# Algorithmes de gestion des fonds dans les jeux

Clément Pillias (HPFool)

30 juin 2001

## Introduction

Puisqu'on me l'avait demandé, voici un rappel des différentes méthodes existantes pour traiter les images de fond dans les jeux, ainsi qu'un algo de ma composition que j'appelle "algo de l'écran cyclique", particulièrement adapté aux scrollings différentiels...

Avant tout j'aimerai préciser trois choses :

- Cet article est tiré d'un message que j'avais écrit sur le newsgroup français des developpeurs pour calculatrices HP : news.hpcalc.dev (accessible depuis les serveurs news.hpcalc.net et news.zoy.org, le second étant plus rapide).
- Cet article est assez gros, aussi je trouverai dommage qu'il ne profite qu'aux gens (peu nombreux) qui lisent ce newsgroup. Avis à ceux qui ont un site web (et aux autres !) : je les autorise à rediffuser cet article sous la forme de leur choix et à le completer comme ils le désirent, tant qu'ils ne modifient pas ce que j'ai écrit, et que les ajouts soient clairement signalés.
- Ca fait longtemps que je me suis intéressé au problème, et comme j'ai déjà eu l'occasion de le montrer, ma mémoire n'est pas éternelle, aussi il est possible que je fasse quelques erreurs que je n'aurais pas faites quelques années plus tôt, j'espère que vous me pardonnerez et que vous me corrigerez si vous en trouvez ;)

Bon on s'y met ? Allez c'est partit !

## 1 De la structure d'un programme utilisant un écran de fond

La structure générale d'un tel programme est la boucle suivante (tout au long de cet article je supposerai qu'on utilise du double buffering et que vous savez ce que c'est !) :

```
'Initialisations diverses
```

```
{
  'Initialisation de l'image que l'on va construire : affichage du
   fond
  'Affichage des plans superposés au fond avec transparence
  'Affichage des sprites
  'Autres affichages divers (nombre de vies, compteur, etc...)
  'Divers (gestion des touches, des déplacements et collisions, IA,
   réseau, café (euh non !), etc...)
  UP
}
```

On peut en fait voir l'image comme une superposition de plusieurs plans successifs transparents (des calques !) sur une image de fond (qui n'a pas de transparence et qui peut éventuellement être un plan de couleur unie). Les sprites représentent un plan particulier puisqu'il contient des éléments mobiles les uns par rapport aux autres, alors que les plans de décors sont "figés". Remarquez qu'il peut y avoir plusieurs plans de sprites (par exemple quand mario saute sur une plate-forme mobile, il passe devant sans la toucher, pourtant il s'agit d'un sprite). Ici je ne parlerais pas des sprites : ni de leur affichage, ni de leur gestion, ni de leurs collisions... on se concentre sur les décors.

Les décors peuvent avoir diverses caractéristiques de taille et de scrolling (différentes pour chaque plan), mais ils ont tous quelque chose en commun : une relation d'ordre. Certains plans sont en effet DEVANT d'autres. Cette relation définit l'ordre dans lesquels les différents plans doivent être affichés : Les plans les plus éloignés (c'est-à-dire les plus "derrière") sont affichés en premiers. Ainsi le fond est la première chose affichée. Le nombre et la forme des plans de décors et du fond nécessite différents traitements et permet différentes optimisations que nous allons voir maintenant en commençant par le cas le plus simple :

# 2 Fond unis et pas d'autre plan de décors

Bon je sais, ce chapitre aurait pu être sauté tellement ça semble évident, mais il pose des bases importantes !

C'est vraiment le plus simple : Il suffit de remplir l'image avec une couleur unie. On a tout de même plusieurs méthodes suivant le niveau d'optimisation souhaité et le codage des images.

Vous n'êtes en effet pas sans ignorer qu'il existe deux manières fondamentales de stocker des images en niveaux de gris (ah vous l'ignoriez ? désolé !) :

- Le format vertical (ou "double hauteur" pour des images en 4 nivs de gris) : l'image est divisée en différentes couches successives en mémoire, ce qui fait un format "GROB largeur x (hauteur * nombre de couches)".

  C'est le format utilisé par la plupart des éditeurs et viewers de GROBs et par les GREYs du MétaKernel. Historiquement c'est le premier apparu, car le plus simple à mettre en place (quand on a pas l'habitude !). Son principal inconvénient est de nécessiter autant de boucles qu'il n'y a de couches dans l'image.
- Le format horizontal (le vrai nom est "entrelacé", mais on dit aussi "double largeur" pour les images en 4 niveaux de gris) : Ici chaque ligne d'une couche est suivie immédiatement des lignes correspondantes dans les autres couches,

avant de passer à la ligne suivante de la première couche. Il en résulte un format du style "GROB (largeur * nombre de couches) x hauteur".

Ce format est principalement utilisé avec les sprites (merde j'avais dit que j'en parlerais pas !) ou dès qu'il y a de la transparence à gérer, tout simplement parce qu'avec la transparence on doit en plus gérer un masque qui s'applique à toutes les couches de l'image, autant alors avoir toutes ces couches le plus près possible !

De plus ce format donne des algos plus courts et plus rapides puisqu'il ne nécessite pas de faire plusieurs passages (une seule boucle, mais un peu plus grosse !). Notez qu'il existe deux variantes de ce format suivant qu'on utilise une largeur ajustée au quartet ou à l'octet près, mais ça n'a pas d'importance pour la suite.

– Le format spirale (ou double zéro 7 dans le cas d'images avec 7 niveaux de gris) : Cherchez pas, c'est une connerie ;)

Revenons à notre fond unis : ces histoires de format n'ont aucune influence si le fond est entièrement blanc ou entièrement noir, mais en ont sinon, puisque les différentes couches de l'image sont de couleurs différentes. On a donc deux algos :

```
%% ALGO 1 : à utiliser pour des images blanches ou noires ou
%% dans le cas d'images au format vertical (l'algo est alors à
%% multiplier autant de fois qu'il y a de couches dans l'image)

%Initialisation : D0 pointe sur le début de l'image
A=0 W
% insérer ici A=-A-1 W si l'image (ou la couche) est noire
LC ((largeur de l'image en quartets)*(hauteur de l'image))/16 - 1
{
  DAT0=A 16
  D0+16
  C-1.B % adapter au champ nécessaire suivant la taille de l'image
  UPNC
}
% ici il faut éventuellement rajouter :
% si le produit largeur*hauteur n'est pas un multiple de 16.
DAT0=A x % où x vaut : ((largeur de l'image en quartets)
         % *(hauteur de l'image)) modulo 16
```

Remarques sur cet algo :

– Il écrit toute l'image d'un coup, sans tenir compte du découpage de celle-ci en lignes, c'est pourquoi il ne peut être utilisé pour des images au format entrelacé.

– Il peut être optimisé en vitesse : Il suffit pour cela de "dérouler la boucle" (c'est une technique classique en informatique qui consiste à écrire $n$ fois le corps de la boucle (sans le test de fin, donc ici : `DAT0=A 16 D0+16`) sans oublier de diviser le nombre d'itérations de la boucle par $n$. Cela à l'avantage de diviser par $n$ le nombre de tests effectués sur le compteur (et donc d'accélérer), mais à l'inconvénient de multiplier la taille de la routine par $n$ (environ).

– Dans la suite nous verrons d'autres algos dérivés de celui ci, les remarques précédentes s'y appliqueront aussi !

```
%% ALGO 2 : A n'utiliser que pour des images au format entrelacé
%% non blanches et non noires.

%Initialisation : D0 pointe sur le début de l'image
LC (hauteur de l'image)-1
B=C.B % adapter au champ nécessaire suivant la hauteur de l'image
A=0 W
C=0 W
C=-C-1 W
{
  % utiliser A ou C suivant la couleur de la couche
  DAT0=A ou C 16 D0+16
  ... % Recopier la ligne précédente suffisamment de fois
  ... % pour remplir toute une ligne.
  ... % par exemple pour une image 131*64, il faut faire :
  ... % DAT0=C.16 D0+16 DAT0=C.16 D0+16 DAT0=C.B D0+2

  ... % recommencer tout ce qu'il y a avant dans la boucle
  ... % autant de fois qu'il y a de couches.
  B-1.B % adapter au bon champ
  UPNC
}
```

Remarques sur cet algo :

– Il est à peine plus rapide que l'algo 1 (sans déroulage de boucle), mais plus gros !

– On peut aussi dérouler la boucle, mais l'algo est déjà assez gros donc il faut vraiment que ce soit nécessaire ! ;)

– Dans la suite nous verrons d'autres algos dérivés de celui ci, la remarque précédente s'y appliquera aussi (mais pas la première car celui-ci sera alors bien plus rapide !).

Attaquons nous maintenant au problème des fonds non unis (images !)

# 3 Fond non unis sans autre plan de décors

Maintenant qu'on doit gérer des images, cela pose d'autres problèmes qu'on n'aurait pas supposé tout d'abord : en effet dans le cas d'une image on pourrait se dire qu'il suffit de reprendre les deux algos précédents et de les adapter un petit peu, en utilisant les deux pointeurs : `D0` pointant sur l'image à créer et `D1` pointant sur l'image de fond enregistrée quelquepart (on ne travaille bien sûr pas directement sur celle-ci pour ne pas l'abîmer !). On n'aurait alors qu'à lire quelque chose avant de l'écrire et incrémenter les pointeurs (en gros on remplacerait les séquences "`DAT0=A 16 D0+16`" par "`A=DAT1 16 DAT0=A 16 D0+16 D1+16`".

C'est en effet une bonne idée mais elle pose deux problèmes :

– Comment gérer les scrollings ?
– Comment stoker l'image ?

En fait ces deux problèmes ne se posent que si l'image de fond est de dimension importante, et essentiellement pour des questions de mémoire. En effet dans ces conditions on ne peut pas utiliser le scrolling "hard" (utilisation de la marge à gauche et de la marge à droite) car l'image à scroller est trop grosse pour qu'on puisse raisonnablement l'utiliser ainsi (en plus son temps de traitement serait prohibitif), il faut donc utiliser une technique de fenêtrage que nous allons voir tout de suite.

Quant au stockage, il existe deux solutions pour l'éviter : la compression et le découpage en éléments de base (dont les tuiles !)

## 3.1 Techniques de fenêtrage pour le scrolling

La technique est super simple : il suffit de faire la même chose que dans l'algo 2 en se limitant à écrire 32 quartets et non la ligne entière. Après les 32 quartets il faut donc sauter (largeur de la ligne en quartets)-32 quartets et c'est bon. Ce saut peut se faire avec des `D0=D0+`... si l'image ne dépasse pas les 362 pixels de large, sinon ça se fait avec une méthode du style `CD0EX C+D.A CD0EX` où `Da` contient la valeur de l'incrément... Je ne m'étends pas plus longtemps là-dessus, j'en ai déjà longuement parlé dans un autre message, avec les études de temps de cycles détaillées.

## 3.2 Techniques de stockage

Il y a tout d'abord la compression. En fait c'est une fausse technique de stockage car la décompression est trop lente pour être faite en temps réel, il faut donc passer par une image temporaire décompréssée du fond et on revient au problème initial (ça permet juste de stocker plus d'images).

Ensuite il y a le découpage en éléments élémentaires. Cette méthode consiste à ne composer que des fonds utilisant des éléments de base (par exemple un arbre, un nuage, etc...) Avec suffisamment d'éléments de base on arrive à produire des fonds relativement corrects du point de vue graphique. L'avantage c'est qu'après on a juste à stocker les éléments de base quelque part, et le fond en lui même consiste en une suite de couples (Identifieur d'élément de base, position de l'élément dans le fond). Ce format en lui même prends déjà beaucoup moins de place en mémoire, et surtout il n'en demande pas de supplémentaire pour l'affichage, puisqu'il suffit de tester pour chaque élément de base composant le décors s'il est visible (au moins en partie) et de l'afficher comme un sprite si c'est le cas. Comme j'ai dit que je ne parlais pas des sprites, je ne vous expliquerait donc pas comment l'afficher ;-p

Toutefois cette méthode pose un problème fondamental : pour un niveau assez grand, il peut y avoir beaucoup d'éléments de base utilisés, et il faut pour chacun vérifier s'il est visible au moins en partie, ce qui peut être assez long, et qui a l'inconvénient de créer des ralentissements pour les niveaux graphiquement chargés. Pour remédier à cela, il existe diverses méthodes. Je ne vais pas rentrer dans le milliard d'optimisations possibles dans ce cas et je ne vais en citer qu'une, particulièrement facile à implanter lorsqu'on a un scrolling unidirectionnel (par exemple de gauche à droite, et pas en hauteur, comme dans Cyclo) : il suffit de trier la liste des éléments de base composant le fond par abscisse croissante. Ceci ne se fait bien sûr pas en temps réel, le fond doit être stocké ainsi (pré-traitement). Ensuite il suffit

de garder un pointeur dans la liste triée vers le premier élément visible à gauche. Alors ceux qui sont situés après dans la liste seront visibles, jusqu'à ce qu'il y en ait un invisible. Cela réduit considérablement le nombre d'éléments de base à tester.

Mais cette technique a encore quelques inconvénients :

- Comme elle utilise une routine de sprites, les éléments de base peuvent être à n'importe quelle abscisse, et il faut donc faire des décalages en conséquence, ce qui coûte du temps. Alors bien sûr, on peut se limiter à des éléments de base bien alignés tous les 4 quartets mais c'est assez vexant ;-)
- Comme elle utilise une routine de sprites, il faut initialiser le fond avant, et hop, retour aux algos 1 et 2, gourmands en temps ! ! !
- J'avais dit que je parlerais pas des sprites :-(
- En fait il ne s'agit pas d'un fond mais d'un plan transparent ! ! !

Heureusement, Zorro est arrivé ! Euh non, je voulais dire : il existe une solution sympathique : les tuiles !

Les tuiles fonctionnent exactement selon le même principe, sauf que tous les éléments de base (qu'on appelle maintenant "tuiles") ont la même taille (généralement un multiple de quatre, au moins en largeur pour résoudre le problème des décalages), et qu'il existe une tuile vierge, correspondant à l'absence de dessin (généralement numérotée 0). De plus, toutes les tuiles sont alignées dans un tableau, de façon à ce qu'elles recouvrent tout l'écran, sans trous. Comme sur un toit quoi, sauf que sur un toit les lignes de tuiles sont décalées ce qui n'est pas le cas ici, elles sont les unes au dessus des autres. Du coup la structure du fond devient un bête tableau contenant le type de la tuile correspondant à chaque case.

Inconvénients : ça prends généralement plus de place, et ça oblige les dessins à être alignés verticalement et horizontalement ce qui est un peu moins beau.

Avantages : pour savoir quelles tuiles sont affichées, il suffit de savoir quelle partie du tableau est visible et y'a rien de plus simple, puisque toutes les tuiles ont la même taille ! De plus comme ça couvre tout l'écran il n'y a pas besoin d'initialiser le fond avant ! Et pour finir (oh joie suprême !) si la largeur des tuiles est un multiple de 4, il n'y a pas de décalages au pixel près à faire ! ! ! (Ce décalage est déjà fait pour les sprites, on ne va pas le faire ici puisqu'on peut utiliser la marge à gauche pour décaler toutes les tuiles d'un coup ! ! !)

Voila, bien évidement tout cela va être réutilisé dans le chapitre suivant sur les plans transparents :

# 4 Les plans transparents

Ce que j'apprécie dans ce chapitre, c'est qu'il n'y a pas de surprise ;) Les plans transparents c'est EXACTEMENT le même problème que pour les fonds images, avec de la transparence en plus. Et cette transparence ce n'est qu'un problème d'affichage et non de stockage ou autres comme précédemment. Je vais donc supposer que vous savez traiter la transparence car elle se fait de la même façon que pour les sprites (avec des masques), et je vais passer au chapitre suivant sans remords !

(En fait j'exagère, car il y a quand même d'autres problèmes, comme les décalages qui étaient gérés par la marge à gauche dans le cas du fond et qui ne peuvent plus l'être dans le cas d'un scrolling différentiel puisque les différents plans ne scrollent pas à la même vitesse. Mais je passe car je ne connais pas de solution miracle à ce problème : Il faut se taper les décalages :( )

# 5 Fini les généralités, place aux optimisations !

Bon on a vu toutes les techniques classiques. Maintenant il faut vous avouer la triste vérité : ça rame !!!

Si vous voulez faire un jeu de plates-formes, avec juste un fond image, quelques plates-formes et quelques sprites, même avec une intelligence artificielle de très bas niveau, en tenant compte de tous les affichages à faire et des tests de collisions et autres choses qu'on peut trouver dans un tel jeu, et bien vous avez intérêt à savoir programmer correctement car sinon ça devient vite trop lent ! On ne s'étonne pas alors qu'il y ait si peu de jeux de plate-formes sur HP !!! On ne s'étonne pas non plus que le seul jeu de plate-formes jouable à disposer d'un scrolling différentiel n'ait qu'un seul sprite (je parle de Cyclo) !!!

A quoi est due cette lenteur ? Tout simplement à un problème de mémoire : tous ces affichages représentent des tas et des tas d'écritures en mémoire (cela se compte en Ko pour les images de fond !) et généralement sur des petits champs, et vous savez sûrement que l'écriture en mémoire est l'instruction la plus lente du Saturn, et que plus le champ est petit plus elle est lente (relativement) !

Il n'y a malheureusement rien à faire pour la plupart des parties du programme, à part en réduisant les possibilités de ce dernier (par exemple en limitant le nombre de sprites ou la taille des niveaux, ou les niveaux de gris, etc...). On peut toutefois faire plusieurs choses pour l'affichage du fond et des plans transparents.

## 5.1 Première optimization : écran tampon avec décalages

On l'as déjà vu, le principal problème posé par le fond et les plans transparents est lié à la mémoire : si on avait assez de mémoire pour stocker toute l'image sans être obligé de passer par des tuiles ou autres ça serait beaucoup plus rapide : on n'aurait que le fenêtrage à gérer ! Seulement c'est pas possible :( On peut toutefois imaginer des solutions intermédiaires, qui auraient l'avantage de la vitesse de l'image complète, et les avantages de mémoire des tuiles et autres, et ces solutions sont liées au scrolling.

En effet, pourquoi reconstruit-on l'image à chaque fois ? Parce qu'avec les scrollings cette image peut changer ! Mais bon c'est un peu exagéré non ? Il y a une bonne partie de l'image qui reste encore valide en cas de scrolling, elle est juste décalée ! Et puis on ne scrolle pas à chaque image non plus !!!

Ces remarques conduisent à une nouvelle optimisation : on construit chaque plan dans une image tampon de la taille de l'écran qu'on a alors plus qu'à recopier avec des algos ressemblant à l'algo 1 ou à l'algo 2. Cette recopie est beaucoup plus rapide que la reconstruction à partir des tuiles ou des éléments de base !!! On y gagne alors considérablement ! Il reste toutefois un problème : en cas de scrolling l'écran tampon n'est plus valide ! On ne va pas le reconstruire, ça n'aurait aucun sens ! On va simplement le décaler d'une colonne ou deux et puis reconstruire les colonnes qui sont apparues (tuiles forever !)

Et c'est effectivement un peu plus rapide, le décalage se faisant sur de grands champs (`W`), et la construction sur de petits (généralement `B`, `X` ou `A`), donc plus lents. Mais un décalage ça reste quand même très bourin !!! En terme de rapidité on peut faire mieux !

## 5.2 deuxième optimisation : fenêtre coulissante

C'est là que vient l'optimisation dite "de la fenêtre coulissante" ! Malheureusement cette optimisation ne marche que pour des scrollings unidirectionnels, mais elle est intéressante quand même. Elle nécessite cette fois-ci un écran tampon de taille double (dans le sens du scroling). Prenons par exemple le scrolling de Cyclo, vers la droite : Il faut une image tampon de largeur double par rapport à l'écran. Initialement elle est remplie de deux copies (une à gauche (pixels 0 à 130), une à droite) de l'image qui doit être affichée.

Imaginons qu'on se déplace vers la droite : on va alors mettre à jour la première colonne à droite de l'image de gauche (la colonne du $132^{\text{ème}}$ pixel). On a alors une image valide correspondant à ce qu'on doit afficher entre les pixels 1 et 132, qui pourra être recopiée sans remords dans l'écran en cours de construction. On va quand même fignoler un peu et recopier la colonne qu'on vient de mettre à jour dans la colonne située à gauche de la nouvelle image (la colonne du pixel 0 donc !). Pourquoi faire cela ? Et bien continuez encore 130 fois ! Vous aurez atteint la limite droite de l'écran tampon (colonnes 131 à 262), mais vous aurez grâce à ces copies de colonnes une copie de l'image dans la partie gauche (colonnes 0 à 131). Et hop vous pourrez continuer indéfiniment en repartant à gauche !!!

En fait pour bien être comprise cette technique nécessiterai un dessin. En voici un simpliste en ASCII-art :

```
Image du fond :

0123456789ABCDEF
_X_X_X_X_X_X_X_X
__XX__XX__XX__XX
____XXXX____XXXX
________XXXXXXXX

Ecran tampon initial (l'écran affiché fait 4 caractères de large) :

++++----
01230123
_X_X_X_X
__XX__XX
--------
--------

Ecran tampon après un décalage :

-++++---
41234123
_X_X_X_X
__XX__XX
X___X___
--------

Ecran tampon après 4 décalages :
```

```
----++++
45674567
_X_X_X_X
__XX__XX
XXXXXXXX
--------
```

Bon je sais que c'est pas très évident en ASCII art mais vous pourrez constater que les deux images sont toujours là mais décalées (comme une télé mal réglée). C'est ce décalage qui évite justement de décaler l'image à la main ! On à donc transféré la phase de décalage lors de la copie de l'écran tampon plutôt que lors de sa mise à jour ! ! ! Le principe des pointeurs appliqué dans toute sa beauté ! ;)

Bon bien sur, j'ai parlé de décalages au pixel près, en pratique on ne fait pas ça, on met à jour directement des colonnes de quartets, voire de tuiles !

Notez aussi que ça marche quelque soit le format des images (vertical ou entrelacé), et également pour des scrollings (uniquement) verticaux.

C'est pas beau ça ? Si, hein ? Le problème c'est que ça demande deux fois plus de mémoire même si ça reste raisonnable, et que ça ne marche pas avec des scrollings multidirectionnels ! En plus lors de la recopie, il faut passer de lignes de 262 pixels de large à des lignes de 131 pixels de large, on se retrouve alors avec le problème du format entrelacé...

...Et ça vous dirais que je vous donne une solution encore plus rapide et qui n'ait aucun de ces problèmes ? ? ?

# 6 Je suis un génie (sisi j'vous jure ! ! !)

Mon algo (enfin l'algo que je m'attribue puisque je l'ai trouvé par moi-même et que je n'en ai jamais entendu parler par quelqu'un d'autre) est basé sur la même idée de décalage, pour reporter la phase de décalage au moment de la recopie plutôt qu'au moment de la mise à jour... Seulement ici j'utilise une image de taille normale (et non double), que je rends CYCLIQUE !

Qu'est-ce que ca veux dire ? Dans le cas précédant l'image était déjà cyclique (comme une télé mal réglée : ce qui disparaît à un bord réapparaît de l'autre coté), mais c'était une cyclicité sur les lignes, alors que chez moi il s'agit d'une cyclicité sur toute l'image, c'est à dire sur la zone mémoire occupée par celle-ci.

En gros dans le cas précédant vous saviez dans quelle colonne de votre tampon débute l'image à recopier. Vous aviez donc un OFFSET sur cette colonne. Chez moi c'est pareil, sauf que l'offset ne désigne plus une colonne mais un quartet de l'image (celui qui contient le pixel en haut à gauche de l'image telle qu'elle doit être affichée).

Encore de l'ASCII-art ? Allez c'est partit (bien que je me demande si on puisse vraiment parler d'ART dans ce cas ;) )

Voici une image :

```
_X_X_X_X
__XX__XX
____XXXX
```

Et sa représentation mémoire (les lignes les unes après les autres) :

```
_X_X_X_X__XX__XX____XXXX
```

Ce qui nous donne, avec un offset de 15 caractères (les 15 derniers caractères sont passés au début) :

```
_XX__XX____XXXX_X_X_X_X_
```

Ce qui correspond à l'image (mais ça on s'en fout, car cette image là on ne la regarde pas, c'est juste pour vous donner une idée) :

```
_XX__XX_
___XXXX_
X_X_X_X_
```

Plus grand chose à voir avec l'original non ? ;)

Mais bon maintenant qu'on a bien rigolé, rentrons plus en détail dans l'algo... Comment dois-je faire pour recopier une telle image dans l'écran que je suis en train de construire ?

Bah c'est simple, je part de mon offset (ici le $15^{\text{ème}}$ caractère), et je commence à recopier l'image. Je recopie $24 - 15 = 9$ caractères pour arriver jusqu'à la fin du tampon, et je continue en retournant au début du tampon pour recopier les 15 caractères qui restent. Facile en fait ! Et rapide !

Et pour la mise à jour du tampon ? Imaginons que je me déplace vers la droite : je vais remplacer la colonne qui commence à l'offset par la colonne que je vois apparaître à droite, en partant de l'offset plus une ligne, puis en descendant. quand je sort du tampon, je remonte à la première ligne du tampon et je continue ! Et puis j'incrémente l'offset de 1 et le tour est joué !!! Facile et rapide !!!

Enfin facile... pas tant que ça, la routine de mise à jour de la colonne est un peu plus rusée que la routine habituelle pour tenir compte de la cyclicité de l'écran, mais ça reste supportable, il faut juste bien s'organiser !

Encore un exemple ? Si je devais faire apparaître la colonne :

```
O
L
E
```

dans l'image précédente j'obtiendrais :

```
_XX__XXL
___XXXXE
X_X_X_XO
```

dont la représentation physique est :

```
_XX__XXL___XXXXEX_X_X_XO
```

En recopiant ça avec la méthode que j'ai donnée et un offset de $15 + 1 = 16$ j'obtiens :

```
X_X_X_XO_XX__XXL___XXXXE
```

Ce qui corresponds à l'image :

```
X_X_X_XO
_XX__XXL
___XXXXE
```

Ce qui est bien ce que je voulais, non ? Simplicité, efficacité, JLD ;)

Bon quels sont les inconvénients ?

– C'est un peu plus dur à mettre en place.
– Ne règle pas le problème des décalages au bit près :( (promis dès que je trouve une méthode d'enfer à ce sujet je vous prévient ;) ).
– Nécessite un écran supplémentaire.

Et les avantages ?

– Presque aussi rapide que si on avait une vrai image plutôt que des tuiles.
– Règle le problème des décalages au quartet prés ;)
– Ne nécessite QU'UN écran supplémentaire ;)
– Marche quelque soit le format d'image utilisé (vertical ou entrelacé)
– Marche aussi bien pour les scrollings multidirectionnels que pour les scrollings unidirectionnels.
– C'est moi qui l'ai trouvée !!! (sisi ça compte ! ;) )

Bon, je crois avoir fait le tour, je sais que j'aurais pu rentrer plus dans les détails sur certains points, mais vous êtes gentils j'ai pas que ça à faire ;) C'est déjà pas mal je trouve. Mais je suppose qu'à cause de ça il y a des trucs que vous ne comprendrez pas alors n'hésitez pas à poser des questions (sur les newsgroups) ...