails. Pour tirer tous les bénéfices possibles de cette unité ou pour la compléter (on pourrait imaginer par exemple un track pour les effets son...), vous devez savoir exactement ce qui s'y passe et à quelles fins on l'utilise. C'est pourquoi nous allons passer ici en revue toutes les variables essentielles, avant d'entrer dans la programmation proprement dite.

Commençons par la description des types dont dispose l'unité :

VocHeader

Dans le VocHeader (que vous trouverez dans l'unité VARIAB.PAS) sont déposées toutes les informations de base d'un fichier VOC. En premier lieu, on y trouve l'identification "Creative Voice File"+1Ah, longue de 20 octets. Nous y trouvons ensuite la position à partir de laquelle commencent les données sous la forme d'un mot (word). Vient ensuite le numéro de version du fichier VOC, d'abord l'octet comportant le numéro High, puis l'octet comportant le numéro Low. Le header se termine enfin par une identification codée du numéro de version, qui ne nous occupera pas ici.

VoiceBlock

Un fichier VOC est divisé en plusieurs blocs. Ceux-ci commencent par un header, chargé dans ce type de données. Le premier octet contient ainsi l'identification. Les trois octets suivants contiennent la longueur. La vitesse du sample est déposée dans l'octet suivant. Le dernier octet fournit les informations sur le type de compactage.

Pt

"Pt" est un type de données simple, qui facilite le travail avec les pointeurs. Il contient d'abord le mot ofs, la partie offset du pointeur, puis sgm, la partie segment du pointeur.

Il n'y a pas d'autre type utilisé. Passons donc maintenant aux constantes :

Incfacts : array[1..99] of longint;

"Incfacts" est un tableau d'intervalles calculés à l'avance. Les valeurs qu'il contient sont à virgule fixe. Les 16 bits inférieurs contiennent la partie décimale du pas à franchir, les 16 bits supérieurs contiennent la partie entière.

Outfading : boolean = FALSE;

Si vous ne voulez pas mettre brusquement fin à votre fichier MOD, vous devez affecter à cette variable la valeur TRUE. Le volume de sortie sera alors diminué lentement jusqu'à 0.

outvolume : byte = 63;

"Outvolume" désigne le volume avec lequel on sort le fichier MOD. La valeur maximale est de 63, la valeur minimale de 0 (=silence). Cette variable est décrémentée par exemple lorsque vous affectez à "outfading" la valeur TRUE.

TailleFiMod : longint = 0;

Cette variable est prévue spécialement pour la mise en oeuvre de l'unité dans les démos et les jeux. Il n'est pas question en effet dans ce cas de laisser les fichiers MOD se balader sous la forme de fichiers isolés. On les compacte quelque part dans un grand fichier son. Il est cependant nécessaire de connaître la taille d'un fichier MOD. C'est pourquoi on doit affecter à cette variable la valeur correspondant à la taille du fichier MOD, dans le cas où vous avez inclus ce dernier dans un grand fichier son. Lorsque la variable prend la valeur 0, la taille du fichier est considérée comme la même que la taille du fichier dans lequel il figure.

Mastervolume : byte = 29;

"Mastervolume" est une variable qui intervient sur les cartes à partir de SB Pro. Ces cartes sont équipées d'une puce de mixage, permettant de régler le volume au niveau du matériel. Le volume maximal est ici 31. Le volume minimal est 0.

Startport : word = $200;

La carte Sound Blaster est reconnue automatiquement. Cela se fait au moyen d'un test des ports par pas de $10 entre le port initial et le port final. Si vous avez installé une carte de réseau et si vous ne voulez pas que le port de la carte soit recherché par ce test (cela risque en effet de conduire au blocage du système tout entier), entrez en conséquence la valeur de départ ou celle d'arrivée. La même chose est vraie lorsque vous voulez tester un seul port, sachant que c'est le bon. Indiquez des valeurs initiale et finale qui soient égales. Seul le port ainsi défini sera testé.

Endport : word = $280;

Reportez-vous à "Startport".

force_irq : boolean = FALSE;

La reconnaissance du port de base peut être entièrement désactivée si vous affectez la valeur TRUE à cette variable. C'est la valeur 200h qui sera adoptée.

interrupt_check : boolean = FALSE;

Cette constante globale permet de faire savoir à la procédure d'interruption SB qu'il s'agit d'un test des interruptions et qu'il n'y a pas de bloc à sortir.

timer_per_second : word = 50;

Dans la variable timer_per_second, on place le nombre d'appels du timer par seconde.

In_Retrace : boolean = FALSE;

Cette variable reçoit la variable TRUE lorsque la procédure mod_waittrace est active. C'est indispensable afin de désactiver la procédure de mixage, quand il y a une synchronisation avec le balayage vertical. Si la procédure n'était pas désactivée, elle serait appelée pendant le temps d'attente, ce qui rendrait impossible la synchronisation.

dsp_irq : byte = $5;

Dans la variable dsp_irq, on trouve le numéro de l'interruption SB. Celle-ci doit être détournée sur une routine de votre crû pour être traitée. Lorsque la reconnaissance de l'interruption a été désactivée, vous pouvez définir une interruption dans laquelle vous affecterez la valeur souhaitée à la variable.

dma_ch : byte = 1;

Vous trouverez ici le numéro du canal DMA utilisé. Si la carte Sound Blaster utilise un autre canal que le 1, vous devez modifier cette variable !

dsp_adr : word = $220;

La variable "dsp_adr" contient l'adresse de base de la carte Sound Blaster. Elle est initialisée automatiquement. Si vous désactivez le processus d'initialisation de l'adresse de base, vous devez lui affecter la valeur correcte à la main.

SbVersMin,SbVersMaj : BYTE = 0;

Dans ces deux variables, la routine de détection dépose le numéro de version de la carte Sound Blaster. Dans SbVersMaj, on trouve le numéro principal de

version. Dans SbVersMin, on trouve le numéro secondaire. Le plus important est sans aucun doute le numéro principal, puisqu'il sert à identifier la carte.

SbRegDetected : Boolean = FALSE;

Cette variable est modifiée par la procédure d'initialisation. Lorsqu'elle prend la valeur TRUE, cela veut dire qu'une carte Sound Blaster a été trouvée. Sinon, vous pouvez interrompre vos autres routines SB, car il n'y a pas de carte Sound Blaster installée.

SbProDetected : Boolean = FALSE;

Cette variable est elle aussi modifiée par la procédure d'initialisation. Lorsque celle-ci a trouvé une carte SB Pro ou SB 16, la variable prend la valeur TRUE.

SB16Detected : Boolean = FALSE;

Si une carte SB16 (ASP) est installée dans l'ordinateur, la procédure d'initialisation donne à cette variable la valeur TRUE. SbProDetected reste dans ce cas elle aussi sur TRUE !

MixerDetected : Boolean = FALSE;

Les cartes Sound Blaster à partir de SB Pro sont équipées d'une puce de mixage. Dans ce cas, la variable reçoit la valeur TRUE.

Voix : integer = 4;

La variable "Voix" contient le nombre de voix utilisées à l'intérieur du fichier MOD. Dans cette version de l'unité, cette variable peut recevoir l'une des valeurs 4 ou 8.

Modoctave : array[1..60] of word;

Les valeurs contenues dans ce tableau correspondent aux sons contenus dans le fichier MOD. Si vous voulez donc déterminer la valeur d'une note, il vous suffit d'indiquer la position de cette valeur dans le tableau.

dma_page : array[0..3] of byte;

Les canaux DMA supportés vont de 0 à 3. Cette variable est indispensable pour la programmation directe du transfert DMA. On évite ainsi l'intervention d'une autre unité.

dma_adr,dma_wc : array[0.33] of byte;

Reportez-vous à dma_page.

sb_16outputlong : word = 0;

Si vous sortez des données sur une SB16 au moyen d'un transfert DMA, il est préférable d'effectuer la sortie au moyen de la commande DMA Continue à partir du second bloc. Cela diminue les craquements dus au DMA. La méthode ne fonctionne toutefois qu'avec des blocs de données de tailles identiques. Cette variable contient à chaque fois la taille du bloc sorti en dernier lieu. Lorsque sa longueur coïncide avec celle du nouveau bloc, on peut faire intervenir la commande DMA Continue.

dern_sortie : boolean = FALSE;

Affectez à cette variable la valeur TRUE pour arrêter la sortie via DMA. La procédure d'interruption ne fait pas démarrer dans ce cas la sortie d'un nouveau bloc.

Nous avons ainsi passé en revue les constantes de notre unité de programmation. Nous allons maintenant passer aux variables. Ici aussi, nous envisagerons uniquement celles qui présentent une certaine importance. Les variables servant de compteurs ou celles qui servent à déposer des valeurs ne seront pas prises en compte.

taillebloc : word;

La taille du bloc à sortir est déposée dans cette variable. Dans le parcours d'une interruption, on compte toujours "taillebloc" données.

dsp_rdy_sb16 : boolean;

Ce flag prend la valeur TRUE lorsque le transfert des données via DMA a pris fin. Si des données sont en cours de transfert, la variable porte donc la valeur FALSE. Ne soyez pas déconcerté par la présence du suffixe "sb16". Il s'explique par l'histoire de cette unité. Il ne signifie pas que la variable convient uniquement pour une SB16. Il faut comprendre qu'elle convient aussi pour une SB16.

OldInt : pointer;

Dans la variable "OldInt" est déposée l'adresse de l'interruption Timer. De cette façon, il est possible de restaurer celle-ci à la fin du programme. De plus, vous pouvez appeler l'ancienne interruption par l'intermédiaire de cette variable.

Irqmsq : Byte;

Cette variable est indispensable pour le traitement de l'interruption. Elle permet de "masquer" et de "démasquer" l'interruption Sound Blaster.

Mixer_proc,nmw_proc,intr_proc : point;

Selon que le fichier MOD possède quatre ou huit voix, il faut appeler des procédures différentes. Pour simplifier l'accès et pour gagner du temps, on ne répète pas à chaque fois le test pour savoir quelle procédure va être appelée. On affecte une fois pour toutes l'adresse des procédures en question aux pointeurs au moment de l'initialisation. Les procédures sont alors accessibles par l'intermédiaire des pointeurs.

Frappe_notes : array[1..8] of integer;

Cette variable permet de contrôler un égalisateur (equalizer) de conception simple. Les valeurs qui figurent dans le tableau représentent la part de chaque canal. Elle est définie à chaque fois sur 500 quand une note est frappée à nouveau.

Rm : array[0..128] of pointer;

Cette variable sert à gérer les patterns. Les pointeurs sont placés sur le début de chaque pattern en mémoire.

Chanson : array[1..128] of byte;

Ce nom de variable (contrairement à certains autres) est sans doute clair pour tout le monde. Vous trouverez ici l'arrangement de la chanson. Les différents patterns seront exécutés comme ils ont été déposés ici.

Samp : array[1..64] of pointer;

Il s'agit d'un tableau de pointeurs sur les différents samples du fichier MOD dans la mémoire principale.

Sam_l : array[1..64] of word;

Ce tableau contient les longueurs des différents samples.

loop_s,loop_l : array[1..64] of word;

Ces deux tableaux contiennent d'une part les positions de départ du looping dans les voix, d'autre part les longueurs de ces loopings.

In_St : array[1..64] of byte;

Ce tableau contient les instruments actifs appartenant à chacune des voix.

nb_patt : byte;

La variable "nb_patt" contient le nombre de patterns du fichier MOD. Il ne s'agit pas de la longueur de la chanson, mais bien du nombre de patterns définis.

Sound-Boucle : byte;

Cette variable contient le nombre de parcours de la procédure de mixage par appel.

music_out : boolean;

Lorsque cette variable reçoit la valeur TRUE, la procédure d'interruption ne fait plus entendre de musique. Les calculs s'arrêtent également.

Notvol : array[1..8] of byte;

Dans la variable "Notvol" se trouvent les volumes de chaque canal. Ce volume peut prendre une valeur située entre 0 et 64.

Old_TCompteur : word;

Cette variable est nécessaire pour la synchronisation avec l'ancienne interruption du timer. Celle-ci doit être appelée 18 fois par seconde comme toujours. La variable est utilisée pour faire le compte dans la nouvelle interruption du timer, pour savoir dans combien de temps l'ancienne interruption va être appelée.

Nom_Chanson : string[20];

Cette variable contient le nom du fichier MOD.

Noms_Inst : array[1..31] of string[22];

Ce tableau contient les noms des différents instruments à l'intérieur du fichier MOD. On utilise souvent ces noms sous forme de petits messages.

Vol_Inst : array[1..31] of byte;

Ce tableau contient les volumes standard des instruments dans le fichier MOD.

ChansonLongueur : byte;

La variable contient la longueur de la chanson exprimée en patterns. L'arrangement contient un nombre d'entrées équivalent.

vocf : file;

Avec cette unité, vous pouvez aussi exécuter des fichiers VOC. Vous pouvez y accéder à l'aide de ce numéro de fichier.

voch : vocheader;

Par l'intermédiaire de la variable "voch", vous pourrez accéder au header du fichier VOC. Vous avez fait connaissance avec sa structure dans le paragraphe consacré aux types.

vblock : voiceblock;

Le bloc actif à l'intérieur du fichier VOC sera adressé par l'intermédiaire de cette variable.

SaveExitProc : pointer;

L'unité utilise une procédure Exit qui lui est propre. Celle-ci restaure les variables modifiées et libère les secteurs alloués en mémoire. Cette variable sert à déposer l'adresse de l'ancienne procédure Exit. En effet, il faut bien l'appeler elle aussi !

Portamento_Up_Voix : array[1..8] of longint;

Cette variable contient la valeur en virgule fixe d'un éventuel "portamento up".

Portamento_Do_Voix : array[1..8] of longint;

Cette variable contiendra la valeur en virgule fixe d'un "portamento down".

Mixingproc_Voix : array[1..8] of pointer;

Chaque voix a sa propre procédure de mixage. Celle-ci est appelée en relation avec l'effet actif en cours. L'adresse de la procédure par l'intermédiaire d'un pointeur simplifie considérablement l'accès à la procédure elle-même.

Voixvide : pointer;

Il s'agit d'un pointeur sur une procédure qui ne fait rien du tout. Quand il n'y a pas de son actuellement dans une voix, c'est cette procédure qui intervient comme procédure de mixage.

Effet_Voix : array[1..8] of byte;

Dans chaque voix, on peut activer un effet déterminé. Le numéro de l'effet actif est déposé dans la variable correspondant à la voix.

Longueur_Voix : array[1..8] of word;

Cette variable contient les longueurs des différentes voix. Elle intervient pour comparer les positions dans le module en assembleur.

Longueur_Loop_Voix : array[1..8] of word;

Les longueurs de loopings pour chaque voix sont déposées dans cette variable. Un looping est pris en compte quand il dépasse les 10 octets.

Start_Loop_Voix : array[1..8] of word;

Cette variable désigne à l'intérieur du sample la position à laquelle commence le looping. La position finale du looping est obtenue en additionnant les valeurs de Start_Loop et de Longueur_Loop.

Position_Voix : array[1..8] of longint;

Contient la position à l'intérieur de la voix sous forme de valeur en virgule fixe. Les 16 bits supérieurs représentent la position à l'intérieur du sample ; les 16 bits inférieurs désignent la position décimale.

Segment_Voix : array[1..8] of word;

Pour accéder au sample, on a besoin de connaître son segment d'adresse. La variable word est chargée dans le registre de segment utilisé pour accéder au sample.

Notvol_Voix : array[1..8] of word;

On utilise cette variable dans le module en assembleur pour l'accès aux volumes des différentes voix.

Incval_Voix : array[1..8] of longint;

Cette variable permet de savoir de quelle quantité il faut augmenter la position à l'intérieur du sample après chaque accès. Il s'agit ici encore d'une valeur en virgule fixe.

Nous en avons ainsi terminé avec toutes les variables importantes dans l'unité. Nous espérons que ces pages vous seront utiles et qu'elles vous aideront à mieux comprendre l'unité que nous allons vous présenter.

Contrôle du temps dans le player MOD - Routines du timer

L'exécution d'un fichier MOD est une action périodique. Il faut appeler en conséquence régulièrement les routines servant à mixer les voix et à sortir le son. La routine de sortie sera contrôlée par l'intermédiaire de l'interruption SB. On pourrait imaginer que le mixage des voix puisse se faire aussi par la même interruption. Pourtant, cela durerait trop longtemps et aboutirait à des sons inesthétiques. C'est pourquoi il faut partager le calcul du bloc en plusieurs sous-parties.

Comment peut-on être sûr, toutefois, que tous ces petits blocs seront prêts quand l'interruption SB apparaîtra ? Nous devons nous servir de l'interrup-

tion timer. Nous détournons en premier lieu l'interruption sur notre propre routine. Bien sûr, les 18 appels par seconde ne sont pas suffisants. Nous avons besoin plutôt d'une fréquence située entre 200 et 1000 interruptions à la seconde. Il faut pour cela changer la programmation de la puce timer. On le fait en calculant la valeur à transmettre à la puce par la formule :

```
Valeur:= 1993180 DIV appels_par_seconde
```

Vous devez alors transmettre au port 43h la valeur 36h. De cette façon, le port sait qu'une nouvelle valeur va être communiquée pour le timer. Vous enverrez ensuite l'octet Low, puis l'octet High de la valeur calculée au port 40h. Le timer est ainsi programmé pour sa nouvelle fréquence. Vous devez maintenant détourner la routine du timer sur votre propre handler d'interruption et c'est parti. Dans cette unité, il est utile d'initialiser encore certaines variables nécessitées par la procédure d'interruption. Cela évite les effets secondaires indésirables. Voyons alors comment se présente la procédure :

```
procedure RegleTimer(Proc : pointer; Freq : word);
var icompteur : word;
    oldv : pointer;
begin;
 asm cli end;
 icompteur := 1193180 DIV Freq;
 Port[$43] := $36;
 Port[$40] := Lo(Icompteur);
 Port[$40] := Hi(Icompteur);
 Getintvec(8,OldV);
 setintvec(OldTimerInt,OldV);
 SetIntVec(8,Proc);
 old_tCompteur := 1;
 seccompteur  := 0;
 Oldintcompteur := 0;
 asm sti end;
end;
```

Vous avez besoin du timer pour vos propres opérations, c'est vrai. Mais quand vous quittez le programme, il faut que le timer ait été rétabli dans son état primitif. Il n'est pas question de laisser des traces de votre passage. Vous devez donc en premier lieu ramener le timer à sa fréquence standard. Vous le ferez en communiquant à nouveau au timer (par l'intermédiaire de la valeur 36h au port 43h) que vous allez lui affecter une nouvelle valeur. Vous écrirez ensuite deux fois la valeur 0 dans le port 40h. De cette façon, le timer revient à son état primitif. Il ne reste alors plus qu'à lui restituer sa routine d'origine.

Sans quoi il pointerait dans le vide, ce qui conduit habituellement à un blocage total du système.

```
procedure RestaureTimer;
var oldv : pointer;
begin;
  asm cli end;
  port[$43] := $36;
  Port[$40] := 0;
  Port[$40] := 0;
  GetIntVec(OldTimerInt,OldV);
  SetIntVec(8,OldV);
  asm sti end;
end;
```

Maintenant que vous savez changer la programmation du timer, nous allons nous occuper de notre procédure timer. En premier lieu, nous devons nous soucier de l'ancienne interruption timer. Celle-ci doit comme avant être appelée 18,2 fois par seconde. Avec 1050 appels de l'interruption, cela signifie que l'ancien timer doit être appelé toutes les 58 interruptions. Vous y parviendrez à l'aide d'une comparaison avec la variable CompteurAncInt. On comptera aussi la durée de la chanson au moyen de l'interruption. En dernier lieu, nous appellerons la procédure la plus importante de l'interruption : celle qui va calculer la musique à exécuter. L'interrogation concernant le balayage est nécessaire car il n'y a pas de synchronisation possible avec celui-ci si des calculs sont effectués pendant le temps d'attente.

```
procedure NouveauTimer; interrupt;
var dr : registers;
begin;
 inc(perfcount);
 inc(Seccompteur);
 inc(Oldintcompteur);
 if Oldintcompteur = 58 then begin;
   Oldintcompteur := 0;
   intr(Oldtimerint,dr);
 end;
 if Seccompteur = timer_per_second then begin;
   Seccompteur := 0;
inc(andycount)
   inc(Secpass);
```

```
    if Secpass = 60 then begin;
      inc(Minpass);
      Secpass := 0;
    end;
  end;
  if not in_retrace then calculate_Music;
  Port[$20] := $20;
end;
```

Les routines son

Les routines servant à contrôler la carte Sound Blaster vous sont certainement familières, puisque nous les avons rencontrées dans le paragraphe consacré à la programmation de Sound Blaster. C'est pourquoi nous n'y reviendrons pas ici en détail. Nous nous contenterons d'indiquer les procédures et les fonctions les plus importantes.

procedure wr_dsp_sb16(v : byte);

Par l'intermédiaire de cette procédure, vous pouvez sortir un octet sur le port de commandes de la carte Sound Blaster.

Function SbReadByte : Byte;

Cette fonction permet de lire un octet de données sur la carte Sound Blaster.

Function Reset_Sb16 : boolean;

Par l'intermédiaire de cette fonction, vous pouvez réaliser un reset de la carte Sound Blaster. Quand le reset a été effectué, la carte renvoie la valeur TRUE, sinon la valeur FALSE. La fonction permet d'obtenir l'adresse de base de la carte.

Function Detect_reg_Sb16 : boolean;

L'obtention de l'adresse de base dont il a été question ci-dessus est menée à bien par cette fonction. Elle donne à la variable dsp_adr l'adresse de base de la carte Sound Blaster. Elle renvoie TRUE quand une adresse de base a été trouvée, sinon FALSE.

Procedure Write_Mixer(Reg,Val : Byte);

Cette procédure écrit la valeur transmise à "Val" dans le registre de mixage "Reg". La procédure ne fonctionne qu'à partir de SB Pro.

Function Read_Mixer(Reg : Byte) : byte;

Au moyen de cette fonction, vous pouvez lire une valeur dans le registre de mixage indiqué dans "Reg".

Procedure Filtre_on;

Cette procédure désactive la mise en valeur des basses à partir de la carte SB Pro.

Procedure Filtre_Mid;

Pour obtenir une sonorité normale sur les cartes à partir de SB Pro, vous pouvez utiliser cette procédure.

Procedure Filtre_out;

En appelant cette procédure, vous faites mieux entendre les sons aigus.

Procedure Set_Balance(Valeur : byte);

Au moyen de cette procédure, vous pouvez régler la balance du son sur les cartes stéréo. "Valeur" peut prendre des valeurs comprises entre 0 (tout à gauche) et 15 (tout à droite).

Procedure Set_Volume(Valeur : byte);

Vous réglez au moyen de cette procédure le volume avec lequel le fichier MOD sera exécuté. A partir de la carte SB Pro, le volume est réglé matériellement au moyen du mixer. Sur la carte SB habituelle, la régulation se fait par des moyens logiciels, à l'aide de la variable "outvolume".

Procedure Reset_Mixer;

Exactement comme pour le DSP, il faut restaurer les valeurs par défaut dans le mixer de la carte SB au moyen d'un reset. Le registre 0 de reset du mixer doit recevoir pour cela la valeur 0. La procédure s'en charge à votre place.

Function Detect_Mixer_Sb16 : boolean;

Cette fonction sert à reconnaître la puce mixer. Elle renvoie la valeur TRUE quand une puce mixer a été trouvée et FALSE dans le cas contraire. Elle est très importante car elle intervient pour distinguer les cartes SB. Lorsque la fonction renvoie la valeur TRUE, cela veut dire que votre appareil est équipé d'une carte SB Pro.

Procedure SbGetDSPVersion;

Pour distinguer entre SB Pro et SB16, vous avez besoin de connaître le numéro de version de la carte. S'il est inférieur à 4, vous avez une SB Pro. Sinon, c'est une SB 16.

Procedure Set_Timeconst_sb16(tc : byte);

Pour définir la constante timer de la Sound Blaster, vous avez besoin de cette fonction. La constante se calcule à l'aide de la formule

```
tc:=256-(1.000.000/Fréquence du sample).
```

La constante que l'on obtient ainsi doit être transmise au moyen de la commande $40. Vous transmettrez à la procédure la constante calculée.

Procedure Exit_Sb16;

Pour mettre fin au programme, il faut faire appel à cette procédure. Elle ramène l'interruption cachée SB à sa valeur précédente et elle restaure le masque initial de l'interruption.

Procedure dsp_block_sb16(dgr : word; bk : pointer; b1,b2 : boolean);

Cette fonction est destinée à vous faciliter la tâche quand il s'agit de sortir un bloc via DMA. Elle sort un bloc de la taille "gr" adressé par le pointeur "bk" via le DMA. Il se peut que vous vouliez sortir un sample ou que vous ayez au contraire besoin de la procédure pour sortir un MOD. Les valeurs à affecter aux variables b1 et Bz ne sont pas les mêmes dans ces deux cas. Si vous exécutez un MOD, il faut que b1 = TRUE et b2 = FALSE. Pour un sample ou un fichier VOC, c'est l'inverse : b1 = FALSE et b2 = TRUE.

Exécution des fichiers MOD

L'exécution des fichiers MOD est réalisée par notre unité de programmation de la manière suivante : par l'intermédiaire de la procédure periodic_on, l'interruption timer est détournée sur une routine qui vous est propre, de façon à être programmée pour 50 appels par seconde. La routine à laquelle on saute incrémente certaines variables et appelle la procédure calculate_music. Celle-ci examine si une autre routine n'est pas en cours de traitement et, si ce n'est pas le cas, elle appelle Sound_handler. Cette procédure examine si le bloc suivant à calculer a réellement besoin de calculs. Si c'est le cas, les calculs sont effectués par la procédure intr_proc. Sinon un test est réalisé, pour voir si le bloc calculé a déjà été transféré. Si c'est le cas, on peut passer au calcul d'un nouveau bloc. Pour cela, il faut cependant charger ses paramètres. On le fait par l'intermédiaire des procédures nmw_proc et Initialiser_mixage. Par l'intermédiaire de nmw_proc, on traite le fichier MOD ligne par ligne. Elle règle aussi les effets.

Vous connaissez maintenant le mode de fonctionnement général de l'unité. Il est temps de l'examiner de plus près. Pour que vous puissiez mieux suivre les éléments qui entrent dans son code source, nous allons présenter les procédures une à une, dans l'ordre où elles se présentent.

La première procédure qui nous intéressera est get_pctunel. Cette procédure permet d'obtenir à partir de la hauteur du son l'intervalle à l'intérieur du sample à exécuter. La hauteur du son est la valeur qui se trouve dans le fichier MOD. Pour obtenir cette hauteur du son, on examine un array avec toutes les hauteurs qui apparaissent, jusqu'à ce que la valeur qui convient ait été trouvée. La position à l'intérieur du tableau des hauteurs sert alors d'index dans un tableau contenant les intervalles.

```
procedure get_pctunel(hauteur : word;Var vk : longint);
{
 La procédure fournit à partir de la hauteur de son transmise
(telle qu'elle figure dans le fichier MOD) la valeur entière et
les décimales nécessaires pour la manipulation de la fréquence.
}
var nct : byte;
    trouve : boolean;
begin;
 nct := 1;
 trouve := false;
 while (nct <= 70) and not trouve do
                          { jusqu'à aboutissement }
 begin;                   { ou une valeur dans le tableau }
   if hauteur > Modoctave[nct] then
     trouve := true;
   inc(nct);
 end;
 if trouve then begin;
   vk  := Incfacts[nct-tpw+12];
 end else begin;
   vk  := 0;              { Va chercher des valeurs dans le tableau. }
 end;
end;
```

La fonction Notes_Nr se comporte de la même façon. On s'en sert pour les effets comme le portamento. Elle permet d'obtenir la position de la hauteur de son transmise à l'intérieur du tableau avec les valeurs MOD.

```
function Notes_Nr(hauteur : word) : integer;
var nct : byte;
    trouve : boolean;
begin;
```

```
  nct := 1;
  trouve := false;
  while (nct <= 70) and not trouve do
                          { Jusqu'à aboutissement }
  begin;                  { ou dernière valeur dans le tableau }
    if hauteur > Modoctave[nct] then
      trouve := true;
    inc(nct);
  end;
  if trouve then begin;
    Notes_nr := nct-1;
  end else begin;
    Notes_nr := -1;
  end;
end;
```

Nous en venons maintenant à deux procédures très importantes. Ce sont les procédures "intra_boucle_4" et intra_boucle_4_stereo". Par l'intermédiaire de ces procédures, vous réalisez le mixage des voix du fichier MOD. On définit ici d'abord la taille à calculer. Puisque nous calculons toujours précisément un bloc qui va être exécuté, nous définirons cette taille comme étant égale à "taillebloc". Comme nous travaillons avec un double tampon, il faut ensuite indiquer le tampon cible actuel. Après quoi on efface le tampon de mixage. Cette opération est indispensable pour ne pas avoir dans le tampon les restes du processus de mixage précédent. On appelle alors pour toutes les voix la procédure de mixage actuelle en assembleur. Cette procédure additionne les données sample de chacune des voix dans le tampon de mixage. Une fois que les données sample sont parvenues de cette manière au tampon de mixage, celui-ci doit encore être transféré au tampon de sortie. Il faut pour cela diviser les valeurs du tampon de mixage par le nombre de voix. C'est nécessaire car la SB ne peut sortir les données que sur 8 bits. Sur la SB 16, vous pouvez éviter cette dernière opération. Dans ce cas, les données doivent réellement être transférées sous la forme de données 16 bits par l'intermédiaire du DMA. Avant de faire écrire les données dans le tampon de sortie par la procédure, on calcule encore le volume de sortie actuel "outvolume". En diminuant progressivement le volume de sortie, vous pouvez générer un effet correct de fading-out. Voyons maintenant le code de la procédure :

```
procedure intra_boucle_4;
{
 On réalise ici le mixage des données. Le tampon
 est rempli avec les données calculées. C'est la
 version MONO de la routine.
```

```
}
begin;
  calc_size := taillebloc;
  if bsw then
    cible := pt(buffer1)
  else
    cible := pt(buffer2);
asm
    les di,mixed_data
    mov ax,8080h
    mov cx,4000
    rep stosw
    mov cx,1
@voix_loop:
    mov mixed_posi,0
    mov voix_actuelle,cx
    mov si,cx
    dec si
    shl si,2
    call dword ptr Mixingproc_voix[si]
    inc cx
    cmp cx,voix
    jbe @voix_loop
    mov mixed_posi,0
    mov cx,calc_size
@Mixed_2_blk:
    les di,mixed_data
    add di,mixed_posi
    mov ax,es:[di]
    push cx
    mov cx,shiftfacteur
    shr ax,cl
    pop cx
    add mixed_posi,2
    mov bx,cible.sgm              { Ecrit les octets dans la cible }
    mov es,bx
    mov bx,cible.ofs
    mul outvolume
```

```
    shr ax,6
    mov es:[bx],al
    inc cible.ofs
    loop @mixed_2_blk
end;
end;
```

La procédure intra_boucle_4_stereo fonctionne presque de la même façon que intra_boucle_4. Vous aurez cependant besoin ici de deux tampons de mixage, l'un pour le canal de gauche, l'autre pour le canal de droite. Le tampon mixed_data sera utilisé pour le canal de gauche et mixed_data_st pour celui de droite. Le mixage des voix s'effectue de manière analogue. La procédure nmw_proc a déjà prévu quelle voix était affectée à quel tampon. Quand vous transférez le contenu du tampon de mixage dans le tampon de sortie, notez que la carte SB a besoin d'une valeur alternativement pour le canal de gauche et pour le canal de droite. Les valeurs doivent donc être chargées alternativement d'un tampon, puis de l'autre.

```
procedure intra_boucle_4_stereo;
{
 On réalise ici le mixage des données. Le tampon
 est rempli par les données calculées. C'est la
 version stéréo de la routine.
}
begin;
  calc_size := taillebloc;
  if bsw then
    cible := pt(buffer1)
  else
    cible := pt(buffer2);
asm
    les di,mixed_data
    mov ax,8080h
    mov cx,4000
    rep stosw
    les di,mixed_data_st
    mov ax,8080h
    mov cx,4000
    rep stosw
    mov cx,1
@voix_loop:
```

```
    mov mixed_posi,0
    mov voix_actuelle,cx
    mov si,cx
    dec si
    shl si,2
    call dword ptr Mixingproc_voix[si]
    inc cx
    cmp cx,voix
    jbe @voix_loop
    mov mixed_posi,0
    mov cx,calc_size
@Mixed_2_blk:
    les di,mixed_data
    add di,mixed_posi
    mov ax,es:[di]
    push cx
    mov cx,shiftfacteur_stereo
    shr ax,cl
    pop cx
    mov bx,cible.sgm            { Ecrit les octets dans la cible }
    mov es,bx
    mov bx,cible.ofs
    mul outvolume
    shr ax,6
    mov es:[bx],al
    inc cible.ofs
    les di,mixed_data_st
    add di,mixed_posi
    mov ax,es:[di]
    push cx
    mov cx,shiftfacteur_stereo
    shr ax,cl
    pop cx
    add mixed_posi,2
    mov bx,cible.sgm            { Ecrit l'octet dans la cible}
    mov es,bx
    mov bx,cible.ofs
    mul outvolume
```

```
    shr ax,6
    mov es:[bx],al
    inc cible.ofs
    loop @mixed_2_blk
end;
end;
```

Le mixage proprement dit s'effectue dans le module en assembleur de l'unité. Un déplacement des routines est inévitable, puisqu'on a besoin du code du 386 pour le fixpoint-handling. La procédure de mixage fonctionne de la manière suivante : tout d'abord, on charge dans le registre SI le numéro de la voix actuelle. Il sert d'index dans les tableaux relatifs aux voix. L'étape suivante consiste à charger la variable calc_size dans le registre CX. Elle désigne le nombre de données à calculer.

Dans une boucle de la procédure de chargement, on examine maintenant si la voix est parvenue à son terme. Si ce n'est pas le cas, on peut commencer directement le chargement de l'octet sample. Si la voix a atteint son terme, il faut vérifier si elle est dans un loop. Si oui, il faut définir la position à l'intérieur en la fixant sur la valeur initiale de la voix. Sinon, le sample est arrivé à son terme. La procédure de mixage est alors modifiée et passe de la procédure actuelle à la procédure dummy "Voixvide". De plus, le volume pour la voix actuelle est défini sur 0 et la procédure saute à la fin de la boucle. Il n'y a plus de données à calculer.

Maintenant que l'on a réglé de cette façon l'interrogation concernant la position, on peut charger l'octet à partir du sample. Pour cela, on charge le segment du sample dans ES et le mot High de la position à l'intérieur de la voix dans BX. Nous chargeons le mot High, car la variable indiquant la position est une variable à virgule fixe. Le mot Low ne contient que les décimales de la position. Par l'intermédiaire de es:[bx], on peut alors charger l'octet du sample dans AL. Pour l'octet dans AL, il faut en premier lieu inverser le bit 7, puisque la SB ne sort que des données "unsigned" (de 0 à 255) et non pas des données signées (de -127 à 128) comme l'Amiga ou la Gravis Ultrasound. On modifie ensuite le volume de la voix. Pour cela, il suffit de multiplier l'octet dans AL par le volume qui se trouve dans Notvol_Voix. Cette variable est à chaque fois remise à 0, avant l'appel de la procédure. La valeur calculée pour l'octet est alors ajoutée au tampon de mixage (en utilisant une instruction "mov", on écraserait les données existantes !).

Il ne reste plus alors qu'à redéfinir les pointeurs. On incrémente pour cela "mixed_posi" de deux unités et l'on redéfinit la position à l'intérieur du sample en lui ajoutant la valeur en virgule fixe qui figure dans "Incval_Voix". La procédure suivante se trouve dans MODMIX.ASM :

```
public voix_normale
voix_normale proc pascal ;voix_actuelle : word;
 pusha
 mov si,voix_actuelle
 dec si
 shl si,2                          ; accès au dword
 mov cx,calc_size
@Loop_Charge:

;{ La voix est-elle parvenue à son terme ? }
  mov bx,w Longueur_Voix[si]
  sub bx,20
  cmp bx,word ptr Position_Voix[si+2]
  ja  @Voix_non_au_terme

;{ Voix à son terme, entre-t-elle dans un loop ? }
  cmp Loop_Longueur_Voix[si],10
  jae @Voix_en_boucle

;{Voix au terme et non en boucle=> on sort}
  mov eax,Voixvide
  mov Mixingproc_voix[si],eax
  mov Notvol_Voix[si],0
  jmp @fin_voix_normale

; {Paramètres pour Voix1 au début de la boucle}
@Voix_en_boucle:
  mov bx,w Loop_Start_Voix[si]
  mov word ptr Position_Voix[si + 2],bx

; {Charger un octet à partir du sample de Voix1}
@Voix_non_au_terme:
  mov bx,w Segment_Voix[si]
  mov es,bx
```

```
  mov bx,word ptr Position_Voix[si + 2]
  mov al,es:[bx]
  sub al,128
  mul b Notvol_Voix[si]
  shr ax,6
@Voix_sortie:
  les di,mixed_data
  add di,mixed_posi
  add es:[di],ax
  add mixed_posi,2

; {Redéfinir le pointeur}
  mov ebx,Incval_Voix[si]
  add dword ptr Position_Voix[si],ebx
  loop @Loop_Charge
@fin_voix_normale:
  popa
  ret
voix_normale endp
```

La procédure chargée du mixage est "calculate_music". Elle examine si une autre procédure est en cours de traitement. Si c'est le cas, la variable "mycli" contient la valeur 1. La procédure passe alors directement à la fin. Egalement dans le cas où la variable "music_out" a une valeur différente de 0, la procédure se termine prématurément. Dans les autres cas, elle appelle "Sound_handler", la procédure qui se chargera de réaliser le mixage proprement dit.

Cette procédure examine si le bloc suivant a déjà été calculé (dans ce cas, Loop_pos est plus grand que Speed) et le calcule le cas échéant. Sinon, elle examine si la procédure d'interruption SB a posé un flag, indiquant que la sortie de l'ancien bloc a déjà eu lieu. Dans ce dernier cas uniquement, on appellera en effet les procédures nmw_proc et Initialiser_mixage. Elles font en sorte que le MOD revienne une ligne en arrière. Si la variable "outfading" est sur TRUE, on diminue le volume de sortie. Dans tous les cas, on décrémente les "frappes de notes" pour réaliser des effets comme celui de l'égaliseur.

```
procedure Sound_handler;
var li : integer;
begin;
 if mycli <> 0 then exit;
```

```
  mycli := 1;
  if (Loop_pos > Speed) then begin;
    if phase_2 then begin;
      Perfcount := 0;
      asm
        call [nmw_proc]
      end;
      Initialiser_mixage;
      phase_2 := false;
      phase_1 := true;
      if outfading then
        if outvolume >= 5 then dec(outvolume,5);
      for li := 1 to 8 do
        if Frappe_notes[li] > 50 then dec(Frappe_notes[li],50);
    end;
  end else begin;
      asm call [proc_intra] end;
    Loop_pos := Speed+2;
  end;
  mycli := 0;
end;

procedure calculate_music; assembler;
asm
  cmp mycli,0
  jne  @fin_stop
  cmp  music_off,0
  jne   @fin_stop
  pusha
  call Sound_handler
  popa
  @fin_stop:
end;
```

Avant d'exécuter le mixage, vous devez toutefois affecter les valeurs correctes aux variables. Vous utiliserez pour cela, outre nmw_proc, la procédure mixe_start_4. Celle-ci permet d'obtenir d'abord pour toutes les voix les intervalles des notes à exécuter. On fournit ensuite les paramètres pour le looping avec les valeurs correspondantes. Puis on augmente ou l'on diminue en conséquence l'intervalle de la voix, lorsque l'on a affaire à un portamento.

```
procedure mixe_start_4;
var rdiff : real;
    dummy : byte;
    var li : integer;
begin;
 for li := 1 to Voix do begin;
   if note[li] <> 0 then begin;
     tonhauteur_Voix[li] := (Rm_Song[mli,li,1] and $OF)*256+
                            Rm_Song[mli,li,2];
     get_pctunel(tonhauteur_Voix[li],Incval_Voix[li]);
   end;
   ls[li] := loop_s[In_st[li]];
   ll[li] := loop_l[In_st[li]];
   if ll[li] > 30 then inl[li] := ll[li]+ls[li];
   Loop_Longueur_Voix[li] := ll[li];
   Loop_Start_Voix[li]  := ls[li];
   case effet_Voix[li] of
     1 : begin;
           inc(Incval_Voix[li],Portamento_up_Voix[li]);
         end;
     2 : begin;
           inc(Incval_Voix[li],Portamento_do_Voix[li]);
         end;
   end;
 end;
end;
```

Venons-en à la procédure chargée de continuer le mouvement à l'intérieur du fichier MOD. Elle s'occupe du traitement ligne par ligne du fichier et initialise les variables nécessitées par la procédure de mixage. Les effets sont envisagés par la procédure effect_handling. Voyons d'abord de plus près celle-ci.

Vous devrez lui transmettre en tant que paramètre le numéro de la voix à traiter. La procédure examine alors s'il y a un effet dans la voix et communique éventuellement le paramètre de l'effet. Elle examine ensuite quel est l'effet en question et elle traite celui-ci.

L'effet 01 représente le "portamento up", 02 le "portamento down". Dans les deux cas, on s'occupe d'abord de la vitesse à laquelle le portamento doit avoir lieu. On se procure ensuite la valeur d'incrémentation au départ et à l'arrivée pour chacun des tons. On divise cette valeur par le nombre de

battements par ligne (playspeed). On obtient ainsi la valeur par laquelle on devra incrémenter ou décrémenter la valeur "Inc" de chaque voix pour chaque battement.

L'effet 09 permet de régler l'offset du sample. Le calcul se fait à l'aide de la formule :

```
Position:=Valeur_effet*256
```

Avec l'effet 11, vous pouvez sauter à l'intérieur de l'arrangement. Il suffit pour cela de mettre fin au pattern actuel et de charger le numéro du nouveau pattern, auquel on veut se rendre.

L'effet suivant est très important et il intervient pratiquement dans tous les morceaux de musique. Le n° 12 est prévu pour la définition du volume dans le canal. On affecte ici à Notvol_Voix la valeur qui se trouve dans l'opérande.

L'effet 13 termine le pattern actuel et poursuit l'exécution dans le pattern suivant. Cet effet intervient lui aussi très souvent. Il suffit ici de faire venir la ligne actuelle à la fin du fichier MOD, ce qui provoque au passage suivant le chargement automatique du pattern qui vient ensuite.

Derrière l'effet 14 se cachent plusieurs effets. Les quatre bits supérieurs de l'opérande donnent ici le sous-numéro de l'effet. Les quatre bits inférieurs contiennent l'opérande proprement dit. Dans ce player, nous ne prenons en compte que le sous-effet 12. Vous verrez à la fin de ce paragraphe comment programmer les autres effets. L'effet 12 porte le nom de "Notecut" et termine l'exécution de chacune des voix. Pour y parvenir, vous devez simplement placer sur 0 tous les paramètres en relation avec l'exécution.

Le dernier effet est le numéro 15. Il sert à définir la vitesse. Si la valeur de l'opérande est 15, il s'agit d'une indication du nombre de battements par ligne (playspeed). A la variable correspondante, on peut affecter directement la valeur voulue. Si l'opérande a cependant une valeur supérieure à 15, l'indication se réfère au BPM (beats per minute). Elle influe sur la taille du bloc sample qui doit être exécuté à chaque émission. Cette taille est calculée à l'aide de la formule suivante :

```
Taille:=Speed*(Frequence_Sampling DIV (bpm*4));
```

De même, nous devons redéfinir la valeur de "Sound_Boucles" dans notre unité. Il faut en tout cas vérifier si les valeurs ainsi déterminées se situent encore dans des proportions raisonnables. En effet, lorsque le fichier MOD contient une valeur déraisonnable, votre player peut facilement déjanter, si la nouvelle taille dépasse celle de la mémoire réservée.

```
procedure effect_handling(li : integer);
var idx : word;
    Portamento_Speed : word;
    Startnote,
    endnote : word;
    startinc,
    endinc : longint;
begin;
 if Rm_Song[mli,li,3] and $0F <= 15 then begin;
   Eff[li] := 0;
   case (Rm_Song[mli,li,3] and $0F) of
      01 : begin;
             effet_Voix[li] := 1;
             Portamento_Speed := Rm_Song[mli,li,4];
             Startnote := Notes_nr(tonhauteur_Voix[li]);
             Endnote   := Startnote+Portamento_Speed;
             get_pctunel(Modoctave[Startnote],Startinc);
             get_pctunel(Modoctave[Endnote],Endinc);
             Portamento_up_Voix[li] := round((Endinc - Startinc) /
             playspeed);
           end;
      02 : begin;
             effet_Voix[li] := 2;
             Portamento_Speed := Rm_Song[mli,li,4];
             Startnote := Notes_nr(tonhauteur_Voix[li]);
             Endnote   := Startnote-Portamento_Speed;
             get_pctunel(Modoctave[Startnote],Startinc);
             get_pctunel(Modoctave[Endnote],Endinc);
             Portamento_do_Voix[li] := round((Endinc -
                                       Startinc) / playspeed);
           end;
       9 : begin;                  { Sample offset }
             Position_Voix[li] := longint(Rm_Song[mli,li,4])
                                  shl 24;
           end;
      13 : begin;
                mli := 64;
              end;
```

```
      11 : begin;
                mli := 64;
                    mlj := Rm_Song[mli,li,4];
             end;
      12 : begin;
           Notvol_Voix[li]  := Volume_notes(li);
         end;
      14 : begin;
           case (Rm_Song[mli,li,4] shr 4) of
              12 : begin;
                   inl[li]    := 0;
                   Notvol_Voix[li]   := 0;
                   inp[li]    := 0;
                   NotVol[li]     := 0;
                 end;
           end;
         end;
      15 : begin;
           idx := Rm_Song[mli,li,4];
           if idx <= $f then begin;
             Playspeed    := idx;
             Speed := Playspeed*105 div 10;
             taillebloc := Speed * Sound_Boucles;
           end else begin;
             bpm := idx;
             mod_SetLoop(Sampling_Frequence div (BPM * 4));
             Speed := Playspeed*105 div 10;
             taillebloc := Speed * Sound_Boucles;
           end;
           if taillebloc < 40 then taillebloc := 40;
           if taillebloc > 4000 then taillebloc := 4000;
              end;
    end;
  end;
end;
```

Nous en venons enfin à la procédure nmw_all_4, que l'on adresse dans le programme par l'intermédiaire du pointeur nmw_proc. Elle commence par une définition pour les procédures stéréo. On affecte ici à chaque fois deux

voix au canal de gauche et deux voix au canal de droite. Vous pouvez changer cette disposition comme bon vous semble, en appliquant simplement les procédures de mixage selon vos besoins.

La procédure fait d'abord augmenter la ligne actuelle. Si celle-ci a une valeur supérieure à 64, cela veut dire que le pattern est parvenu à son terme, puisque seules sont définies les lignes 1 à 64. Il faut donc replacer sur 1 la variable désignant la ligne actuelle. En même temps, on incrémente la position dans l'arrangement. Lorsque la nouvelle position dépasse la longueur de la chanson, cela veut dire que le fichier MOD est parvenu à la fin et l'exécution reprend à partir du début. Sinon, on détermine à partir de l'arrangement le numéro du nouveau pattern à exécuter. Celui-ci est alors chargé dans notre pattern provisoire, auquel nous accédons toujours directement avec nos procédures.

Une fois que la position dans le MOD est ainsi corrigée et que le pattern actuel a été chargé, on peut commencer le traitement des différentes voix. Pour chacune d'elles, on pose d'abord sur 0 le flag de l'effet actuel, puisque les effets se réfèrent toujours à une ligne déterminée dans le pattern. On définit ensuite la note qui se trouve dans le MOD et on la dépose dans la variable "note". Lorsque cette variable a une valeur différente de 0, une note a été frappée et nous devons poursuivre nos vérifications. Si nous sommes en mode stéréo, on détermine la procédure de mixage à utiliser pour la voix à partir du tableau de constantes "Stereoprocs". Sinon, on fait toujours intervenir la procédure en assembleur "Voix_normale".

La variable "Frappe_notes" est définie sur 500, car une nouvelle note vient d'être frappée. En interrogeant cette variable, on peut par exemple réaliser un égaliseur de facture simple.

La variable "Vo_In" (Voix Instrument) sert à accéder aux valeurs reliées à l'instrument. On lui affecte la valeur de "Note". Par son intermédiaire, on transmet ensuite à la variable "inst" le pointeur dirigé sur le sample à exécuter. Après quoi on cherche la longueur de la voix et on la dépose dans la variable "Longueur_voix". La position à l'intérieur du sample est située au début et on la définit donc avec la valeur 0. Le volume du sample s'obtient par l'intermédiaire du volume par défaut du sample. On l'écrit dans "Notvol_voix". Enfin, il reste à déposer le segment du sample correspondant à la voix dans "Segment_voix".

De cette façon, on en a fini avec le traitement de la voix. Il ne reste plus qu'à prendre en compte les effets. C'est la procédure "effect_handling", déjà envisagée, qui s'en charge.

```
procedure nmw_all_4;
const stereoprocs : array[1..8] of pointer =
  (@Voix_normale,@Voix_normale,@Voix_normale_st,
   @Voix_normale_st,
   @Voix_normale,@Voix_normale,@Voix_normale_st,
   @Voix_normale_st);
var idx : byte;
    li : integer;
begin;
   inc(mli);
   if mli > 64 then mli := 1;
   if mli = 1 then begin;
     inc(mlj);
     if mlj > Chansonlongueur then begin;
       if mloop then begin;
         mlj := 1;
         Modp := Addr(Rm_Song);
         move(pt(rm[Chanson[mlj]]),pt(Modp),1024);
       end else begin;
         asm
           call [periodic_arret]
         end;
         music_off := true;
         Mod_Fin := true;
       end;
     end else begin;
         Modp := Addr(Rm_Song);
         move(pt(rm[Chanson[mlj]]),pt(Modp),1024);
     end;
   end;
  for li := 1 to Voix do begin;
    effet_Voix[li] := 0;
    note[li] := (Rm_Song[mli,li,1] AND $F0)+((Rm_Song[mli,li,3]
                AND $F0) shr 4);
    if note[li] <> 0 then begin;
      if stereo then begin;
        Mixingproc_voix[li] := stereoprocs[li];
      end else begin;
```

```
        Mixingproc_voix[li] := @Voix_normale;
      end;
      Frappe_notes[li] := 500;
      In_St[li]  := note[li];
      inst[li]  := Ptr(pt(Samp[In_St[li]]).sgm,pt
                   (Samp[In_St[li]]).ofs);
      Longueur_Voix[li]      := sam_l[In_St[li]];
      Position_Voix[li]    := 0;
      Notvol_Voix[li]      := Vol_inst[in_St[li]];
      Segment_Voix[li]     := seg(inst[li]^);
    end;
    effect_handling(li);
  end;
end;
```

L'ensemble du traitement du fichier MOD ne sert encore à rien, tant que nous n'avons pas de fichier MOD en mémoire. Nous devons donc en tout premier lieu nous occuper de charger le fichier MOD. La procédure de chargement appelée par le programme principal réalise une interrogation indispensable avant d'appeler la procédure de chargement proprement dite, intitulée "init_Song". Cette procédure charge dans la mémoire principale le fichier MOD indiqué au moyen de la variable globale "Mod_Nom".

On commence par définir les variables correspondant aux instruments à exécuter, ainsi que l'arrangement (Song) complet, en leur donnant la valeur 0. On ouvre ensuite le fichier MOD. Si une erreur est survenue, la routine s'interrompt avec la valeur FALSE. La routine offre la possibilité de charger un fichier MOD à partir d'un grand fichier. C'est particulièrement intéressant pour les démos. Afin d'utiliser cette fonctionnalité, vous devez en premier lieu indiquer la taille du fichier MOD en octets dans la variable "Taillefimod". Si celle-ci a la valeur 0, on accepte la taille du fichier ouvert. La position de départ à l'intérieur du fichier MOD est désignée par la variable "msp" (MOD-Start-Position). Par défaut, celle-ci contient la valeur 0.

Une fois que le fichier a été ouvert correctement, on examine s'il s'agit réellement d'un fichier MOD que l'on peut traiter. Notre player est capable d'exécuter des fichiers MOD à quatre et à huit voix. Nous devons donc vérifier si le numéro d'identification est l'un des numéros propres aux fichiers MOD, soit 'M.K.', 'FLT4' ou '8CHN'. Pour les fichiers à 31 voix, le numéro d'identification se trouve à la position 1080. Lorsque ce numéro ne convient pas, vous avez probablement affaire à un fichier MOD à 15 voix. Il se peut aussi que ce ne soit pas du tout un fichier MOD, ou encore que ce soit un fichier MOD sans numéro d'identification.

Une fois que le numéro d'identification est connu, on sait combien de voix possède le fichier MOD (31ou 15). Il faut alors lire toutes les voix une par une. Ce sera fait dans la boucle de chargement des voix. Elle démarre à la position voulue à l'intérieur du fichier MOD. Elle lit dans cette position la taille du sample, sur deux octets. La taille du sample est une indication en format Word Amiga et il faut encore la convertir en format PC. Pour cela, on échange par la fonction Swap les octets Low et High de la valeur que l'on vient de lire. On multiplie ensuite la valeur obtenue par 2, puisque l'indication dans le fichier MOD se réfère aux mots (words) et non aux octets.

La variable interne "Inststart" est décrémentée de la taille de l'instrument. Au début, on lui a donné pour valeur la taille du fichier et elle sert à déterminer le nombre de patterns. En effet, ce nombre n'est pas calculé dans le fichier MOD. On l'obtient à partir de la taille totale du fichier, duquel on soustrait le header et la taille des instruments. Cette valeur doit encore être divisée par la taille des patterns, donc par 1024 ou 2048 octets. On obtient ainsi le nombre de patterns définis.

Après la longueur du sample, on lit le volume par défaut de l'instrument. Il s'agit d'un octet qui n'a pas besoin d'être converti. Les deux valeurs suivantes à lire dans le fichier ont de nouveau besoin d'être converties. Il s'agit du loop-start et de la longueur du looping, qui sont des mots en format Amiga. Il faut les convertir selon le procédé que nous avons décrit ci-dessus.

Maintenant que nous avons lu les paramètres de base pour les voix, nous devons examiner s'il s'agit d'un fichier MOD à quatre ou à huit voix. Cet examen se fait au moyen d'une comparaison du numéro d'identification MOD avec '8CHN'. S'il y a coïncidence, on a affaire à un fichier MOD à huit voix. Dans ce cas, la variable "TaillePatt" reçoit la valeur 2048 et "Voix" reçoit la valeur 8. Sinon, on se sert des valeurs 1024 et 4. Selon que le fichier doit être exécuté en stéréo ou en mono, on pose alors les pointeurs destinés aux procédures de contrôle du mixage sur la version stéréo ou mono de la routine.

On charge ensuite les samples. Il faut réserver ce faisant la place nécessaire en mémoire pour tous les instruments. Les données sample sont ensuite chargées du disque à la mémoire.

L'étape la plus importante est le chargement des patterns. Il faut connaître pour cela tout d'abord le nombre de patterns dans le fichier MOD. Dans une boucle de chargement, vous pouvez alors charger ces patterns. Notez bien qu'un pattern à huit voix exige deux fois plus de place en mémoire qu'un pattern à quatre voix. Il ne vous reste plus qu'à charger l'arrangement long de 128 octets et la longueur de la chanson. L'arrangement se trouve à la position offset 952 et la longueur de la chanson à la position 950.

Voilà, c'est fait. Le fichier MOD se trouve bien dans la mémoire. Pour être exhaustif, chargez encore le nom du fichier MOD et les noms des instruments. Ils ne sont sans doute pas indispensables pour l'exécution, mais il est préférable de les sortir dans un player. Le nom de la chanson se trouve juste au début du fichier. Les noms des samples se trouvent toujours au début des informations sur les samples. Les noms sont conservés sous la forme de chaînes terminées par 0. Cela veut dire que leur dernier caractère est #0. Ces chaînes doivent être converties par vos soins en format Pascal. Vous y parviendrez en vous servant de la fonction ConvertString. Lorsque le chargement s'est fait correctement, la fonction présentée ici renvoie la valeur TRUE. Les variables pour la ligne actuelle et pour la position dans le pattern sont toutes deux ramenées à la position initiale.

```
FUNCTION ConvertString(Source : Pointer; Size : BYTE):String;
VAR
   WorkStr : String;
BEGIN
   Move(Source^,WorkStr[1],Size);
   WorkStr[0] := CHR(Size);
   ConvertString := WorkStr;
END;

function init_Song : boolean;
const idt1 : string = 'FLT4';
      idt2 : string = 'M.K.';
      idt3 : string = '8CHN';
var rmod : file;
    sgr : word;                 { taille d'un sample }
    instStart : longint;        { Position où démarrent les données
                                  sample dans le fichier }
    t_fichier : longint;        { La taille du fichier MOD }
    Mkg : array[1..4] of char;  { pour reconnaître le type de mode m}
    helpsp : ^byte;
    strptr : pointer;
    idtch : array[1..4] of char;
    idtstr : string;
    instance : byte;
    idx : integer;
begin;
 In_St[1] := 0;
 In_St[2] := 0;
```

```
In_St[3] := 0;
In_St[4] := 0;
In_St[5] := 0;
In_St[6] := 0;
In_St[7] := 0;
In_St[8] := 0;
for mlj := 0 to 128 do
  Chanson[mlj] := 0;
{$I-}
assign(rmod,Mod_Name);
reset(rmod,1);
{$I+}
if IOresult <> 0 then begin;
  init_song := false;
  exit;
end;
if taillefimod <> 0 then t_fichier := taillefimod else
t_fichier := filesize(rmod);
instStart := t_fichier;
seek(rmod,1080);
blockread(rmod,idtch,4);
idtstr := idtch;
if (idtstr <> idt1) and (idtstr <> idt2)
and (idtstr <> idt3) then begin;
  instance := 15;
end else begin;
  instance := 31;
end;
if instance = 31 then begin;     { 31 voix obtenus par leurs numéros
                                   d'identité}
for mlj := 1 to 31 do begin;
 idx := mlj;
 seek(rmod,msp+42+(idx-1)*30);
 blockread(rmod,sgr,2);
 sgr := swap(sgr) * 2;
 if sgr <> 0 then instStart := instStart - sgr;
 Sam_l[idx] := sgr;
 seek(rmod,msp+45+(idx-1)*30);
```

```
 blockread(rmod,vol_inst[idx],1);
 blockread(rmod,loop_s[idx],2);
 blockread(rmod,loop_l[idx],2);
 loop_s[idx] := swap(loop_s[idx])*2;
 loop_l[idx] := swap(loop_l[idx])*2;
end;
seek(rmod,msp+1080);
blockread(rmod,Mkg,4);
if pos('8CHN',Mkg) <> 0 then begin;
  Taille_patt    := 2048;
  Voix        := 8;
  shiftfacteur        := 3;
  shiftfacteur_stereo := 3;
end else begin;
  Taille_patt    := 1024;
  Voix        := 4;
  shiftfacteur        := 2;
  shiftfacteur_stereo := 2;
end;
Mixer_Proc := @mixe_start_4;
nmw_Proc         := @nmw_all_4;
if stereo then
  proc_intra     := @intra_boucle_4_stereo
else
  proc_intra     := @intra_boucle_4;
seek(rmod,msp+instst art);
for mlj := 1 to 31 do begin;
 idx := mlj;
 getmem(Samp[idx],Sam_l[idx]);
 blockread(rmod,Samp[idx]^,sam_l[idx]);
end;
t_fichier := instst art - 1083;
nb_patt := t_fichier div Taille_patt;
for mlj := 0 to nb_patt-1 do begin;
  getmem(rm[mlj],2048);
  fillchar(rm[mlj]^,2048,0);
  seek(rmod,msp+1084+mlj*Taille_patt);
  helpsp := ptr(seg(rm[mlj]^),ofs(rm[mlj]^));
```

```
    for mli := 0 to 63 do begin;
      helpsp := ptr(seg(rm[mlj]^),ofs(rm[mlj]^)+mli*32);
        blockread(rmod,helpsp^,Taille_patt div 64);
    end;
  end;
  seek(rmod,msp+952);
  blockread(rmod,Chanson,128);
  getmem(strptr,25);
  for i := 0 to 30 do begin;
    seek(rmod,msp+20+i*30);
    blockread(rmod,strptr^,22);
    noms_inst[i+1] := convertstring(strptr,22);
  end;
  seek(rmod,msp);
  blockread(rmod,strptr^,20);
  Nom_Chanson := convertstring(strptr,20);
  freemem(strptr,25);
  seek(rmod,msp+950);                { à partir de 470}
  blockread(rmod,Chansonlongueur,1);
  end else begin;
  for mlj := 1 to 15 do begin;
   seek(rmod,msp+42+(mlj-1)*30);
   blockread(rmod,sgr,2);
   sgr := swap(sgr) * 2;
   if sgr <> 0 then inststart := inststart - sgr;
   Sam_l[mlj] := sgr;
   seek(rmod,msp+45+(mlj-1)*30);
   blockread(rmod,vol_inst[mlj],1);
   blockread(rmod,loop_s[mlj],2);
   blockread(rmod,loop_l[mlj],2);
   loop_s[mlj] := swap(loop_s[mlj])*2;
   loop_l[mlj] := swap(loop_l[mlj])*2;
  end;
  for mlj := 16 to 31 do begin;
   Sam_l[mlj] := 0;
   loop_s[mlj] := 0;
   loop_l[mlj] := 0;
  end;
```

```
if pos('8CHN',Mkg) <> 0 then begin;
  Taille_patt    := 2048;
  Voix           := 8;
  shiftfacteur        := 3;
  shiftfacteur_stereo := 3;
end else begin;
                                  { Fichier MOD à quatre voix }
  Taille_patt    := 1024;
  Voix        := 4;
  shiftfacteur        := 2;
  shiftfacteur_stereo := 2;
end;
Mixer_Proc := @mixe_start_4;
nmw_Proc       := @nmw_all_4;
if stereo then
  proc_intra     := @intra_boucle_4_stereo
else
  proc_intra     := @intra_boucle_4;
seek(rmod,msp+inststart);
for mlj := 1 to 15 do begin;
  getmem(Samp[mlj],Sam_l[mlj]);
  blockread(rmod,Samp[mlj]^,sam_l[mlj]);
end;
t_fichier := inststart - 603;
nb_patt := t_fichier div Taille_patt;
for mlj := 0 to nb_patt-1 do begin;
  getmem(rm[mlj],2048);
  fillchar(rm[mlj]^,2048,0);
  seek(rmod,msp+1084+mlj*Taille_patt);
  helpsp := ptr(seg(rm[mlj]^),ofs(rm[mlj]^));
  for mli := 0 to 63 do begin;
    helpsp := ptr(seg(rm[mlj]^),ofs(rm[mlj]^)+mli*32);
      blockread(rmod,helpsp^,Taille_patt div 64);
  end;
end;
seek(rmod,msp+472);
blockread(rmod,Chanson,128);
getmem(strptr,25);
```

```
  for i := 0 to 14 do begin;
    seek(rmod,msp+20+i*30);
    blockread(rmod,strptr^,22);
    noms_inst[i+1] := convertstring(strptr,22);
  end;
  for i := 15 to 30 do begin;
    noms_inst[i+1] := '';
  end;
  seek(rmod,msp);
  blockread(rmod,strptr^,20);
  Nom_Chanson := convertstring(strptr,20);
  freemem(strptr,25);
  seek(rmod,msp+470);
  blockread(rmod,Chansonlongueur,1);
  end;
  mlj := 0;
  mli := 0;
  close(rmod);
  init_song := true;
end;
```

Le chargement du fichier MOD ne se limite pourtant pas à cela. Il faut encore initialiser toute une série d'autres variables, tâche dont la fonction "charge_fichiermod" s'acquitte fort bien à votre place. Vous devrez lui transmettre comme premier paramètre le nom du fichier MOD que vous voulez charger, avec son chemin d'accès complet si c'est nécessaire. Pour le second et le troisième paramètre, indiquez simplement AUTO. Le dernier paramètre définit la fréquence de sortie. Si vous ne disposez pas d'un 386SX à 16 MHz, vous pouvez indiquer ici sans vous inquiéter la valeur 22. Le fichier MOD est alors reproduit sur une fréquence de 22 KHz. Sinon, vous obtenez des résultats satisfaisants même en indiquant la valeur 16 (MHz).

La fonction vérifie d'abord si le nom du fichier est valide. Si ce n'est pas le cas, vous déclenchez le message d'erreur -1. Elle vérifie ensuite si la sortie doit se faire en mono (force_mono=TRUE). Dans ce cas la variable "Stereo" reçoit la valeur FALSE. Lorsqu'une SBPro est installée dans l'ordinateur, il faut activer le mode correct de sortie sur la carte. Il n'en va ainsi que sur la SBPro. Vous n'incriminerez pour cela que les développeurs de Creative Labs. Une fois que les opérations relatives à la stéréo ont été réalisées, il ne vous reste plus qu'à initialiser certaines données. La procédure init_data s'en charge pour vous.

Par l'intermédiaire de "init_song", vous chargez alors le fichier MOD dans la mémoire. Vous définissez ensuite la fréquence de sortie souhaitée et la vitesse de sortie standard. Il faut alors envoyer le facteur de sampling voulu à la carte Sound Blaster et activer le haut-parleur. Quand vous disposez d'une Sound Blaster avec des capacités stéréo, la balance du son à la sortie est placée sur le milieu et le volume de sortie est réglé sur le volume master. La fonction se termine en retournant le résultat 0.

```
function charge_fichiermod(modnom : string;ispeed,iloop :
integer;freq :
byte) : integer;
var df : file;
    sterreg : byte;
    t_f : longint;
begin;
 PLAYING_MOD := true;
 PLAYING_VOC := false;
 outfading := false;
 outvolume := 63;
 Mod_Nom := modnom;
 {$I-}
 assign(df,Mod_nom);
 reset(df,1);
 {$I+}
 if IOResult <> 0 then begin;
   {$I-}
   close(df);
   charge_fichiermod := -1;       { Fichier non trouvé !      }
   exit;
 end;
 {$I-}
 t_f := filesize(df);
 close(df);
 music_played := true;
 music_off := false;
 Mod_fin := false;
 if ispeed <> AUTO then Speed3 := ispeed;
 if iloop <> AUTO then Loop3  := iloop;
 if force_mono then stereo := false;
 if force_sb then begin;
```

```
    if Sb16Detected then stereo := false;
    Sb16Detected := false;
  end;
  if SBProdetected then begin;
    if stereo then begin;
      sterreg := Read_Mixer($0e);
      write_Mixer($0e,sterreg OR 2);
    end else begin;
      sterreg := Read_Mixer($0e);
      write_Mixer($0e,sterreg AND $FD);
    end;
  end;
  init_data;
  if init_song then begin;
    phase_1 := false;
    phase_2 := true;
    mycli := 0;
    mod_Samplefreq(freq);
    Playspeed    := 6;
    Speed := Playspeed*105 div 10;
    bpm := 125;
    mod_SetLoop(Sampling_Frequence div (BPM * 4));
    taillebloc := Speed * Sound_Boucle;
    if taillebloc < 100 then taillebloc := 100;
    if taillebloc > 4000 then taillebloc := 4000;
    asm call [nmw_proc] end;
    set_timeconst_sb16(Sampling_Rate);
    Initialiser_mixage;
    Secpass := 0;
    Minpass := 0;
    wr_dsp_sb16($D1);
    if sb16detected or sbprodetected then begin;
       Filtre_Mid;
      Set_Balance(Balance);
      Set_Volume(Mastervolume);
    end;
    Charge_fichiermod := 0;
  end else begin;
```

```
    Charge_fichiermod := -3;    { Erreur au chargement de la chanson }
  end;
end;
```

En dernier lieu, vous devez maintenant apprendre à démarrer le fichier MOD. L'appel ne présente pas de problème. Il se fait par l'intermédiaire de la procédure "periodic_on". L'exécution de fichier MOD commence dès que vous appelez cette procédure. Celle-ci pose pour cela le flag "dern_sortie" sur FALSE, puisque nous ne faisons que commencer la sortie des données et que nous n'entendons pas y mettre fin. Il s'agit ensuite de sortir par le DMA le bloc calculé dans la procédure de chargement. Par l'intermédiaire d'une inversion de la variable booléenne "bsw", passez au bloc suivant du double tampon. Il faut encore charger les valeurs destinées à la ligne suivante, ce que vous ferez à nouveau à l'aide des procédures "nmw_proc" et "Initialiser_mixage". La dernière opération est pratiquement la plus importante : l'interruption timer doit être détournée sur une procédure qui vous est propre et sa fréquence doit être ainsi redéfinie. Vous le ferez en vous servant de la procédure "ActiveTimer". Celle-ci doit recevoir comme premier paramètre la procédure d'interruption "NouveauTimer", que nous allons examiner d'un peu plus près dans un instant. Le second paramètre sera la fréquence d'interruption par seconde. Vous devez indiquer ici la constante "timer_per_second", qui a la valeur 50. C'est tout ce que vous avez à faire. Le fichier MOD se laissera faire sans rechigner quand vous appellerez la procédure.

```
Procedure periodic_on;
Begin
 dern_sortie := false;
{ for Loop_pos := 1 to Speed do begin;}
   asm call [proc_intra] end;
{ end;}
 dsp_block_sb16(taillebloc,taillebloc,buffer1,true,false);
 bsw := not bsw;
 Loop_pos := 0;
 asm
   call [nmw_proc]
 end;
 Initialiser_mixage;
 init_sbperiod(@periodic_off);
 music_played := true;
 RegleTimer(@NouveauTimer,timer_per_second);
End;
```

Vous venez de lancer le fichier MOD. Fort bien. Mais il faut bien tout de même que vous vous donniez les moyens d'entendre l'exécution. C'est aussi simple à réaliser que le démarrage du fichier MOD. Il suffit pour cela d'appeler la procédure periodic_off. Aussitôt, vous entendrez les sons produits par le fichier MOD. La procédure affecte à la variable "dern_sortie" la valeur TRUE, ce qui a pour effet d'interdire la sortie d'un nouveau bloc dans la procédure d'interruption. Ensuite, vous désactiverez l'interruption timer. Cela veut dire que vous devez restaurer ses anciennes valeurs. C'est la procédure "RestaureTimer" qui s'en charge.

```
Procedure periodic_off;
Begin
  dern_sortie := true;
  RestaureTimer;
End;
```

Vous êtes désormais en mesure d'exécuter des fichiers MOD avec cette unité. Nous allons parler maintenant de la sortie des fichiers VOC. Pour cela, vous avez besoin en tout et pour tout de deux procédures : l'une d'elles est Init_Voc, qui permet de lancer la sortie, l'autre est Voc_Done, qui permet d'y mettre fin. Voyons d'abord de plus près la première.

Vous devez transmettre à cette procédure en tant que premier paramètre le nom du fichier VOC à exécuter, avec le chemin d'accès complet si c'est nécessaire. La procédure pose d'abord les flags Playing_MOD sur FALSE et Playing_VOC sur TRUE. De cette façon, elle fait comprendre à la procédure d'interruption qu'un fichier VOC va être exécuté maintenant et non pas un fichier MOD. La sortie stéréo est désactivée par défaut. Un fichier VOC stéréo la réactive. On ouvre ensuite le fichier VOC et on lit le header. Si la caractérisation contenue dans le header n'est pas 'Creative Voice File'+#1A, vous n'avez pas affaire à un fichier VOC correct et peut-être pas à un fichier VOC du tout.

Vous devez lire ensuite le premier bloc Voice du fichier VOC. Son identification se trouve en position 2 du tableau que vous lisez. Il existe trois blocs Voice susceptibles de nous intéresser. Le premier est le bloc 1. C'est le type le plus simple. On peut y lire directement le facteur sample du bloc. Celui-ci se trouve en position et on peut l'envoyer directement au DSP.

Le type 8 est beaucoup moins commode. Les développeurs géniaux ont encore frappé. En effet, au lieu de déposer ici la fréquence originale du sample, ils ont étendu la constante de sampling interne de 8 à 16 bits. Il faut donc commencer par lire ce nouveau facteur de sampling, puis calculer la fréquence originale du sample au moyen de la formule :

```
Fréquence:=256000000 DIV (65536 - SR)
```

Il faut ensuite vérifier s'il y a un sample stéréo. Si c'est le cas, la sortie stéréo doit à nouveau être activée et la fréquence de sampling être divisée par deux. A partir de la fréquence de sampling ainsi obtenue, on calcule alors la fréquence de sampling à transmettre à la carte Sound Blaster, en se servant cette fois de la formule :

```
SB_SR:=256 - (10000000 DIV (Fréquence de sampling))
```

Il faut enfin mentionner le type de bloc Voice numéro 9. Ce type de bloc Voice a été introduit avec la SB16. Il semble que Creative suive là le mot d'ordre : "Un nouveau format à chaque nouvelle carte". Quant à la compatibilité... Avec ce format, on a enfin la fréquence de sampling originale du fichier. Elle se trouve dans les octets 6 et 7. L'octet 11 contient la valeur 2 quand il existe un sample stéréo. Dans ce cas, il faut encore une fois commencer par activer la sortie stéréo. Après quoi, il faut vérifier si la carte est une SBPro (et seulement une SBPro). Il faut dans ce cas multiplier la fréquence de sampling. Sinon, elle peut être adoptée telle quelle. La fréquence à communiquer à la carte Sound Blaster sera calculée ici encore à l'aide de la formule :

```
SB_SR:=256 - (10000000 DIV (Fréquence de sampling))
```

Avec un bloc MONO, on peut calculer la fréquence directement au moyen de cette formule.

Nous avons ainsi dégagé du fichier VOC à la force du poignet la fréquence de sampling à utiliser. Nous devons maintenant la communiquer à la carte Sound Blaster. C'est la procédure "set_timeconst_sb16" qui s'en chargera pour nous. Il faut juste lui transmettre la fréquence de sampling que nous venons de déterminer.

L'opération suivante consiste à charger à partir du fichier VOC le premier bloc contenant des données sample. Un bloc a ici une taille standard de 2500 octets. Le premier bloc sera chargé dans Buffer1. Lorsque la taille du fichier restant est supérieure à la taille du tampon, on charge aussi le second bloc, cette fois dans Buffer2.

Ce n'est qu'ensuite qu'il faut activer le haut-parleur. Il faut en effet attendre le moment où l'on sort un fichier VOC, car rien ne peut sortir d'un haut-parleur activé. S'il n'y a pas de carte SB16 installée dans l'ordinateur, il faut changer la programmation de la puce de mixage pour la sortie mono et stéréo. Maintenant que nous avons surmonté cet obstacle supplémentaire dressé sur nos pas par Creative Labs, on peut lancer la sortie du bloc au moyen de la procédure dsp_block_sb16. Tout le reste est pris en charge par l'interruption SB.

Pour mettre fin à la sortie du fichier VOC, appelez la procédure Voc_Done. Elle affecte la valeur TRUE à la variable "lastone" et fait savoir à la procédure d'interruption qu'il n'y a pas d'autre bloc à sortir. On peut alors refermer le

fichier VOC et effectuer un reset de la carte Sound Blaster. Le reset est important pour que d'autres routines SB puissent trouver la carte dans son état original.

```
procedure Init_Voc(filename : string);
const VOCidt : string = 'Creative Voice File'+#$1A;
var ch : char;
    idtstr : string;
    ct : byte;
    h : byte;
    error : integer;
    srlo,srhi : byte;
    SR : word;
    Samplingr : word;
    stereoreg : byte;
begin;
  PLAYING_MOD := false;
  PLAYING_VOC := true;
  VOC_READY   := false;
  vocsstereo := stereo;
  stereo := false;
  assign(vocf,filename);
  reset(vocf,1);
  if filesize(vocf) < 5000 then begin;
    VOC_READY   := true;
    exit;
  end;
  blockread(vocf,voch,$19);
  idtstr := voch.ident;
  if idtstr <> VOCidt then begin;
    VOC_READY   := true;
    exit;
  end;
  Blockread(vocf,inread,20);
  vblock.Ident_Code := inread[2];
  if vblock.Ident_Code = 1 then begin;
    vblock.SR := inread[6];
  end;
  if vblock.Ident_Code = 8 then begin;
```

```
    SR := inread[6]+(inread[7]*256);
    Samplingr := 256000000 div (65536 - SR);
    if inread[9] = 1 then begin; {stereo}
      if sb16detected then samplingr := samplingr shr 1;
      stereo := true;
    end;
    vblock.SR := 256 - longint(1000000 DIV samplingr);
  end;
  if vblock.Ident_Code = 9 then begin;
    Samplingr := inread[6]+(inread[7]*256);
    if inread[11] = 2 then begin;{stereo}
      stereo := true;
      if sbprodetected then samplingr := samplingr * 2;
      vblock.SR := 256 - longint(1000000 DIV (samplingr));
    end else begin;
      vblock.SR := 256 - longint(1000000 DIV samplingr);
    end;
  end;
  if vblock.SR < 130 then vblock.SR := 166;
  set_timeconst_sb16(vblock.SR);
  t_bloc := filesize(vocf) - 31;
  if t_bloc > 2500 then t_bloc := 2500;
  blockread(vocf,buffer1^,t_bloc);
  ch := #0;
  t_f := filesize(vocf) - 32;
  t_f := t_f - t_bloc;
  Bloc_actif := 1;
  if t_f > 1 then begin;
    blockread(vocf,buffer2^,t_bloc);
    t_f := t_f - t_bloc;
  end;
  wr_dsp_sb16($D1);
lastone := false;
if not sb16Detected then begin;
  if Stereo then begin;
    stereoreg := Read_Mixer($0E);
    stereoreg := stereoreg OR 2;
    Write_Mixer($0E,stereoreg);
```

```
    end else begin;
      stereoreg := Read_Mixer($0E);
      stereoreg := stereoreg AND $FD;
      Write_Mixer($0E,stereoreg);
    end;
  end;
  dsp_block_sb16(t_bloc,t_bloc,buffer1,false,true);
end;

procedure voc_done;
var h : byte;
begin;
  lastone := true;
{ repeat until dsp_rdy_sb16;}
  close(vocf);
  Reset_Sb16;
  stereo := vocsstereo;
end;
```

Pour finir, venons-en à une procédure absolument essentielle, la procédure d'interruption SB. On passe toujours à celle-ci quand la Sound Blaster annonce qu'un transfert DMA a pris fin. La procédure examine d'abord s'il s'agit d'un "interrupt-check" pour la reconnaissance de la carte au niveau matériel. Dans ce cas, le flag IRQDetected est posé sur TRUE. La routine de détection sait alors que l'interruption correcte a été trouvée.

S'il s'agit au contraire d'une interruption normale, on distingue deux cas : l'unité peut être occupée par la sortie d'un fichier MOD ou d'un fichier VOC. Quand c'est un fichier MOD qui est en cours de traitement, on lira d'abord un octet dans le port de données de la SB. C'est nécessaire pour accéder à la carte. On affecte ensuite à la variable booléenne dsp_rdy_sb16 la valeur TRUE. Cela indique que le bloc est parvenu à son terme. On vérifie ensuite si la sortie doit être poursuivie. C'est le cas lorsque "dern_sortie" contient la valeur FALSE. On lance alors la sortie du bloc actif par l'intermédiaire d'un appel de dsp_block_sb16. En dernier lieu, on pose un flag indiquant que la procédure timer peut faire passer le fichier MOD à la ligne suivante.

Lorsque l'on a affaire à un fichier VOC, le procédé est un peu différent. Ici aussi, il faut d'abord accéder en lecture à la carte Sound Blaster. Mais l'opération consiste cette fois à vérifier si le fichier VOC n'est pas parvenu à son terme et s'il faut encore procéder aux opérations de sortie. Si c'est le cas, on envoie le bloc actif à la carte par l'intermédiaire de la procédure dsp_block_sb16. A la suite de quoi on charge dans l'autre tampon de

nouvelles données sample en provenance du fichier VOC. On fait alors passer le bloc passif à l'état actif et vice versa. Si le fichier VOC est parvenu à son terme ou s'il n'y a pas d'autre sortie souhaitée, on pose dsp_rdy_sb16 sur TRUE et on désactive le haut-parleur. Il faut ensuite affecter à la variable booléenne Voc_Ready la valeur TRUE. Notez bien que vous devez écrire la valeur 20h sur le port 20h au terme de la procédure d'interruption, car sinon aucune autre interruption ne pourra être exécutée.

Conseils pour la programmation des effets

Dans l'unité que nous venons de vous présenter, nous n'avons pas implémenté tous les effets Protracker. Les effets intégrés suffisent pour reproduire correctement environ 80% de tous les fichiers MOD. Si vous voulez prendre en compte les effets restants ou écrire votre propre player, vous vous trouverez devant le problème suivant : comment fait-on pour programmer les effets ?

Puisque les informations à ce sujet ne courent pas les rues, nous allons présenter ici tous les effets, en indiquant à chaque fois brièvement comment il faut les traiter. En parlant ici du paramètre X, nous désignerons les quatre bits supérieurs de l'opérande de l'effet. Le paramètre Y désigne au contraire les quatre bits inférieurs de l'opérande. XY représente l'opérande en son entier. Passons en revue les effets tels qu'ils sont numérotés :

Arpeggio :

La note est jouée successivement sur trois tons différents. Les tons sont les suivants : d'abord le ton original, puis ce ton + le paramètre X, enfin ce ton + le paramètre Y. Vous pouvez activer pour chaque battement le ton suivant et vous pouvez alors calculer normalement la voix.

Portamento Up/Portamento down :

L'effet Portamento est lui aussi relativement simple à programmer. A chaque battement, la valeur indiquée dans XY est soustraite de l'intervalle ou lui est au contraire ajoutée. Les calculs se font pour le reste tout à fait normalement.

Portamento to Note :

L'effet "Portamento to Note" exige un effort plus conséquent, mais il fonctionne en fait selon le même principe que le "Portamento up/down". Lorsque vous utilisez cet effet, la note indiquée dans la ligne n'est pas comprise comme la note à jouer, mais comme la note cible du portamento. Il faut d'abord vérifier si l'on doit utiliser un portamento up ou down. Il faut ensuite augmenter ou diminuer à chaque battement l'intervalle de la note d'une certaine valeur, indiquée dans XY. XY fournit ainsi la vitesse du portamento. Celui-ci est exécuté jusqu'à ce que l'on atteigne la note cible. Lorsque celle-ci n'a pas été atteinte à l'intérieur d'une ligne, le portamento se poursuit dans la ligne suivante, à condition qu'il n'y ait pas d'autre effet indiqué dans

celle-ci. Si le tableau des notes est vide dans cette ligne, la note cible est celle qui est indiquée en dernier lieu. Sinon, c'est celle qui se trouve dans la ligne en question.

Vibrato :

L'effet "vibrato" est assez complexe. On a besoin de trois tableaux contenant les trois formes possibles de sinusoïdes. Ces formes sont les suivantes : en premier lieu une sinusoïde tout à fait normale ; en second lieu, une sinusoïde rectangulaire ; enfin une sinusoïde en dents de scie, appelée "ramp down". Les entrées dans ces tableaux doivent se situer entre -255 et 255. Une demi-oscillation, c'est-à-dire l'intervalle qui se trouve entre deux intersections de la courbe des oscillations avec l'axe des Y, doit comporter 32 entrées. L'oscillation entière comprend ainsi 64 entrées. A chaque battement, vous devez augmenter la position de l'index à l'intérieur du tableau d'un paramètre x. La valeur obtenue dans le tableau par l'intermédiaire de l'index sera multipliée par le paramètre y et divisée par 256. Cette nouvelle valeur sera ajoutée alors à l'intervalle de la note. Lorsque le champ xy contient la valeur 0, vous devez utiliser la valeur qui intervenait en dernier lieu pour le vibrato. Celui-ci dure jusqu'à ce que vous "frappiez" une nouvelle note. Notez que vous n'effectuez pas ici de reset en passant à la ligne suivante dans la chanson.

Portamento et Volume sliding :

Cette combinaison des deux effets peut être facilement séparée en deux, chacun des effets étant alors traité à part. Vous devez d'abord à chaque battement diminuer le volume de la voix, puis modifier le ton de la voix comme nous l'avons expliqué plus haut.

Vibrato et Volume sliding :

Avec cette combinaison, le vibrato se poursuit et le volume de la voix est diminué ou élevé en conséquence.

Tremolo :

Cet effet est très proche de l'effet vibrato. La seule différence est qu'on ne modifie pas ici la vitesse d'exécution des voix, mais leur volume. Cet effet est souvent désigné comme un vibrato du volume et c'est une méthode tout à fait légitime pour remplacer le vibrato. Vous pouvez utiliser le même procédé que pour ce dernier et aussi les mêmes tableaux. Veillez seulement, lorsque vous modifiez les volumes, à ne pas dépasser les valeurs maximales.

Volume sliding :

Le "volume sliding" peut être réalisé vers le haut ou le bas. Lorsqu'on utilise le paramètre x, il s'agit d'une augmentation du volume. Avec le paramètre y, il s'agit au contraire d'une diminution. A chaque battement, on modifie le volume de la valeur indiquée par le paramètre, jusqu'à ce que l'on parvienne

à l'extrémité de la ligne ou jusqu'à ce que l'on atteigne une valeur fixée à l'avance.

Position Jump :

Cet effet est très simple à traiter. Il consiste à sauter à une position indiquée par xy à l'intérieur de l'arrangement. Cela ne devrait poser aucun problème, puisque vous devez simplement passer à la fin du pattern actuel et affecter alors à la variable contenant la position dans l'arrangement la valeur fournie par xy.

Set Volume :

La signification de cet effet a conduit à de grandes discussions dans les milieux spécialisés. Donnez simplement au volume défini pour la voix actuelle la valeur transmise dans XY. C'est tout ce qu'il y a à faire.

Pattern Block :

Cet effet est lui aussi très simple à réaliser. Il n'est pourtant pas exécuté correctement par tous les players MOD (par exemple DMP 2.92). Vous devez d'abord mettre fin au pattern actuel. En général, il n'y a rien à redire à ce sujet. C'est cependant la signification du paramètre xy qui est ignorée. Normalement, celui-ci contient la valeur 0 (dans 90% des MOD). S'il ne contient pas cette valeur, il désigne dans le pattern suivant la ligne à laquelle il faut commencer. Il s'agit donc non pas de sauter au début du pattern, mais bel et bien à cette ligne.

Set Speed :

Cet effet sert à définir la vitesse d'exécution du MOD. La valeur standard est de 125 Beats per Minute (BPM) pour une vitesse de 6 battements. Si xy correspond à une valeur inférieure à 0Fh, l'indication fournie est cependant celle de la vitesse, c'est-à-dire précisément du nombre de battements. Sinon, la valeur donne le nombre de BPM.

C'est tout pour les effets normaux. Venons-en enfin à certains effets étendus. Ils représentent vraiment le "fin du fin", mais il n'y a aucune raison de ne pas les faire figurer dans un player top-niveau.

Set Filtre :

C'est le plus simple à programmer parmi ces effets. Puisqu'il est spécifique à Amiga, vous vous contenterez de l'ignorer !

Fine Portamento up/down

Cet effet fonctionne exactement comme un portamento normal. La seule différence est que le ton est modifié non pas à chaque battement, mais en une seule fois au moment de l'initialisation.

Glissando :

Vous pouvez programmer le glissando de la manière suivante : générez un tableau contenant toutes les notes possibles. Exécutez alors votre portamento tout à fait normalement. Vous devrez arrondir simplement la valeur obtenue avec la valeur suivante dans le tableau de toutes les notes.

Set Vibrato Waveform :

Cet effet vous permet de choisir l'un des trois tableaux possibles pour le vibrato. Le mieux est d'adresser le tableau actuel au moyen d'un pointeur. Pour cet effet, il vous suffira alors de poser le pointeur sur l'adresse du tableau choisi.

Set Loop :

Déposez la position actuelle (ligne) dans le pattern. Vous reviendrez alors facilement à cette position par un loop.

Jump to Loop :

"Jump to Loop" est la deuxième partie de la commande Loop. Elle permet de sauter à la position marquée par l'intermédiaire de "Set Loop". Vous répéterez cette opération aussi souvent que l'indique la valeur "y".

Set Tremolo Waveform :

On peut reprendre ici ce qui a été dit pour "Set Vibrato Waveform".

Retigger Note :

Lorsque le battement actuel est égal au paramètre y, il faut charger la position à l'intérieur du sample en la ramenant à la position de départ.

Fine Volume slide up/down :

Lors de l'initialisation, vous augmentez ou diminuez le volume de la voix de la valeur indiquée avec "y". Il n'y a pas de remise à jour pendant les battements.

Note Cut :

Par l'intermédiaire de cette commande, vous arrêtez l'exécution d'une voix. Le paramètre "y" indique ici le battement auquel la voix doit cesser d'être exécutée. Pour mettre fin à la voix, vous devez seulement modifier la position dans la voix ou la longueur de la voix elle-même.

Note Delay :

Le paramètre Y indique la valeur définissant le retard avec lequel il faut retarder la sortie de la voix. L'indication est fournie en nombre de battements.

Pattern Delay :

Cet effet est analogue au précédent. Simplement, ce n'est pas ici la sortie d'une voix unique qui est en cause, mais celle de toutes les voix. Le retard à la sortie est indiqué en nombre de battements par le paramètre Y.

Invert Loop :

Pour l'interprétation de cet effet, nous devons nous baser sur des suppositions. Supposons donc que le pattern est exécuté vers l'arrière de la position actuelle jusqu'à la position caractérisée par "Set Loop". Nous n'avons cependant trouvé aucun MOD dans lequel cette commande soit utilisée. Elle n'est admise par aucun player MOD de notre connaissance.

Le player MOD386

La mise en oeuvre de l'unité MOD_SB sera illustrée au mieux avec le petit programme MOD386. Il vous permettra d'exécuter des fichiers MOD et VOC. Si vous lancez le programme sans paramètre, vous aboutissez à un menu de sélection, dans lequel vous pouvez choisir l'un des fichiers son contenus dans le répertoire actuel. Si vous indiquez le paramètre -r, vous êtes en mode Repeat. Celui-ci vous permet, après exécution d'un fichier son, de sélectionner le fichier suivant, et cela jusqu'à ce que vous ayez appuyé sur **ESC**.

Le programme utilise l'unité "Design". Celle-ci contient des fonctions générales utilisant la technique des fenêtres et faisant intervenir la sélection des fichiers. Nous ne l'aborderons pas ici. Vous trouverez cette unité sur le CD accompagnant ce livre.

Nous allons maintenant présenter et expliquer les différentes procédures du programme.

procedure Scala_boxes;

Cette procédure dessine l'écran du player. Elle ne devrait pas poser de problème.

procedure Scala;

"Scala" est prévue pour la sortie des informations pendant le fonctionnement du programme. Elle affiche le volume sous forme de barres et met à jour les informations sur la position actuelle dans la chanson, ainsi que sur les instruments qui sont actifs en ce moment. Cette procédure, elle non plus, ne devrait pas présenter de difficultés.

procedure Play_the_Mod(s : string)

Cette procédure est la seule qui nous intéresse vraiment dans ce contexte. Elle exécute un fichier MOD et réagit aux saisies de l'utilisateur.

En premier lieu, elle effectue un reset de la carte Sound Blaster. C'est nécessaire car nous ne savons dans quel état elle se trouve. On pose ensuite quelques paramètres nécessaires pour la sortie. Pour "Samfreq", vous pouvez tranquillement définir une valeur de 22 (KHz), quand vous ne disposez pas d'un 386sx-16. Par l'intermédiaire de charge_fichiermod, le fichier MOD est chargé dans la mémoire. S'il se pose des problèmes pendant le chargement, cette fonction retourne une valeur différente de 0. Dans ce cas, le programme affiche un message d'erreur et s'interrompt. Sinon, on lance au moyen de periodic_on l'exécution du fichier MOD contrôlée par l'interruption.

On peut alors élaborer l'écran. Dans une boucle, on vérifie si ESC ou une autre touche définie a été pressée. Dans ce cas, il faut réagir en conséquence. Cette partie de la procédure est sans doute compréhensible sans autre explication.

Si la touche **ESC** a été pressée, on met fin au fichier MOD. On affecte pour cela la valeur TRUE à la variable "outfading", ce qui a pour effet d'éteindre le morceau. On n'entend plus de son lorsque "outvolume" a atteint une valeur inférieure à 1. Il faut également remettre l'écran à jour, ce que l'on fait par l'intermédiaire de la procédure "Scala". Quand cette valeur est atteinte, l'interruption timer est ramenée à son ancienne valeur par l'intermédiaire de periodic_off. On libère aussi la mémoire occupée par le fichier MOD au moyen de fin_mod. En dernier lieu, on effectue un reset de la SB, pour la ramener à son état original.

```
procedure Play_the_Mod(s : string);
var h : byte;
    error : integer;
    li : integer;
begin;
 Reset_Sb16;
 mod_SetSpeed(66);
 mod_Samplefreq(Samfreq);
 dsp_rdy_sb16 := true;
 error := charge_fichiermod(s,AUTO,AUTO,Samfreq);
 if error <> 0 then begin;
   clrscr;
   writeln('Erreur pendant le chargement du fichier MOD ! ');
   if error = -1 then writeln('Fichier introuvable !');
```

```
  if error = -2 then writeln('Mémoire insuffisante !');
  halt(0);
end;
periodic_on;                    { Active l'exécution périodique }
Scala_boxes;
ch := #255;
while not (ch=#27) and not (upcase(ch)='X')
  and not (upcase(ch)='N') do begin;
 Scala;
 if keypressed then ch := readkey;
 case ch of
   #0  : begin;
           dch := readkey;
           case dch of
             #61 : begin;        { F3 }
                     if Mastervolume > 0 then dec(Mastervolume);
                     Set_Volume(Mastervolume);
                     textbackground(black);
                     textcolor(lightblue);
                     writexy(47,2,'Volume: ');
                     textcolor(lightcyan);
                     write(Mastervolume:2);
                     ch := #255;
                   end;
             #62 : begin;        { F4 }
                     if Mastervolume < 31 then inc(Mastervolume);
                     Set_Volume(Mastervolume);
                     textbackground(black);
                     textcolor(lightblue);
                     writexy(47,2,'Volume: ');
                     textcolor(lightcyan);
                     write(Mastervolume:2);
                     ch := #255;
                   end;
             #63 : begin;        { F5 }
                     if Balance > 0 then dec(Balance);
                     Set_Balance(Balance);
                     textcolor(lightblue);
```

```
                textbackground(black);
                writexy(58,2,'Balance ');
                textcolor(14);
                writexy(66,2,'■■■■■■■■■■■■■■');
                textcolor(4);
                writexy(78-Balance DIV 2,2,'■');
                ch := #255;
              end;
        #64 : begin;        { F6 }
                if Balance < 24 then inc(Balance);
                Set_Balance(Balance);
                textcolor(lightblue);
                textbackground(black);
                writexy(58,2,'Balance ');
                textcolor(14);
                writexy(66,2,'■■■■■■■■■■■■■■');
                textcolor(4);
                writexy(78-Balance DIV 2,2,'■');
                ch := #255;
              end;
        else begin;
                ch := #255;
             end;
      end;
    end;
'6' : begin;
      inc(mli);
      ch := #255;
    end;
'f' : begin;
      filtre_actif := not filtre_actif;
      if filtre_actif then begin;
                           filtre_on;
        textcolor(lightblue);
        textbackground(black);
        writexy(36,2,'Filtre');
        textcolor(lightcyan);
        write(' ON ');
```

```
          end else begin;
                        filtre_mid;
            textcolor(lightblue);
            textbackground(black);
            writexy(36,2,'Filtre');
            textcolor(lightcyan);
            write(' OFF');
          end;
          ch := #255;
        end;
'4' : begin;
          if mli > 0 then
            dec(mli)
          else begin;
            if mlj > 0 then begin;
              dec(mlj);
              mli := 63
            end else begin;
              mli := 0;
              mlj := 0;
            end;
          end;
          ch := #255;
        end;
'3' : begin;
          mli := 0;
          inc(mlj);
          ch := #255;
        end;
'1' : begin;
          if mlj > 0 then begin;
            dec(mlj);
            mli := 0;
          end;
          ch := #255;
        end;
'N',
'n' : begin;
```

```
            next_song := 1;
          end;
    'x' : begin;
            next_song := 255;
          end;
    #27 : begin;
            next_song := 255;
          end;
    else begin;
           ch := #255;
    end;
   end;
 end;
 outfading := true;
 while outvolume > 1 do begin;
   Scala;
 end;
 periodic_off;
 fin_mod;
 Reset_Sb16;
end;
```

La procédure qui nous intéressera ensuite est "play_sound". Il faut lui transmettre le nom du fichier son à exécuter. S'il s'agit d'un fichier MOD, on appelle la procédure Play_the_mod. Sinon, on a probablement affaire à un fichier VOC. Pour sortir celui-ci, il faut d'abord effectuer un reset de la carte Sound Blaster. On sort ensuite un court texte d'information et on lance la sortie du fichier VOC par l'intermédiaire de "Init_Voc". Vous devez ensuite attendre que la variable Voc_ready prenne la valeur TRUE ou que l'utilisateur presse une touche. Par l'intermédiaire de voc_pause, vous pouvez interrompre l'émission du fichier VOC. Vous pouvez poursuivre l'émission du fichier par l'intermédiaire de voc_continue. Lorsque la sortie a pris fin (le fichier est parvenu à son terme) ou lorsque l'utilisateur a interrompu l'exécution prématurément, on appelle la procédure Voc_done. Celle-ci met fin à l'émission et ferme le fichier ouvert. Il faut appeler sans faute cette procédure.

```
procedure write_vocmessage;
begin;
  clrscr;
  writexy(10,08,'Attention !  Le Voc entrera dans une boucle sans
                              fin !!!');
```

```
  writexy(10,10,'Commencer  par  Q  ');
  writexy(10,11,'Pause      avec  P  ');
  writexy(10,12,'Continuer avec  C  ');
  writexy(10,13,'Reprendre avec  N  ');
  writexy(10,21,'                          E N J O Y');
end;

procedure play_sound(datname : string);
var li : integer;
    ch : char;
begin;
  for li := 1 to length(datname) do
     datname[li] := upcase(Datname[li]);
  if pos('.MOD',datname) <> 0 then begin;
    Play_The_Mod(datname);
    exit;
  end;
  if pos('.VOC',datname) <> 0 then begin;
  repeat
    Reset_Sb16;
    write_vocmessage;
    Init_Voc(datname);
    ch := #0;
    repeat
      if keypressed then ch := readkey;
      if ch = 'p' then begin;
        voc_pause;
        repeat
          ch := readkey;
        until ch = 'c';
        voc_continue;
      end;
    until VOC_READY or (ch = 'n') or (upcase(ch) = 'Q');
    VOC_DONE;
  until upcase(ch) = 'Q';
  end;
end;
```

Le code du programme principal est trivial. Veillez à appeler d'abord la procédure Init_the_mod, avant d'essayer de sortir avec telle ou telle routine. Si cette procédure n'a pas été appelée, la carte Sound Blaster ne sera pas initialisée correctement, ce qui conduira très vraisemblablement au blocage du programme et du système.

```
begin;
 Samfreq := 22;
 clrscr;
 cursor_off;
 interprete_commandline;
 if (Nummods = 0) and not repeatmode then begin;
   textcolor(15);
   textbackground(1);
   clrscr;
   Nummods := 1;
   modd[1] := select_fichier('*.?o?','*.?o?','','Prière de
                                                  sélectionner
              le fichier son');
   if modd[1] = 'xxxx' then begin;
     clrscr;
         writeln('Vous avez donc déjà un fichier son !');
     Cursor_on;
     halt(0);
   end;
 end;
 for i := 1 to Nummods do begin;
   if pos('.',modd[i]) = 0 then modd[i] := modd[i]+'.mod';
 end;
 Init_The_Mod;
 stereo := false;
 next_song := random(Nummods)+1;
 textcolor(lightgray);
 textbackground(black);
 write_sbconfig;
 repeat
   if repeatmode then begin;
     textcolor(15);
     textbackground(1);
```

```
      clrscr;
      modd[1] := select_fichier('*.?o?','*.?o?','','');
      if modd[1] = 'xxxx' then next_song := 255
      else Play_Sound(modd[1]);
    end else
      Play_Sound(modd[next_song]);
    if next_song <> 255 then next_song := random(Nummods)+1;
  until next_song = 255;
  cursor_on;
  textmode(3);
end.
```

C'est tout ce que vous devez savoir sur la programmation de la SB. Nous vous souhaitons beaucoup de succès dans vos essais et vos projets d'amélioration. Les exemples fournis offrent certainement des possibilités étendues d'optimisation et d'intégration dans vos propres programmes. Vous pourrez peut-être utiliser l'unité programmée dans votre prochaine démo ou votre prochain jeu. Ou alors vous pourrez écrire un éditeur MOD. Tant de choses sont possibles...

15.3. Comment on programme un player MOD pour la Gravis Ultrasound

La carte Gravis Ultrasound est extraordinairement propice à la reproduction des fichiers MOD. Elle offre la possibilité de charger les samples complets du module dans la RAM de la carte et d'exécuter les voix directement en fonction de l'arrangement dans le MOD. Les données à émettre n'ont plus besoin d'être mixées manuellement, c'est la GUS qui s'en charge pour vous.

Comment pouvons-nous faire usage de ces capacités ? Nous avons besoin d'une unité de contrôle, avec laquelle nous pourrons émettre simplement des fichiers MOD. Vous trouverez cette unité sous le nom de "gus_mod". Elle vous propose quelques fonctions élémentaires en petit nombre, par l'intermédiaire desquelles vous pouvez initialiser la carte, charger un fichier MOD et l'exécuter. Si vous êtes pressé, il vous suffit de lire le paragraphe qui suit à propos de l'interface. Nous y expliquons comment réaliser ces quelques fonctions.

Structure du player MOD

L'exécution d'un fichier MOD est facile à réaliser. Vous devez d'abord initialiser la Gravis Ultrasound. Vous avez besoin pour cela du port de base de la carte GUS. Vous l'obtiendrez au moyen de la fonction _gus_init_env. Elle retourne la valeur TRUE lorsque le port a pu être connu au moyen de la variable d'environnement "Ultrasnd". Sinon, vous obtenez la valeur FALSE. Vous pouvez alors initialiser la carte par l'intermédiaire de _initialiser_gus.

La carte est alors prête à recevoir des données. Transférez celles-ci par l'intermédiaire de la fonction _gus_modload. Transmettez à cette fonction le nom du fichier à charger. La fonction retourne la valeur TRUE lorsque le fichier a pu être chargé, sinon la valeur FALSE. Une fois le fichier MOD chargé de cette manière dans la mémoire, vous pouvez lancer l'émission par l'intermédiaire _gus_modstart. Le fichier MOD entre automatiquement dans un loop lorsqu'il parvient à son terme. Pour mettre fin à l'émission, appelez la procédure _gus_mod_quitter. Elle arrête l'exécution du MOD et l'enlève de la mémoire. Vous savez maintenant tout ce qui est nécessaire pour intégrer dans vos programmes un player MOD pour la carte GUS.

Variables importantes du player MOD pour GUS

Lorsque vous voulez réaliser un player MOD, il ne suffit pas de vous donner les moyens d'émettre un fichier MOD. Vous devez aussi pouvoir influer sur le programme pendant l'exécution et donner à l'utilisateur des informations sur l'état actuel du player.

Vous y parviendrez par l'intermédiaire des différentes variables de l'unité programmée. Passez en revue les variables globales qui interviennent.

Play_Chanel : array[1..14] of byte;

Les entrées du tableau "Play_Chanel" peuvent prendre des valeurs égales à 0 ou à 1. La valeur 1 signifie que le canal correspondant est activé. La valeur 0 désactive au contraire le canal. On réalise ainsi de façon simple un "muting" du canal.

Canaux : array[1..31] of PCanalInfo;

Dans ce tableau figurent les informations sur les différents canaux. Le tableau est constitué de pointeurs sur les structures des canaux. La structure d'un canal se présente de la façon suivante :

```
TCanalInfo = record
  InstNr     : byte;          { variables relatives au hardware }
  Mempos     : longint;
  Fin        : longint;
```

```
    Loop_Start : longint;
    Loop_Fin   : longint;
    Volume     : integer;        { Variables relatives au fichier MOD }
    Freq       : word;
    Looping    : byte;
    Son        : integer;
    Son_init   : integer;
    Son_final  : integer;
    Effet      : byte;
    Operand    : byte;
    Effetx,                      { Variables relatives aux effets}
    Effety      : integer;
    Appegpos    : integer;
    slidespeed  : integer;
    vslide      : integer;
    retrig_count : byte;
    vibpos      : byte;
    vibx        : byte;
    viby        : byte;
  end;
```

Les éléments de cette structure sont à interpréter ainsi :

InstNr

"InstNr" contient le numéro de l'instrument à exécuter, tel qu'il intervient dans le fichier MOD. Il peut prendre une valeur située entre 1 et 31.

Mempos

La position de départ de la voix active dans la RAM de la carte GUS se trouve dans "Mempos".

Fin

"Fin" désigne la position finale de la voix dans la RAM de la carte GUS.

Loop_Start

Une voix peut être soumise à un loop. Dans ce cas, vous trouverez ici la position de départ du looping à l'intérieur de la RAM de la carte GUS.

Loop_Fin

Cette variable contient la fin du looping. Elle se réfère elle aussi à la RAM de la carte GUS.

Volume

On trouve ici le volume avec lequel la voix est exécutée. On utilise les volumes du fichier MOD, situés entre 0 et 63.

Freq

La fréquence de la voix, telle qu'elle est déposée dans le fichier MOD, se trouve dans cet élément de la structure. Il ne s'agit pas de la fréquence d'émission réelle, mais de la fréquence utilisée dans le MOD, à partir de laquelle il faut encore effectuer un calcul. Elle est nécessaire en particulier pour les différents effets à produire.

Looping

Cette variable indique si la voix est placée dans un looping ou non. Elle contient la valeur 0 quand il n'y a pas de looping. Sinon, elle contient la valeur 8. Cette valeur est nécessaire pour la programmation de la carte GUS. Si le troisième bit de la commande de démarrage d'une voix est posé, cela signifie que la voix entre dans un loop.

Son, Son_init, Son_final

Ces variables interviennent dans le calcul à l'intérieur du MOD. Il faut plusieurs variables, car il arrive souvent que l'on ait besoin de conserver des valeurs intermédiaires.

Effet

On dépose ici l'effet actuel qui figure dans le fichier MOD.

Operand

Il s'agit ici de l'opérande correspondant à l'effet recherché.

Effetx, Effety

Dans "Effetx", on trouve les quatre bits supérieurs de l'opérande. Dans "Effety", ce sont les quatre bits inférieurs.

Appegpos

Dans un arpège, on frappe trois notes successives. Cette variable indique quelles sont ces notes.

slidespeed

Cette variable est nécessaire pour le sliding de la fréquence. Elle indique la vitesse avec laquelle ce dernier sera réalisé.

vslide

Comme pour la précédente, cette variable indique la vitesse d'un sliding du volume.

retrig_count

Cette variable est utilisée pour réaliser l'effet "retrigger". Elle indique la position correspondant au redémarrage de la voix.

vibpos

Vous trouverez ici la position actuelle à l'intérieur d'un tableau de vibrato.

vibx, viby

Les paramètres x et y de l'effet vibrato sont conservés dans ces deux variables.

Instruments : array[1..31] of PIntrumentnrInfo

Les informations pour les 31 instruments d'un fichier MOD se trouvent ici. Il s'agit d'un tableau de pointeurs sur la structure suivante :

```
PInstrumentnrInfo = ^TInstrumentInfo;
 TInstrumentInfo = record
   Nom        : string[22];
   Mempos     : longint;
   Fin        : longint;
   l_Start    : longint;
   l_Fin      : longint;
   Taille     : word;
   Loop_Start : word;
   Loop_Fin   : word;
   Volume     : word;
   Looping    : byte;
 end;
```

Les différents éléments de cette structure ont la signification suivante :

Nom

On trouve ici le nom de chacun des instruments.

Mempos

"Mempos" contient la position des données sample à l'intérieur de la RAM de la carte GUS.

Fin

"Fin" désigne la position finale de l'instrument dans la RAM de la carte GUS.

L_Start

Un instrument peut entrer dans un loop. Dans ce cas, vous trouverez ici la position de départ du looping dans la RAM de la carte GUS.

L_Fin

On trouve dans cette variable l'extrémité du looping. Elle se réfère elle aussi à la RAM de la carte GUS.

Taille

La longueur de la voix, telle qu'elle est déposée dans le fichier MOD, est reportée ici (exprimée en octets).

Loop_Start

Cet élément de la structure désigne l'offset de l'instrument où commence le looping de la voix.

Loop_Fin

On trouve ici l'offset où le looping prend fin dans les données sample de l'instrument.

Volume

Le volume de la voix est déposé ici. Il peut se situer entre 0 et 63.

Looping

Cette variable indique si la voix entre ou non dans un loop. Elle contient la valeur 0 si la voix n'est pas soumise à un loop, sinon la valeur 8. Cette valeur intervient dans la programmation de la carte GUS.

MOD_Voix : word

En fonction du nombre de voix dans la chanson, vous trouverez ici la valeur 4 ou 8.

Mod_Patternsize : word

La taille d'un pattern est conservée dans cette variable. On la calcule à l'aide de la formule MOD_Voix*256. Cela fait qu'un pattern de 4 voix a une taille de 1024 octets.

Stop_TheVoice : array[1..8] of boolean;

Ce tableau sert à arrêter une voix donnée. Si vous affectez à la variable qui lui correspond la valeur TRUE, la voix s'arrête.

Modinf : Modinfo;

"Modinf" contient des informations globales sur le fichier MOD.

Titre

Le "titre" de la chanson est conservé ici.

Nb_Patt

On trouve ici le nombre de patterns définis.

Runinf : Runinfo;

Cette variable contient les informations qui apparaîtront pendant l'exécution, par exemple celles dont vous aurez besoin pour réaliser un player MOD ou pour synchroniser un graphique avec le fichier MOD. Ses éléments sont les suivants :

Frappe[1..8]

Pour chaque voix, on simule ici un "pseudo-equalizer". La valeur est toujours posée sur 63 lorsqu'on frappe une note. A chaque battement, cette valeur est décrémentée.

Ligne

"Ligne" désigne la ligne actuelle dans le pattern en cours d'exécution.

Pattnr

On trouve ici la position à l'intérieur de l'arrangement. Grâce à cette position et à la variable "Ligne", on peut synchroniser correctement certains événements ou certaines images de l'animation avec la musique que l'on entend.

Volumes[1..8]

On trouve ici le volume actuel de chacune des voix.

Speed, BPM

Cette variable contient la vitesse d'exécution du MOD. "Speed" a une valeur située entre 1 et 15 et indique la vitesse en relation avec la vitesse de base actuelle en BPM (par défaut : 125).

chpos : array[1..8] of byte;

On entre ici la position des voix. Les valeurs possibles vont de 0 (tout à fait à gauche) jusqu'à 15 (tout à fait à droite).

VibratoTable : array[0..63] of integer;

Dans la table du vibrato, on trouve toutes les entrées nécessaires pour exécuter le vibrato. La table comprend 64 entrées, permettant de décrire avec exactitude une oscillation. Les valeurs extrêmes se trouvent dans Pos 15 et 47. Pour 0 et 32, elles sont égales à 0. Cette table est utilisée aussi bien pour l'effet vibrato que pour l'effet Tremolo.

Routines centrales du player MOD

Il est temps maintenant d'en venir aux procédures. De quoi avons-nous donc besoin pour réaliser un player MOD ? Tout d'abord, bien sûr, de procédures permettant de charger le MOD à partir du disque. Nous devons ensuite amener d'une façon ou d'une autre les données sample sur la carte GUS. Enfin, nous avons également besoin de routines pour l'exécution du MOD.

Le chargement d'un fichier MOD

Regardons donc en premier lieu comment on charge un fichier MOD. Dans cette unité, c'est la fonction _gus_modload qui s'en charge. Il faut lui transmettre le nom complet du fichier MOD à charger. Lorsque le fichier a été chargé, la fonction retourne la valeur TRUE. Sinon, elle s'interrompt avec le résultat FALSE.

La fonction réserve d'abord la mémoire nécessaire pour les structures des canaux et pour les informations sur les instruments. Elle définit ensuite dans "Runinf" la "Ligne" sur 0 et le "Pattnr" sur -1. De cette façon, la routine d'exécution commencera au début du fichier MOD.

On peut maintenant ouvrir le fichier. La procédure sauvegarde l'écran et affiche un graphique ANSI, qui indique le chargement en cours du fichier MOD (Save_Screen; display_loading). Ces deux appels peuvent être laissés de côté dans votre player ou être remplacés par vos propres procédures d'affichage.

"longreste" reçoit la valeur "1084", correspondant à la taille du fichier. Cette variable contient la taille du fichier MOD sans le header. Par son intermédiaire, on détermine le nombre de patterns dans le MOD. On vérifie ensuite au moyen du numéro d'identification s'il s'agit vraiment d'un fichier MOD et on se demande combien de voix il possède. On affecte les valeurs correspondantes aux variables "MOD_Voix" et "MOD_Patternsize". On peut alors charger entièrement le header. Nous commençons par le nom du fichier MOD. Celui-ci est une chaîne terminée par 0 et il ne peut pas être traité directement sous Pascal. Pour le convertir en format Pascal, nous utilisons la fonction "ConvertString", à laquelle il faut transmettre un pointeur sur un secteur de mémoire dans lequel se trouve la chaîne en question et aussi la longueur maximale de cette dernière. La fonction retourne alors la chaîne en Pascal, et nous pouvons utiliser celle-ci dans notre programme.

Nous allons maintenant chercher la longueur de la chanson, c'est-à-dire le nombre d'entrées valides définies dans l'arrangement. Cette longueur n'a aucun rapport avec le nombre de patterns déposés dans le fichier MOD ! On doit ensuite lire l'arrangement long de 128 octets, avant de pouvoir charger les instruments.

Le problème qui se pose à nous est qu'il existe des MOD avec 31 instruments, mais aussi certains modules à l'ancienne mode avec seulement 15 voix. Ce serait un peu compliqué (bien que ce soit tout à fait possible) d'exécuter le chargement avec deux secteurs séparés pour le code. Nous préférons utiliser ici deux variables. L'une d'elles est la variable "vh.Num_Inst". Elle contient le nombre d'instruments existant effectivement dans le fichier MOD. L'autre est "ias", destinée à contenir la position de départ en fonction des instruments. Avec 31 instruments, cette variable contient la valeur 0. Avec 15 instruments, elle contient la valeur -16*30. Cette dernière s'explique comme suit : on a 16 instruments de moins et chacun d'eux possède une information longue de 30 octets. Nous travaillons de manière interne avec les offsets d'un fichier à 31 voix, nous devons donc soustraire l'offset correspondant aux 16 instruments en moins.

Une fois que nous disposons de ces deux variables, nous pouvons lire tous les instruments définis dans une boucle. Cela se fait de la façon suivante :

Nous lisons d'abord le nom de l'instrument, long de 22 octets. Ici aussi, il s'agit d'une chaîne terminée par 0, que nous devons convertir en une chaîne Pascal au moyen de la fonction ConvertString. Nous lisons ensuite la taille du sample exprimée en octets. Cette longueur ne correspond malheureusement pas à la taille réelle. Il s'agit d'une indication word Amiga. Cela veut dire que nous devons d'abord échanger les octets Low et High pour convertir l'indication en format PC. Nous devons ensuite multiplier la valeur obtenue par 2, puisqu'il s'agit d'une indication exprimée en "words", alors que nous avons besoin de connaître la longueur exprimée en octets.

Ce système de conversion est identique pour toutes les indications Amiga données en "mots". L'indication suivante, celle du volume, est une valeur exprimée directement en octets et elle est située entre 0 et 63. Nous n'avons donc pas besoin de la convertir. Viennent ensuite la position de départ du looping de sample (considérée comme un offset dans le sample) et la longueur de ce looping. Les deux sont des indications Amiga en mots et elles doivent être converties. A partir de la longueur, nous déterminons directement la position finale du sample. Nous en aurons besoin pendant le traitement du MOD.

La fonction de chargement examine alors si l'instrument considéré entre dans un loop, donc si la différence entre le début et la fin du loop est >=10. Dans ce cas, la variable "Looping" pour les instruments est pourvue de la valeur avec_loop (=8). Sinon, on lui affecte la valeur "no_loop". Pour finir, on décrémente "longreste" de la longueur du MOD. De cette façon, après lecture de toutes les informations sur les instruments, cette variable contient une valeur qui correspond au nombre d'octets de tous les patterns définis. On peut déterminer à partir de là le nombre de patterns définis, au moyen d'une simple division par la taille d'un pattern.

Pour finir, on charge les instruments. Cela se fait également dans une boucle. On utilise pour cela la procédure "Charge_Instrument", que nous allons examiner de plus près dans un instant. Dans la boucle, tout de suite après l'appel de la procédure, on trouve une manipulation de la mémoire d'écran, que l'on adresse par l'intermédiaire de la variable "Screen". On modifie toujours ainsi l'attribut (ici la couleur de premier plan) du caractère dans l'échelle progress de l'Ansi pour le chargement. C'est une opération nécessaire uniquement pour le player TCP. Vous pouvez la mettre de côté dans vos propres routines ou la remplacer par les graphiques concoctés par vos soins.

Vous avez ainsi chargé tout le fichier MOD dans la mémoire et vous pouvez alors le fermer. Dans la fonction de chargement, on initialise encore quelques variables importantes. On définit d'abord correctement toutes les voix de la carte GUS. On ordonne ensuite les canaux dans une sorte de demi-cercle. Cela permet de simuler une sonorité spatiale. Pour cela, on affecte aux variables chpos[1] à chpos[Mod_Voix] des valeurs situées entre 0 et 15. Par l'intermédiaire de _gus_set_chanelpos, on distribue ensuite ces variables aux différentes voix. On règle ensuite la vitesse d'interruption (interrupt-speed) sur la valeur standard 125 et on restaure l'écran au moyen de la procédure "restore_screen". Dans votre propre routine de chargement, vous pouvez à la rigueur vous passer de cette restauration de l'écran, puisqu'elle est nécessaire uniquement pour TCP.

```
function _gus_modload(nom : string) : boolean;
{
 Charge le fichier MOD transmis par "nom". Suppose que le fichier
 indiqué dans le chemin d'accès existe et n'est pas protégé contre
 l'écriture.
}
var dummya : array[0..30] of byte;
                                { Pour le traitement de la chaîne }
    daptr : pointer;            { pointeur sur dummya }
    dumw : word;                { Variable dummy pour lire}
    dumb : byte;
    Longreste : longint;        { pour connaître le nombre de
                                  patterns }
    li : integer;
    Identit : array[1..4] of char;
                                { Identité du fichier MOD }
    ias : integer;              { Position de départ fonction de
                                  l'instrument }
begin;
  U_Ram_freepos := 32;
  for li := 0 to 15 do begin;
    new(Canaux[li]);
    canaux[li]^.vibpos := 0;
  end;
  for li := 0 to 31 do begin;
     new(Instruments[li]);
     Instruments[li]^.Nom := '';
  end;
  runinf.Ligne := 0;
  runinf.Pattnr := -1;
  tickcounter := 0;
  ticklimit := 6;
  runinf.speed := 6;
  runinf.bpm := 125;
  ias := 0;
  daptr := @dummya;
  assign(gusmf,nom);            { Ouvrir fichier + initialiser la
                                  longueur }
```

```
reset(gusmf,1);
save_screen;
display_loading(nom);
Longreste := filesize(gusmf);
Longreste := Longreste  - 1084;
seek(gusmf,1080);                { Vérifier si c'est un MOD à 15 ou
                                   à 31 voix }
Blockread(gusmf,Identit,4);
if pos(Identit,Modidentit) = 0 then begin;
  { 15 Voix? }
  seek(gusmf,600);
  Blockread(gusmf,Identit,4);
  if pos(Identit,Modidentit) = 0 then begin;
    { Pas de fichier MOD valide }
    writeln('Pas de fichier MOD valide !!!');
    halt(0);
  end else begin;
    Modinstance := 15;
    ias := -16*30;
  end;
end;
if (Identit = MODId[1]) or      { Nombre de voix du fichier MOD ?}
   (Identit = MODId[2]) or
   (Identit = MODId[3])
then begin;
  MOD_Voix := 4;
  MOD_Patternsize := 4*256;
end else
  if (Identit = CHn6) then begin;
    _gus_modload := false;
    exit;
  end else
    if (Identit = CHn8) then begin;
      MOD_Voix := 8;
      MOD_Patternsize := 8*256;
    end;
seek(gusmf,0);
Blockread(gusmf,dummya,20);     { Connaître le nom du fichier }
```

```
vh.SongName := ConvertString(daptr,20);
seek(gusmf,950+ias);              { Longueur en patterns }
Blockread(gusmf,vh.longchanson,1);
seek(gusmf,952+ias);              { lit l'arrangement }
Blockread(gusmf,vh.Arrang,128);
vh.Num_Inst := Modinstance;       { lit les instruments (15/31) }
seek(gusmf,20+ias);
for li := 1 to 32 do Instruments[li]^.Nom := '';
for li := 1 to vh.Num_Inst do begin;
  Blockread(gusmf,dummya,22);  { Noms des instruments }
  Instruments[li]^.Nom := ConvertString(daptr,22);
  Blockread(gusmf,dumw,2);        { Longueur du sample }
  Instruments[li]^.Taille := swap(dumw) * 2;
  Blockread(gusmf,dumb,1);        { lit le volume }
  Blockread(gusmf,dumb,1);
  Instruments[li]^.Volume := dumb;
  Blockread(gusmf,dumw,2);        { lit le départ du loop }
  Instruments[li]^.Loop_Start := swap(dumw) * 2;
  Blockread(gusmf,dumw,2);
  dumw := swap(dumw) * 2;         { lit la fin du loop : début +
                                    longueur}
  Instruments[li]^.Loop_Fin := Instruments[li]^.Loop_Start+dumw;
  if (Instruments[li]^.Loop_Fin -
                                  { Looping dans l'instrument ?    }
      Instruments[li]^.Loop_Start) >= 10 then begin;
    Instruments[li]^.Looping := mit_loop;
  end else begin;
    Instruments[li]^.Looping := no_loop;
  end;
  Dec(Longreste,Instruments[li]^.Taille);
end;
Vh.Num_Patts := Longreste DIV MOD_Patternsize ;
                                  { nombre de patterns ?    }
seek(gusmf,1084+ias);
for li := 1 to Vh.Num_Patts do begin;
                                  { lit les patterns }
  dos_getmem(Pattern[li],MOD_Patternsize );
  Blockread(gusmf,Pattern[li]^,MOD_Patternsize );
```

```
end;
for li := 1 to vh.Num_Inst do begin;
                                { lit les instruments }
  Charge_Instrument(li);
  screen[16,23+li].a := 5;
end;
close(gusmf);
for i := 1 to 31 do begin;      { initialise les variables de
                                  canaux }
  u_VoiceBalance (i,7) ;
  u_VoiceVolume (i,0) ;
  u_VoiceFreq (i,12000);
  U_StartVoice(i,Stop_Voice);
  u_Voicedata(0,0,0,i);
end;
runinf.Ligne := 0;              { initialise les variables run }
runinf.Pattnr := -1;
tickcounter := 0;
ticklimit := 6;
runinf.speed := 6;
runinf.bpm := 125;
if MOD_Voix = 4 then begin;     { ordonne les voix en demi-cercle }
  chpos[1] := 2;
  chpos[2] := 5;
  chpos[3] := 9;
  chpos[4] := 12;
  _gus_set_chanelpos;;
end;
if MOD_Voix = 8 then begin;
  chpos[1] := 1;
  chpos[2] := 3;
  chpos[3] := 5;
  chpos[4] := 7;
  chpos[5] := 7;
  chpos[6] := 9;
  chpos[7] := 11;
  chpos[8] := 13;
  _gus_set_chanelpos;;
```

```
  end;
  nouv_interrupt_Speed(runinf.bpm);
  restore_screen;
  Modinf.Voix  := MOD_Voix;      { Constante MOD-Infos dans la
                                   structure }
  Modinf.Titre   := vh.Songname; { à transmettre }
  Modinf.Nb_patt := Vh.Longchanson;
  _gus_modload    := true;
end;
```

Il nous reste ici à examiner la procédure prévue pour charger les instruments. Elle porte le nom de "Charge_Instrument" et on doit lui transmettre le numéro de l'instrument à charger. Si celui-ci a une taille supérieure à 10 octets, il est effectivement chargé dans la RAM de la carte GUS. La procédure fait venir pour cela tout d'abord le sample dans un tampon de 64 Ko. Elle cherche la position suivante aux adresses de paragraphe (adresse divisée par 16) dans la RAM de la carte GUS. Cette position est la position de départ du sample. Elle est affectée à la variable "Mempos" de l'instrument. Ensuite, la procédure "Ultra_Mem2Gus" transfère le sample dans la RAM de la carte GUS. Le tampon réservé est alors libéré. Après quoi on reporte dans les informations concernant l'instrument la position de départ du looping et la fin de la voix dans la RAM de la carte. Jetons maintenant un coup d'œil sur la réalisation technique de ces opérations sous forme programmée :

```
procedure Charge_Instrument(Nr : byte);
{
 Charge l'instrument de numéro nr dans la RAM de la carte GUS
}
var gr : longint;
    samp : pointer;
begin;
   gr := Instruments[nr]^.Taille;
   if gr > 10 then begin;       { ne charge que si > 10! }
     dos_getmem(samp,gr);
     Blockread(gusmf,samp^,gr);
     U_Ram_freepos := U_Ram_freepos + (16-(U_Ram_freepos MOD 16));
     Instruments[nr]^.Mempos := U_Ram_freepos;
     Ultra_Mem2Gus(samp,Instruments[nr]^.Mempos,gr);
     dos_freemem(samp);          { Initialise les variables des voix}
     Instruments[nr]^.l_start :=
        Instruments[nr]^.Mempos + Instruments[nr]^.Loop_Start;
```

```
      if Instruments[nr]^.Looping = Avec_loop then begin;
        Instruments[nr]^.fin :=
          Instruments[nr]^.Mempos + Instruments[nr]^.Loop_Fin;
      end else begin;
        Instruments[nr]^.fin := Instruments[nr]^.Mempos + gr - 25;
      end;
      Inc(U_Ram_Freepos,gr);       { repousser le pointeur de gestion }
    end;
  end;
```

L'exécution d'un fichier MOD chargé

Maintenant que le fichier MOD est chargé, nous voulons bien sûr l'exécuter. Comment le faire de la façon la plus élégante possible ? Théoriquement, on pourrait toujours incrémenter une variable dans une boucle ou effectuer la synchronisation avec le balayage vertical de la carte graphique. Cette méthode serait toutefois trop imprécise et, ce qui est pire, elle manquerait singulièrement de souplesse. Nous nous servirons donc plutôt de l'interruption timer. Nous devons pour cela programmer l'interruption avec une nouvelle fréquence et détourner la routine d'exécution sur notre propre procédure. Dans celle-ci, nous devons incrémenter un compteur à chaque battement (donc à chaque appel). Lorsque ce compteur atteint la vitesse spécifiée dans le fichier MOD, soit "Speed", le fichier MOD doit passer à la ligne suivante. Sinon, on doit traiter certains effets qui interviennent éventuellement en cours d'exécution. N'oubliez pas d'envoyer à la fin un EOI (End Of Interrupt) en écrivant la valeur 20h au port 20h.

Notre procédure de démarrage du fichier MOD porte le nom _gus_modstart. Elle calcule la fréquence spécifiée pour le timer à partir des BPM (Beats Per Minute) actifs dans le MOD. On programme le timer par l'intermédiaire des ports 43h et 40h. Le timer est réglé de telle sorte qu'il déclenche une interruption périodique. Avant d'affecter au timer notre propre procédure, nous devons encore sauvegarder l'ancienne interruption. C'est nécessaire pour que nous puissions ultérieurement la restaurer. Lorsque notre programme arrivera à son terme, il ne faut pas que le timer continue à pointer sur notre propre procédure, car cela conduirait immanquablement à un blocage du système.

La restauration de l'interruption est prise en charge par la procédure "timerint_rest". Elle effectue un reset du timer en lui rendant sa fréquence originale et elle rend à l'interruption sa routine originale.

Du fait que nous utilisons l'interruption timer, nous pouvons aussi réaliser très simplement une fonction "Pause". Si nous voulons ainsi interrompre l'émission, il nous suffit de détourner le timer sur une autre routine, chargée

uniquement d'envoyer un EOI. Toutefois, il faut définir auparavant le volume des différents canaux sur 0. Lorsque l'on veut alors poursuivre l'exécution, il suffit de replacer l'interruption sur la routine du player et d'activer les voix à nouveau.

Voyons maintenant comment se présente l'unité programmée :

```
{$F+}
procedure mytimer; interrupt;
{
 Mon interruption de timer
}
begin;
  tick_effects;
  inc(tickcounter);
  if tickcounter >= ticklimit then begin;
    Tickcounter := 0;
    Play_Pattern_gus;
  end;
  Port[$20] := $20;
end;

procedure rien; interrupt;
{
 Interrupt dummy. On passe à cette procédure pour arrêter la sortie
 du son.
}
begin;
  port[$20] := $20;
end;

procedure _gus_modstart;
{
 Lance la sortie du fichier MOD par l'intermédiaire de
 l'interruption du timer. Le fichier MOD doit avoir été chargé au
 préalable !
}
var compteur : word;
    loz,hiz : byte;
begin;
```

```
 compteur := 1193180 DIV interrupt_speed;
 loz := lo(compteur);
 hiz := hi(compteur);
 asm
  cli
  mov dx,43h
  mov al,36h
  out dx,al
  mov dx,40h
  mov al,loz
  out dx,al
  mov al,hiz
  out dx,al
 end;
 getintvec(8,ancientimer);
 setintvec(8,@Mytimer);
 asm sti end;
end;

procedure _gus_player_pause;
{
 Arrête la sortie par l'interruption du timer
}
var li : integer;
begin;
 setintvec(8,@Rien);
 for li := 0 to 31 do
   u_VoiceVolume (li,0) ;
end;

procedure _gus_player_continue;
{
 Continue la sortie par l'interruption du timer.
}
var li : integer;
begin;
 setintvec(8,@Mytimer);
 for li := 1 to 31 do
```

```
    Voice_Rampin(li,Canaux[li]^.volume);
end;

procedure timerint_rest;
{
 Restaure l'interruption du timer en lui restituant ses valeurs
 initiales.
}
begin;
 asm
  cli
  mov dx,43h
  mov al,36h
  out dx,al
  xor ax,ax
  mov dx,40h
  out dx,al
  out dx,al
 end;
 setintvec(8,ancientimer);
 asm sti end;
end;
```

Chaque fois que le fichier MOD passe à la ligne suivante, on appelle la procédure play_pattern_gus. Elle est chargée de l'exécution des voix conformément à l'arrangement et aux patterns. Elle fait d'abord passer le fichier MOD une ligne plus loin. Elle vérifie ensuite si le numéro de ligne incrémenté est supérieur à 64. Dans ce cas, il faut que le MOD passe au pattern suivant et on en vient au début de la ligne, c'est-à-dire à 1. Si le numéro du pattern suivant, c'est-à-dire la position dans l'arrangement, dépasse la longueur de la chanson, on ramène la position au début de la chanson. On transfère alors la ligne actuelle dans le pattern au tableau La_Ligne. Cela simplifie l'accès aux différents octets.

Il faut ensuite traiter la ligne en question. Pour chacune des voix, on vérifie d'abord si elle doit effectivement être exécutée. C'est le cas lorsque l'entrée correspondante dans play_chanel contient la valeur 1. On va alors chercher dans la ligne la note à jouer, l'instrument à utiliser et éventuellement l'effet prévu. Il s'agit ensuite de savoir si une note doit être frappée ou si l'on doit utiliser un autre instrument.

Lorsque l'entrée pour la note est différente de 0, c'est qu'une fréquence a été indiquée. Il faut alors vérifier si l'effet placé dans la ligne est égal à 3. Dans ce cas, la note est une note cible qui doit être atteinte. Sinon, c'est une note à jouer tout de suite. On affecte en conséquence le son aux variables de canaux "Son_final" ou "Son" et "Son_initial".

Si l'entrée pour l'instrument est différente de 0, il faut charger à nouveau les variables de canaux pour les paramètres des instruments. On transfère d'abord le numéro de l'instrument qui intervient, puis les positions pertinentes dans la RAM de la carte GUS. Il ne reste plus qu'à définir le volume et à transférer un looping éventuel. Il est essentiel d'écrire les valeurs ainsi obtenues directement dans les paramètres des différentes voix, car l'opération de frappe dépend de la note.

Maintenant que vous avez traité aussi bien les instruments que la frappe de la note, passez au traitement des effets. Nous y appliquerons une procédure particulière, que nous allons présenter dans un instant. Elle est trop complexe pour la traiter tout de suite. Le code source deviendrait vite incompréhensible.

En dernier lieu, il faut vérifier si l'on doit émettre un son (Son<>0). Pour cela, vous devez remettre la main sur la fréquence prévue pour le canal. Vous le ferez en divisant la fréquence de base pour le nombre de voix par la fréquence qui figure dans le fichier MOD. Cette fréquence pourra être définie par l'intermédiaire de la procédure u_VoiceFreq. Lorsque la commande Set_Volume a été envoyée, vous devez encore vérifier si elle se trouve dans le cadre voulu. Définissez alors le volume de l'instrument et arrêtez l'émission de l'ancienne voix. La voix peut alors être lancée avec les données que vous venez de programmer.

Voici la traduction de ces opérations en Pascal :

```
procedure play_pattern_gus;
{
 On appellera cette procédure régulièrement. Elle exécute une ligne
 du fichier MOD.
}
var li        : integer;
    dumw      : word;
    La_ligne : array[1..8,0..3] of Byte;
    Effet     : byte;
    Ton       : word;
    Inst      : byte;
begin;
{
```

```
 ************************************************************************
 ***                  Passer dans le fichier MOD                      ***
 ************************************************************************
}
  inc(runinf.Ligne);                    { En avant d'une ligne }
  if runinf.Ligne > 64 then runinf.Ligne := 1;
  if runinf.Ligne = 1 then begin;       { Nouveau pattern ? }
    inc(runinf.Pattnr);
    if runinf.Pattnr > vh.Longchanson then runinf.Pattnr := 1;
  end;
                                        { Charge les notes }
  move(ptr(seg(pattern[vh.Arrang[runinf.Pattnr]+1]^),
       ofs(pattern[vh.Arrang[runinf.Pattnr]+1]^)+
          (runinf.Ligne-1)*4*Mod_Voix)^,
       La_ligne,4*8);
{
 ************************************************************************
 ***                       Traiter les voix                           ***
 ************************************************************************
}
  for li := 1 to MOD_Voix do begin;
    if play_chanel[li] = 1 then begin;
      stop_Thevoice[li] := false;
      Ton  := ((La_ligne[li,0] AND $0f) shl 8)+La_ligne[li,1];
      Inst := (La_ligne[li,0] AND $f0)+((La_ligne[li,2] AND $F0)
                                                    SHR 4);
      Canaux[li]^.Effet   := La_ligne[li,2] AND $0f;
      Canaux[li]^.Operand := La_ligne[li,3];
      Canaux[li]^.Son_init := oldv[li];
      if Ton <> 0 then begin;  { Est-ce qu'un son a été indiqué ??? }
        if Canaux[li]^.Effet = 3 then begin;
          Canaux[li]^.Son_final := Ton;
        end else begin;
          Canaux[li]^.Son := Ton;
          Canaux[li]^.Son_init := Ton;
          oldv[li] := Canaux[li]^.Son_init;
        end;
      end;
```

```
If Inst <> 0 then begin; { utiliser un nouvel instrument ??? }
  Canaux[li]^.InstNr      := Inst;
  Canaux[li]^.Mempos      := Instruments[Canaux[li]^.InstNr]^.
                             Mempos;
  Canaux[li]^.Loop_Start  := Instruments[Canaux[li]^.InstNr]^.
                             l_start;
  Canaux[li]^.Fin         := Instruments[Canaux[li]^.InstNr]^.
                             Fin;
  Canaux[li]^.volume      := Instruments[Canaux[li]^.InstNr]^.
                             volume;
  Canaux[li]^.Looping     := Instruments[Canaux[li]^.InstNr]^.
                             Looping;
  u_Voicedata(Canaux[li]^.Mempos,Canaux[li]^.Loop_Start,
              Canaux[li]^.Fin,li);
end;
Canaux[li]^.Retrig_count := 0;
Initialiser_Effets(li);
If (Ton <> 0) then begin;{ note frappée }
  Canaux[li]^.Freq := longint(Voice_Base[14] div Canaux[li]^.
                      Son_init);
  u_VoiceFre(li,Canaux[li]^.Frequ);
                      { définir la fréquence }
  if Canaux[li]^.Effet = $c then begin;
                      { Extra, sinon trop tôt ! }
    if Canaux[li]^.Operand > 63 then Canaux[li]^.Operand := 63;
    if Canaux[li]^.Operand < 1 then
    begin
      Canaux[li]^.volume := 0;
      u_VoiceVolume(li,0);
      U_StartVoice(li,Stop_Voice);
      stop_Thevoice[li] := true;
    end else begin
      Canaux[li]^.volume := Canaux[li]^.Operand;
      u_VoiceVolume(li,Canaux[li]^.volume);
      Runinf.Volumes[li] := 63;
    end;
  end else begin;
    if Canaux[li]^.volume > 63 then Canaux[li]^.volume := 63;
```

```
            voice_rampin(li,Canaux[li]^.volume);
            Runinf.Volumes[li] := 63;
          end;
          U_StartVoice(li,Stop_Voice);
                                  { Arrêter l'ancienne voix }
          u_Voicedata(Canaux[li]^.Mempos,Canaux[li]^.Loop_Start,
                      Canaux[li]^.Fin,li);
          if not stop_Thevoice[li] then begin;
                                  { lancer la nouvelle voix }
            U_StartVoice(li,Play_Voice+Bit8+Canaux[li]^.
                         Looping+Unidirect);
            Runinf.Frappe[li] := Canaux[li]^.volume * 4;
                                  { Pour égaliseur }
          end;
        end;                      { note frappée }
      end else begin;
        u_VoiceVolume(li,0);
      end;
    end;                          {for}
  end;
```

Le plus intéressant reste toutefois la réalisation des effets. Pour cela, nous devons distinguer entre la procédure d'initialisation des effets et la procédure de remise à jour des effets à chaque battement du MOD. Voyons d'abord la procédure d'initialisation, appelée par play_pattern_gus.

La procédure vérifie d'abord si l'effet n'est pas 0. Si c'est 0, la procédure s'arrête, puisqu'aucun effet n'est défini. On vérifie alors dans une boucle "Case" quel est l'effet demandé. Si vous définissez une procédure pour chacun des effets, vous pouvez aussi travailler ici avec une table "Pump", dans laquelle vous reporterez les adresses de toutes les procédures concernant les effets. Puisque le player destiné à la carte GUS n'est pas spécialement concerné par les questions de vitesse, nous pouvons faire confiance à la structure "Case". Elle présente en effet l'avantage d'être un peu plus lisible.

Comme dans toute structure Case, nous effectuerons les modifications nécessaires dans les paramètres du canal concerné, en fonction du numéro de l'effet qui a été prévu. Passons une fois en revue tous les effets intégrés, en espérant que les choses seront plus claires de cette façon :

Effet 0 - Appeggio

Dans cet effet, le son est joué successivement sur trois tons différents. Le ton de départ est celui qui est indiqué dans "Son_initial". Le nibble X est la première valeur d'incrémentation du ton. Le nibble Y est la seconde valeur. Sur cette base, il faut recalculer et définir la fréquence à chaque battement.

```
procedure Initialiser_Effets(nr : byte);
var swaplong      : longint;
    vibswap       : integer;
begin;
  if Canaux[nr]^.Effet = 0 then exit;
  case Canaux[nr]^.Effet of
    0 : begin;                    { Appegio }
          Canaux[nr]^.Appegpos := 0;
          Canaux[nr]^.Effetx := Canaux[nr]^.Operand shr 4;
          Canaux[nr]^.Effety := Canaux[nr]^.Operand and $0f;
          inc(Canaux[nr]^.Appegpos);
          case (Canaux[nr]^.Appegpos MOD 3) of
            0 : begin;            {ap = 3 !}
                  Canaux[nr]^.Son_init :=
                  Canaux[nr]^.Ton + Canaux[nr]^.Effety;
                end;
            1 : begin;            {ap = 1 !}
                  Canaux[nr]^.Son_init :=
                  Canaux[nr]^.Son;
                end;
            2 : begin;            {ap = 2 !}
                  Canaux[nr]^.Son_init :=
                  Canaux[nr]^.Son + Canaux[nr]^.Effetx;
                end;
          end;
          if Canaux[nr]^.Son_init < 1 then
            Canaux[nr]^.Son_init := 1;
          Canaux[nr]^.Freq :=
            longint(Voice_Base[14] div Canaux[nr]^.Son_init);
          u_VoiceFreq(nr,Canaux[nr]^.Frequ);
        end;
```

Effet 1 - Portamento Up

Avec "Portamento Up", la fréquence indiquée doit être diminuée à chaque battement de la valeur qui figure dans l'opérande. La fréquence ainsi obtenue doit être redéfinie à chaque battement.

```
1 : begin;                    { Portamento up }
      dec(Canaux[nr]^.Son_init,Canaux[nr]^.Operand);
      if Canaux[nr]^.Son_init < 1 then
        Canaux[nr]^.Son_init := 1;
      Canaux[nr]^.Freq :=
        longint(Voice_Base[14] div Canaux[nr]^.Son_init);
      u_VoiceFreq(nr,Canaux[nr]^.Freq);
    end;
```

Effet 2 - Portamento Down

Il s'agit de la même opération que précédemment, à cette différence près qu'il faut ici incrémenter la fréquence au lieu de la décrémenter.

```
2 : begin;                    { Portamento down }
      inc(Canaux[nr]^.Son_init,Canaux[nr]^.Operand);
      if Canaux[nr]^.Son_init < 1 then
        Canaux[nr]^.Son_init := 1;
      Canaux[nr]^.Freq :=
        longint(Voice_Base[14] div Canaux[nr]^.Son_init);
      u_VoiceFreq(nr,Canaux[nr]^.Freq);
    end;
```

Effet 3 - Tone Portamento

Dans cet effet, vous devez d'abord savoir si la note cible est supérieure ou inférieure à la note présente. En conséquence de quoi vous devrez incrémenter ou décrémenter cette dernière. La vitesse de modification est indiquée par l'opérande. La fréquence doit être modifiée à chaque battement et redéfinie sur la carte. L'initialisation nécessaire est intégrée dans l'interrogation Case au moyen d'une procédure particulière.

```
3 : begin;                    { Tone Portamento }
      EI_toneportamento(nr);
    end;
procedure EI_toneportamento(nr : byte);
```

```
{
 Init pour la procédure vibrato tirée du traitement des effets
}
begin;
  { Définit le facteur Inc }
  if Canaux[nr]^.Operand <> 0 then
  begin;
    if Canaux[nr]^.Son_init > Canaux[nr]^.Son_final then
    begin;
      Canaux[nr]^.slidespeed := -(Canaux[nr]^.Operand);
    end else begin;
      Canaux[nr]^.slidespeed := (Canaux[nr]^.Operand);
    end;
  end;
  if Canaux[nr]^.Son_init < 1 then
    Canaux[nr]^.Son_init := 1;
  Canaux[nr]^.Freq :=
    longint(Voice_Base[14] div Canaux[nr]^.Son_init);
  u_VoiceFreq(nr,Canaux[nr]^.Freq);
  oldv[nr] := Canaux[nr]^.Son_init;
end;
```

Effet 4 - Vibrato

Dans l'effet vibrato, vous devez d'abord connaître la vitesse du vibrato et sa profondeur, grâce aux nibbles X et Y de l'opérande. Vous appellerez ensuite la procédure du vibrato. Dans celle-ci, on définit un pointeur à l'intérieur de la table du vibrato. Lorsque ce pointeur dépasse le "Ran" (>64), il est décrémenté de 64 et revient dans les limites du tableau. A l'aide de ce pointeur, on obtient une valeur prise dans la table du vibrato. On multiplie cette valeur par la profondeur du vibrato et on divise par 256. On incrémente ensuite la fréquence du son de cette valeur. Cette procédure doit être appelée et la fréquence redéfinie à chaque battement.

```
    4 : begin;                    { Vibrato  *new* }
          Canaux[nr]^.vibx := Canaux[nr]^.Operand shr 4;
          Canaux[nr]^.viby := Canaux[nr]^.Operand and $0f;
          effet_vibrato(nr);
        end;
procedure effet_vibrato(nr : byte);
```

```
{
 Procédure vibrato tirée du traitement des effets
}
var vibswap : integer;
begin;
  inc(Canaux[nr]^.vibpos,Canaux[nr]^.vibx);
  if Canaux[nr]^.vibpos > 64 then
    dec(Canaux[nr]^.vibpos,64);
  vibswap :=
    (VibratoTable[Canaux[nr]^.vibpos] * Canaux[nr]^.viby) div 256;
  inc(Canaux[nr]^.Son_init,vibswap);
  if Canaux[nr]^.Son_init < 1 then
    Canaux[nr]^.Son_init := 1;
  Canaux[nr]^.Freq :=
    longint(Voice_Base[14] div Canaux[nr]^.Son_init);
  u_VoiceFreq(nr,Canaux[nr]^.Freq);
end;
```

Effet 5 - Note & Volume sliding

Cet effet peut être divisé en deux sections : d'une part le "volume sliding", d'autre part le "Portamento". Vous devez d'abord vérifier si l'opérande est inférieur à 0fh. Dans ce cas, vous avez affaire à un "Sliding down" et vous devez calculer la vitesse de celui-ci à l'aide de la formule

```
speed:=Opérande AND $0f
```

Sinon, c'est un "Sliding up" et vous devez réaliser un shift de 4 sur l'opérande pour obtenir la vitesse. Une fois que vous avez obtenu celle-ci, vous modifierez d'abord le volume de la voix. Veillez à ce que les valeurs demeurent dans les limites permises. La même chose vaut pour la fréquence. Vous devez faire attention à ce que la fréquence n'augmente pas indéfiniment ou qu'elle ne prenne pas des valeurs inférieures à 1. Indiquez alors le volume obtenu et la fréquence sur la carte GUS.

```
    5 : begin;                    {NOTE SLIDE + VOLUME SLIDE:   *new* }
          { init }
          if Canaux[nr]^.Operand <= $0f then
          begin;
            Canaux[nr]^.vslide := -(Canaux[nr]^.Operand AND $0f);
            Canaux[nr]^.slidespeed := -(Canaux[nr]^.Operand AND $0f);
          end else begin;
```

```
        Canaux[nr]^.vslide := (Canaux[nr]^.Operand shr 4);
        Canaux[nr]^.slidespeed := (Canaux[nr]^.Operand shr 4);
      end;
      { volume slide }
      inc(Canaux[nr]^.volume,Canaux[nr]^.vslide);
      if Canaux[nr]^.volume < 0 then Canaux[nr]^.volume := 0;
      if Canaux[nr]^.volume > 63 then Canaux[nr]^.volume := 63;
      u_VoiceVolume(Nr,Canaux[nr]^.volume);
      { Note slide }
      inc(Canaux[nr]^.Son_init,Canaux[nr]^.slidespeed);
      if Canaux[nr]^.Son_init < 1 then
        Canaux[nr]^.Son_init := 1;
      Canaux[nr]^.Frequ :=
        longint(Voice_Base[14] div Canaux[nr]^.Son_init);
      u_VoiceFreq(nr,Canaux[nr]^.Frequ);
    end;
```

Effet 6 - Vibrato & Volume sliding

Cet effet peut lui aussi être divisé en deux sous-effets. pour le "Volume sliding", on se reportera à ce qui a été dit pour l'effet 5. Pour le vibrato, vous pouvez utiliser les mêmes routines que pour l'effet vibrato (numéro 4).

```
6 : begin;                    { Vibrato & Volume slide  *new* }
      { init }
      Canaux[nr]^.vibx := Canaux[nr]^.Operand shr 4;
      Canaux[nr]^.viby := Canaux[nr]^.Operand and $0f;
      if Canaux[nr]^.Operand <= $0f then
      begin;
        Canaux[nr]^.vslide := -(Canaux[nr]^.Operand AND $0f);
      end else begin;
        Canaux[nr]^.vslide := (Canaux[nr]^.Operand shr 4);
      end;
      { volume slide }
      inc(Canaux[nr]^.volume,Canaux[nr]^.vslide);
      if Canaux[nr]^.volume < 0 then Canaux[nr]^.volume := 0;
      if Canaux[nr]^.volume > 63 then Canaux[nr]^.volume := 63;
      u_VoiceVolume(Nr,Canaux[nr]^.volume);
      { vibrato }
      effet_vibrato(nr);
```

```
    end;
```

Effet 7 - Tremolo

L'effet Tremolo est entièrement identique à l'effet vibrato dans son mode de fonctionnement. La différence est que l'on manipule ici non pas la fréquence mais le volume.

```
7 : begin;                    { tremolo  *new* }
      Canaux[nr]^.vibx := Canaux[nr]^.Operand shr 4;
      Canaux[nr]^.viby := Canaux[nr]^.Operand and $0f;
      inc(Canaux[nr]^.vibpos,Canaux[nr]^.vibx);
      if Canaux[nr]^.vibpos > 64 then
        dec(Canaux[nr]^.vibpos);
      vibswap :=
        (VibratoTable[Canaux[nr]^.vibpos] * Canaux[nr]^.viby)
                                                    div 256;
      inc(Canaux[nr]^.Volume,vibswap);
      if Canaux[nr]^.Volume < 0 then Canaux[nr]^.Volume := 0;
      if Canaux[nr]^.Volume > 63 then Canaux[nr]^.Volume := 63;
      u_VoiceVolume(nr,Canaux[nr]^.volume);
    end;
```

Effet 8 - Non utilisé

Cet effet n'est pas utilisé. Vous pouvez cependant vous en servir pour construire vos propres effets ou pour réaliser la synchronisation avec le graphique.

```
8 : begin;                    { non utilisé !!! }
      {
      Non utilisé officiellement. Se prête donc à la
      programmation pour synchroniser des événements dans une
      démo...}
    end;
```

Effet 9 - Sample-Offset

Avec la commande "Sample-Offset", l'opérande est interprété comme offset à l'intérieur du sample. L'octet désigne l'octet High de l'offset. Pour parvenir à celui-ci, vous devez donc multiplier l'opérande par 256. Additionnez l'offset de départ ainsi obtenu à la position du sample dans la RAM de la carte GUS et placez alors le canal sur cette position.

```
9 : begin;                    { Sample - Offset *new* }
      swaplong := longint((Canaux[nr]^.Operand+1)) * 256;
      Canaux[nr]^.Mempos := Canaux[nr]^.Mempos+swaplong;
      u_Voicedata(Canaux[nr]^.Mempos,Canaux[nr]^.Loop_Start,
                  Canaux[nr]^.Fin,nr);
      U_StartVoice(nr,Play_Voice+Bit8+Canaux[nr]^.Looping+
      Unidirect);
    end;
```

Effet 0ah - Volume sliding

C'est l'un des effets les plus faciles à réaliser. Si l'opérande a une valeur inférieure à 16, vous devez à chaque battement soustraire les 4 bits inférieurs de l'opérande à la valeur actuelle du volume. Sinon, vous devez shifter l'opérande de "quatre" unités vers la droite et ajouter cette valeur à chaque battement. Veillez à ce que le volume reste dans les limites permises.

```
$a : begin;                   { Volume sliding  *new* }
      if Canaux[nr]^.Operand <= $0f then
      begin;
        Canaux[nr]^.vslide := -(Canaux[nr]^.Operand AND $0f);
      end else begin;
        Canaux[nr]^.vslide := (Canaux[nr]^.Operand shr 4);
      end;
      inc(Canaux[nr]^.volume,Canaux[nr]^.vslide);
      if Canaux[nr]^.volume < 0 then Canaux[nr]^.volume := 0;
      if Canaux[nr]^.volume > 63 then Canaux[nr]^.volume := 63;
      u_VoiceVolume(Nr,Canaux[nr]^.volume);
    end;
```

Effet 0bh - Position Jump

Ici, l'opérande indique dans l'arrangement le numéro de la position à laquelle on doit sauter. Placez simplement la ligne actuelle à la fin du pattern. De cette façon, vous passerez au pattern suivant au prochain appel.

```
$b : begin;                   { Position Jump *ok* }
      runinf.Ligne := 64;
      runinf.Pattnr := Canaux[nr]^.Operand;
    end;
```

Effet 0ch - Set Note Volume

Cet effet définit le volume d'une voix. Affectez simplement au volume de la voix la valeur qui figure dans l'opérande. Veillez à ce que le volume reste dans les limites permises.

```
$c : begin;                    { Set Note Volume *ok* }
       if Canaux[nr]^.Operand > 63 then Canaux[nr]^.Operand := 63;
       if Canaux[nr]^.Operand < 1 then
       begin
         Canaux[nr]^.volume := 0;
         u_VoiceVolume(nr,0);
         U_StartVoice(nr,Stop_Voice);
         stop_Thevoice[nr] := true;
       end else begin
         Canaux[nr]^.volume := Canaux[nr]^.Operand;
         u_VoiceVolume(Nr,Canaux[nr]^.volume);
         Runinf.Volumes[nr] := 63;
       end;
     end;
```

Effet 0dh - Pattern Break

Avec cet effet, vous sautez au pattern suivant du fichier MOD. Si vous voulez programmer l'effet tout à fait correctement, vous devez utiliser une variable booléenne, que vous conserverez toujours sur TRUE lorsque vous voulez sauter à un pattern. Vous pouvez utiliser pour la ligne la valeur indiquée dans l'opérande. Pourtant, il suffit de faire venir la ligne à la fin du pattern, de façon à sauter automatiquement au pattern suivant.

```
$d : begin;                    { Pattern Break *ok* }
       runinf.Ligne := 64;
     end;
```

Effet 0eh - Commande d'effet étendue

L'effet 0eh contient plusieurs sous-effets. Le numéro du sous-effet s'obtient à partir des quatre bits supérieurs de l'opérande. Voici les sous-effets qui présentent un intérêt pour nous :

Sous-effet 1 - Fine Sliding Up

Cet effet fonctionne exactement comme l'effet "Portamento Up". Pourtant, on ne réalise pas ici une mise à jour à chaque battement. La fréquence est modifiée une seule fois au moment de l'initialisation.

```
$e : begin;                    { Effet étendu }
       case (Canaux[nr]^.Operand shr 4) of
         1 : begin;              { Fine slide up }
               dec(Canaux[nr]^.Son_init,Canaux[nr]^.Operand and
               $0f);
               if Canaux[nr]^.Son_init < 1 then Canaux[nr]^.
                                Son_init := 1;
               Canaux[nr]^.Freq :=
                 longint(Voice_Base[14] div Canaux[nr]^.Son_init);
               u_VoiceFreq(nr,Canaux[nr]^.Freq);
             end;
```

Sous-effet 2 - Fine Sliding Down

Cet effet est analogue au précédent, à cette différence qu'il fait descendre la note au lieu de la monter.

```
         2 : begin;              { Fine slide down }
               inc(Canaux[nr]^.Son_init,Canaux[nr]^.Operand and
                                                    $0f);
               if Canaux[nr]^.Son_init < 1 then Canaux[nr]^.
                                Son_init := 1;
               Canaux[nr]^.Freq :=
                 longint(Voice_Base[14] div Canaux[nr]^.Son_init);
               u_VoiceFreq(nr,Canaux[nr]^.Freq);
             end;
```

Sous-effet 9 - Retriggering Note

Dans cet effet, on refrappe la note après le nombre de battements indiqué dans l'opérande. Cela veut dire qu'on relance le sample à partir du début.

```
9 : begin;          { Retriggering !!! *new* }
      Canaux[nr]^.Retrig_count :=
        Canaux[nr]^.Operand and $0f;
    end;
```

Sous-effet 0ah - Fine Volume Slide Up

Le volume du canal est augmenté ici de la valeur indiquée par les quatre bits inférieurs. Il ne s'ensuit pas une remise à jour du volume dans les battements.

```
$a : begin;          { fine volume slide up }
      Canaux[nr]^.vslide := (Canaux[nr]^.Operand AND
                             $0f);
      inc(Canaux[nr]^.volume,Canaux[nr]^.vslide);
      if Canaux[nr]^.volume < 0 then Canaux[nr]^.
                             volume := 0;
      if Canaux[nr]^.volume > 63 then Canaux[nr]^.
                             volume := 63;
      u_VoiceVolume(Nr,Canaux[nr]^.volume);
    end;
```

Sous-effet 0bh - Fine Volume Slide Down

Le volume du canal est diminué de la valeur indiquée dans les quatre bits inférieurs. Il ne s'ensuit pas une remise à jour du volume dans les battements.

```
$b : begin;          { fine volume slide down }
      Canaux[nr]^.vslide := (Canaux[nr]^.
                             Operand AND $0f);
      dec(Canaux[nr]^.volume,Canaux[nr]^.vslide);
      if Canaux[nr]^.volume < 0 then Canaux[nr]^.
                             volume := 0;
      if Canaux[nr]^.volume > 63 then Canaux[nr]^.
                             volume := 63;
      u_VoiceVolume(Nr,Canaux[nr]^.volume);
    end;
```

Sous-effet 0ch - Cut Voice

Avec cet effet, vous arrêtez l'émission d'un canal. Dans notre player, il suffit de placer la variable Stop_TheVoice sur TRUE.

```
          $c : begin;          { Cut Voice *ok* }
                 stop_Thevoice[nr] := true;
               end;
           end;
        end;
```

Sous-effet 0fh - Set Speed

Cet effet définit la vitesse avec laquelle le fichier MOD sera exécuté. Si l'opérande a une valeur inférieure à 16, l'indication se réfère au Mod-Speed, sinon aux BPM. Dans le premier cas, on affecte à la variable "ticklimit" la valeur de l'opérande. Dans le second cas, on redéfinit la valeur des BPM par l'intermédiaire de la procédure nouv_interrupt_speed.

```
    $f : begin;                  { Set Speed  *ok* }
           if Canaux[nr]^.Operand <= $f then begin;
             ticklimit := Canaux[nr]^.Operand;
             runinf.speed := ticklimit;
           end else begin;
             runinf.bpm := Canaux[nr]^.Operand;
             nouv_interrupt_Speed(Canaux[nr]^.Operand);
           end;
         end;
  end;
end;
```

Les effets doivent bien sûr être initialisés, mais il faut aussi mettre à jour chaque battement. Les algorithmes nécessaires à cet effet sont en principe identiques à ceux de l'initialisation. La différence est que l'initialisation des variables devient le plus souvent inutile. Voici un aperçu rapide de la procédure qui est chargée de cette remise à jour.

```
procedure tick_effects;
var li : integer;
    vibswap : integer;
begin;
  for li := 1 to MOD_Voix do begin;
    if runinf.volumes[li] > 0 then
```

```
    dec(runinf.volumes[li]);
  case Canaux[li]^.Effet of       { Traitement des effets en cours
                                    d'exécution }
    0 : begin;
          inc(Canaux[li]^.Appegpos);
          case (Canaux[li]^.Appegpos MOD 3) of
            0 : begin;            {ap = 3 !}
                  Canaux[li]^.Son_init :=
                  Canaux[li]^.Son + Canaux[li]^.Effety;
                end;
            1 : begin;            {ap = 1 !}
                  Canaux[li]^.Son_init :=
                  Canaux[li]^.Son;
                end;
            2 : begin;            {ap = 2 !}
                  Canaux[li]^.Son_init :=
                  Canaux[li]^.Son + Canaux[li]^.Effetx;
                end;
           end;
        end;
    1 : begin;
        {!"! new }
          Canaux[li]^.Operand := Canaux[li]^.Operand and $0F;
        {!"! new end }
          dec(Canaux[li]^.Son_init,Canaux[li]^.Operand);
          if Canaux[li]^.Son_init < 1 then
            Canaux[li]^.Son_init := 1;
          Canaux[li]^.Freq :=
            longint(Voice_Base[14] div Canaux[li]^.Son_init);
          u_VoiceFreq(li,Canaux[li]^.Freq);
        end;
    2 : begin;
        {!"! new }
          Canaux[li]^.Operand := Canaux[li]^.Operand and $0F;
        {!"! new end }
          inc(Canaux[li]^.Son_init,Canaux[li]^.Operand);
          if Canaux[li]^.Son_init < 1 then
            Canaux[li]^.Son_init := 1;
```

```
            Canaux[li]^.Freq :=
              longint(Voice_Base[14] div Canaux[li]^.Son_init);
            u_VoiceFreq(li,Canaux[li]^.Freq);
          end;
      3 : begin;                          { Tone Portamento }
            E_toneportamento(li);
{
 ------------------------------------------------------------------------
procedure E_toneportamento(nr : byte);
{
 Procédure TonePortamento tirée du traitement des effets
}
begin;
  if Canaux[nr]^.slidespeed < 0 then
  begin
    inc(Canaux[nr]^.Son_init,Canaux[nr]^.slidespeed);
    if Canaux[nr]^.Son_init < Canaux[nr]^.Son_final then
      Canaux[nr]^.Son_init := Canaux[nr]^.Son_final;
  end else begin
    inc(Canaux[nr]^.Son_init,Canaux[nr]^.slidespeed);
    if Canaux[nr]^.Son_init > Canaux[nr]^.Son_final then
      Canaux[nr]^.Son_init := Canaux[nr]^.Son_final;
  end;
  if Canaux[nr]^.Son_init < 1 then
    Canaux[nr]^.Son_init := 1;
  Canaux[nr]^.Freq :=
    longint(Voice_Base[14] div Canaux[nr]^.Son_init);
  u_VoiceFreq(nr,Canaux[nr]^.Freq);
  oldv[nr] := Canaux[nr]^.Son_init;
end;
 ------------------------------------------------------------------------
}
          end;
      4 : begin;                          { vibrato  *new* }
            effet_vibrato(li);
          end;
      5 : begin;
            { volume slide }
```

```
        inc(Canaux[li]^.volume,Canaux[li]^.vslide);
        if Canaux[li]^.volume < 0 then Canaux[li]^.volume := 0;
        if Canaux[li]^.volume > 63 then Canaux[li]^.volume := 63;
        u_VoiceVolume(li,Canaux[li]^.volume);
        { Note slide }
        inc(Canaux[li]^.Son_init,Canaux[li]^.slidespeed);
         if Canaux[li]^.Son_init < 1 then
           Canaux[li]^.Son_init := 1;
        Canaux[li]^.Freq :=
          longint(Voice_Base[14] div Canaux[li]^.Son_init);
        u_VoiceFreq(li,Canaux[li]^.Freq);
      end;
  6 : begin;
        { volume slide }
        inc(Canaux[li]^.volume,Canaux[li]^.vslide);
        if Canaux[li]^.volume < 0 then Canaux[li]^.volume := 0;
        if Canaux[li]^.volume > 63 then Canaux[li]^.volume := 63;
        u_VoiceVolume(li,Canaux[li]^.volume);
        { vibrato }
        inc(Canaux[li]^.vibpos,Canaux[li]^.vibx);
        if Canaux[li]^.vibpos > 64 then
         dec(Canaux[li]^.vibpos);
        vibswap :=
          (VibratoTable[Canaux[li]^.vibpos] * Canaux[li]^.
                                      viby) div 256;
        inc(Canaux[li]^.Son_init,vibswap);
         if Canaux[li]^.Son_init < 1 then
           Canaux[li]^.Son_init := 1;
        Canaux[li]^.Freq :=
          longint(Voice_Base[14] div Canaux[li]^.Son_init);
        u_VoiceFreq(li,Canaux[li]^.Freq);
      end;
  7 : begin;                    { tremolo  *new* }
         inc(Canaux[li]^.vibpos,Canaux[li]^.vibx);
         if Canaux[li]^.vibpos > 64 then
           dec(Canaux[li]^.vibpos);
         vibswap :=
```

```
            (VibratoTable[Canaux[li]^.vibpos] * Canaux[li]^.viby)
                                                        div 256;
          inc(Canaux[li]^.Volue,vibswap);
          if Canaux[li]^.Volume < 0 then Canaux[li]^.Volume := 0;
          if Canaux[li]^.Volume > 63 then Canaux[li]^.Volume := 63;
          u_VoiceVolume(li,Canaux[li]^.volume);
        end;
    8 : begin;                  { not used !!! }
        end;
    $a : begin;                 { Volume sliding   **new* }
          inc(Canaux[li]^.volume,Canaux[li]^.vslide);
          if Canaux[li]^.volume < 0 then Canaux[li]^.volume := 0;
          if Canaux[li]^.volume > 63 then Canaux[li]^.volume := 63;
          u_VoiceVolume(li,Canaux[li]^.volume);
        end;
    $e : begin;                 { Effet étendu }
         case (Canaux[li]^.Operand shr 4) of
            9: begin;           { Retriggering !!! }
                 if Canaux[li]^.Operand and $0f <> 0 then begin;
                   dec(Canaux[li]^.Retrig_count);
                   if Canaux[li]^.Retrig_count = 0 then begin;
                     Canaux[li]^.Retrig_count := Canaux[li]^.
                                  Operand and $0f;
                     u_Voicedata(Canaux[li]^.Mempos,Canaux[li]^.
                                                Loop_Start,
                           Canaux[li]^.Fin,li);
                     U_StartVoice(li,Play_Voice+Bit8+Canaux[li]^.
                                  Looping+
                     Unidirect);
                   end;
                 end;
               end;
          end;
        end;
    end;
  end;
end;
```

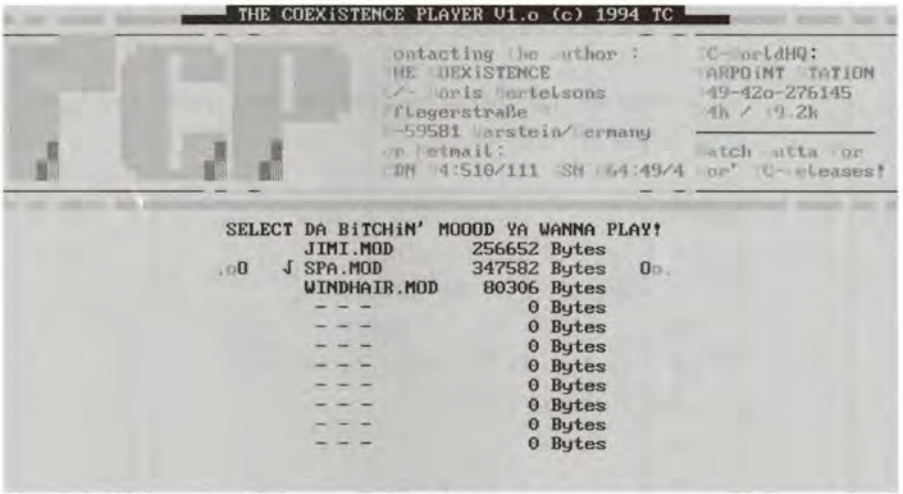
Nous en avons fini avec les procédures essentielles de l'unité GUS_MOD. Vous savez maintenant comment charger et exécuter un MOD. Il ne vous reste plus qu'à apprendre comment sortir à nouveau le fichier MOD de la mémoire. Vous vous servirez pour cela de la procédure _gus_mod_quitter. Celle-ci appelle d'abord la procédure "timerint_rest", que vous avez déjà rencontrée. La mémoire réservée est ensuite libérée par la procédure "dispose_mod". Cette procédure arrête d'abord tous les canaux de la carte. Elle libère ensuite la mémoire occupée par les patterns, les informations sur les canaux et les informations sur les instruments, au moyen de boucles adéquates.

```
procedure dispose_mod;
{
 Supprime un MOD chargé en mémoire principale. Les samples sur la GUS
 ne sont PAS supprimés.
}
begin;
  for i := 0 to 31 do begin;
    U_StartVoice(i,Stop_Voice);
  end;
  for i := 1 to Vh.Num_Patts do begin;
    dos_freemem(Pattern[i]);
  end;
  for i := 0 to 15 do begin;
    dispose(Canaux[i]);
  end;
  for i := 0 to 31 do begin;
     dispose(Instruments[i]);
  end;
end;

procedure _gus_mod_quitter;
{
 Met fin à la sortie d'un MOD
}
begin;
  timerint_rest;
  dispose_mod;
end;
```

C'est tout ce que nous devons savoir pour programmer un player MOD avec la carte GUS. Nous vous souhaitons un franc succès dans vos essais de développement à partir des éléments dont vous disposez. Que diriez-vous par exemple d'un player MOD en mode graphique ?

Le player GUS TCP de "The CoExistence" pour les mordus

Nous nous contenterons ici pour commencer d'un player GUS en mode texte. Pour recevoir plus d'informations à l'écran, nous choisirons le mode en 50 lignes.



Sélection d'un fichier MOD dans le player TCP

Le player est d'un fonctionnement relativement simple. On initialise d'abord la carte GUS avec _gus_init_env. Quand la procédure ne trouve pas de carte GUS ou quand une erreur surgit pendant l'initialisation, on obtient l'affichage d'un texte d'aide. Celui-ci a été créé avec The Draw et déposé en format Object de Pascal. Ce format est une pure reproduction de la mémoire d'écran. Vous pouvez lier le fichier objet correspondant dans votre programme Pascal et adresser l'image sous la procédure indiquée. Si vous avez placé par exemple le texte à afficher sous le nom de procédure "helptext", vous pouvez représenter l'écran ANSI à l'aide de :

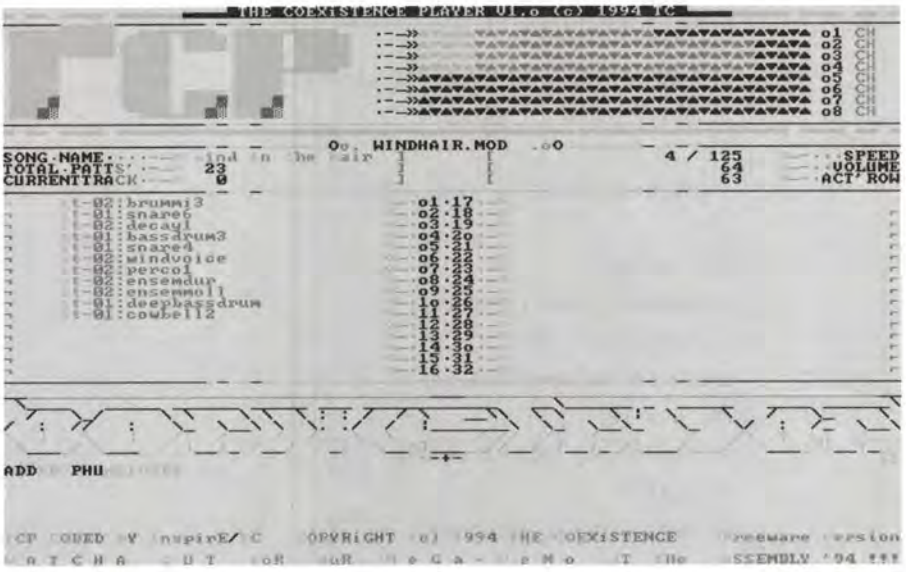
```
move (@helptxt^,ptr($B800,0)^,80*50*2)
```

Lorsque l'initialisation s'est faite avec succès, on réserve de la mémoire pour la structure d'exécution du fichier. Cette structure fait partie de l'unité "fselect", que vous trouverez sur le CD qui accompagne ce livre. On vérifie ensuite si un nom de fichier a été indiqué par l'intermédiaire de la ligne de commande. C'est la fonction check_commandline qui s'en charge. Elle ren-

voie la valeur TRUE lorsqu'un nom de fichier a été indiqué. Sinon, on peut sélectionner un ou plusieurs fichiers par l'intermédiaire de l'unité fselect.

On réalise ensuite quelques initialisations nécessaires pour l'unité MOD au moyen de _gus_initialiser. On affiche l'ANSI pour l'écran principal. On appelle la procédure write-phunliners. Cette procédure lit trois lignes dans le fichier texte PHUN.TXT et les affiche au bas de l'écran. Les entrées dans le fichier PHUN.TXT sont des BBS-On-liner. Le fichier peut être édité librement.

Après l'affichage des "phunliners", on charge le fichier MOD par l'intermédiaire _gus_modload. Nous avons déjà examiné cette fonction en détail. Une fois que le fichier MOD a été chargé, le nom du fichier MOD, ainsi que toutes les informations importantes (noms des instruments !) sont affichés à l'écran. Lorsque toutes ces opérations graphiques ont été réalisées, l'émission du fichier MOD commence avec _gus_modstart.



L'écran principal du Player de The CoExistence

On appelle alors "User". Cette procédure met à jour régulièrement l'écran (les barres indiquant le volume) et s'occupe en même temps des saisies éventuelles de l'utilisateur. Quand on la quitte, le fichier MOD actuel est supprimé de la mémoire. Selon que l'utilisateur avait ou non sélectionné le fichier dans le menu de sélection présenté au début, il revient dans ce menu ou il quitte le player.

Le codage du player en soi est trivial. Il repose pour l'essentiel sur l'unité GUS_MOD. Voici sa source :

```
{$A+,B-,D+,E+,F+,G+,I+,L+,N-,O-,P-,Q-,R-,S+,T-,V+,X+,Y+}
{$M 16384,0,250000}
program gusdemo;
uses crt,dos,design,fselect, gus_mod;
type
  Pphun = ^TPhun;
  TPhun = array[0..799] of string[80];
const mod_chem = '';
const Programm_quitter : boolean = false;
var phun : Pphun;
    phuncount : integer;
    modify_voice : integer;
    i : integer;
    Les_files : Pfileselect_struct;
    curr_modnr : integer;
{
  Intégration des écrans ANSI
}
{$L tcpans}
procedure tcpans; external;
{$L we_are}
procedure we_are; external;
{$L buy_it}
procedure buy_it; external;
{$L call}
procedure call; external;
{$L helptxt}

procedure helptxt; external;
function fichier_exists(dname : string) : boolean;
{
 Vérifie si le fichier transmis existe
}
var dumf : file;
begin;
  {$I-}
```

```
  assign(dumf,dname);
  reset(dumf,1);
  {$I+}
  if IOResult <> 0 then
    fichier_exists := false
  else begin;
    fichier_exists := true;
    close(dumf);
  end;
end;

procedure color_writeln(s : string);
{
 Pour sortir une chaîne dans la combinaison de couleurs TC
}
var colpos,li : integer;
begin;
  colpos := 1;
  for li := 1 to length(s) do begin;
    if s[li] = ' ' then colpos := 0;
    inc(colpos);
    case colpos of
        1..2 : begin;
                 textcolor(8);
               end;
        3..4 : begin;
                 textcolor(2);
               end;
      5..$ff : begin;
                 textcolor(10);
               end;
    end;
    write(s[li]);
  end;
end;

procedure write_nomfichier(s : string);
{
```

```
 Affiche le nom de fichier de la chanson, centré
}
var li,slen : integer;
begin;
  gotoxy(33,13);
  while pos('\',s) <> 0 do begin;
    delete(s,1,pos('\',s));
  end;
  slen := length(s);
  slen := (15 - slen) div 2;
  for li := 1 to slen do s := ' '+s;
  write(s);
end;

procedure write_phunliners;
{
 Lit trois lignes dans le fichier "Phun.txt" et les affiche
 à l'écran. Le fichier "Phun.txt" est un fichier texte
 que l'on peut éditer librement !
}
var tf : text;
begin;
  randomize;
  if not fichier_exists('phun.txt') then exit;
  assign(tf,'phun.txt');
  reset(tf);
  phuncount := 0;
  {$I+}
  if ioresult = 0 then begin;
    while not eof(tf) do begin;
      readln(tf,phun^[phuncount]);
      inc(phuncount);
    end;
    close(tf);
    gotoxy(3,43);
    color_writeln(phun^[random(phuncount)]);
    gotoxy(3,44);
    color_writeln(phun^[random(phuncount)]);
```

```
    gotoxy(3,45);
    color_writeln(phun^[random(phuncount)]);
  end;
  {$I-}
end;

procedure display_modinfos;
{
 Affiche le nom des instruments dans le module actuel
}
var li : integer;
begin;
  textcolor(14);
  textbackground(black);
  for li := 1 to 16 do begin;
    gotoxy(6,17+li);
    color_writeln(Instruments[li]^.nom);
    gotoxy(50,17+li);
    color_writeln(Instruments[li+16]^.nom);
  end;
end;

procedure exit_program;
{
 Avant de quitter le programme, vite une dernière indication sur le
 TC WHQ, la "Farpoint Station" (04202 76145)...
}
begin;
  display_ansi(@call,co80+font8x8);
  cursor_off;
  repeat until keypressed;
  while keypressed do readkey;
  cursor_on;
  asm mov ax,03; int 10h; end;
  halt;
end;
```

```
procedure prochain_mod;
{
 Lance la sortie du MOD sélectionné suivant
}
begin;
  _gus_mod_quitter;
  inc(curr_modnr);
  if curr_modnr > Les_Files^.nofiles then
    curr_modnr := 1;
  if not _gus_modload(Les_Files^.fn[curr_modnr]) then begin;
    clrscr;
    gotoxy(10,10);
    write('Sorry dude, Cant''t handle this MOD-File');
    delay(1200);
    exit_program;
  end;
  display_ansi(@tcpans,co80+font8x8);
  cursor_off;
  write_phunliners;
  write_nomfichier(Les_Files^.fn[curr_modnr]);
  display_modinfos;
  fillchar(Play_Chanel,14,1);
  _gus_modstart;
end;

procedure display_we_are;
{
 Sortir un ANSI avec Infos par le groupe THE COEXISTENCE
}
begin;
  display_ansi(@we_are,co80+font8x8);
  cursor_off;
  repeat until keypressed;
  while keypressed do readkey;
  display_ansi(@tcpans,co80+font8x8);
  cursor_off;
  write_phunliners;
  write_nomfichier(Les_Files^.fn[curr_modnr]);
```

```
  display_modinfos;
end;

procedure display_buy_it;
{
 Affiche une réclame pour le livre PC Interdit
}
begin;
  display_ansi(@buy_it,co80+font8x8);
  cursor_off;
  repeat until keypressed;
  while keypressed do readkey;
  display_ansi(@tcpans,co80+font8x8);
  cursor_off;
  write_phunliners;
  write_nomfichier(Les_Files^.fn[curr_modnr]);
  display_modinfos;
end;

procedure handle_keys(key1,key2 : char);
{
 Réagit aux saisies de l'utilisateur
}
var pchan : byte;
begin;
  case key1 of
    #00 : begin;
            case key2 of
              #45 : begin;
                      Programm_quitter := true;
                    end;
              #72 : begin;
                      if modify_voice > 1 then
                        dec(modify_voice);
                    end;
              #80 : begin;
                      if modify_voice < Modinf.Voix then
                        inc(modify_voice);
```

```
                end;
          #75 : begin;        { cursor left  }
                  runinf.Ligne := 64;
                  dec(runinf.Pattnr,2);
                  if runinf.Pattnr < -1 then runinf.Pattnr := -1;
                end;
          #77 : begin;        { cursor right  }
                  runinf.Ligne := 64;
                  inc(runinf.Pattnr);
                end;
        end;
      end;
#27 : begin;
        Programm_quitter := true;
      end;
#32,
'W',
'w',
'I',
'i' : begin;
        display_we_are;
      end;
'D',
'd',
'b',
'B' : begin;
        display_buy_it;
      end;
'L',
'l' : begin;
        chpos[modify_voice] := 1;
        _gus_set_chanelpos;
      end;
'R',
'r' : begin;
        chpos[modify_voice] := 15;
        _gus_set_chanelpos;
      end;
```

```
'M',
'm' : begin;
        chpos[modify_voice] := 7;
        _gus_set_chanelpos;
      end;
'U',
'u' : begin;
        if Modinf.Voix = 4 then
        begin
          chpos[1] := 2;
          chpos[2] := 5;
          chpos[3] := 9;
          chpos[4] := 12;
        end;
        if Modinf.Voix = 8 then
        begin
          chpos[1] := 1;
          chpos[2] := 3;
          chpos[3] := 5;
          chpos[4] := 7;
          chpos[5] := 7;
          chpos[6] := 9;
          chpos[7] := 11;
          chpos[8] := 13;
        end;
        _gus_set_chanelpos;
      end;
',' : begin;                    { vers la gauche }
        if chpos[modify_voice] > 1 then
          dec(chpos[modify_voice]);
        _gus_set_chanelpos;
      end;
'.' : begin;                    { vers la droite !!! }
        if chpos[modify_voice] < 15 then
          inc(chpos[modify_voice]);
        _gus_set_chanelpos;
      end;
'1'..
```

```
    '8' : begin;
            pchan := ord(key1)-48;
            if Play_Chanel[pchan] = 1 then begin;
              Play_Chanel[pchan] := 0;
              textcolor(10);   gotoxy(77,2+pchan);
              write('M');      textcolor(2);
              write('UTE');
            end else begin;
              Play_Chanel[pchan] := 1;
              textcolor(10);   gotoxy(77,2+pchan);
              write('C');      textcolor(2);
              write('H  ');
            end;
          end;
    'n',
    'N' : begin;
            prochain_mod;
          end;
  end;
end;

procedure screen_update;
const colvals : array[1..35] of byte =
   (08,08,08,08,08,02,02,02,02,10,10,10,10,10,10,10,10,
    10,10,10,10,10,10,10,10,10,10,10,10,10,05,05,05,05,05);
var volstr : string[66];
    li : integer;
    auss : integer;
begin;
  { Remettre à jour les barres indiquant le volume }
  for li := 1 to Modinf.Voix do begin;
    for auss := 1 to round(Runinf.Volumes[li] / 1.78) do begin;
      screen[li+2,37+auss].a := colvals[auss];
    end;
    for auss := round(Runinf.Volumes[li] / 1.78)  to 36 do begin;
      screen[li+2,38+auss].a := 7;
    end;
  end;
```

```
  { Définir l'arrière-plan en couleurs pour la flèche }
  for li := 1 to 8 do begin;
    if li = modify_voice then begin;
      screen[2+li,34].a := 05;
      screen[2+li,35].a := 05;
      screen[2+li,36].a := 05;
      screen[2+li,37].a := 05;
    end else begin;
      screen[2+li,34].a := 07;
      screen[2+li,35].a := 07;
      screen[2+li,36].a := 07;
      screen[2+li,37].a := 07;
    end;
  end;
  { Afficher des informations en temps réel sur le MOD }
  gotoxy(18,14);
  color_writeln(Modinf.Titre);
  textcolor(7);
  gotoxy(18,16);
  write(runinf.pattnr:3);
  gotoxy(64,16);
  write(runinf.ligne:3);
  gotoxy(64,15);
  write(64:3);
  gotoxy(18,15);
  write(modinf.Nb_Patt:3);
  gotoxy(60,14);
  write(runinf.speed,' / ',runinf.bpm);
end;

procedure user;
{
 Vérifie les saisies aux claviers et remet à jour l'écran
}
var ch1,ch2 : char;
begin;
  repeat
    ch1 := #255;
```

```
    ch2 := #255;
    if keypressed then begin;
      ch1 := readkey;
      if keypressed then ch2 := readkey;
      handle_keys(ch1,ch2);
    end;
   screen_update;
  until Programm_quitter;
end;

procedure display_help;
{
 Affiche l'aide Ansi, même quand aucune carte GUS n'a été trouvée
}
begin;
  display_ansi(@helptxt,co80+font8x8);
  cursor_off;
  repeat until keypressed;
  while keypressed do readkey;
  exit_program;
end;

function check_commandline : boolean;
{
 retourne true, lorsqu'un nom de module a été donné
}
var pst : string;
    ist_mod : boolean;
    li : integer;
    retval : boolean;
begin;
  retval := false;
  for li := 1 to 9 do begin;
    pst := paramstr(li);
    ist_mod := true;
    if (pos('-h',pst) <> 0) or (pos('-H',pst) <> 0) or
       (pos('-?',pst) <> 0) then
    begin;
```

```
      ist_mod := false;
      display_help;
    end;
    if (pst <> '') and ist_mod then begin;
      if pos('.',pst) = 0 then pst := pst + '.mod';
      if fichier_exists(pst) then { mode valide }
      begin
        inc(Les_Files^.nofiles);
        Les_Files^.fn[Les_Files^.nofiles] := pst;
        retval := true;
      end;
    end;
  end;
  check_commandline := retval;
end;
begin;
  cursor_off;
  clrscr;
  if not _gus_init_env then display_help;
  new(Les_Files);
  new(phun);
  Les_Files^.path := mod_chem;
  Les_Files^.Mask := '*.mod';
  Les_Files^.sx   := 24;
  Les_Files^.sy   := 10;
  Les_Files^.nofiles := 0;
  Les_Files^.Titre := 'Choisir un fichier MOD!!!';
  modify_voice := 1;
  for i := 1 to 30 do
    Les_Files^.fn[i] := '---';
  save_screen;
  if not check_commandline then begin;
    select_packfichiers(Les_Files);
    repeat
      restore_screen;
      if Les_Files^.fn[1] = '---' then exit_program;
        _gus_initialiser;
      display_ansi(@tcpans,co80+font8x8);
```

```
      cursor_off;
      write_phunliners;
      curr_modnr := 1;
      if not _gus_modload(Les_Files^.fn[1]) then begin;
        clrscr;
        gotoxy(10,10);
        write('Sorry dude, Cant''t handle this MOD-File');
        delay(1200);
        exit_program;
      end;
      write_nomfichier(Les_Files^.fn[1]);
      display_modinfos;
      fillchar(Play_Chanel,14,1);
      _gus_modstart;
      user;
      _gus_mod_quitter;
      dispose(Les_Files);
      new(Les_Files);
      Les_Files^.path := Mod_chem;
      Les_Files^.Mask := '*.mod';
      Les_Files^.sx   := 24;
      Les_Files^.sy   := 10;
      Les_Files^.nofiles := 0;
      for i := 1 to 30 do
        Les_Files^.fn[i] := '---';
      Programm_quitter := false;
      select_packfichiers(Les_Files);
    until Les_Files^.fn[1] = '---';
      dispose(Les_Files);
    dispose(phun);
    exit_program;
  end else begin;
  restore_screen;
  if Les_Files^.fn[1] = '---' then exit_program;
  _gus_initialiser;
  display_ansi(@tcpans,co80+font8x8);
  cursor_off;
  write_phunliners;
```

```
  curr_modnr := 1;
  _gus_modload(Les_Files^.fn[1]);
  write_nomfichier(Les_Files^.fn[1]);
  display_modinfos;
  fillchar(Play_Chanel,14,1);
  _gus_modstart;
  user;
  _gus_mod_quitter;
  dispose(Les_Files);
  dispose(phun);
  exit_program;
  end;
end.
```

16. Utilitaires en shareware

Dans ce chapitre, nous allons présenter quelques très bons utilitaires qui vous rendront de grands services. Vous y trouverez un choix conséquent de logiciels en shareware, depuis l'Assembler A86 jusqu'à Povray.

Pour ceux qui aiment les programmes Windows, nous dirons aussi quelques mots des programmes de présentation graphique des fichiers, connus sous les noms de Graphics Workshop et Paintshop Pro.

16.1. Retrouvez de la place sur votre disque dur - ARJ 2.41

Le logiciel ARJ 2.41 est un programme de compression des fichiers, exactement comme PKZIP. Après ce dernier, c'est sans doute le format de compactage le plus courant. Les fichiers compactés avec ce logiciel portent l'extension ARJ ou - lorsque les données sont distribuées sur plusieurs disquettes - les extensions A01, A02, etc.

L'utilisation de ARJ

Lorsque vous entrez ARJ derrière le prompt de DOS, vous obtenez l'écran d'aide suivant :

```
ARJ 2.41a Copyright (c) 1990-93 Robert K Jung. Jul 10 1993
*** This SHAREWARE program is NOT REGISTERED for use in a business, commercial,
*** government, or institutional environment except for evaluation purposes.

List of frequently used commands and switches.  Type ARJ -? for more help.

Usage:     ARJ <command> [-<sw> [-<sw>...]] <archive_name> [<file_names>...]
Examples:  ARJ a -e archive, ARJ e archive, ARJ l archive *.doc
<Commands>
  a: Add files to archive                  m: Move files to archive
  d: Delete files from archive             t: Test integrity of archive
  e: Extract files from archive            u: Update files to archive
  f: Freshen files in archive              v: Verbosely list contents of archive
  l: List contents of archive              x: eXtract files with full pathname
<Switches>
  c: skip time-stamp Check                 r: Recurse subdirectories
  e: Exclude paths from names              s: set archive time-Stamp to newest
  f: Freshen existing files                u: Update files (new and newer)
  g: Garble with password                  v: enable multiple Volumes
  i: with no progress Indicator            w: assign Work directory
  m: with Method 0, 1, 2, 3, 4             x: eXclude selected files
  n: only New files (not exist)            y: assume Yes on all queries
```

Voici la signification des différents paramètres :

a: Add files to archive

Cette fonction permet de créer une archive ou d'ajouter des fichiers à une archive existante.

Exemple : `C:\>ARJ a TEST.ARJ C:\TEST\*.*`

Cette commande ajoute tous les fichiers du répertoire TEST à l'archive TEST.ARJ. Au cas où l'archive TEST.ARJ n'existe pas, elle est créée automatiquement.

m: Move files to archive

Cette fonction permet de recevoir des fichiers dans l'archive et de supprimer ensuite les fichiers originaux si le compactage a réussi.

Exemple : `C:\>ARJ m TEST.ARJ C:\TEST\*.*`

Tous les fichiers seront supprimés une fois qu'ils auront été reçus correctement dans l'archive.

d: Delete files from archive

ARJ autorise aussi une suppression des fichiers dans une archive existante. On utilise pour cela le paramètre d.

Exemple : `C:\>ARJ d TEST.ARJ TEST.TXT`

Le fichier TEST.TXT est ainsi supprimé de l'archive TEST.ARJ. Vous pouvez aussi utiliser des jokers, afin de supprimer plusieurs fichiers qui ont les mêmes propriétés :

Exemple : `C:\>ARJ d TEST.ARJ *.TXT`

t: Test integrity of archive

Le paramètre t permet de tester l'intégrité, c'est-à-dire l'état, de l'archive ARJ. ARJ examine alors si l'archive est détruite ou endommagée et elle affiche les messages d'erreur correspondants.

Exemple : `C:\>ARJ t TEST.ARJ`

Si l'archive est endommagée, ARJ annonce les erreurs. Vous pouvez prendre alors les mesures nécessaires pour réparer l'archive ARJ.

e: Extract files from archive

Cette commande permet de décompacter un ou plusieurs fichiers dans le répertoire actuel (ou dans le répertoire indiqué). ARJ vous demande éventuellement si des fichiers existants dans ce répertoire doivent être écrasés. Si vous répondez N ou NO, ARJ vous demande d'introduire un nouveau nom pour le fichier à décompacter.

Exemple : `C:\>ARJ e TEST.ARJ C:\TEST *.TXT`

Cette ligne de commande provoque le décompactage de tous les fichiers portant l'extension TXT dans l'archive TEST.ARJ et de les placer dans le répertoire C:\TEST.

u: Update files to archive

Permet de remplacer les anciens fichiers de l'archive par des fichiers remis à jour et d'ajouter de nouveaux fichiers.

Exemple : `C:\>ARJ u TEST.ARJ`

f: Refresh files in archive

La fonction f permet de renouveler les fichiers de l'archive comme le fait la fonction "Update". Toutefois, les seuls fichiers renouvelés ici sont ceux qui sont plus anciens que les dates sélectionnées pour les supports de données.

Exemple : `C:\>ARJ f TEST.ARJ *.TXT *.DOC`

En renouvelant les fichiers, il vaut mieux toutefois utiliser les mêmes fonctions que pour le renouvellement de l'archive :

```
C:\>ARJ a TEST.ARJ C:\TEST\ *.* -rRC:\>ARJ f TEST.ARJ C:\TEST\
*.* -r
```

v: Verbosely list contents of archive

Cette commande vous montre le contenu de l'archive avec les noms de répertoire au complet ainsi que les commentaires de l'archive.

Exemple : `C:\>ARJ v TEST.ARJ`

l: List contents of archive

Permet également d'obtenir le contenu de l'archive. Toutefois, contrairement à ce qui se passe avec la fonction v, les fichiers sont affichés dans l'ordre dans lequel ils ont été sauvegardés.

Exemple : `C:\>ARJ l TEST.ARJ -jp`

Si vous utilisez cette commande, une pause viendra s'insérer après chaque page d'écran, comme pour la commande v avec l'option -jp.

x: eXtract files with full pathname

Cette commande permet de décompacter un ou plusieurs fichiers de l'archive dans les répertoires, conservés en même temps dans l'archive.

Exemple : `C:\>ARJ x TEST.ARJ`

r: Recurse subdirectories

Cette instruction permet d'inclure tous les sous-répertoires dans le fichier compacté ou les fichiers spécifiés dans ces sous-répertoires au moyen des jokers.

e: Exclude paths from names

Cette option vous permet de ne pas sauvegarder les noms de répertoire dans l'archive. Vous vous dispensez ainsi de connaître au décompactage les répertoires dans lesquels les fichiers étaient déposés.

g: Garble with password

Sauvegardez votre archive à l'aide d'un mot de passe en utilisant l'option g. Vous pouvez éviter ainsi tout accès non autorisé.

Exemple : `C:\>ARJ a TEST.ARJ -gTEST`

v: enable multiple Volumes

Cette option est l'une des plus intéressantes dans le pack logiciel ARJ. Elle vous permet de créer ce qu'on appelle des volumes multiples, c'est-à-dire une archive divisée en plusieurs archives, afin de pouvoir la partager sur plusieurs disquettes.

Exemple : `C:\>ARJ a -v720 TEST.000`

On crée ici une archive dont la taille est limitée à 720 Ko.

y: assume Yes on all queries

ARJ demande par exemple avant d'écraser un fichier si vous souhaitez qu'il en soit ainsi. Si vous voulez sauter la question, faites intervenir simplement l'option y.

Exemple : `C:\>ARJ e -y TEST.ARJ`

ARJ - un meilleur choix que PKZIP ?

ARJ est certainement l'un des meilleurs logiciels de compactage. Il possède toutefois certains défauts, en comparaison avec PKZIP :

- ARJ ne supporte pas la mémoire EMS/XMS. En cas de mémoire insuffisante, vous ne pouvez donc pas travailler avec ARJ.
- ARJ ne reconnaît pas et n'utilise pas de DPMI (DOS Protected Mode Interface).

Malgré toutes ses qualités, ARJ ne peut donc venir qu'au second rang derrière PKZIP. Nous le conseillerons en priorité uniquement dans le cas où vous devez distribuer une archive sur plusieurs disquettes.

16.2. EPARGNER DE LA PLACE AVEC PKLITE

Le programme de compression PKLite est un utilitaire qui rend de grands services. Il est disponible en shareware et on peut l'utiliser pour des projets non commerciaux. Il comprime les fichiers EXE en leur laissant la faculté d'être des fichiers exécutables. Les programmes sont décompactés pendant leur fonctionnement.

PKLite domine sa catégorie en raison de sa vitesse et de la taille des programmes générés. Les marques de logiciel utilisent pour la plupart cette version commerciale pour réduire la taille de leurs logiciels.

PKLite sera appelé de la manière suivante :

```
PKLITE [Options] Fichier_original [Fichier_cible]
```

Fichier_original est le nom du fichier EXE à compacter. Si le fichier compacté porte le même nom, vous n'avez pas besoin d'ajouter le dernier paramètre. Sinon, vous pouvez entrer le nouveau nom. Vous disposez en outre des options suivantes :

-a Les fichiers ayant des overlays internes seront toujours comprimés. De plus, le programme exécutera une optimisation de la "relocation".

-b Quand on indique cette option, PKLite crée automatiquement un fichier backup avant de remplacer un fichier. Quand vous n'êtes pas tout à fait sûr que le programme souhaité se laissera comprimer sans problème, il est préférable d'utiliser cette option.

-e Dans la version en ligne de commande, vous pouvez indiquer au moyen de cette option que vous voulez choisir une méthode de compression particulière. Les données comprimées de cette façon ne peuvent alors plus être décompactées par PKLite.

-l Si vous vous intéressez aux détails de la licence de PKLite, vous pouvez demander ici leur affichage.

-n Quand vous indiquez cette option, les fichiers programme contenant des overlays ne sont pas comprimés. De même, le logiciel ne tente pas d'optimiser les relocations.

-o Cette commande est en quelque sorte l'inverse de l'option -b. Quand vous l'indiquez, tous les fichiers qui existent avec le même nom seront écrasés, sans interrogation de sécurité.

-r Cette option doit être consommée avec modération. Elle permet de couper toutes les données qui se trouvent à la fin d'un fichier EXE. Elles peuvent être des données de configuration, mais aussi des données overlay. Vous ne devrez utiliser cette option que si vous savez vraiment ce que vous faites.

-u De manière standard, PKLite conserve l'heure et la date du fichier original. Si vous voulez retenir au contraire l'heure et la date actuelles pour ce fichier, vous devez faire intervenir cette option.

-x Pour décompacter un fichier, entrez cette option. Il se peut que ce soit utile ou même nécessaire, par exemple quand vous cherchez des informations ou des images précises, ou encore lorsqu'un programme ne fonctionne pas correctement quand il est comprimé.

Si une erreur survient au moment de la compression du fichier, PKLite génère l'un des messages d'erreur suivants :

Syntax Error Cette erreur surgit quand vous avez fait une erreur dans la ligne de commande. Il se peut que vous ayez mal écrit le nom du fichier ou que vous ayez oublié le tiret devant une option. PKLite affiche à toutes fins utiles un écran contenant toutes les options.

File extension must be EXE or COM PKLite comprime seulement les fichiers qui ont une extension .EXE ou .COM. Si votre fichier possède une autre extension et si vous êtes sûr qu'il s'agit d'un fichier exécutable, vous pouvez changer cette extension en la remplaçant par .EXE, puis comprimer à nouveau.

Cannot open input file Ce message d'erreur peut avoir plusieurs causes. Il se peut que le disque ou la disquette présente une malformation. Il se peut aussi que vous travailliez en réseau et que le fichier soit justement en cours d'utilisation.

Could not open output file Pour ce message d'erreur également, il y a plusieurs causes à envisager. Il se peut que le disque dur soit défectueux ou qu'il soit bloqué par une autre application. Mais il se peut aussi que vous ayez indiqué un chemin d'accès invalide et que PKLite ne puisse pas générer le fichier pour cette raison.

Write error Cette erreur surgit lorsque le disque ou la disquette sur lesquels vous voulez écrire sont défectueux ou lorsque le fichier est verrouillé par une autre application.

Disk full error Ce message d'erreur surgit lorsque vous n'avez plus assez de place disponible sur le disque. PKLite n'efface l'ancien fichier que lorsque le nouveau a été entièrement généré. Vous devez donc avoir assez de place pour recevoir ces deux fichiers en même temps.

Read error Une erreur est survenue pendant la lecture de l'ancien fichier.

Create error PKLite n'a pas pu créer le fichier indiqué. Cela peut tenir au fait que vous aviez indiqué un répertoire erroné ou au fait que le disque est plein.

Memory error PKLite ne disposait pas d'un espace suffisant en mémoire principale pour réaliser ses opérations. Le programme a besoin d'environ 85 Ko de mémoire libre.

Cannot compress files into itself Ce message surgit lorsque vous indiquez comme second paramètre le même nom que le fichier original. Lorsque vous voulez utiliser le même nom, vous devez simplement ignorer le second paramètre. Sinon, vous êtes obligés d'indiquer un nom différent.

EXE header error Ce message signifie que le header EXE contient trop d'informations pour être comprimé. Dans la plupart des cas, cela permet au programme d'avoir plus de place en mémoire principale.

No extract code error Cette erreur surgit lorsque vous essayez de décompacter un fichier par l'option -x. Cela peut tenir au fait que le fichier n'a pas été compacté avec PKLite. Il se peut aussi que l'identification permettant le décompactage ait disparu.

Data error Cette erreur est déclenchée quand PKLite a trouvé un défaut dans le fichier compacté. Ce dernier est corrompu.

Compressing many files into one file error PKLite ne peut pas comprimer plusieurs fichiers en un seul. Vous avez essayé par exemple d'exécuter PKLite *.exe Hello.exe.

Output wildcards error Vous avez fait intervenir des jokers dans le fichier cible (* ou ?). Ce n'est pas autorisé.

16.3. Game Wizard

Game Wizard est un programme TSR, qui permet de rechercher des éléments en mémoire et de les modifier. C'est une sorte de "trainer" universel. La version en shareware présente certaines restrictions par rapport à la version enregistrée. Dans cette dernière, vous pouvez déposer des tableaux de mémoire (Memory Tables) entiers. De cette façon, il vous suffit de trouver une fois les variables qui interviennent par exemple dans votre jeu favori et vous pouvez ensuite les charger à chaque fois que vous voulez y jouer. Ces tables peuvent aussi être échangées avec vos amis et connaissances, ce qui vous permet d'être rapidement à la tête d'une armada de jeux.

L'appel de Game Wizard

Copiez simplement tous les fichiers du répertoire Game Wizard dans un sous-répertoire de votre disque. Lancez le programme en entrant GWSHARE derrière le prompt. Si l'introduction abrégée de ce livre ne vous suffit pas, vous pouvez aussi appeler le didacticiel de Game Wizard, qui est très bien fait. Il s'appelle GWTUTOR et il se trouve dans le même répertoire. Il vous introduit pas à pas dans la manipulation de Game Wizard.

Lorsque vous avez lancé Game Wizard, il s'installe dans la mémoire en tant que programme résident.

La manipulation de Game Wizard

Dans ce qui suit, nous allons vous familiariser quelque peu avec la manipulation de Game Wizard. Nous allons vous présenter pour cela en détail les différents menus, dont la distribution est conçue de manière logique.

Memory Address Search

Cette option vous permet de parcourir la mémoire utilisée par votre programme à la recherche de variables comme "Argent", "Vie", "Munition". Vous lancez la recherche en entrant simplement la valeur à chercher dans la fenêtre de saisie et en appuyant sur Entrée. Vous avez aussi les combinaisons de touches suivantes à votre disposition :

Touches	Signification
CTRL+E	Dans ce cas, la recherche commence après une nouvelle entrée en mémoire.
CTRL+P	Par l'intermédiaire de cette combinaison, vous pouvez revenir en arrière si vous avez entré une mauvaise valeur lors de la recherche précédente. Vous ne pouvez cependant remonter en arrière que d'une seule étape.

Comment se déroule une recherche de variable ?

Supposons que vous ayez 1000 écus au début d'un jeu. Vous activez Game Wizard et vous choisissez la commande "Memory Address Search". Vous entrez 1000 et vous appuyez sur **Entrée**. Dans le menu principal, vous actionnez **ESC** pour revenir au jeu. Vous continuez à jouer jusqu'à ce que vous ayez à dépenser de l'argent. Si vous n'avez plus par exemple que 765 écus, vous revenez vite au programme Game Wizard et vous entrez la nouvelle valeur 765. Lorsque Game Wizard trouve l'emplacement recherché dans la mémoire, il affiche le résultat de son action, c'est-à-dire l'adresse à laquelle se trouve la variable. Sinon, vous devez continuer votre recherche.

Table of Memory Locations

Une fois que vous avez trouvé la variable recherchée, vous pouvez la "geler" ou la modifier. "Geler" signifie qu'elle ne sera plus modifiée par la suite.

A l'écran vous pouvez voir quatre colonnes pour chaque valeur trouvée. Dans la première colonne, vous indiquerez si vous voulez geler la variable. Dans la seconde, vous donnerez une description pour pouvoir identifier la variable. La troisième colonne contient la position de la variable dans la mémoire et la quatrième contient sa valeur.

Pour traiter les entrées de ce tableau, voici les touches qui sont à votre disposition :

Touche	Signification	
O	Cette touche permet de modifier la valeur de la variable dans le tableau.	
C	Vous devez presser cette touche pour supprimer l'entrée dans le tableau.	
E	Par l'intermédiaire de cette touche, vous passez en mode édition. Plusieurs combinaisons de touches sont alors à votre disposition :	
	TAB	Permet de passer dans la colonne suivante.
	CTRL + S	Sert à entrer l'adresse choisie.
	ENTREE	Sauvegarde les modifications effectuées.
	ESC	Permet de quitter l'éditeur.
	F	Pour indiquer qu'une variable doit être gelée. Une entrée gelée sera affichée en blanc.
	ESC	Permet de revenir au menu principal.

Edit Memory Contents

Cette commande sert à modifier le contenu de secteurs plus ou moins volumineux. C'est particulièrement utile pour les jeux de rôles et les aventures, dans lesquels il est possible de modifier les noms, les descriptions, etc. Voici les touches que vous pouvez utiliser ici :

Touche	Signification	
E	Permet de passer en mode édition. Dans ce mode, il est possible de changer le contenu affiché de la mémoire. Les opérations possibles sont les suivantes :	
	TAB	Fait passer du mode ASCII au mode hexadécimal.
	CTRL + S	Sauvegarde les modifications.
	ESC	Permet de quitter l'éditeur.
G	Cette touche permet d'entrer directement une adresse en mémoire, à laquelle doit sauter l'éditeur.	
H	Autorise la conversion d'un nombre décimal en un nombre hexadécimal et inversement.	
N	Répète la recherche que l'on vient de lancer.	
S	Recherche une combinaison de nombres ou de caractères ASCII. La recherche démarre à la position actuelle en mémoire.	
CTRL + G	Par l'intermédiaire de cette combinaison, vous pouvez sauter directement à un endroit déterminé dans la mémoire. Il se peut que ce soit plus rapide que par l'intermédiaire de "Table of Memory Locations".	
ESC	Vous revenez au menu principal.	

Game Playing Speed

Game Wizard vous offre aussi la possibilité de modifier la vitesse à laquelle se déroule le jeu. Ce peut être une mesure utile pour les jeux d'action et ceux qui ont été écrits pour des 286 relativement lents. L'indice des vitesses sera modifié au moyen des touches de direction.

View Current Program Screen

C'est une fonction très utile. Elle affiche l'écran actuel du programme. Vous en aurez besoin par exemple si vous avez oublié la valeur exacte d'une variable déterminée.

Crash to DOS

Cette fonction met fin au programme exécuté. Ceci peut se révéler utile lorsque le programme s'est encore bloqué et que vous n'avez pas envie de faire redémarrer l'ordinateur.

Nous vous avons ainsi présenté les fonctions essentielles de Game Wizard. Avec ce programme, il est sans doute préférable d'être enregistré, car la version complète est beaucoup plus étendue, à la fois par ses fonctionnalités et les méthodes de recherche qu'elle propose. Si vous n'avez pas envie d'écrire un "trainer" pour un jeu donné, ce programme est sans doute le meilleur choix que vous puissiez faire.

16.4. COMPOSEZ AVEC SCREAM TRACKER 3.01b

Le Scream Tracker 3.01 beta de Sami Tammiletho ("Psi" de Future Crew) a été rendu public début 94 et il appartient aux meilleurs trackers sur le PC. Sa particularité est qu'il est capable d'effectuer lectures et modifications non seulement en format MOD standard, mais aussi dans le format S3M développé par Future Crew. Avec ce format S3M, le Scream Tracker peut utiliser 16 voix numériques simultanément (16 samples en même temps) et 9 voix FM (Adlib).

La description qui va suivre des fonctions du Scream Tracker donne seulement un petit aperçu des nombreuses fonctions proposées. Il vous servira seulement d'introduction. Nous abrégeons la désignation "Scream Tracker 3.01 beta" en ST3 !

L'utilisation du Scream Tracker

Lancer le Scream Tracker

Après la saisie de ST3 derrière le prompt DOS, Scream Tracker démarre avec une image de titre et l'auto-détection de la carte son. Si ST3 ne trouve pas celle-ci, vous devez entrer les paramètres à la main :

-s1 ou -s5	SB
-s2	GUS
-mxxx	Mixing speed
-axxx	Adresse
-ixx	Interrupt
-cx	DMA de la carte son

Pour un fonctionnement sans dommages de ST3, il est également nécessaire d'avoir suffisamment de mémoire libre. Si ce n'est pas le cas, vous devez faire démarrer un gestionnaire de mémoire comme EMM386 dans le CONFIG.SYS.

1. Le menu principal (ESC)

Après avoir lancé le programme, vous vous trouvez dans l'éditeur de patterns. Nous allons y revenir un peu plus loin. Avec la touche **ESC**, vous pouvez faire apparaître le menu principal à partir de là. Dans ce menu, il est possible de charger ou de sauvegarder des fichiers MOD ou S3M, d'appeler le setup pour la configuration (Settings), d'exécuter ou d'éditer des fichiers MOD/S3M ou encore de vérifier l'emplacement libre en mémoire (Status). On peut aussi appeler ces fonctions confortablement par l'intermédiaire des touches de fonction.

2. Le "Song Ordner" (F1)

A gauche, dans le "Song Ordner", on définit l'ordre dans lequel les patterns seront exécutés dans la chanson. Le mot anglais "pattern" désigne une unité regroupant des notes et d'autres informations sur la musique à exécuter. On peut ainsi demander la répétition de patterns en faisant exécuter le pattern 00 à la position 1 et continuer ainsi, avec par exemple à nouveau le pattern 00 en position 10.

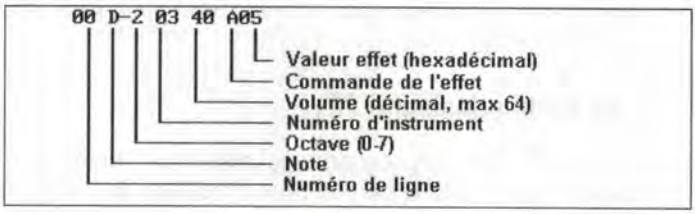
La touche TAB permet alors de passer au milieu de l'écran, où l'on définit la position stéréo des différents canaux. Tout à fait à droite, on peut modifier la vitesse et le volume dans le "Song Setup".

3. Le "Pattern Editor" (F2)

Dans l'éditeur de patterns, on peut créer soi-même un pattern.

Voici comment se présente approximativement un pattern. On peut ici reconnaître cinq sillons dans le sens horizontal (01-05). A côté de ceux-ci, on voit s'il s'agit d'un canal de droite ou de gauche sur la carte son (par exemple L1- gauche), ce qui ne présente bien sûr d'intérêt que pour les cartes son stéréo. Avec la touche TAB, on peut passer d'une piste à l'autre. Il est possible d'atteindre ainsi les pistes qui se trouvent derrière (6-32). Dans le sens vertical, un pattern possède 64 lignes, ce qui veut dire que l'on peut saisir 64 notes par piste.

Une piste se compose à son tour des éléments suivants :



Structure d'une piste MOD

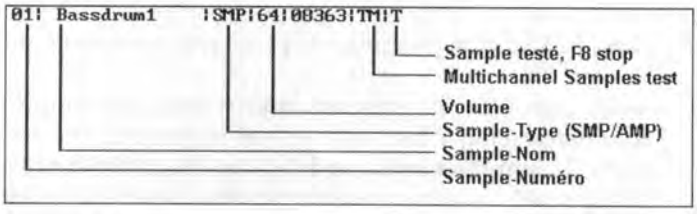
La position du curseur de saisie dans la piste peut être modifée à l'aide des touches de direction. On augmente ou diminue l'octave avec <*> et </> sur le pavé numérique.

Nous allons expliquer un peu plus loin certains effets importants. La page d'aide (**F10**) est également utile, car elle affiche l'ensemble des effets. Le point. sert à effacer une note. Les notes seront saisies par l'intermédiaire du clavier, qui simule un clavier de piano.

D'autres touches importantes de l'éditeur sont représentées par <+> et <->, qui servent à changer de pattern, ainsi que ?, qui sert à changer d'instrument.

4. L'éditeur d'instruments (F3)

On peut modifier ici la liste des instruments en ajoutant de nouveaux instruments, en définissant les noms et les volumes et en exécutant les instruments pour les tester.



Un élément dans la liste des instruments

Dans cette liste, on peut trouver aussi bien des samples numérisés (SMP) que des sons FM de la carte AdLib, qui sont caractérisés par les lettres AME. Il faut noter que les sons AdLib ne peuvent pas être exécutés par la carte Gravis Ultrasound.

On charge un nouveau sample en passant avec le curseur à un sample à remplacer ou à un numéro de sample vide, puis en appuyant sur **ENTREE** ou **F4**. On parvient ainsi directement dans la bibliothèque des samples, dans

laquelle on peut sélectionner un nouveau sample, avant de confirmer avec **ENTREE**.

A l'aide du menu qui se trouve à droite, on peut définir le loop d'un sample, c'est-à-dire indiquer au bout de combien de temps il devra se répéter indéfiniment.

5. La bibliothèque des samples (F4)

Il est utile ici de créer plusieurs répertoires (derrière le prompt DOS, ST3\I), dans lesquels on déposera les différents samples disponibles.

Il est préférable de donner à ces répertoires des noms reconnaissables.

Par exemple : ST3\I\DRUMS pour tous les samples qui reproduisent des percussions.

6. Exécution avec le Scream Tracker

Exécuter une chanson (F5)

Avec **F5**, on passe à l'exécution de la chanson actuelle dans la suite des patterns telle qu'elle a été définie à l'intérieur du "Song-Ordner". Avec les touches **PGUP** et **PGDN**, vous pouvez demander l'affichage des patterns en cours et du spectromètre. L'exécution continuera à se dérouler jusqu'à ce que vous l'arrêtiez explicitement à l'aide de **F8**.

Exécuter le pattern actuel (F6)

Comme pour **F5**, mais on exécute ici le pattern sélectionné à l'aide des touches <+> et <->.

Exécuter le pattern actuel, à la position actuelle (F7)

Comme pour **F6**, à cette différence que l'exécution commence à la position actuelle du curseur.

Song stop (F8)

Permet d'arrêter les chansons, les patterns et les samples soumis à un loop.

7. Afficher la mémoire (F9)

Cette fonction sert à afficher la mémoire encore libre et les informations qui concernent la chanson en cours.

8. Page d'aide (F10)

Cette touche permet d'appeler la fonction d'aide. Celle-ci propose des informations concernant spécialement le domaine dans lequel on se trouve.

Composer un S3M avec le Scream Tracker

Instruments

La première opération pour composer un module consiste à créer des samples. On emprunte pour cela au début des samples dans des fichiers S3M existants (certains sont déjà disponibles avec ST3) ou alors on crée ses propres samples, par exemple avec VOC386 ou DIGIPLAY 3.0, à partir de CD audio ou à partir du clavier.

L'avantage de ST3 est qu'il peut traiter les samples reçus en format Amiga aussi bien qu'en format PC.

Il en va différemment pour la réception des sons AdLib, puisqu'on ne peut pas les transformer en samples. Il faut les recevoir en indiquant différents paramètres. Si l'on ne veut pas s'en occuper dans le détail, le mieux est de reprendre ceux-ci dans le FC Startport Intro, qui se trouve lui aussi dans le répertoire ST3.

Patterns

On peut alors essayer d'entrer sur différentes pistes (par l'intermédiaire de l'éditeur de patterns) des voix ayant trait par exemple à la percussion, à la basse ou à la mélodie. Il faut veiller tout de même à mixer de façon équilibrée les volumes des différentes voix.

Effets

Les nombreux effets donnent parfois le tournis aux débutants qui composent leurs premiers fichiers MOD ou S3M. Le guide en anglais et la page d'aide (**F10**) dans l'éditeur de patterns fournissent toutes les indications voulues sur ces effets. Nous nous contenterons donc ici d'en citer quelques-uns parmi les plus importants, pour préciser les principes qui régissent leur manipulation.

ST3 n'utilise malheureusement pas les commandes standard ProTracker d'Amiga. C'est pourquoi il faut refaire une sorte d'apprentissage, quand on a travaillé auparavant avec d'autres trackers.

Voici quelques exemples pour les effets importants et leur signification :

Axx Vitesse

La valeur xx indique en hexadécimal la vitesse de la chanson. On peut la modifier par exemple dans chaque nouveau pattern. 01 correspond à la plus grande vitesse possible, FF à la plus faible. La valeur par défaut est 06.

D0x Diminuer rapidement le volume (slide down)

La valeur x indique la diminution du volume. Un x faible fait peu diminuer le volume, un x élevé le fait chuter au contraire fortement. Avec cette fonction, on peut obtenir un bel effet de fading, en frappant une note, puis en mettant en œuvre plusieurs commandes successives et en maintenant une faible valeur de x. La diminution du son dépend aussi de la vitesse générale du pattern.

Dx0 Augmenter rapidement le volume (slide up)

Comme ci-dessus, mais pour une augmentation. La valeur limite du volume est de 64.

Gxx note Portamento

La valeur xx indique la vitesse à laquelle la hauteur de la note précédente sera modifiée pour atteindre la note actuelle (fournie par la commande).

Quand on travaille sur les effets, il peut aussi être utile d'écouter beaucoup d'exemples et d'en tirer des leçons. On rencontre beaucoup de méthodes astucieuses de cette façon. Ne vous laissez pas effrayer par la multiplicité des fonctions offertes. Composez vos morceaux en toute quiétude.

16.5. Assembler A86

Pour l'installation, vous devez copier A86V370.ARJ dans un répertoire donné et décompacter ensuite le programme.

Cet assembleur en shareware est un choix possible, qui peut concurrencer les assembleurs commerciaux largement répandus que sont MASM et TASM. Les caractéristiques remarquables de ce produit sont les structures simples et proches de la machine, pouvant simplifier le travail pour les petits projets. Les textes source simplifiés ont pour effet sans aucun doute une certaine incompatibilité, dont la conséquence est que beaucoup de programmes TASM doivent être réécrits avant d'être assemblés avec A86.

Certaines instructions importantes de TASM/MASM (comme ASSUME) sont ainsi ignorées, de sorte qu'on est obligé de s'occuper soi-même des "segments override" (mov ax,CS:var).

L'A86 offre aussi quelques améliorations, permettant de simplifier certaines séquences qui reviennent souvent :

- On peut placer dans une commande unique le PUSH de plusieurs registres :

```
PUSH ax, bx, cx au lieu de PUSH ax; PUSH bx; PUSH cx;
```

- Les commandes INC et DEC possèdent un opérande supplémentaire, qui indique le terme à additionner (INC ax,2 = ax pour augmenter de deux unités).

- Les distinctions de cas ont été simplifiées. Au lieu de :

```
jnz label
mov ax, bx
label:
```

vous pouvez écrire simplement :

```
if z mov ax, bx
```

Vous trouverez certaines autres améliorations de moindre importance dans la documentation, au chapitre 5 (A05.DOC).

Bien entendu, le processeur ne peut pas traiter directement ces commandes. Elles sont transformées dans leur version habituelle pendant l'assemblage.

Une amélioration très pratique concerne le traitement des erreurs. Les messages d'erreur ne sont pas affichés à l'écran, mais insérés directement dans le texte source. Au prochain assemblage, ils sont supprimés automatiquement. Cette particularité a toutefois perdu depuis de son importance, car les assembleurs modernes sont intégrés dans les interfaces de Pascal et de C. Les messages d'erreur sont affichés plus confortablement dans la fenêtre des messages.

On appelle A86 à l'aide de :

```
A86 Fichier_source [to] [fichier_binaire] [fichier symboles]
[options]
```

Le fichier source est celui qui contient le texte source. Le "to" qui vient ensuite est optionnel. Il indique que le résultat de l'assemblage doit parvenir au fichier binaire. A l'aide de l'extension, on décide ici s'il s'agit d'un fichier COM, d'un fichier objet destiné à être lié à d'autres fichiers (.OBJ) ou d'un fichier binaire (.BIN) pour les EPROM.

Le fichier des symboles contient une liste des symboles utilisés dans le programme. Ce fichier doit porter l'extension .SYM. Si l'on indique plusieurs fichiers source, leurs noms seront classés par ordre alphabétique et ils seront assemblés dans cet ordre, comme s'il s'agissait d'un seul grand fichier.

On peut encore indiquer quelques options, qui essaient entre autres d'augmenter le degré de compatibilité avec TASM/MASM :

-+D:

L'A86 fait normalement la différence entre les bases de numération à l'aide du premier chiffre. Si celui-ci est un 0, il s'agit d'un nombre en hexadécimal. Sinon, c'est un nombre en système décimal. Grâce à l'option +D, les nombres seront tous considérés comme des nombres décimaux (comme c'est le cas dans TASM/MASM), sauf s'ils sont déclarés explicitement hexadécimaux par la lettre h qu'on leur ajoute.

-+G15

Cette option permet de supprimer certaines particularités du code généré, de façon à ce que le code objet soit compatible avec TASM/MASM. Vous trouverez tous les détails voulus dans la documentation (A03.DOC).

-+O:

Equivaut au fait d'ajouter un fichier OBJ à la ligne de commande. On génère un fichier objet au lieu d'un fichier COM.

-+f:

Correspond au commutateur **/e** de TASM et active l'émulation du coprocesseur (seulement dans le cas de l'intégration dans un langage évolué).

La portabilité de programmes tout prêts d'une certaine longueur sur A86 demande un peu de travail, mais elle est possible. Il faut d'abord parvenir à une compatibilité suffisante avec les options +G15 et +D. Il faut pourtant réaliser certaines modifications dans le texte source.

L'A86 ne dispose pas de la puissante directive ASSUME. Il semble que les auteurs l'aient exclu volontairement, comme on peut le lire dans leur présentation : "The ASSUME mechanism creates far, far more confusion than it solves. So I scrapped it..." (cf. le fichier A06A.DOC). De ce fait, il faut réaliser à la main, avec le segment-override voulu, tous les accès aux segments qui ne

correspondent pas au segment standard de la commande. Si la variable "Var" se trouve ainsi dans le segment de code, il faut écrire pour la lecture mov ax,CS:Var au lieu d'écrire simplement : mov ax, Var.

Dans l'A86, les labels de la forme L14 (une lettre suivie par plusieurs chiffres) sont considérés comme des labels locaux, qui peuvent apparaître plusieurs fois dans le texte source. L'inconvénient de cette disposition est qu'il faut décider clairement quel est le label concerné par un saut. Pour cela A86 utilise le caractère , qui indique un saut vers l'avant. Ainsi jmp L2 saute au label L2 suivant, tandis que jmp L2 saute au L2 précédent. Si vous avez utilisé des labels de ce type dans le texte source TASM/MASM, vous devez insérer des caractères plus élevés pour les sauts vers l'avant.

Les macros doivent être soumises à des modifications assez considérables. A86 n'utilise pas de noms pour les paramètres. Il utilise à leur place les signes #1 à #9. La fin de la macro est indiquée par #em.

Vous trouverez dans A12.DOC d'autres synonymes pour les commandes macro. Ce document contient aussi un guide pour le portage des textes source dans les deux directions.

16.6. Déboqueur D86

Pour l'installation, copiez D86V370.ARJ dans un répertoire et décompactez le fichier.

Ce débogueur sert de complément à l'assembleur A86 du même auteur. Il permet d'afficher les fichiers exécutables, comme le Turbo-Debugger de Borland, puis de les exécuter pas à pas. A la différence de ce qui se passe avec la première génération de débogueurs (DEBUG.COM de MS-DOS), les informations essentielles restent affichées à l'écran. Cela concerne le code programme en mémoire, les registres et les flags, ainsi que les secteurs de mémoire sélectionnés.

Le D86 est donc très proche du A86. Celui-ci est même conservé simultanément en mémoire. Il est ainsi possible non seulement d'insérer de nouvelles commandes en assembleur dans le code existant, mais aussi d'envoyer directement des instructions au processeur, comme c'est d'ailleurs aussi le cas avec les interpréteurs en Basic. Ce débogueur ne donne pas la possibilité par exemple de modifier le contenu d'un registre. On le fait en exécutant la commande Mov correspondante.

Le débogage symbolique (l'utilisation de labels et de désignations au lieu des adresses à vrai dire peu parlantes) est possible quand on a affaire à un fichier

SYM, dans laquel se trouve la liste des symboles utilisés à l'intérieur du programme. Ce fichier SYM est créé lors de l'assemblage du texte source par A86 ou par conversion des fichiers MAP provenant d'autres assembleurs. La dernière possibilité n'est toutefois ouverte qu'aux utilisateurs enregistrés.

On lance D86 avec :

```
D86 [options] [nom [paramètres]]
```

Les paramètres entre crochets peuvent être laissés de côté.

Le programme indiqué par "nom" est chargé au départ et pourvu des paramètres signalés. Un appel de D86 XCOPY A: vous permet donc d'examiner comment la commande XCopy copie des fichiers à partir du lecteur A:.

Il n'existe que deux options, l'une d'elles étant aujourd'hui d'ailleurs superflue, puisqu'elle permet de supporter des ordinateurs qui ne sont pas compatibles IBM. La seconde option, soit +v, active la possibilité d'utiliser deux moniteurs (un moniteur couleur et un monochrome) lors du débogage.

Après le lancement du programme, on voit en haut à gauche la fenêtre de désassemblage. Le code programme est affiché ici à la position actuelle. En bas à gauche, on voit une liste des registres et des flags. Ces derniers sont affichés quand ils sont posés et effacés quand ils sont sur 0. Les désignations correspondent à la norme. Seul le flag de parité (p) est symbolisé ici par un "e".

En haut à droite, on a une fenêtre destinée à plusieurs usages distincts. Elle sert à afficher des textes d'aide, des secteurs de mémoire et des messages d'état. En bas à droite, on peut voir les secteurs de mémoire sur six lignes. La ligne inférieure est initialisée à 0 au départ du programme. Elle montre l'état de la pile. On y trouve d'abord le nombre de mots contenus dans la pile, puis un double-point, enfin le contenu proprement dit.

En actionnant la combinaison de touches **ALT + F10**, on fait passer la fenêtre du mode Aide au mode Etat. A l'intérieur de ces modes, le choix de l'information à afficher se fait au moyen de la touche **F10**. En mode Aide, on a la signification des touches de fonction, des touches simples et des combinaisons de touches avec **CTRL**. En mode Etat, on peut obtenir le message de copyright, les informations concernant l'état, un extrait de la mémoire et enfin les informations sur le coprocesseur.

Les informations concernant l'état peuvent être appelées directement au moyen des combinaisons de touches suivantes :

CTRL+F	Coprocesseur
CTRL+N	Page suivante de la mémoire
CTRL+P	Page précédente de la mémoire
CTRL+S	Informations concernant l'état
CTRL+T	Supprime la pile
CTRL+D	Octet suivant dans le code programme (contrairement à D, qui fait passer à la commande suivante)
CTRL+U	Octet précédent dans le code programme
CTRL+E	Saut à la fin du programme
HOME	Saut à la dernière commande exécutée, puis au début du programme

L'extrait de mémoire dans la fenêtre en haut à droite représente toujours une extension du dernier bloc de mémoire sélectionné dans les lignes (1-6) en bas à droite de l'écran.

Les touches de fonction permettent d'influer sur l'exécution du programme à examiner :

F1	Pas à pas dans les fonctions, mais non dans les interruptions
CTRL+X	Pas à pas aussi dans les appels d'interruption
F2	Pas à pas avec les appels de procédure
F3	Répéter la dernière instruction
F6	Exécuter jusqu'au prochain RET
F7	Saisir/assembler de nouvelles commandes
F9	Exécute le programme de la dernière position à la position actuelle
MAJ+F7	Sélectionne la position pour une recherche ultérieure (cf. touche F)
ALT+F9	Elabore à nouveau l'écran

L'exécution directe de commandes en assembleur se fait au moyen d'une simple saisie de celles-ci. Si l'on appuie par exemple sur la touche M, la fenêtre d'aide (à condition qu'elle soit activée) annonce que l'on doit continuer la saisie de la commande. Il faut donc compléter la commande par OV AX, BX.

Le débogueur est contrôlé par des séquences de commandes comme pour les instructions données directement en assembleur. Les fonctions utilisées le plus couramment sont alors :

B, Adresse [,Adresse]

Breakpoint : définit l'un des breakpoints disponibles ou les deux à la fois, pour interrompre à cet endroit l'exécution du programme.

F, Longueur [,Offset]

Find : Cherche la section du code marquée à l'aide de **MAJ+F7** à partir de la position actuelle. Compare un nombre d'octets donné par "longueur". Quand la recherche aboutit, le code programme est déplacé vers le bas de "offset" octets.

G [,Adresse] [,Adresse]

Go : Exécute le programme et pose au plus deux breakpoints provisoires, qui seront supprimés après l'exécution.

J, Adresse

Jump : Saute à l'adresse indiquée.

L [,Adresse] [,nom_fichier]

List : Sort (dans un fichier) ou imprime le listing depuis la position actuelle jusqu'à l'adresse indiquée. Lorsqu'aucune adresse n'est indiquée, le programme tout entier est imprimé.

O, limite inférieure [,limite supérieure]

Arrête le programme quand on appelle la fonction DOS (Int 21h) numéro "limite inférieure". Si l'on indique également une limite supérieure, toutes les fonctions qui se situent entre les deux limites sont surveillées.

Q

Quit : Permet de sortir du débogueur.

W

Write : Ecrit le programme sur le disque dur (seulement pour les fichiers COM) et recrée une table des symboles (*.SYM).

Il ne reste plus que les lignes contenant les "dumps" de mémoire, en bas à droite. On les sélectionne en appuyant sur la touche portant le numéro correspondant à la ligne voulue. On peut alors définir le secteur à afficher et le mode d'affichage.

La syntaxe qui intervient ici est la suivante :

Type, Segment, Offset

Le segment et l'offset indiquent la position de l'extrait de mémoire. Il est également possible d'utiliser des registres. Quant au type, il y a beaucoup de possibilités prévues pour le contrôle. Nous ne présentons ici que les types les plus importants :

Type	Signification
A	ASCII, chaînes avec caractères de contrôle
B	Octets hexadécimaux
D	Words décimaux
F	Virgule flottante (coprocesseur)
N	Octets décimaux
S	Chaînes ASCIIZ (correspondent aux chaînes en C)
W	Words hexadécimaux

16.7. TOS-Copy - un programme de copie très pratique

Pour l'installation du programme, copiez TOS_COPY.EXE dans un répertoire et appelez-le. Le décompactage se fait automatiquement.

Tos-Copy est un programme de copie pour les disquettes possèdant quelques particularités qui le distinguent de DISKCOPY de MS-DOS : TOS-Copy peut lire une disquette entière d'un seul coup, en se servant de la mémoire XMS et du disque dur. En outre, on peut accéder plusieurs fois en lecture aux disquettes qui ont été lues. TOS-Copy est appelé par l'intermédiaire de la commande

```
TC [Source][Cible]
```

L'indication des lecteurs source et cible est optionnelle. On peut aussi les sélectionner dans le programme. Après le lancement du programme, on voit en haut à droite un message d'état, qui donne la liste des lecteurs existants, la mémoire XMS disponible et aussi la durée approximative d'une copie, quand celle-ci est en cours d'exécution.

En haut à gauche, on trouve une fenêtre à plusieurs usages, qui affiche des éléments différents selon la situation. Elle contient en outre le menu. En dessous, on peut voir l'indication concernant le processus de copie en cours avec la ligne de texte et l'affichage du track.

L'affichage multiple possède trois niveaux, entre lesquels on passe avec les boutons OK (niveau supérieur) et SETUP (niveau inférieur). Le niveau le plus bas contient quelques options Sound. Le niveau moyen sert à indiquer les lecteurs source et cible (la flèche clignotante indique la combinaison présente).

On peut aussi réaliser des copies entre des types de lecteur différents (3"1/2 vers 5"1/4 et inversement). Toutes les combinaisons sont possibles, mais toutes ne sont pas utiles. Une copie d'un lecteur 1,44 Mo (3"1/2) vers un lecteur 1,2 Mo (5"1/4) ne donnera que rarement un résultat correct.

Le niveau supérieur affiche les lecteurs source et cible agrandis et rend disponibles les commandes READ, WRITE et COPY. READ permet de charger une disquette du lecteur source dans le tampon. WRITE écrit le contenu du tampon sur une disquette, ce qui peut se fait plusieurs fois de suite. On génère ainsi autant de copies que l'on veut.

COPY réunit les deux processus précédents. Après lecteur de la disquette source, le programme demande à l'utilisateur d'introduire la disquette cible, celle sur laquelle la copie va se faire. Après appel de COPY, le contenu de la disquette se trouve toujours dans le tampon et on peut l'écrire sur d'autres disquettes avec WRITE. La dernière commande est QUIT. Elle permet de quitter le programme.

16.8. VPIC 6.1 - Visualiseur universel de fichiers graphiques

Pour installer ce logiciel, copiez simplement VPIC61.EXE dans un répertoire et appelez-le. Le programme se décompacte automatiquement.

VPIC est un classique parmi les programmes de visualisation et de conversion des graphiques. Il se distingue en particulier par son grand choix de modes graphiques, que l'on peut définir individuellement pour chaque jeu de puces graphiques.

Le responsable de cette adaptation est le programme CONFIG.EXE, à l'aide duquel on peut sélectionner un gestionnaire approprié à partir d'une liste assez fournie. Si vous voyez toutefois qu'il n'y a pas de gestionnaire qui fonctionne avec votre carte graphique (ce qui est tout de même hautement

improblable), vous pouvez aussi créer un gestionnaire (toutes les directives sont données dans le fichier CONFIG.DOC). L'appel de VPIC se fait au moyen de :

```
VPIC [nom_image][options]
```

Quand il n'y a pas de nom d'image mentionné, on peut sélectionner ce nom dans un menu. Les options sont très nombreuses, mais elles servent surtout à automatiser les processus pour générer des présentations. Elles ne nous intéressent donc pas ici.

Dans le menu, on peut sélectionner des fichiers en appuyant sur la touche d'espacement, puis les afficher avec **ENTREE**. Les images affichées peuvent ensuite être converties en d'autres formats à l'aide de touches particulières. La correspondance entre les touches et les formats est la suivante :

Touche	Format/Fonction
C	Dr. Halo (.CUT)
D	Deluxe Paint (.LBM)
G	GIF
P	PIC
R	ColorRIX (.RIX)
S	TIF
T	TGA
W	Windows-BMP
Z	PCX
A	Lancement d'une rotation des palettes (pour les fractales)
B	Génère une image en noir et blanc
F1-F10	Correction des couleurs de base
ALT-F10	Recrée la palette initiale
/ ou **?**	Aide, affichage des touches disponibles

Dans le menu de sélection, en dehors de la touche **ENTREE** et de la touche d'espacement, voici les possibilités :

Touche	Signification
F1	Aide
F2	Lance une présentation automatique de diapositives ; les images sélectionnées sont affichées l'une après l'autre. On règle l'intervalle entre deux images avec **ALT+D**.

Touche	Signification
F3	Informations sur le fichier sélectionné
F4	Passage de 16 à 256 couleurs et inversement
F5/F6	Diminuer/augmenter la résolution
F7	Fixer la résolution (LOCK, automatisme désactivé)
ALT+F7	Fixer la résolution couleur HiColor
F9/ALT+F9	Définir le chemin d'accès
ALT+C	Copier les fichiers choisis
ALT+M	Déplacer les fichiers choisis
ALT+X	Supprimer les fichiers choisis

16.9. Convertir des graphiques : Image Alchemy 1.7

Image Alchemy est un convertisseur graphique performant pour DOS. En dehors de la conversion de nombreux formats, il permet de modifier légèrement les dimensions, les palettes et d'autres données.

Vous appellerez Alchemy à l'aide de :

```
ALCHEMY [options]fichier_source [fichier_cible]
```

Si l'on n'indique pas de format sortie indiquée, il faut de toute façon l'un des paramètres suivants, afin d'obtenir la sortie à l'écran. Toutes les options V peuvent être suivies de la résolution horizontale.

-v	8 bits (256 couleurs)
-V	8 bits, image adaptée à la résolution
--v	Comme -v, mais avec TrueColor
--V	Comme -V, mais avec TrueColor

Les options générales les plus importantes sont :

-x	Afficher les informations sur l'image
-h	Afficher l'aide
-o	Remplacer le fichier

La taille de l'image peut être manipulée à l'aide des options suivantes :

-X[t]n	Mettre à l'échelle sur n pixels dans la direction x. Utiliser la méthode t.
-Y[t]n	Comme -X, mais dans la direction y.
-Dn m	Indiquer la résolution en dpi (n*m)

Les couleurs pourront être modifiées comme suit :

-n	Niveaux de couleurs en bits/pixel (par exemple -16 pour 16 bits/pixel)
-b	Image en noir et blanc
-cn	Définir le nombre de couleurs
-dn	Indique la méthode Dither à utiliser (0-15, 20-22)
-f	Charger la palette à partir d'une autre image, en pratique quand plusieurs images présentent la même palette.
-N	Inverser l'image

Tous les formats de sortie courants sont disponibles, même des formats peu commodes comme PDS, utilisé par la NASA pour les images de satellites sur CD. Vous obtiendrez une liste de tous les formats disponibles en appelant ALCHEMY -h2 et h3.

16.10. Afficher et convertir des graphiques sous MS-Windows avec PaintShop Pro

Vous installerez le programme en copiant PSPPRO.EXE dans un répertoire et en l'appelant. Le décompactage se fait automatiquement.

Sous Windows également, il existe un grand nombre de convertisseurs graphiques. Un exemple remarquable en est le Picture Shop Pro. Celui-ci peut non seulement vous permettre de visualiser et de convertir des images, il peut

aussi vous offrir des services inestimables pour le traitement des images. Il dispose de larges possibilités pour la correction des couleurs et le filtrage.

Les opérations de contrôle du logiciel (malheureusement en langue anglaise) sont évidemment très intuitives, de sorte que l'apprentissage en est facilité d'autant.

Dans le menu **File**, vous avez les fonctions standard comme **Open**, **Close** ou "Print" et une option très intéressante : **Batch Conversion**. Vous pouvez ici envoyer des instructions massives au convertisseur. Dans la moitié gauche de la boîte de dialogue, vous sélectionnez le chemin d'accès de la source et vous choisissez les fichiers à convertir (en appuyant simultanément sur la touche MAJ, vous pouvez choisir plusieurs fichiers en même temps). Dans la moitié droite, vous devez indiquer le chemin cible et le format de sortie. Après quoi PSP convertit tous ces fichiers dans le format souhaité.

Le menu **View** permet de fixer le facteur de zoom et active les outils, ainsi que l'histogramme. La fenêtre des outils contient les icônes destinées au zoom (loupe : bouton gauche de la souris pour agrandir, bouton droit pour réduire), au déplacement du contenu de l'image (main), à la sélection d'un domaine (rectangle en pointillés), à l'édition de la sélection (flèches) et à la copie du secteur dans une nouvelle position (deux rectangles). L'histogramme représente une vue d'ensemble sur les différentes intensités de l'image. Sur l'axe des x, les intensités sont reportées dans le sens croissant. Sur l'axe des y, ce sont les valeurs en pourcentages qui sont reportées.

Dans le menu **Image**, vous pouvez faire basculer l'image dans le sens vertical (Flip) et horizontal (Mirror). Vous pouvez aussi lui faire subir une rotation. **Resample** et **Resize** offrent la possibilité de doter l'image de nouvelles dimensions et d'une nouvelle échelle. De plus, **Resample** opère un lissage à l'aide d'un effet **anti-aliasing**.

En outre, vous disposez d'un grand nombre de filtres. On utilise les filtres **Edge** pour rechercher et renforcer les contours, le filtre **Normal** pour dessiner de manière floue (Blur/Soften) ou précise (Sharpen). Les filtres regroupés sous le titre **Special** sont intéressants par les effets qu'ils permettent de produire. Nous mentionnerons avant tout **Despeckle** et **Emboss**. Le premier permet de supprimer la **neige** qui apparaît en particulier quand on numérise des images par scanner. Il améliore ainsi énormément la qualité de ce type d'images. **Emboss** génère un effet de relief, qui laisse une impression de spatialité.

Sous **Colors**, on peut manipuler les couleurs de l'image. **Brightness/Contrast** permet de régler l'intensité et le contraste de l'image dans sa totalité. **Grey Scale** génère une image en niveaux de gris et **Negative Image** permet d'obtenir le négatif de l'image. A l'aide de **Red/Green/Blue**, on peut régler les différentes composantes à part. La palette peut être modifiée sous **Edit Palette** (à condition que l'image possède une palette ; les images HiColor et TrueColor n'entrent

pas dans cette catégorie). Une information très intéressante est fournie par la commande **Count Colors Used**, qui compte le nombre de couleurs effectivement utilisées dans l'image et permet ainsi d'évaluer la perte de qualité. Vous serez étonné de voir le petit nombre de couleurs utilisées même par des images TrueColor (le plus souvent moins de mille, alors qu'on peut en faire intervenir 16,7 millions).

A l'aide des commande **Decrease Color Depth** et **Increase Color Depth**, on convertit des images d'un système de couleurs à un autre, par exemple de TrueColor à 256 couleurs.

Une fonction souvent adoptée par les programmes externes est ici intégrée sous le menu **Capture**. Celui-ci permet de créer des photos d'écran (screenshots) des écrans Windows entiers ou partiels. Lorsqu'on a choisi l'une des commandes de ce menu, on déclenche le processus de **capture** en cliquant avec le bouton droit de la souris. Les différentes possibilités qui s'offrent alors sont présentées dans le tableau suivant :

Area	Permet, après activation de la capture, de choisir le secteur à copier.
Full Screen	Sauvegarde l'ensemble de l'écran.
Client Area	Sauvegarde le contenu de la fenêtre actuelle.
Window	Sauvegarde la fenêtre avec son cadre et sa barre de titre.

16.11. Graphic Workshop for Windows - convertir et afficher des graphiques

Pour installer ce programme, vous devez copier le fichier GWSWN11I.ZIP dans un répertoire et le décompacter. Exécutez ensuite à partir de Windows SETUP.EXE et indiquez le chemin où vous désirez installer l'application.

Voici un autre représentant de la classe des convertisseurs graphiques pour Windows. Si vous connaissez la version précédente de ce logiciel, vous remarquerez en premier lieu les "thumbnails" au démarrage du programme. Il s'agit de petits graphiques de visualisation créés pour chacun des graphiques disponibles. Lorsqu'un graphique n'a pas encore été enregistré, vous devez d'abord l'ajouter à l'aide de la commande **Thumbnails/Add Thumbnails**. Si vous préférez afficher plusieurs fichiers simultanément plutôt que de grandes icônes, vous pouvez désactiver cette fonction par l'intermédiaire de **Use Thumbnails**.

En cliquant sur l'un de ces "onglets" ou sur les noms de fichier, vous sélectionnez les fichiers voulus pour les modifier (vous pouvez sélectionner plusieurs fichiers en même temps). Vous pouvez ensuite soumettre les graphiques choisis à certaines procédures à l'aide de boutons ou de commandes de même nom.

Convert sert à convertir les fichiers choisis dans un format cible, pris dans la sélection offerte.

Crop sert à afficher le graphique complet, de manière à le faire entrer dans l'écran. Avec la souris, vous pouvez sélectionner un secteur qui sera sauvegardé avec **Save. Next** permet d'aller à l'image suivante lorsque l'on a plusieurs images sélectionnées. **Get Info** fournit les informations disponibles sur cette image.

View est très proche de **Crop**. Ici, l'image est représentée à l'échelle 1:1 et on peut l'agrandir avec **Zoom. Save** permet d'enregistrer le graphique au complet, après l'avoir éventuellement modifié au moyen de **Adjust**, en ce qui concerne les couleurs de base, le contraste ou l'intensité.

Dither sert à réduire l'image choisie au noir et blanc, donc à un seul niveau de couleurs (1 bit). On peut ici choisir entre plusieurs procédés Dither et entre différents facteurs d'agrandissement.

Effects contient les effets ***Color Reduction*** (diminuer le nombre de couleurs), ***Greyscale*** (compter les niveaux de gris), ***Sharpen*** et ***Soften*** (dessiner flou ou précis), ***Smudge*** (selon la fonction d'aide, il s'agit d'un effet "comme si on voyait l'image à travers une goutte d'eau"), ***Spatial Posterization*** (effet mosaïque) et ***Promote to 24 bits*** (changer le nombre de couleurs originales).

Transform contient toutes les possibilités qui permettent de faire basculer ou tourner l'image.

16.12. Calculer des graphes fantastiques de fractales avec FractInt 18.2

Pour installer FractInt, vous devez copier FRACT182.EXE dans un répertoire et le décompacter.

Le logiciel dispose d'une fonction d'aide détaillée. Si vous avez besoin malgré tout d'une documentation, un appel de FRACTINT MAKEDOC génère un fichier texte portant le nom de FRACTINT.DOC.

FracInt est un générateur de fractales convaincant par le grand choix de fractales proposées, par la rapidité d'affichage et par les nombreuses fonctions disponibles. FracInt doit sa grande vitesse d'exécution avant tout à l'arithmétique des entiers, utilisée ici en lieu et place du calcul en virgule flottante, qui demande toujours beaucoup plus de temps. Pourtant, ce dernier est également possible pour les utilisateurs qui disposent d'un coprocesseur.

Dans le menu principal, toutes les touches qui interviennent sont expliquées. Elles sont d'ailleurs disponibles partout dans le programme et pas seulement ici. On parvient au menu principal à partir de n'importe quel point du programme en appuyant simplement sur **ESC**.

Select Video Mode ou la touche **DEL** permet de choisir le mode graphique à utiliser. Celui-ci sera activé au moyen des touches de direction ou grâce aux touches de fonction et de leurs combinaisons (celles-ci sont également disponibles dans tout le programme). Après le choix du mode, vous faites démarrer tout de suite le calcul de la fractale choisie.

Select Fractal Type ou la touche **T** permet d'obtenir des informations sur l'image dont le calcul est en cours, à supposer que ce soit le cas.

Une image en cours de calcul peut être manipulée de multiples façons. En cliquant avec le bouton gauche de la souris, on parvient au mode Zoom. Les mouvements de la souris permettent de déplacer le secteur à zoomer. Quand on maintient le bouton de la souris enfoncé, on peut agrandir ou réduire le secteur en question. En gardant enfoncé le bouton droit de la souris, on fait subir une rotation à l'image. Une double clic avec le bouton de gauche provoque un zoom sur le secteur. Un double clic à droite supprime le zoom.

Ce zoom n'a rien à voir avec le simple zoom sur les pixels proposé par le convertisseur d'images. Dans FracInt, l'image sur laquelle on zoome est entièrement recalculée, avec utilisation de la résolution complète.

En appuyant sur **C**, vous activez le mode **Color Cycling** (avec **ESC**, vous le quittez à nouveau). Dans ce cas, les touches **<+>** et **<->** provoquent une rotation des couleurs dans une certaine direction, ce qui produit un effet très intéressant. Une palette entièrement neuve est chargée avec L. Cette fonction trouve dans ce livre son application avec le "Voxel-Spacing", où un nuage de plasma a été créé avec la palette "landscap.map".

Save Image to File ou **S** permet d'enregistrer (en dehors du mode Color-Cycling !) une image sur le disque dur (avec le nom FRACT001.GIF, etc). Cette image peut ensuite être utilisée par les puissants outils 3D de FracInt :

3-D Transform from File ou **3** génère une projection en trois dimensions d'un fichier GIF quelconque (par exemple un nuage de plasma). Après indication du fichier source et du mode vidéo à utiliser, on dispose de nombreuses

options pour la mise en forme. L'écran suivant demande quelle doit être la méthode de présentation. Vous devez savoir que la qualité (et donc le temps de calcul) augmente au fur et à mesure que vous descendez dans les options proposées. Après chaque option, on a plusieurs écrans de sous-options à confirmer avec la touche **ENTREE**.

16.13. Calculer des graphiques fantastiques avec Povray 2.0

Pour installer ce logiciel, il vous suffit de copier le programme POVIBM.EXE dans un répertoire et de l'appeler. Le décompactage se fait automatiquement.

Povray est un "raytracer" orienté texte disponible en freeware. Il possède des capacités extraordinaires. On peut ainsi reproduire des scènes complexes, comme des espaces entiers ou des paysages fantastiques de manière très fidèle, en se servant des éléments proposés par le langage Povray. On peut conférer à chaque corps une surface spéciale, définie dans ses diverses particularités (motif, couleur, réflexion, etc.) Bien entendu, on dispose de multiples textures prédéfinies, qui représentent l'eau, le ciel, le marbre.

Provray lit les scènes dans un fichier texte. Cela veut dire que le monde à représenter doit être saisi tout entier avec toutes les coordonnées et sa définition précise dans un éditeur, comme avec un langage de programmation. Il sera ensuite "compilé" par Povray.

Nous ne pouvons évidemment pas vous détailler ici la totalité des fonctions disponibles (POVRAY.DOC dépasse les 100 pages). Pourtant, à l'aide du programme Moray, on peut éditer des scènes de manière plus confortable.

Nous mentionnerons un petit exemple, pour montrer quelle est la structure du langage (proche du langage C) :

```
#include "colors.inc"
#include "textures.inc"
camera { location 10,0,0
         look_at 0,0,0 }
sphere { 0,0,0 , 3
         texture { Red_Marble } }
light_source { 10,10,-10 color White }
```

Les deux commandes Include contiennent toutes prêtes les définitions de couleurs et de textures dont on aura besoin par la suite. L'instruction

"camera" définit la position de celui qui regarde ("location") et le point vers lequel son regard est dirigé ("look_at"). Il existe beaucoup d'autres commandes dans le cadre de "camera", comme pour les autres objets de cette démontration. Ces commandes sont explicitées en détail dans la documentation. Les coordonnées ou les vecteurs sont encadrés dans Povray par des caractères majuscules/minuscules et séparés par des virgules.

On définit un premier corps avec "sphere". Dans ce cas, le centre de la sphère est à l'origine des coordonnées et le rayon fait 3 unités. La texture demandée est une texture prédéfinie (Red_Marble, que l'on trouve dans TEXTURE.INC).

Pour finir, il faut encore une source de lumière. Dans ce cas, elle se trouve au point de coordonnées 10,10,-10 et elle diffuse une lumière blanche.

Cette scène (appelée DEMO.POV) est rendue par :

```
POVRAY +IDEMO.POV +X +D +LC:\POVRAY\DOCS
```

Vous pouvez transmettre à Povray entre autres les options suivantes :

+I	Définit le fichier source.
+X	Active l'interrogation du clavier (interruption quand on appuie sur une touche déterminée).
+D	Active la présentation VGA de l'écran. Si vous souhaitez un affichage TruColor, ajoutez l'option +DOT, au lieu de +D.
+V	Fournit des informations sur l'écran de texte.
+L	Définit le chemin vers les fichiers Include.
+O	Définit le fichier de sortie (par défaut: DATA.TGA).
+Qn	Définit la qualité (n= 0 - 9).
+W	Indique la largeur de l'image.
+H	Indique la hauteur de l'image.
+C	Poursuit un calcul qui a été interrompu.

16.14. UNE MAGNIFIQUE INTERFACE UTILISATEUR POUR POVRAY - MORAY 1.3

Pour installer ce logiciel, vous devez copier MORAY13.ZIP dans un répertoire et le décompacter à l'aide de PKUNZIP.

Puisque ce programme ne dispose pas d'une routine d'installation, vous devez porter à la main les chemins d'accès à Povray dans le fichier MORAYPOV.CFG. Ecrivez pour cela sous l'option [config] (plutôt vers la fin) le chemin voulu dans la chaîne placée derrière PrintPath10 et PrintPath20. Le passage se présentera comme suit :

```
[config]
PrintPath10  'C:POVRAY\'
PrintPath20  'C:POVRAY\'
```

Moray est une interface graphique conçue pour développer des scènes Povray. Avec la souris, vous pouvez créer, transformer des objets et leur conférer des textures. La sortie dans un fichier texte, exigée par Povray, est prise en charge par Moray.

Après le lancement à l'aide de :

```
Moray [Nom de la scène]
```

on voit apparaître à l'écran trois vues de la scène, de côté, de devant et d'en haut. En bas à droite, vous pouvez visualiser la façon dont la caméra voit la scène et la façon dont Povray la rend. Les trois fenêtres 2-D peuvent être zoomées. Vous devez conserver pour cela la touche **ALT** enfoncée et cliquer avec le bouton gauche de la souris. Vous déplacerez les fenêtres en appuyant sur **CTRL**.

Dans le menu qui se trouve sur le bord droit, vous avez les actions suivantes à votre disposition :

Create génère un nouvel objet : **Cube, Sphere, Cylinder, Cone** ou **Torus** sont ce qu'on appelle les formes primitives. Vous avez en outre les **sweeps**, qui génèrent un corps en trois dimensions à partir d'une surface déterminée. **Rotational** fait tourner une droite autour de l'axe des z. **Translational** la déplace parallèlement à cet axe. **Conical** produit un cone à partir de cette droite. On peut aussi rassembler des groupes d'objets et les transformer en un objet unique (CSG, il est possible d'obtenir l'addition, la soustraction et l'intersection de deux objets).

Pointlight génère une source de lumière, de même que **Spotlight.** Cette dernière commande éclaire seulement dans une direction définie.

Edit permet de manipuler un objet. Vous pouvez choisir une texture ou définir des propriétés particulières pour le corps sélectionné.

Textures ouvre l'éditeur de texture. **Create** génère une nouvelle texture (prédéfinie). Celle-ci apparaît alors dans la liste des textures de toutes les fenêtres d'édition et il ne reste plus qu'à l'insérer dans les objets voulus.

Copy copie l'objet sélectionné et le transforme en même temps.

Files sert à sauvegarder, à charger et à exporter (Povray) des scènes. Si la scène n'a pas encore de nom, on peut lui en affecter en cliquant sur une ligne pour appeler le sommaire.

Pour manipuler un objet, il faut d'abord le sélectionner. On peut pour cela cliquer sur le bouton **Select.** On peut aussi maintenir la touche MAJ enfoncée et créer un cadre avec la souris (bouton gauche enfoncé) autour de l'objet.

Si l'on se trouve dans le menu principal, il est possible de transformer l'objet à partir de là. Il faut choisir pour cela SCL pour les changements d'échelle quelconques, USCL (Uniform Scale) pour les changements d'échelle uniformes dans toutes les directions, **Rotate** pour les rotations ou **Trans** pour les translations. On peut alors exécuter la fonction voulue soit en entrant les valeurs au clavier, soit encore à l'aide de la souris dans les champs 2-D.

La caméra joue d'ailleurs un rôle spécial : USCL modifie ici l'angle d'ouverture. **Trans** déplace d'abord le point de vue de la caméra : on peut modifier le point visé en appuyant sur **L** (Look_at). Inversement, on revient à la position initiale avec **P.**

Index

A

C

D

E

F

G

H

I

J

L

M

O

P

R

S

T

V

W

X

Achevé d'imprimer
sur les presses de l'imprimerie SAGER
28240 La Loupe
Dépôt légal - Mars 1995

D:\BONUS

* AARDVARK