Les compacteurs graphiques : des kilos en moins !

Ce qui apparaît à l'écran d'un 520 ou d'un 1040 ST représente en fait le contenu de 32 Ko de mémoire vive. D'où les efforts des programmeurs pour coder les graphismes sous la forme la plus dense possible.

Ne vous êtes-vous jamais demandé comment certains programmes, notamment de jeux, pouvaient afficher dix graphismes de suite sur des machines disposant par exemple d'une mémoire vidéo (ou VRAM) de 16 Ko et d'une mémoire vive totale de 64 Ko, le tout sans accès disque... Magie ? Bien sûr que non.

Il n'existe pas de méthode universelle de compactage des graphismes. La programmation de ces utilitaires dépend de la machine et de la structure de sa VRAM, mais aussi de l'application dans laquelle cette routine prendra place. Il est courant qu'un même programmeur développe plusieurs routines répondant aux différents cas rencontrés dans ses programmes. Aucun algorithme de compactage n'est parfait. Il doit résulter du meilleur compromis possible entre le gain de place et la rapidité d'affichage lors du décompactage.

Langages et machines : les plus rapides

Si tous les langages de programmation conviennent en principe, seules les routines écrites en assembleur, en C ou dans un autre langage compilé, seront assez rapides.

D'autre part, sur certains ordinateurs, la VRAM n'est accessible que par l'intermédiaire d'un processeur spécialisé, le VDP (*Video Display Processor*) ; c'est le cas, entre autres, du standard MSX. La lourde gestion de cette mémoire rend alors inutile de songer seulement à écrire un compacteur. Au contraire, sur les ordinateurs comme l'Atari ST, la gestion de l'écran est dite « bit map » (1), ce qui signifie que tous les points sont différenciés et non pas regroupés avec des attributs graphiques définis uniquement au niveau de ces ensembles.

Dernière remarque relative à l'écriture d'une routine de compactage (et de décompactage) : on évite d'utiliser les ressources-système (lecture de la couleur d'un point par exemple) à cause de leur lenteur.

La VRAM des Atari occupe 32 Ko consécutifs, logeables n'importe où en

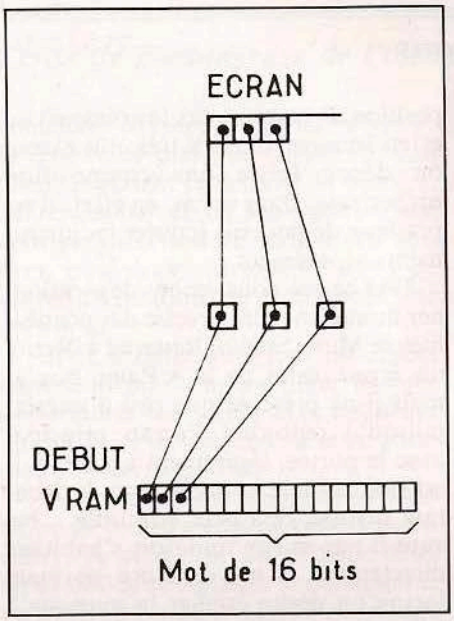

Fig.1 En haute résolution

RAM. Cependant, à l'initialisation de la machine, elle occupe les 32 derniers Ko de la RAM, soit de h78000 à h7FFFF sur 520 ST et de hF8000 à hFFFFF sur 1040 ST.

Si nécessaire, le processeur vidéo ira chercher ailleurs en mémoire les données relatives à l'affichage. Le déplacement de la VRAM doit être un multiple de 256 octets, car le processeur vidéo donne automatiquement la valeur 00 à l'octet de poids faible de l'adresse de base de la VRAM.

L'adresse où débute la VRAM étant mobile, elle est conservée dans un double-mot (32 bits) à une adresse qui, elle, ne bouge jamais (h44E sur le 520 ST comme sur le 1040).

Il ne nous reste plus qu'à comprendre la façon dont est gérée cette mémoire écran sur l'Atari, sachant que l'unité de mesure est le mot de 16 bits. Le premier mot définit les pixels du coin supérieur gauche de l'écran. Le bit de poids fort d'un mot correspond au pixel de gauche. En mode haute-résolution, toutes les lignes de l'affichage sont composées de 40 mots déterminant chacun 16 points. Selon qu'il est à 0 ou à 1, le bit indique si le point auquel il correspond est noir ou blanc.

Point par point

On obtient donc 40 * 16 = 640 points. Au total donc, 40 mots * 400 lignes = 16 Kmots, soit 32 Ko (voir figure 1).

En moyenne résolution, avec quatre couleurs possibles, il faut deux bits (2^2 = 4 combinaisons : 00, 01, 10 et 11) pour coder l'état d'un point. Dans ce mode, chaque ligne est représentée par 80 mots. Les bits occupant la même position dans chaque paire de mots consécutifs indiquent la couleur utilisée (voir figure 2).

Pas de surprise : la résolution est bien proportionnelle au nombre des couleurs simultanément disponibles. Si je multiplie par deux le nombre de bits utili-

(1) Bit map : un affichage est dit « bit map » quand il est entièrement graphique (dessins et *textes). Chaque pixel est alors indépendant de tous les autres.*

sés pour définir chaque pixel, alors je divise d'autant la résolution obtenue, car je travaille toujours avec 32 Ko d'espace mémoire.

Enfin, en basse résolution, la marche à suivre est la même que pour le mode intermédiaire mais en prenant cette fois quatre mots consécutifs d'affilée (voir figure 3).

Du plus simple au plus compliqué

Votre compacteur idéal sera nécessairement le fruit de plusieurs essais. En effet, à moins d'être devin, vous ne pouvez pas prévoir le rapport gain de place/vitesse de traitement que vous obtiendrez.

La première solution qui vient à l'esprit est un algorithme simple :

```
INDICE pointe sur le début d'une liste
(tableau ou espace mémoire) ;
si mode écran = haute résolution
alors TOTAL—POINTS = 256000
sinon
   si mode écran = moyenne résolution
   alors TOTAL—POINTS = 128000
   sinon TOTAL—POINTS = 64000
POINT = 1
DERNIERE COULEUR = couleur(POINT)
COULEUR = DERNIERE COULEUR
(*)COMPTEUR = 0

tant que COULEUR = DERNIERE COULEUR et que POINT < =
TOTAL POINTS faire :
   COMPTEUR = COMPTEUR + 1 ; POINT + POINT + 1
   COULEUR = couleur(POINT)
   fin bloc
ranger COMPTEUR ou pointe INDICE
INDICE = INDICE + 1
ranger DERNIERE COULEUR ou pointe INDICE
INDICE = INDICE + 1
DERNIERE—COULEUR = COULEUR
si point < = TOTAL POINTS
alors aller en (*)
```

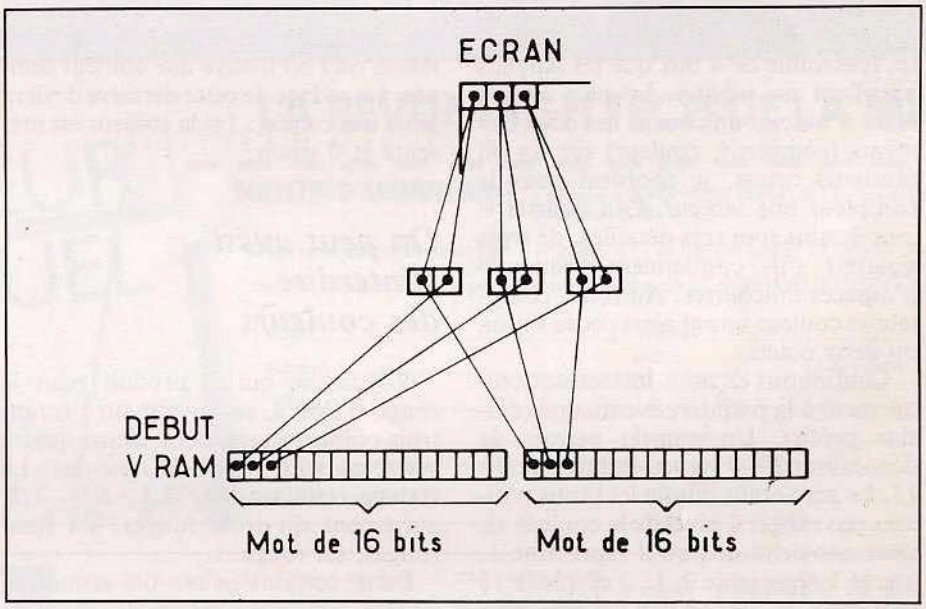

Fig. 2 En moyenne résolution

Cette méthode de compactage qui consiste à compter le nombre de pixels consécutifs et de la même couleur en sauvant à chaque changement compteur et couleur associée a l'avantage de la simplicité et n'est pas la moins efficace.

En fait, la difficulté réside dans les choix à effectuer lors de la programmation proprement dite. Le but étant le gain de place, il faut choisir le codage adéquat pour le rangement du compteur et de la couleur associée. Dans notre exemple, il n'est pas précisé le type utilisé pour le compteur (entier, réel ou encore octet, etc.).

Si l'on choisit le type entier, chaque rupture de couleur « consomme » deux octets pour le rangement du compteur. Si le dessin contient des trames, il y a de fortes chances pour que le résultat soit plus gourmand en place mémoire que son modèle: C'est la raison pour laquelle il est prudent de toujours vérifier à la fin du « compactage » que le code obtenu est plus compact que le code initial...

De plus, vous aurez peut-être remarqué une faille dans mes explications. Pour ne rien laisser au hasard, il faut rappeler qu'un entier, dans la majorité des langages, est une valeur comprise entre -32768 et +32767 ; or le nombre total de points même dans la plus basse résolution est de 64000, ce qui peut entraîner des erreurs. Le choix de la « largeur » (2) de codage doit répondre à deux critères : la cohérence et la facilité de traitement. Le premier point décidera du pourcentage de place gagnée, le second de la vitesse de décompactage (ainsi que du temps de conception de la routine).

Conservons notre premier exemple et étudions le cas de la basse résolution. Une façon de procéder consiste à fixer la grandeur du codage du compteur en

Les compacteurs graphiques : des kilos en moins !

fonction de la largeur occupée par le codage de la couleur. On tient compte aussi du nombre total de pixels à l'écran.

Une couleur est ici codée sur un quartet (ensemble de 4 bits que les Anglais appellent un *nibble*). Le plus simple étant d'obtenir un codage des deux éléments (compteur, couleur) sur un ou plusieurs octets, je choisirai pour le compteur une largeur d'un quartet si mes dessins sont très détaillés, de trois quartets s'ils contiennent beaucoup d'espaces unicolores. Au total, compteur et couleur seront ainsi codés sur un ou deux octets.

Continuons en nous intéressant uniquement à la première éventualité (dessins précis). Un quartet permet de dénombrer $2^4 = 16$ possibilités, de 0 à 15. Le zéro étant inutile ici (nous n'allons pas ranger 0 pixel de la couleur z), nous conviendrons qu'il représente 1, que le 1 représente 2, (...) et que le 15 représente 16. Il n'en faut pas plus pour gagner très probablement de précieux octets à l'exécution. C'est un premier pas vers l'optimisation. Ce choix nous oblige à ajouter à l'algorithme donné en exemple un test empêchant que la valeur du compteur ne dépasse 16.

Un tout autre raisonnement consiste à se demander quels sont les emplacements où l'on trouve une couleur donnée. Le codage de cette dernière devient alors très concis : 1 si la couleur est présente et 0 sinon.

On peut aussi s'interdire des couleurs

Voyons ce qui se produit pour le rouge si l'on a, se suivant sur l'écran, trois points rouges, deux bleus, quatre verts, puis un rouge de nouveau. Le codage résultant est : 3,1 - 6,0 - 1,1, autrement dit trois rouges, six non-rouges, un rouge...

Dans certains jeux, on remarque que les principaux graphismes ne mettent en œuvre qu'un nombre limité de couleurs. Il s'agit là d'une solution simple et rapide visant à diviser par deux la place occupée par le dessin. En moyenne résolution, on ne s'autorise que deux couleurs simultanées au lieu de quatre et le dessin n'occupe plus que la moitié de son espace théorique.

En basse résolution, les avantages de cette méthode sont moindres : il faut diviser par quatre le nombre des couleurs disponibles pour diviser par deux la taille de la mémoire occupée par un dessin (je vous laisse le soin des calculs).

D'autres systèmes de compactage proposent la définition de graphismes de base (sous forme de carrés en général) qui sont affichés dans l'ordre donné par le programmeur. Le principe est le même que celui qui vaut pour les affichages de caractères.

Bien entendu, nous n'avons pas passé en revue toutes les méthodes possibles. Elles sont trop nombreuses et, qui plus est, souvent employées combinées les unes aux autres. Fait intéressant à noter, la vitesse du décompactage ne dépend pas nécessairement de celle de l'opération inverse. S'il faut plus d'une minute à certaines routines pour coder une image, rien ne dit qu'elle ne s'affichera pas en moins de deux secondes...

□ **Yannick Cadin**

(2) Entier, octet, quartet ou toute autre combinaison de bits.

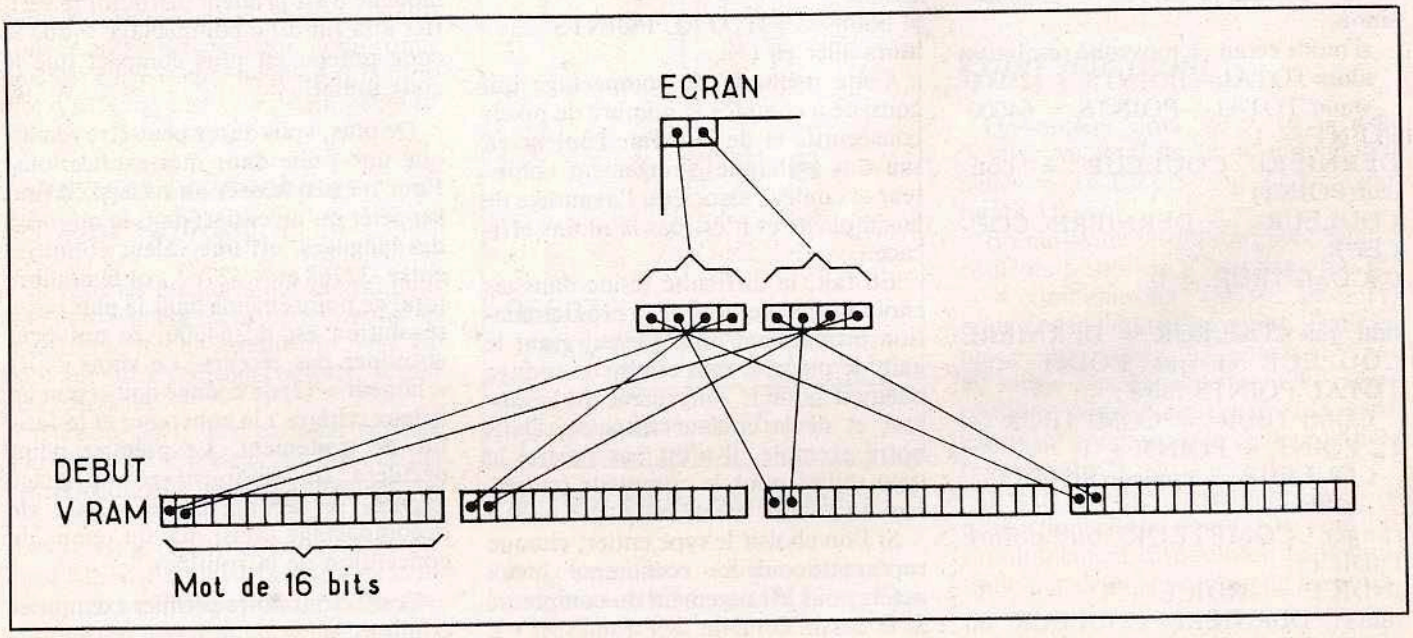

Fig. 3 En basse résolution