LA MÉMOIRE ET LES SPRITES PARTIE 1

Lecon du jour

La memoire et les sprites Partie 1

Comme promis dans Côté Gamers 4 nous allons commencer à aborder la notion de *sprites* avec des exemples provenant de la prodigieuse logithèque de l'Amstrad CPC. Nous sommes aidés par Christophe PETIT qui a entre autres travaillé sur « CPC Aventure » en 2005.

DÉFINITION

D'après Wikipedia : un *sprite* (en français lutin) est dans le jeu vidéo un élément graphique qui peut se déplacer sur l'écran. En principe, un *sprite* est en partie transparent, et il peut être animé (c'est-à-dire qu'il est formé de plusieurs *bitmaps* qui s'affichent les uns après les autres).

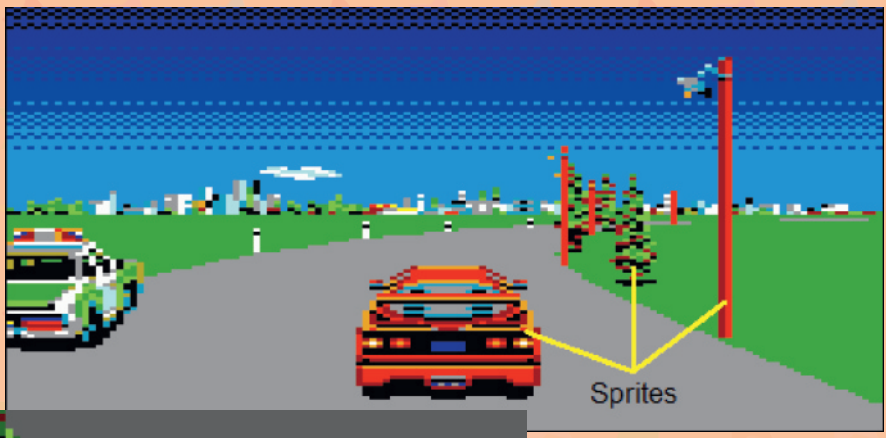
CARACTÉRISTIQUES GÉNÉRALES DES *SPRITES*

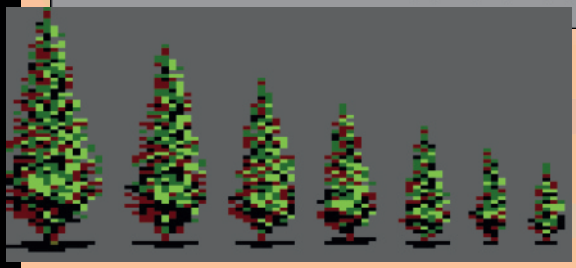
ILS SONT TOUJOURS EN 2D

Les *sprites* sont de simples images 2D (Deux dimensions). D'ailleurs, dans les premiers jeux en 3D (Trois dimensions) où ils étaient encore utilisés pour les adversaires, les monstres ou les arbres, on les reconnaissait au fait qu'ils présentaient toujours la même face quand on tournait autour (Vous vous rappelez sans doute sur PC de "Wolfenstein 3D" © 1992 ID Software, de "Doom" © 1993 ID Software ou encore de "Doom II: Hell on Earth" © 1994 ID Software eh bien si vous regardez les deux écrans ci-dessous, vous constaterez que les cadavres vous font toujours face que vous regardiez d'un côté ou de l'autre du couloir, ce sont nos fameux *sprites* en 2D).

Revenons sur notre Amstrad CPC.

Exemple : dans "Crazy Cars 2" 1989 © Titus, les voitures et tous les éléments de décors sont des *sprites*.





L'arbre possède 7*sprites* de taille différente permettant de faire une animation pour simuler un éloignement.

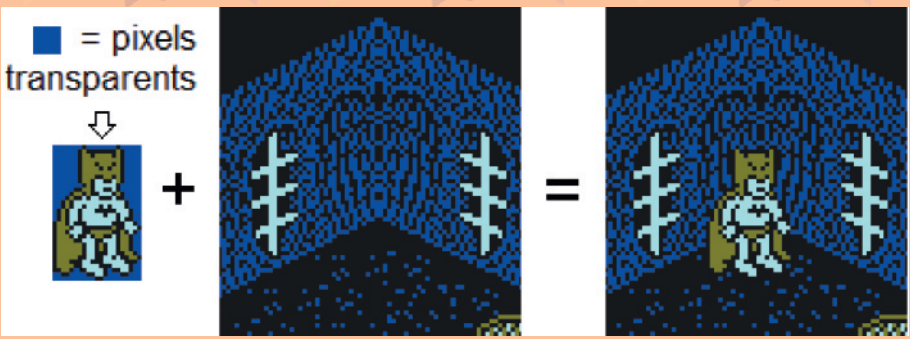
Votre Ferrari F40 possède 8 *sprites*, ce qui permet d'obtenir le 360° si spectaculaire lorsqu'on rétrograde brutalement et qu'on donne un coup de volant à droite ou à gauche. Certes nous n'avons que l'équivalent de seulement 180°, mais grâce à une technique de FLIP il est possible d'afficher les 180° manquants.

ILS SONT EN PARTIE TRANSPARENTS

Les *sprites* sont des images rectangulaires. Bien entendu il y a des exceptions, comme dans certaines productions du célèbre David PERRY où les *sprites* sont auto générés (*sprites* codés) et que ceux -ci sont gérés selon leur forme et pas dans un rectangle. C'est donc presque toujours vrai mais pas toujours.

Lors de leur affichage, les pixels inutiles sont soit remplacés par les pixels du décor soit non affichés (tout dépend comment c'est géré). On dit alors que ces pixels sont transparents.

Chose intéressante dans ce jeu, nous avons des *sprites* en MODE 2 (2 couleurs) alors que le jeu lui est en MODE 1 (4 couleurs). Bien entendu, le programmeur a bien le droit de sortir des sentiers battus.



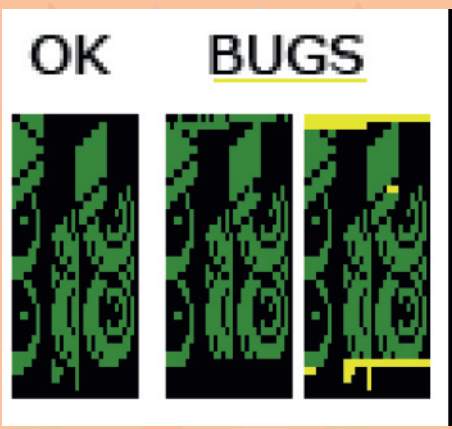
"Batman" 1986 © Ocean Software, possède quatre couleurs dont l'une est utilisée pour la transparence. Il ne lui reste donc que trois couleurs affichables : kaki, noir et bleu ciel (C'est ce qui aurait dû être). La couleur transparente est en général la 0, nous verrons pourquoi en programmation. Ici cette couleur est la seule qui n'est pas utilisée par Batman. C'est donc le bleu foncé qui sert également au décor.

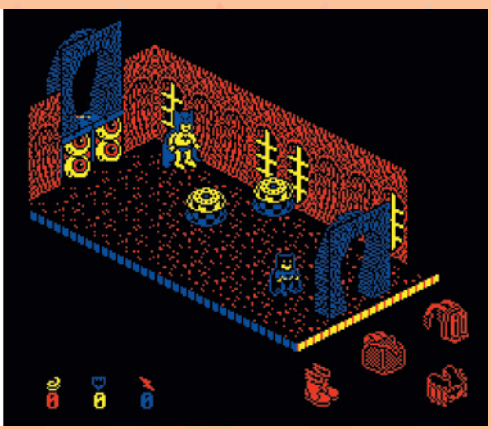
Le programmeur aurait pu faire ça pour économiser de l'espace mémoire (mais en même temps, il aurait perdu du temps CPU pour transformer les informations du MODE 2 en MODE1), car du coup, 8 pixels = 1 octet alors que si le *sprite* était en 4 couleurs, 8 pixels = 2 octets, ça c'est ce que nous aurions pu croire, mais la vérité est ailleurs.

La raison du choix d'utiliser des graphismes en 1bitplan est dûe au fait que c'est un "Speccy Port" (Une adaptation d'un jeu fonctionnant sur ordinateur ZX Spectrum, et ils sont légion sur Amstrad CPC à avoir été adaptés de cette machine).

La version ZX Spectrum (ça manque de couleurs, on comprend mieux pourquoi nous avons 2 couleurs en mémoire sur Amstrad CPC).

A noter que sur la version disquette et uniquement sur cette dernière (contenant un fichier de 44Ko), Il existe un bug graphique au niveau des yeux sous les portes (Nous ne retrouvons pas le bug dans la version cassette), le fait de savoir cela nous a permis de corriger nos disquettes et nous verrons un peu plus loin dans l'article que Tiles ou Sprites, la frontière est très proche :





Dans "Renegade" 1987 © Ocean Software édité sous le label Imagine Software, la couleur qui a été sacrifiée est la grise (encre 0). Pour économiser de la mémoire, les jambes, le corps et la tête sont stockés séparément, ce qui permet de réutiliser par exemple les jambes sur plusieurs adversaires, ainsi que le corps.

TRANSPARENCE ET MASQUES

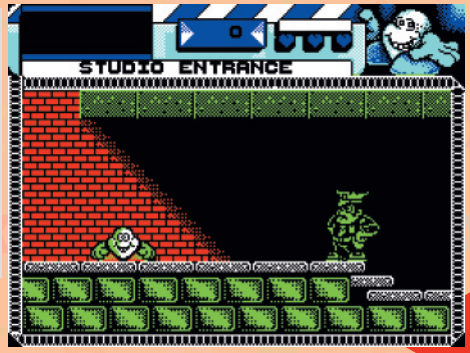
Dans "Rath-Tha" 1989 © Positive, au lieu d'avoir une image, nous en avons deux (des masques + des *sprites*), ce qui permet de ne pas sacrifier de couleurs (faudrait-il encore que le logiciel utilise toutes les couleurs, ce n'est pas le cas, car tout simplement nous sommes encore en présence d'un portage d'une version ZX Spectrum).

Le masque est constitué de deux couleurs, une couleur qui correspond à ce qui sera affiché (ici le noir) et une couleur qui correspond à ce qui ne sera pas pris en compte (ici le jaune).

Dans l'écran du jeu, nous retrouvons notre espèce de canette jaune ainsi que nos tirs dans la planche de *sprites* au dessus. Notre vaisseau est lui aussi présent.

Amstrad CPC. **ZX Spectrum.**

Dans "Seymour At The Movies" 1991 © Codemasters Software, nous avons là aussi l'utilisation d'un masque, mais il est différent du logiciel précédent, certes nous avons toujours l'équivalent de deux images utilisées, mais celle-ci est alternée entre un octet de masque suivi d'un octet pour le *sprite* et ainsi de suite. A noter, que notre personnage possède bien 4 couleurs (blanc, noir, cyan, bleu) aucune couleur n'est donc sacrifiée dans ce jeu en MODE 1 (4 couleurs). L'alternance des octets de *sprite* et de masque est une optimisation pour la routine d'affichage. Ainsi on n'a pas besoin d'aller voir ailleurs en RAM puisque les octets se suivent.



Exception : les pixels transparents de la même couleur que le fond

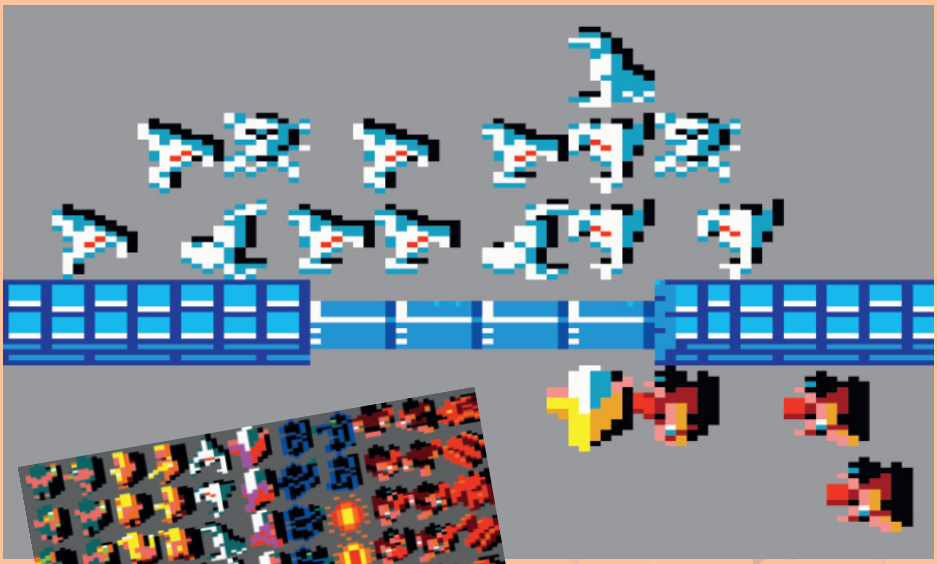
Dans les années 1980, les décors à fond d'écran uniforme étaient parfaitement acceptables. Du coup il suffisait que les pixels transparents soient de la même couleur que le fond pour régler au moins deux problèmes :

- Plus de gestion de la transparence : gain de temps et de fluidité.
- Gestion de collision simplifiée : tout ce qui n'est pas couleur de fond est un obstacle ou un ennemi.

Afficher les *sprites* tels quels permettait donc de gagner de précieux cycles machine.

Exemple : dans "Gauntlet II" 1986 © U.S.Gold, le gris sert à la fois de couleur de sol et de transparence. Les *sprites* peuvent donc être affichés tels quels avec une routine rapide.

En plus, les décors ont des couleurs différentes de celles des monstres, ce qui simplifie la détection de collision.



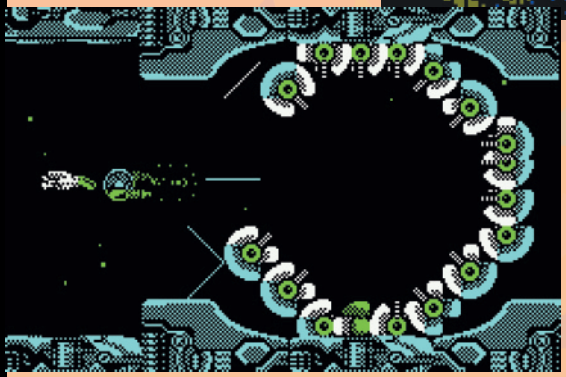
Une planche avec quelques graphismes du jeu concernant le 1er joueur (à dominance verte) et le 2ème joueur (à dominance jaune), ainsi que plusieurs adversaires (Les couleurs des joueurs varient suivant ce que vous avez choisi lors de la sélection des personnages).

ILS PEUVENT ÊTRE DÉPLACÉS

Les *sprites* ne sont pas réservés qu'aux personnages ou aux vaisseaux. Il peut également s'agir d'objets déplaçables. ►

ILS PEUVENT S'ADDITIONNER

Il est en effet possible de coller plusieurs *sprites* ensemble pour en former un autre plus gros. ▼



ILS PEUVENT ÊTRE ANIMÉS

Les *sprites* peuvent être animés, mais ce n'est pas une obligation. Par exemple un vaisseau spatial n'a pas forcément besoin d'être animé. En revanche un personnage doit généralement l'être, au moins au niveau des jambes.

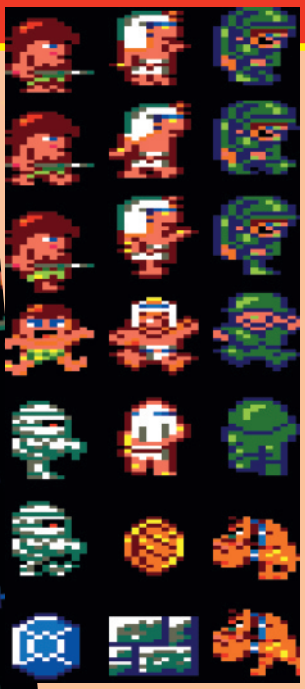
Il y a deux façons d'animer un *sprite* : en modifiant la palette de couleurs ou le bitmap du *sprite*. La méthode par changements de couleurs a très peu été utilisée sur CPC, sauf pour un effet de clignotement de temps en temps. Par contre l'animation par changement de bitmap est la base de quasiment tous les jeux.

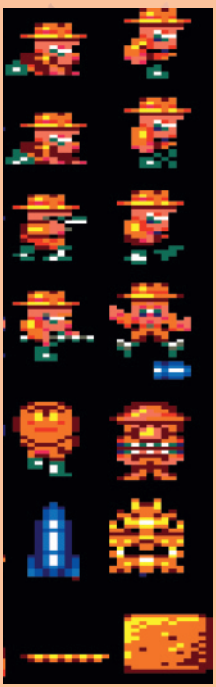
Exemple : Dans "Zub" 1987 © Mastertronic, quand le héros saute, quatre bitmaps s'enchaînent dans l'ordre 1,2,3,4,3,2,1.

S'ILS NE BOUGENT PAS, ÇA N'EN N'EST PAS

Les *tiles* (tuiles en français) sont des motifs répétitifs permettant de construire un décor peu coûteux en mémoire. Comme les *sprites*, ce sont des images rectangulaires pouvant être animées (exemple : une torche sur un mur). La différence c'est qu'ils ne peuvent pas bouger, ce ne sont donc pas des *sprites*.

Ci-dessous, une capture de plusieurs *tiles* et *sprites* de Rick Dangerous, on constate que pour la chauve-souris, la momie ou encore le chien, nous n'avons que deux *sprites*, il n'est parfois pas besoin de plus pour créer l'illusion de vie dans un jeu.





LES TYPES DE *SPRITES* SUR CPC

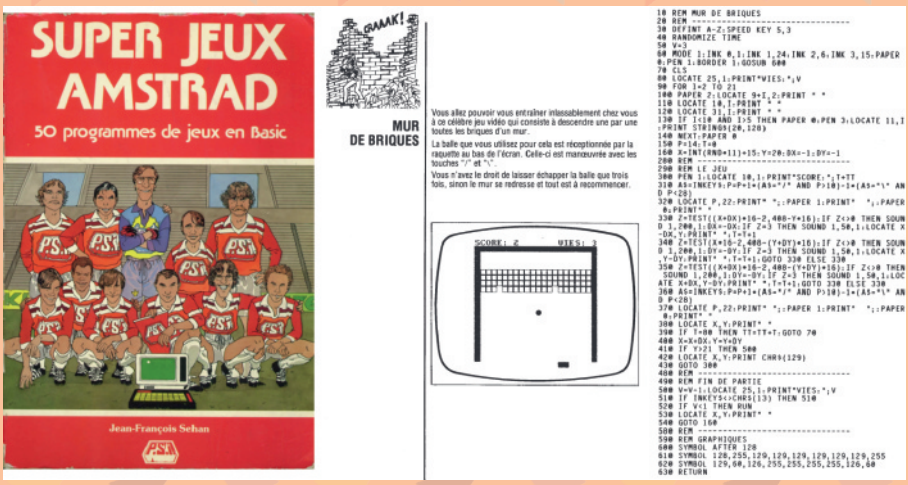
INTRODUCTION

Sur CPC, il y a généralement quatre types de *sprites* :

1-Les *sprites* textes réalisés à partir de la table des caractères ASCII.
2-Les *sprites* logiciels conventionnels, avec un bloc mémoire par *sprite* et des routines d'affichage, d'effacement, d'animation et de déplacement.
3-Les *sprites* codés qui fusionnent les *sprites* et leur code d'affichage dans une même routine (*sprites* auto générés).
4-Les *sprites* hard des CPC+ intégrés dans un circuit électronique spécialisé, l'ASIC.

LES SPRITES TEXTES

Les caractères textes sont des images que l'on peut déplacer, animer, assembler et même modifier. Bref, ce sont aussi des *sprites*. A l'époque c'était l'idéal pour tous les novices qui souhaitaient apprendre à programmer. Quelques dizaines de lignes en BASIC suffisaient à créer un jeu, comme le montre le livre « Super jeux AMSTRAD » de chez PSI.



Ils ont toutefois été très peu utilisés dans les jeux, car même si nous pouvons fusionner des caractères entre eux grâce à l'activation de la transparence avec PRINT CHR$(22);chr$(1) et sa désactivation avec PRINT CHR$(22);chr$(0), ça reste très lent et très lourd à utiliser et nous sommes tributaires du couple colonne/ligne.

Exemple : "Jet Set Willy" 1984 © Software Projects est l'adaptation du même jeu sorti sur ZX Spectrum. La gestion particulière des couleurs sur ZX Spectrum faisait qu'il était naturel d'utiliser des sprites textes pour rester fidèle à l'original.

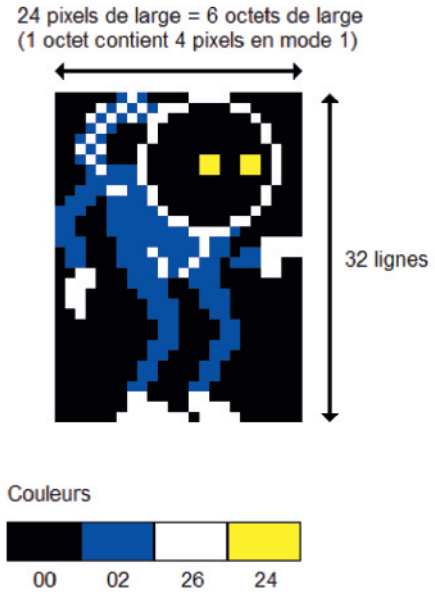
Chose étrange, les *sprites* sont créés dans des zones en 16x16 avant d'être affichés sur l'écran. Ce qui permet au programmeur de remplacer la couleur du *sprite* à l'écran.

LES SPRITES LOGICIELS CONVENTIONNELS

Les *sprites* logiciels conventionnels sont de loin les plus utilisés sur CPC. Par rapport aux *sprites* textes, leur intérêt est évident : ils peuvent avoir plusieurs couleurs et ne sont pas limités à une taille multiple de 8x8 pixels. Ils sont mémorisés sous forme de chaînes d'octets en mémoire RAM. S'ils contiennent plusieurs bitmaps, ceux-ci sont généralement stockés les uns après les autres.

Voici par exemple comment pourrait être mémorisé un des *sprites* de Zub en RAM (assembleur WinAPE).

```
ZUB_SPR1        DB &00,&21,&80,&07,&08,&00
                DB &10,&F0,&08,&00,&00,&04
                DB &60,&30,&D2,&B4,&70,&0C
                DB &00,&00,&70,&10,&C0,&00
                DB &00,&5A,&4B,&08,&06,&00
                DB &30,&96,&84,&00,&00,&08
                DB &03,&10,&D2,&78,&00,&0C
                DB &00,&00,&60,&10,&80,&00
                DB &01,&A5,&86,&00,&01,&00
                DB &70,&70,&84,&00,&00,&08
                DB &07,&10,&E0,&70,&00,&00
                DB &00,&00,&C0,&30,&00,&00
                DB &10,&48,&04,&00,&00,&08
                DB &E0,&70,&C2,&00,&01,&00
                DB &06,&00,&E0,&30,&80,&00
                DB &00,&00,&C0,&30,&00,&00
                DB &21,&80,&04,&00,&00,&08
                DB &C0,&70,&E1,&08,&06,&00
                DB &06,&00,&70,&30,&80,&00
                DB &00,&10,&80,&60,&00,&00
                DB &12,&08,&08,&00,&00,&04
                DB &C0,&70,&F0,&87,&48,&00
                DB &02,&00,&70,&10,&80,&00
                DB &00,&01,&08,&06,&00,&00
                DB &21,&80,&08,&33,&33,&04
                DB &E0,&30,&F0,&D2,&80,&0F
                DB &00,&00,&30,&00,&C0,&00
                DB &00,&01,&0E,&07,&08,&00
                DB &10,&78,&08,&33,&33,&04
                DB &60,&30,&B4,&D2,&B0,&0F
                DB &00,&00,&30,&00,&C0,&00
                DB &00,&03,&0F,&0B,&0C,&00
```



Les valeurs hexadécimales dépendent de l'ordre des couleurs et du mode utilisé. Le nombre d'octets peut monter très vite, surtout s'il y a de nombreux bitmaps. Et la compression n'est recommandée que pour le chargement en mémoire. Ces *sprites* nécessitent généralement les fonctions suivantes :

- Mémorisation du fond d'écran derrière le *sprite* (sauf s'il est unicolore).
- Effacement du *sprite* (restitution du fond).
- Affichage du *sprite* à une position donnée avec un bitmap donné.
- Gestion des mouvements et de l'animation du *sprite*.
- Détection de collision.

LES SPRITES CODÉS / SPRITES AUTO GÉNÉRÉS

L'idée est ici de réduire le nombre de cycles machines au strict minimum pour l'affichage des *sprites*. Nous verrons comment dans la partie programmation, mais il y a plusieurs combines :

- Remplacer la recopie d'un *sprite* vers l'écran par le chargement direct des octets à l'écran.
- Linéariser les boucles.

• Eviter les demi-octets transparents pour ne charger que des octets pleins.
• Précharger les valeurs d'octets répétitives dans des registres pour accélérer les chargements à l'écran.

Il y a cependant des inconvénients qui font qu'il est difficile de généraliser cette méthode :

Les *sprites* deviennent des routines, toute modification du *sprite* entraîne une modification de code.
Les *sprites* codés prennent largement plus de mémoire que les *sprites* normaux.
Le code d'un *sprite* codé n'est pas réutilisable par un autre *sprite*.

Cette méthode est aussi très efficace pour la construction de décors en tuiles nécessitant un scrolling rapide ("Gauntlet" 1985 © U.S.Gold, "Titan" 1988 © Titus...). Elle convient également très bien aux jeux avec beaucoup de *sprites* rapides ("R-Type" 1988 © Activision, "Gauntlet" 1985 © U.S.Gold...).

Dans "Asphalt" 1987 © UBI Soft, le camion pourrait avoir été créé en sprite codé, avec quelques instructions supplémentaires pour le canon.

Les lignes horizontales du camion permettent en plus d'avoir des octets répétitifs, ce qui est idéal à leur préchargement en registres.

LES SPRITES « HARD » DES CPC 464+, CPC 6128+ ET CONSOLE GX4000

Dès le départ les développeurs avaient compris que les microprocesseurs ne pourraient jamais répondre aux nouveaux besoins en matière de vidéo. En 1982, le Commodore C64 intégrait déjà un processeur graphique VIC-II (Video Interface Chip II) pouvant assurer la gestion de 8 *sprites* « hard », non pas gérés en RAM mais directement par la puce graphique. Ce n'est qu'en 1990 qu'Amstrad tente de rattraper son retard en incorporant des *sprites* hard dans ses CPC+. Mais il est trop tard et peu de jeux sortent, comme par exemple l'excellent "Prehistorik II - Back To Hungerland" 1993 © Titus.

L'ASIC

Le CPC+ intègre un nouveau circuit électronique pour la vidéo : l'ASIC (un circuit électronique conçu pour un besoin spécifique). Il remplace les circuits CPC suivants :

• Le « Gate-Array » (circuit de gestion des bancs RAM et ROM, des modes graphiques et des couleurs).
• Le PAL (qui gère l'affichage sur le tube cathodique)
• Le CRTC 6845 (le contrôleur vidéo gère l'écran point de vue forme et envoie les signaux HBL et VBL).
• Le PPI 8255 (l'interface entrées / sorties).

Particularités des *sprites* hard

• Il y en a 16. Techniquement parlant ils n'ont pas de numéro (même si nous avons tendance à dire qu'ils vont du 0 au 15) mais juste des adresses d'accès, tous stockés dans l'ASIC.
• Le *sprite* 0 est le *sprite* le plus devant, et le 15 le plus derrière. C'est-à-dire que si deux *sprites* se croisent, c'est celui qui a le plus petit nombre qui est affiché devant l'autre à l'écran (A noter qu'il existe un bug permettant d'inverser les priorités).
• Ils font 16x16 pixels dans la résolution haute (mode 2, 640x200), mais ils peuvent être zoomés.
• Chaque pixel occupe un octet, donc les *sprites* occupent 256 octets de mémoire.
• Ils disposent d'une palette à part de 15 couleurs parmi les 4096 couleurs de l'ASIC, plus la couleur 0 réservée à la transparence. Cette palette est indépendante de celle de l'écran, il est donc possible d'avoir 31 couleurs à l'écran (et plus avec les rasters) ; 32 couleurs si nous comptons la couleur du Border et que celle-ci est différente des couleurs déjà utilisées.
• Ils ont leur propre mode vidéo. Ainsi il est possible d'afficher des *sprites* en mode 0 sur un décor en mode 1.

Nous aborderons les Sprites Hard dans un prochain numéro de Côté Gamers.

PROGRAMMATION DES *SPRITES* TEXTES EN BASIC

Les *sprites* textes ont peu d'intérêt par rapport aux *sprites* logiciels ou aux *sprites* hard. Mais ils sont vraiment simples à utiliser pour qui souhaite se lancer dans la création de petits jeux en BASIC. Le CPC propose par défaut une table ASCII de 256 caractères, dont certains sont déjà conçus pour les jeux. On trouve des petits personnages, des fonds et des murs pour la construction de décors, etc.

L'affichage de ces caractères peut se faire en cordonnées textes, grâce à l'instruction LOCATE x,y (avec x=1 à 40 et y=1 à 25).

Exemple : affichage d'un personnage, le caractère 249, aux coordonnées texte 20,12.

Pour l'effacer avant de le déplacer, il suffit d'afficher un espace aux mêmes coordonnées.

Il peut aussi se faire en coordonnées graphiques avec les instructions MOVE x,y (x=0 à 639 et y=0 à 399) et TAG. Mais il faut bien penser au point-virgule après le PRINT, sinon le BASIC affiche aussi les caractères de retour chariot (13 et 10).

Allez savoir pourquoi on peut aller jusqu'à 639,399, car la résolution en mode 1 reste de 320x200. Peut-être était-ce pour impressionner les débutants.

Ces caractères peuvent être momentanément redéfinis à l'aide de la commande BASIC SYMBOL.

```
list
10 SYMBOL 249,3,5,10,148,104,112,240,200
20 PRINT:PRINT CHR$(249):PRINT
Ready
run

Ready
```

128	64	32	16	8	4	2	1	
						■	■	%00000011 = 3
					■		■	%00000101 = 5
				■		■		%00001010 = 10
■			■		■			%00010100 = 148
	■	■		■				%01101000 = 104
	■	■	■					%01110000 = 112
■	■	■	■					%11110000 = 240
■	■			■				%11001000 = 200

En modifiant deux caractères et en les assemblant, il est possible d'obtenir un *sprite* plus gros.

```
list
10 SYMBOL 249,60,60,126,52,62,60,24,60
20 SYMBOL 250,126,126,247,251,60,118,110
,119
30 x=20:y=12
40 LOCATE x,y:PRINT CHR$(249)
50 LOCATE x,y+1:PRINT CHR$(250)
Ready
run

Ready
```

0	0	1	1	1	1	0	0	60
0	0	1	1	1	1	0	0	60
0	1	1	1	1	1	1	0	126
0	0	1	1	0	1	0	0	52
0	0	1	1	1	1	1	0	62
0	0	1	1	1	1	0	0	60
0	0	0	1	1	0	0	0	24
0	0	1	1	1	1	0	0	60
0	1	1	1	1	1	1	0	126
0	1	1	1	1	1	1	0	126
1	1	1	1	0	1	1	1	247
1	1	1	1	1	0	1	1	251
0	0	1	1	1	1	0	0	60
0	1	1	1	0	1	1	0	118
0	1	1	0	1	1	1	0	110
0	1	1	1	0	1	1	1	119

Par défaut, il n'est possible de modifier que les caractères 240 à 255. Pour étendre le nombre de caractères modifiables, il faut utiliser la commande SYMBOL AFTER :

SYMBOL AFTER : permet de redéfinir tous les caractères, y compris les chiffres, les lettres, le retour chariot…
SYMBOL AFTER 128 : permet de redéfinir les caractères 128 à 255.

PROGRAMMATION DES *SPRITES* LOGICIELS CONVENTIONNELS

INTRODUCTION

La programmation des *sprites* logiciels sur CPC n'est pas vraiment simple. Tout d'abord il est nécessaire de maîtriser l'assembleur Z80, car le BASIC n'est pas du tout à la hauteur (bien trop lent). Ensuite il faut comprendre comment fonctionne la vidéo sur CPC (les synchros verticales, les adresses écran, le codage des octets de couleur,…). Et quand il faut ajouter une gestion de la transparence et des collisions, on comprend vite que faire un jeu rapide ne sera pas une mince affaire.

CONVERSION (X,Y) => ADRESSE ÉCRAN

La mémoire vidéo par défaut est le bloc de RAM &C000-&FFFF. Mais seulement 16000 octets sur 16384 sont visibles (uniquement si on ne joue pas avec les Registres du CRTC).

Les 384 octets non visibles peuvent servir à stocker des routines ou des variables (nous vous avions déjà donné les 8 adresses dans côté gamers n° 4).

Cette mémoire n'est pas linéaire, les lignes écran ne se suivent pas comme dans un livre. Par exemple, l'octet qui suit le dernier octet de la ligne 1 (&C04F) est le premier octet de la ligne 9 (&C050). Pour passer de la ligne 1 à la ligne 2 il faut ajouter &800, mais pour passer de la ligne 8 à la ligne 9 il faut ajouter &C850 (&F800 + &C850 = &C050 + retenue).

Si le BASIC se charge des conversions, en assembleur il faut tout faire soi-même... Nous verrons qu'il est préférable de travailler en adresses malgré tout. Mais certains déplacements compliqués peuvent nécessiter de rester en X,Y.

	Octet 0	Octet 1	...	Octet 79
Ligne 1	&C000	&C001	...	&C04F
Ligne 2	&C800	&C801	...	&C84F
Ligne 3	&D000	&D001	...	&D04F
Ligne 4	&D800	&D801	...	&D84F
Ligne 5	&E000	&E001	...	&E04F
Ligne 6	&E800	&E801	...	&E84F
Ligne 7	&F000	&F001	...	&F04F
Ligne 8	&F800	&F801	...	&F84F
Ligne 9	&C050	&C051	...	&C09F
Ligne 10	&C850	&C851	...	&C89F
...	...	...	...	...
Ligne 200	&FF80	&FF81	...	&FFCF

Pour trouver rapidement l'adresse écran d'une coordonnée X,Y, le mieux est de passer par une table contenant les adresses de chaque début de ligne. Exemple en assembleur avec l'émulateur Winape :

```
; Table des adresses des lignes de l'ecran 400 octets
LIGNES   DW &C000,&C800,&D000,&D800,&E000,&E800,&F000,&F800 ; 0-7
         DW &C050,&C850,&D050,&D850,&E050,&E850,&F050,&F850 ; 8-15
         DW &C0A0,&C8A0,&D0A0,&D8A0,&E0A0,&E8A0,&F0A0,&F8A0 ; 16-23
         DW &C0F0,&C8F0,&D0F0,&D8F0,&E0F0,&E8F0,&F0F0,&F8F0 ; 24-31
         DW &C140,&C940,&D140,&D940,&E140,&E940,&F140,&F940 ; 32-39
         DW &C190,&C990,&D190,&D990,&E190,&E990,&F190,&F990 ; 40-47
         DW &C1E0,&C9E0,&D1E0,&D9E0,&E1E0,&E9E0,&F1E0,&F9E0 ; 48-55
         DW &C230,&CA30,&D230,&DA30,&E230,&EA30,&F230,&FA30 ; 56-63
         DW &C280,&CA80,&D280,&DA80,&E280,&EA80,&F280,&FA80 ; 64-71
         DW &C2D0,&CAD0,&D2D0,&DAD0,&E2D0,&EAD0,&F2D0,&FAD0 ; 72-79
         DW &C320,&CB20,&D320,&DB20,&E320,&EB20,&F320,&FB20 ; 80-87
         DW &C370,&CB70,&D370,&DB70,&E370,&EB70,&F370,&FB70 ; 88-95
         DW &C3C0,&CBC0,&D3C0,&DBC0,&E3C0,&EBC0,&F3C0,&FBC0 ; 96-103
         DW &C410,&CC10,&D410,&DC10,&E410,&EC10,&F410,&FC10 ; 104-111
         DW &C460,&CC60,&D460,&DC60,&E460,&EC60,&F460,&FC60 ; 112-119
         DW &C4B0,&CCB0,&D4B0,&DCB0,&E4B0,&ECB0,&F4B0,&FCB0 ; 120-127
         DW &C500,&CD00,&D500,&DD00,&E500,&ED00,&F500,&FD00 ; 128-135
         DW &C550,&CD50,&D550,&DD50,&E550,&ED50,&F550,&FD50 ; 136-143
         DW &C5A0,&CDA0,&D5A0,&DDA0,&E5A0,&EDA0,&F5A0,&FDA0 ; 144-151
         DW &C5F0,&CDF0,&D5F0,&DDF0,&E5F0,&EDF0,&F5F0,&FDF0 ; 152-159
         DW &C640,&CE40,&D640,&DE40,&E640,&EE40,&F640,&FE40 ; 160-167
         DW &C690,&CE90,&D690,&DE90,&E690,&EE90,&F690,&FE90 ; 168-175
         DW &C6E0,&CEE0,&D6E0,&DEE0,&E6E0,&EEE0,&F6E0,&FEE0 ; 176-183
         DW &C730,&CF30,&D730,&DF30,&E730,&EF30,&F730,&FF30 ; 184-191
         DW &C780,&CF80,&D780,&DF80,&E780,&EF80,&F780,&FF80 ; 192-199
```

Pour trouver l'adresse écran correspondant aux coordonnées (X,Y) d'un *sprite*, il suffit ensuite d'appliquer la formule suivante :

Adresse = (Adresse de LIGNE + 2 x (YSPRITE)) + Division entière [(XSPRITE) / (PIX_MODE)]

Avec :

- () = contenu de la variable ou de l'adresse mémoire.
- Adresse de Ligne = l'adresse à laquelle est stockée le tableau LIGNES ci-dessus.
- (YSPRITE) = coordonnée Y du *sprite* sur 16 bits.
- (XSPRITE) = coordonnée X du *sprite* sur 16 bits.
- (PIX_MODE) = nombre de pixels par octet (2 pour mode 0, 4 pour mode 1, 8 pour mode 2).

(YSPRITE) est multiplié par 2, car dans le tableau LIGNES les adresses sont sur 2 octets. Donc si Y=1, l'adresse se trouve à LIGNE + 2 octets. Si Y=2, l'adresse se trouve à LIGNE + 4 octets,…

L'octet en X est la division entière de (XSPRITE) par le nombre de pixels du mode (PIX_MODE).
Exemples :

- (XSPRITE) = 3 en mode 1 => Octet = division entière (3 / 4) = 0 (comprendre octet n°0 de la ligne).
- (XSPRITE) = 5 en mode 1 => Octet = division entière (5 / 4) = 1
- (XSPRITE) = 5 en mode 0 => Octet = division entière (5 / 2) = 2

Exemple de code assembleur pour trouver l'adresse écran d'un *sprite* en mode 1 (assembleur Winape) :

```
XSPRITE  DW 160
YSPRITE  DW 100
ADSPRITE DW 0

; Conversion X,Y => adresse ecran
; ADSPRITE = (LIGNES + 2 x (YSPRITE)) + (XSPRITE) / 4
XY_ADR   LD HL,LIGNES           ; HL = adresse du tableau LIGNES
         LD DE,(YSPRITE)        ; DE = coordonnee Y du sprite
         ADD HL,DE
         ADD HL,DE              ; HL = LIGNES + 2 x (YSPRITE)
         LD E,(HL)
         INC HL
         LD D,(HL)              ; DE = (LIGNES + 2 x (YSPRITE))
         LD HL,(XSPRITE)
         SRL H
         RR L
         SRL H
         RR L                   ; HL = (XSPRITE)/4 = octet de la coordonnee X
         ADD HL,DE              ; HL = adresse ecran du sprite
         LD (ADSPRITE),HL       ; memorisation de cette adresse
         RET
```

En remplaçant XSPRITE par l'octet X directement, il est possible de gagner des cycles. Un déplacement horizontal d'un octet revient alors à un déplacement de 4 pixels de mode 1. Mais cette formule a toute son utilité pour des déplacements en X au pixel près.

En BASIC lorsque vous faites un **LOCATE colonne, ligne** vous déplacez le curseur texte à une nouvelle adresse écran.
Imaginons que nous souhaitions connaître l'adresse écran du coin haut gauche de cette position.

```
10 MODE 1                                  Le mode
20 DEF FN pix(x,y)=&C000+(y-1)*80+(x-1)*2  La définition de la fonction de calcul par rapport au  MODE 1
30 x=40:y=25                               x=la colonne (1 a 40) ; y = la ligne (1 a 25)
40 adr=FN pix(x,y)                         stockage dans adr du calcul de l'adresse écran
50 PRINT HEX$(adr)                         Affichage de la valeur hexadécimales : &C7CE
```

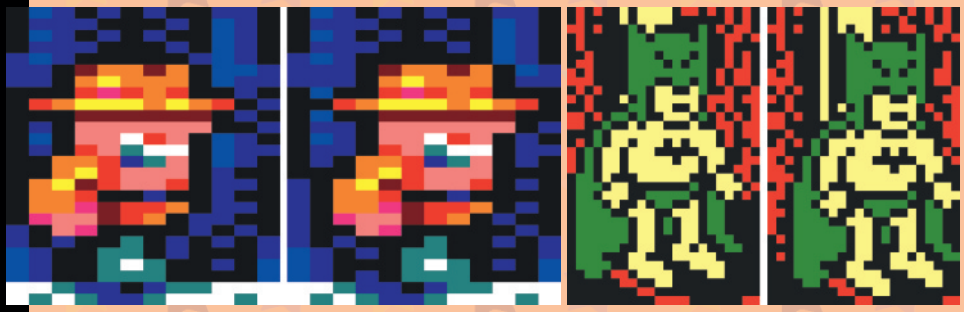
En MODE 0 :
X pourra prendre une valeur entre 1 et 20 (et oui, seulement 20 colonnes), attention notre fonction doit être modifiée
DEF FN pix(x,y)=&C000+(y-1)*80+(x-1)***4**

Pour le mode 2 nous avons 80 colonnes et la fonction devient : DEF FN pix(x,y)=&C000+(y-1)*80+(x-1)

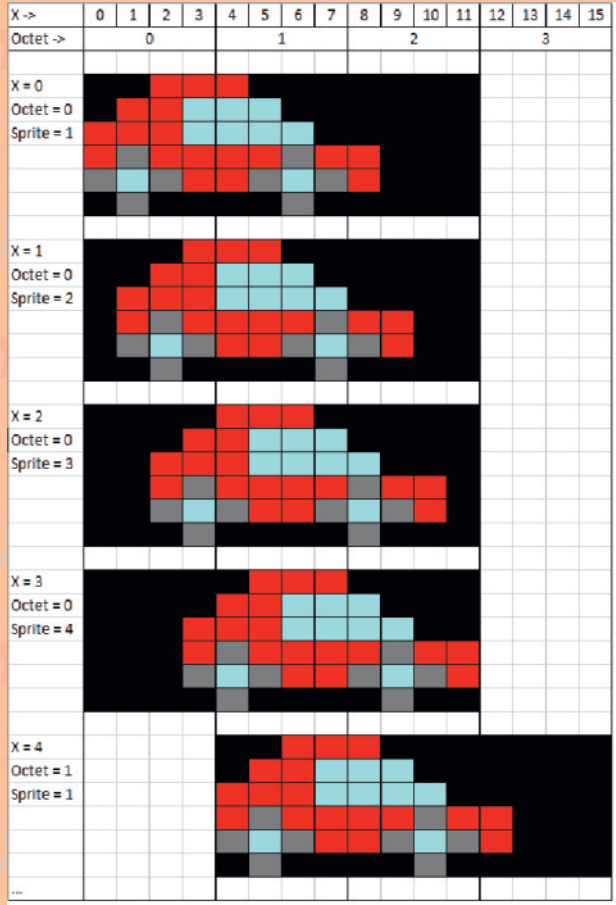
UN POSITIONNEMENT HORIZONTAL AU PIXEL PRÈS ?

Les amateurs de jeux de plateforme se souviendront des heures passées à essayer de positionner un personnage au pixel près, pour pouvoir atteindre une plateforme au-dessus. Comme par exemple dans "Cauldron 2" 1986 © Palace Software, avec cette maudite citrouille qui rebondissait partout sauf là où on le voulait. On notera l'exploit de J34n qui nous a fourni le SNR (enregistrement de sa partie sans tricher sous WinApe) nous montrant qu'il est possible de finir "Cauldron 2" sans perdre aucune vie et ça en seulement 11 minutes et 38 secondes. Un véritable exploit, j'en suis encore baba.

Finalement peu de développeurs ont positionné leurs *sprites* horizontalement au pixel près. Généralement ils étaient positionnés à l'octet près, soient à 2 pixels de mode 0 près, ou à 4 pixels de mode 1 près, ou à 8 pixels de mode 2 près. Malgré tout, on trouve quelques jeux avec une précision inférieure à l'octet : "Rick Dangerous" 1989 © Firebird (1 pixel de mode 0, soit ½ octet) ou "Batman" 1986 © Ocean Software (2 pixels de mode 1, soit ½ octet également).



Mais alors comment est-il possible de positionner un *sprite* avec une précision inférieure à l'octet ? Et bien en prévoyant plusieurs *sprites*. Si vous voulez une précision en X de 1 pixel de mode 1 par exemple, il vous faut 4 *sprites*.




```
XSPRITE  DW 163

; Selection du sprite de la voiture en fonction de XSPRITE
; HL pointe le sprite correspondant
XVOITURE LD A,(XSPRITE)          ; A = poids faible de XSPRITE
         AND %00000011           ; A = position dans l'octet de X (0,1,2 ou 3)
         RLCA
         LD B,A
         RLCA
         RLCA
         RLCA
         ADD B                   ; A <= 18A (18 octets par sprite de voiture)
         LD C,A
         LD B,0
         LD HL,VOITURE1          ; HL pointe le sprite de la voiture
         ADD HL,BC               ; correspondant a la position 0,1,2 ou 3
         RET

; Sprite de la voiture en fonction des position 0,1,2 ou 3
VOITURE1 DB &03,&08,&00
         DB &17,&CC,&00
         DB &1F,&EE,&00
         DB &4B,&2D,&08
         DB &E5,&7A,&08
         DB &40,&20,&00

VOITURE2 DB &01,&0C,&00
         DB &03,&EE,&00
         DB &07,&FF,&00
         DB &25,&1E,&0C
         DB &72,&3D,&84
         DB &20,&10,&00

VOITURE3 DB &00,&0E,&00
         DB &01,&7F,&00
         DB &03,&7F,&88
         DB &12,&0F,&86
         DB &31,&96,&CA
         DB &10,&00,&80

VOITURE4 DB &00,&07,&00
         DB &00,&3F,&88
         DB &01,&3F,&CC
         DB &01,&87,&4B
         DB &10,&CB,&E5
         DB &00,&80,&40
```

La routine XVOITURE isole la position dans l'octet de la coordonnée X. XSPRITE = 163 donnera l'octet 40 (division entière de 160/4) et la position 3. On est dans le cas de figure X=3, *sprite* = 4 ci-dessus. La position est ensuite multipliée par 18, le nombre d'octets de chaque *sprite*, puis ajoutée à l'adresse de VOITURE1.

HL pointera donc :
- VOITURE1 + 0 = VOITURE1 pour la position 0.
- VOITURE1 + 18 = VOITURE2 pour la position 1.
- VOITURE1 + 36 = VOITURE3 pour la position 2.
- VOITURE1 + 52 = VOITURE4 pour la position 3.

XVOITURE = 163 => HL pointe VOITURE4

AFFICHAGE D'UN SPRITE ENTIER SANS TRANSPARENCE

Une fois l'adresse écran du *sprite* obtenue, on va enfin pouvoir travailler en adresses uniquement. Le code d'affichage d'un *sprite* entier sans transparence est généralement de ce genre, quel que soit le mode :

```
;-----------------------------------------------------------------------------
; Affichage d'un sprite entier sans transparence
; Entrees ;
; - LIGNES (DB) = nombre de lignes du sprite.
; - LARGEUR (DW) = largeur du sprite en octets.
; - HL = Adresse de la table du sprite.
; - DE = Adresse ecran du sprite.
;-----------------------------------------------------------------------------
AFF_SPRITE      LD A,(LIGNES)
AFF_SPRITE1     PUSH DE
                LD BC,(LARGEUR)         ; BC = nombre d'octet a copier par ligne
                LDIR                    ; Copie d'une ligne
                POP DE
                PUSH HL
                LD HL,&800
                ADD HL,DE
                JR NC,AFF_SPRITE2
                LD BC,&C050
                ADD HL,BC
AFF_SPRITE2     EX HL,DE                ; DE = ligne suivante a l'ecran
                POP HL
                DEC A
                JR NZ,AFF_SPRITE1
                RET
```

LDIR permet de copier un bloc de (BC) octets, d'une adresse (HL) vers une adresse (DE). Chaque ligne du *sprite* est donc copiée d'un bloc de la table vers l'écran. à noter que HL et DE sont incrémentés après chaque copie d'octet.

Pour le passage à la ligne suivante, il y a une subtilité liée à la non linéarité de l'écran. On ajoute d'abord &800 à DE (seule l'instruction ADD HL,DE le permet), et si ça dépasse &FFFF (carry = 1), alors on ajoute &C050 de plus. On a donc bien +&800 ou +&C850 selon la ligne où on se trouve.

Avant l'addition, DE est replacé en chaque début de ligne de *sprite* (push pop DE). Enfin, il suffit de reboucler tant que toutes les lignes ne sont pas lues.

S'il n'y a pas trop de variations dans la taille des *sprites*. On peut prévoir des routines à largeur fixe, ce qui permet de gagner un nombre de cycles non négligeables. Exemple, avec des *sprites* de 5 octets de large.

```
AFF_SPRITE      LD A,(LIGNES)
AFF_SPRITE1     LDI                     ; Copie d'une ligne de 5 octets
                LDI
                LDI
                LDI
                LDI
                PUSH HL
                LD HL,&800-5
                ADD HL,DE
                JR NC,AFF_SPRITE2
                LD BC,&C050
                ADD HL,BC
AFF_SPRITE2     EX HL,DE                ; DE = ligne suivante a l'ecran
                POP HL
                DEC A
                JR NZ,AFF_SPRITE1
                RET
```

LDI ne copie qu'un octet à la fois, mais économise 1 cycle par rapport à chaque octet copié par LDIR. On économise ainsi un LD BC,(LARGEUR), 1 cycle par copie d'octet et un push/pop de DE puisqu'il suffit d'ajouter &800-5 pour la ligne suivante. Soient 17 cycles économisés par ligne de 5 octets !

COUPURE DES SPRITES PAR LES BORDURES DE L'ÉCRAN

Il arrive que les *sprites* sortent de l'écran. Il y a plusieurs façons de gérer cette sortie :

- Effacer le *sprite* dès qu'il arrive au bord de l'écran.
- Réduire la fenêtre d'affichage pour que le *sprite* soit automatiquement coupé quand il sort de cette fenêtre.
- Afficher le *sprite* partiellement en coupant ce qui dépasse.

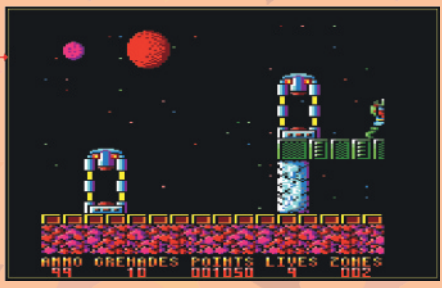
La première solution ne pose pas vraiment de problème. Il faut juste prendre en compte la largeur du *sprite* s'il sort à droite, et la hauteur s'il sort en bas. C'est le choix fait par "Buggy II" 1986 © Chip, dans lequel les bosses et les tonneaux sont effacés s'ils mordent sur les bords gauche ou droite.

Soit OCTET = XSPRITE / 2 en mode 0 (2 pixels par octet), avec OCTET variant de 0 à 79 :

Effacer le sprite si :
- OCTET – 1 < 0
- OCTET + Largeur de sprite en octets + 1 > 79

La deuxième solution a été utilisée par de nombreux jeux d'action (ex : "Gryzor 1987" © Ocean Software, "Exolon" 1987 © Hewson...). Réduire la fenêtre d'affichage permet de continuer à afficher des *sprites* entièrement jusqu'aux bords de l'écran. C'est la fenêtre d'affichage qui les coupe automatiquement. L'utilisation de routines rapides sans gestion de coupure permet de gagner de précieux cycles.

Nous ne nous attarderons pas sur la réduction de fenêtre dans ce dossier car ce n'est pas le sujet. Pour cela voyez plutôt du côté du CRTC (cf Côté Gamers n°3).



Limite normale de l'écran.

Certains jeux ont choisi la troisième solution, comme par exemple "Asphalt" 1987 © UBI Soft. L'écran est alors de taille normale et les sprites des voitures et des motos sont partiellement dessinés quand ils débordent à gauche ou à droite.

Dans ce cas, il faut adapter la routine d'affichage précédente pour prendre en compte la largeur à afficher.

```
;-----------------------------------------------------------------------------
; Affichage d'un sprite entier sans transparence
; Entrees ;
; - LIGNES (DB) = nombre de lignes a afficher.
; - LARGEUR (DW) = largeur du sprite en octets.
; - NBOCTETS (DW) = nombres d'octets a afficher par ligne.
; - HL = Adresse de la table du sprite.
; - DE = Adresse ecran du sprite.
;-----------------------------------------------------------------------------
AFF_SPRITE      LD A,(LIGNES)
AFF_SPRITE1     PUSH HL
                PUSH DE
                LD BC,(NBOCTETS)
                LDIR
                POP DE
                LD HL,&800
                ADD HL,DE
                JR NC,AFF_SPRITE2
                LD BC,&C050
                ADD HL,BC
AFF_SPRITE2     EX HL,DE                ; DE = ligne d'ecran suivante
                POP HL
                LD BC,(LARGEUR)
                ADD HL,BC               ; HL = ligne de sprite suivante
                DEC A
                JR NZ,AFF_SPRITE1
                RET
```

NBOCTETS est le nombre d'octets à afficher par lignes de *sprite*. Le passage à la ligne écran suivante ne change pas. Par contre on ajoute un passage à la ligne suivante dans le *sprite* (HL + LARGEUR). HL pointera donc toujours le même rang d'octet à chaque ligne (le 1er de chaque ligne, ou le 2ème de chaque ligne ...).

Cette routine s'adapte à tous les cas de figure :

1- Si le *sprite* est coupé par le bas de l'écran, il suffit de réduire (LIGNES) pour n'afficher que les premières lignes visibles du *sprite*.

2- Si le *sprite* est coupé par le haut de l'écran, il faut réduire (LIGNES) et faire pointer HL sur la partie du *sprite* qui reste visible. Exemple, si les 3 premières lignes du *sprites* sont coupées, il suffit de faire pointer HL sur la quatrième ligne du *sprite*.

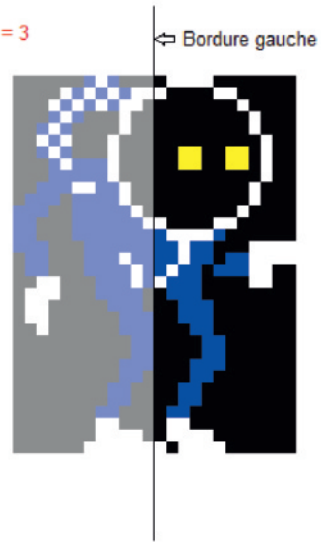
3- Si le *sprite* est coupé par la droite de l'écran, il faut ajuster (NBOCTETS) pour n'afficher que les premiers octets visibles sur chaque ligne.

4- Si le *sprite* est coupé par la gauche de l'écran, il faut ajuster (NBOCTETS) et faire pointer HL sur le premier octet visible de la première ligne. Exemple, si les 3 premiers octets de chaque ligne sont coupés, il suffit de faire pointer HL sur le quatrième octet de la première ligne.

5- Si le *sprite* est coupé dans un coin, il suffit de combiner deux des solutions ci-dessus.

Exemple : en imaginant Zub coupé par la bordure gauche de l'écran, ça donnerait ceci.

```
ZUB_SPR1    DB &00,&21,&80,&07,&08,&00   HL pointe le 4ème octet
            DB &10,&F0,&08,&00,&00,&04
            DB &60,&30,&D2,&B4,&70,&0C   LARGEUR = 6
            DB &00,&00,&70,&10,&C0,&00
            DB &00,&5A,&4B,&08,&06,&00   NBOCTET = 3
            DB &30,&96,&84,&00,&00,&08
            DB &03,&10,&D2,&78,&00,&0C
            DB &00,&00,&60,&10,&80,&00
            DB &01,&A5,&86,&00,&01,&00
            DB &70,&70,&84,&00,&00,&08
            DB &07,&10,&E0,&70,&00,&00
            DB &00,&00,&C0,&30,&00,&00
            DB &10,&48,&04,&00,&00,&08
            DB &E0,&70,&C2,&00,&01,&00
            DB &06,&00,&E0,&30,&80,&00
            DB &00,&00,&C0,&30,&00,&00
            DB &21,&80,&04,&00,&00,&08
            DB &C0,&70,&E1,&08,&06,&00
            DB &06,&00,&70,&30,&80,&00
            DB &00,&10,&80,&60,&00,&00
            DB &12,&08,&08,&00,&00,&04
            DB &C0,&70,&F0,&87,&48,&00
            DB &02,&00,&70,&10,&80,&00
            DB &00,&01,&08,&06,&00,&00
            DB &21,&80,&08,&33,&33,&04
            DB &E0,&30,&F0,&D2,&80,&0F
            DB &00,&00,&30,&00,&C0,&00
            DB &00,&01,&0E,&07,&08,&00
            DB &10,&78,&08,&33,&33,&04
            DB &60,&30,&B4,&D2,&B0,&0F
            DB &00,&00,&30,&00,&C0,&00
            DB &00,&03,&0F,&0B,&0C,&00   LIGNES = 32
```



MASQUAGE DES SPRITES PAR DES ÉLÉMENTS DE DÉCOR

Il s'agit ici de gérer le déplacement d'un *sprite* derrière un élément de décor, ce qui entraîne le masquage du *sprite* par cet élément. Quel que soit le type de jeu (2D ou 3D isométrique), la technique consiste généralement à redessiner les éléments de décors et le *sprite* dans la zone concernée, en les affichant de l'arrière plan au premier plan.

Masquage des *sprites* dans un décor 2D

Dans "Exolon" 1987 © Hewson, le personnage passe régulièrement derrière des pylones ou des télé-porteurs. En y regardant de plus près, le télé-porteur de l'image ci-contre est constitué de tuiles 4x8 pixels (8 octets verticaux).

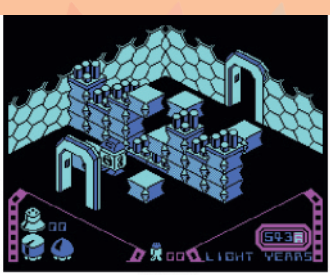
Ces tuiles n'ont pas de transparence. Il est donc facile de les afficher très rapidement avec des LD (nn),r. On peut donc déduire que dans ce cas, le masquage consiste simplement à afficher le personnage puis à rafraîchir les tuiles du télé-porteur dans la zone du *sprite*.

Masquage des *sprites* dans un décor 3D isométrique

Les jeux en 3D isométrique sont un genre particulier. Les plus aboutis sont certainement "Crafton & Xunk / Get Dexter" 1986 © Ere Informatique et "Batman" 1986 © Ocean Software, mais on peut aussi citer "Head Over Heels" 1987 © Ocean Software, "Alien 8" 1985 © Ultimate Play the Game, "Bactron" 1986 © Loriciels, "Knight Lore" 1984 © Ultimate Play the Game,... Il s'agit d'une fausse 3D, construite avec de vraies tuiles 2D ayant des formes de cube vus de dessus.

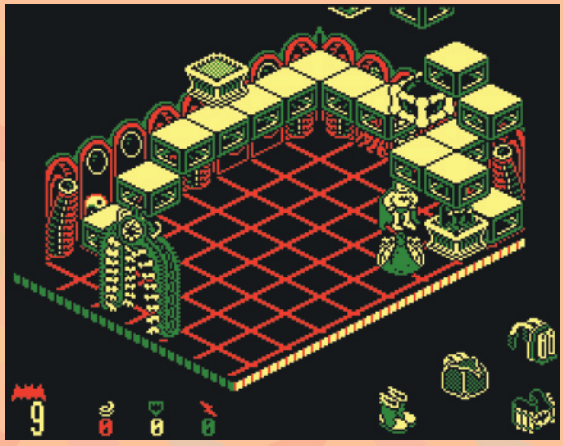
Sur CPC le genre est apparu avec les jeux d'Ultimate Play the Game. Leur technique, appelée « Filmation game engine », a bluffé tout le monde à l'époque, notamment avec "Knight Lore" et "Alien 8". C'était la première fois qu'on pouvait tourner autour d'un objet, dans un décor pseudo 3D. La technique a ensuite été améliorée dans "Gunfright" et "Night Shade" 1985 © Ultimate Play the Game, avec la possibilité d'afficher les murs ou non.


Knight Lore


Alien 8

Gunfright

Dans "Batman" 1986 © Ocean Software, les objets se cachent les uns les autres, et les *sprites* passent aussi bien devant que derrière les cubes en tout genre. La technique est inspirée du « *Filmation game engine* ».

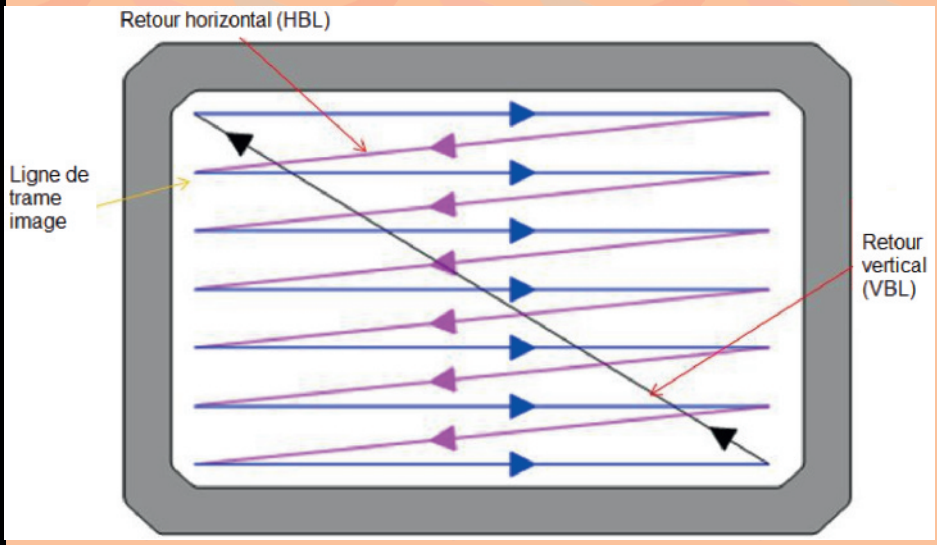


Jon RITMAN, le développeur de Batman, a expliqué dans une interview qu'il avait utilisé des listes chaînées pour retrouver facilement l'ordre des cubes à redessiner dans une zone précise. Les cubes derrière un *sprite* sont identifiés dans la liste et redessinés. Puis le sprite est affiché, et enfin les cubes devant le *sprite* sont identifiés dans la liste et redessinés.

D'ailleurs il explique que quand Batman pousse un objet vers la caméra, c'est plus fluide, car les objets derrière sont déjà calculés et affichés. Dans le cas contraire, il doit parcourir la liste à chaque pas.

LA LIGNE BLANCHE VERTICALE (VBL)

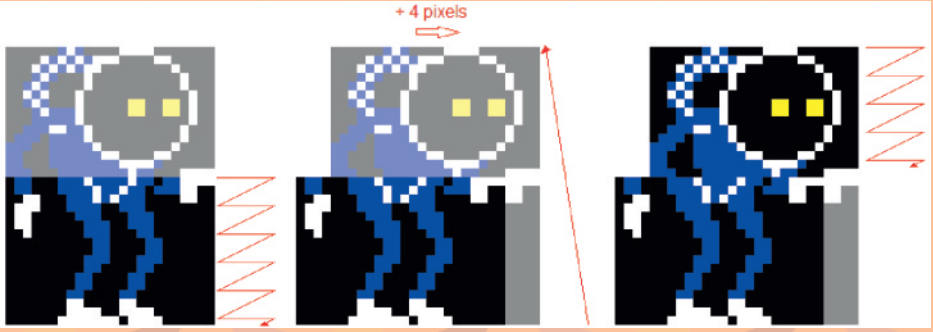
Sur les écrans à tubes cathodiques, les images étaient affichées à l'aide d'un faisceau d'électrons qui balayait l'écran 50 fois par seconde. Il parcourait chaque ligne de trame écran (les lignes affichées), puis revenait à la ligne sans rien afficher d'autre au passage (retour horizontal ou « Horizontal Blank Line » en anglais). Quand il arrivait en bas à droite, il remontait en haut à gauche sans rien afficher à ce moment-là (retour vertical ou « Vertical Blank Line » en anglais).



Le CPC n'a pas prévu la détection des retours horizontaux, ce qui aurait été bien pratique pour les rasters. En revanche, comme sur la plupart des ordinateurs de l'époque, il a prévu la détection du retour vertical. C'est la fameuse VBL comme « Vertical Blank Line » en anglais.

Pourquoi cette VBL est si importante quand on manipule les *sprites* ? Eh bien parce que si vous affichez un *sprite* au moment où le faisceau est déjà à la moitié de celui-ci, puis que vous déplacez le *sprite*, vous obtenez un effet de cisaillement… En effet, l'œil voit d'abord se dessiner la partie basse du *sprite*, puis la partie haute décalée. Et plus le *sprite* se déplace vite, plus l'effet s'amplifie. Ce qui devient très vite désagréable à regarder.

Pour éviter ce genre de désagrément, et avoir des *sprites* bien fluides, il faut les afficher pendant le retour vertical, dès que la VBL est active. Comme ça on est sûr qu'ils seront balayés d'un coup.



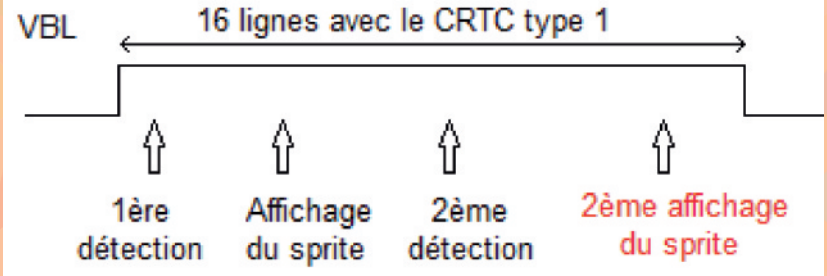
Il y a deux façons de détecter la VBL :

- en appelant le vecteur &BD19 avec un CALL. Celui-ci rend la main quand la VBL s'active.
- en interrogeant le port &F5xx avec une boucle d'attente.

La deuxième permet d'économiser quelques précieux cycles. Et puis elle fait plus classe dans un programme. Pour la VBL, voici une petite précision par rapport aux CRTC de type 1. Ces CRTC ont une Vsync

```
; Attends le debut de la VLB
ATTENDS_VBL      LD B,&F5              ; Port de la VBL
ATTENDS_VBL2     IN A,(C)
                 RRA                   ; VLB active?
                 JR NC,ATTENDS_VBL2    ; Si Non on reboucle
                 RET                   ; Si Oui on sort
```

(Synchronisation Verticale) qui dure 16 lignes. C'est énorme et ça pose des problèmes si nous affichons des *sprites* en moins de 16 lignes. Si on utilise la méthode par Gate-Array, il se peut qu'on détecte deux fois la même VBL, donc que nous ayons un risque d'afficher 2 fois le *sprite* dans la même VBL.



2 solutions :

1 - ajouter 3 HALT, c'est efficace mais ça gaspille du temps.

2 - Attendre que la VBL soit inactive puis active (détection du front montant).

C'est ce qui est utilisé dans la version du programme modifié ci-dessous :

```
WAIT_FOR_VBL      LD B,&F5                  ; Port VBL
                  IN A,(C)
                  RRA
                  JR C,WAIT_FOR_VBL+2       ; Boucler tant que la VBL est active

WAIT_FOR_VBL2     IN A,(C)
                  RRA
                  JR NC,WAIT_FOR_VBL2       ; Boucler tant que la VBL est inactive
                  RET
```

Le dossier sur les sprites vous plait ?

Vous aimeriez en savoir plus ?

Ça tombe bien, il continuera dans le prochain numéro de Côté Gamers.

L'équipe de cpc-power

SHMUP VOL.2 - LA PC-Engine

Côté Gamers 6

Revival Gamers 3

Côté Gamers Horizons

Game Start

EN 2018 chez Côté Gamers !

Projet "LOST IN TIME"

Sprites - Vol Spécial PC-Engine SHMUP