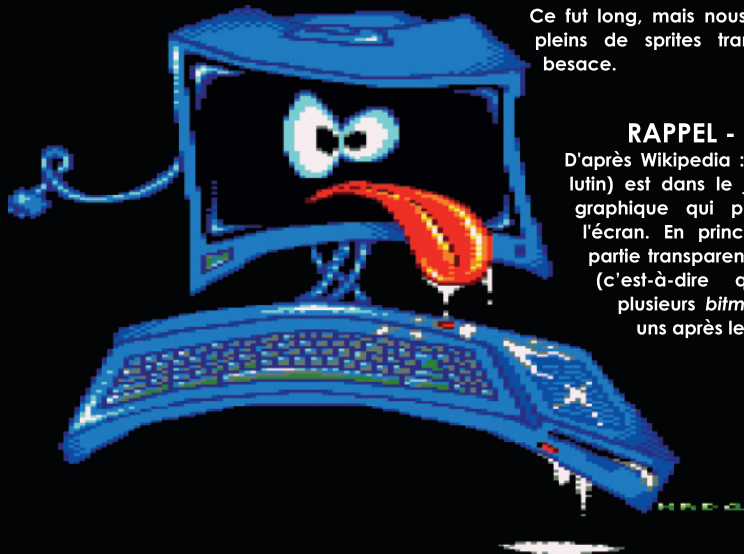


Leçon du jour

La mémoire et les sprites Partie 2

Ce fut long, mais nous voilà de retour avec pleins de sprites transparents dans notre besace.



RAPPEL - 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).

GESTION DE LA TRANSPARENCE

Voici donc un sujet brûlant. Si les pixels de notre CPC avaient tenu sur un octet, ça n'aurait pas vraiment posé de problème. En balayant le sprite, il suffisait de remplacer les octets &00 par l'octet de fond (en admettant que le &00 soit réservé pour la transparence). Mais voilà, l'organisation des octets est telle qu'il faut soit utiliser des comparaisons et des combinaisons logiques, soit trander comme dans "Gauntlet II" 1986 © U.S.Gold.

Pour bien comprendre où se situe le problème avec la gestion de la transparence, il faut revenir à l'encodage des octets de couleur dans la mémoire vidéo (Nous vous en avons parlé dans le premier numéro de Côté Gamers). Nous avons vu que selon le mode 0, 1 ou 2, un octet permet l'affichage de 2, 4 ou 8 pixels. L'ennui, c'est que cet octet est organisé de manière étrange, et pas avec la même logique selon le mode.

MODE 0

Soient deux couleurs $C0 = C1 = \%0100$ (donc couleur n°4 en mode 0) nous aurions pu croire, mais la vérité est ailleurs.

	Bit 3	Bit 2	Bit 1	Bit 0
C0	0	1	0	0
C1	0	1	0	0

Octet de mode 0								
Bits	7	6	5	4	3	2	1	0
Codage	C0 - Bit 0	C1 - Bit 0	C0 - Bit 2	C1 - Bit 2	C0 - Bit 1	C1 - Bit 1	C0 - Bit 3	C1 - Bit 3
Résultat	0	0	1	1	0	0	0	0

Au final, les deux couleurs sont combinées de la manière suivante dans un octet de mode 0 :
Donc quand on combine les deux couleurs $\%0100$, ça donne $\%00110000$.
Ce n'est pas logique mais c'est comme ça.

MODE 1

Soit quatre couleurs $C0 = C1 = C2 = C3 = \%10$ (couleur n°2 en mode 1).

	Bit 1	Bit 0
C0	1	0
C1	1	0
C2	1	0
C3	1	0

Octet de mode 1								
Bits	7	6	5	4	3	2	1	0
Codage	C0 - Bit 0	C1 - Bit 0	C2 - Bit 0	C3 - Bit 0	C0 - Bit 1	C1 - Bit 1	C2 - Bit 1	C3 - Bit 1
Résultat	0	0	0	0	1	1	1	1

Les bits de poids fort des numéros de couleurs sont en poids faible de l'octet, et inversement.
La combinaison des quatre couleurs $\%10$ donne donc $\%00001111$.

MODE 2

Le mode 2 est le plus simple, mais le moins utilisé dans les jeux. Soient 8 couleurs de 1 bit :

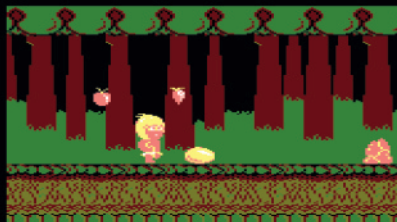
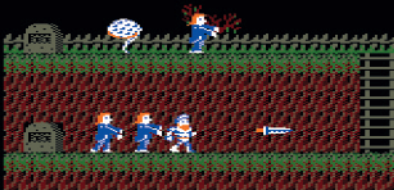
Octet de mode 2								
Bits	7	6	5	4	3	2	1	0
Codage	C0	C1	C2	C3	C4	C5	C6	C7

Dans ce cas, on comprend qu'isoler une couleur, vérifier si elle est égale à 0 et si oui ajouter la couleur du fond, tout cela nécessite quelques opérations logiques ET et OU pour chaque pixel. D'où une grosse perte de cycles en perspective.

SASFÉPU... LA MEMOIRE ET LES SPRITES - PARTIE 2

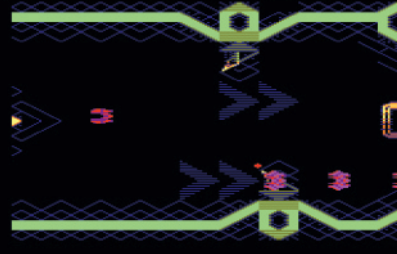
La méthode « Ghosts'n Goblins » (Mot clef "Dual Playfield" sur cpc-power)

Je l'appelle la méthode "Ghosts'N Goblins" © Elite Systems (1986), mais d'autres jeux l'ont utilisée comme "Bomb Jack" © Elite Systems (1986). Sauf que dans le premier cas çà a vraiment dégradé l'adaptation du jeu, limité à 7 couleurs en mode 0 dont 3 pour les sprites. La méthode est reconnaissable au nombre limité de couleurs, et au fait que les sprites sont transparents avec le décor mais pas entre eux.



"Ghosts'N Goblins" 1986 © Elite Systems

"Wonder Boy" 1987 © Activision

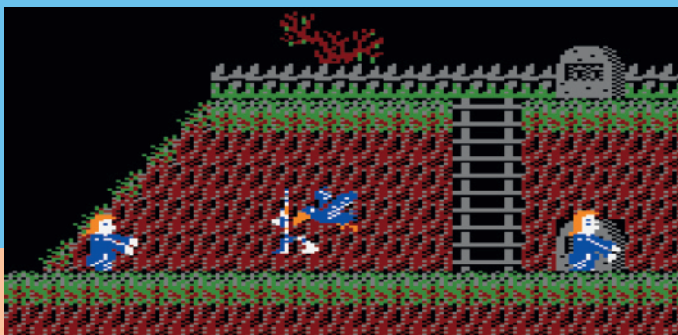


"3D Munchies" 1987 © MicroHobby Amstrad Semanal





"Relentless" 2013 © Psytronik Software





Eh mec, t'es pas transparent !

Dans "Ghosts'N Goblins" 1986 © Elite Systems, les sprites sont transparents avec le décor, mais pas avec les autres sprites. Par exemple un monstre qui passe devant le joueur ou un autre monstre l'efface en partie voire complètement.



L'idée est de séparer les quatre bits d'un pixel en deux, pour mémoriser à la fois la couleur de sprite et la couleur de décor qui se trouve derrière. Du coup chaque couleur est codée sur 2 bits, soient 4 couleurs maximum...

Décor					
Couleur n°	Code Binaire				Couleur
0	0	0	0	0	
4	0	1	0	0	
8	1	0	0	0	
12	1	1	0	0	

Sprites					
Couleur n°	Code Binaire				Couleur
0	0	0	0	0	
1,5,9,13	x	x	0	1	
2,6,10,14	x	x	1	0	
3,7,11,15	x	x	1	1	

n°	Codage de 1 pixel				Commentaire
	Décor	Sprites	Couleur		
0	0	0	0	0	Noir du décor OU Transparence = Noir
1	0	0	0	1	Le bleu du sprite prime sur le fond noir
2	0	0	1	0	Idem pour le blanc du sprite
3	0	0	1	1	Idem pour le orange du sprite
4	0	1	0	0	Gris du décor OU Transparence = Gris
5	0	1	0	1	Le bleu du sprite prime sur le fond gris
6	0	1	1	0	Idem pour le blanc du sprite
7	0	1	1	1	Idem pour l'orange du sprite
8	1	0	0	0	Marron du décor OU Transparence = Marron
9	1	0	0	1	Le bleu du sprite prime sur le fond marron
10	1	0	1	0	Idem pour le blanc du sprite
11	1	0	1	1	Idem pour l'orange du sprite
12	1	1	0	0	Vert du décor OU Transparence = Vert
13	1	1	0	1	Le bleu du sprite prime sur le fond vert
14	1	1	1	0	Idem pour le blanc du sprite
15	1	1	1	1	Idem pour l'orange du sprite

Quelle que soit la couleur du fond, si on ajoute une couleur de sprite différente de %0000, celle-ci s'impose :

- %1100 (pixel de fond vert) OU %0011 (pixel de sprite orange) = %1111 (pixel écran orange).
- %0100 (pixel de fond gris) OU %0001 (pixel de sprite bleu) = %0101 (pixel écran bleu).

En effaçant les deux bits de poids faible ou en imposant un pixel de sprite transparent, la couleur de fond réapparaît :

- %1111 (pixel écran orange) AND %1100 (effacement des 2 bits de poids faible) = %1100 (pixel de fond vert).
- %0101 (pixel écran bleu) AND %1100 = %0100 (pixel de fond gris)

La couleur de sprite %0000 est plus que transparente, elle permet aussi de nettoyer le fond. Car comme le sprite est entouré de pixels transparents, ce sont eux qui restituent le fond au fur et à mesure des déplacements. C'est pour ça que deux sprites ne peuvent pas se superposer. Car celui du dessus restitue le fond même s'il y a déjà un sprite affiché.

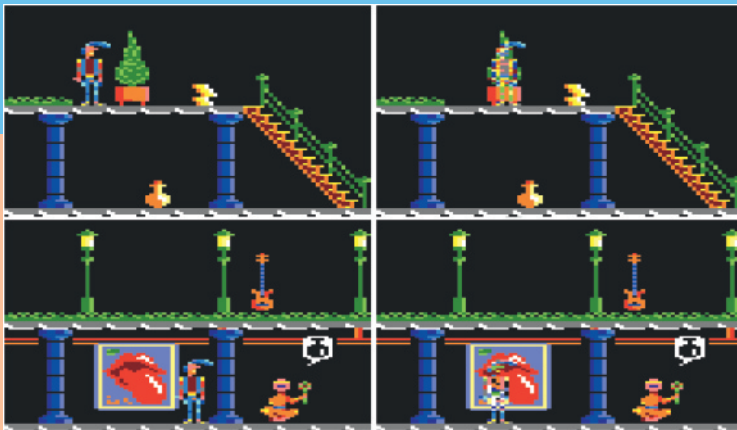
Au final, peu importe l'organisation des bits de couleur dans les octets de la mémoire vidéo, puisque les bits de fond ne se mélangent pas avec les bits des sprites. Voilà comment on gagne de très nombreux cycles au prix d'une dégradation des graphismes.

SASFÉPU... LA MEMOIRE ET LES SPRITES - PARTIE 2

La méthode « Billy la banlieue »

Là encore "Billy La Banlieue" 1986 © Loricels, n'est pas le seul à l'appliquer, sauf qu'ici elle est très visible. "Cauldron II" 1986 © Palace Software l'utilise également, mais la citrouille passe rarement devant des objets, donc on s'en aperçoit moins. Elle se caractérise par le fait que les sprites subissent des modifications de couleur quand ils passent devant des objets (au lieu de cacher ces objets).

Comme "Ghosts'n Goblins", Billy aime la nuit. Mais il aime aussi les couleurs. Et quand il passe devant les obstacles, il en voit de toutes les couleurs ! Cela dit il est TRES rapide ! Alors c'est vous qui voyez...



Le secret de Billy, c'est le OU Exclusif (XOR en anglais). Il permet de disposer de 16 couleurs et d'afficher puis d'effacer le sprite avec une seule et même opération booléenne. L'inconvénient c'est que ça créé des mélanges bizarres.

Codage de 1 pixel									
n°	Code binaire				Couleur	OU Exclusif			
0	0	0	0	0	Black	A	B	S	
1	0	0	0	1	Blue	0	0	0	
2	0	0	1	0	Dark Blue	0	1	1	
3	0	0	1	1	Dark Red	1	0	1	
4	0	1	0	0	Red	1	1	0	
5	0	1	0	1	Green	$A \oplus B = S$			
6	0	1	1	0	Light Blue				
7	0	1	1	1	Grey				
8	1	0	0	0	Light Purple				
9	1	0	0	1	Orange				
10	1	0	1	0	Pink				
11	1	0	1	1	Light Green				
12	1	1	0	0	Cyan				
13	1	1	0	1	Yellow				
14	1	1	1	0	Light Yellow				
15	1	1	1	1	White				

Exemple 1 : La couleur de fond rouge est modifiée en blanc puis restituée rouge quand elle est mélangée deux fois par OUEX avec un pixel de sprite vert fluo.

- %0100 (rouge) %1011 (vert fluo) = %1111 (blanc).
- %1111 (blanc) %1011 (vert fluo) = %0100 (rouge).

Donc on affiche le sprite, ça modifie les couleurs de fond. Puis on réaffiche le même sprite à la même place et ça restitue le fond tout en effaçant le sprite (réafficher le sprite = l'effacer).

Exemple 2 : le noir de transparence ne modifie pas une couleur de fond avec le OUEX.

- %1001 (orange) %0000 (noir transparent) = %1001 (orange).

Et réafficher le noir ne changera pas plus les choses.

Donc Billy fait un premier OUEX de son sprite avec le décor. Les pixels noirs de transparences ne modifient pas le décor, ils sont donc bien transparents. Les autres modifient les couleurs de décor. Mais un deuxième OUEX du sprite avec le décor permet de le restituer complètement tout en effaçant le sprite.

Les deux méthodes que l'on vient de voir ont deux avantages :

- elles ne sont pas destructives pour le décor car une simple opération logique permet de le restituer (mise à 0 des 2 bits de poids faible dans un cas, OUEX dans l'autre cas). Il n'y a donc pas besoin de copier le fond derrière le sprite puis de le restituer, comme cela se fait dans une gestion de sprites classique.
- Elles limitent les opérations logiques au minimum pour gérer l'affichage, la transparence et la restitution de fond. Ce qui permet de gagner de nombreux cycles machine par octet.

Méthode totale en mode 0

Donc vous préférez 16 couleurs avec de la vraie transparence, un vrai décor, un scrolling et plein de sprites ?

Bien sûr que ça se fait sur CPC.... par contre en réduisant un peu l'écran comme dans "1943"1988 © Go! (une marque d'U.S.Gold).



Pour faire de la vraie transparence en mode 0 il faut :

- Sauvegarder le décor derrière le sprite.
- Afficher chaque pixel du sprite différent de %0000 (couleur transparente), et laisser les pixels de fond sinon.
- Restituer le décor derrière le sprite.
- Déplacer le sprite et recommencer à l'étape 1.

Les deux premières étapes peuvent être faites dans la même boucle. Mais par rapport aux deux méthodes précédentes, on a une très grosse perte de cycles. Il vaut généralement mieux recourir aux sprites codés, mais rien n'empêche d'utiliser cette méthode.

SASFÉPU... LA MEMOIRE ET LES SPRITES - PARTIE 1

```
;-----  
; Affiche un sprite transparent. Certaines parties du code sont redondantes  
; pour eviter les sauts au maximum.  
;  
; Entrees ;  
; - B = nombre de lignes  
; - C = largeur du sprite en octets.  
; - HL = adresse memoire du sprite.  
; - DE = adresse ecran du sprite.  
; - IX = adresse du buffer de sauvegarde.  
;-----  
TRAN_SPR_PR    PUSH BC  
               PUSH DE  
  
TRAN_SPR_PO    LD A,(DE)           ; Sauvegarde de l'octet d'écran courant  
               LD (IX),A  
               INC IX  
               LD A,(HL)           ; A = octet du sprite courant.  
               LD B,A  
               OR A                ; est-il entièrement transparent?  
               JR Z,TRAN_SPR_P1    ; Oui => octet suivant  
  
               AND %10101010      ; Le pixel gauche est-il transparent?  
               JR NZ,TRAN_SPR_P3   ; Non => TRAN_SPR_P3  
  
               LD A,(DE)           ; Oui => Octet = couleur du pixel droit  
               AND %10101010      ; du sprite + couleur du pixel gauche  
               OR B                ; de l'écran.  
               LD (DE),A           ; Octet affiche  
  
TRAN_SPR_P1    INC HL             ; Octet suivant  
               INC DE  
               DEC C  
               JR NZ,TRAN_SPR_PO  
  
               POP DE              ; DE = ligne écran suivante  
               EX HL,DE  
               LD BC,&800  
               ADD HL,BC  
               JR NC,TRAN_SPR_P2  
               LD BC,&C050  
               ADD HL,BC  
  
TRAN_SPR_P2    EX HL,DE  
               POP BC  
               DJNZ TRAN_SPR_PR  
               RET  
  
TRAN_SPR_P3    LD A,B  
               AND %01010101      ; Le pixel droit est-il transparent?  
               JR NZ,TRAN_SPR_P5   ; Non => TRAN_SPR_P5  
  
               LD A,(DE)           ; Oui => Octet = couleur du pixel gauche  
               AND %01010101      ; du sprite + couleur du pixel droit  
               OR B                ; de l'écran.  
               LD (DE),A           ; Octet affiche
```

```

LD A, (DE) ; Oui => Octet = couleur du pixel gauche
AND %01010101 ; du sprite + couleur du pixel droit
OR B ; de l'écran.
LD (DE), A ; Octet affiche

INC HL ; Octet suivant
INC DE
DEC C
JR NZ, TRAN_SPR_P0

POP DE ; DE = ligne écran suivante
EX HL, DE
LD BC, &800
ADD HL, BC
JR NC, TRAN_SPR_P4
LD BC, &C050
ADD HL, BC
TRAN_SPR_P4 EX HL, DE
POP BC
DJNZ TRAN_SPR_FR
RET

TRAN_SPR_P5 LD A, (DE)
LD A, B
LD (DE), A ; octet à afficher = octet du sprite

INC HL ; Octet suivant
INC DE
DEC C
JR NZ, TRAN_SPR_P0

POP DE ; DE = ligne écran suivante
EX HL, DE
LD BC, &800
ADD HL, BC
JR NC, TRAN_SPR_P6
LD BC, &C050
ADD HL, BC
TRAN_SPR_P6 EX HL, DE
POP BC
DJNZ TRAN_SPR_FR
RET

```

Ce code paraît énorme pour l'affichage d'un sprite, mais il y a beaucoup de codes redondants pour éviter les sauts au maximum. Chaque octet de sprite pointé par HL est décortiqué pour vérifier si le pixel gauche, ou le droit, ou les deux sont transparents. Chaque pixel transparent est alors remplacé par le pixel de fond (celui pointé par DE à l'écran).

La restitution du fond se fait ensuite avec la routine d'affichage d'un sprite non transparent. En effet, la case mémoire où est mémorisé le fond, avec une taille égale au sprite, peut aussi être considérée comme un sprite non transparent. Sauvegarder et restituer tout le fond du sprite, pixels transparents compris, est beaucoup plus rapide qu'une sauvegarde et restitution conditionnelle nécessitant un LD, un AND et un JR conditionnel.

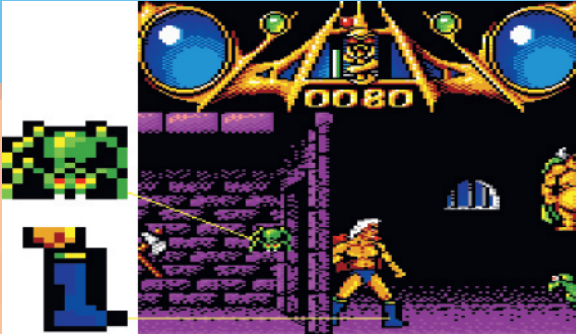
SASFÉPU... LA MEMOIRE ET LES SPRITES - PARTIE 2

Méthode simplifiée sans demi-transparence

Pour réduire cette gestion très coûteuse en cycles, certains jeux n'hésitent pas à supprimer les demi-octets transparents. Les octets ont donc soit deux pixels transparents, soit deux pixels pleins.

Exemple : dans "Savage"1988 © Firebird, les octets de sprites sont soit 100% transparents, soit 100% pleins. Si besoin, les pixels significatifs sont complétés par des pixels noirs pour éviter la demi-transparence (un octet avec un pixel plein et un pixel transparent).

Comme le jeu est très rapide et que les décors sont très sombres, la méthode passe plutôt inaperçue.



Le code est alors nettement simplifié, donc beaucoup plus rapide :

```
-----  
; Affiche un sprite transparent sans demi-transparence (les octets sont soit  
; 100% transparents, soit 100% pleins).  
;  
; Entrees ;  
; - B = nombre de lignes.  
; - C = largeur du sprite en octets.  
; - HL = Adresse mémoire du sprite.  
; - DE = Adresse écran du sprite.  
; - IX = Adresse du buffer de sauvegarde.  
-----  
TRAN_SPR_FR    PUSH BC  
               PUSH DE  
  
TRAN_SPR_P0    LD A, (DE)           ; Sauvegarde de l'octet d'écran courant  
               LD (IX), A  
               INC IX  
               LD A, (HL)           ; A = octet du sprite courant  
               OR A                ; Est-il entièrement transparent?  
               JR Z, TRAN_SPR_P1    ; Oui => octet suivant  
  
               LD (DE), A          ; Non => affichage de l'octet  
  
TRAN_SPR_P1    INC HL              ; Octet suivant  
               INC DE  
               DEC C  
               JR NZ, TRAN_SPR_P0  
  
               POP DE              ; DE = Ligne écran suivante  
               EX HL, DE  
               LD BC, &800  
               ADD HL, BC  
               JR NC, TRAN_SPR_P2  
               LD BC, &C050  
               ADD HL, BC  
  
TRAN_SPR_P2    EX HL, DE  
               POP BC  
               DJNZ TRAN_SPR_FR  
               RET
```

Cette fois on vérifie si l'octet courant est entièrement transparent. Si oui on passe à l'octet suivant. Sinon l'octet est copié directement à l'écran sans aucune opération logique.

On voit clairement la différence avec une gestion complète de la transparence par demi-octets. Il est donc souhaitable de prendre en compte cet aspect lors de la création des sprites. Mais ceci n'est pas toujours réalisable, comme dans le cas de "1943" où les avions ont des formes très précises.

Méthode totale en mode 1

Gérer totalement la transparence en mode 1 est un peu plus simple, mais nécessite quand même une bonne quantité d'opérations logiques par octet. L'algorithme joue sur l'encodage des bits, dont les poids forts sont à droite, et les poids faibles sont à gauche. Grâce à ça, il devient facile de créer des masques d'ajout ou de suppression de couleurs.

Exemple : soit un sprite avec une couleur transparente 0. Il impose ses couleurs sur le fond sauf pour la transparente.

Sprite	2	2	1	0	=	0	0	1	0	1	1	0	0
Fond	0	3	3	1	=	0	1	1	1	0	1	1	0
Résultat	2	2	1	1	=	0	0	1	1	1	1	0	0

Algorithme :

- A = octet de sprite = %00101100
- C = A
- A = inversion poids faible <-> poids fort = %11000010
- A <= A OR C (masque de couleur 3) = % 11101110
- B = CPL A = masque des couleurs de fond à supprimer = %00010001
- A = octet de fond d'écran = % 0011 1100
- A <= A AND B (suppression des couleurs gênantes) = % 00010000
- A <= A OR C (ajout des couleurs du sprite) = % 00111100

```

-----
; Affichage d'un sprite entier avec transparence en mode 1
; Entrées ;
; - LIGNES (DB) = nombre de lignes du sprite.
; - LARGEUR (DW) = largeur du sprite en octets.
; - HL = Adresse de la table de sprite.
; - DE = Adresse écran du sprite.
-----
AFF_SPRITE      LD C, (LIGNES)
AFF_SPRITE1     PUSH DE
                LD B, (LARGEUR)          ; B = nombre d'octets à copier par ligne
AFF_SPRITE2     PUSH BC
                LD A, (HL)              ; A = octet courant du sprite
                LD C, A                 ; puis C
                RLCA
                RLCA
                RLCA
                RLCA
                OR C                    ; Inversion poids faible <-> poids fort
                CPL                      ; A = masque des couleurs de sprite à conserver
                LD B, A                 ; inversion pour masque des couleurs de fond à supprimer
                AND B                   ; B = masque des couleurs à supprimer
                OR C                     ; A = octet de fond d'écran courant
                LD (DE), A              ; suppression des couleurs gênantes
                INC HL                  ; ajout des couleurs du sprite
                INC DE                  ; affichage du résultat
                POP BC
                DJNZ AFF_SPRITE2       ; Rebouclage jusqu'à traitement complet de la ligne

```

SASFÉPU... LA MEMOIRE ET LES SPRITES - PARTIE 2

```
POP DE
PUSH HL
LD HL, &800
ADD HL, DE
JR NC, AFF_SPRITES
LE A, C
LE BC, &C050
ADD HL, BC
LE C, A
EX HL, DE ; DE = ligne écran suivante
POP HL
DEC C
JR NZ, AFF_SPRITE1
RET
```

Le LDIR est remplacé par une boucle de gestion de transparence de chaque octet d'une ligne.

Méthode totale en mode 2

En mode 2 rien de bien compliqué. Comme il n'y a qu'un bit par couleur, il suffit de faire un OU entre le sprite et le fond. Les 1 du sprite s'imposent, et les 0 prendront la valeur du bit de fond, soient 0 ou 1.

Pour restituer le fond, il faut quand même le mémoriser car le OU est destructif. On ne peut pas savoir si derrière un 1 il y avait un 0 ou un 1 (0 OU 1 = 1 OU 1 = 1).

GESTION DES COLLISIONS

La gestion des collisions concerne aussi bien l'interaction entre sprites et décor, que l'interaction des sprites entre eux.

Collisions avec un décor 2D à fond uniforme

Dans les jeux à fond uniforme, la détection avec les décors est simplifiée. Par exemple, nous avons vu le cas de "Gauntlet II" 1986 © U.S. Gold dont les différences de couleur simplifient les détections des murs et des ennemis.

Cela dit rien n'empêche d'utiliser la méthode suivante, surtout si le décor est construit en tuiles comme dans "Jet Set Willy" 1984 © Software Projects ou "Cauldron II" 1986 © Palace Software. Le fond noir simplifie les déplacements, car la voie est libre tant qu'aucune couleur n'est détectée. Si la citrouille croise une couleur, un calcul est fait pour trouver la tuile ou le sprite correspondant.

La détection de couleur n'a pas besoin d'être faite sur tous les octets d'un sprite. Elle peut se faire sur deux ou trois octets dans le sens de déplacement (ex : à gauche s'il se déplace à gauche).

Comme pour la transparence totale en mode 1, la détection de collision en mode 1 nécessite également quelques combinaisons logiques.

Exemple : si un pixel du sprite de couleur (1) se superpose à un pixel de couleur du fond (2), il y a collision.

Sprite	2	2	1	0	=	0	0	1	0	1	1	0	0
Fond	0	0	2	3	=	0	0	0	1	0	0	1	1

Algorithme :

- A = octet de fond = %0001 0011
- On isole le poids faible : A <= A AND &0F = %0000 0011
- Poids faible = poids fort (masque de couleur 3) : A = %0011 0011
- B = A
- A = octet de la voiture = %0010 1100
- A AND B = %0010 0000 différent de 0 donc collision !

Le code correspondant peut-être celui-ci :

```
DETECT      LD A,(DE)      ; A = octet de fond d'écran courant
            AND &0F      ; Poids faible pour analyse du blanc et rouge (coul. 2 &
            LD B,A       ; B = poids faible
            RLCA
            RLCA
            RLCA
            RLCA      ; A = Inversion poids faible <-> poids fort
            OR B       ; poids fort = poids faible (masque de couleurs 3)
            LD B,A     ; recopie dans B
            LD A,(HL)  ; A = octet courant du sprite
            AND B      ; A l'emplacement du masque = couleur 0 (transparente)?
            JR Z,SUITE ; Oui => SUITE
            LD A,1
            LD (OBSTACLE),A ; non => indicateur d'obstacle rencontré
```

Collisions avec un décor 2D à fond plein

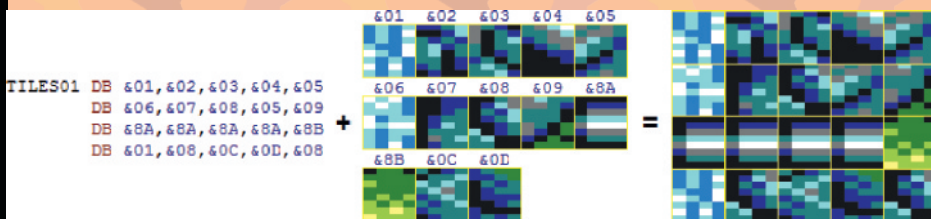
Dans un décor plein très coloré, il n'est plus possible de détecter les obstacles à l'aide des couleurs.

Alors comment les sprites font-ils pour reconnaître les plateformes sur lesquelles ils peuvent marcher dans "Gryzor"1987 © Ocean Software ?

Simple : le décor est construit avec des tuiles. Et certaines tuiles sont identifiées avec un code de sol en mémoire.



Exemple : grossissons une partie du décor et imaginons qu'elle est codée à part avec ses propres tuiles. Ici 13 tuiles de 4x8 pixels (8 octets) permettent de construire une portion de décor de 20 tuiles. Chaque tuile a un code distinct, et un tableau TILES01 décrit comment les positionner. Les tuiles de sol ont un code avec le bit de poids fort à 1 (&8x).



Donc quand le personnage se déplace ou qu'il saute, le jeu vérifie que le code de la tuile sur laquelle il se tient est un code de sol (&8x). Il suffit pour cela d'une simple conversion des coordonnées X,Y en coordonnées de tuile dans le tableau de construction du tableau.

Remarque : ce n'est pas forcément comme ça que "Gryzor" est codé, c'est juste pour illustrer le principe.

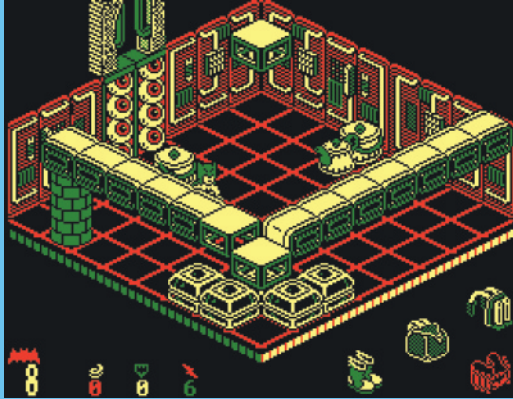
SASFÉPU... LA MEMOIRE ET LES SPRITES - PARTIE 2

Collisions avec un décor 3D isométrique

Et cette fois, il fait comment "Batman" 1986 © Ocean Software pour savoir où il peut aller et sur quoi il peut marcher ?

Jon RITMANa expliqué dans son interview que quand un sprite peut tomber, le moteur vérifie à chaque mouvement qu'il est toujours relié au cube sur lequel il est. Sinon le sprite tombe jusqu'à ce qu'il soit relié à un nouveau cube dans la liste chaînée.

A chaque pas, le moteur vérifie aussi si les objets entourent le sprite du joueur dans la liste chaînée. Si un objet est touché, une action « push » lui est attribuée, le forçant à se déplacer ou non. Il a aussi expliqué que quand le joueur atteint le bord d'une porte, cela provoque un effet de poussée sur le joueur, qui est attiré vers le centre de la porte.



Collisions entre sprites dans un décor 2D à fond uniforme

Comme dans le cas de la détection de décors, si un sprite détecte une couleur, alors un calcul est réalisé pour savoir à qui il appartient. On peut alors imaginer une comparaison de coordonnées entre le sprite et les autres présents à l'écran. Dans "Gauntlet II" 1986 © U.S.Gold, la couleur semble suffisante, puisque les monstres ne sont pas de la même couleur que les murs.

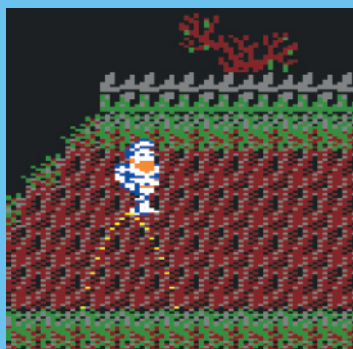
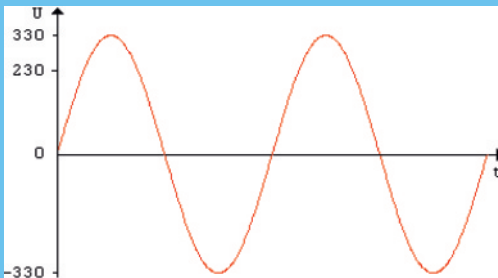
Collisions entre sprites dans un décor 2D à fond plein

Idem, une comparaison de coordonnées peut être faite à chaque déplacement d'un sprite (balle, ennemi...).

DÉPLACEMENTS PRÉCALCULÉS

Certains sprites peuvent avoir des mouvements difficiles à calculer, comme par exemple des courbes, des cercles,...

Comme dans "Ghosts'N Goblins", les personnages sautent généralement selon une courbe qui rappelle la tension alternative de nos prises électriques. Il s'agit d'une demi-sinusoïde.



Hors de question de réaliser des calculs trigonométriques avec notre Z80 dépourvu de fonctions mathématiques. Pour savoir de combien de pixels notre sprite doit bouger à chaque fois, on utilise une table de mouvements précalculés.

Pour simplifier, on travaille en adresses écran, et on passe par des additions de nombres relatifs 16 bits. Par exemple, $&DE56 - &1800 = &C656$ (-3 lignes à partir de l'adresse écran $&DE56$). Mais on peut aussi faire $&DE56 + &E800 = &C656 + \text{retenue}$. Il suffit ensuite de garder le résultat sans se soucier de la retenue.

Le dessin ci-dessous montre les offset pour +/- 2 octets et +/- 3 lignes à partir de l'adresse écran $&DE56$. Donc quelle que soit la direction, on ajoute des nombres 16 bits à l'adresse.

C654	$+&E7FE$	55	C656	$+&E800$	C657	C658	$+&E802$
CE54		CE55	CE56		CE57	CE58	
D654		D655	D656		D657	D658	
DE54	$+&FFFE$	DE55	DE56		DE57	DE58	$+&0002$
E654		E655	E656		E657	E658	
EE54		EE55	EE56		EE57	EE58	
F654	$+&17FE$	F655	F656	$+&1800$	F657	F658	$+&1802$

Evidemment il y a un problème : pour passer à la ligne suivante, on ne sait jamais s'il faut ajouter $&800$ ou $&C850$. L'exemple ci-dessus n'était qu'un cas particulier. Alors comment faire un tableau avec des valeurs fixes pour +/- 1 ligne, +/- 2 lignes, ... ? Simple : en s'arrangeant pour que les sprites soient toujours positionnés sur une ligne multiple de 8.

Exemple "dans" "Ghosts'N Goblins", le héros se balade le plus souvent sur la ligne 168, d'adresse $&FE40$.

En s'arrangeant pour que le personnage soit toujours posé sur une ligne multiple de 8 (8, 16, 24, ..., 168, ..., 200), la courbe de saut aura toujours le même enchaînement de valeurs relatives.

Et c'est d'ailleurs pour cette raison que de nombreux jeux de plateforme sont construits avec des tuiles de 8 lignes de haut. Comme ça les sprites sont toujours posés sur des lignes multiples de 8. Et les tuiles sont ultra rapides à dessiner car il faut juste ajouter $&0050$ pour passer à la ligne suivante, sans avoir à vérifier si ça dépasse $&FFFF$.

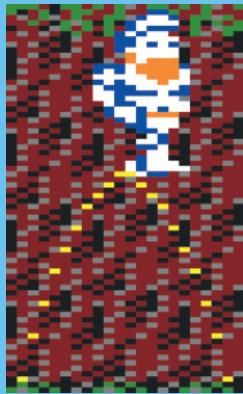
137	C590	C595	C596	C597	C598	C599	C59A	C59B
138	CD50	CD95	CD96	CD97	CD98	CD99	CD9A	CD9B
139	D550	D595	D596	D597	D598	D599	D59A	D59B
140	DD50	DD95	DD96	DD97	DD98	DD99	DD9A	DD9B
141	E550	E595	E596	E597	E598	E599	E59A	E59B
142	ED50	ED95	ED96	ED97	ED98	ED99	ED9A	ED9B
143	F550	F595	F596	F597	F598	F599	F59A	F59B
144	FD50	FD95	FD96	FD97	FD98	FD99	FD9A	FD9B
145	CSA0	CSB5	CSB6	CSB7	CSB8	CSB9	CSBA	CSBB
146	CDA0	CD95	CD96	CD97	CD98	CD99	CD9A	CD9B
147	DA00	DA95	DA96	DA97	DA98	DA99	DA9A	DA9B
148	DDA0	DD95	DD96	DD97	DD98	DD99	DD9A	DD9B
149	ESA0	ESB5	ESB6	ESB7	ESB8	ESB9	ESBA	ESBB
150	EDA0	ED95	ED96	ED97	ED98	ED99	ED9A	ED9B
151	FA00	F595	F596	F597	F598	F599	F59A	F59B
152	FD00	FD95	FD96	FD97	FD98	FD99	FD9A	FD9B
153	CSF0	CS95	CS96	CS97	CS98	CS99	CS9A	CS9B
154	CDF0	CD95	CD96	CD97	CD98	CD99	CD9A	CD9B
155	DSF0	DS95	DS96	DS97	DS98	DS99	DS9A	DS9B
156	DDF0	DD95	DD96	DD97	DD98	DD99	DD9A	DD9B
157	ESF0	E595	E596	E597	E598	E599	E59A	E59B
158	EDF0	ED95	ED96	ED97	ED98	ED99	ED9A	ED9B
159	FSF0	F595	F596	F597	F598	F599	F59A	F59B
160	FDf0	FD95	FD96	FD97	FD98	FD99	FD9A	FD9B
161	C640	C655	C656	C657	C658	C659	C65A	C65B
162	CE40	CE55	CE56	CE57	CE58	CE59	CE5A	CE5B
163	D640	D655	D656	D657	D658	D659	D65A	D65B
164	DE40	DE55	DE56	DE57	DE58	DE59	DE5A	DE5B
165	E640	E655	E656	E657	E658	E659	E65A	E65B
166	EE40	EE55	EE56	EE57	EE58	EE59	EE5A	EE5B
167	F640	F655	F656	F657	F658	F659	F65A	F65B
168	FE40	FE55	FE56	FE57	FE58	FE59	FE5A	FE5B
169	C690	C6A5	C6A6	C6A7	C6A8	C6A9	C6AA	C6AB

SASFÉPU... LA MEMOIRE ET LES SPRITES - PARTIE 2

Bien entendu, c'est par rapport au bord en haut à gauche du sprite qu'il faut se baser pour le calcul des déplacements relatifs. On vérifie quand même que le sprite est bien « posé » sur une ligne multiple de 8 avant.

Attention ici il y a une subtilité. Chaque point jaune se déplace de un pixel en X à chaque fois. Il y a donc deux sprites pour avoir une précision en X de 1/2 octet (1 pixel de mode 0 = 4 bits) et une courbe de saut plus harmonieuse. Dans ce cas on aura le tableau de valeurs relatives suivant (Δ veut dire écart par rapport à la dernière position) :

Position	N° sprite	Δ Octets	Δ Ligne
1	1	0	0
2	2	0	-8
3	1	+1	-6
4	2	0	-6
5	1	+1	-4
6	2	0	-4
7	1	+1	-2
8	2	0	-2
9	1	+1	0
10	2	0	0
11	1	+1	0
---	---	---	---



Il suffit ensuite de calculer les adresses relatives par rapport à une possible position du sprite. On obtient donc les valeurs suivantes : &FFB0, &0FB1, &0FB0, &E001, &1FB0, &E001, &2FB0, &F001, &F000, &0001,...

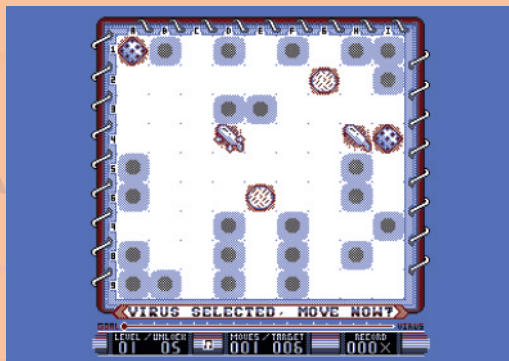
Ces valeurs seront ensuite ajoutées successivement aux coordonnées du sprite jusqu'à l'atterrissage et la reprise en main par le joueur.

Les positions du sprite sont bien alignées sur les octets en X et non sur les pixels. C'est la modification du sprite à chaque étape qui donne l'impression d'un déplacement en X au pixel près.

D3F4	D3F5	D3F6	D3F7	D3F8	D3F9
D3FA	D3FB	D3FC	D3FD	D3FE	D3FF
D400	D401	D402	D403	D404	D405
D406	D407	D408	D409	D40A	D40B
D40C	D40D	D40E	D40F	D410	D411
D412	D413	D414	D415	D416	D417
D418	D419	D41A	D41B	D41C	D41D
D41E	D41F	D420	D421	D422	D423
D424	D425	D426	D427	D428	D429
D42A	D42B	D42C	D42D	D42E	D42F
D430	D431	D432	D433	D434	D435
D436	D437	D438	D439	D43A	D43B
D43C	D43D	D43E	D43F	D440	D441
D442	D443	D444	D445	D446	D447
D448	D449	D44A	D44B	D44C	D44D
D44E	D44F	D450	D451	D452	D453
D454	D455	D456	D457	D458	D459
D45A	D45B	D45C	D45D	D45E	D45F
D460	D461	D462	D463	D464	D465
D466	D467	D468	D469	D46A	D46B
D46C	D46D	D46E	D46F	D470	D471
D472	D473	D474	D475	D476	D477
D478	D479	D47A	D47B	D47C	D47D
D47E	D47F	D480	D481	D482	D483
D484	D485	D486	D487	D488	D489
D48A	D48B	D48C	D48D	D48E	D48F
D490	D491	D492	D493	D494	D495
D496	D497	D498	D499	D49A	D49B
D49C	D49D	D49E	D49F	D4A0	D4A1
D4A2	D4A3	D4A4	D4A5	D4A6	D4A7
D4A8	D4A9	D4AA	D4AB	D4AC	D4AD
D4AE	D4AF	D4B0	D4B1	D4B2	D4B3
D4B4	D4B5	D4B6	D4B7	D4B8	D4B9
D4BA	D4BB	D4BC	D4BD	D4BE	D4BF
D4C0	D4C1	D4C2	D4C3	D4C4	D4C5
D4C6	D4C7	D4C8	D4C9	D4CA	D4CB
D4CC	D4CD	D4CE	D4CF	D4D0	D4D1
D4D2	D4D3	D4D4	D4D5	D4D6	D4D7
D4DA	D4DB	D4DC	D4DD	D4DE	D4DF
D4E0	D4E1	D4E2	D4E3	D4E4	D4E5
D4E6	D4E7	D4E8	D4E9	D4EA	D4EB
D4ED	D4EE	D4EF	D4F0	D4F1	D4F2
D4F4	D4F5	D4F6	D4F7	D4F8	D4F9
D4FB	D4FC	D4FD	D4FE	D4FF	D500
D501	D502	D503	D504	D505	D506
D507	D508	D509	D50A	D50B	D50C
D50E	D50F	D510	D511	D512	D513
D514	D515	D516	D517	D518	D519
D51B	D51C	D51D	D51E	D51F	D520
D522	D523	D524	D525	D526	D527
D529	D52A	D52B	D52C	D52D	D52E
D52F	D530	D531	D532	D533	D534
D536	D537	D538	D539	D53A	D53B
D53D	D53E	D53F	D540	D541	D542
D544	D545	D546	D547	D548	D549
D54B	D54C	D54D	D54E	D54F	D550
D552	D553	D554	D555	D556	D557
D559	D55A	D55B	D55C	D55D	D55E
D55F	D560	D561	D562	D563	D564
D566	D567	D568	D569	D56A	D56B
D56D	D56E	D56F	D570	D571	D572
D574	D575	D576	D577	D578	D579
D57B	D57C	D57D	D57E	D57F	D580
D582	D583	D584	D585	D586	D587
D589	D58A	D58B	D58C	D58D	D58E
D58F	D590	D591	D592	D593	D594
D596	D597	D598	D599	D59A	D59B
D59D	D59E	D59F	D5A0	D5A1	D5A2
D5A4	D5A5	D5A6	D5A7	D5A8	D5A9
D5AB	D5AC	D5AD	D5AE	D5AF	D5B0
D5B2	D5B3	D5B4	D5B5	D5B6	D5B7
D5BA	D5BB	D5BC	D5BD	D5BE	D5BF
D5C0	D5C1	D5C2	D5C3	D5C4	D5C5
D5C6	D5C7	D5C8	D5C9	D5CA	D5CB
D5CE	D5CF	D5D0	D5D1	D5D2	D5D3
D5D4	D5D5	D5D6	D5D7	D5D8	D5D9
D5DB	D5DC	D5DD	D5DE	D5DF	D5E0
D5E2	D5E3	D5E4	D5E5	D5E6	D5E7
D5EA	D5EB	D5EC	D5ED	D5EE	D5EF
D5F0	D5F1	D5F2	D5F3	D5F4	D5F5
D5F6	D5F7	D5F8	D5F9	D5FA	D5FB
D5FD	D5FE	D5FF	D600	D601	D602

EXPLOITATION DES SPRITES EN BASIC

S'il vaut mieux créer ses routines de sprites en assembleur, rien n'empêche de les exploiter ensuite en BASIC. En effet, il est parfaitement possible d'interfacer du BASIC avec du code machine. Ce qui permet de gérer facilement une interface tout en ayant des routines d'affichage rapides (comme par exemple avec "Virusdog" 2016 © cpc-power.com).



Les instructions BASIC d'interface avec le code machine

Une fois les routines de sprite compilées et chargées en mémoire, il existe quelques instructions permettant de faire communiquer ces routines avec le BASIC.

Instruction BASIC	Description
CALL adresse, paramètres	<p>L'instruction CALL permet d'exécuter une sous-routine à une adresse donnée. Une fois la sous-routine terminée, l'interpréteur BASIC passe à l'instruction qui suit le CALL.</p> <p>Exemple : exécuter la sous-routine à l'adresse &4000 (16384 en décimal) <code>CALL &4000</code></p> <p>32 paramètres de 16 bits peuvent être passés en arguments, afin de communiquer facilement avec une routine en code machine. Ils sont ensuite accessibles par le registre IX :</p> <ul style="list-style-type: none"> •(IX) = contenu du registre IX = valeur du dernier paramètre dans le sens (IX) = poids faible et (IX+1) = poids fort. •(IX+2) = valeur de l'avant-dernier paramètre. •(IX+4) = ... <p>A noter que certains registres sont modifiés :</p> <ul style="list-style-type: none"> •DE contient également la valeur du dernier paramètre. •C contient la valeur d'état ROM/RAM (en général &FF). •HL contient l'adresse de fin du CALL. <p>Exemple : <code>10 ECRAN=&C000:X=160:Y=100</code> <code>20 CALL &4000,ECRAN,X,Y</code></p> <p>A l'exécution de ce programme, les valeurs ECRAN, X et Y sont « pushées » en RAM. (IX) = Y = 100, (IX+2) = X = 160, (IX+4) = contenu du registre DE = &C000 Puis la sous-routine &4000 est appelée.</p>
POKE adresse, valeur	<p>Écrit la valeur 8 bits à l'adresse RAM indiquée. Cette instruction peut permettre de remplir diverses variables stockées en RAM, pour qu'elles soient ensuite utilisées par la routine.</p> <p>Exemple : écrire la valeur &FF à l'adresse &5000 <code>POKE &5000,&FF</code></p>
PEEK (adresse)	<p>Lit la valeur 8 bits à l'adresse RAM indiquée. Cette instruction peut permettre de lire des variables de résultat, calculées par la routine.</p> <p>Exemple : lire la valeur de X à l'adresse &5000 <code>X=PEEK(&5000)</code></p>

SASFÉPU... LA MEMOIRE ET LES SPRITES - PARTIE 2

Exemple d'appel d'une sous-routine d'affichage de sprite avec passage par argument

La sous-routine suivante affiche un sprite non transparent, avec comme paramètres des arguments de nombre de ligne du sprite, sa largeur, l'adresse de sa table, et de l'adresse de son affichage à l'écran. La routine est compilée en &4000 et l'adresse de VOITURE est soigneusement récupérée (&4029). On identifie aussi la longueur du code compilé (&403B) et on sauvegarde de code compilé avec la commande BASIC suivante :

save"sprite.bin",b,&4000,&403B

```

ORG &4000

;-----
; Affichage d'un sprite entier sans transparence, avec passage par arguments
; Arguments ;
; - (IX) = nombre de lignes du sprite.
; - (IX+2) = largeur du sprite en octets.
; - (IX+4) = Adresse de la table de sprite.
; - (IX+6) = Adresse écran du sprite.
;-----

AFF_SPRITE      LD A, (IX)           ; A = nombre de lignes du sprite
                LD L, (IX+4)       ; HL = Adresse de la table de sprite
                LD H, (IX+5)
                LD E, (IX+6)       ; DE = Adresse écran du sprite
                LD D, (IX+7)

AFF_SPRITE1     PUSH DE
                LD C, (IX+2)       ; BC = nombre d'octets à copier par ligne
                LD B, 0
                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 à l'écran
                POP HL
                DEC A
                JR NZ, AFF_SPRITE1
                RET

VOITURE         DB &03, &08, &00
                DB &17, &CC, &00
                DB &1F, &EE, &00
                DB &4E, &2D, &08
                DB &E5, &7A, &08
                DB &40, &20, &00

000006 4000          ; - (IX) = nombre de lignes du sprite.
000007 4000          ; - (IX+2) = largeur du sprite en octets.
000008 4000          ; - (IX+4) = Adresse de la table de sprite.
000009 4000          ; - (IX+6) = Adresse écran du sprite.
000010 4000          ;-----
000011 4000 DD 7E 00  AFF_SPRITE LD A, (IX)           ; A = nombre de lignes du sprite
000012 4003 DD 6E 04          LD L, (IX+4)       ; HL = Adresse de la table de sprite
000013 4006 DD 66 05          LD H, (IX+5)
000014 4009 DD 5E 06          LD E, (IX+6)       ; DE = Adresse écran du sprite
000015 400C DD 56 07          LD D, (IX+7)
000016 400F D6          AFF_SPRITE1 PUSH DE
000017 4010 DD 4E 02          LD C, (IX+2)       ; BC = nombre d'octets à copier par ligne
000018 4013 06 00          LD B, 0
000019 4015 ED B0          LDIR               ; Copie d'une ligne
000020 4017 D1          POP DE
000021 4018 E5          PUSH HL
000022 4019 21 00 08          LD HL, &800
000023 401C 19          ADD HL, DE
000024 401D 30 04          JR NC, AFF_SPRITE2
000025 401F 01 80 C0          LD BC, &C050
000026 4022 09          ADD HL, BC
000027 4023 EB          AFF_SPRITE2 EX HL, DE         ; DE = ligne suivante à l'écran
000028 4024 E1          POP HL
000029 4025 3D          DEC A
000030 4026 20 E7          JR NZ, AFF_SPRITE1
000031 4028 C9          RET
000033 4029 03 09 00  VOITURE  DB &03, &08, &00
000034 402C 17 CC 00          DB &17, &CC, &00
000035 402F 1F EE 00          DB &1F, &EE, &00
000036 4032 4E 2D 08          DB &4E, &2D, &08
000037 4035 E5 7A 08          DB &E5, &7A, &08
000038 4038 40 20 00          DB &40, &20, &00

```

Et on le lance avec le programme BASIC qui va bien :

```
list
10 MEMORY &3FFF
20 INK 0,0:INK 1,13:INK 2,6:INK 3,23
30 PAPER 0:LOAD"$SPRITE.BIN",&4000
40 CALL &4000,&C000,&4029,3,6
Ready
run
```

Bon normalement il faut mettre un « MODE 1 » au début du code, sinon l'adressage écran devient aléatoire. Je l'ai volontairement omis pour pouvoir lister le programme puis l'exécuter sans que ça efface tout.

MEMORY &3FFF permet de limiter la taille réservée au BASIC (Attention, nos variables en BASIC seront stockées de &3FFF en descendant alors que votre programme BASIC lui va de &0170 et monte, vous aurez compris que la rencontre entre les deux est à éviter à tout prix) pour pouvoir charger notre routine en &4000. Mais elle aurait très bien pu être compilée et exécutée en &8000.

La routine est ensuite appelée avec le CALL &4000, suivi des arguments suivants :

- &C000 : adresse écran où afficher le sprite.
- &4029 : adresse à laquelle se trouve la table VOITURE.
- 3 : largeur du sprite.
- 6 : nombre de lignes.

Pour ceux qui préfèrent, il y a aussi la méthode qui consiste à « paker » des variables utilisables par le programme. Mais ça complique le code BASIC car les POKE se font octets par octets.

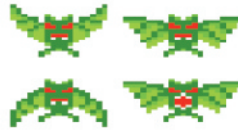
Modification des sprites par changement de couleur

Dans les jeux où l'espace mémoire est vraiment critique, il peut être intéressant d'utiliser une simple modification de palette pour donner l'illusion d'avoir différents monstres d'une même espèce. Si ce sont en plus des sprites codés, l'économie de mémoire peut être de plusieurs ko. Par exemple, dans le projet "The Shadows Of Sergoth" (sorti le Dimanche 6 mai 2018 sur le site www.cpc-power.com par Christophe PETIT & Kukulcan), les chauve-souris ont trois couleurs variables. Ce qui permet d'en avoir de trois sortes en changeant juste ces trois couleurs. Les bitmaps et l'animation restent pourtant les mêmes. (Vous ne rêvez pas, les captures écrans ne proviennent pas de la version finale mais d'une version de juin 2017)

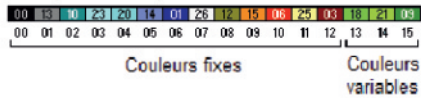




Occupation de la palette en mode 0



Occupation de la palette en mode 0



Occupation de la palette en mode 0



- 1 - Chauve souris vampire
- 2 - Chauve souris venimeuse
- 3 - Ze boss

Cette technique donne en plus l'illusion que le jeu a bien plus que 16 couleurs. L'inconvénient est qu'il n'est pas possible d'afficher les différentes sortes en même temps, sinon elles se mettraient à clignoter. Il faut donc s'assurer qu'elles sont bien cloisonnées dans différents secteurs.

PROGRAMMATION DES SPRITES CODÉS

Introduction

Précédemment nous avons vu quelques techniques pour accélérer l'affichage des sprites logiciels conventionnels :

- Chargements par LDI pour les sprites non transparents, mais aucune analyse possible des octets copiés.
- Réduction de la palette pour faire tenir la couleur de fond et la couleur de sprite dans un seul pixel.
- Mélange des couleurs par OU Exclusif pour simplifier l'affichage et l'effacement du sprite.
- Suppression de la demi-transparence, c'est-à-dire un octet avec un pixel plein et un pixel transparent.

Toutes ces méthodes conviennent peut-être dans certains jeux. Mais elles sont difficilement acceptables dans des beaux jeux rapides et bien colorés tels que "1943", "Gryzor"... Et puis ça reste des sprites logiciels conventionnels, des blocs de RAM recopiés vers l'écran. L'objectif de ce chapitre est de réunir les méthodes les plus efficaces pour afficher des sprites sur les CPC 464, 472, 664 et 6128.

Qu'est-ce qu'un sprite codé ?

Un sprite codé est d'abord la fusion du sprite avec son code d'affichage. C'est le chargement direct des couleurs à l'écran, par l'intermédiaire d'un pointeur HL uniquement.

Exemple : affichage d'une petite voiture en sprite codé non transparent

```

                                ORG &4000

SPRITE_CODE  LD HL,&C001      ; Voiture affiche en &C000
              LD B,&08        ; Prechargement de la valeur &08
              LD (HL),B
              DEC HL
              LD (HL),&03
              LD A,H          ; Passage a la ligne suivante
              ADD B           ; H = H + &08 => HL = HL + &0800
              LD H,A
              LD (HL),&17
              INC HL
              LD (HL),&CC
              LD A,H
              ADD B
              LD H,A
              LD (HL),&EE
              DEC HL
              LD (HL),&1F
              LD A,H
              ADD B
              LD H,A
              LD (HL),&4B
              INC HL
              LD (HL),&2D
              INC HL
              LD (HL),B
              LD A,H
              ADD B
              LD H,A
              LD (HL),B
              DEC HL
              LD (HL),&7A
              DEC HL
              LD (HL),&E5
              LD A,H
              ADD B
              LD H,A
              LD (HL),&40
              INC HL
              LD (HL),&20
              RET

```

Ligne	Adr. déb.	Adr. fin								
1	C000	C02B		03			08			00
2	C800	C82B		17			CC			00
3	D000	D02B		1F			EE			00
4	D800	D82B		4B			2D			08
5	E000	E02B		E5			7A			08
6	E800	E82B		40			20			00

```

list
10 INK 0,0:INK 1,13:INK 2,6:INK 3,23
20 CALL &4000
Ready
run
Ready

```

SASFÉPU... LA MEMOIRE ET LES SPRITES - PARTIE 2

Affichage direct à l'écran

Les valeurs d'octet sont rentrées, soit avec des valeurs immédiates, soit avec le registre B quand il y a des répétitions. Evidemment, le chargement par registre permet de gagner un cycle par rapport à une valeur immédiate (ou 3 cycles T).

LD (HL),&03 ; 2 octets / 3 cycles / 10T

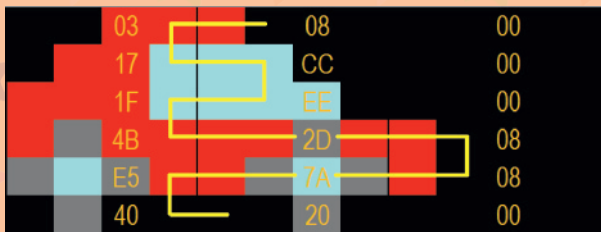
LD (HL),B ; 1 octet / 2 cycles / 7T

C'est pour ça qu'il est préférable d'avoir un maximum de répétitions, pour pouvoir les précharger dans B,C,D ou E.

HL est ensuite incrémenté ou décrémenté pour passer à l'octet suivant. A noter qu'ici on aurait pu seulement incrémenter ou décrémenter L pour gagner quelques précieux cycles T. Mais c'est difficilement possible dans un jeu, sauf si le sprite est très localisé. En effet, les transitions &xxFF => &xy00 sont très mal placées à l'écran, donc difficiles à éviter.

Seuls les octets importants sont pris en compte

L'affichage du sprite se fait en zigzagant, tout en évitant les octets &00. Seuls les octets importants sont parcourus et affichés. C'est l'un des gros avantages des sprites codés par rapport aux sprites conventionnels. Pas besoin de parcourir tout un sprite avec une double boucle X/Y. En allant directement à l'essentiel, on économise des sauts et des analyses.



Passage à la ligne suivante

Ici le passage à la ligne suivante se fait en ajoutant &08 à H, donc &0800 à HL. Oui mais... Pour cela il faut s'assurer que le sprite est toujours aligné sur les blocs de 8 lignes. C'est comme les sprites de "Gryzor" qui sont toujours posés sur des lignes multiples de 8, même principe ! Comme ça on économise pas mal de cycles en addition. Et si le sprite fait plus de 8 lignes de haut, alors on additionne &C850 toutes les 8 lignes.

Cette solution est ultra rapide, mais pose quand même quelques petits problèmes :

- Le code devient difficile à déboguer et à modifier.
- Le code n'est pas réexploitable.
- Ça prend deux à trois fois plus de mémoire que les sprites conventionnels.

Ligne	Adr. déb.	Adr. fin						
1	C000	C02B	03			08		00
2	C800	C82B	17			CC		00
3	D000	D02B	1F			EE		00
4	D800	D82B	4B			2D		08
5	E000	E02B	E5			7A		08
6	E800	E82B	40			20		00
7	F000	F02B	00			00		00
8	F800	F82B	00			00		00
9	C050	C07B	03			08		00
10	C850	C87B	17			CC		00
11	D050	D07B	1F			EE		00
12	D850	D87B	4B			2D		08
13	E050	E07B	E5			7A		08
14	E850	E87B	40			20		00
15	F050	F07B	00			00		00
16	F850	F87B	00			00		00
17	C0A0	C0CB	03			08		00
18	C8A0	C8CB	17			CC		00
19	D0A0	D0CB	1F			EE		00
20	D8A0	D8CB	4B			2D		08
21	E0A0	E0CB	E5			7A		08
22	E8A0	E8CB	40			20		00
23	F0A0	F0CB	00			00		00
24	F8A0	F8CB	00			00		00

Cette solution est ultra rapide, mais pose quand même quelques petits problèmes :

- Le code devient difficile à déboguer et à modifier.
- Le code n'est pas réexploitable.
- Ça prend deux à trois fois plus de mémoire que les sprites conventionnels.

Gestion de la transparence

Avec les sprites codés, les octets entièrement transparents ne sont même pas pris en compte dans le code. Par contre, pour les octets contenant des pixels transparents et des pixels non transparents, c'est plus délicat. Par exemple, le premier octet affiché de la voiture est &08, ce qui signifie l'organisation suivante en mode 1 :



Le rouge correspond bien à la couleur d'encre $2 = \%10$ (bit de poids fort dans la moitié droite de l'octet). Pour prendre en compte la transparence, il faudrait modifier le (HL),b par :

```

SPRITE_CODE    LD HL,&C001
                LD B,&08          ; Prechargement de la valeur &08
                LD A,(HL)
                AND %01111111    ; Affichage du rouge avec transparence
                OR %00001000
                LD (HL),A
    
```

Le AND impose le 0 de poids faible du rouge, sans toucher aux autres couleurs. Et le OU impose le 1 de poids fort du rouge sans toucher aux autres couleurs. Là aussi il est possible de précharger les masques ET/OU dans des registres, mais on voit que la semi-transparence fait perdre pas mal de cycles. D'où l'intérêt d'avoir des octets pleins autant que possible.

Sauvegarde du fond d'écran

Si l'on gère la transparence, c'est qu'il y a un fond d'écran à sauvegarder puis à restituer. Ceci peut être fait en intercalant une sauvegarde à chaque octet, à l'aide DE, pointeur sur un buffer de sauvegarde.

```

SPRITE_CODE    LD HL,&C001
                LD B,&08          ; Prechargement de la valeur &08
                LD A,(HL)
                LD (DE),A        ; Octet de fond sauvegarde
                INC DE
                AND %01111111    ; Affichage du rouge avec transparence
                OR %00001000
                LD (HL),A
    
```

Restitution du fond d'écran

Du coup, si on sauvegarde avec la méthode ci-dessus, il faut une routine symétrique à celle de l'affichage du sprite. Celle-ci suivra le même cheminement, sauf qu'elle affichera les octets sauvegardés dans le buffer. Ce qui effacera la voiture de la même manière qu'elle a été affichée.

```

RESTAURE_CODE  LD HL,&C001
                LD B,&08          ; Prechargement de la valeur &08
                LD A,(DE)        ; Restitution de l'octet de fond
                INC DE
                LD (HL),A
                DEC HL
                LD A,(DE)        ; Restitution de l'octet de fond
                INC DE
                LD (HL),A
                LD A,H           ; Passage a la ligne suivante
                ADD B             ; H = H + &08 => HL = HL + &0800
                LD H,A
    
```

Désactivation des interruptions

Une astuce qui permet de gagner encore quelques précieux cycles consiste à désactiver les interruptions pendant la gestion des sprites. Une interruption est un signal qui oblige le microprocesseur à interrompre sa tâche en cours, pour aller effectuer une autre tâche. Une fois cette tâche effectuée, il revient à sa tâche précédente. Il y a trois modes d'interruption sur le CPC :

- **Interruptions mode 0 (IM0)** : Ce sont des interruptions envoyées par d'autres circuits sur le bus de données. Elles ne sont pas exploitables par programme. Elles servent surtout à la réinitialisation du système.

- **Interruptions mode 1 (IM1)** : Ici c'est le VGA qui génère ce signal 300 fois par secondes (toutes les 3,3 ms). Dès que le Z80 la reçoit, il sauvegarde ses registres dans la pile et saute à l'adresse RAM &0038. Une fois les instructions effectuées, il revient à son traitement précédent.

L'adresse &0038 est régulièrement détournée dans les jeux pour y mettre un code qui nécessite un traitement régulier (exemple classique : déroulement d'une musique pendant un jeu).

- **Interruptions Mode 2 (IM2)** : Sur simple chargement du registre I avec une valeur 8 bits, un CALL indirect peut être effectué vers n'importe quelle adresse mémoire. Le programmeur doit toutefois maintenir une table d'adresses 16 bits pour chaque interruption.

Le problème est qu'en désactivant les interruptions, certaines fonctions ne sont plus accessibles :

- L'instruction HALT qui attend la prochaine interruption ne fonctionne plus. Il n'est donc plus possible de créer des boucles d'attente ou de synchronisation avec. Mais il y a toujours la VBL.

- Certains vecteurs système d'entrée/sortie ne fonctionnent plus ou mal. C'est pour ça qu'il ne faut pas désactiver tout le temps les interruptions, mais juste le temps de traiter les routines graphiques par exemple.

Le dossier sur les sprites continuera dans un prochain numéro avec l'utilisation d'un Amstrad CPC Plus et de sa palette étendue.

L'équipe de cpc-power

