

8 bits de PODER

Sur votre CPC AMSTRAD



"Les limites ne sont pas un problème,
mais une source d'inspiration".

V42_00

Jose Javier García Aranda

INDEX

1 POURQUOI PROGRAMMER UNE MACHINE 1984 AUJOURD'HUI ?.....	9
2 FONCTIONS 8BP ET UTILISATION DE LA MÉMOIRE	11
2.1 QU'EST-CE QU'UNE BIBLIOTHÈQUE RSX ?	12
2.2 FONCTIONS DU 8BP	13
2.3 AMSTRAD CPC ARCHITECTURE	14
2.3.1 <i>GOSUB /RETURN Pile</i>	19
2.3.2 <i>Une expérience pour voir la ROM avec Winape</i>	20
2.4 CARTE MÉMOIRE DU 8BP ET OPTIONS D'ASSEMBLAGE	21
3 OUTILS NÉCESSAIRES.....	25
4 PREMIERS PAS AVEC 8BP	27
4.1 INSTALLER WINAPE	27
4.2 SE FAMILIARISER AVEC WINAPE : "HELLO WORLD"	27
4.3 TÉLÉCHARGER LA BIBLIOTHÈQUE 8BP	27
4.4 EXÉCUTER LES DÉMONSTRATIONS	28
4.5 CRÉER SON PREMIER PROGRAMME AVEC LA 8BP	29
4.6 CRÉEZ VOTRE .DSK AVEC VOTRE JEU 8BP.....	32
5 LES ÉTAPES À SUIVRE POUR CRÉER UN JEU.....	33
5.1 STRUCTURE DES RÉPERTOIRES DE VOTRE PROJET.....	33
5.2 VOTRE JEU EN 3 FICHIERS SEULEMENT	34
5.3 CRÉEZ UN DISQUE OU UNE CASSETTE AVEC VOTRE JEU	36
5.3.1 <i>Fabrication d'un disque</i>	36
5.3.2 <i>Créer un ruban avec Winape</i>	36
5.3.3 <i>Créer une cassette facilement avec CPCDiskXP, 2cdt et tape2wav</i>	38
5.3.4 <i>Dépannage des fonctions LOAD et MEMORY</i>	39
6 BIBLIOTHÈQUE ASSEMBLAGE, MUSIQUE ET GRAPHISME	42
6.1 MAKE_ALL.ASM.....	43
6.2 STRUCTURE DU FICHIER IMAGE	45
6.3 STRUCTURE DU FICHIER DE LA SÉQUENCE D'ANIMATION.....	45
6.4 STRUCTURE DU FICHIER DE ROUTAGE	46
6.5 STRUCTURE DU FICHIER DE LA CARTE DU MONDE	46
7 CYCLE DE JEU	47
7.1 COMMENT MESURER LE FPS DE VOTRE CYCLE DE JEU	47
8 SPRITES.....	49
8.1 ÉDITION DE SPRITES AVEC SPEDIT ET ASSEMBLAGE DE SPRITES	49
8.2 IMPRIMER UN SPRITE	54
8.3 RETOURNEMENT DE SPRITES	55
8.4 SPRITES AVEC ÉCRASEMENT.....	57
8.4.1 <i>Utilisation de l'écrasement pour améliorer les chevauchements de sprites</i> 61	61
8.4.2 <i>Ecrasement avec 4 couleurs d'arrière-plan</i>	62
8.4.3 <i>Écraser dans le MODE 1</i>	63
8.4.4 <i>Comment peindre des sprites "derrière" l'arrière-plan</i>	64
8.4.5 <i>Comment utiliser plus de couleurs avec l'écrasement</i>	65
8.5 SPRITES AVEC IMAGES DE FOND	67

8.6	TABLEAU DES ATTRIBUTS DES SPRITES	69
8.7	TOUS LES SPRITES IMPRIMÉS ET TRIÉS.....	74
8.8	COLLISIONS ENTRE SPRITES	76
8.9	AJUSTEMENT DE LA SENSIBILITÉ DES COLLISIONS DE SPRITES.....	78
8.10	QUI ENTRE EN COLLISION ET AVEC QUI : COLSPALL	79
8.10.1	<i>Comment programmer un tir multiple à l'aide de COLSPALL</i>	80
8.10.2	<i>Qui s'effondre lorsqu'il y a plusieurs chevauchements</i>	81
8.10.3	<i>Utilisation avancée de l'octet de statut dans les collisions.....</i>	83
8.11	TABLEAU DES SÉQUENCES D'ANIMATION	84
8.12	SÉQUENCES D'ANIMATION SPÉCIALES	85
8.12.1	<i>Séquences de décès</i>	86
8.12.2	<i>Séquences de fin.....</i>	87
8.12.3	<i>Séquences enchaînées</i>	87
8.12.4	<i>Macroséquences d'animation.....</i>	87
9	VOTRE PREMIER JEU SIMPLE	91
9.1	MAINTENANT, SAUTONS ! BOING, BOING !	91
10	JEUX D'ÉCRANS : MISE EN PAGE OU CARTE DES CARREAUX	95
10.1	DÉFINITION ET UTILISATION DE LA MISE EN PAGE	95
10.2	EXEMPLE DE JEU AVEC MISE EN PAGE	98
10.3	COMMENT OUVRIR UN PORTAIL DANS LA MISE EN PAGE.....	100
10.4	UN JEU DE PUZZLE : LAYOUT AVEC UN ARRIÈRE-PLAN	101
10.5	COMMENT ÉCONOMISER DE LA MÉMOIRE DANS VOS MISES EN PAGE	103
11	PROGRAMMATION AVANCÉE ET "BULK LOGICS	105
11.1	MESURE DE LA VITESSE DES COMMANDES	105
11.2	UNE SEULE LOGIQUE POUR RÉGIR TOUS VOS ÉCRANS.....	112
11.3	TECHNIQUE DE LA "LOGIQUE DE MASSE	113
11.3.1	<i>Déplace 32 sprites avec des logiques massives</i>	114
11.3.2	<i>Exécution en cascade alternée et périodique.....</i>	115
11.3.3	<i>Exemple simple de logique de masse</i>	117
11.3.4	<i>Bloquer" le mouvement des escadrilles</i>	118
11.3.5	<i>Technique de logique massive dans les jeux de type "pacman</i>	119
11.3.6	<i>Réduction du nombre d'instructions dans un cycle donné.....</i>	121
11.3.7	<i>Routes qui accélèrent le jeu en manipulant l'état</i>	125
11.3.8	<i>Routage des sprites avec des "logiques massives".....</i>	125
12	CHEMINS COMPLEXES : COMMANDE ROUTEALL.....	129
12.1	PLACE UN SPRITE AU MILIEU D'UNE ROUTE : ROUTESP.....	131
12.2	CRÉATION D'ITINÉRAIRES AVANCÉS	133
12.2.1	<i>Changements d'état forcés des routes.....</i>	133
12.2.2	<i>Changements de séquence forcés à partir des itinéraires</i>	135
12.2.3	<i>Changements d'image forcés sur les itinéraires</i>	135
12.2.4	<i>Rerouting forcé à partir d'itinéraires</i>	139
12.2.5	<i>Changements de chemin forcés à partir de BASIC.....</i>	139
12.2.6	<i>Animation forcée des itinéraires</i>	141
12.2.7	<i>Comment construire des itinéraires "dynamiques" (non pré définis)</i>	141
12.2.8	<i>Programmation d'itinéraires, y compris de schémas.....</i>	142
12.2.9	<i>Typologie des itinéraires.....</i>	143

13 DEMI-OCTET MOUVEMENT DOUX.....	145
14 JEUX À DÉFILEMENT.....	147
14.1 ETOILES : ROULEAU D'ETOILES OU TERRE MOUCHETÉE	147
14.2 DÉFILEMENT À L'AIDE DE MOVERALL ET/OU AUTOALL.....	150
14.3 TECHNIQUE DU "REPÉRAGE"	152
14.4 MAP2SP : SCROLL BASÉ SUR UNE CARTE DU MONDE	155
14.4.1 <i>Carte du monde (Tableau des cartes)</i>	157
14.4.2 <i>Utiliser la fonction MAP2SP</i>	158
14.4.3 <i>Exemple de fichier de phase</i>	161
14.4.4 <i>Collision de l'ennemi avec la carte</i>	163
14.4.5 <i>Images d'arrière-plan dans votre défilement</i>	164
14.5 PARALLAXE SCROLL	165
14.6 MISE À JOUR DE LA CARTE DYNAMIQUE : UMAP	166
14.7 ANIMATION ET DÉFILEMENT DE L'ENCRE : COMMANDE RINK	168
14.7.1 <i>Course de voitures en 2D</i>	169
14.7.2 <i>Parchemin de brique</i>	171
15 JEUX DE PLATEFORME	173
16 LES HORDES D'ENNEMIS DANS LES JEUX À DÉFILEMENT	175
17 MINI-CARACTÈRES REDÉFINISSABLES : PRINTAT	177
17.1 CRÉEZ VOTRE PROPRE MINI-ALPHABET	178
17.2 ALPHABET PAR DÉFAUT POUR LE MODE 1	179
18 PSEUDO 3D	181
18.1 PROJECTION 3D	183
18.1.1 <i>Mathématiques de la pseudo-projection 3D</i>	184
18.1.2 <i>Courbes</i>	188
18.2 ZOOM IMAGES	189
18.3 UTILISATION DES SEGMENTS	191
19 MUSIQUE	193
19.1 EDITER DE LA MUSIQUE AVEC WYZ TRACKER	193
19.2 ASSEMBLAGE DES CHANSONS	194
19.3 QUE FAIRE SI VOUS NE POUVEZ PAS FAIRE TENIR DE LA MUSIQUE DANS 1400 OCTETS ?	195
20 C PROGRAMMATION AVEC 8BP.....	197
20.1 PREMIÈRE ÉTAPE : PROGRAMMER VOTRE JEU EN BASIC.....	198
20.2 ÉTAPE 2 : TRADUIRE VOTRE CYCLE DE JEU DE BASIC À C.....	199
20.2.1 <i>GOSUB et RETURN en C</i>	203
20.2.2 <i>Communication entre BASIC et C avec des variables BASIC</i>	204
20.2.3 <i>Chaînes de texte BASIC et C</i>	207
20.3 TROISIÈME ÉTAPE : COMPILER EN UTILISANT "COMPILA.BAT"	208
20.4 ÉTAPE 4 : VÉRIFIER LES LIMITES DE LA MÉMOIRE	210
20.5 CINQUIÈME ÉTAPE : LOCALISER L'ADRESSE DE LA FONCTION À INVOQUER À PARTIR DE BASIC211	
20.6 ÉTAPE 6 : INCLUDE DANS VOTRE JEU .DSK LE NOUVEAU BINAIRE	212
20.7 RÉFÉRENCE DE LA FONCTION 8BP EN C	213

20.8	RÉFÉRENCE À UNE FONCTION BASIC EN C ("MINIBASIC").....	215
21	GUIDE DE RÉFÉRENCE DE LA BIBLIOTHÈQUE 8BP.....	217
21.1	FONCTIONS DE LA BIBLIOTHÈQUE	217
21.1.1	<i>3D</i>	217
21.1.2	<i>ANIMA</i>	218
21.1.3	<i>ANIMALL</i>	219
21.1.4	<i>AUTO</i>	220
21.1.5	<i>AUTOALL</i>	220
21.1.6	<i>COLAY</i>	220
21.1.7	<i>COLSP</i>	221
21.1.8	<i>COLSPALL</i>	223
21.1.9	<i>LAYOUT</i>	224
21.1.10	<i>LOCATESP</i>	226
21.1.11	<i>MAP2SP</i>	226
21.1.12	<i>MOVER</i>	228
21.1.13	<i>TOUT LE MONDE</i>	228
21.1.14	<i>MUSIC</i>	229
21.1.15	<i>PEEK</i>	229
21.1.16	<i>POKE</i>	230
21.1.17	<i>PRINTAT</i>	230
21.1.18	<i>PRINTSP</i>	231
21.1.19	<i>PRINTSPALL</i>	232
21.1.20	<i>RINK</i>	234
21.1.21	<i>ROUTEALL</i>	235
21.1.22	<i>ROUTESP</i>	237
21.1.23	<i>SETLIMITS</i>	238
21.1.24	<i>SETUPSP</i>	238
21.1.25	<i>STARS</i>	240
21.1.26	<i>UMAP</i>	242
22	COMMENT FAIRE UN TABLEAU D'AFFICHAGE.....	243
23	AMÉLIORATIONS FUTURES POSSIBLES DE LA BIBLIOTHÈQUE.....	245
23.1	MÉMOIRE POUR LA LOCALISATION DE NOUVELLES FONCTIONS.....	245
23.2	RÉSOLUTION DES PIXELS D'IMPRESSION	245
23.3	LAYOUT DE MODE 1	245
23.4	CAPACITÉ DE FILMAGE.....	245
23.5	FONCTIONS DE DÉFILEMENT DU MATÉRIEL.....	246
23.6	MIGRATION DE LA BIBLIOTHÈQUE 8BP VERS D'AUTRES MICRO-ORDINATEURS	247
24	QUELQUES JEUX RÉALISÉS AVEC 8BP.....	249
24.1	MUTANT MONTOYA.....	249
24.2	ANUNNAKI, NOTRE PASSÉ EXTRATERRESTRE.....	250
24.3	NIBIRU	251
24.4	FRUITS ET LÉGUMES FRAIS	252
24.5	"3D RACING ONE.....	252
24.6	FANTÔME DE L'ESPACE.....	253
24.7	FROGGER ÉTERNEL	254
24.8	ERIDU : LE PORT SPATIAL.....	255
24.9	HAPPY MONTY.....	256
24.10	PILOTE DU BLASTER	256

24.11	NOMWARS.....	257
24.12	PACO, L'HOMME	258
24.13	MINI JEUX.....	259
24.13.1	<i>Mini-pong</i>	259
24.13.2	<i>Mini-Invaders</i>	260
25	ANNEXE I : ORGANISATION DE LA MEMOIRE VIDEO	261
25.1	L'ŒIL HUMAIN ET LA RÉSOLUTION DU CPC.....	261
25.2	MÉMOIRE VIDÉO	261
25.2.1	<i>Mode 2</i>	261
25.2.2	<i>Mode 1</i>	261
25.2.3	<i>Mode 0</i>	262
25.2.4	<i>Mémoire d'affichage</i>	262
25.3	CALCUL D'UNE ADRESSE D'ÉCRAN.....	264
25.4	CHIFFONS POUR ÉCRANS	264
25.5	COMMENT CRÉER UN ÉCRAN DE CHARGEMENT POUR VOTRE JEU	265
26	ANNEXE II : LA PALETTE	269
27	ANNEXE III : CODES INKEY	271
29	ANNEXE IV : TABLEAU ASCII DU CPC AMSTRAD	273
30	ANNEXE V : QUELQUES EFFETS SONORES	275
31	ANNEXE VI : ROUTINES INTÉRESSANTES POUR LES MICROPROGRAMMES.....	277
32	ANNEXE VII : TABLEAU DES ATTRIBUTS DES SPRITES.....	279
33	ANNEXE VIII : CARTE MÉMOIRE 8BP	281
34	ANNEXE IX : COMMANDES DISPONIBLES 8BP	283
35	ANNEXE X : OPTIONS D'ASSEMBLAGE 8BP.....	285
36	ANNEXE XI : CORRESPONDANCE RSX/APPEL	287
37	ANNEXE XII : FONCTIONS 8BP EN C	289
38	ANNEXE XIII : MINIBASIC EN C	291

1 Pourquoi programmer une machine de 1984 aujourd'hui ?

Parce que les limites ne sont pas un problème mais une source d'inspiration.

Les limites, qu'il s'agisse d'une machine ou d'un être humain, ou en général de toute ressource disponible, stimulent notre imagination pour les surmonter. L'AMSTRAD, une machine de 1984 basée sur le microprocesseur Z80, a une petite mémoire (64 Ko) et une petite capacité de traitement, mais seulement en comparaison avec les ordinateurs d'aujourd'hui. Cette machine est en fait un million de fois plus rapide que celle qu'Alan Turing a construite pour déchiffrer les messages de la machine Enigma en 1944. Comme tous les ordinateurs des années 1980, le CPC AMSTRAD démarrait en moins d'une seconde, l'interpréteur BASIC étant prêt à recevoir les commandes de l'utilisateur, BASIC étant le langage avec lequel les programmeurs apprenaient et réalisaient leurs premiers développements. L'AMSTRAD BASIC était particulièrement rapide par rapport à ses concurrents et, d'un point de vue esthétique, c'était un ordinateur très séduisant !



Fig. 1 : Le légendaire modèle AMSTRAD CPC464

Quant au microprocesseur Z80, il n'est même pas capable de multiplier (en BASIC, on peut multiplier, mais cela repose sur un programme interne qui implémente la multiplication au moyen d'additions ou de déplacements de registres), il ne peut faire que des additions, des soustractions et des opérations logiques. Malgré cela, c'était le meilleur processeur 8 bits et il ne comportait que 8500 transistors, contrairement à d'autres processeurs tels que le M68000 dont le nom vient précisément du fait qu'il comporte 68000 transistors.

UNITÉ CENTRALE	Nombre de de transistors	MIPS (millions d'instructions par seconde)	Ordinateurs et consoles qui l'intègrent
6502	3.500	0,43 @1Mhz	COMMODORE 64, NES, ATARI 800...
Z80	8.500	0,58 @4Mhz	AMSTRAD, COLECOVISION, SPECTRUM, MSX...
Motorola 68000	68.000	2.188 @ 12.5 Mhz	AMIGA, SINCLAIR QL, ATARI ST...
Intel 386DX	275.000	2.1 @16Mhz	PC
Intel 486DX	1.180.000	11 @ 33 Mhz	PC
Pentium	3.100.000	188 @ 100Mhz	PC
ARM1176		4744 @ 1Ghz (1186 par cœur)	Raspberry pi 2, Nintendo 3DS, Samsung galaxy, ...
Intel i7 (5ème génération)	2.600.000.000	238310 @ 3Ghz (presque !) 500.000 fois plus rapide qu'un Z80)	PC
AMD Ryzen 9 3950X (16 core)	3.800.000.000	749070 @4.8Ghz (1,3 million de fois plus rapide que le Z80)	PC

Tableau 1Comparaison des MIPS

Cela rend la programmation extrêmement intéressante et stimulante pour obtenir des résultats satisfaisants. Toute notre programmation doit viser à réduire la complexité informatique spatiale (mémoire) et temporelle (opérations), ce qui nous oblige à inventer des astuces, des ruses, des algorithmes, etc. et fait de la programmation une aventure passionnante. C'est pourquoi la programmation des machines à faible capacité de traitement est un concept intemporel, qui n'est pas soumis aux modes ni conditionné par l'évolution des technologies.

Tout le code de ce livre, y compris la bibliothèque qui vous permet de créer vos propres jeux ou de contribuer à la bibliothèque, se trouve dans le projet GitHub "8BP", à cette URL. Il suffit de télécharger le zip (dans un chapitre ultérieur, je vous donnerai un pas-à-pas).

<https://github.com/jjaranda13/8BP>

Il existe également un blog contenant de nombreuses informations :

<http://8bitsdepoder.blogspot.com.es>

Et une chaîne YouTube :

https://www.youtube.com/channel/UCThvesOT-jLU_s8a5ZOjBFA

2 Fonctions du 8BP et utilisation de la mémoire

La bibliothèque 8BP n'est pas un "moteur de jeu". Elle se situe entre une simple extension de commande BASIC et un moteur de jeu.

Les moteurs de jeu tels que game-maker, AGD (Arcade Game Designer), Unity, et bien d'autres, limitent dans une certaine mesure l'imagination du programmeur, l'obligeant à utiliser certaines structures, à programmer dans un langage de script limité la logique d'un ennemi, à définir et à relier les écrans de jeu, etc.

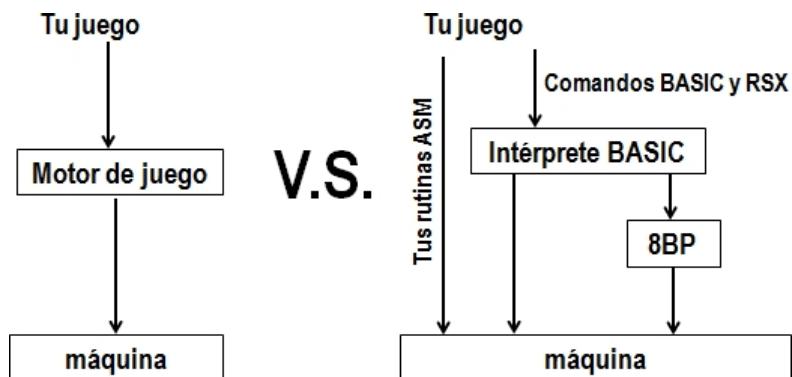


Fig. 2 Play Engines versus 8BP

La bibliothèque 8BP est différente. C'est une bibliothèque capable d'exécuter rapidement ce que BASIC ne peut pas faire. Des choses comme l'impression de sprites à pleine vitesse, ou le déplacement de banques d'étoiles autour de l'écran, sont des choses que BASIC ne peut pas faire, et 8BP le fait.

Et il n'occupe que 8 Ko !

Le BASIC est un langage interprété. Cela signifie que chaque fois que l'ordinateur exécute une ligne de programme, il doit d'abord vérifier qu'il s'agit d'une commande valide en comparant la chaîne de commande à toutes les chaînes de commande valides. Il doit ensuite valider syntaxiquement l'expression, les paramètres de la commande et même les plages autorisées pour les valeurs de ces paramètres. En outre, il lit les paramètres au format texte (ASCII) et doit les convertir en données numériques. Une fois tout ce travail effectué, il procède à l'exécution. Or, c'est tout ce travail effectué dans chaque instruction qui différencie un programme compilé d'un programme interprété comme ceux écrits en BASIC.

En équipant BASIC des commandes fournies par 8BP, il est possible de réaliser des jeux de qualité professionnelle, puisque la logique de jeu que vous programmez peut être exécutée en BASIC tandis que les opérations intensives du CPU telles que l'impression à l'écran ou la détection des collisions entre sprites, etc. sont effectuées en code machine par la bibliothèque. Cependant, tout n'est pas simple et sans problème. Bien que la bibliothèque 8BP vous fournisse des fonctions très utiles dans les jeux vidéo, vous devrez l'utiliser avec prudence car chaque commande que vous invoquerez passera par la couche d'analyse du BASIC, avant d'atteindre le monde souterrain du code machine où se trouve la fonction, de sorte que les performances ne seront jamais optimales. Vous devrez faire preuve d'ingéniosité et sauvegarder des instructions, mesurer les temps d'exécution des instructions et des morceaux de votre programme et réfléchir à des stratégies pour gagner du temps d'exécution. C'est une aventure d'ingéniosité et de plaisir. Vous apprendrez ici comment faire et je vous présenterai même une technique que j'ai appelée "logiques massives" qui vous permettra d'accélérer

vos jeux jusqu'à des limites que vous auriez pu croire impossibles.

En plus de la bibliothèque, vous disposez d'un éditeur de sprites et de graphiques simple mais complet et d'une série d'outils magnifiques qui vous permettront de vivre l'aventure de la programmation d'un micro-ordinateur au 21ème siècle.

Le "joli" dessin suivant schématise les possibilités offertes par le 8BP et a été utilisé dans le cadre d'une foire "rétro". Aujourd'hui, il offre davantage de possibilités.

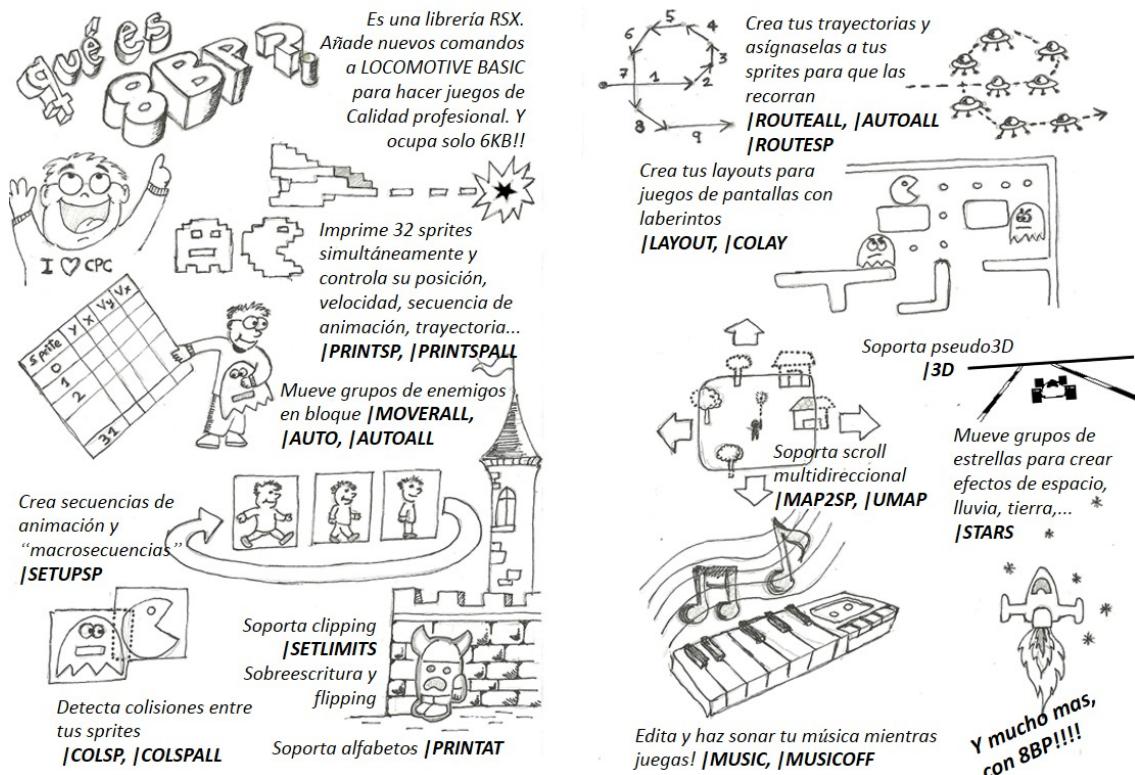


Fig. 3 Résumé du 8BP (actuellement plus de commandes)

2.1 Qu'est-ce qu'une bibliothèque RSX ?

RSX est l'acronyme de Resident System eXtensions. Les bibliothèques telles que 8BP qui fournissent des commandes pour étendre BASIC sont appelées bibliothèques RSX.

Sur le CPC6128, certaines des commandes utilisées pour piloter le lecteur de disque sont des commandes RSX préinstallées, telles que |TAPE, |DISC, |A, |B, |Si cette fonctionnalité n'existe pas, chaque routine 8BP devrait être invoquée avec un CALL <adresse>. Si cette fonctionnalité n'existe pas, chaque routine 8BP devrait être invoquée avec un CALL <adresse>, de sorte que l'existence de RSX rend les programmes plus compréhensibles.

Tout n'est pas toujours paisible et harmonieux. L'utilisation de RSX est plus lente que l'utilisation directe de CALL et si nous déclarons 10 nouvelles commandes dans une bibliothèque, la dixième commande peut prendre 1 ms de plus que la première pour commencer à s'exécuter. La bibliothèque 8BP contient 27 commandes et la dernière commence à s'exécuter 2 ms plus tard parce qu'elle se trouve en bas de la liste. C'est l'un des problèmes liés au fait d'être sous l'interpréteur BASIC.

Le 8BP compense ce problème en plaçant les commandes les plus fréquemment utilisées en tête de liste, et en laissant les commandes les moins fréquemment utilisées en fin de liste. Comme vous le constaterez bientôt, la commande la plus fréquemment utilisée dans 8BP est |PRINTSPALL, qui affiche tous les sprites à l'écran. Cette commande se trouve donc en tête de liste.

2.2 Fonctions du 8BP

Après avoir chargé la bibliothèque avec la commande : LOAD "8BP.BIN" et invoquer à partir de BASIC la fonction _INSTALL_RSX (définie dans le code machine) à l'aide de la commande BASIC :

APPEL &6b78

Vous disposerez des commandes suivantes, que vous apprendrez à utiliser dans ce livre

3D, <flag>, #, offsety 3D, 0	Active le mode de projection pseudo 3D.
ANIMA, #	Change l'image d'un sprite en fonction de sa séquence
ANIMALL	Modifie l'image des sprites dont le drapeau d'animation est activé (il n'est pas nécessaire de l'invoquer, un drapeau dans l'instruction PRINTSPALL suffit à l'invoquer).
AUTO, #	Déplacement automatique d'un sprite en fonction de ses Vy,Vx
AUTOALL, <flag routed>, <flag routed>, <flag routed>, <flag routed>, <flag routed>, <flag routed>.	Mouvement de tous les sprites dont le drapeau de déplacement automatique est activé
COLAY, threshold_ascii, @collision, # COLAY, @collision, # COLAY, # COLAY	Détecte les collisions avec la disposition et renvoie 1 s'il y a collision. Accepte un nombre variable de paramètres (toujours dans le même ordre) de 4 à aucun.
COLSP, #, @collided%, @collided%, @COLSP, #, @collided%, @collided%. COLSP, 32, ini, end COLSP, 33, @collided% COLSP, 34, dy, dx COLSP, #	Renvoie le premier sprite avec lequel # entre en collision. La commande peut être configurée avec les codes 32, 33 et 34.
COLSPALL,@who%, @withwho%. COLSPALL, collisionneur COLSPALL	Retourne qui est entré en collision (collider) et avec qui il est entré en collision (collided).
LAYOUT, y, x, @string\$, @string\$, @string\$, @string\$, @string\$.	Imprime une bande d'image 8x8 et remplit la carte.
LOCATESP, #, y, x	Modifie les coordonnées d'un sprite (sans l'imprimer)
MAP2SP, y, x MAP2SP, statut	Crée des sprites pour peindre le monde dans les jeux à défilement. Les sprites sont créés avec state = status
MOVER, #, dy, dx dy,dx, dy,dx, dy,dx GLOBAL	mouvement relatif d'un seul sprite
MUSIQUE, C, drapeau, chanson, vitesse MUSIQUE, drapeau, chanson, vitesse MUSIQUE	Mouvement relatif de tous les sprites dont le drapeau de mouvement relatif est actif
IPEEK, dir, @variable%	Une mélodie commence à être jouée. Le canal C peut être désactivé pour être utilisé avec des effets FX si nécessaire Aucun paramètre ne s'arrête.
POKE, dir, value	Lit une donnée de 16 bits (peut être négative)
POKE, dir, value	entrer une donnée de 16 bits (qui peut être négative)
PRINTAT, flag, y, x, @string	Imprime une chaîne de "mini-caractères" redéfinissables.
PRINTSP, #, y, x PRINTSP, # PRINTSP,32, bits	imprime un seul sprite (# est son numéro) indépendamment de l'octet de statut. Si 32 est spécifié, les bits d'arrière-plan sont activés
Le projet de loi a été adopté à l'unanimité par l'Assemblée nationale. L'impression de l'étiquette, le mode d'emploi, le mode d'emploi PRINTSPALL	Imprime tous les sprites dont le drapeau d'impression est actif. S'il est invoqué avec un seul paramètre, le mode de commande est défini.
RINK,tini,colour1,colour2,...,colourN RINK, sauter	Fait tourner un ensemble d'encre selon un modèle définissable composé d'un nombre quelconque

	d'encre.
ROUTEESP, #, étapes	Vous fait passer par N étapes de la route des sprites en une seule fois.
ROUTEALL	Modifier la vitesse du sprite avec le drapeau de chemin (pas besoin de l'invoquer, il suffit de mettre le drapeau dans AUTOALL).
SETLIMITS, xmin, xmax, ymin, ymax	Définit la fenêtre de jeu dans laquelle le découpage est effectué.
SETUPSP, #, nombre_de_paramètres, valeur SETUPSP, #, 5, Vy, Vx	Modifie un paramètre d'un sprite. Si le paramètre 5 est spécifié, Vx peut être fourni en option.
STARS, initstar, num, colour, dy, dx	Parchemin d'un ensemble d'étoiles
UMAP,adr_ini, adr_end, yini, yfin, xini, xfin	Met à jour les éléments de la carte du monde avec un sous-ensemble d'éléments d'une carte plus grande.

Tableau 2 Commandes disponibles dans la bibliothèque 8BP

Notez qu'une barre verticale apparaît au début de chaque variable car il s'agit d'une "extension" de BASIC. Certaines variables sont accompagnées du symbole "%" pour indiquer qu'il s'agit d'entiers (et non de décimales), mais si vous utilisez DEFINT pour forcer toutes les variables à être des entiers, vous n'avez pas besoin du "%".

En outre, vous disposez d'un commandement expérimental :

|RETROTIME, date

Cette commande vous permet de transformer votre CPC en machine à remonter le temps, simplement en entrant la date cible souhaitée. La seule limite de cette commande est que vous devez entrer une date égale ou postérieure à la date de naissance du CPC AMSTRAD, soit avril 1984,

|RETROTIME, "01/04/1984".

Veuillez utiliser cette fonction avec prudence. Vous pourriez créer un paradoxe temporel et détruire le monde.

Bien que vous puissiez être sceptique pour le moment quant à ce que vous pouvez faire avec la bibliothèque 8BP, vous découvrirez bientôt que l'utilisation de cette bibliothèque et des techniques de programmation avancées que vous apprendrez dans ce livre vous permettra de créer des jeux professionnels en BASIC, ce que vous auriez pu croire impossible.

Note importante pour le programmeur :

La bibliothèque 8BP est optimisée pour être très rapide. C'est pourquoi elle ne vérifie pas que vous avez défini correctement les paramètres de chaque commande, ni qu'ils ont la bonne valeur. Si un paramètre est mal défini, il est très probable que l'ordinateur se bloque lors de l'exécution de la commande. La vérification de ces éléments prend du temps d'exécution et le temps est une ressource qui ne peut être gaspillée, pas même une milliseconde.

2.3 AMSTRAD CPC Architecture

Cette section est utile pour comprendre plus tard comment la bibliothèque 8BP utilise la mémoire.

L'AMSTRAD est un ordinateur basé sur le microprocesseur Z80, fonctionnant à 4 MHz. Comme le montre le schéma de son architecture, l'unité centrale et la matrice logique vidéo (appelée "gate array") accèdent toutes deux à la mémoire vive, de sorte que pour "se relayer", les accès à la mémoire de l'unité centrale sont retardés, ce qui se traduit par une vitesse effective de 3,3Mhz. Cela représente tout de même une puissance importante.

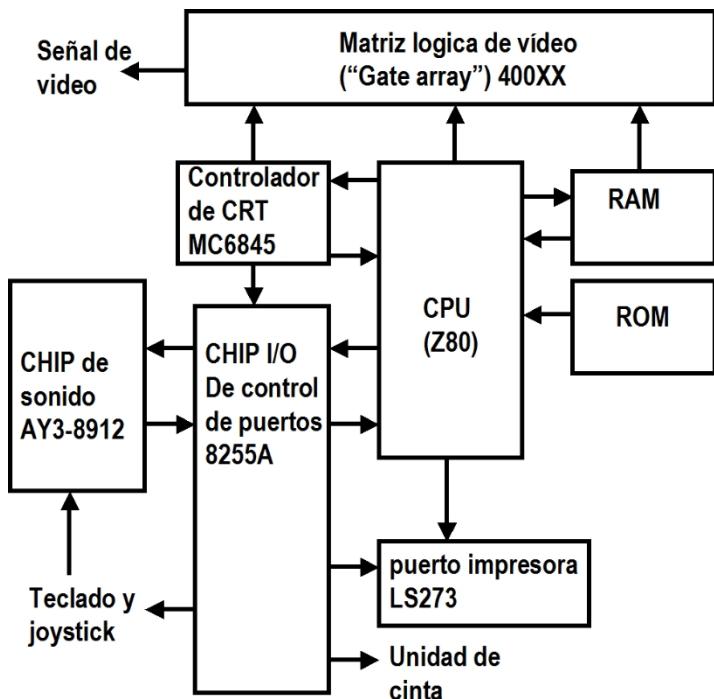


Fig. 4 Architecture de l'AMSTRAD

La RAM vidéo est accédée par la puce gate array 50 fois par seconde afin d'envoyer une image à l'écran. Dans les ordinateurs plus anciens (comme le Sinclair ZX81), cette tâche était confiée au processeur, ce qui lui enlevait encore plus de puissance.

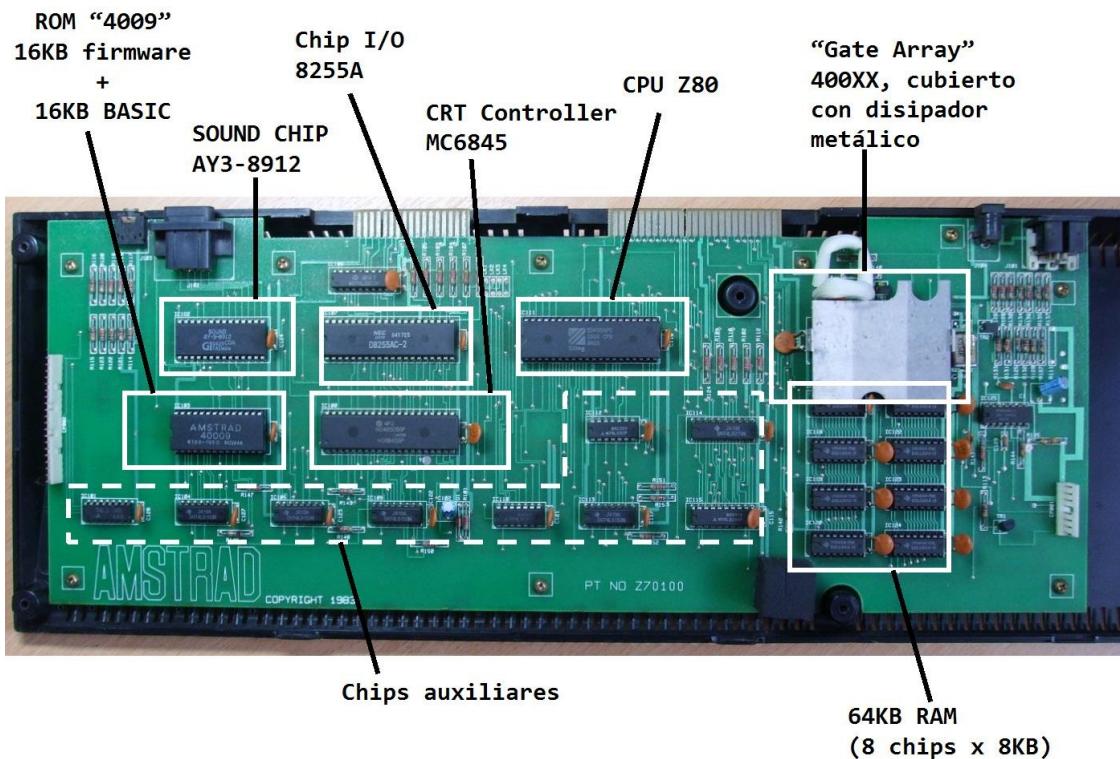


Fig. 5 Identification des composants sur la carte

Le gate array est une puce contenant de nombreuses portes logiques, conçue spécifiquement pour l'AMSTRAD. Dans le ZX Spectrum, il y a une puce similaire appelée ULA (Uncommitted Logic Array - à ne pas confondre avec ALU). L'amstrad et le ZX sont des puces conçues exclusivement pour ces ordinateurs. Sur le ZX, en plus d'être utilisé pour générer des données, l'amstrad est également utilisé pour générer des

données.

l'image vidéo était également utilisée pour lire l'entrée du clavier et de la cassette, mais sur l'AMSTRAD, ces fonctions sont assurées par d'autres puces telles que le 8255A. Le Z80 a un bus d'adresse de 16 bits, il n'est donc pas capable d'adresser plus de 64 Ko. Cependant, l'Amstrad a 64 Ko de RAM et 32 Ko de ROM. Pour les adresser, l'AMSTRAD est capable de "passer" d'une banque à l'autre, de sorte que, par exemple, si une commande BASIC est invoquée, il passe à la banque ROM où l'interpréteur BASIC est stocké, ce qui chevauche les 16 Ko d'espace d'écran. Ce mécanisme est simple et efficace.

En plus de la ROM contenant l'interpréteur BASIC de 16 Ko située dans la zone de mémoire haute, il y a une autre ROM de 16 Ko située dans la zone de mémoire basse, où se trouvent les routines du micrologiciel (ce qui pourrait être considéré comme le système d'exploitation de cette machine). Au total (interpréteur BASIC et microprogramme), ils totalisent 32 Ko.

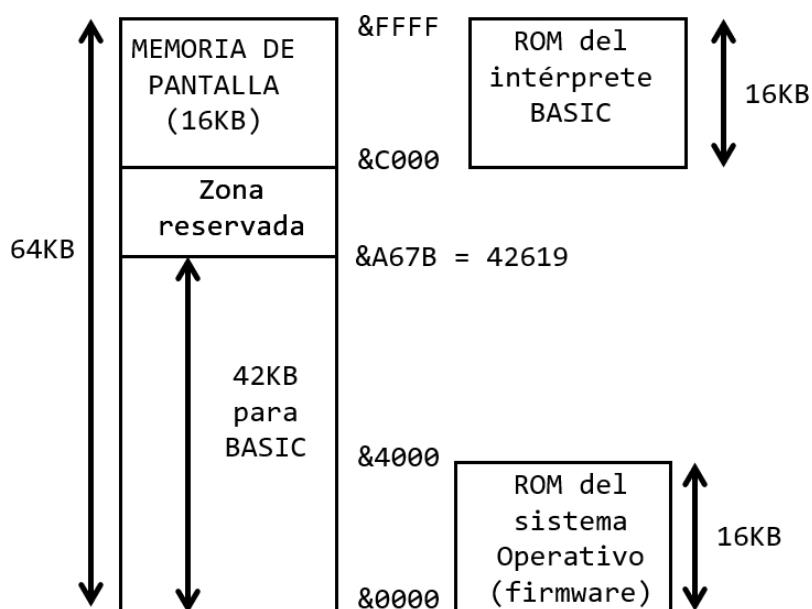


Fig. 6 Mémoire AMSTRAD

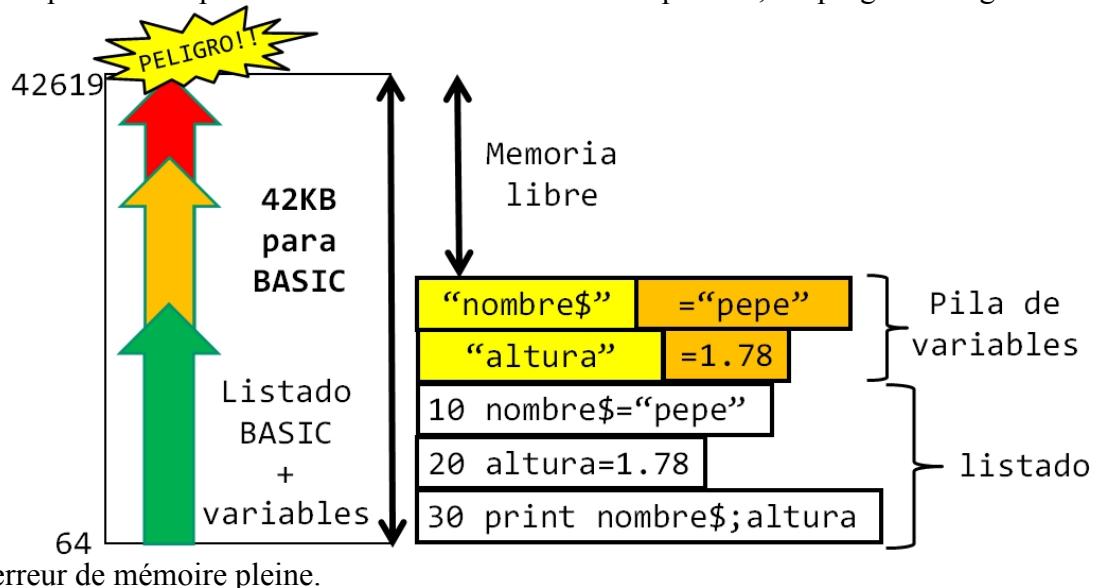
Comme le montre la carte mémoire, sur les 64 Ko de RAM, 16 Ko (de &C000 à &FFFF) sont de la mémoire vidéo. Les programmes BASIC peuvent occuper de la position &40 (adresse 64) à 42619, car au-delà il y a des variables système. Cela signifie qu'environ 42KB sont disponibles pour BASIC, comme nous pouvons le voir en imprimant la variable système HIMEM (abréviation de "High Memory").

```
print HIMEM
42619
Ready
```

Fig. 7 Variable du système HIMEM

Le fonctionnement de BASIC tient compte du stockage du programme dans des adresses croissantes à partir de la position &40. Une fois exécutées, les variables déclarées doivent occuper un espace pour stocker les valeurs qu'elles prennent et, comme elles ne peuvent pas occuper la même zone que celle où est stocké le programme, elles commencent simplement à être stockées au-dessus de la dernière adresse occupée par la liste BASIC.

Sur l'AMSTRAD, chaque variable occupe des informations avec son nom et sa valeur. Une variable numérique de type entier occupera 2 octets de mémoire pour la valeur, mais occupera autant d'octets pour stocker le nom. Les variables réelles (avec des décimales) occupent 5 octets. Chaque fois que nous utilisons une variable, nous consommons de la mémoire à partir de la première adresse libre au-dessus du listing BASIC. Si nous créons beaucoup de variables et que notre liste BASIC est très longue, la pile de variables risque de consommer toute la mémoire libre. Dans ce cas, le programme BASIC serait corrompu et cesserait de fonctionner. Dès que le programme commence à fonctionner, les variables commencent à occuper de l'espace et lorsqu'elles remplissent toute la mémoire vive disponible, le programme génère une



erreur de mémoire pleine.

Fig. 8 Croissance de la consommation de mémoire des listes et variables BASIC

Vous pouvez contrôler à tout moment la quantité de mémoire dont vous disposez avec la commande **FRE(0)**, dont vous pouvez imprimer la valeur ou la charger dans une variable. Vous pouvez faire cette expérience, vous verrez comment vous arrivez à consommer toute la mémoire du CPC avec un tableau d'entiers :

```
print fre(0)
42209
Ready
list
10 CLEAR:DEFINT a-z
20 DIM a(21099)
30 PRINT FRE(0)
Ready
run
0
Ready
```

Ce programme simple occupe toute la mémoire entre la somme de la liste BASIC et le tableau.

Remarquez que FRE(0) nous indique qu'il ne reste plus un seul octet.

Chaque fois qu'une valeur est attribuée à une variable littérale, telle que name\$, la variable est déplacée, laissant de moins en moins d'espace disponible, bien que l'espace qu'elle occupait précédemment soit marqué comme "disponible". Lorsqu'un programme manque de mémoire, il compacte toutes les variables, libérant ainsi tout l'espace "disponible". Cet effet est également obtenu en exécutant **FRE("")**. Mais attention, Amstrad BASIC ne vous permet pas d'exécuter simplement **FRE("")**, car il vous donnera une "erreur de syntaxe". Vous devez au moins assigner une variable, par exemple **p=FRE("")**. Cette commande exécute ce qui est aujourd'hui connu dans certains langages sous le nom de "garbage collection".

Si, au cours de l'exécution d'un programme, vous avez un problème de mémoire pleine, vous pouvez le résoudre simplement en exécutant périodiquement une ligne comme celle-ci :

100 c=c+1

110 if c and 63 =0 then 120 else p=fre("") :FRE every 64 cycles

120 le programme se poursuit

Cette solution ne fonctionne que si le problème est dû à une instruction qui consomme un peu plus que ce qui est libre juste avant l'exécution automatique de la FRE. Si c'est le cas, cette solution fonctionnera. Cependant, il est très "rare" de résoudre le problème de cette façon, parce que si l'Amstrad n'a pas de mémoire, il exécute le FRE automatiquement et en théorie il n'a pas besoin de le faire. Une autre solution plus simple (et plus efficace) est de supprimer ou de raccourcir les lignes REM, pour donner un peu plus de mémoire libre à l'Amstrad. S'il y a assez de mémoire et que vous avez toujours une mémoire pleine, c'est un problème de trop de GOSUB sans RETURN.

Chaque variable numérique consomme la mémoire suivante :

- le nom de la variable : N octets
- la valeur, le dernier octet étant mis à zéro pour indiquer la fin du littéral : 2 octets

Chaque variable littérale consomme la mémoire suivante :

- le nom de la variable : N octets
- adresse mémoire (ou "pointeur") où commence sa valeur : 2 octets
- la valeur, le dernier octet étant mis à zéro pour indiquer la fin de la valeur littérale : N octets

Chaque fois qu'une variable littérale est relocalisée en lui assignant une nouvelle valeur, la nouvelle valeur est chargée dans la zone de mémoire disponible et le pointeur est réassigné pour pointer sur la nouvelle adresse, laissant l'ancienne valeur non pointée par une quelconque variable. L'espace précédent ou "gap" est disponible, mais un nettoyage est nécessaire pour compacter toutes les variables et libérer tous les gaps disponibles.

Voyons un exemple simple qui montre l'espace disponible lors de la relocalisation d'une variable littérale (ou "chaîne"), qui dépense apparemment de plus en plus de mémoire, bien que la mémoire inutilisée reste disponible (il s'agit de "trous" dans la pile de variables). Si vous essayez de faire la même chose avec une variable numérique, la consommation de mémoire sera constante parce que les variables numériques occupent toujours la même place et ne se déplacent pas en cas de changement de valeur, alors que les variables littérales (ou "chaînes") changent de longueur en cas de changement de valeur.

```

10 nombre=rnd*100
20 c$=str$(nombre)
30 print fre(0)
40 goto 10

```

Comme vous pouvez le voir, à chaque itération, il y a moins de mémoire disponible, mais si vous le laissez fonctionner indéfiniment, AMTRAD exécutera une procédure de nettoyage de la mémoire inutilisée lorsqu'il sera à court de mémoire et il disposera à nouveau de mémoire. Cette procédure est la même que lorsque vous exécutez FRE(" ").

A ne pas confondre avec la commande CLEAR qui libère de l'espace en supprimant toutes les variables de la pile.

Il est conseillé d'exécuter FRE(" ") périodiquement dans votre programme, car l'opération de compactage de toutes les variables lorsque toute la mémoire a été consommée représente un travail plus important (et donc plus lent) que si vous exécutez périodiquement un FRE(" ") qui a peu de travail à faire.

Si votre programme produit une erreur de mémoire pleine après un certain temps, essayez d'exécuter périodiquement un FRE("") , supprimez les lignes "REM" pour lui donner plus de mémoire libre ou vérifiez qu'il ne s'agit pas d'un excès.

l'imbrication de GOSUB, qui est l'erreur la plus fréquente.

Ready
run
42150
42137
42124
42111
42098
42086
42073
42060
42047
42034
42021
42008
41995
41982
41969
41956
41943
41931
41918

<après de nombreuses itérations>

141
128
115
102
89
76
64
51
38
25
12
42137
42124
42111
42098
42085
42072
42060

Enfin, une curiosité : le CPC 464 vous donne un FRE(0) de 43 553 octets libres alors que le 6128 vous donne moins de mémoire, 42 249 octets. Ceci est directement lié au fait que si vous faites un PRINT HIMEM sur le CPC464 vous obtenez 43903, alors que sur le CPC6128 vous obtenez 42619.

2.3.1 GOSUB /RETURN Pile

Chaque fois que vous exécutez GOSUB dans l'Amstrad BASIC, le système doit indiquer l'endroit où il doit retourner lorsque vous faites RETURN. L'Amstrad CPC dispose d'une pile de 83 positions pour stocker les sauts. Il est courant de faire une erreur de programmation et dans certaines IF de la sous-routine de retourner avec un GOTO (c'est-à-dire de ne pas retourner), ils s'accumulent donc si vous ne faites pas attention et vous pouvez obtenir une erreur de "mémoire pleine" pendant l'exécution de votre programme.

<pre> 10 GOSUB 100 20 REM ici n'arrive jamais 100 i=i+1 110 PRINT i 120 GOTO 10 </pre>	<pre> 73 74 75 76 77 78 79 80 81 82 83 Memory full in 100 Ready </pre>	
---	--	--

Dans cet exemple, même si vous imprimez la mémoire restante en remplaçant la ligne 110 par :

110 print i : print fre(0)

Vous verrez que la mémoire disponible ne diminue pas et pourtant, même si vous avez presque toute la mémoire libre, l'Amstrad indique memory full. Ce cas est dû à l'imbrication des **GOSUB**, et non pas au fait qu'il n'y a pas de mémoire libre. **Seuls 83 GOSUB sont autorisés sans RETURN.**

Je ne peux pas vous dire avec certitude quelle est la zone de mémoire où l'interpréteur BASIC de l'Amstrad stocke les 83 adresses pour RETURN, mais c'est probablement dans la zone de mémoire système (6KB de 42619 à 49152). Il s'agit d'une zone de 6 Ko juste avant la mémoire écran (49152 est &C0000) où BASIC stocke les caractères réinscriptibles, ainsi que la "pile". Il s'agit d'une zone utilisée pour stocker l'adresse mémoire où se trouve un programme avant de passer à une routine, afin qu'il puisse revenir à l'endroit où il se trouvait à la fin de la routine. Elle est utilisée en bas niveau (langage d'assemblage). La pile augmente vers des adresses plus basses au fur et à mesure qu'elle stocke des adresses (c'est-à-dire qu'elle commence à 49152 et prend des adresses de plus en plus petites) et lorsqu'elle revient d'une routine, elle diminue à nouveau. On pourrait penser que, si une routine appelle une autre routine et que cette routine appelle une autre routine et ainsi de suite, la pile croîtrait tellement qu'elle quitterait la zone "système" et envahirait d'autres zones, mais ne vous inquiétez pas, le 8BP garde la pile sous contrôle et l'Amstrad BASIC aussi.

2.3.2 Une expérience pour voir la ROM avec Winape

Vous pouvez utiliser Winape pour voir ce que contient la ROM de l'interpréteur BASIC qui chevauche les adresses de l'écran, et de même pour la ROM du système d'exploitation. Dans la fenêtre d'émulation, tapez

POKE &C000, &FF

Ensuite, appuyez sur pause sur l'émulateur et vous verrez un écran avec les adresses et les valeurs de la mémoire. Recherchez l'adresse &C000 et son contenu. Il devrait s'agir de &FF. En bas de l'écran, vous verrez quelque chose comme :

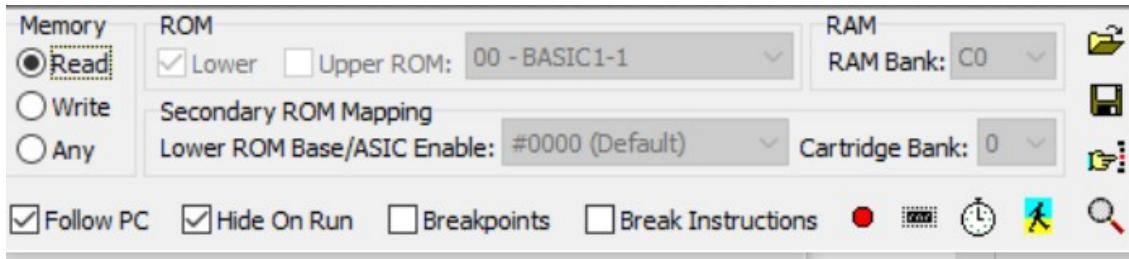


Fig. 9 Winape montrant la RAM

Eh bien, si l'on procède de la manière suivante :

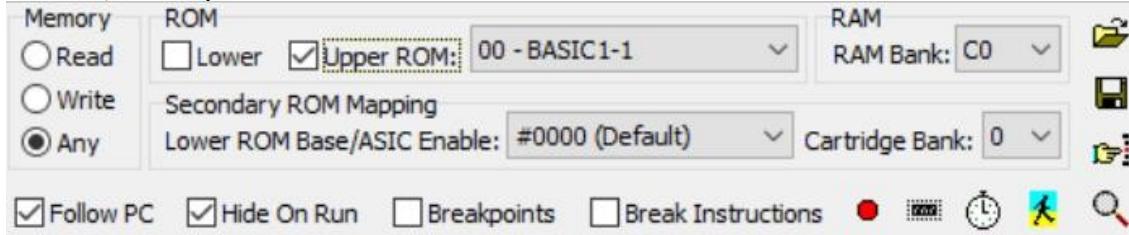


Fig. 10 Winape montrant la ROM

Vous verrez le contenu de l'adresse &C000 dans la banque ROM contenant l'interpréteur BASIC. Il est très intéressant de pouvoir faire cela avec Winape.

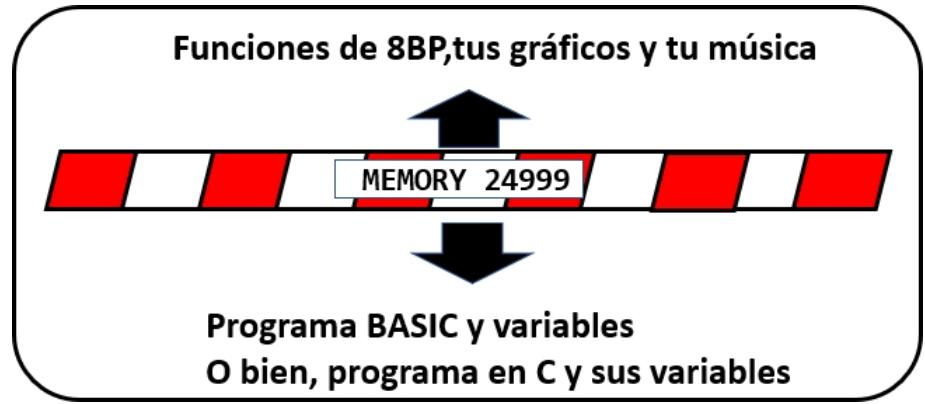
2.4 Carte mémoire du 8BP et options d'assemblage

Comme le BASIC de l'Amstrad commence par consommer les adresses les plus basses, la bibliothèque 8BP est chargée dans la zone de mémoire la plus élevée afin de permettre la "coexistence". Il est important de comprendre le fonctionnement du BASIC pour pouvoir utiliser la bibliothèque, car vous devrez utiliser la commande BASIC "**MEMORY**".

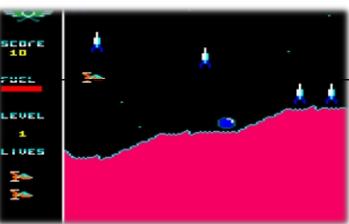
```
10 a=5
20 k=@a
30 PRINT k
Ready
run
411
Ready
```

Le texte d'un programme écrit en BASIC est stocké à partir de l'adresse &40 (en décimal c'est 64, une adresse très "basse") et les variables que votre programme crée sont stockées en occupant des positions juste après la mémoire occupée par la liste. L'exemple suivant montre où est stockée la variable "A", qui se trouve être à l'adresse 441.

La liste et les variables BASIC peuvent devenir très grandes si votre programme est très volumineux, et pourraient envahir la zone de mémoire où sont stockées les routines 8BP. Pour éviter cela, vous devez exécuter une instruction **MEMORY** qui protège la zone de mémoire où sont stockés le 8BP, vos graphiques et votre musique. La commande **MEMORY** est comme une "**barrière**" qui empêche BASIC et ses variables d'occuper la mémoire au-delà de la limite que nous lui indiquons. Si c'est le cas, il affichera une erreur "**MEMOIRE PLEINE**" et cessera de fonctionner.



La bibliothèque 8BP vous laisse entre 24 et 25 KB libres pour votre listing BASIC ou votre programme compilé en C (dans 8BP vous pouvez aussi programmer en C). A partir de la version V42 de 8BP, il est possible de choisir plusieurs options d'assemblage en fonction du type de jeu que vous allez développer. L'idée est simple : si vous voulez faire un jeu avec défilement, vous avez besoin des commandes qui traitent du défilement, mais vous n'avez pas besoin des commandes qui traitent du dessin des labyrinthes (LAYOUT dans 8BP) ou de la détection des collisions avec les murs du labyrinthe. En d'autres termes, à partir de la version 42, certaines fonctions 8BP n'utiliseront pas de mémoire si elles ne sont pas nécessaires. Le 8BP peut donc vous laisser plus de mémoire libre pour votre listing BASIC ou votre programme C. Vous devrez choisir une option d'assemblage en fonction du type de jeu que vous voulez programmer. Ce tableau résume les options disponibles. Vous comprendrez vite la commande SAVE qui apparaît dans chaque description. Ce tableau est également disponible en annexe

Option	Description de l'option	Exemple de jeu typique
0	<p>Vous pouvez faire n'importe quel jeu Toutes les commandes sont disponibles Vous devez utiliser MEMOIRE 23499</p> <p>Pour sauvegarder la bibliothèque + les graphiques + la musique :</p> <p>SAVE "8BP0.bin",b,23500,19119</p>	n'importe qui
1	<p>jeux de labyrinthe ou de passage d'écran Vous devez utiliser la MEMOIRE 24999</p> <p>Non disponible dans ce mode :</p> <p> MAP2SP, UMAP, 3D</p> <p>Pour sauvegarder la bibliothèque + les graphiques + la musique :</p> <p>SAVE "8BP1.bin",b,25000,17619</p>	
	<p>Pour les jeux à défilement</p> <p>Vous devez utiliser MEMOIRE 24799 Non disponible dans ce mode :</p> <p> LAYOUT, COLAY, 3D</p> <p>Pour sauvegarder la bibliothèque + les graphiques + la musique :</p> <p>SAVE "8BP2.bin", b,24800,17819</p>	

Pour les jeux en pseudo 3D, vous devez utiliser **MEMOIRE 23999**
Non disponible dans ce mode :
|LAYOUT, |COLAY
Pour sauvegarder la bibliothèque + les graphiques + la musique :
SAVE "8BP3.bin", b,24000,18619



Comme vous pouvez le constater, l'option qui vous laisse le plus de mémoire libre pour votre programme est l'option 1, car elle vous laisse 25 Ko de mémoire libre pour votre jeu. L'option 3 vous laisse 24 Ko et l'option zéro vous laisse 23,5 Ko. Je ne vous recommande pas d'utiliser l'option zéro, sauf si vous en avez vraiment besoin. Il est préférable de choisir la bonne option pour votre jeu et vous aurez plus de mémoire disponible.

Ces 24-25 KB libres sont destinés à votre listing, car la musique et les graphiques de votre jeu sont stockés dans une autre zone "supérieure". Par exemple, avec l'option 1, vous avez :

- 25 KB gratuits pour votre listing (BASIC ou C) et vos variables
- 1.5 KB gratuit pour votre musique
- 8.5 KB gratuit pour vos graphiques

TOTAL : 35 KB libres pour votre jeu

L'option d'assemblage que vous choisissez est définie dans un fichier 8BP appelé "make_all_mygame.asm". Ce fichier contient une ligne que **vous devez éditer pour choisir l'option que vous préférez**.

```

; Makefile para los videojuegos que usan 8bits de poder
; si alteras solo una parte solo tienes que ensamblar el make correspondiente
; por ejemplo puedes ensamblar el make_graficos si cambias dibujos

; DESDE LA V42 EXISTEN "OPCIONES" DE ENSAMBLAJE
; -----
; ASSEMBLING_OPTION = 0 --> todos los comandos disponibles.

; ASSEMBLING_OPTION = 1 --> para juegos de laberintos. MEMORY 25000
;                               disponibles los comandos |LAYOUT, |COLAY
;

; ASSEMBLING_OPTION = 2 --> para juegos con scroll, MEMORY 24800
;                               disponibles los comandos |MAP2SP, |UMA
;

; ASSEMBLING_OPTION = 3 --> para juegos pseudo 3D , MEMORY 24000
;                               disponible comando |3D
;

; ASSEMBLING_OPTION = 4 --> uso futuro

let ASSEMBLING_OPTION = 1
;-----CODIGO-----
;incluye la libreria 8bp y el playerWYZ de musica
read "make_codigo_mygame.asm"

;-----MUSICA-----
; incluye las canciones.
read "make_musica_mygame.asm"

; ----- GRAFICOS -----
; esta parte incluye imagenes y secuencias de animacion
; y la tabla de sprites inicializada con dichas imagenes y secuencias
read "make_graficos_mygame.asm"

```

Avant de charger la bibliothèque, vous devez exécuter la commande **MEMORY** avec la limite associée au mode d'assemblage que vous avez choisi pour votre type de jeu. A partir de la version V42, il y a 24KB, 24.8 KB ou 25KB de libre en fonction de l'option d'assemblage que vous avez choisie.

Quelle que soit l'option d'assemblage choisie, les adresses mémoire où se trouvent toutes

les commandes sont exactement les mêmes (annexe XI). Dans le cas d'un

Par exemple, la commande LAYOUT peut être invoquée si vous choisissez l'option d'assemblage 1 ou 2, mais si vous utilisez l'option d'assemblage 2, l'invocation de la commande LAYOUT ne fera absolument rien, tout comme la commande MAP2SP si vous utilisez l'option d'assemblage 1.

```

AMSTRAD CPC464 8BP MEMORY MAP

;
; &FFFF +-----+
; | affichage + 8 segments cachés de 48 octets chacun
; &C000 +-----+
; | système (symboles redéfinissables, pointeur de pile,
etc.) -----
; 42619 +
; | -banque-de 40 étoiles (de 42540 à 42619 = 80bytes)
; 42540 +
; | Carte de disposition des caractères (25x20 =500 octets)
; | et carte du monde (jusqu'à 82 éléments peuvent tenir dans 500
octets)
; | -Les-deux-sont stockés dans la même zone de mémoire.
; | sprites(tiles) 18Ko de plus
; 42040 dessins avec 8440 octets s'il n'y a pas de séquences et
; | pas de routes)
; +-----Les images de l'alphabet sont également stockées ici.
; | définitions d'itinéraires (d'une longueur variable)
; +-----+
; | Séquences d'animation de 8 images (16 octets chacune)
; | les séquences d'animation et les groupes de séquences
; 33600 d'animation (macro-séquences)
; | chansons
; | (1500 Bytes pour la musique éditée avec WYZtracker
; 32100 +-----2.0.1.0)
; | Routines 8BP(8100 octets ou 7100 octets)
; | voici toutes les routines et la table des sprites
; | inclut le lecteur de musique "wyz" 2.0.1.0
; 25000 +-----+
; |
; | VOTRE LISTE DE BASE ou C
; | 24 Ko, 24,8 Ko ou jusqu'à 25 Ko libres pour BASIC ou C en
fonction de l'option d'assemblage que vous utilisez pour
; | 8BP
; |
; 0 +-----+

```

Fig. 11 Mémoire utilisant 8BP

Si vous manquez d'espace pour les graphiques ou la musique et que vous avez besoin de plus d'espace, le 8BP vous permet de placer des images et de la musique dans d'autres zones (en dessous de l'adresse 24000) et cela fonctionnera très bien.

3 Outils nécessaires

Winape : émulateur pour Windows OS avec éditeur pour éditer et tester votre programme BASIC. Il permet également d'assembler des graphiques et de la musique.

SPEDIT : ("Simple Sprite Editor") Outil BASIC pour éditer vos graphiques. La sortie de spedit est un code assembleur qui est envoyé à l'imprimante Amstrad CPC. En exécutant l'outil dans Winape, l'imprimante est redirigée vers un fichier texte de sorte que vos graphiques seront stockés dans un fichier txt. Cet outil a été créé pour compléter la bibliothèque 8BP et les graphiques de tous les jeux que j'ai créés ont été réalisés avec SPEDIT.

Wyztracker : pour composer de la musique, sous Windows. Le programme capable de jouer les mélodies composées par Wyztracker est Wyzplayer, qui est intégré à 8BP. Après avoir assemblé la musique, vous pouvez la faire jouer avec une simple commande |MUSIC

Bibliothèque 8BP : installe de nouvelles commandes accessibles à partir de BASIC pour votre programme. Comme vous le verrez, il s'agit du "cœur" de la machine que vous construisez.

CPCDiskXP : permet d'enregistrer une disquette 3,5" que vous pourrez ensuite insérer dans votre CPC6128 si vous disposez d'un câble pour connecter un lecteur de disquette. Si vous voulez faire une cassette audio CPC464, cet outil n'est pas nécessaire.

2CDT : outil essentiel pour créer des fichiers .cdt. J'extrais généralement les fichiers d'un fichier .dsk sur le disque Windows à l'aide de CPCDiskXP, puis j'utilise 2cdt pour créer le fichier .cdt.

Tape2wav : outil permettant de créer des fichiers .wav à partir de fichiers .cdt

EN OPTION :

ConvImgCPC : éditeur d'images pour vos jeux. Convertit également les BMP. Programmé par Ludovic Deplanque ("DEMONIAK")

RGAS : (Retro Game Asset Studio) puissant éditeur de sprites, développé à partir de l'outil AMSprite, créé par Lachlan Keown. Cet éditeur de sprites est compatible 8BP et fonctionne sous Windows. Lorsque vous ne pourrez plus utiliser Spedit, ce sera peut-être la meilleure option.

DÉCONSEILLÉ :

Fabacom : compilateur exécutable à l'intérieur de l'AMSTRAD CPC 6128 ou à partir de l'émulateur Winape pour compiler votre programme BASIC et le rendre plus rapide. Il est compatible avec les appels de commande de la bibliothèque 8BP. Cependant, il n'est pas recommandé pour plusieurs raisons :

- Votre programme occupera beaucoup plus d'espace parce que fabacom a besoin

de 10 Ko supplémentaires pour ses bibliothèques, et aussi, une fois qu'il aura compilé votre programme, il occupera toujours la même quantité d'espace.

de sorte qu'un programme BASIC de 10 Ko se transforme en un programme BASIC de 20 Ko.

- Il existe des problèmes documentés d'incompatibilité de ce compilateur avec certaines instructions BASIC.
- De plus, comme vous le verrez tout au long de cet ouvrage, il est possible d'atteindre une vitesse très élevée sans compiler.

Compilateur CPC BASIC : compilateur exécutable pour Windows. Il est compatible avec les appels de commande de la bibliothèque 8BP. Contrairement à fabacom, le programme compilé n'occupe que 5 Ko supplémentaires, mais il réserve 16 Ko de travail à exécuter, de sorte qu'il ne laisse pratiquement pas d'espace pour votre programme, puisqu'il "vole" 20 Ko au total. De plus, il n'est pas compatible à 100% avec le BASIC de la locomotive.

Le gain de vitesse avec fabacom et le compilateur CPC Basic peut aller jusqu'à 50 %, selon le jeu. C'est-à-dire qu'un jeu tournant à 20 FPS tournera à 30 FPS. Ce n'est pas mal, mais pensez que nous sommes passés d'un BASIC interprété à un code machine et normalement on dit que la vitesse doit être multipliée par au moins 100 (nous parlerions d'une augmentation de 10000%). Or, nous n'avons gagné que 50 %. La raison d'un gain aussi "faible" est que les instructions 8BP font déjà tout le travail et que le compilateur ne traduit en code machine que la partie la moins lourde, la logique du jeu.

Enfin, vous devez savoir que, si vous souhaitez que vos programmes s'exécutent beaucoup plus rapidement, depuis la version 40 du 8BP, le langage C est pris en charge. Vous pouvez donc soit programmer votre jeu entier directement en C, soit programmer en BASIC et traduire uniquement le "cycle du jeu" en C. Un chapitre de ce manuel est exclusivement consacré à ce sujet. Un chapitre de ce manuel est consacré exclusivement à ce sujet.

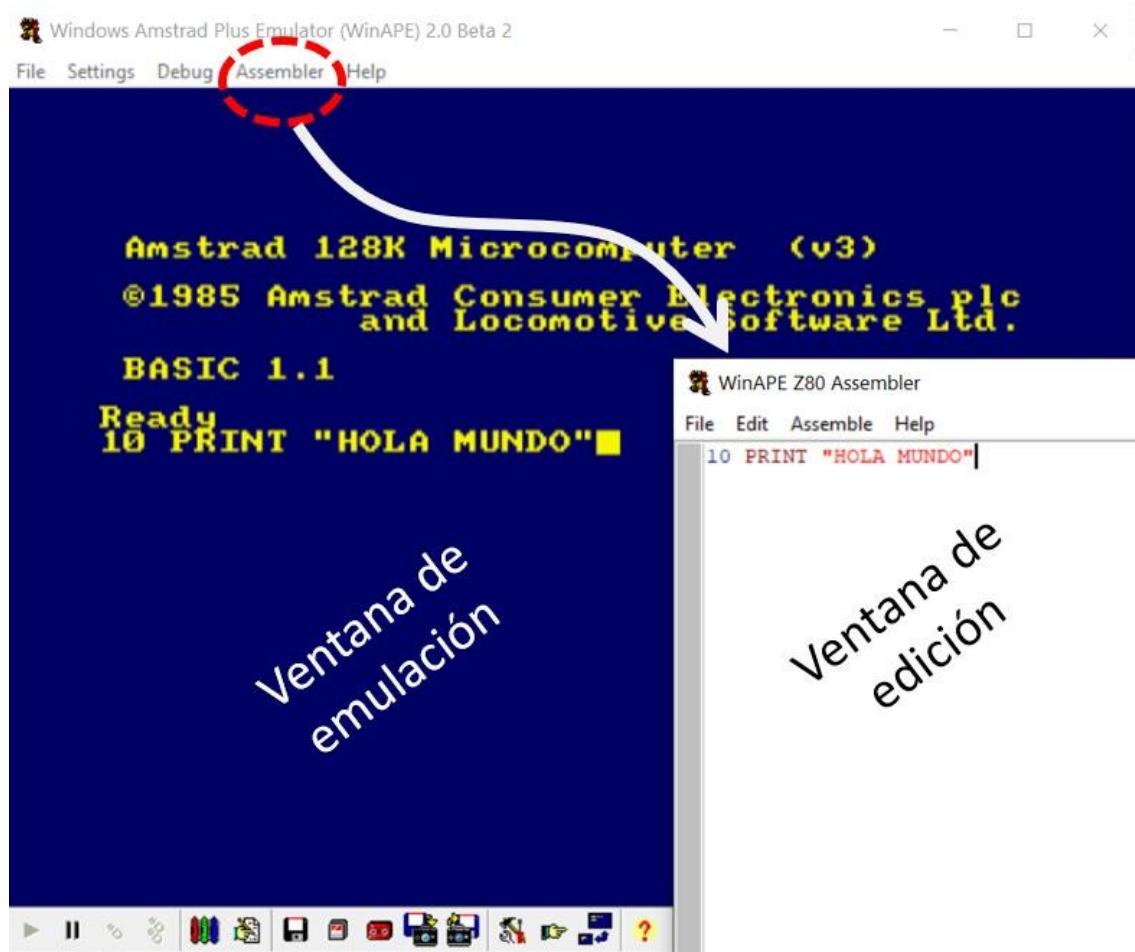
4 Premiers pas avec le 8BP

4.1 Installer winape

La première chose à faire est d'installer la dernière version de **Winape**, qui est un émulateur, un éditeur et un assembleur Amstrad. Vous pouvez le télécharger à partir de www.winape.net

4.2 Se familiariser avec winape : "hello world".

Une fois Winape installé, familiarisez-vous avec lui en essayant quelques jeux Amstrad et en essayant de modifier la configuration. Essayez d'ouvrir le menu assembleur intégré et éditez un "hello world" dans la fenêtre assembleur. Copiez et collez ensuite le texte dans la fenêtre d'émulation. Vous verrez comment il est collé caractère par caractère

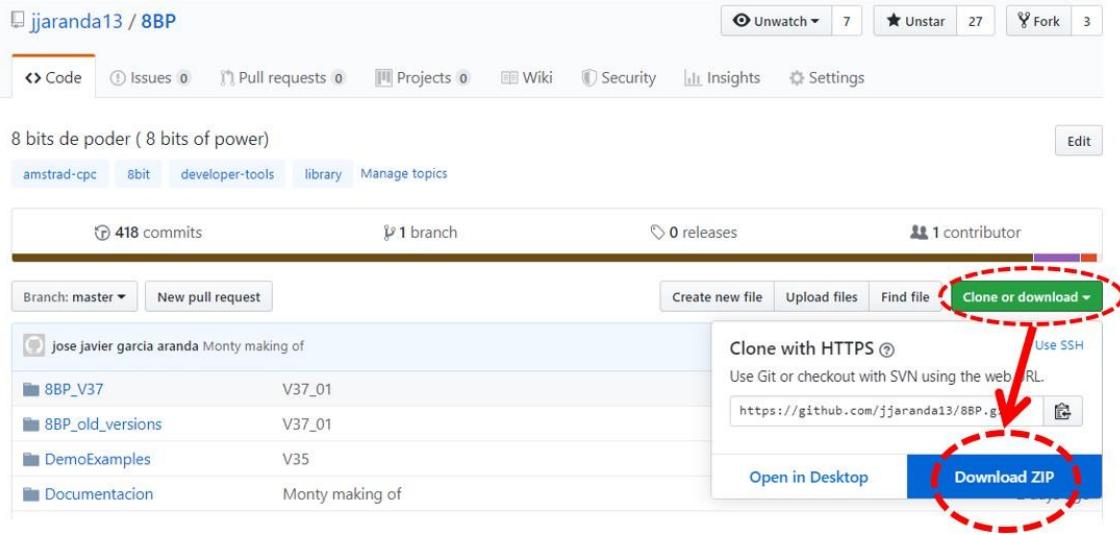


Si vous créez un programme plus long, pour le copier dans la fenêtre d'émulation, il est très intéressant d'utiliser l'option paramètres->haute vitesse. Vous verrez à quelle vitesse il est copié.

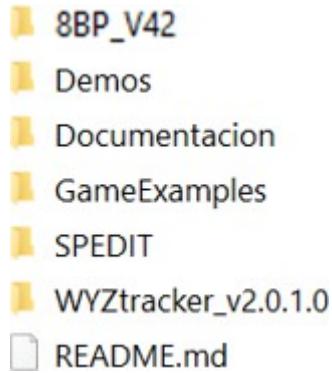
4.3 Télécharger la bibliothèque 8BP

Nous arrivons à la partie intéressante. Allez à l'url <https://github.com/jjaranda13/8BP> et téléchargez le zip (<https://github.com/jjaranda13/8BP/archive/master.zip>) . Pour ce faire

vous pouvez simplement cliquer sur le bouton vert "cloner ou télécharger" et ensuite sélectionner le fichier zip



Une fois que vous l'avez téléchargé, décompressez-le dans le répertoire de votre choix. Vous obtiendrez le résultat suivant :



4.4 Exécuter les démonstrations

Nous allons maintenant prendre un premier contact avec le 8BP en regardant quelques démos. Allez dans le répertoire Demos. Vous y trouverez une série de sous-dossiers : **ASM, BASIC, C, DSK, MUSIC, ASM, BASIC, C, DSK, MUSIC**.

Allez dans DSK . Vous y trouverez un fichier .dsk avec les démos. Dans winape, allez dans le menu File.

>Lecteur A-> Insérer l'image du disque et sélectionner le fichier de démonstration
Une fois sélectionné, dans la fenêtre d'émulation Amstrad, tapez CAT pour voir les fichiers.

cat

Drive A: user 0

8BP0	.BIN	18K	DEMO15	.BAS	2K
8BP1	.BIN	18K	DEMO2	.BAS	2K
8BP2	.BIN	19K	DEMO3	.BAS	1K
CICLO	.BIN	4K	DEMO4	.BAS	2K
DEMO1	.BAS	2K	DEMO5	.BAS	2K
DEMO10	.BAS	2K	DEMO6	.BAS	1K
DEMO11	.BAS	1K	DEMO7	.BAS	2K
DEMO11	.BIN	1K	DEMO8	.BAS	1K
DEMO12	.BAS	3K	DEMO9	.BAS	1K
DEMO13	.BAS	2K	LOADER	.BAS	2K
DEMO14	.BAS	3K			

89K free

Ready

Chaque fichier .BAS est une démo dans laquelle vous pouvez voir certaines des fonctionnalités du 8BP (toutes les fonctionnalités ne peuvent pas être vues dans les démos, mais il y en a quelques unes qui sont représentatives).

Exécutez la commande **RUN "LOADER.BAS"**. Vous obtiendrez le menu suivant :

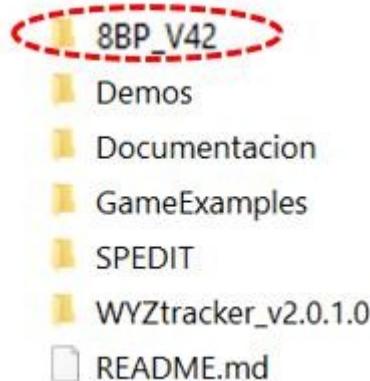


Vous pouvez maintenant choisir une démo et l'essayer. Amusez-vous bien. Dans la prochaine et dernière étape, nous allons commencer la création d'un jeu.

4.5 Créer son premier programme avec la 8BP

Nous avons testé un fichier .dsk contenant de nombreuses démos, avec des graphiques et de la musique. Les répertoires **Demos/ASM** et **Demos/MUSIC** contiennent les graphiques et la musique des démos que vous avez testées. Toutefois, si vous souhaitez créer votre propre jeu ou démo, il est préférable de commencer par des fichiers "propres" sans tous les graphiques requis par les démos que vous avez testées.

Nous allons dans le répertoire racine. Vous y trouverez un dossier appelé "**8BP_V42**". Je vous recommande de faire une copie de ce dossier et de le renommer "**my_game**". De cette façon, vous conserverez le dossier original "**8BP_V42**", même si vous commencez à changer des choses.



Dans le dossier "**8BP_V42**", vous trouverez les dossiers ASM, BASIC, DSK, MUSIC, TAPE et output_spedit.

Dans la fenêtre de l'assembleur winape, ouvrez le fichier "**ASM/make_all_mygame.asm**" et lancez-le (dans le menu de l'assembleur z80 de winape, sélectionnez simplement "Assemble" ou appuyez sur Ctrl+F9). Cela commencera à assembler (copier en mémoire) la bibliothèque et les graphiques dans la mémoire de l'Amstrad. Dans ce cas, nous allons assembler très peu de graphiques, seulement ceux qui sont essentiels pour un petit jeu. Les graphiques se trouvent dans le fichier "**images_mygame.asm**".

Après avoir tout assemblé, vous obtiendrez un message comme celui ci-dessous :

```

; Makefile para los videojuegos que usan 8bits de poder
; si alteras solo una parte solo tienes que ensamblar el make correspondiente
; por ejemplo puedes ensamblar el make_graficos si cambias dibujos

; DESDE LA V42 EXISTEN "OPCIONES" DE ENSAMBLAJE
; -----
; ASSEMBLING_OPTION = 0 --> todos los comandos disponibles.

; ASSEMBLING_OPTION = 1 --> para juegos de laberintos. MEMORY 1
;                   disponibles los comandos |LAYOUT

; ASSEMBLING_OPTION = 2 --> para juegos con scroll, MEMORY 2
;                   disponibles los comandos |MAP2SP

; ASSEMBLING_OPTION = 3 --> para juegos pseudo 3D , MEMORY 2
;                   disponible comando |3D

; ASSEMBLING_OPTION = 4 --> uso futuro

let ASSEMBLING_OPTION = 0
;-----CODIGO-----
;incluye la libreria 8bp y el playerWYZ de musica
read "make_codigo_mygame.asm"

;-----MUSICA-----
; incluye las canciones.
read "make_musica_mygame.asm"

; ----- GRAFICOS -----
; esta parte incluye imagenes y secuencias de animacion
; y la tabla de sprites inicializada con dichas imagenes y secuencias
read "make_graficos_mygame.asm"

```

Appuyez sur "ok" et à partir de la fenêtre de l'assembleur, nous allons ouvrir un autre fichier. Dans ce cas, nous allons ouvrir un fichier BASIC, plus précisément "**your_first_game.bas**" qui se trouve dans le dossier BASIC. Après l'avoir ouvert, vous verrez à l'écran la liste suivante, qui contient 32 lignes :

```

WinAPE Z80 Assembler
File Edit Assemble Help
10 MEMORY 23499:' ASSEMBLING OPTION =0
20 MODE 0: DEFINT A-Z: CALL &6B78:' install RSX
21 ENT 1,10,100,3
30 ON BREAK GOSUB 320
40 CALL &BC02:'restaura paleta por defecto por si acaso
50 INK 0,0:'fondo negro
60 FOR j=0 TO 31:|SETUPSP,j,0,0:NEXT:|3D,0:'reset sprites
70 |SETLIMITS,0,80,0,124: ' establecemos los limites de la pantalla de juego
80 PLOT 0,74*2:DRAW 640,74*2
90 x=40:y=100:' coordenadas del personaje
100 PRINT "SCORE:"
110 |SETUPSP,31,0,1+32:' status del personaje
120 |SETUPSP,31,7,1'secuencia de animacion asignada al empezar
130 |LOCATESP,31,y,x:'colocamos al sprite (sin imprimirla aun)
140 |MUSIC,1,0,0,5:puntos=0
150 cor=32:cod=32:|COLSPALL,@cor,@cod:' configura comando de colision
160 |PRINTSPALL,0,0,0,0: 'configura comando de impresion
170 '--- ciclo de juego ---
180 c=c+1
190 ' lee el teclado y posiciona al personaje
191 IF INKEY(27)=0 THEN IF dir<>0 THEN |SETUPSP,31,7,1:dir=0 ELSE |ANIMA,31:x=x+1
192 IF INKEY(34)=0 THEN IF dir<>1 THEN |SETUPSP,31,7,2:dir=1 ELSE |ANIMA,31:x=x-1
195 |LOCATESP,31,y,x
200 |AUTOALL:|PRINTSPALL
210 |COLSPALL
220 IF cod<32 THEN BORDER 7:SOUND 4,638,30,15,0,1:puntos=puntos-1:|SETUPSP,cod,0,
230 IF c MOD 20=0 THEN puntos=puntos+10 :LOCATE 7,1:PRINT puntos
240 IF c MOD 5=0 THEN |SETUPSP,i,9,19:|SETUPSP,i,5,4,RND*3-1:|SETUPSP,i,0,11:|LOC
250 IF c <1000 GOTO 180
310 '---fin del juego---
320 |MUSIC: INK 0,0:PEN 1:BORDER |p

```

Sélectionnez tout et copiez-le. Ensuite, allez dans la fenêtre d'émulation du CPC et collez-la en utilisant le menu **FICHIER->Coller**.

Comme la liste est un peu plus longue que "hello world", utilisez le menu winape et sélectionnez **settings->high speed** pour la copier, puis revenez à "normal speed". Comme la bibliothèque et les graphiques sont déjà assemblés, vous pouvez RUN et le jeu se lancera. Vous devez esquiver les balles qui tombent du ciel pour ne pas mourir, en vous déplaçant à gauche et à droite.



Vous pouvez essayer de modifier le programme et d'en voir les effets. Petit à petit, vous apprendrez le 8BP et vous pourrez faire des modifications intéressantes comme changer la fréquence de sortie des balles ennemis ou leur vitesse, ou remplacer le soldat par un vaisseau spatial et les balles par des vaisseaux ennemis avec des trajectoires différentes.

Si vous voulez tester la rapidité du jeu en langage C, il vous suffit de charger le fichier .dsk et d'exécuter "loader.bas", qui chargera une version du jeu vous permettant de choisir entre la version BASIC et la version C afin de pouvoir comparer. Un chapitre de ce livre est consacré à la programmation en C à l'aide du 8BP et d'un mini BASIC afin que vous puissiez programmer en C comme vous le feriez en BASIC. Si vous n'avez pas beaucoup d'expérience en programmation C, je vous recommande d'y aller doucement et de programmer en BASIC, les résultats seront rapides et professionnels.

4.6 Créez votre .dsk avec votre jeu de 8BP

Enfin, nous allons créer un disque avec votre jeu. Pour ce faire, après avoir lancé le jeu, vous devez suivre les étapes suivantes

- Créer un nouveau disque via winape : FICHIER-> lecteur A-> nouveau disque Blanc
- Formatez-le : FICHIER->lecteur A->Formater l'image du disque
- Après avoir créé votre fichier dsk, exécutez les commandes suivantes dans la fenêtre d'émulation :

SAVE "8BP0.bin", b, 23499,19119

SAUVEGARDE "juego.bas"

La première commande permet de sauvegarder la bibliothèque 8BP, les graphiques et la musique sur le disque. Dans ce cas, nous utilisons l'option d'assemblage 0 (voir les options dans le chapitre précédent), mais vous pouvez parfaitement utiliser n'importe quelle option pour cet exemple. J'ai nommé le fichier "**8BP0**" pour indiquer d'une certaine manière que l'option d'assemblage 0 a été utilisée, mais le nom de ce binaire peut être n'importe quoi.

Nous avons presque terminé. Vous devez maintenant sélectionner un autre disque dans le menu winape ou quitter winape pour que le fichier .dsk que vous avez créé devienne une réalité dans le système de fichiers de Windows.

Quittez winape et rouvrez-le.

Sélectionnez le disque que vous avez créé et exécutez :

MEMOIRE 23499

CHARGER "8BP0.BIN"

**Exécuter "game.bas" Exécuter "game.bas" Exécuter "game.bas" Exécuter
"game.bas" Exécuter "game.bas" Exécuter**

Alléluia !

5 Étapes à suivre pour créer un jeu

5.1 Structure des répertoires de votre projet

Lors de la programmation de votre jeu, il est recommandé de structurer les différents fichiers dans 7 dossiers, en fonction du type de fichier.

Il est tout à fait possible de tout mettre dans le même répertoire et de travailler sans dossiers, mais il est plus "propre" de le faire de la manière que je vais présenter ci-

- 📁 ASM
- 📁 BASIC
- 📁 C
- 📁 dsk
- 📁 MUSIC
- 📁 output_spedit

dessous.

Fig. 12 Structure du répertoire

- **ASM :** vous y placerez des fichiers texte écrits en assembleur (.asm), tels que la bibliothèque 8BP elle-même, les sprites générés avec l'éditeur de sprites SPEDIT et quelques fichiers auxiliaires.
- **BASIC :** vous y placerez votre jeu et des utilitaires tels que SPEDIT et Loader.
- **C :** ce dossier est destiné aux utilisateurs "avancés" qui souhaitent programmer en C l'ensemble du jeu ou au moins le cycle du jeu. Il n'est nécessaire que si vous allez programmer une partie de votre jeu en langage C.
- **Dsk :** c'est ici que vous placerez le fichier .dsk prêt à fonctionner sur un Amstrad CPC. A l'intérieur, vous devrez placer 5 fichiers dont nous parlerons dans la section suivante
- **Musique :** avec le séquenceur musical WYZtracker, vous pouvez créer vos chansons et les stocker au format .wyz dans ce répertoire. Une fois que vous les aurez "exportées", un fichier .asm sera généré que vous devrez sauvegarder dans le dossier ASM et un fichier binaire que vous stockerez également dans le dossier ASM (vous pouvez également les laisser dans ce dossier, à condition de les référencer correctement dans le fichier make_musica.asm dans le dossier ASM).
- **Output_spedit :** dans ce dossier, vous pouvez stocker le fichier texte généré par spedit. SPEDIT envoie les sprites à l'imprimante en format assembleur et l'émulateur winape peut collecter la sortie de l'imprimante Amstrad dans un fichier. Nous le placerons ici
- **Tape :** ici vous pouvez stocker le fichier .wav si vous voulez faire une cassette à charger dans l'Amstrad CPC464, ou le fichier .cdt.

5.2 Votre jeu en 3 fichiers seulement

Pour générer votre jeu, vous aurez besoin d'un fichier binaire et de deux fichiers BASIC. Vous pouvez générer plusieurs fichiers binaires indépendants (un avec la bibliothèque 8BP, un avec les graphiques, un avec la musique...) mais comme ces zones de mémoire sont contiguës, il est préférable de générer un seul fichier binaire.

Le fichier binaire contient la bibliothèque, la musique, les graphiques, la zone mémoire de la carte du monde et éventuellement la banque d'étoiles. L'idée est de stocker tous les composants binaires dans un seul fichier, comme ceci (en fonction de l'option d'assemblage, vous utiliserez l'une ou l'autre de ces commandes). Le nom du fichier se termine par un numéro indiquant l'option d'assemblage, mais il peut s'appeler comme vous le souhaitez.

```
SAUV "yourgame0.bin",b,2350 19119
EGAR      0,
DE
SAUV "yourgame1.bin",b,2500 17619
EGAR      0,
DE
SAUV "yourgame2.bin",b,2480 17819
EGAR      0,
DE
SAUV "yourgame3.bin",b,2400 18619
EGAR      0,
DE
```

La longitude est l'adresse finale moins l'adresse initiale. L'adresse finale, y compris la musique et les graphiques, la carte et la banque d'étoiles, est 42619, de sorte que la longueur peut être calculée en soustrayant l'adresse initiale de 42620. Par exemple, dans l'option d'assemblage 1, nous avons $42619 - 25000 = 17619$, juste la longueur indiquée dans cette commande.

Le deuxième fichier est votre ensemble de base
SAUVEGARDE "tujuego.bas"

Le troisième fichier est le chargeur ("**loader.bas**"), qui sera simplement :

```
10 MÉMOIRE 23499
15 LOAD "!paint.scr",&c000 : 'seulement si votre jeu a un écran de
chargement
20 LOAD " yourgame0.bin"
50 RUN " !yourgame.bas"
```

J'ai placé un chargement d'écran initial à l'adresse de la mémoire d'écran initiale (&C000). À la fin de ce manuel, vous trouverez l'une des nombreuses façons d'effectuer un chargement d'écran. Il est facultatif. Vous pouvez créer un jeu avec ou sans écran de chargement.

Cette méthode des trois fichiers est particulièrement utile si vous occupez la quasi-totalité de la mémoire disponible pour les graphiques. Sur les jeux à cassette, le chargement de 18 Ko peut prendre un certain temps et si vous n'utilisez pas les 8,5 Ko de graphiques, il peut être préférable de charger les différents binaires séparément pour gagner du temps. Par exemple, si vous n'utilisez que 2 Ko de graphiques, avec un seul binaire de longueur 18619, vous chargerez 8,5 Ko de graphiques, soit 6,5 Ko supplémentaires vides. Ce temps de chargement de la bande peut représenter près de

deux minutes supplémentaires. Sur disque (CPC 6128), cela n'a pas d'importance car le chargement ne prend pas de temps. Dans ce cas, il est préférable de créer un ou deux binaires qui ne stockent que la mémoire utilisée.

Pour créer ces trois fichiers, vous devez suivre les étapes suivantes :

ÉTAPE 1

Editer des graphiques avec SPEDIT et le résultat (SPEDIT l'envoie dans un fichier .txt) le copier dans
`images_mygame.asm`

ÉTAPE 2

Editer de la musique avec WYZtracker

Modifier music_mygame.asm pour inclure les musiques créées

Les morceaux seront assemblés l'un après l'autre, de sorte que chaque morceau commencera à une adresse mémoire différente en fonction de la taille du morceau.

ÉTAPE 3

Ré-assembler la bibliothèque 8BP, afin que la partie de la bibliothèque qui sélectionne les musiques (le player wyz) puisse savoir dans quelles adresses mémoire elles ont été assemblées (il y a d'autres dépendances, mais c'est l'une d'entre elles). Une fois réassemblé, vous devrez tout sauvegarder avec l'une de ces commandes en fonction de l'option d'assemblage que vous utilisez (le nom peut être ce que vous voulez, j'ai mis un numéro à la fin pour indiquer d'une certaine manière l'option d'assemblage) :

```
SAUV "yourgame0.bin",b,2350 19119
EGAR      0,
DE
SAUV "yourgame1.bin",b,2500 17619
EGAR      0,
DE
SAUV "yourgame2.bin",b,2480 17819
EGAR      0,
DE
SAUV "yourgame3.bin",b,2400 18619
EGAR      0,
DE
```

Il s'agit d'une version de la bibliothèque spécifique à votre jeu. Par exemple, la commande **|MUSIC,0,0,0,3,6** jouera la mélodie numéro 3 que vous avez composée vous-même. La mélodie numéro 3 peut être complètement différente dans un autre jeu.

ÉTAPE 4

Programmez votre jeu, qui doit d'abord exécuter l'appel pour installer les commandes RSX, c'est-à-dire **CALL &6b78**. Et quelque chose de très important : n'oubliez pas d'inclure la commande **MEMORY** au début, pour éviter que le BASIC en cours d'exécution ne stocke des variables au-dessus de l'adresse où 8BP commence.

MEMOIRE 23499 :rem utiliser cette MEMOIRE pour l'option 0
MEMOIRE 24999 :rem utiliser cette MEMOIRE pour l'option 1
MEMOIRE 24799 :rem utiliser cette MEMOIRE pour l'option 2
MEMOIRE 23999 :rem utiliser cette MEMOIRE pour l'option 3
MEMOIRE 23999 :rem utiliser cette MEMOIRE pour l'option 3

Votre jeu peut être programmé à l'aide de l'éditeur winape, qui est beaucoup plus polyvalent que l'éditeur AMSTRAD et peut être utilisé à la fois pour l'édition en assembleur (.asm) et en BASIC (.bas). L'éditeur winape est sensible aux mots-clés et change leur couleur automatiquement, ce qui facilite la programmation. Après avoir écrit un programme BASIC, vous devez le copier/coller dans la fenêtre CPC de winape. Pour accélérer le processus, vous pouvez activer l'option "High Speed" de winape pendant le collage, de sorte que le processus de collage sera immédiat.

ÉTAPE 5

Chargez tout avec un loader.bas , ce que vous devrez faire en BASIC.

ÉTAPE 6

Créer une cassette ou un disque avec votre jeu

5.3 Créer un disque ou une cassette avec votre jeu

5.3.1 Crédit d'un disque

Pour créer un nouveau disque à partir de winape, nous procéderons comme suit

Fichier->lecteur A-> nouveau disque vierge

Une fenêtre de gestion de fichiers s'ouvre alors, dans laquelle vous pouvez nommer le nouveau fichier .dsk.

Une fois créés, vous pouvez sauvegarder les fichiers avec la commande SAVE. Pour effacer un fichier, vous utilisez la commande "|ERA" (abréviation de ERASE), qui n'existe que sur le CPC 6128 dans le cadre du système d'exploitation "AMSDOS" (elle n'existe pas sur le CPC464, qui fonctionnait avec des cassettes).

|ERA, "game.*"

Et ils seront effacés

Pour charger le jeu, vous avez besoin d'un chargeur qui charge les fichiers nécessaires un par un. Quelque chose comme (la commande MEMORY dépend de l'option assembly) :

```
10 MEMORY 23499 : "La commande de mémoire dépend de l'option d'assemblage".
15 LOAD "!pant.scr",&c000 : 'seulement si votre jeu a un écran de
chargement
20 LOAD "yourgame0.bin"
50 RUN " !yourgame.bas"
```

Pour enregistrer chacun des fichiers, vous devez utiliser la commande SAVE avec les paramètres nécessaires, par exemple :

```
ENREGISTRER "LOADER.BAS"
SAVE "yourgame0.bin",b,23500,
19119 SAVE
"yourgame0.bin",b,23500, 19119
SAVE "yourgame.BAS"
```

Si vous voulez graver le .dsk sur une disquette 3,5" et le connecter à un lecteur de disquette externe de votre AMSTRAD CPC 6128, vous aurez besoin du programme CPCDiskXP, facile à utiliser. À partir d'un .dsk, vous pouvez graver une disquette 3,5" en double densité (n'oubliez pas de couvrir le trou de la disquette pour "tromper" le PC).

5.3.2 Réalisation d'une cassette avec Winape

La chose la plus importante lors de la création d'une bande est de sauvegarder les fichiers dans l'ordre dans lequel ils seront chargés par l'ordinateur. Une bande n'est pas comme un disque sur lequel vous pouvez charger n'importe quel fichier stocké, mais les fichiers sont l'un après l'autre, vous devez donc faire particulièrement attention à ce point.

Si votre chargeur de jeu est comme ceci :

10 MÉMOIRE 23499

15 LOAD "screen.scr",&c000 : rem si votre jeu a un écran de chargement

20 LOAD " !yourgame0.bin"

50 RUN " !yourgame.bas"

L'exclamation " !" est très importante pour que l'Amstrad ne reçoive pas le message "press play then any key" lors de l'exécution de chaque LOAD.

Vous devez d'abord enregistrer le chargeur (disons qu'il s'appelle "**loader.bas**"), puis le fichier "**tujuego.bin**" et enfin "**tujuego.BAS**".

Pour créer un fichier ".wav" ou à partir de winape

fichier->tape->presser l'enregistrement

Un menu de gestion des fichiers vous sera alors présenté afin que nous puissions décider du nom à donner au fichier ".wav".

Si vous êtes en mode CPC 6128, vous devez ensuite lancer le programme BASIC.

|TAPE

Et puis

VITESSE D'ECRITURE 1

Cette commande permet d'indiquer à l'AMSTRAD d'enregistrer à 2000 bauds. Cela réduira le temps de chargement. Si vous n'exécutez pas cette commande, l'enregistrement se fera à 1000 bauds, ce qui est plus sûr mais beaucoup plus lent.

ENREGISTRER "LOADER.BAS"

Vous obtiendrez un message vous demandant d'appuyer sur rec&play, puis sur "ENTER". Sauvegardez ensuite chaque fichier :

```
SAVE "yourgame0.bin",b,23500,  
19119 SAVE  
"yourgame0.bin",b,23500, 19119  
SAVE "yourgame.BAS"
```

Enfin, nous devons effectuer une dernière opération pour que winape ferme le fichier.

fichier->supprimer la bande

Après avoir effectué l'opération "remove tape", le fichier acquiert sa taille (si vous ne le faites pas, vous pouvez constater que sur le disque de votre PC, le fichier n'augmente pas et c'est parce qu'il n'a pas été déchargé sur le disque).

Pour charger le jeu, si vous êtes sur un CPC6128

|TAPE **RUN ""**

Pour réutiliser le disque

|DISC

Si vous souhaitez sauvegarder un écran de chargement, reportez-vous à l'annexe I sur l'organisation de la mémoire vidéo, où j'explique comment procéder.

5.3.3 Créer une cassette facilement avec CPCDiskXP, 2cdt et tape2wav

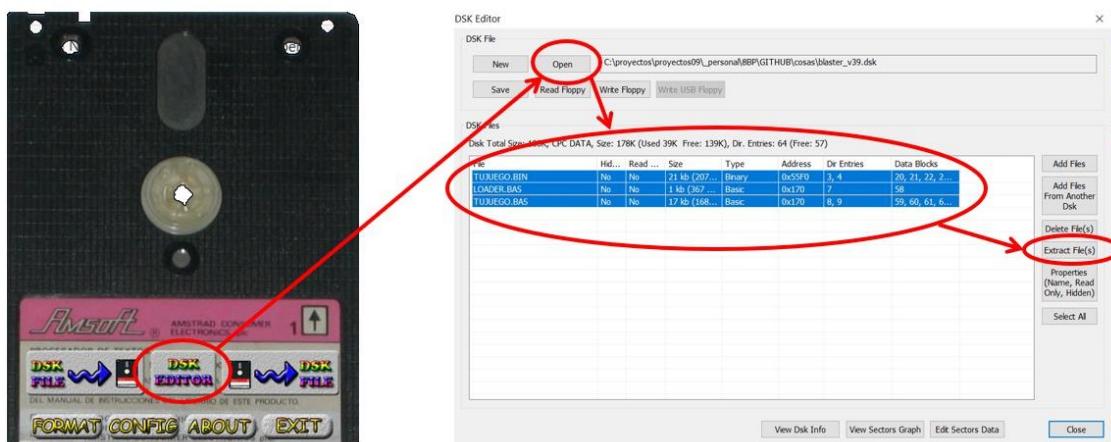
Winape est un outil formidable pour la programmation et l'émulation. Il a cependant une petite limitation : il ne permet pas de faire une bande au format cdt, mais seulement au format wav.

Il existe une méthode très rapide et fiable qui vous permettra de créer des fichiers .cdt et wav pour le

vous avez besoin des outils CPCDiskXP, 2cdt et tape2wav.

Partons du principe que vous avez créé un fichier .dsk avec vos fichiers. Dans ce cas, vous devez suivre les étapes suivantes :

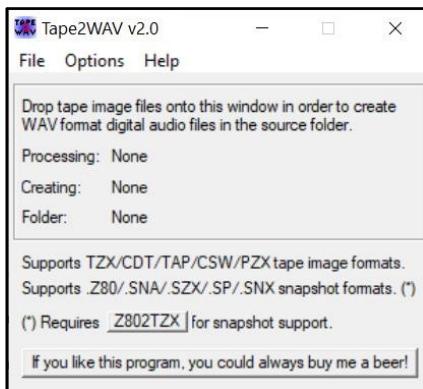
- 1) Tout d'abord, l'outil CPCDiskXP vous permet d'ouvrir le fichier .dsk et d'extraire les fichiers nécessaires à votre jeu (le "loader.bas", le "tujuego.bin" et le "tujuego.bas"). Pour ce faire, il vous suffit de cliquer sur "édition de disque", puis sur "ouvrir", d'ouvrir votre .dsk, de sélectionner les fichiers et enfin de cliquer sur "extraire les fichiers". Une fois que vous avez fait cela, vous aurez les fichiers dans le système de fichiers de Windows.



- 2) Ensuite, vous utilisez l'outil 2cdt pour graver les fichiers, un par un, dans un fichier .cdt. Les commandes seraient les suivantes :

```
2cdt.exe -n -s 1 -r "LOADER.bas" "loader.bas" tucinta.cdt
2cdt.exe -b 2000 -r "tujuego0.bin" "tujuego0.bin" tucinta.cdt
2cdt.exe -b 2000 -r "tujuego.bas" "tujuego.bas" tucinta.cdt
```

- 3) Vous avez maintenant créé le fichier tucinta.cdt. Si vous voulez aussi avoir un .wav pour pouvoir le charger sur un vrai CPC 464, vous pouvez utiliser l'outil tape2wav. Il suffit de le lancer et de faire glisser le fichier .cdt avec la souris dans l'outil. Le tape2wav créera immédiatement un fichier "tucinta.wav" dans le répertoire où se trouvait "tucinta.cdt".



5.3.4 Dépannage de LOAD et MEMORY

Avant de charger un fichier BASIC, l'Amstrad s'assure qu'il aura de l'espace disponible pour l'exécuter. Le programme BASIC peut n'utiliser que 1KB de variables, mais l'Amstrad ne le sait pas, il est donc plus conservateur et demande que vous ayez 5KB de mémoire vide en plus. Ces 5 Ko peuvent sembler excessifs, mais l'Amstrad ne sait pas a priori combien de variables vous allez déclarer dans votre programme et il préfère avoir beaucoup d'espace libre pour stocker les variables plutôt que de manquer d'espace et de voir le programme échouer.

Cela signifie que si vous avez précédemment chargé un ou plusieurs fichiers binaires (avec des graphiques, la bibliothèque 8BP ou autre) et qu'il vous reste 20 Ko de libre, vous ne pourrez pas charger un jeu BASIC de 20 Ko, mais tout au plus un jeu de 15 Ko. Mais ne vous inquiétez pas, il y a plusieurs façons de résoudre ce problème. Je vais vous expliquer la plus simple

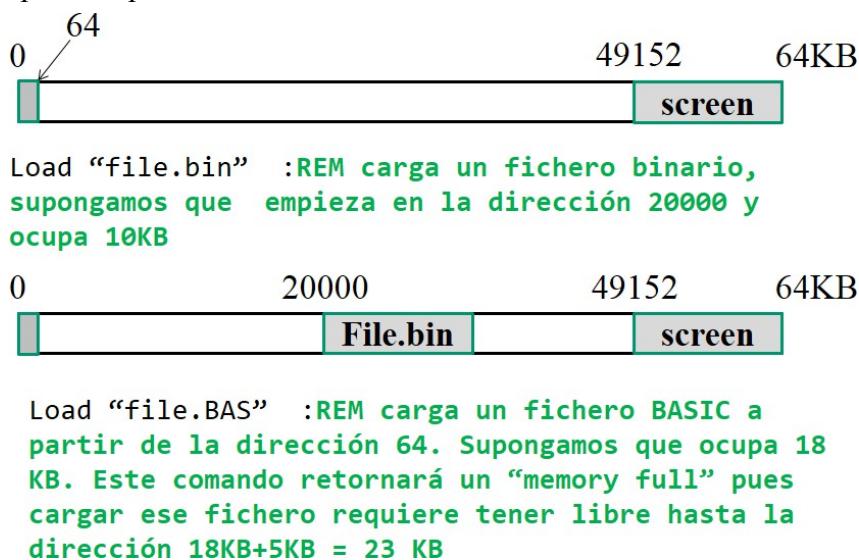


Fig. 13 Le problème LOAD

La solution consiste à créer un fichier "loader.bas" qui modifie la MEMOIRE. Supposons que vous ayez des données binaires à partir de l'adresse 20 000 et que votre programme prenne 18 Ko, ce qui laisse moins de 5 Ko d'espace. Tout ce que vous avez à faire, c'est.. :

```

10 MÉMOIRE 19999
20 LOAD "game.bin" : rem charge les données à partir de 20000
30 CLEAR : MEMORY 23000 : rem pour vous donner plus de RAM libre. 40
RUN "game.bas" : rem la première ligne de game.bas doit être en
mémoire 19999

```

Cette méthode est très simple et fiable à 100 % car, bien que vous laissiez des parties des données binaires non protégées pendant le chargement de BASIC, la première chose que vous faites en BASIC est d'exécuter la MEMOIRE, de sorte qu'elle est à nouveau protégée avant que vous n'ayez créé la moindre variable.

Un autre problème typique lié à l'erreur MEMOIRE PLEINE se produit lorsque nous chargeons un programme et qu'au milieu de son exécution nous l'arrêtions (en appuyant deux fois sur ESC). Il est possible que nous obtenions **MEMOIRE PLEINE** lorsque nous essayons de faire des choses comme accéder au disque avec une commande CAT.

```
Break in 420
Ready
CAT
Memory full
Ready
```

Ceci est dû au fait que notre programme BASIC peut consommer beaucoup de mémoire RAM de variables. Le fait de l'avoir arrêté ne signifie pas que les variables ont disparu du programme, en fait, elles existent toujours et vous pouvez même les imprimer pour voir leur valeur. Ce que nous allons faire dans ce cas, c'est simplement exécuter la commande **CLEAR**, qui libère la mémoire de ces variables et ensuite notre commande CAT (ou celle que nous voulons).

```
Break in 420
Ready
CAT
Memory full
Ready
CLEAR
Ready
CAT

Drive A: user 0
LOADER :BAK 1K SP .BAS 18K
LOADER :BAS 1K SP :BIN 19K
SP :BAK 18K SP :SCR 17K

104K free
Ready
```

Si vous avez créé un programme BASIC si volumineux que vous n'avez pas 5 Ko de libre entre le programme BASIC et le binaire assemblé, vous obtiendrez logiquement l'erreur **MEMOIRE PLEINE** lorsque vous voudrez sauvegarder votre jeu sur le disque.

Si vous essayez de sauvegarder le binaire, vous obtiendrez une erreur **MEMOIRE PLEINE**, mais c'est très facile à résoudre. Il suffit de ne pas charger le listing BASIC dans votre émulateur, et de ne pas exécuter de commandes MEMORY. Lorsque vous assemblez avec winape le fichier "make_all.asm", vous aurez tous les graphiques, la musique et la bibliothèque 8BP assemblés dans la mémoire de l'Amstrad. Lancez alors la commande SAVE et cela fonctionnera. La commande pour sauvegarder le tout dans un seul binaire dépend de l'option d'assemblage que vous avez mise dans le fichier **make_all_mygame.asm**, et sera l'une de celles-ci :

```
SAUV "yourgame0.bin",b,2350 19119
EGAR      0,
DE
SAUV "yourgame1.bin",b,2500 17619
EGAR      0,
DE
SAUV "yourgame2.bin",b,2480 17819
EGAR      0,
DE
SAUV "yourgame3.bin",b,2400 18619
EGAR      0,
```

Toutefois, si vous avez stocké des éléments binaires (graphiques supplémentaires,

cartes, etc.) sous l'adresse initiale de 8BP, vous devrez en tenir compte. Par exemple, si votre jeu commence à utiliser la mémoire à l'adresse 20000, la commande sera la suivante (notez que j'ai augmenté la longueur de la commande pour qu'elle s'étende de 20000 à 42619)

```
SAVE "yourgame.bin",b,20000, 22619
```

Vous pouvez maintenant copier et coller votre programme BASIC dans l'émulateur. Une fois copié, n'exécutez aucune commande **MEMORY** et sauvegardez votre fichier .BAS sur le disque.

Comme vous n'avez pas encore exécuté de commande **MEMORY** à ce stade, l'Amstrad "pense" que vous avez plus de 5KB de libre au-dessus de votre programme BASIC et ne nous donnera pas le message **MEMORY FULL**. Une fois que vous avez exécuté votre commande **MEMORY** (par exemple, si vos données binaires commencent à 20000, ce sera une **MEMORY 19999** au lieu de 23999), l'Amstrad vérifiera si vous avez 5KB de libre entre votre programme BASIC et l'adresse **MEMORY** et s'il n'y en a pas assez, il vous donnera une erreur lors de l'exécution de la commande **SAVE**, même si le jeu fonctionne. Si pendant l'exécution votre jeu essaie de consommer plus d'espace qu'il n'en a de libre à l'adresse **MEMOIRE**, il s'arrêtera et donnera une erreur **MEMOIRE PLEINE**.

6 Bibliothèque, musique et assemblage graphique

Ce chapitre détaille un peu plus ce qui se passe lorsque vous exécutez le fichier "make_all" et vous permettra de mieux comprendre l'ensemble du processus, **bien que si vous avez envie de commencer à apprendre à programmer avec 8BP, vous pouvez le sauter et revenir ici plus tard**, lorsque vous voudrez mieux comprendre où placer les graphiques et la musique et comment assembler la bibliothèque avec eux.

En effet, par exemple, le lecteur de musique est intégré dans la bibliothèque et doit savoir où commence chaque chanson (adresse mémoire). Il est donc nécessaire de rassembler et de sauvegarder la version de la bibliothèque spécifique à votre jeu, ainsi que le fichier graphique assemblé et le fichier musical assemblé.

Comme je l'ai expliqué dans la section "étapes", il s'agira d'une version de la bibliothèque spécifique à votre jeu. Par exemple, la commande **|MUSIC,0,0,3,6** jouera la mélodie numéro 3 que vous avez composée vous-même. La mélodie numéro 3 peut être complètement différente dans un autre jeu. Il en va de même pour les données du fichier d'instrument. Il existe certaines dépendances entre le code du lecteur de musique et les adresses où les données de l'instrument et les mélodies sont assemblées.

C'est très simple, mais vous devez comprendre la structure de la bibliothèque pour le faire, c'est-à-dire la structure des fichiers .asm que vous devez manipuler et leurs dépendances.

Le diagramme suivant présente tous les fichiers .asm d'un jeu utilisant 8BP ainsi que les dépendances entre eux. Dans la figure, **en gris, se trouvent les fichiers que vous devez éditer**, tels que :

- le fichier de chansons et d'instruments, que vous générez avec le WYZtracker
- le fichier **make_musica** dans lequel vous indiquez quels fichiers ".mus" doivent être assemblés
- le fichier image que vous créez avec SPEDIT
- la table des sprites dans laquelle vous attribuez des images aux sprites (bien que cela ne soit pas strictement nécessaire puisque vous disposez de la commande **|SETUPSP**).
- la table des séquences, où vous définissez les images qui composent une séquence,
- la carte du monde : vous pouvez définir jusqu'à 64 éléments qui composent le monde.
- le fichier des chemins : où vous définissez les chemins des sprites que vous souhaitez utiliser.
- le fichier alphabet.asm : si vous souhaitez créer un alphabet différent de l'alphabet standard 8BP

Vous pouvez assembler le tout en ouvrant le fichier "**make_all.asm**" et en appuyant sur "assembler" dans le menu Winape. Vous pouvez ensuite utiliser la commande SAVE pour sauvegarder les images, la musique et la bibliothèque 8BP dans différents fichiers binaires, ou dans un seul, comme nous l'avons vu.

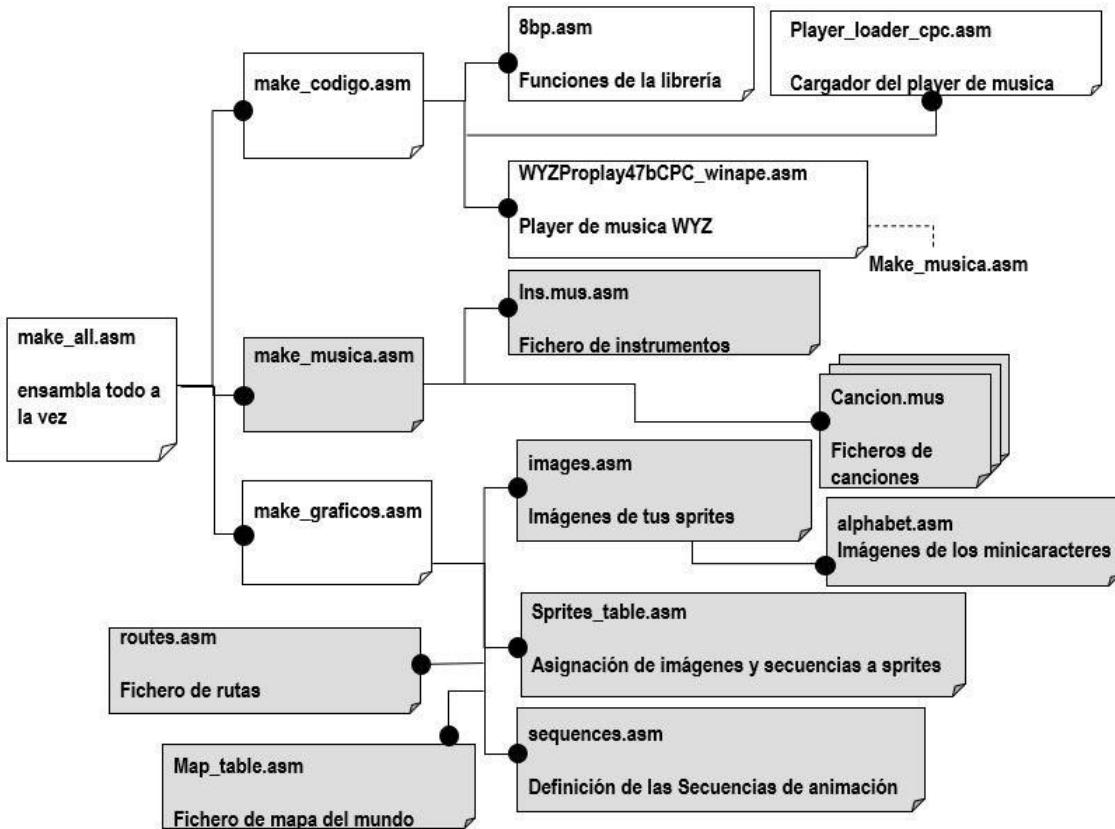


Fig. 14 Fiches d'assemblage

Si vous ne modifiez que les graphiques, vous pouvez les assembler séparément en sélectionnant le fichier "make_graphics.asm" et en appuyant sur assemble.

Si vous changez la musique, vous devez réassembler le code de la bibliothèque car il existe une dépendance entre le code et les chansons, car le code doit savoir où commence chaque chanson. Donc si vous changez ou ajoutez des chansons, vous devez assembler avec make_all.asm . J'ai reflété cette dépendance par une ligne pointillée entre le lecteur et le fichier make_musica.asm.

Il se peut que vous ayez besoin d'assembler quelque chose d'autre, comme une carte de piste de course qui utilise vos images. Dans ce cas, ajoutez-le au fichier "Make_graphics.asm" pour qu'il soit assemblé après images.asm. L'ordre d'assemblage est important. Vous devez d'abord assembler les images, leur associer des étiquettes, puis vous pouvez assembler les cartes ou les pistes qui utilisent ces étiquettes.

6.1 Make_all.asm

C'est le fichier qui permet de tout assembler. En interne, il invoque trois fichiers qui assemblent le code de la bibliothèque et du lecteur de musique, les chansons et les graphiques.

Makefile pour les jeux vidéo utilisant la puissance 8bit
Si vous en modifiez une partie, il vous suffit de
; assembler les marques correspondantes
par exemple, vous pouvez assembler le make_graphics si vous changez de dessin
DEPUIS LA VERSION 42, IL EXISTE DES "OPTIONS" D'ASSEMBLAGE.

```

; OPTION_ASSEMBLAGE = 0 --> toutes les commandes disponibles.

; OPTION_ASSEMBLAGE = 1 --> pour les jeux de labyrinthe. MEMOIRE 25000
;                               disponible les commandes |LAYOUT, |COLAY
;
; OPTION_ASSEMBLAGE =      --> pour les jeux à défilement, MEMOIRE
;                               24800
;                               disponible les commandes |MAP2SP, |UMA
;
; OPTION_ASSEMBLAGE =      --> pour les jeux en pseudo 3D, MEMOIRE 24000
;                               commande disponible |3D
;
; OPTION_ASSEMBLAGE =      --> utilisation future

let ASSEMBLING_OPTION = 0
;-----CODIGO-----
;comprend la bibliothèque 8bp et le lecteur de
musiqueWYZ lire "make_codigo_mygame.asm".

;-----MUSICA-----
lire "make_musica_mygame.asm";
comprend les chansons.

; ----- GRAPHIQUES -----
Cette partie comprend des images et des séquences d'animation.
; et la table des sprites initialisée avec ces images et séquences lues
dans "make_graficos_mygame.asm".

```

Chacun de ces trois fichiers est chargé d'assembler différentes choses et, par exemple, le fichier graphique invoque d'autres fichiers tels que le fichier d'images, le fichier de séquences, le fichier d'itinéraires et la carte du monde.

Utilisez toujours l'option d'assemblage qui vous donne le plus de mémoire libre. Si votre jeu est un jeu de labyrinthe, utilisez l'option 1, et si c'est un jeu de défilement, utilisez l'option 2. Si c'est un jeu pseudo3D, utilisez l'option 3. En général, je ne vous recommande pas d'utiliser l'option 0 car c'est celle qui vous donne le moins de mémoire libre et vous pourrez probablement utiliser l'une des autres options.

6.2 Structure du fichier image

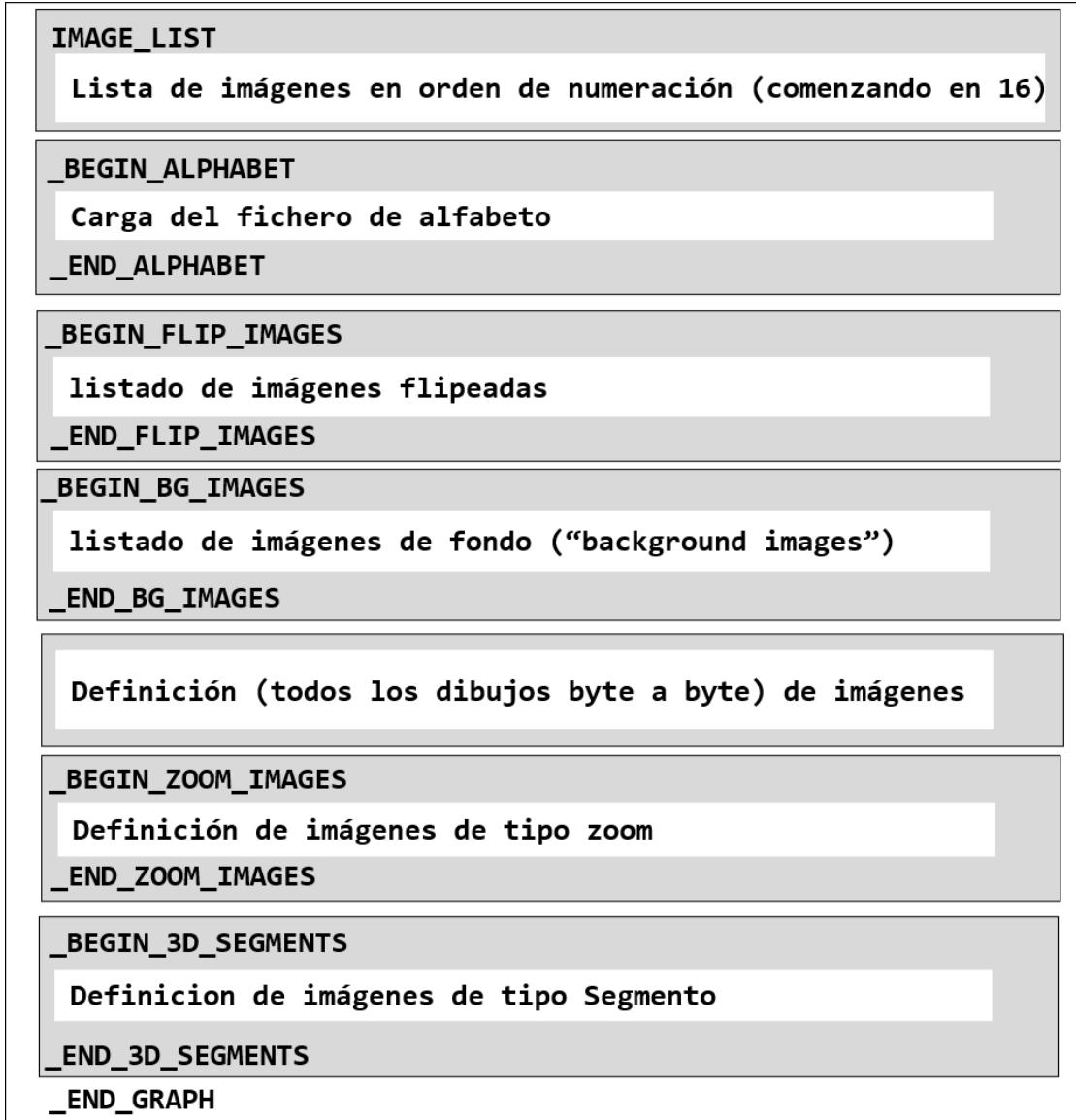


Fig. 15 Structure du fichier image

6.3 Structure du fichier de la séquence d'animation

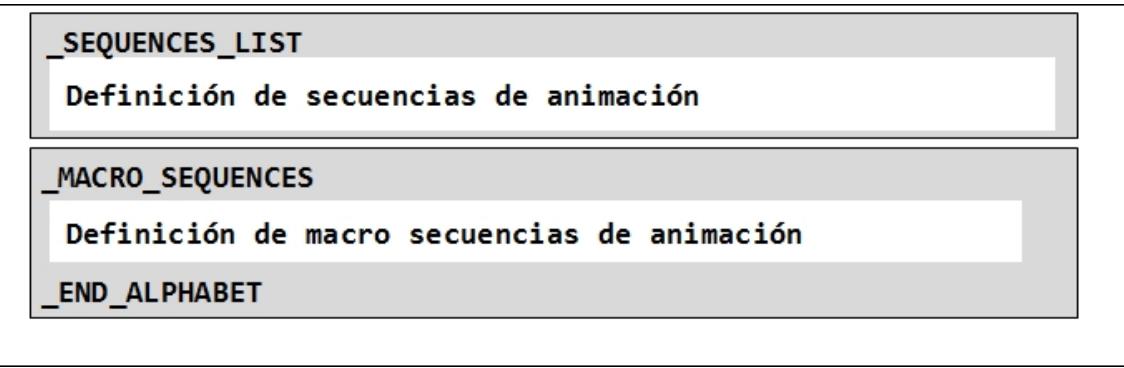


Fig. 16 Structure du fichier de la séquence d'animation

6.4 Structure du fichier de routage

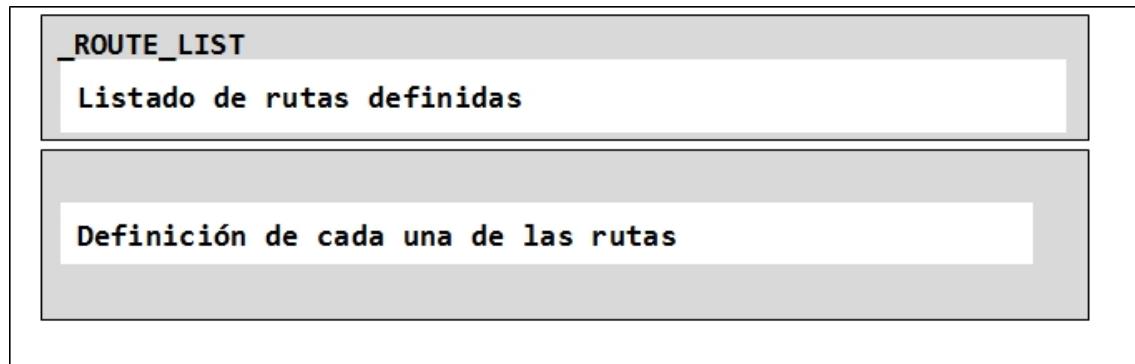


Fig. 17 Structure du fichier de routage

6.5 Structure du fichier de la carte du monde

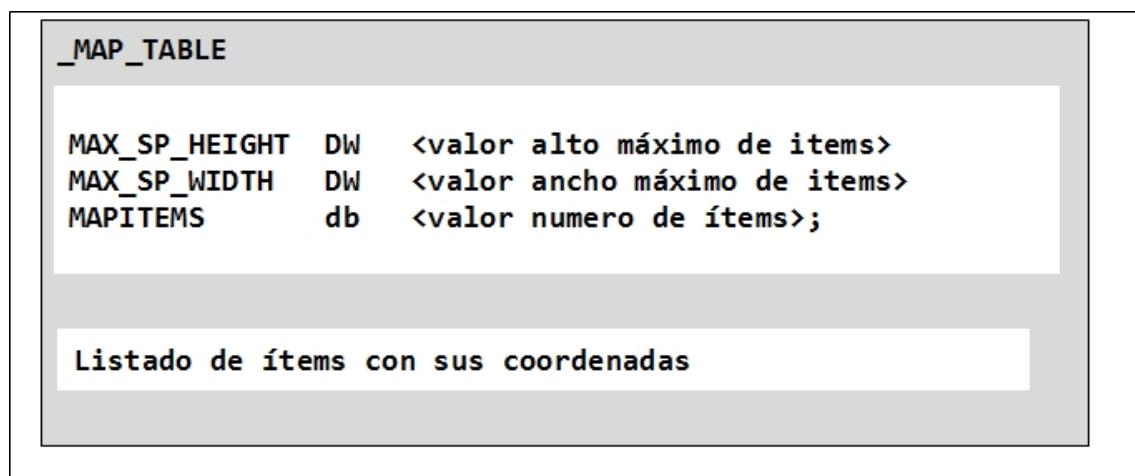


Fig. 18 Structure du fichier de la carte du monde

7 Cycle de jeu

Un jeu vidéo d'arcade, de plateforme ou d'aventure a généralement une structure similaire, dans laquelle certaines opérations sont répétées de manière cyclique dans ce que nous appellerons un "cycle de jeu".

À chaque cycle de jeu, nous mettons à jour les positions des sprites et imprimons les sprites à l'écran, de sorte que le nombre de cycles de jeu exécutés par seconde est égal au nombre d'images par seconde (FPS) du jeu. Le pseudo-code suivant décrit la structure de base d'un jeu

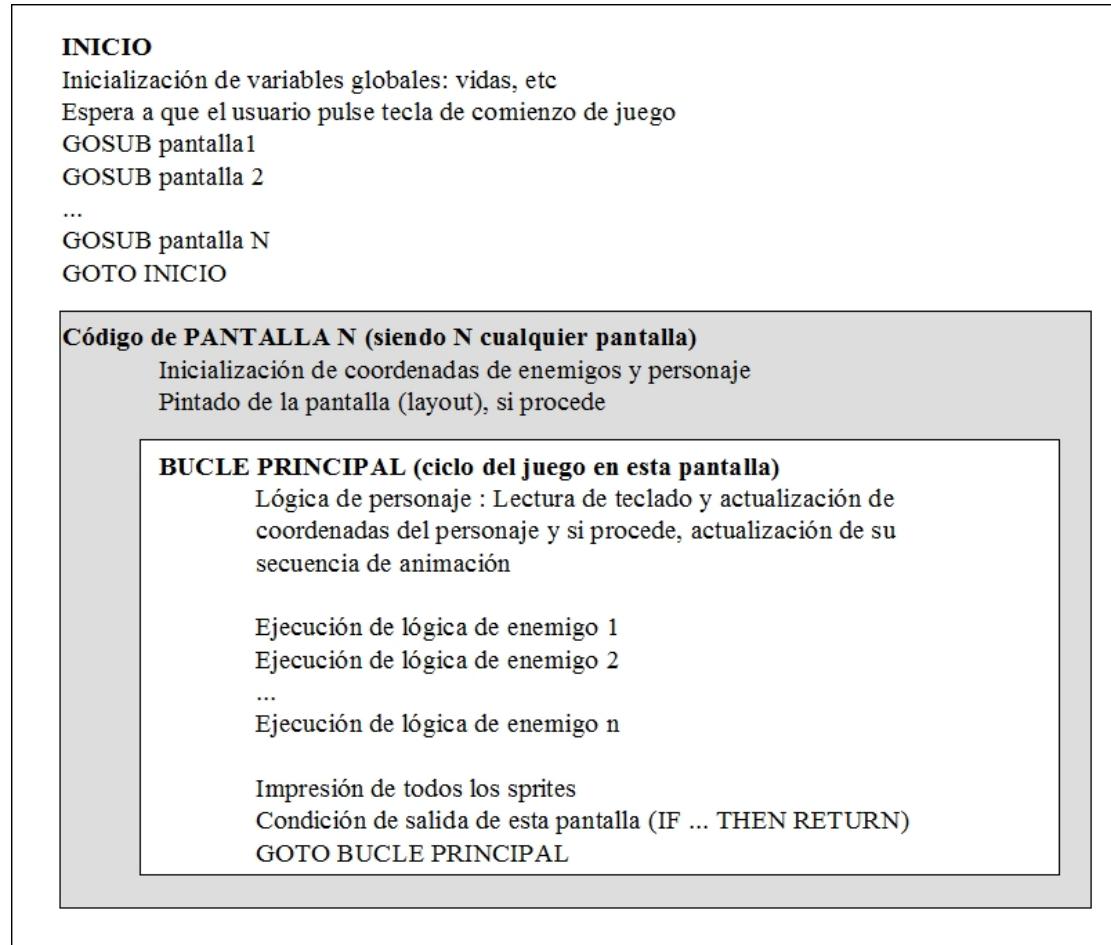


Fig. 19 Structure de base d'un jeu

Si la logique des ennemis est trop lourde en raison d'un trop grand nombre d'ennemis ou trop complexe, cela consommera plus de temps à chaque cycle de jeu et le nombre de cycles par seconde sera donc réduit. Essayez de ne pas descendre en dessous de 10fps pour que le jeu conserve un niveau d'action acceptable.

7.1 Comment mesurer le FPS de votre cycle de jeu

Pour savoir si votre jeu a un niveau d'action acceptable, il n'y a rien de mieux que d'y jouer et si vous l'aimez, il sera parfait. Cependant, vous voudrez peut-être mesurer exactement le nombre d'images par seconde que votre jeu est capable de générer, car vous pourrez alors prendre des décisions dans la logique de votre programme et mesurer à quel point ces décisions de programmation lui nuisent ou lui profitent.

Ce que nous allons mesurer, c'est simplement prendre note de l'instant qui précède le début du premier cycle de jeu, au début du "code N-écran". Ensuite, nous prendrons note du temps après quelques cycles de jeu et nous ferons une simple division. Voyons cela étape par étape :

A=TIME : rem cette ligne stocke dans la variable A le temps en fractions de 1/300 de seconde.

Le nombre à stocker dans A peut être très grand, en effet, il peut être plus grand que ce qu'une variable entière comme "A" est capable de stocker. Pour que l'affectation ne produise pas d'erreur, il est pratique de remettre à zéro la minuterie de l'AMSTRAD, qui est lancée à chaque fois que la machine est mise en marche. Pour le remettre à zéro, avant d'assigner la variable "A", il suffit d'exécuter la commande suivante :

Sur un CPC 6128

POKE &b8b4,0 : POKE &b8b5,0 : POKE &b8b6,0 : POKE &b8b7,0

Sur un CPC 464

POKE &b187,0 : POKE &b188,0 : POKE &b189,0 : POKE &b18a,0

Pour différencier la machine sur laquelle se trouve votre programme, vous devez désactiver la musique et consulter une adresse avec PEEK.

| MUSIC : Si peek(&39)=57 alors Modèle=464 sinon modèle=6128 Si modèle=464 alors ...

Vous aurez ainsi mis à zéro les adresses mémoire où l'AMSTRAD stocke le timer. Ensuite, nous exécutons autant de cycles de jeu que nous le souhaitons, et nous contrôlons le cycle dans lequel nous nous trouvons à l'aide de la variable "cycle", que nous augmentons d'une unité à chaque cycle. Après avoir quitté cette phase ou cet écran, nous exécutons :

FPS= cycle * 300/ (TIME - A)

Et maintenant, nous avons le FPS de notre jeu. Je vais vous présenter tout cela dans l'ordre ci-dessous :

Rem suppose que nous sommes dans un 6128

POKE &b8b4,0 : POKE &b8b5,0 : POKE &b8b6,0 : POKE &b8b7,0

A=TIME

**<Ici aller à le programme qui gère votre cycle
de cycle, y compris cycle=cycle+1 >**

Nous obtiendrons ensuite la condition de sortie de l'affichage FPS= cycle * 300/ (TIME - A)

PRINT "FPS =";FPS

En clair : le temps écoulé entre le début et la fin du programme est le TEMPS-A exprimé en 1/300 de seconde. Pour le convertir en secondes, il faut le diviser par 300.

Secondes= (TIME -A) /300

Si n cycles ont été exécutés pendant ces secondes (par exemple), un cycle a duré : **Tc= 300*(TIME-A)/n**

Et le nombre de cycles pouvant être exécutés en une seconde (le FPS) est l'inverse, c'est-à-dire **FPS = 1/Tc = n*300/(TIME-A)**.

8 Sprites

8.1 Editer des sprites avec SPEDIT et les assembler

Spedit (Simple Sprite Editor) est un outil qui vous permet de créer vos propres images de personnages et d'ennemis et de les utiliser dans vos programmes BASIC.

Spedit est conçu en BASIC, et il est très simple, de sorte que vous pouvez le modifier pour faire des choses qui ne sont pas envisagées et qui vous intéressent. Il fonctionne sur l'Amstrad CPC, bien qu'il soit conçu pour être utilisé à partir de l'émulateur winape.

La première chose à faire est de configurer winape pour que la sortie de l'imprimante se fasse dans un fichier. Dans cet exemple, j'ai placé la sortie de l'imprimante dans le fichier printer5.txt.

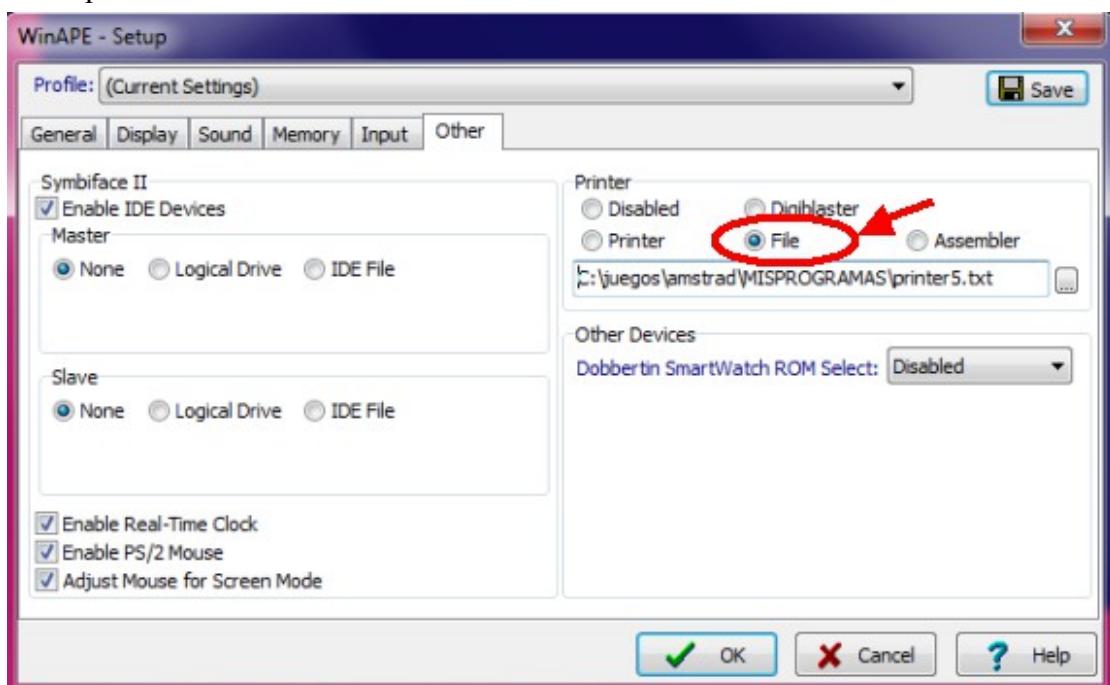


Fig. 20 Redirection de l'imprimante CPC vers un fichier avec Winape

Lorsque vous lancez SPEDIT, vous obtenez le menu suivant, dans lequel vous pouvez choisir entre l'édition d'un Sprite ou la capture d'un Sprite à partir d'un fichier image (.scr).

La capture de sprites est disponible à partir de SPEDIT V14. Si vous souhaitez capturer un sprite, vous devez disposer d'un fichier image (.scr) sur le disque, qui n'est qu'un fichier binaire de 16384 octets. Il peut s'agir d'une image provenant d'un jeu que vous avez capturé et qui contient des images de personnages que vous aimez et que vous ne voulez pas perdre de temps à éditer.

Si vous choisissez d'éditer un Sprite, le programme vous demande la palette à utiliser. Vous pouvez choisir une palette par défaut ou une palette personnelle que vous souhaitez définir. Vous pouvez également utiliser une palette que vous avez

précédemment sauvegardée (elle est toujours sauvegardée dans pal.dat, il suffit d'appuyer sur "i" lors de l'édition d'un sprite). Si vous décidez de définir votre propre palette, vous devrez reprogrammer les lignes BASIC où la palette alternative (ou "personnalisée") est définie.

est une sous-routine invoquée par GOSUB lorsque vous appuyez sur "2" dans la réponse à la question concernant la palette que vous souhaitez utiliser.

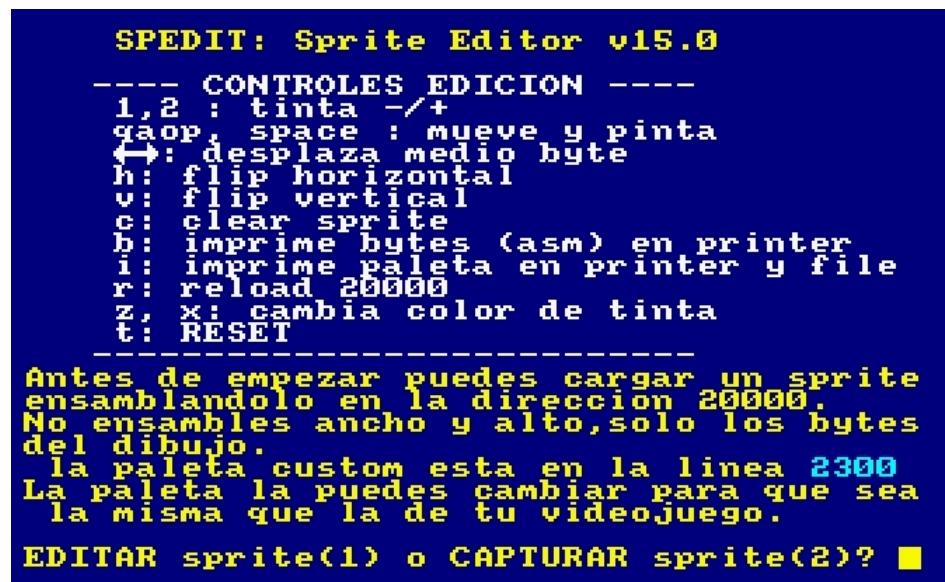


Fig. 21 Écran initial de SPEDIT

L'outil vous permet de choisir le mode 1 ou le mode 0. Une fois en mode édition, il vous permet d'éditer des dessins, avec une aide à l'écran. Vous manipulez un pixel clignotant et les coordonnées de l'endroit où vous vous trouvez ainsi que la valeur de l'octet dans lequel vous êtes sont affichées en bas.

Lorsqu'il demande la largeur et la hauteur du sprite, rappelez-vous que **la hauteur maximale d'un sprite en 8BP est de 127 lignes**, ainsi que sa largeur maximale en octets. Notez également que la largeur est demandée en pixels, mais vous devez savoir que l'éditeur travaille en interne en octets, donc, si vous allez faire une image en mode 0, la largeur doit être un nombre pair (un octet = 2 pixels) et si vous allez faire une image en mode 1, la largeur doit être un multiple de 4 (un octet = 4 pixels).



Fig. 22 Écran d'édition SPEDIT

SPEDIT vous permet de "refléter" votre image pour faire marcher la même figure vers la gauche sans effort, en appuyant simplement sur H (retournement horizontal) et la même chose peut être faite verticalement. Il vous permet également de "refléter l'image" par rapport à un axe imaginaire situé au centre du personnage, à la fois verticalement et horizontalement. Ceci est très utile pour les caractères symétriques ou presque symétriques, pour lesquels une aide au dessin est toujours utile.

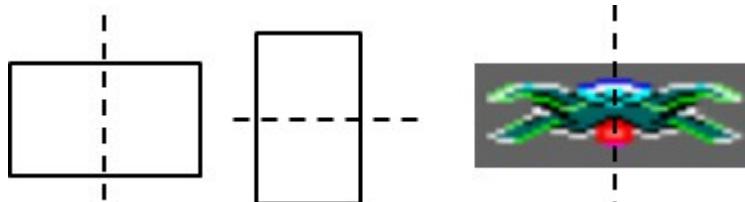


Fig. 23 sprites symétriques avec SPEDIT

Depuis la version 11 de SPEDIT, le mode 1 d'AMSTRAD est pris en charge. Vous pouvez donc éditer des sprites en mode 1 sans problème et utiliser le mécanisme de mise en miroir.



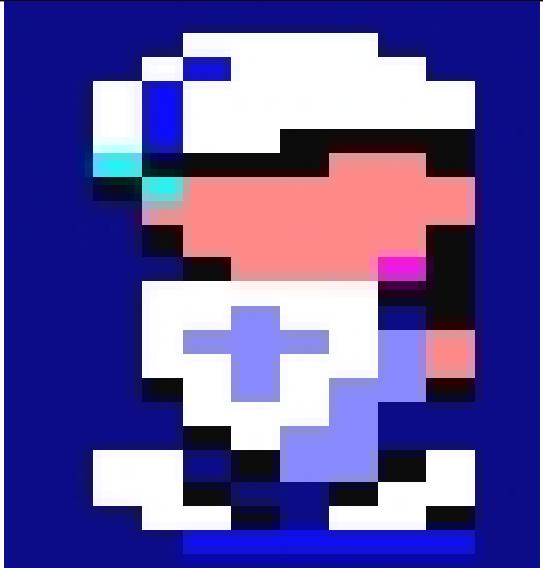
Fig. 24 édition des sprites dans le MODE 1 avec SPEDIT

Une fois que vous avez défini votre mannequin, pour extraire le code d'assemblage, vous devez appuyer sur la touche "b". Cela enverra à l'imprimante (dans le fichier que nous avons défini comme sortie) un texte comme le suivant, auquel vous pouvez ajouter un nom, je l'ai appelé "SOLDADO_R1".

```

;----- BEGIN IMAGE -----
SOLDADO_R1
db 6 ; ancho
db 24 ; alto
db 0 , 0 , 0 , 0 , 0 , 0
db 0 , 0 , 0 , 0 , 0 , 0
db 0 , 0 , 48 , 48 , 0 , 0
db 0 , 16 , 56 , 48 , 32 , 0
db 0 , 52 , 48 , 48 , 48 , 0
db 0 , 52 , 48 , 48 , 48 , 0
db 0 , 52 , 48 , 240 , 240 , 0
db 0 , 88 , 240 , 229 , 218 , 0
db 0 , 164 , 207 , 207 , 207 , 0
db 0 , 69 , 207 , 207 , 207 , 0
db 0 , 80 , 207 , 207 , 218 , 0
db 0 , 0 , 229 , 207 , 248 , 0
db 0 , 16 , 48 , 48 , 240 , 0
db 0 , 16 , 37 , 48 , 80 , 0
db 0 , 16 , 15 , 26 , 79 , 0
db 0 , 16 , 37 , 48 , 79 , 0
db 0 , 80 , 37 , 37 , 90 , 0
db 0 , 0 , 48 , 37 , 0 , 0
db 0 , 0 , 176 , 15 , 0 , 0
db 0 , 48 , 80 , 15 , 176 , 0
db 0 , 48 , 160 , 80 , 48 , 0
db 0 , 16 , 112 , 16 , 112 , 0
db 0 , 0 , 60 , 60 , 60 , 0
db 0 , 0 , 0 , 0 , 0 , 0
;----- END IMAGE -----

```



Remarquez que j'ai toujours laissé un octet à gauche, à zéro. J'ai fait cela pour que, lorsque le soldat se déplace vers la droite, il "s'efface", sinon il laisserait une trace, "tachant" l'écran au fur et à mesure qu'il avance.

Fig. 25 Soldat au format .asm

Une fois que vous avez réalisé le premier cadre de votre soldat, vous pouvez abandonner le travail et continuer un autre jour. Pour partir du soldat que vous avez dessiné et continuer à le retoucher ou à le modifier pour construire une autre image, vous pouvez assembler le soldat à l'adresse **20000**, en supprimant la largeur et la hauteur. Une fois assemblé à partir de winape, vous dites à SPEDIT que vous allez éditer un sprite de la même taille et, une fois dans l'écran d'édition, vous appuyez sur "r" (reload). Le sprite sera chargé à partir de l'adresse **20000**, qui est celle où vous l'avez "assemblé".

Une grande partie de l'attrait d'un jeu réside dans ses sprites. Ne lésinez pas sur ce point, faites-le lentement et avec goûts et votre jeu sera bien plus beau.

```

org 20000
----- BEGIN IMAGE -----
SOLDADO_R1
;db 6 ; ancho ojo! comentamos esta linea
;db 24 ; alto ojo! comentamos esta linea
db 0 , 0 , 0 , 0 , 0 , 0
db 0 , 0 , 0 , 0 , 0 , 0
db 0 , 0 , 48 , 48 , 0 , 0
db 0 , 16 , 56 , 48 , 32 , 0
db 0 , 52 , 48 , 48 , 48 , 0
db 0 , 52 , 48 , 48 , 48 , 0
db 0 , 52 , 48 , 240 , 240 , 0
db 0 , 88 , 240 , 229 , 218 , 0
db 0 , 164 , 207 , 207 , 207 , 0
db 0 , 69 , 207 , 207 , 207 , 0
db 0 , 80 , 207 , 207 , 218 , 0
db 0 , 0 , 229 , 207 , 248 , 0
db 0 , 16 , 48 , 48 , 240 , 0
db 0 , 16 , 37 , 48 , 80 , 0
db 0 , 16 , 15 , 26 , 79 , 0
db 0 , 16 , 37 , 48 , 79 , 0
db 0 , 80 , 37 , 37 , 90 , 0
db 0 , 0 , 48 , 37 , 0 , 0
db 0 , 0 , 176 , 15 , 0 , 0
db 0 , 48 , 80 , 15 , 176 , 0
db 0 , 48 , 160 , 80 , 48 , 0
db 0 , 16 , 112 , 16 , 112 , 0
db 0 , 0 , 60 , 60 , 60 , 0
db 0 , 0 , 0 , 0 , 0 , 0
----- END IMAGE -----

```

Vous savez ainsi ce que signifie "assembler" un sprite. Il s'agit simplement de placer les octets de données qui le constituent à des adresses mémoire consécutives, dans ce cas en commençant par **20000**.

SPEDIT occupe très peu de mémoire et cette adresse est très éloignée du programme, de sorte qu'il n'y a aucun problème à "endommager" le programme SPEDIT en l'assemblant.

Fig. 26 Assemblage graphique

Pour savoir à quelle adresse mémoire chaque image a été assemblée, utilisez le menu winape : Assemble->symbols

Vous obtiendrez ainsi une relation entre les balises que vous avez définies, comme "SOLDADO_R1", et l'adresse mémoire (en hexadécimal) à partir de laquelle elles ont été assemblées. Pour transformer les adresses hexadécimales en adresses décimales, vous pouvez utiliser la calculatrice Windows en mode "programmateur".

Assembler Symbols		
Filter:		
soldado_12	8678	✓
soldado_r0	839E	✓
soldado_r1	8430	✓
soldado_r2	84C2	✓
song	7CF8	✓
song_0	7F3C	✓
song_0_end	802C	✗

Fig. 27 détail de ce que les symboles affichent dans Winape

Une fois que vous avez réalisé les différentes phases d'animation de votre soldat, vous pouvez les regrouper dans une "séquence" d'animation. Les séquences d'animation sont des listes d'images et ne sont pas définies avec SPEDIT. Avec SPEDIT, il vous suffit d'éditer le fichier

"frames". Dans une section ultérieure, j'expliquerai comment indiquer à la bibliothèque 8BP quel ensemble d'images constitue une séquence d'animation.

Les images que vous créez pour votre jeu sont sauvegardées dans un seul fichier, appelé par exemple "images_mygame.asm". Ce fichier commence par une liste d'images que vous pouvez référencer dans les commandes 8BP en BASIC avec un index, quelle que soit l'adresse où elles sont assemblées, par exemple :

LISTE D'IMAGES

La première image sera toujours attribuée à l'index 16, l'image suivante à l'index 16 et l'image suivante à l'index 16.

17 et ainsi de suite

;

**dw SOLDIER_R0 ; 16
dw SOLDIER_R1 ; 17
dw SOLDIER_R2 ; 18**

Une fois que toutes les images sont terminées, vous pouvez assembler la bibliothèque avec la musique et les graphiques.

TRES IMPORTANT : veillez à ne pas dépasser 8440 octets de graphiques. Pour ce faire, vérifiez l'endroit où la balise "**_END_GRAPH**" est assemblée, qui doit être inférieur à 42040 (puisque $42040 - 33600 = 8440$ octets). S'il est assemblé à une adresse supérieure, vous "écrasez" des adresses nécessaires à l'interpréteur BASIC et l'ordinateur risque de se bloquer. Si vous avez besoin de plus de mémoire pour les graphiques, vous devez assembler les graphiques "supplémentaires" dans une zone de mémoire inoccupée, par exemple 22000, et utiliser une MEMOIRE 21999 dans votre programme, réduisant ainsi la mémoire disponible pour le BASIC.

8.2 Imprimer un sprite

Voyons les bases de l'impression d'un sprite. Supposons que vous ayez dessiné un soldat et que vous l'ayez inclus dans le fichier images_mygame.asm. Supposons qu'il s'agisse de votre première image et qu'elle porte donc l'identifiant 16.

En 8BP, vous disposez de 32 sprites (numérotés de 0 à 31). Vous pouvez assigner une image à n'importe quel sprite avec la commande |SETUPSP. Cette commande vous permet de modifier de nombreux attributs d'un sprite, et pas seulement son image. L'attribut du sprite que vous souhaitez modifier est le deuxième paramètre de la commande SETUPSP.

|SETUPSP, <sprite id>, <paramètre> , <valeur>, <sprite id>, <paramètre> , <valeur>, <valeur>.

Pour assigner une image, vous devez utiliser le paramètre 9. Un programme très simple qui imprime un sprite serait le suivant :

```
10 MÉMOIRE 23499
20 CALL &6B78 : REM installe les commandes RSX
30 DEFINT A-Z : variables numériques entières REM (plus rapide)
40 |SETUPSP,31,9,16 : REM attribue l'image 16 au sprite 31
50 x=40:y=100 : coordonnées REM à l'endroit où nous voulons imprimer
60 |LOCATESP,31,y,x : REM place le sprite 31
70 |PRINTSP,31 : REM imprime le sprite 31
```

Voici le résultat



Fig. 28 Impression d'un sprite à l'écran

Vous utiliserez souvent la commande **SETUPSP** et vous apprendrez progressivement à la connaître en profondeur. Les paramètres de SETUPSP ne sont pas des nombres quelconques. Il y a 7 valeurs possibles et elles sont les suivantes (0,5,6,7,8,9,15) :

- Paramètre 0 : modifie l'octet d'état du sprite
- Paramètre 5 : modifier Vy. Vx peut également être modifié en même temps si nous l'ajoutons à la fin en tant que paramètre supplémentaire (comme ceci : SETUPSP, <id>,5, Vy,Vx).
- Paramètre 6 : variations Vx
- Paramètre 7 : séquence de changement (prend les valeurs 0..31)
- Paramètre 8 : change frame_id (prend les valeurs 0..7)
- Paramètre 9 : changer l'image. L'image spécifiée peut faire partie de la liste initiale d'images du fichier images_mygame.asm,
- Paramètre 15 : chemin de changement (occupe 1 octet)

En lisant ce manuel, vous comprendrez la signification des attributs d'un sprite et vous saurez comment les utiliser en fonction de vos besoins.

8.3 Retournement des sprites

Vous devrez souvent dessiner des personnages marchant dans des directions différentes, avec des images différentes pour chaque cas. L'image du sprite dans la direction gauche sera l'image miroir de la direction droite. Vous pouvez définir deux images et les stocker en mémoire, mais depuis la version 33, il existe un moyen d'éviter la consommation de mémoire vive pour ces images. C'est ce qu'on appelle les images "retournées".



Fig. 29 Exemple d'images inversées

Une image "retournée" est une image miroir d'une autre image qui a été créée et incluse dans le fichier image. En définissant une image de cette manière, vous évitez de devoir la stocker. Pour ce faire, il suffit d'inclure une liste d'images retournées dans le fichier image (que j'appelle généralement "images_mygame.asm"). Vous trouverez au début du fichier une section délimitée par les balises "_BEGIN_FLIP_IMAGES" et "_END_FLIP_IMAGES" à cet effet.

```
;;
DÉBUT DU RETOURNEMENT DES IMAGES
Ici, on place des images qui sont définies comme d'autres images existantes, mais retournées horizontalement.
JOE_LEFT dw JOE_RIGHT ; joe_left sera la version inversée de joe_right

Je définis les images du soldat de gauche comme étant des images inversées
SOLDIER_L0 dw SOLDIER_R0 ;
SOLDADO_L1 dw SOLDADO_R1 ;
SOLDADO_L2 dw SOLDADO_R2 ;
SOLDADO_L1_UP dw SOLDADO_R1_UP
SOLDADO_L1_DOWN dw SOLDADO_R1_DOWN

FIN_FLIP_IMAGES
;-----
```

Les images retournées peuvent être utilisées comme des images normales. Vous découvrirez ci-dessous comment créer des séquences d'animation, que vous pouvez construire avec des images flippées ou non flippées. À toutes fins utiles, c'est comme si une image "retournée" était réelle, bien qu'il s'agisse d'une image "virtuelle", qui n'est pas stockée et qui est calculée comme une image miroir d'une image existante lorsqu'elle est imprimée. Les images retournées sont prises en charge en mode 0 et en mode 1.

L'inconvénient des images inversées est qu'elles sont plus chères à imprimer et qu'elles prennent 1,8 fois plus de temps qu'une impression normale, ce qui peut se traduire par un ralentissement de la vitesse de votre jeu. Si votre jeu est un jeu d'arcade (un shoot'em up) où vous avez besoin d'une vitesse maximale, je vous recommande de ne pas utiliser d'images massivement scintillantes. Cependant, dans les jeux d'aventure, les jeux de passage d'écran, les jeux de labyrinthe, etc. c'est un excellent choix. Dans tous les cas, essayez de les utiliser dans votre salle d'arcade, car s'il n'y a pas beaucoup de flips en même temps, la vitesse résultante peut être très acceptable.

J'ai procédé à un retournement horizontal et non à un retournement vertical parce que, normalement, un personnage marchant vers la gauche est l'image inversée du même personnage marchant vers la droite, tandis que le fait de marcher vers le haut montre le dos et le fait de marcher vers le bas montre la poitrine et le visage. Par conséquent, le retournement vertical n'est pas aussi utile que le retournement horizontal et, afin de réduire la taille du 8BP, je ne l'ai pas inclus dans ses capacités.

IMPORTANT : le retournement n'est pas applicable aux images de type segment qui peuvent être utilisées dans le mode pseudo-3D du 8BP.

8.4 Sprites avec écrasement

Depuis la version v22 de 8BP, il est possible d'éditer des sprites transparents, c'est-à-dire des sprites qui peuvent voler au-dessus d'un arrière-plan et le réinitialiser au passage. Pour ce faire, les sprites qui bénéficient de cette possibilité doivent être configurés avec un "1" dans le drapeau d'écrasement de l'octet de statut (bit 6). L'octet de statut sera expliqué en détail dans la section suivante. Voyons maintenant comment éditer un sprite doté de cette capacité avec SPEDIT.

De nombreux jeux utilisent une technique appelée "double buffering" pour pouvoir restaurer l'arrière-plan lorsqu'un sprite se déplace sur l'écran. Cette technique consiste à avoir une copie de l'écran (ou de la zone de jeu) dans une autre zone de la mémoire, de sorte que même si nos sprites détruisent l'arrière-plan, nous pouvons toujours regarder dans cette zone pour voir ce qu'il y avait en dessous et le restaurer. En fait, c'est le principe de base, mais c'est un peu plus complexe. L'image est imprimée dans le double tampon (également appelé "backbuffer") et lorsqu'elle est entièrement imprimée, elle est soit vidée sur l'écran, soit l'adresse de départ de la mémoire vidéo passe de l'adresse d'origine de l'écran à la nouvelle, l'adresse du double tampon. La commutation est instantanée (selon le type de machine). Pour construire la trame suivante, l'adresse d'origine de l'écran est utilisée là où la mémoire vidéo ne pointe plus. C'est là que la nouvelle trame est construite, puis rebasculée, alternativement, à chaque trame. Ces techniques, bien qu'elles fonctionnent très bien, présentent quelques inconvénients pour nos besoins : elles prennent plus de temps au processeur et consomment beaucoup plus de mémoire (jusqu'à 16 Ko supplémentaires), ce qui nous laisse très peu de mémoire pour notre programme BASIC. Si un jeu est entièrement développé en assembleur, ce n'est pas si grave, car 10 Ko d'assembleur suffisent largement, mais 10 Ko de BASIC, c'est trop peu. Certains jeux vidéo réduisent la surface de jeu afin de ne pas utiliser autant de mémoire, mais cela les rend un peu plus pauvres.

La solution adoptée par 8BP est inspirée par le programmeur Paul Shirley (auteur de "Mission Genocide"), mais elle est légèrement différente. Je raconterai directement l'histoire du 8BP :



Fig. 30 Sprites avec écrasement en 8BP

L'idée est que **le fond n'est jamais détruit par les sprites qui passent dessus**, et qu'il n'est donc pas nécessaire de le sauvegarder. Cette apparente "magie" a sa logique : elle consiste à "cacher" la couleur de l'arrière-plan dans la couleur du sprite qui est peint par-dessus.

Sur l'AMSTRAD, un pixel de mode 0 est représenté par 4 bits, ce qui permet d'obtenir jusqu'à 16 couleurs différentes sur une palette de 27 couleurs. Ainsi, si nous utilisons un bit pour la couleur d'arrière-plan et 3 bits pour les couleurs du sprite, nous aurons un total de 2 couleurs d'arrière-plan.

7 couleurs + 7 couleurs + 1 couleur pour indiquer la transparence = 9 couleurs au total. Cela permet de "cacher" la couleur de fond dans la couleur du sprite, au prix d'une réduction du nombre de couleurs de 16 à 9 seulement. Cependant, certains éléments ornementaux de l'écran de jeu peuvent avoir plus de couleur, car les sprites ne passeront pas dessus (comme les feuilles des arbres ou le toit dans l'exemple ci-dessous), ce qui nous permet d'obtenir une certaine quantité de couleur dans notre jeu.

Pour éditer ce type de sprites, nous devons utiliser une palette appropriée, de 9 couleurs, où pour chaque couleur de sprite deux codes binaires sont utilisés (correspondant au 0 et au 1 du bit d'arrière-plan). Dans SPEDIT, si vous choisissez l'option de palette "2", vous aurez une palette définie de cette manière, bien que vous puissiez la modifier à votre guise. Elle est construite comme suit :

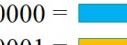
```

2300 REM ----- PALETA sprites transparentes MODE 0-----
2301 INK 0,11: REM azul claro
2302 INK 1,15: REM naranja
2303 INK 2,0 : REM negro
2304 INK 3,0 :
2305 INK 4,26: REM blanco
2306 INK 5,26:
2307 INK 6,6: REM rojo
2308 INK 7,6:
2309 INK 8,18: REM verde
2310 INK 9,18:
2311 INK 10,24: REM amarillo
2312 INK 11,24:
2313 INK 12,4: REM magenta
2314 INK 13,4:
2315 INK 14,16 : REM naranja
2316 INK 15, 16:
2317 AMARILLO=10
2420 RETURN

```

Fig. 31 Exemple de palette d'écrasement

Comme vous pouvez le voir, après les couleurs 0 et 1, toutes les couleurs sont répétées deux fois. Vous pouvez construire votre propre palette de cette manière. Vous pouvez vous aider en consultant l'annexe de ce manuel consacrée à la palette de couleurs.

000 1	=		Color de fondo	Paleta ejemplo
110 0	=		Color de sprite	0000 =  0001 = 
<i>Cuando el sprite se imprime:</i>				
Fondo OR sprite = 1101 =			1100 =  1101 = 	
<i>Cuando el sprite se marcha:</i>				
Pixel OR 0001 = 0001 =				
<i>El fondo nunca fue destruido, estaba "escondido" en el sprite</i>				

La technique pourrait être pour résumer en disant que **l'arrière-plan n'est jamais réellement détruit par les sprites, mais qu'il est "caché"** dans les sprites eux-mêmes qui sont imprimés sur l'arrière-plan.

Fig. 32 Mécanisme d'écrasement sur le 8BP

Avec l'éditeur SPEDIT, vous pouvez modifier la palette à votre guise sans avoir à l'éditer manuellement avec les commandes INK, et vous pouvez l'exporter pour la copier dans nos programmes BASIC. L'exportation se fait en envoyant les commandes INK qui composent la palette à l'imprimante (l'imprimante est redirigée vers un fichier de winape). Nous disposons des touches z/x pour modifier la palette et de l'option "i" pour l'exporter vers le fichier de sortie. Voici un exemple de ce qu'il exporte (il s'agit d'une palette **BEGIN ERASER**) :

```
INK 0 , 1
INK 1 ,
INK ,
INK ,
INK , 26
INK 5 , 0
INK ,
INK , 8
INK 8 , 10
INK ,
INK 10 , 14
INK , 16
INK , 18
INK , 22
INK , 0
INK ---- END PALETA -----
```

En appuyant sur la touche "i", vous enregistrez également le fichier "pal.dat" sur le disque (le .dsk) afin de pouvoir le charger ultérieurement en choisissant l'option 3 pour répondre à la question du choix de la palette.

Les sprites utilisés pour construire les dessins d'arrière-plan ne peuvent avoir que les couleurs 0 et 1, mais les sprites utilisés pour l'ornementation, où les sprites en mouvement ne passeront pas, peuvent utiliser les 9 couleurs.

Vous pouvez également augmenter la couleur du paysage en utilisant des éléments de sprites au lieu des arrière-plans, comme le chaudron vert dans l'exemple ci-dessus. Vous obtiendrez ainsi des résultats très colorés.

L'encre 0001 a une utilisation "spéciale". Si vous éditez un sprite qui n'utilise pas le drapeau d'écrasement, l'encre 1 sera simplement une couleur. Mais si vous éditez un sprite avec un drapeau d'écrasement actif dans son octet de statut, lors de l'impression, ces pixels ne seront pas peints, respectant ce qui se trouve en dessous. Cela permet aux collisions entre sprites de ne pas être "rectangulaires", mais de préserver la forme du sprite.

code	Signification
0000	Couleur d'arrière-plan 1. Si un sprite l'utilise et que vous activez le drapeau d'écrasement, cela signifie qu'il y a "transparence", c'est-à-dire que l'impression se fait en remettant à zéro le drapeau d'écrasement. fonds
0001	Couleur de fond 2. Si un sprite l'utilise et que vous activez le drapeau d'écrasement, il ne s'agit plus d'une couleur, mais de "ne pas imprimer". Tout ce qui se trouve dans ce pixel est respecté, par exemple un pixel coloré par un autre sprite précédemment imprimé avec l'attribut que nous nous chevauchons.
0010	couleur du sprite 1
0011	
0100	

code	sens
0110	couleur du sprite 3
0111	
	couleur 4 du sprite
1001	
1010	couleur du sprite 5
1011	
	couleur du sprite 6
1101	
1110	couleur du sprite 7
1111	

9 couleurs au total :

- 2 en arrière-plan
- 7 pour les sprites (en fait 8 mais un -000- signifie la transparence)
- Les éléments ornementaux peuvent utiliser les 9.

Ensuite, je vais vous montrer un sprite où j'ai peint à l'encre 0001 ce qui ne va pas être peint, c'est-à-dire où le fond ne va même pas être restauré, car avec le reste des pixels à 0000 il suffit d'effacer la trace du sprite pendant qu'il se déplace.

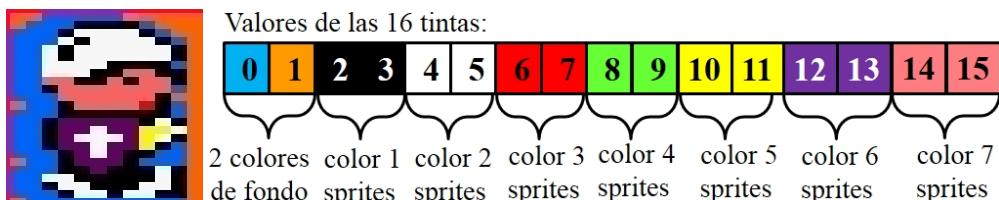


Fig. 33 sprite et palette conçus pour l'écrasement

IMPORTANT, LORSQUE VOUS ÉDITEZ VOS SPRITES AVEC OVERWRITE :

N'utilisez pas les bits d'arrière-plan pour les encres de vos sprites, à moins que vous ne recherchez des effets spéciaux tels que décrits plus loin dans ce chapitre. Si vous ne respectez pas cette règle, vous risquez de constater des effets "bizarres". En d'autres termes :

- Si l'arrière-plan est en 1 bit, colorez vos sprites avec des encres se terminant par 0 (c'est-à-dire les encres 2, 4, 6, 8, 10, 12, 14).
- Si l'arrière-plan a 2 bits, colorez vos sprites avec des encres se terminant par 00 (c'est-à-dire les encres 0100, 1000, 1100 qui sont respectivement les encres 4, 8 et 12).

Comme vous pouvez l'imaginer, dans le cas du chaudron, étant donné qu'il s'agit d'un

sprite qui ne bouge pas et qui ne s'efface donc pas, tout son contour est peint avec l'encre 0001. Cela permet d'obtenir des collisions parfaites, sans que les formes rectangulaires ne fassent ressortir que les sprites sont en fait des

sont des rectangles. Le résultat final est illustré ci-dessous dans une collision multiple.



Fig. 34 Collision multiple, effet d'encre 0001

Les opérations d'impression avec ce mécanisme sont très rapides, sans qu'il soit nécessaire de définir ce que l'on appelle des "masques de sprites". Les masques de sprites sont des bitmaps de la taille d'un sprite qui servent à accélérer les opérations d'impression. Dans le cas présent, ils ne sont pas nécessaires. La figure suivante représente un masque typique associé à un sprite. L'opération ET est généralement effectuée entre l'arrière-plan et le masque, puis l'opération OU est effectuée avec le sprite. En 8BP, c'est plus rapide, car le sprite ne touche pas le bit destiné à l'arrière-plan, donc l'opération OU entre l'arrière-plan et le sprite respecte l'arrière-plan tout en peignant le sprite. Si vous ne comprenez pas très bien cela, ne vous inquiétez pas, il n'est pas important de le comprendre car ce n'est pas nécessaire en 8BP.

sprite	mask	Metodo convencional:
0 2 2 0	1 0 0 1	Se imprime
2 3 3 2	0 0 0 0	Fondo AND mask OR sprite
0 2 2 0	1 0 0 1	
0 2 2 0	1 0 0 1	
0 0 0 0	1 1 1 1	

Fig. 35 En 8BP, aucun masque n'est nécessaire.

L'impression de sprites avec le drapeau d'écrasement actif est plus coûteuse que l'impression sans écrasement. Bien qu'elle ne nécessite pas de masque et soit très rapide, elle prend environ 1,6 fois plus de temps que l'impression d'un sprite sans overwrite. Pour cette raison, n'utilisez cette option qu'en cas de nécessité, et ne l'utilisez pas si votre jeu ne comporte pas de dessin d'arrière-plan que les sprites doivent respecter. La combinaison de l'écrasement et du retournement est encore plus coûteuse (elle consomme 2,2 fois le temps d'une impression normale sans écrasement ni retournement) ; tenez-en compte dans vos jeux.

8.4.1 Utilisation de l'écrasement pour améliorer les chevauchements de sprites

Une caractéristique très utile de l'écrasement est qu'il permet aux chevauchements entre les sprites d'être "parfaits", car si, lorsque vous éditez le sprite (que vous allez écraser), vous utilisez l'encre 1 autour de lui, lorsque vous l'imprimez, tous les pixels qui ont

l'encre 1 seront "écrasés".

deviendra "transparent", c'est-à-dire que si vous avez précédemment peint un autre sprite, ses pixels seront respectés, ce qui entraînera des chevauchements "parfaits". Cette fonction d'écrasement des sprites peut vous intéresser, même si votre jeu ne nécessite pas d'écrasement parce qu'il a un arrière-plan noir ou monochrome.

Seuls les pixels que vous avez dessinés avec zéro encre seront effacés (en réinitialisant l'arrière-plan). Dans l'exemple de la balle, les pixels à zéro sont les pixels arrière afin que vous puissiez l'effacer lorsque vous la déplacez vers la droite.

L'exemple suivant permet de le comprendre parfaitement. Vous perdez de la couleur parce qu'il n'y a que 9 couleurs, mais les chevauchements entre les sprites sont très bons. Attention, vous perdez également en vitesse d'impression.

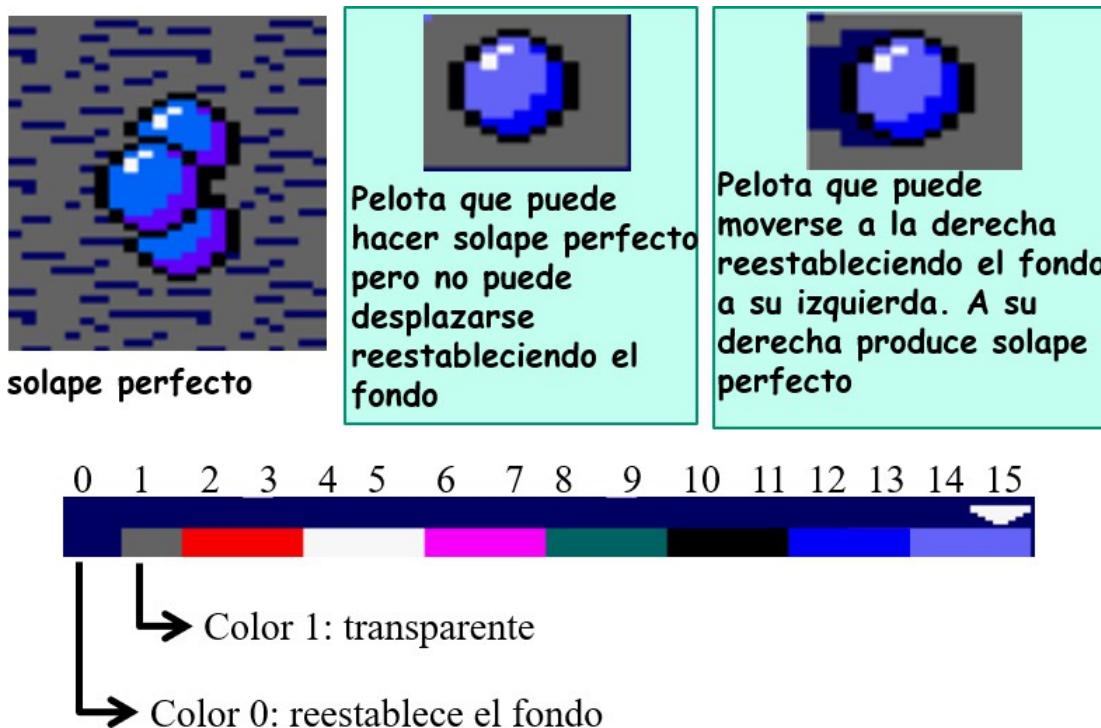


Fig. 36 Des chevauchements parfaits

8.4.2 Ecrasement avec 4 couleurs d'arrière-plan

Depuis la version V33, il est possible de choisir le nombre de bits utilisés pour le fond en invoquant la commande |PRINTSP en spécifiant le Sprite 32, qui n'existe pas. Vous pouvez choisir 1 ou 2 bits pour l'arrière-plan, ce qui permet respectivement 2 et 4 couleurs d'arrière-plan.

|PRINTSP, 32, <numéro du bit d'arrière-plan>.

Exemples :

 PRINTSP, 32, 1 : ' avec un arrière-plan de 1 bit, nous avons 2 couleurs pour l'arrière-plan
 PRINTSP, 32, 2 : ' avec un arrière-plan de 2 bits, nous disposons de 4 couleurs pour l'arrière-plan

Une fois cette commande invoquée, la bibliothèque 8BP est configurée pour prendre en compte le nombre de bits à utiliser comme bits d'arrière-plan. Si nous définissons 2 bits d'arrière-plan, notre palette de couleurs devra être cohérente avec cette circonstance. Un

exemple est donné ci-dessous

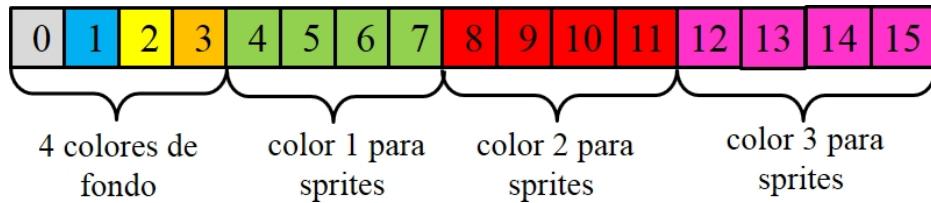


Fig. 37 Exemple de palette avec quatre couleurs d'arrière-plan (arrière-plan 2 bits)

Dans cet exemple, vous avez 4 couleurs d'arrière-plan et 3 couleurs de sprites. Tout comme lorsque vous utilisez 1 bit pour l'arrière-plan, lorsque vous définissez vos sprites, vous devez garder à l'esprit que 0 signifie transparence et 1 signifie que l'arrière-plan n'est pas réinitialisé, ce qui permet d'obtenir des sprites de forme non rectangulaire. Dans l'exemple suivant, les trois couleurs choisies pour les sprites sont le noir, le vert clair et le blanc.



Fig. 38 Exemple de jeu de palette avec jusqu'à quatre couleurs d'arrière-plan (2 bits)

Bien qu'avec deux bits pour l'arrière-plan, vous puissiez obtenir de plus belles décos, l'inconvénient est qu'il ne vous reste que 3 couleurs pour les sprites, alors que si vous utilisez un bit pour l'arrière-plan, vous avez jusqu'à 7 couleurs pour vos sprites.

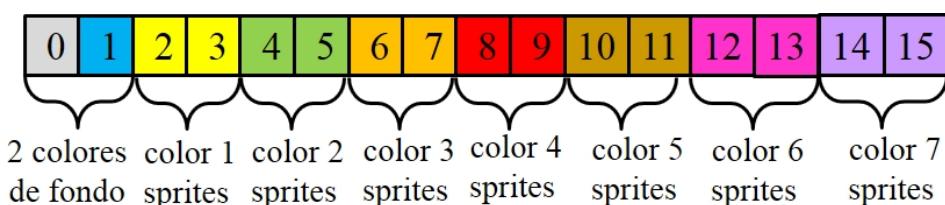


Fig. 39 Exemple de palette avec deux couleurs d'arrière-plan (arrière-plan 1 bit)

8.4.3 Écasser dans le MODE 1

Depuis la version V34 du 8BP, il est possible d'utiliser des sprites avec écrasement dans le MODE 1. Nous avons ici une très forte limitation car, bien que nous ayons deux couleurs pour l'arrière-plan, nous n'avons qu'une seule couleur pour les sprites.

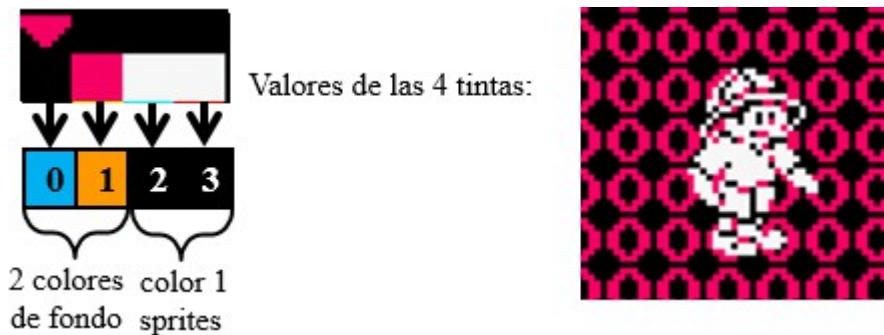
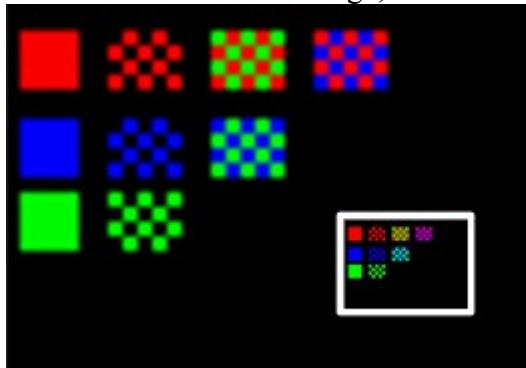


Fig. 40 Exemple de palette en mode 1

Il est facile de faire l'erreur de penser que les sprites ont n'importe quelle couleur et, de plus, qu'ils sont noirs. Ce n'est pas le cas. Le noir est une autre couleur et, à ce titre, il doit consommer deux encres avec ce mécanisme. Nous n'avons que deux encres pour le sprite et nous les avons utilisées sur le blanc (dans cet exemple). Comme vous pouvez le voir, le personnage qui n'est pas blanc est transparent et non noir.

Malgré cette limitation stricte, si vous travaillez dur, vous pouvez créer des sprites très attrayants dans le MODE 1, et si vous changez les couleurs de fond sur chaque écran, et utilisez des mélanges de couleurs (treillis) sur les marqueurs de jeu, vous pouvez obtenir un résultat très satisfaisant.

En utilisant le mélange dans le MODE 1, vous pouvez simuler 10 couleurs avec seulement 4. Voici un exemple. Les couleurs du treillis sont fusionnées et, par exemple, le vert + le rouge donnent du jaune. Vous ne le verrez peut-être pas de façon aussi convaincante sur cette image, mais sur l'écran de votre Amstrad, vous le verrez bien.



8.4.4 Comment peindre des sprites "derrière" l'arrière-plan

Le même mécanisme d'impression des sprites sur l'arrière-plan peut être utilisé pour peindre derrière l'arrière-plan.

Comme nous l'avons déjà vu, pour imprimer devant l'arrière-plan, nous utilisons des bits qui ne sont pas utilisés dans l'arrière-plan, de sorte que, bien que nous réduisions le nombre de couleurs, nous n'endommageons pas le(s) bit(s) de l'arrière-plan. Si nous utilisons un bit pour l'arrière-plan, nous devrons utiliser deux encres pour représenter la même couleur de sprite : une avec le bit d'arrière-plan à zéro et une avec le bit d'arrière-plan à 1.

Cependant, si au lieu d'attribuer la même couleur à ces deux encres, nous attribuons la même couleur que l'arrière-plan à celle dont le bit d'arrière-plan est fixé à 1, alors si le sprite chevauche l'arrière-plan, la couleur de l'arrière-plan sera visible, donnant l'impression que le sprite passe derrière lui. Cela fonctionne aussi bien dans le MODE 0

que dans le MODE 1.

Dans l'exemple suivant, un bit est utilisé pour l'arrière-plan, qui consiste en des lettres jaunes sur un fond noir. Les sprites sont des "pièces de monnaie" qui, comme vous pouvez le voir, ont apparemment été peintes derrière le fond.

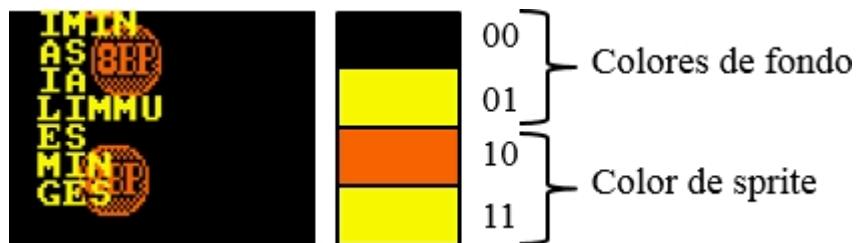


Fig. 41 Sprites imprimés "derrière" les lettres

8.4.5 Comment utiliser plus de couleurs avec l'écrasement

Si vous avez fait vos premiers essais avec l'écrasement et que vous avez besoin de plus de couleurs dans votre jeu, il y a trois façons d'y parvenir, mais vous devez comprendre la méthode 8BP :

- 1) Utiliser certains sprites avec écrasement et d'autres sans écrasement
- 2) Utiliser des sprites dont la couleur dépend de l'arrière-plan (couleur conditionnelle).
- 3) Utiliser un sprite comme arrière-plan

Examinons ces trois "trucs" un par un :

Utiliser certains sprites avec écrasement et d'autres sans écrasement :

Il s'agit de l'astuce la plus simple et la plus couramment utilisée. Supposons qu'un seul de vos sprites nécessite un écrasement et consomme 3 couleurs. Cela signifie que vous devez allouer 6 encres à ce sprite. Toutefois, si les autres sprites ne nécessitent pas d'écrasement, vous pouvez les imprimer sans écrasement et utiliser plus de couleurs, c'est-à-dire que vous devez allouer 6 encres à ce sprite :

- 2 encres pour le fond (2 couleurs)
- 6 encres pour le sprite avec écrasement (3 couleurs)
- 8 encres pour les autres sprites (8 couleurs)

Dans ce cas, vous pouvez utiliser un total de $2+3+8 = 13$ couleurs !!!!. Il suffit d'activer le drapeau d'écrasement sur le sprite qui en a besoin et de le laisser inactif sur les autres sprites. Sur les sprites qui n'utilisent pas le drapeau overwrite, vous pouvez utiliser les 13 couleurs, sur le sprite avec overwrite, vous devez utiliser 3 couleurs et sur l'arrière-plan, vous devez utiliser 2 couleurs.

D'autres exemples sont possibles. Par exemple, si les sprites avec écrasement ont besoin de 4 couleurs, ils utiliseront 8 encres. En outre, nous aurons 2 encres pour l'arrière-plan et les 6 encres restantes peuvent identifier 6 couleurs différentes, c'est-à-dire que nous pouvons utiliser un total de 12 couleurs.

Utiliser des sprites dont la couleur dépend de l'arrière-plan (couleur conditionnelle) :

Elle consiste à utiliser une palette où, au lieu de répéter chaque couleur, on utilise deux couleurs différentes. Dans ce cas, lorsque le fond est à zéro, on aura un sprite d'une

couleur et lorsque le fond est à 1, on aura une autre couleur. Cela peut être utile pour dessiner des oiseaux blancs volant au-dessus d'un ciel bleu (couleur 0) tandis que des ours rouges marchent sur un sol marron (couleur 1). En d'autres termes, nous pouvons utiliser plus de couleurs tant que la texture du ciel ou du sol

L'ours n'aura pas trop de changements de couleur, car chaque fois qu'il passera sur un pixel de fond bleu, nous verrons un pixel blanc, et si un oiseau passe sur un pixel de fond marron, nous verrons un pixel rouge. Voyons un exemple :

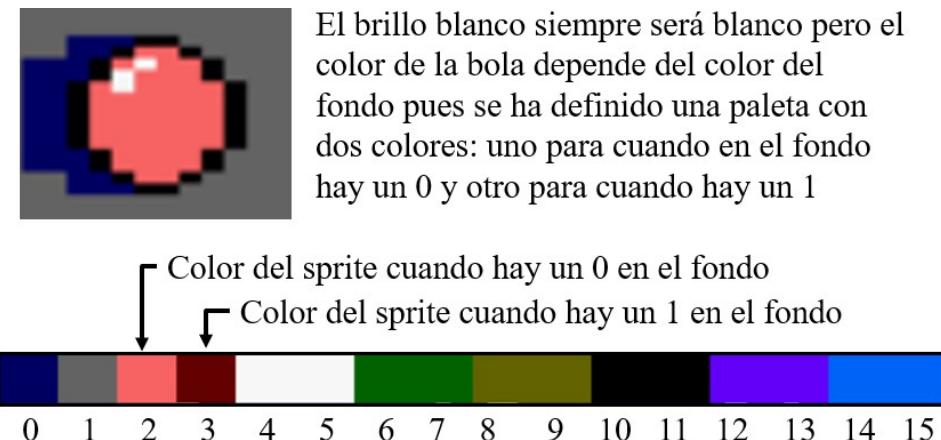


Fig. 42 Sprites créés avec une couleur "conditionnelle"

Une fois que le sprite avec la couleur conditionnelle a été créé, nous pouvons voir dans l'image suivante l'effet qu'il produit lorsqu'il est imprimé dans deux zones de l'écran, l'une avec le fond 0 et l'autre avec le fond 1.

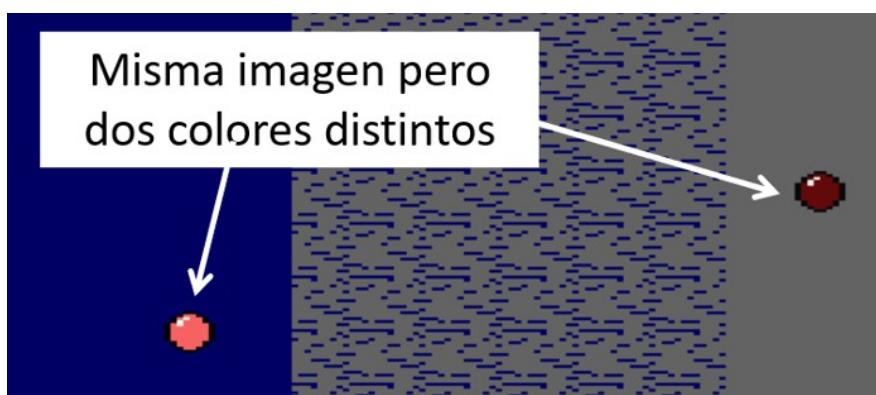


Fig. 43 Effet de couleur "conditionnel"

Utiliser un sprite comme arrière-plan :

Cette astuce permet d'utiliser jusqu'à 16 couleurs d'arrière-plan avec l'écrasement. L'astuce est basée sur le fait que, lorsque l'écrasement est activé sur un sprite, tous les pixels définis avec "1" respecteront la valeur du pixel à l'écran, qu'il s'agisse d'un pixel d'un arrière-plan réel ou d'un sprite imprimé précédemment. Nous imprimons à chaque cycle à la fois le sprite qui est l'arrière-plan et les sprites qui seront surimprimés. Ou du moins à chaque cycle où nous les déplaçons.

Le sprite que nous utilisons comme arrière-plan, nous l'imprimerons **sans activer l'écrasement**, tandis que les sprites que nous imprimons par-dessus l'auront activé. Le seul inconvénient est que si le sprite d'arrière-plan est trop grand, il prendra beaucoup de temps à s'imprimer et le nombre d'images par seconde du jeu ne permettra pas de faire un jeu d'arcade, bien qu'il puisse être utile pour les jeux d'aventure. **N'oubliez pas non plus qu'en 8BP, la hauteur maximale d'un sprite est de 127 lignes.** La largeur maximale n'est pas un problème puisqu'elle est également de 127 octets et que l'écran ne fait que 80 octets de large.

Nous allons créer le sprite que nous allons imprimer sur le dessus, entouré de uns, sans zéros, car il ne réinitialisera pas l'arrière-plan lorsqu'il se déplacera. Chaque cycle imprimera à la fois le sprite d'arrière-plan et nos petits sprites par-dessus. Dans cet exemple, nous avons dessiné une sorte de pointeur ou de flèche que vous contrôlez avec le clavier, sur un arrière-plan qui occupe la moitié d'un écran, soit 8 Ko (80 octets de large x 100 lignes).

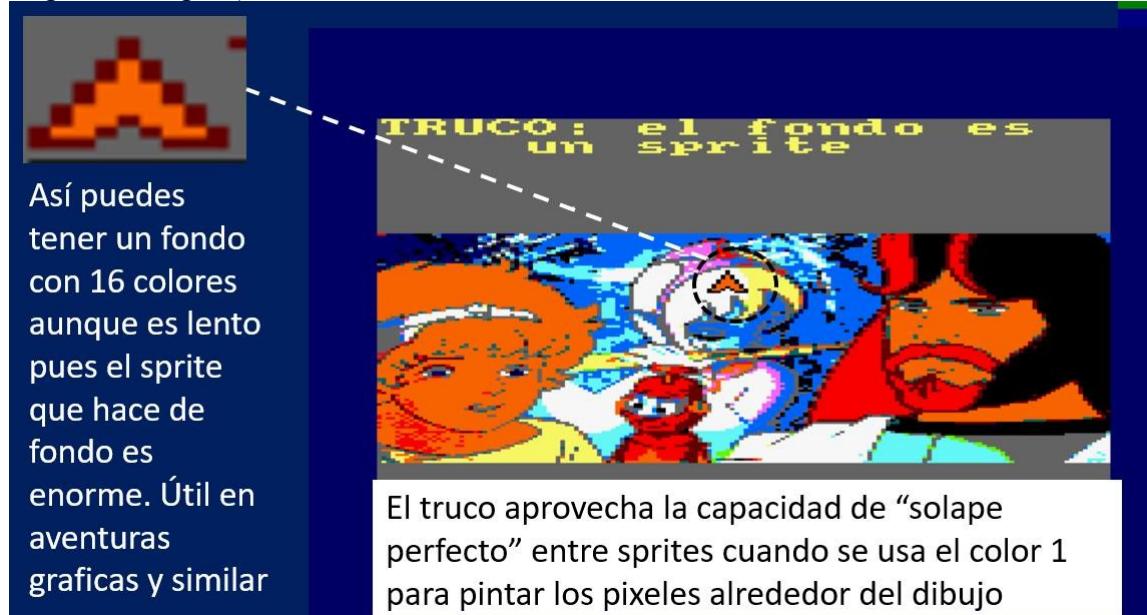


Fig. 44 Un sprite en arrière-plan

Comme le sprite pointeur est écrasé, il subira les effets de l'astuce précédente (couleur conditionnelle) à moins que vous ne définissiez des couleurs "doubles" pour ne pas subir cet effet. Autrement dit, si par exemple vous répétez une couleur pour ne pas subir cet effet sur un sprite, vous aurez une palette de 15 couleurs différentes, et non 16. Autrement dit, vous devez d'une manière ou d'une autre appliquer le principe de la première astuce : un sprite sans érasement comme arrière-plan et un ou plusieurs sprites avec érasement qui sont imprimés par-dessus. Plus l'arrière-plan est coloré, moins les sprites avec érasement seront colorés. Par exemple (d'autres combinaisons sont possibles) vous pouvez utiliser :

- **Fond** : 2 couleurs de fond + 8 couleurs + 3 couleurs répétées = 13 couleurs
- **Sprites en surimpression** : 6 encres = 3 couleurs répétées

Une façon d'accélérer la vitesse du sprite d'arrière-plan "géant" est de le diviser en bandes. Par exemple, en 8 sprites horizontaux de hauteur 16. Les sprites étirés horizontalement s'impriment plus rapidement que les sprites étirés verticalement. Dans ce cas, vous ne devez imprimer que la bande ou la paire de bandes qui intersectent votre sprite de pointage.

8.5 Sprites avec images d'arrière-plan

Les images d'arrière-plan sont une fonctionnalité de 8BP V42. L'idée est que dans un défilement, des arbres ou des maisons peuvent passer sous votre avion sans le faire scintiller. Même si l'avion a une impression transparente et est peint sur l'arrière-plan, si l'arrière-plan se déplace, il ne respectera pas le sprite et provoquera un scintillement. Grâce à cette nouvelle fonctionnalité, vous pouvez créer des jeux avec défilement dans

lesquels votre protagoniste ou votre navire passe au-dessus de choses **sans scintillement**. Pour mieux comprendre cela, vous devez comprendre le mécanisme de défilement du 8BP, qui est expliqué ci-dessous.

La série de démonstrations 8BP comprend une démonstration de l'effet de l'utilisation d'images d'arrière-plan dans votre défilement.



Maisons de l'image de fond

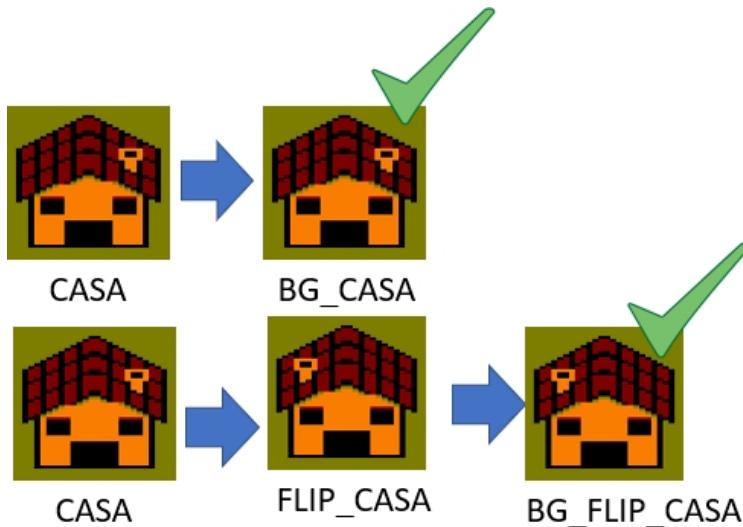
Sans cette possibilité, (avant la version V42 de 8BP) les sprites pouvaient être écrasés sans scintillement tant que l'arrière-plan ne bougeait pas, mais le mouvement de l'arrière-plan (le défilement) provoquait un scintillement inévitable. Depuis la version V42, il est possible de créer des images d'arrière-plan et de les affecter aux sprites.

Il suffit d'attribuer au Sprite une image de fond qui, dans le fichier image, doit être contenue dans les balises :

```
BG_HOME DW  
HOME DW HOME  
BG_HOUSE_FLIP DW HOUSE_FLIP  
FIN DE L'IMAGE
```

L'image **CASA** est une image normalement créée avec une particularité : elle n'utilise que les couleurs de fond. Ainsi, lorsqu'un Sprite dans un jeu se voit attribuer l'image **BG_CASA**, il se verra automatiquement attribuer un type spécial de transparence, dans lequel seuls les bits représentant les couleurs d'arrière-plan seront modifiés et tout Sprite situé au-dessus du dessin sera respecté, évitant ainsi le scintillement.

Le processus de création d'images d'arrière-plan est très simple, mais aussi très strict : à partir d'une image (en utilisant uniquement les encres d'arrière-plan), vous pouvez créer une image d'arrière-plan. Si vous souhaitez utiliser une image retournée, vous devez d'abord retourner l'image, puis construire l'image de fond avec l'image retournée.



Si vous souhaitez retourner une image d'arrière-plan, vous devez d'abord la retourner à l'aide de la section **FLIP_IMAGES**, puis l'utiliser pour créer une image d'arrière-plan. C'est dans cet ordre qu'il faut procéder et non dans l'autre. En d'autres termes, vous ne pouvez pas créer une image, créer une image d'arrière-plan avec cette image, puis essayer de retourner une image d'arrière-plan.

Les images d'arrière-plan, lorsqu'elles sont affectées à un Sprite, sont imprimées avec cette transparence spéciale, que le Sprite ait ou non le drapeau de transparence affecté dans son octet de statut (nous allons voir maintenant ce que c'est).

IMPORTANT : les images d'arrière-plan sont chères à imprimer, et si vous les retournez, elles sont encore plus chères. Si votre jeu comporte un défilement, essayez de les utiliser pour les éléments que votre personnage ou vos ennemis survoleront. Par exemple, pour accélérer le défilement, utilisez des images normales sur les maisons ou les rochers qui apparaissent sur les côtés de votre défilement et qui ne seront pas souvent recouverts par des sprites.

8.6 Table d'attributs des sprites

Les sprites sont stockés dans un tableau contenant un total de 32 sprites.

Chaque entrée de la table contient tous les attributs du sprite et occupe 16 octets pour des raisons de performance, puisque 16 est un multiple de 2 et que cela permet d'accéder à n'importe quel sprite avec une multiplication très peu coûteuse. La table est située à l'adresse mémoire 27000, et on peut donc y accéder depuis BASIC avec PEEK et POKE, mais il existe aussi des commandes RSX pour manipuler les données de cette table, telles que |SETUPSP ou |LOCATESP.

Les sprites disposent d'un ensemble de paramètres, dont le premier est l'octet des drapeaux d'état. Dans cet octet, chaque bit représente un drapeau et chaque drapeau signifie une chose, à savoir si le sprite est pris en considération lors de l'exécution de certaines fonctions. Le tableau suivant résume ce qui se passe lorsqu'ils sont actifs (à "1")

7 ROUTEALL lo ruta	6 Sobre- escritura	5 COLSPALL collider	4 MOVERALL lo mueve	3 AUTOALL lo mueve	2 ANIMALL lo anima	1 COLSP collided	0 PRINTSPALL lo imprime
--------------------------	--------------------------	---------------------------	---------------------------	--------------------------	--------------------------	------------------------	-------------------------------

Fig. 45 drapeaux dans l'octet de statut

Pour comprendre la puissance de ces drapeaux, prenons quelques exemples :

- **Bit 0** : drapeau d'impression : notre personnage ou les vaisseaux ennemis l'auront activé et à chaque cycle de jeu nous invoquerons |PRINTSPALL et ils seront tous imprimés en même temps.
- **Bit 1** : drapeau de collision : un fruit ou une pièce de monnaie, par exemple, peut ne pas avoir de drapeau d'impression, mais peut avoir un drapeau de collision. Ce drapeau signifie qu'un sprite "entrant en collision" peut entrer en collision avec le sprite dont le drapeau est actif.
- **Bit 2** : drapeau d'animation automatique : il est pris en compte dans |ANIMALL. Dans le cas du personnage, je recommande de le désactiver, car si je reste immobile, je n'ai pas besoin de changer de frame.
- **Bit 3** : Indicateur de mouvement automatique. Se déplace uniquement lorsque l'option AUTOALL est invoquée, en tenant compte de sa vitesse. Utile pour les météores et les gardes qui vont et viennent.
- **Bit 4** : indicateur de mouvement relatif. Tous les sprites dotés de cet indicateur se déplacent en même temps lorsqu'ils invoquent "|OVERALL, dy, dx", ce qui est très utile pour les navires en formation et les arrivées de planètes. Il peut également être utilisé pour simuler un défillement si vous laissez votre personnage au centre et que vous appuyez sur les commandes pour faire défiler les maisons ou les éléments environnants. Vous aurez l'impression que votre personnage se déplace dans un territoire.
- **Bit 5** : indicateur de collision. Tous les sprites ayant ce drapeau actif sont pris en compte par la fonction |COLSPALL, lors de la détection de leur éventuelle collision avec le reste des sprites.
- **Bit 6** : drapeau d'écrasement : si ce drapeau est actif, le sprite pourra se déplacer sur un arrière-plan, en le réinitialisant au passage. Il s'agit d'une option avancée qui implique l'utilisation d'une palette de couleurs spécialement préparée, plus limitée en nombre de couleurs. L'écrasement est à ce "prix".
- **Bit 7** : drapeau de chemin : si ce drapeau est activé, la commande |ROUTEALL vous permet de déplacer un sprite de manière cyclique à travers un chemin que vous définissez, défini par une série de segments. La commande |ROUTEALL sait dans quel segment et dans quelle position se trouve chaque sprite et si elle atteint un changement de segment, elle modifie la vitesse du sprite en fonction des conditions du segment suivant.
|ROUTEALL ne déplace pas le sprite, il modifie seulement sa vitesse. Pour le déplacer, il doit être utilisé en conjonction avec |AUTOALL.

Exemples d'affectation des valeurs de l'octet d'état :

Ennemi typique : un sprite à imprimer à chaque cycle, avec détection des collisions avec d'autres sprites et animation obligatoire :

$$\text{status} = 1(\text{bit 0}) + 2 (\text{bit1}) + 4 (\text{bit 2}) = 7 = \&x0111$$

Une maison qui bouge en même temps que nous bougeons : un sprite qui est imprimé à chaque cycle, mais sans détection de collision et sans mouvement relatif.

status = 1(bit 0) + 0 (bit1) + 0 (bit 2) + 0 (bit 3) + 16 (bit 4) = 17 =&x10001

Un fruit bonus : il s'agit d'un sprite qui n'est pas imprimé à chaque cycle, mais qui dispose d'une détection de collision.

$$\text{statut} = 0(\text{bit } 0) + 2 (\text{bit}1) = 2 = \&x10$$

Un navire qui suit une trajectoire prédéfinie. Il aura besoin du drapeau de trajectoire, du drapeau de mouvement automatique, du drapeau d'animation, du drapeau de collision et du drapeau d'impression. Cette fois-ci, je vais le mettre directement en binaire. Comme vous pouvez le voir, j'ai mis le bit 7 à 1, puis il y a 3 zéros car je n'ai pas mis les drapeaux d'érassement, de collision ou de mouvement relatif et enfin j'ai mis 4 drapeaux actifs correspondant respectivement aux drapeaux de mouvement automatique, d'animation, de collision et d'impression.

$$\text{statut}=10001111$$

La table d'attributs des sprites se compose de 32 entrées de 16 octets chacune, commençant à l'adresse 27000.

La raison d'avoir 16 octets n'est autre que la performance, puisque calculer l'adresse du sprite N implique de multiplier par 16, ce qui, étant un multiple de 2, peut être fait avec un décalage. Ceci est utile pour les opérations impliquant un seul sprite. Pour les opérations qui traversent la table des sprites (telles que |PRINTSPALL ou |COLSP), la table est parcourue en interne avec un index auquel 16 est ajouté pour passer d'un sprite au suivant. L'addition est la plus rapide dans ce cas.

Les attributs de chaque sprite sont les suivants :

attribut	Octet	Longueur (octets)	Signification
statut	0	1	Octet contenant les indicateurs d'état pour les opérations PRINTSPALL, COLSPALL, ANIMALL, AUTOALL, MOVERALL, COLSPALL et ROUTEALL.
Y	1		Coordonner Y [-32768..32768] les valeurs correspondant à l'intérieur de l'écran sont [0..199].
X			Coordonnée X en octets[-32768..32768] les valeurs correspondantes à l'intérieur de l'écran sont [0..79].
Vy	5	1	Passage au mouvement automatique
Vx		1	Passage au mouvement automatique
Séquence		1	Identificateur de séquence d'animation [0..31]. S'il n'y a pas de séquence, un zéro doit être attribué.
Toujours d'après	8	1	Numéro de trame dans la séquence [0..7].
Image			Adresse de la mémoire où se trouve l'image
Précédent Sprite	10		Utilisation interne pour le mécanisme de tri des sprites
Sprite suivant			Utilisation interne pour le mécanisme de tri des sprites
Itinéraire		1	Identifiant du chemin à suivre par le sprite

L'adresse mémoire où sont stockées les coordonnées de chaque sprite peut être calculée comme suit :

Adresse des coordonnées Y = $27000 + 16*N +1$

Adresse des coordonnées X = $27000 + 16*N +3$

En accédant avec **POKE** à ces adresses, nous pouvons modifier leur valeur, bien que vous puissiez également utiliser **|LOCATESP**.

La bibliothèque 8BP n'utilise pas de "pixels" pour la coordonnée X, mais des octets, de sorte que la coordonnée X qui tombe à l'intérieur de l'écran est comprise dans l'intervalle [0..79]. La coordonnée Y est représentée en lignes, de sorte que la plage représentable à l'écran est [0..200]. Si vous placez un sprite en dehors de ces plages, mais qu'une partie du sprite se trouve sur l'écran, la bibliothèque effectuera le découpage et peindra la partie qui doit être vue.

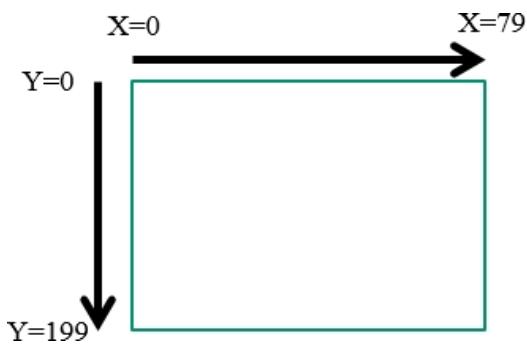


Fig. 46 Coordonnées de l'écran

Les adresses des attributs des 32 sprites peuvent être manipulées avec PEEK et POKE, bien que la séquence d'animation et l'assignation du chemin impliquent plus d'opérations et si vous voulez les changer, un POKE n'est pas suffisant, vous devez utiliser |SETUPSP. Voici la liste des adresses de tous les attributs des 32 sprites :

	1byte	2 bytes	2 bytes	1byte	1byte	1byte	1byte	2 bytes	1byte
sprite	status	coordy	coordx	vy	vx	seq	frame	imagen	ruta
0	27000	27001	27003	27005	27006	27007	27008	27009	27015
1	27016	27017	27019	27021	27022	27023	27024	27025	27031
2	27032	27033	27035	27037	27038	27039	27040	27041	27047
3	27048	27049	27051	27053	27054	27055	27056	27057	27063
4	27064	27065	27067	27069	27070	27071	27072	27073	27079
5	27080	27081	27083	27085	27086	27087	27088	27089	27095
6	27096	27097	27099	27101	27102	27103	27104	27105	27111
7	27112	27113	27115	27117	27118	27119	27120	27121	27127
8	27128	27129	27131	27133	27134	27135	27136	27137	27143
9	27144	27145	27147	27149	27150	27151	27152	27153	27159
10	27160	27161	27163	27165	27166	27167	27168	27169	27175
11	27176	27177	27179	27181	27182	27183	27184	27185	27191
12	27192	27193	27195	27197	27198	27199	27200	27201	27207
13	27208	27209	27211	27213	27214	27215	27216	27217	27223
14	27224	27225	27227	27229	27230	27231	27232	27233	27239
15	27240	27241	27243	27245	27246	27247	27248	27249	27255
16	27256	27257	27259	27261	27262	27263	27264	27265	27271
17	27272	27273	27275	27277	27278	27279	27280	27281	27287
18	27288	27289	27291	27293	27294	27295	27296	27297	27303
19	27304	27305	27307	27309	27310	27311	27312	27313	27319
20	27320	27321	27323	27325	27326	27327	27328	27329	27335
21	27336	27337	27339	27341	27342	27343	27344	27345	27351
22	27352	27353	27355	27357	27358	27359	27360	27361	27367
23	27368	27369	27371	27373	27374	27375	27376	27377	27383
24	27384	27385	27387	27389	27390	27391	27392	27393	27399
25	27400	27401	27403	27405	27406	27407	27408	27409	27415
26	27416	27417	27419	27421	27422	27423	27424	27425	27431
27	27432	27433	27435	27437	27438	27439	27440	27441	27447
28	27448	27449	27451	27453	27454	27455	27456	27457	27463
29	27464	27465	27467	27469	27470	27471	27472	27473	27479
30	27480	27481	27483	27485	27486	27487	27488	27489	27495
31	27496	27497	27499	27501	27502	27503	27504	27505	27511

Tableau 3 Adresses des attributs des 32 sprites

L'espace occupé par chaque sprite dans le tableau est de 16 octets. Comme vous pouvez le voir, les coordonnées X et Y sont des nombres de 2 octets. Les sprites acceptent des coordonnées négatives, ce qui permet d'imprimer partiellement un sprite à l'écran, en donnant l'impression qu'il arrive petit à petit. Vous ne pouvez pas définir de coordonnées négatives avec POKE, mais vous pouvez le faire avec |LOCATESP et aussi avec |POKE, qui est une version de la commande POKE de BASIC mais qui accepte les nombres négatifs (et les nombres de 16 bits).

Il est conseillé de placer le personnage ou le vaisseau spatial à la position 31 (il y a 32 sprites numérotés de 0 à 31). Si votre vaisseau spatial se trouve en position 31, il sera imprimé en dernier, au-dessus du reste des sprites en cas de chevauchement.

8.7 Tous les sprites sont imprimés et triés

Dans la bibliothèque 8BP, vous disposez d'une commande qui affiche tous les sprites dont le drapeau d'impression est actif au même moment. Il s'agit de la commande |PRINTSPALL

Cette commande a 4 paramètres, mais vous ne devez les remplir que la première fois que vous l'invoquez, car les fois suivantes, elle se souviendra des paramètres, et vous ne devrez les remplir à nouveau que si vous souhaitez modifier l'un d'entre eux. Ceci est utile car le passage de paramètres est très long (vous pouvez gagner plus de 1ms en évitant le passage de paramètres).

Les paramètres sont les suivants

| Les données de l'enquête sont disponibles sur le site Internet de la Commission européenne, à l'adresse suivante : <PRINTSPALL>, <ordenini>, <ordenfin>, <animé>, <synchronisme>.

- **Le paramètre de synchronisation** peut prendre les valeurs 0 ou 1 et indique qu'il attendra une interruption du balayage de l'écran avant d'imprimer. Si vous voulez de la vitesse, je ne le recommande pas. Si vous voulez plus de fluidité, peut-être oui.
- **Le paramètre d'animation** peut prendre les valeurs 0 ou 1. S'il est actif, avant d'imprimer chaque sprite, son drapeau d'animation dans l'octet de statut sera vérifié et s'il est actif, il passera à l'image suivante. C'est très utile avec les ennemis, mais pas avec votre personnage, car vous pouvez vouloir l'animer uniquement lorsque vous le déplacez et non à chaque image. IMPORTANT : l'animation est effectuée avant l'impression, et non après. Cela signifie que si vous venez d'attribuer une séquence d'animation, vous ne verrez pas la première image de cette séquence.
- **Les paramètres d'ordre ("ordenini", "ordenfin")** indiquent les sprites initiaux et finaux qui définissent le groupe de sprites ordonnés par la coordonnée "Y" que nous allons imprimer. Par exemple, si nous attribuons les valeurs 0,0, ils seront imprimés séquentiellement du sprite 0 au sprite 31. Si nous attribuons 0,8, ils seront imprimés de 0 à 8 dans l'ordre (9 sprites) et de 10 à 31 séquentiellement. Si nous attribuons 0,31, tous les sprites seront imprimés dans l'ordre. Si nous attribuons 10,20, les sprites seront imprimés séquentiellement de 0 à 9, puis ils seront imprimés dans l'ordre de 10 à 20 et enfin ils seront imprimés séquentiellement de 21 à 31.

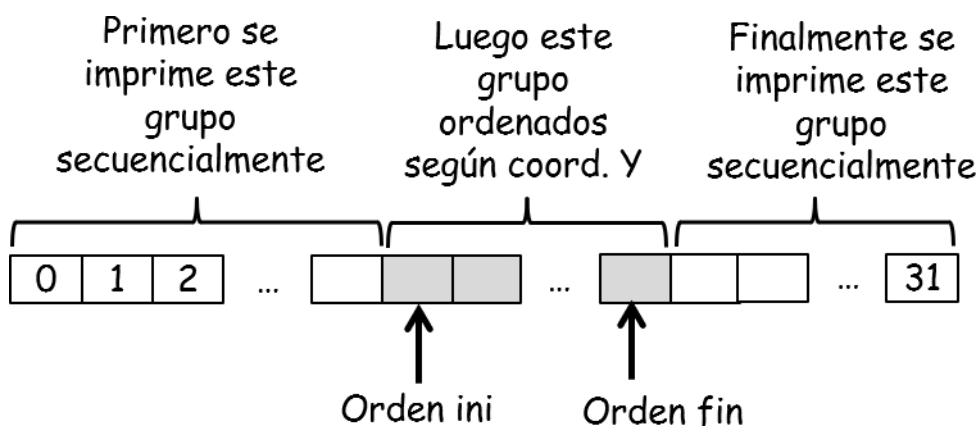


Fig. 47 Groupes de sprites séquentiels et ordonnés

L'ordre est très utile pour réaliser des jeux comme "Renegade" ou "Golden AXE", où il est nécessaire de donner un effet de profondeur. L'ordre est apprécié lorsqu'il y a des chevauchements entre les sprites.



Fig. 48 Effet de l'ordre des sprites

- | **PRINTSPALL, 0,0,1,0 : imprime séquentiellement tous les sprites**
- | **PRINTSPALL, 0,31,1,0 : imprime tous les sprites de manière ordonnée**
- | **PRINTSPALL, 0,7,1,0 : imprime 8 de manière ordonnée et le reste de manière séquentielle**
- | **PRINTSPALL, 16,24,1,0 : 16 séquentielles, 9 triées et 7 séquentielles**

Si le paramètre "ordenini" est omis, c'est la dernière valeur attribuée qui est prise en compte, ou zéro si aucune valeur n'a jamais été attribuée. Par ailleurs, si vous avez l'intention de modifier l'un des deux paramètres de tri, il est conseillé d'exécuter d'abord PRINTSPALL,0,0,0,0,0 pour que les sprites soient d'abord réordonnés séquentiellement avant d'être triés avec une nouvelle configuration.

L'impression dans l'ordre est plus coûteuse en termes de calcul que l'impression séquentielle. Si vous n'avez que 5 sprites à trier, passez par exemple 0.4 comme paramètre de tri, ne passez pas 0.31. Le tri de tous les sprites prend environ 2,5 ms, mais si vous ne triez que 5 sprites, vous pouvez gagner 2 ms. Il se peut que vous ayez beaucoup de sprites et qu'il ne soit pas utile de trier certains d'entre eux, comme les plans ou les sprites dont vous savez qu'ils ne se chevaucheront pas.

L'algorithme utilisé pour trier les sprites est une variante de l'algorithme dit "à bulles". Bien que vous trouviez dans la littérature que l'algorithme dit "à bulles" est très inefficace, cela est dit par ceux qui parlent d'une liste de nombres aléatoires à trier. Dans le cas présent, les sprites sont normalement presque ordonnés et, d'une image à l'autre, seuls un ou deux sprites sont désordonnés, pas plus, car leurs coordonnées évoluent "en douceur". L'algorithme parcourt donc la liste des sprites et lorsqu'il trouve quelques sprites désordonnés, il les retourne et arrête le tri. Il est extrêmement rapide, et même s'il n'est capable de trier que quelques sprites à la fois, il est idéal pour ce cas d'utilisation. Dans le seul cas où il y a 2 sprites en désordre et qu'ils se chevauchent, il y aura une image où l'on verra l'un d'eux s'imprimer dans le désordre, mais cela sera corrigé dans l'image suivante. C'est imperceptible.

Il peut arriver que vous souhaitiez que le tri soit complet à chaque image. En d'autres termes, il ne s'agit pas de trier quelques sprites à chaque invocation de |PRINTSPALL, mais de s'assurer qu'ils sont tous triés. La bibliothèque **8BP** vous permet de le faire grâce à ses quatre modes de tri, que vous pouvez définir en invoquant la commande |Ce qui suit est un PRINTSPALL à paramètre unique (il suffit de l'exécuter une fois pour

définir le mode de tri) :

PRINTSPALL,0 : tri partiel en utilisant Ymin **PRINTSPALL,1 : tri complet en utilisant Ymin** **PRINTSPALL,2 : tri partiel en utilisant Ymax** **PRINTSPALL,3 : tri complet en utilisant Ymax**

Le tri utilisant Ymax est basé sur la plus grande coordonnée Y des sprites, c'est-à-dire l'endroit où se trouvent leurs pieds plutôt que leur tête. Si les sprites sont de la même taille, un tri basé sur Ymin peut fonctionner, mais si les sprites sont de hauteurs différentes, vous pouvez vouloir trier en fonction de l'emplacement des pieds de chaque personnage, et pour cela vous devrez utiliser le mode de tri 2 ou 3.

Les modes de tri Ymax sont plus lents, environ 0,128 ms par sprite, et ne sont donc utilisés que lorsque vous en avez vraiment besoin.

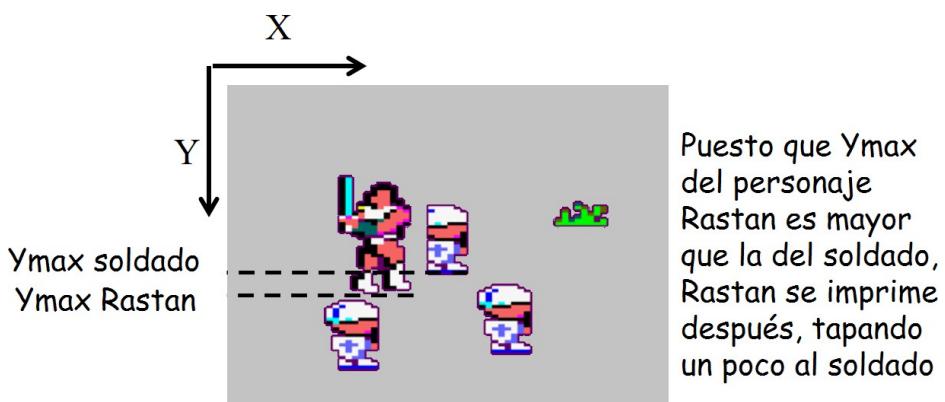


Fig. 49 Tri selon Ymax

Le tri complet consomme très peu plus que le tri partiel (environ 0,3 ms). Cela s'explique par le fait que les sprites ne sont pratiquement jamais mélangés d'une image à l'autre, mais même ces 0,3 ms valent la peine d'être économisées si possible.

Rappelons que la commande PRINTSPALL a une "mémoire", il suffit donc de l'invoquer la première fois avec des paramètres et à partir de ce moment on peut invoquer PRINTSPALL sans paramètres car la commande "garde" les valeurs des paramètres avec lesquels elle a été invoquée et il n'est pas nécessaire de les lui passer à moins qu'ils ne changent. Cela permet d'économiser plus de 1 ms, car l'analyseur syntaxique travaille moins.

8.8 Collisions entre sprites

Pour vérifier si votre personnage ou votre tir est entré en collision avec d'autres sprites, vous pouvez utiliser la commande

|COLSP, <numéro du texte>, @collision%.

Où le numéro de sprite est le sprite que vous voulez tester (votre personnage ou votre tir) et la variable "collision" est une variable entière qu'il fallait auparavant définir, en lui assignant une valeur initiale, par exemple :

collision%=0

|COLSP, 1, @collision%, 1, @collision%, 1, @collision%.

La variable "collision" sera remplie avec le premier identifiant de sprite détecté comme étant entré en collision avec votre sprite, bien que plusieurs collisions puissent se produire, mais la commande ne donne qu'un seul résultat.

En interne, la bibliothèque 8BP parcourt les sprites qui entrent en collision de 31 à 0 (dans l'ordre inverse), et si le drapeau de collision est actif (bit 1 de l'octet de statut), elle vérifie s'il entre en collision avec votre sprite. S'il n'y a pas de collision avec l'un d'entre eux, la variable collision% est mise à 32. Si c'est le cas, elle renvoie le numéro du sprite qui entre en collision avec votre sprite. Si, par exemple, 4 et 12 entrent en collision, la fonction renverra un 12 car elle vérifie 12 avant 4.

Ni votre personnage ni votre tir ne doivent avoir le drapeau "collider" (bit 1) et "collider" (bit 5) actifs en même temps, sinon ils entreront toujours en collision... avec eux-mêmes ! En d'autres termes, un sprite ne peut pas avoir les bits 1 et 5 actifs en même temps.

La collision entre les sprites est une tâche coûteuse. En interne, la bibliothèque doit calculer l'intersection entre les rectangles contenant chaque sprite pour déterminer s'il y a un chevauchement entre eux. Pour économiser des calculs, il est préférable de placer les ennemis dans des positions de sprites consécutives. Si, par exemple, les ennemis avec lesquels nous pouvons entrer en collision sont les sprites 15 à 25, nous pouvons configurer la collision pour qu'elle ne vérifie que ces sprites. Pour ce faire, nous invoquons la collision sur le sprite 32, qui n'existe pas. Cela indiquera à la bibliothèque 8BP qu'il s'agit d'une information de configuration pour la commande, indiquant la **plage de sprites collisionnables à analyser pour chaque collisionneur** :

|COLSP, 32, <sprite initial>, <sprite final>.

Exemple :

|COLSP, 32, 15, 25

Cette optimisation n'est pas très significative, mais elle le devient lorsque COLSP est invoqué plusieurs fois ou lorsque l'on utilise la commande |COLSPALL, qui invoque COLSP plusieurs fois en interne.

Une autre optimisation intéressante, capable d'économiser 1 milliseconde à chaque invocation, consiste à indiquer à la commande de toujours utiliser la même variable BASIC pour laisser le résultat de la collision. Pour ce faire, nous l'indiquerons en utilisant 33 comme sprite, qui n'existe pas non plus.

col%=0

|COLSP, 33, @col%, @col%, @COLSP, 33, @col%, @COLSP, 33, @col%, @COLSP, 33, @col%

Une fois ces deux lignes exécutées, les invocations suivantes de COLSP laisseront le résultat dans la variable col, sans qu'il soit nécessaire de l'indiquer, par exemple :

|COLSP, 23 : REM cette invocation est équivalente à |COLSP, 23, @col%.

IMPORTANT : La variable de collision dans la commande |COLSP n'est pas celle

utilisée dans la commande **|COLSPALL**. Il s'agit de variables différentes (à moins que vous ne transmettiez aux deux commandes la même variable pour agir sur elle).

8.9 Réglage de la sensibilité aux collisions des sprites

Il est possible d'ajuster la sensibilité de la commande COLSP, en décidant si le chevauchement entre les sprites doit être de plusieurs pixels ou d'un seul pixel pour qu'une collision soit considérée comme ayant eu lieu.

Cela peut être fait en définissant le nombre de pixels (pixels dans la direction Y, octets dans la direction X) de chevauchement nécessaire dans les directions Y et X, en utilisant la commande COLSP et en spécifiant le sprite 34 (qui n'existe pas).

|COLSP, 34, <dy>, <dx>, <dx>, <dx>, <dx>, <dx>, <dx>, <dx>.

La bibliothèque 8BP n'utilise pas de "pixels" dans la coordonnée X, mais des octets. Vous devez donc tenir compte du fait qu'une collision de 1 octet correspond en réalité à 2 pixels et qu'il s'agit de la collision minimale possible lorsque vous définissez dx=0.

Pour la coordonnée y, la bibliothèque travaille avec des lignes, de sorte que dy=0 signifie une collision d'un seul pixel.

Une collision stricte, utile pour le tir, serait une collision qui ne tolère aucune marge, considérant qu'il y a collision dès qu'il y a un chevauchement minimum entre les sprites (1 pixel dans la direction Y ou un octet dans la direction X).

|COLSP, 34, 0, 0 : rem collision dès qu'il y a un chevauchement minimum

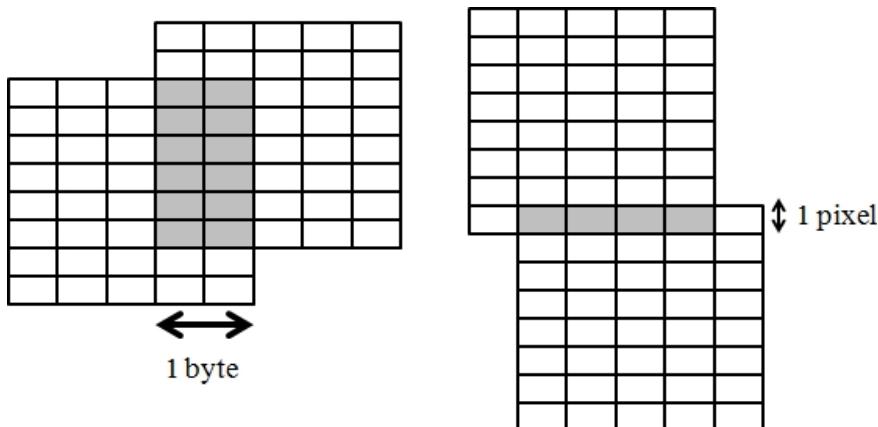


Fig. 50 Collision stricte avec COLSP, 34, 0, 0

Toutefois, si nous créons un jeu en MODE 0, où les pixels sont plus larges que hauts, il est peut-être plus approprié de donner une marge sur l'axe des Y, mais aucune sur l'axe des X.

X. Par exemple :

|COLSP, 34, 2, 0 : rem collision avec 3 pixels en Y et 1 octet en X

Ma recommandation est que, s'il y a des plans étroits ou petits, vous devriez définir la collision à (dy=1, dx=0) alors que s'il n'y a que des personnages larges, vous pouvez laisser plus de marge (dy=2, dx=1). Vous devez également prendre en compte le fait que si vos sprites ont une "marge" d'effacement autour d'eux pour se déplacer en s'effaçant eux-mêmes, cette marge ne doit pas être prise en compte dans la collision, il est donc logique que dy et dx soient tous deux différents de zéro. Dans tous les cas, c'est quelque

chose que vous déciderez en fonction du type de jeu que vous créez.

8.10 Qui entre en collision et avec qui : COLSPALL

Avec la fonction **|COLSP que** nous avons vue jusqu'à présent, la détection de collision d'un sprite avec tous les autres est possible. Cependant, si nous avons un tir multiple, où par exemple notre vaisseau peut tirer jusqu'à 3 coups simultanément, nous devrions détecter la collision de chacun d'entre eux et en plus celle de notre vaisseau, ce qui résulterait en 4 invocations de **|COLSP**.

Gardez à l'esprit que chaque invocation passe par la couche d'analyse, de sorte que quatre invocations sont coûteuses. Pour cela, nous disposons d'une commande très puissante : **|COLSPALL**.

Cette fonction fonctionne en deux étapes : nous devons d'abord spécifier quelles variables vont stocker le collisionneur et le sprite percuté. L'instruction suivante ne sera exécutée qu'une seule fois et sert à définir les variables sur lesquelles nous obtiendrons les résultats, qui doivent exister au préalable :

|COLSPALL, @collider%, @collider%, @collider%.

Par la suite, à chaque cycle de jeu, nous invoquons simplement la fonction **|COLSPALL** sans paramètres.

La fonction considérera comme sprites "en collision" ceux dont le drapeau de collision est à "1" dans l'octet de statut (bit 5), et comme sprites "en collision" ceux dont le drapeau de collision (bit 1) de l'octet de statut est à "1". Les sprites en collision devraient être notre vaisseau et nos tirs, et les sprites en collision devraient être tous ceux avec lesquels nous pouvons entrer en collision : vaisseaux et tirs ennemis, montagnes, etc. Comme je l'ai déjà dit, un sprite ne doit pas avoir les deux bits actifs (=1) en même temps.

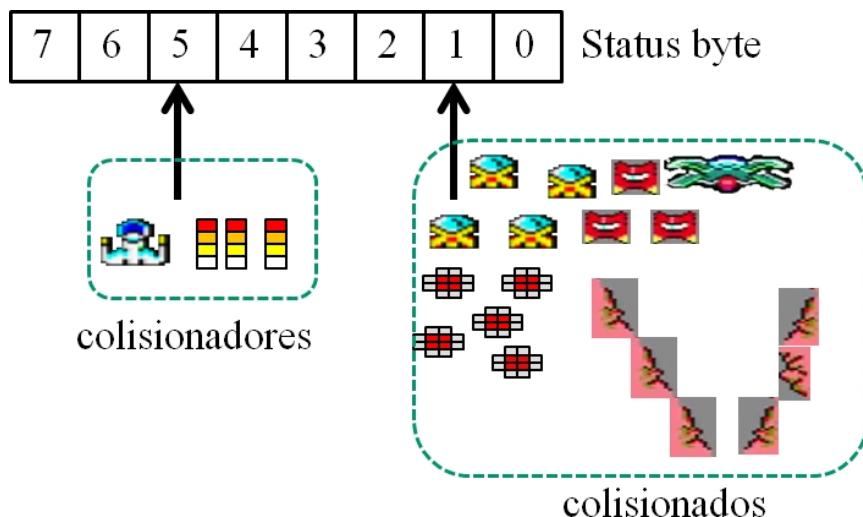


Fig. 51 Colliders versus colliders

La fonction **|COLSPALL** commence par vérifier le sprite 31 (s'il s'agit d'un collider) et descend jusqu'au sprite 0, en invoquant en interne **|COLSP** pour chaque sprite de collider. Pour chaque collisionneur, les sprites de collision sont également parcourus dans l'ordre décroissant (de 31 à 0). Dès qu'il détecte une collision, il interrompt son exécution et renvoie la valeur du collider et du collider. Il est donc important que notre vaisseau ait un sprite plus élevé que nos tirs. Ainsi, si nous sommes touchés

nous le détecterons, même si nous avons touché un ennemi par un tir au même moment.

Dans chaque cycle de jeu, une seule collision peut être détectée, mais c'est suffisant. Le fait qu'un seul ennemi puisse commencer à "exploser" dans chaque image ne constitue pas une limitation majeure. Si, par exemple, vous lancez une grenade et qu'un groupe de 5 soldats est touché, chaque soldat commencera à mourir dans une image différente et, au bout de 5 images, ils auront tous explosé. L'utilisation de |COLSPALL ne les fera pas tous exploser en même temps, mais votre jeu sera plus rapide, ce qui est très important dans un jeu d'arcade.

Si l'on invoque |COLSPALL avec un seul paramètre,

|COLSPALL, <Collisionneur initial>.

Les collisionneurs seront analysés à partir du collisionneur indiqué -1 jusqu'au sprite zéro, dans l'ordre décroissant. Ainsi, si vous devez détecter plus d'une collision par cycle de jeu, vous pouvez le faire en invoquant successivement **COLSPALL, <collider>** jusqu'à ce que la variable collider prenne la valeur 32.

Exemple :

|COLSPALL, 7 : recherche de collisions à partir du collisionneur 6

8.10.1 Comment programmer des tirs multiples avec COLSPALL

La première chose à faire est de déterminer le nombre de tirs actifs qui peuvent être effectués. Si vous décidez que votre navire peut tirer 3 projectiles à la fois, vous devez alors

réservez 3 identifiants de sprites à déclencher. Ensuite, vous devez ajuster le délai entre un tir et le suivant pour éviter que deux projectiles ne sortent presque côté à côté si vous tirez trop vite. Cela peut se faire en définissant un délai minimum entre les tirs.

Dans l'exemple suivant, j'ai fixé un délai entre les tirs de 10 cycles de jeu. Pour ce faire, en appuyant sur la barre d'espacement (touche 47), on vérifie si au moins 10 cycles se sont écoulés depuis le dernier tir. Si ce n'est pas le cas, le tir n'a pas lieu.

```
130 ----- "cycle de jeu".
150 |AUTOALL,1:|PRINTSPALL,0,1,0
170 ' character movement routine -----
    IF INKEY(47)=0 ALORS IF delay<cycle-10 ALORS delay=cycle:disp= 1+
    disp MOD 3 :|LOCATESP,10+disp,PEEK(27001)+8,PEEK(27003) :
    |SETUPSP,10+disp,0,137: |SETUPSP,10+disp,15,3+dir
        SI INKEY(27)=0 ALORS dir=0:|SETUPSP,0,6,1:'aller à droite
    190 SI INKEY(34)=0 ALORS dir=1:|SETUPSP,0,6,-1:'aller à gauche
    193 cycle=cycle+1
    310 GOTO 150
```

Pour choisir le sprite à utiliser comme déclencheur, l'instruction est exécutée :

disp = 1+ disp Mod 3

Cela permettra à votre tir de prendre les valeurs 1,2,3,4 alternativement. Si vous avez besoin que les sprites prennent les valeurs 20,21,22,23, vous pouvez utiliser **disp = 20 + disp Mod 3 .**

car si vous mettez 21 comme somme, cela ne fonctionne pas (essayez vous-même). C'est le problème de l'arithmétique modulaire.

Nous en arrivons maintenant aux collisions et à l'avantage d'utiliser COLSPALL, beaucoup plus rapide que d'invoquer COLSP plusieurs fois. Les seules recommandations importantes sont les suivantes :

- Que notre <nombre de sprites> soit plus élevé que nos déclencheurs, de manière à ce que |COLSPALL| le vérifie avant de tirer.
- Que nous avons configuré |COLSP| pour ne vérifier que la liste des sprites qui sont des ennemis et qui doivent entrer en collision, en utilisant |COLSP 32, <début>, <fin>|.

Avant de commencer le cycle de jeu, nous définissons nos variables :

collider=32:collided=32:|COLSPALL,@collider,@collided

Dans le cycle de jeu, nous mettrons :

**|COLSPALL:IF collider<32 THEN if collider=31 THEN GOSUB 300:goto 2000 :
ELSE GOSUB 770**

Avec cette ligne, nous savons déjà s'il y a une collision, parce que la variable "collider" sera <32>. De plus, si elle est égale à 31, alors c'est notre vaisseau (nous avons été touchés) et si ce n'est pas le cas, alors il est certain que l'un de nos tirs a touché un vaisseau ennemi et nous irons dans la routine située à la ligne 770, où nous trouverons quelque chose comme ceci :

**769' --- routine collision shot -----
770 |SETUPSP, collider, 9, img deleted : "associer l'image supprimée à l'image tir
772 |PRINTSP, collisionneur : "nous supprimons la prise de vue
775 |SETUPSP, collider, 0, 0 : 'désactiver le déclenchement
777 if collided>=duros then return:'enemy indestructible
778 ' La séquence 4 est une séquence d'animation de la "mort", un explosion
780 |SETUPSP, collision, 7, 4:|SETUPSP, collision, 0, &x101 : retour**

En bref, une seule invocation de COLSPALL permet de savoir qui est entré en collision ("collider") et avec qui il est entré en collision ("collided").

8.10.2 Qui s'effondre lorsqu'il y a plusieurs chevauchements

Il est très important de garder à l'esprit que 8BP parcourt les colliders de 31 à 0 et pour chacun d'entre eux, il parcourt les colliderables de 31 à 0. Nous devons associer les Sprite IDs à nos sprites en fonction de la façon dont nous voulons qu'ils entrent en collision.

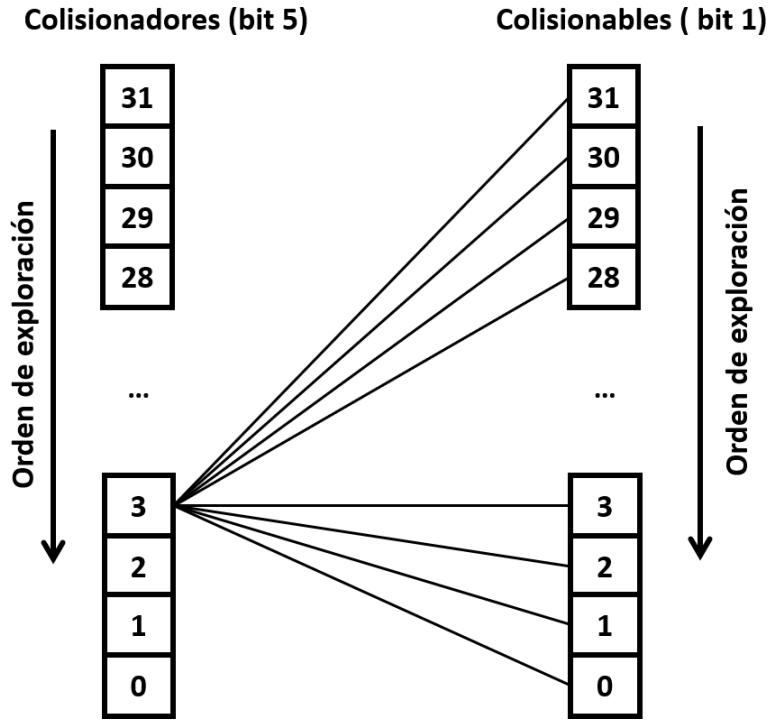


Fig. 52 Ordre de contrôle des collisions

Si notre sprite (collider) entre en collision avec deux sprites dans la même zone, nous pouvons savoir a priori avec lequel nous allons entrer en collision, ce qui est très utile pour certains jeux.

Prenons l'exemple du jeu "frogger", dans lequel une grenouille doit traverser une rivière en sautant par-dessus des troncs d'arbre. Si la collision a lieu sur un rondin, nous ne mourrons pas, mais si la collision a lieu sur une rivière, nous mourrons.

Pour le programmer, on peut mettre 4 rivières (4 sprites allongés immobiles) et sur celles-ci se déplacent des sprites qui sont des troncs. Nous pourrions mettre en place la distribution suivante :

- La grenouille est le sprite 31 (on suppose qu'il s'agit d'un collisionneur).
- Les troncs sont les sprites 4, 5, 6, 7 (collisions).
- Les rivières sont les sprites 0, 1, 2, 3 (collisibles).

Les rivières peuvent avoir le drapeau d'impression désactivé, de sorte qu'elles peuvent entrer en collision avec la grenouille (drapeau d'entrée en collision) sans être imprimées. En d'autres termes, nous leur attribuons le statut =2



Fig. 53 En cas de chevauchement, nous nous intéressons à la collision du tronc.

Comme les rondins et la rivière se chevauchent, au moment où la grenouille grimpe sur un rondin, elle entre en collision avec les deux, mais le rondin est vérifié en premier car il a un sprite d'ID plus élevé. La commande de collision ne détectera que la collision avec le rondin. Au contraire, si la grenouille saute au-dessus de l'eau, la commande collision détectera la rivière et après avoir évalué en **BASIC** la variable "collided" et vu qu'il s'agit d'une rivière, nous déterminerons que notre grenouille doit mourir.

8.10.3 Utilisation avancée de l'octet de statut dans les collisions

Il peut arriver que vous souhaitiez qu'un ennemi ne vous tue pas en entrant en collision avec votre personnage parce qu'il se trouve dans un état spécial ou parce qu'il est simplement loin dans un jeu qui prétend que les ennemis approchent.

Certaines circonstances particulières peuvent nécessiter l'utilisation d'une "marque" sur le sprite pour indiquer que, bien qu'il y ait eu une collision, il ne doit pas mourir, ou ne doit pas vous tuer.

Pour cela, vous pouvez utiliser les drapeaux que vous n'utilisez pas dans le sprite et les vérifier dans votre routine de collision.

Prenons l'exemple d'un ennemi que nous voulons rendre inoffensif parce qu'il est loin (simulant la 3D) ou qu'il a une faible énergie, mais qui entre en collision avec nous.

Supposons que votre personnage soit collé et que l'ennemi soit un collider. Vous pouvez forcer que la collision ne soit détectée qu'avec des sprites plus grands que 25 et si l'ennemi est le numéro 20 (par exemple) il ne pourra pas entrer en collision avec lui-même.

|COLSP,32,25,31

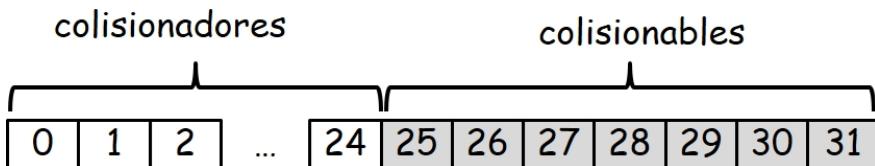


Fig. 54 effet de COLSP,32,25,31

Comme indicateur "inoffensif", nous utiliserons le drapeau de collision, en le mettant à 1. Nous mettrons donc le drapeau de l'ennemi (sprite 20) à 1 dans son octet d'état.

Supposons maintenant que la commande |COLSPALL détecte une collision et laisse le résultat dans collider et collided.

|COLSPALL

Si collider<32 alors GOSUB 100
<instructions>

```
100 routine de collision rem
110 dir=27000 + collider*16 :rem byte address status of collider
120 if PEEK (dir) and 2 THEN RETURN : rem harmless if bit collided=1
130 <un ennemi est entré en collision et n'est pas inoffensif>.
```

L'instruction "**if PEEK (dir) and 2**" est celle qui vérifie le bit en collision puisque 2 en binaire est 00000010, juste la position de ce bit dans l'état du sprite.

Lorsque l'ennemi n'est plus inoffensif, il suffit de mettre son drapeau de collision à 0 et, en cas de nouvelle collision, il nous tuera.

Cette technique est parfaitement valable avec n'importe quel autre drapeau non utilisé.

8.11 Tableau des séquences d'animation

Les animations sont généralement composées d'un nombre pair d'images, même si ce n'est pas une règle stricte. Pensez par exemple à une simple animation de personnage avec seulement deux images : les jambes ouvertes et fermées. Cela fait deux images. Pensez maintenant à une animation améliorée, avec une phase de mouvement intermédiaire. Cela signifie qu'il faut créer la séquence suivante : fermé-intermédiaire-ouvert-ouvert-intermédiaire-et recommencer. Comme vous pouvez le constater, il s'agit d'un nombre pair, c'est-à-dire 4.

Les séquences d'animation 8BP sont des listes de 8 images, elles ne peuvent pas en avoir plus, bien qu'il soit toujours possible de créer des séquences plus courtes.

Les images d'une séquence d'animation sont les adresses de mémoire où sont assemblées les images qui les composent, et elles peuvent être de taille différente, bien qu'elles soient normalement identiques. Si, au milieu de la séquence, vous entrez un zéro, cela signifie que la séquence est terminée.

Bien qu'avant V33 il existait une commande RSX appelée **|SETUPSQ** pour créer des séquences à partir de BASIC, je l'ai supprimée dans V33, parce que son utilisation est plus complexe que la définition de séquences à partir du fichier sequences.asm et, en fait, je n'ai jamais utilisé la commande **|SETUPSQ** dans aucun de mes jeux, j'ai donc décidé de la sacrifier pour économiser de la mémoire.

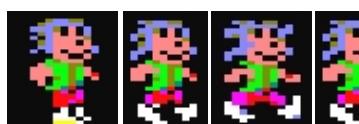
Voyons un exemple de création d'une séquence dans le fichier sequences.asm.

```
dw MONTOYA_R0,MONTOYA_R1,MONTOYA_R2,MONTOYA_R1,0,0,0,0,0,0,0,0
```

Avant d'expliquer comment assigner une séquence à un sprite, permettez-moi de vous rappeler comment assigner des images à des sprites : Depuis la version V26 du 8BP, il est possible d'inclure une liste d'images (leurs étiquettes) dans une liste appelée IMAGE_LIST dans votre fichier images_your_game.asm. Vous pouvez ainsi référencer les images à partir du BASIC avec un index au lieu d'une adresse mémoire. De cette façon, vous n'avez pas à rechercher les adresses mémoire à chaque fois que vous assemblez. Ceci s'applique à l'instruction :

|SETUPSP, #, 9, <adresse>.

L'exemple montre une séquence d'animation de 3 images différentes, mais pour la rendre fluide avant de recommencer, il faut repasser par l'image "du milieu" (notez que la deuxième et la quatrième image sont identiques), ce qui fait qu'au final, il y a 4 images :



et retour à la case
départ

Fig. 55 Séquence d'animation

Si vous souhaitez créer une séquence de plus de 8 images, vous pouvez simplement enchaîner deux séquences à la suite et, lorsque le personnage atteint la dernière image de la première séquence, utiliser la commande **|SETUPSP** pour lui attribuer la deuxième séquence.

Les séquences d'animation sont assemblées à partir de l'adresse 33600 et vous pouvez définir jusqu'à 31 séquences d'animation (à partir du numéro 32, elles ne sont pas considérées comme des séquences, mais comme des "macro-séquences", ce qui est un autre concept). Chaque séquence est identifiée par un numéro compris dans l'intervalle

[1..31]. La séquence zéro n'existe pas, elle **est utilisée pour indiquer qu'un sprite n'a pas de séquence**.

Pour attribuer une séquence à un Sprite, utilisez la commande SETUPSP avec le paramètre 7 :

|SETUPSP, <id_sprite>, 7, <numéro de séquence>

Avec cette commande, la séquence d'animation est assignée au sprite dans le champ correspondant de la table des sprites, et un zéro est placé dans le champ d'identification de l'image. En outre, l'image correspondante est attribuée à la première image de la séquence. Si vous utilisez **|ANIMALL** avant l'impression ou **|PRINTSPALL** avec le drapeau d'animation, même si SETUPSP place l'animation à l'image zéro, vous sauterez à l'image 1 avant l'impression. Ce n'est normalement pas un problème, mais dans le cas d'une "séquence de mort" (nous y reviendrons plus tard) où, par exemple, la première image consiste à effacer le sprite, vous ne voudrez peut-être pas sauter directement à l'image 1.

1. Dans ce cas, une astuce simple peut consister à répéter l'image zéro dans la définition de la séquence de mort. De cette manière, vous vous assurez que l'image est visible. Une autre option consiste à supprimer le drapeau d'animation et à l'animer avec **ANIMASP** après l'impression.

Chaque séquence stocke 8 adresses mémoire correspondant aux 8 trames, soit 16 octets consommés par chaque séquence.

Votre fichier de séquence d'animation peut ressembler à ceci :

jusqu'à 31 séquences d'animation
doit être un tableau fixe et non un tableau variable
chaque séquence contient les adresses des images d'animation cycliques
Chaque séquence correspond à 8 adresses de mémoire image.
nombre pair car les animations sont généralement un nombre pair
un zéro signifie la fin de la séquence, bien que 8 mots soient toujours
dépensés.
; par séquence
Lorsqu'un zéro est trouvé, un nouveau départ est effectué.
s'il n'y a pas de zéro, il recommence après la huitième image.
Si la séquence zéro est attribuée à un Sprite, il n'a pas de séquence.
Nous commençons par la séquence 1
----- séquences d'animation de personnages montoya -----
LISTE DES SÉQUENCES
dw MONTOYA_R0,MONTOYA_R1,MONTOYA_R2,MONTOYA_R1,0,0,0,0,0,0,0,1 ;2
dw MONTOYA_U0,MONTOYA_UR1,MONTOYA_UR2,MONTOYA_UR1,0,0,0,0,0,0
dw MONTOYA_U0,MONTOYA_U1,MONTOYA_U0,MONTOYA_U2,0,0,0,0,0,0 ;3
dw MONTOYA_UL0,MONTOYA_UL1,MONTOYA_UL2,MONTOYA_UL1,0,0,0,0,0,0 ;4
dw MONTOYA_L0,MONTOYA_L1,MONTOYA_L2,MONTOYA_L1,0,0,0,0,0,0 ;5
dw MONTOYA_DL0,MONTOYA_DL1,MONTOYA_DL2,MONTOYA_DL1,0,0,0,0,0,0 ;6
dw MONTOYA_D0,MONTOYA_D1,MONTOYA_D0,MONTOYA_D2,0,0,0,0,0,0 ;7
dw MONTOYA_DR0,MONTOYA_DR1,MONTOYA_DR2,MONTOYA_DR1,0,0,0,0,0,0 ;8
----- séquences d'animation des soldats ----- dw
SOLDIER_R0,SOLDIER_R2,SOLDIER_R1,SOLDIER_R2,0,0,0,0,0,0 ;9
dw SOLDIER_L0 SOLDIER_L2 SOLDIER_L1 SOLDIER_L2 0 0 0 0 0 0 ;10

8.12 Séquences d'animation spéciales

Différents types de séquences d'animation sont disponibles dans la 8BP :

Type de séquence	description
Séquence normale	La séquence d'images d'animation est répétée à l'infini. Elles se terminent soit dans le sens de l'image, soit dans le sens de l'image, soit dans le sens de l'image.
	un zéro si l'on veut faire une séquence de moins de 8 images
Séquence de décès	La dernière image de la séquence est un "1". Ce chiffre indique au 8BP que l'état du sprite doit être mis à zéro. Normalement, l'image précédant le "1" est une image d'effacement.
Fin de la séquence	Après avoir exécuté la séquence, le sprite n'a plus d'animation. La dernière image de la séquence est un "2". Cela indique à 8BP de supprimer le drapeau d'animation de l'état Sprite.
Séquence enchaînée	Après avoir parcouru la séquence, la dernière trame indique l'identifiant de la séquence suivante qui sera attribuée à la Sprite. En utilisant ce type de séquences, vous pouvez construire des séquences d'animation de plus de 8 images.
macro-séquences	Ils permettent d'associer une séquence en fonction de la vitesse du Sprite, automatiquement.

8.12.1 Séquences de décès

La bibliothèque 8BP vous permet de réaliser des "séquences de mort", c'est-à-dire des séquences dans lesquelles le sprite passe à un état inactif à la fin de la séquence. Ceci est indiqué par un simple "1" comme valeur de l'adresse mémoire de la dernière image. Ces séquences sont très utiles pour définir des explosions d'ennemis qui sont animées avec |ANIMA ou

|ANIMALL. Après les avoir touchés avec votre tir, vous pouvez leur associer une séquence d'animation de mort et dans les cycles de jeu suivants, ils passeront par les différentes phases d'animation de l'explosion, et lorsqu'ils atteindront la dernière, ils passeront à l'état inactif, n'étant plus imprimés. Cet état inactif se fait automatiquement, donc ce que vous avez à faire est simplement de vérifier la collision de votre tir avec les ennemis et s'il entre en collision avec l'un d'entre eux, vous changez l'état avec SETUPSP pour qu'il ne puisse plus entrer en collision et vous lui assignez la séquence d'animation de la mort, également avec SETUPSP.

Si vous utilisez une séquence de mort, n'oubliez pas de vous assurer que la dernière image avant de trouver le "1" est complètement vide, afin qu'il n'y ait aucune trace de l'explosion.

Exemple d'une séquence de décès (notez qu'elle comprend un "1") :

dw EXPLOSION_1,EXPLOSION_2,Explosion_3,1,0,0,0,0,0,0

Un effet intéressant consiste à parcourir plusieurs images de façon répétée avant de terminer par un cadre noir qui sert à effacer l'image.

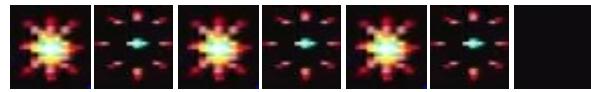


Fig. 56 Séquence de décès

Le "1" apparaît maintenant en huitième position :

```
dw EXPLOSION_1,EXPLOSION_2, EXPLOSION_1,EXPLOSION_2, EXPLOSION_1,EXPLOSION_2,  
ExPLOSION_3,1
```

N'oubliez pas que si vous utilisez la commande |PRINTSPALL alors que le drapeau d'animation est actif, l'image est d'abord animée puis imprimée, de sorte que la première image de la séquence de mort ne sera pas visible. Une astuce simple pour la faire apparaître consiste à la répéter deux fois.

8.12.2 Séquences de fin

Les séquences de fin existent depuis la version V42. Elles permettent d'associer à un Sprite une séquence d'animation que l'on souhaite ne voir exécutée qu'une seule fois. A la fin de la séquence, le Sprite n'aura plus de drapeau d'animation. Les autres drapeaux de son état sont respectés.

Il suffit de mettre un "2" dans la dernière image de la séquence.

Voici deux exemples. Dès que la séquence atteint l'image "2", le sprite n'est plus animé.

SEQUENCES_LIST	
dw	;1
MONTOYA_R0,MONTOYA_R1,MONTOYA_R2,MONTOYA_R1,2,0,0,0,0	;2
dw IMG1, IMG2, IMG3, IMG4, IMG5, IMG6, IMG7,2	

8.12.3 Séquences enchaînées

Les séquences enchaînées existent depuis la version 8BP V42 . Elles permettent de construire des séquences de plus de 8 images en enchaînant des séquences entre elles.

Le mécanisme est simple. Il suffit que la dernière image de la séquence soit le numéro de la séquence que vous voulez attribuer ensuite.

Comme vous pouvez l'imaginer, vous ne pourrez pas attribuer la séquence "1" ou "2", puisque ces numéros signifient "mourir" ou "fin". En d'autres termes, vous pourrez enchaîner des séquences à partir du numéro 3.

Dans cet exemple, j'ai enchaîné les séquences 4 et 5 de sorte qu'après l'une, l'autre est assignée et vice versa. Cela revient à avoir une séquence de 14 images. Nous pourrions enchaîner davantage de séquences et rendre l'animation beaucoup plus longue. Chaque séquence ajoutée représente 7 images supplémentaires (et non 8), car nous avons besoin de la dernière pour indiquer la séquence suivante.

Vous pouvez également les raccourcir et les remplir de zéros jusqu'à 8 images, mais le numéro de séquence doit apparaître avant tout zéro, car lorsqu'un zéro est rencontré, l'animation s'arrête.

LISTE DES SÉQUENCES	
dw MONTOYA_R0,MONTOYA_R1,MONTOYA_R2,MONTOYA_R1,0,0,0,0,0,0, ;1	
dw MONTOYA_U0,MONTOYA_UR1,MONTOYA_UR2,MONTOYA_UR1,0,0,0,0,0,0	;2
dw MONTOYA_U0,MONTOYA_U1,MONTOYA_U0,MONTOYA_U2,0,0,0,0,0	;3
dw IMG11, IMG2, IMG3, IMG4, IMG5, IMG6, IMG7, 5	;4
dw IMG18, IMG9, IMG10, IMG11, IMG12, IMG13, IMG14, 4	;5

8.12.4 Macroséquences d'animation

Il s'agit d'une fonctionnalité "avancée" disponible à partir de la version V25 de la bibliothèque 8BP. Une "macro-séquence" est une séquence composée de séquences. Chacune des

Les séquences d'animation constitutives sont les animations à effectuer dans une direction particulière. La direction est déterminée par les attributs de vitesse du sprite, qui se trouvent dans la table des sprites. Ainsi, lorsque nous animons un sprite avec |ANIMALL, il changera automatiquement sa séquence d'animation sans que nous ayons à faire quoi que ce soit (vous n'avez pas besoin d'invoquer |ANIMALL car |PRINTSPALL le fait déjà en interne si vous définissez un paramètre).

Les séquences de macros sont numérotées à partir de 32. Il est très important de placer les séquences à l'intérieur de la séquence macro dans l'ordre correct, c'est-à-dire que la première séquence doit être utilisée lorsque le personnage est immobile, la suivante lorsque le personnage va à gauche ($Vx < 0$, $Vy = 0$), la suivante à droite ($Vx > 0$, $Vy = 0$), etc, dans l'ordre suivant (attention, il est facile de se tromper) :



Fig. 57 Ordre des séquences dans une macro-séquence

Si la séquence attribuée à la position immobile est nulle, elle est simplement animée avec la dernière séquence attribuée.

Les macros séquences doivent être spécifiées dans le fichier sequences_yourgame.asm, dont un exemple est donné ci-dessous :

```
;-----  
; séquences d'animation  
;-----  
LISTE DES SÉQUENCES  
dw NAVE,0,0,0,0,0,0,0,0;1  
dw JOE1,JOE2,0,0,0,0,0,0,0;2 UP  
JOE dw JOE7,JOE8,0,0,0,0,0,0,0;3  
DW JOE dw  
JOE3,JOE4,0,0,0,0,0,0,0,0;4 R  
JOE dw  
JOE5,JOE6,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;5 L JOE  
  
SÉQUENCES MACRO  
;-----SÉQUENCES MACRO -----  
sont des groupes de séquences, un pour chaque direction. la signification est la suivante :  
; encore, gauche, droite, haut, haut-gauche, haut-droite, bas, bas-gauche, bas-droite  
Les numéros sont numérotés à partir de 32  
db 0,5,4,2,5,4,3,5,4;la séquence 32 contient les séquences du soldat Joe
```

Avec cette définition de séquence, nous pouvons créer un jeu simple qui nous permet de déplacer "Joe" sur l'écran sans contrôler sa séquence d'animation. Nous assignons la séquence

32 et de modifier la vitesse, la commande |ANIMA (invoquée à partir de l'intérieur de l'application

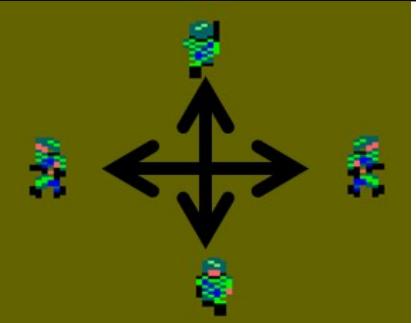
|PRINTSPALL) est chargé de modifier la séquence d'animation si sa vitesse indique un changement de direction. Pour déplacer le sprite, nous devons invoquer |AUTOALL, car le fait d'appuyer sur les commandes ne modifie pas ses coordonnées mais sa vitesse et |AUTOALL mettra à jour les coordonnées du sprite en fonction de sa vitesse.

10 MÉMOIRE 24999 20 MODE 0:INK 0,12 30 ON BREAK GOSUB 280 40 APPEL &6B78 50 DEFINT a-z 111 x=36:y=100 120 SETUPSP,31,0,0,&X1111 130 SETUPSP,31,7,2: SETUPSP,31,7,32	
--	--

```

140 |LOCATESP,31,y,x
160 |SETLIMITS,0,80,0,200
161 |PRINTSPALL,0,1,0
190 "commence le cycle de jeu
199 vy=0:vx=0
200 SI INKEY(27)=0 ALORS vx=1 : GOTO 220
210 SI INKEY(34)=0 ALORS vx=-1
220 SI INKEY(69)=0 ALORS vy=2 : GOTO 240
230 SI INKEY(67)=0 ALORS vy=-2
240 |SETUPSP,31,5,vy,vx
250 |AUTOALL:|PRINTSPALL
270 GOTO 199
280 |MUSIQUE:MODE 1 : ENCRE 0,0:STYLO 1

```



Notez que je n'ai pas défini la séquence lorsque le personnage ne bouge pas. Dans cette position, j'ai mis un zéro dans la macro-séquence. Cela signifie que, si le personnage démarre immobile, vous ne savez pas quelle séquence assigner puisqu'il n'y a pas de "dernière" séquence utilisée. C'est pourquoi j'assigne la séquence 2 avant d'assigner la 32, afin de m'assurer que le personnage a déjà une séquence, même s'il est immobile.

130 SETUPSP, 31, 7, 7, 2: SETUPSP, 31, 7, 32
--

9 Votre premier jeu simple

Vous avez maintenant les connaissances nécessaires pour faire un premier pas dans la création de jeux vidéo. Pour cela, prenons un exemple simple d'un soldat que vous allez contrôler, en le faisant marcher de gauche à droite sur l'écran.

Supposons que nous ayons édité un soldat, grâce à SPEDIT. Et nous avons construit ses séquences d'animation, qui ont été définies dans le fichier "**sequences_mygame.asm**" et ont été laissées avec les identifiants 9 et 10 pour les directions de mouvement gauche et droite respectivement.

Les deux séquences d'animation ont été créées à partir du fichier sequences.asm.

```
10 MÉMOIRE 24499
20 MODE 0 : DEFINT A-Z : CALL &6B78:' install RSX
25 call &bc02 : "restore default palette just in case".
26 encre 0,0 : "fond noir"
30 FOR j=0 TO 31:|SETUPSP,j,0,&X0:NEXT:'réinitialiser les sprites
40 |SETLIMITS,12,80,0,186 : ' définit les limites de l'écran de jeu
50 x=40:y=100:' coordonnées du personnage
51 |SETUPSP,0,0,1:' statut du personnage
52|SETUPSP,0,7,9:'séquence d'animation assignée au démarrage
53|LOCATESP,0,y,x:'placer le sprite (sans l'imprimer pour
l'instant)

60 "cycle de jeu
70 gosub 100
80 |PRINTSPALL,0,0
90 goto 60

99 ' routine de déplacement des caractères -----
100 SI INKEY(27)=0 ALORS SI dir<>>0 ALORS |SETUPSP,0,7,9:dir=0:return ELSE
|ANIMA,0:x=x+1:GOTO 120
110 SI INKEY(34)=0 ALORS SI dir<>>1 ALORS |SETUPSP,0,7,10:dir=1:return ELSE
|ANIMA,0:x=x-1
120 |LOCATESP,0,y,x
130 RETOUR
```

Avec cette liste, vous avez déjà un mini-jeu qui vous permet de contrôler un soldat et de le faire courir horizontalement. Notez que si, en marchant vers la gauche, vous dépassez la valeur minimale de la limite fixée avec |SETLIMITS, le personnage sera "coupé", ne montrant que la partie qui se trouve à l'intérieur de la zone de jeu autorisée.

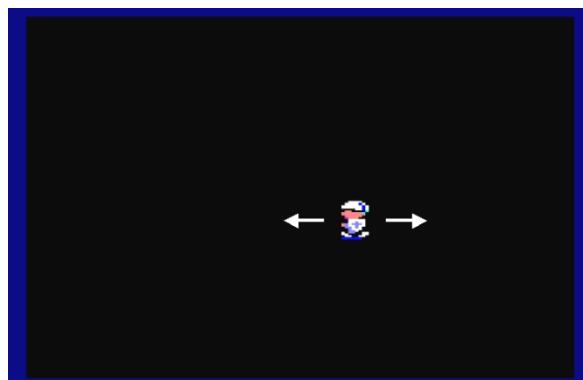


Fig. 58 Un jeu simple

9.1 Maintenant, sautons ! Boing, boing !

Dans l'exemple ci-dessus, notre personnage se déplace uniquement de gauche à droite.

Si nous voulons programmer un saut, nous pouvons le faire en stockant la trajectoire

verticale dans un fichier

BASIC. Nous verrons plus tard une meilleure façon de procéder (avec des chemins 8BP), mais pour l'instant cela servira d'exemple.

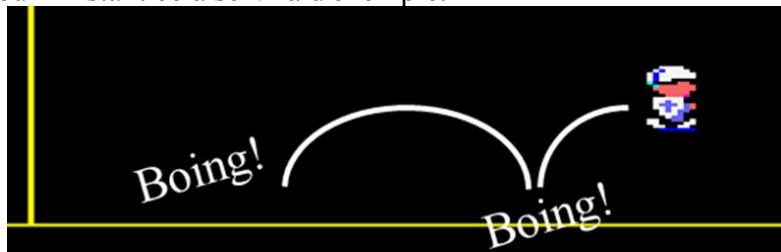


Fig. 59 notre personnage peut sauter

La trajectoire de saut est définie pour la coordonnée Y. La trajectoire de saut est définie pour la coordonnée Y. La trajectoire de saut est définie pour la coordonnée Y. D'abord, on monte 5 lignes d'un coup, puis 4, puis 3, etc. jusqu'à ce qu'on atteigne zéro. À ce moment-là, le sens du mouvement est inversé et nous commençons à descendre 1 ligne, puis 2, puis 3, etc. jusqu'à 5.

Pour que le mannequin ne laisse pas de trace lorsqu'il monte et descend, nous devrons avoir un dessin du mannequin qui monte avec 5 lignes noires en dessous pour s'effacer lorsqu'il monte et de la même manière, une autre image avec 5 lignes noires au-dessus pour descendre. Dans ce cas, il s'agit des images 22 et 23 pour sauter à droite et 24,25 à gauche.

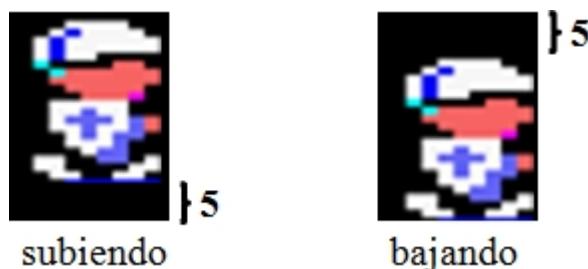


Fig. 60 Image de haut en bas

Au point zénithal du saut, l'image du haut doit être transformée en image du bas, mais il faut d'abord remonter 5 lignes d'un coup, car si l'on compare les deux images, on s'aperçoit que si l'on passe directement de l'une à l'autre, c'est comme si l'on descendait de 5 lignes.

```

10 MÉMOIRE 24999
20 MODE 0 : DEFINT A-Z : CALL &6B78:' install RSX
25 À LA PAUSE GOSUB 2800
30 CALL &BC02 : "restaurer la palette par défaut au cas où".
40 INK 0,0 : "fond noir"
50 FOR j=0 TO 31:|SETUPSP,j,0,&X0:NEXT:'réinitialiser les sprites
80 |SETLIMITS,12,80,0,186 : ' nous fixons les limites de l'écran
90 x=40:y=100:' coordonnées du caractère
100 |SETUPSP,0,0,1:' statut du caractère
110 |SETUPSP,0,7,9:'Séquence d'animation assignée au départ
120 |LOCATESP,0,y,x:'place le sprite (sans l'imprimer)
121 DIM jump(24):' saut de données
122 for i=-5 to 5 : k=k+1:skip(k)=i : k=k+1 : skip(k)=i : next
: skip(11)=-5 : skip(23)=5

125 PLOT 1,150:DRAW 640,150 : plot 92,150:draw 92,400 : "sol et mur
126 |MUSIC,0,0,5 : 'la musique commence à jouer
130 ----- "cycle de jeu".
```

```
150 |LOCATESP,0,y,x:|PRINTSPALL,0,0  
      GOSUB 170  
160 GOTO 130:' fin du cycle de jeu
```

```

170 ' routine de déplacement des caractères -----
171 SI saut =0 ALORS SI INKEY(67)=0 ALORS saut=1:|SETUPSP,0,9,DIR*2+22
180 IF INKEY(27)=0 THEN x=x+1:if jump=0 then IF dir<>0 THEN
|SETUPSP,0,7,9:dir=0:x=x-1:RETURN ELSE |ANIMA,0:GOTO 210
190 SI INKEY(34)=0 ALORS x=x-1:si saut=0 ALORS SI dir<>1 ALORS
|SETUPSP,0,7,10:dir=1:x=x+1:RETURN ELSE |ANIMA,0
210 if jump=0 then RETURN
260 routine de saut -----
270 IF jump=11 THEN |SETUPSP,0,9,DIR*2+23 ELSE IF jump=23 THEN
y=y+jump(jump):jump=0:|SETUPSP,0,7,DIR+9:return
280 y=y+jump(saut)
    jump=jump+1
310 retour
2800 |MUSIQUE:MODE 1 : ENCRE 0,0:STYLO 1

```

Comme vous pouvez le voir dans la liste, si vous appuyez sur la touche "Q", la variable "jump" est égale à 1. À ce moment-là, la logique du mannequin se complique car il faut exécuter une instruction IF pour changer l'image lorsqu'elle atteint le point zénithal et il est également nécessaire de mettre à jour la coordonnée Y du mannequin et la variable "jump".

Plus tard, nous verrons comment faire cela avec une technique plus avancée, en utilisant les "routes" des sprites. **Les routes programmables de 8BP fournissent une méthode plus efficace pour faire ce genre de choses, et vous verrez donc votre personnage sauter beaucoup plus rapidement.** Les routes vous permettront d'exécuter une trajectoire (un saut, un cercle, etc.) sans avoir à vérifier les coordonnées à chaque instant. Et vous pouvez aussi changer l'état d'un sprite au milieu d'une route, ou changer son image associée, sa séquence ou même changer sa route, en concaténant différentes routes.

10 Jeux d'écrans : mise en page ou "carte de tuiles".

10.1 Définition et utilisation de la mise en page

Souvent, vous voudrez que vos jeux consistent en une série d'écrans où le personnage doit collecter des trésors ou esquiver des ennemis dans un labyrinthe. Dans ce cas, il devient indispensable d'utiliser une matrice dans laquelle vous définissez les blocs constitutifs de chaque "labyrinthe" ou de ce que l'on appelle la "disposition" de l'écran. Ce concept est parfois appelé "carte de tuiles" (tile est le mot anglais pour "carreau").

Dans la bibliothèque **8BP**, vous disposez d'un mécanisme simple pour ce faire, qui fournit également une fonction de collision vous permettant de vérifier si votre personnage s'est déplacé dans une zone occupée par une "brique". Ce mécanisme est appelé "disposition". En 8BP, une disposition est définie par une matrice de 20x25 "blocs" de 8x8 pixels, qui peuvent être occupés ou non. Il y a donc autant de blocs qu'il y a de caractères à l'écran en mode 0.

Pour imprimer une mise en page à l'écran, vous disposez de la commande :

| LAYOUT, <y>, <x>, @string\$

Cette routine imprime une rangée de sprites pour construire la disposition ou le "labyrinthe" de chaque écran. La matrice ou "carte de disposition" est stockée dans une zone de mémoire qui gère 8BP, de sorte que lorsque vous imprimez des blocs, vous **n'imprimez pas seulement sur l'écran, mais vous remplissez également la zone de mémoire occupée par la disposition** (20x25 octets), où chaque octet représente un bloc.

Les coordonnées y,x sont transmises sous forme de caractères, c'est-à-dire que y prend les valeurs [0,24].
x prend les valeurs [0,19].

Les blocs imprimés par la fonction |LAYOUT sont construits avec des chaînes de caractères et chaque caractère correspond à un sprite qui doit exister. Ainsi, le bloc "Z" correspond à l'image attribuée au sprite 31, le bloc "Y" correspond à l'image attribuée au sprite 30, et ainsi de suite.

Les sprites à imprimer sont définis par une chaîne de caractères, dont les caractères (32 possibles) représentent l'un des sprites suivant cette règle simple, où la seule exception est l'espace vide représentant l'absence de sprite.

Caractère	Identifiant du sprite	Code ASCII
" "	AUCUN	
" ;"	0	59
"<"	1	
"="		
">"		
" ?"		63
"@"	5	
"A"		65
"B"		
"C"	8	67
"D"		
"E"	10	69
"F"		70
"G"		71
"H"		
"I"		
"J"		
"K"		75
"L"		
"M"		
"N"		78
"O"		79
"P"	21	80
"Q"		81
"R"	23	82
"S"		
"T"	25	84
"U"	26	85
"V"		86
"W"		87
"X"	29	88
"Y"	30	89
"Z"	31	90

Tableau 4 Correspondance des caractères et des sprites pour la commande |AYOUT

La @string est une variable de type chaîne de caractères. Vous ne pouvez pas passer la chaîne directement. En d'autres termes, il serait illégal de faire quelque chose comme :

|AYOUT, 1, 0, "ZZZ YYY".

La réponse est correcte :

String\$ = "ZZZ YYY".

|AYOUT, 1, 0, @string\$

Veillez à ce que la chaîne ne soit pas vide, sinon l'ordinateur risque de se bloquer ! En outre, vous devez préfixer la variable chaîne par le symbole "@" afin de

que la bibliothèque peut aller à l'adresse mémoire où la chaîne est stockée et la parcourir, en imprimant les sprites correspondants un par un.

Vous devez noter que les blancs signifient qu'il n'y a pas de sprite, c'est-à-dire que rien n'est imprimé dans les positions correspondant aux blancs. S'il y avait déjà quelque chose à cet endroit, cela ne sera pas effacé. Si vous voulez effacer, vous devez définir un sprite d'effacement 8x8, où tout est composé de zéros.

Bien que vous utilisiez les sprites pour imprimer la mise en page, vous pouvez, juste après l'impression, redéfinir les sprites avec |SETUPSP et leur attribuer des images de soldats, de monstres ou de ce que vous voulez, c'est-à-dire que la mise en page "s'appuie" sur le mécanisme des sprites pour imprimer, mais ne limite pas le nombre de sprites, car vous pouvez utiliser les 32 sprites comme bon vous semble juste après l'impression de la mise en page.

Pour détecter les collisions avec la disposition, vous disposez de la fonction COLAY, qui peut être utilisée avec un nombre variable de paramètres.

| Les données de l'interface de communication sont les suivantes :

<COLAY,<seuil ASCII>, @collision , <numéro de l'empreinte>.

| COLAY, @collision , <numéro d'empreinte>.

| COLAY, <nombre de caractères>, <nombre de caractères>.

| COLAY

Étant donné un sprite et en fonction de ses coordonnées et de sa taille, cette fonction déterminera s'il entre en collision avec la disposition et vous en avertira par le biais de la variable de collision, qui doit être définie au préalable.

Le paramètre <Seuil ASCII> est facultatif et est utilisé pour que la commande ne considère pas les codes ASCII inférieurs à ce seuil comme des collisions. Par défaut, il est de 32 (ce qui correspond à l'espace blanc). Pour comprendre cela, il faut tenir compte de la correspondance entre les valeurs ASCII et les Sprites montrée dans le tableau précédent. Par exemple, si nous fixons le seuil à 69 (code "E", sprite 10), alors les sprites 9, 8, 7, 6, 5, 4, 3, 2, 1 et 0 ne seront pas "collisibles", donc si notre personnage passe dessus, la collision ne sera tout simplement pas détectée.

Il n'est nécessaire d'invoquer COLAY avec le paramètre de seuil qu'une seule fois, car les invocations suivantes tiennent déjà compte de ce seuil.

Exemple d'utilisation :

col%=0

| COLAY, @col%, 20 : REM voici un exemple avec spriteID=20

Si vous invoquez la commande COLAY sans paramètres, elle prendra en compte les derniers paramètres utilisés. Vous pouvez ainsi économiser le passage des paramètres et accélérer la commande de 0,5 ms.

S'il n'y a pas de collision, la variable prendra la valeur zéro. Il y a collision si le sprite entre en collision avec un élément de mise en page autre qu'un espace blanc (" "), dont le code ASCII est 32. Si le seuil est utilisé, il y a collision si l'élément de mise en page a un code ASCII supérieur au seuil que vous avez défini.

Nous allons voir un exemple de création d'une mise en page et de déplacement d'un personnage dans la mise en page, en corrigeant sa position s'il est entré en collision.

Deux dernières considérations sur la commande **|COLAY** :

- La commande **|COLAY** n'est pas affectée par le paramètre de sensibilité aux collisions des sprites (configurable avec **|COLSP,34, dy, dx**). Le paramètre de sensibilité aux collisions n'affecte que les commandes **|COLSP** et **|COLSPALL**.
- La commande **|COLAY** ne tient pas compte de la taille réelle des images utilisées comme "tuiles" ou "blocs" de la mise en page. En d'autres termes, elle considère que tous les blocs spécifiés dans la commande **|LAYOUT** ont une taille de 8x8 pixels en mode 0 (4 octets x 8 lignes), même si vous placez des images plus grandes.

10.2 Exemple de jeu avec mise en page

Nous allons faire évoluer un peu le gameplay présenté dans le chapitre précédent, en ce qui concerne le contrôle du personnage. Cette fois-ci, nous allons prendre l'exemple de Montoya, qui dispose de 8 séquences d'animation, chacune permettant de se déplacer dans une direction différente. Les séquences d'animation ont été numérotées de 1 à 8.

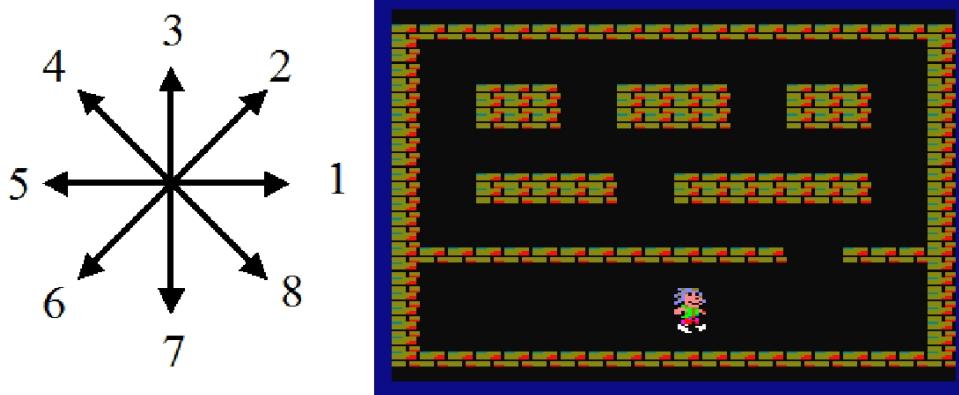


Fig. 61 Utilisation de la mise en page dans un jeu

Dans la routine de contrôle du personnage, nous avons inclus la collision avec la disposition. En fonction de la direction dans laquelle nous nous déplaçons, nous modifions les "nouvelles" coordonnées (yn , xn) et appelons la fonction de collision avec la disposition **|COLAY,0** pour vérifier si le sprite 0 (notre personnage) est entré en collision. Si c'est le cas, nous corrigéons les coordonnées (l'une d'entre elles ou les deux) pour le placer dans une position sans collision avant de l'imprimer à nouveau.

```

10 MÉMOIRE 24999
20 MODE 0 : DEFINT A-Z : CALL &6B78: 'install RSX
25 CALL &bc02 : "restore default palette just in case".
26 encre 0,0 : "fond noir"
30 FOR j=0 TO 31: |SETUPSP,j,0,&X0:NEXT: 'réinitialiser les sprites
40 |SETLIMITS,0,80,0,200 : ' fixe les limites de l'écran de jeu
50 dim c$(25):for i=0 to 24:c$(i)=" ":next
100 c$(1)= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ".
110 c$(2)= "Z"           Z"
120 c$(3)= "Z"           Z"
125 c$(4)= "Z"           Z"
130 c$(5)= "Z   ZZZ   ZZZZ   ZZZ   Z" 
140 c$(6)= "Z   ZZZ   ZZZZ   ZZZ   Z" 

```

150 c\$(7)= "z zzz zzzz zzz z"

10.3 Comment ouvrir un portail dans le schéma

Si vous voulez que votre personnage soit capable de prendre une clé et d'ouvrir une porte ou, plus généralement, d'enlever une partie de la structure pour permettre l'accès, vous devez procéder en deux étapes :

1) Définir un sprite d'essuyage 8x8 où tous les éléments sont des zéros. En utilisant |LAYOUT vous l'imprimez dans les positions que vous souhaitez

2) Ensuite, en utilisant à nouveau |LAYOUT, vous imprimez les espaces que vous avez supprimés. La carte de la disposition se retrouvera alors avec le caractère " " à ces endroits et la fonction de collision avec la disposition aboutira à un résultat nul.

Dans le jeu "Mutant Montoya", cette technique est utilisée pour ouvrir la porte du château, ainsi que les portes menant à la princesse.



Fig. 62 Modification de la présentation lors de la prise de la clé

L'exemple suivant illustre le concept en ouvrant une porte aux coordonnées (10, 12) avec une taille de 2 blocs, en saisissant une clé définie avec le sprite 16.

Dès que vous ramassez la clé, la porte s'ouvre et la clé est désactivée afin de ne plus évaluer la collision avec elle, c'est-à-dire que la commande |COLSP renverra un 32 à partir du moment où vous ramassez la clé si vous entrez à nouveau en collision avec elle.

Après l'ouverture de la porte, si vous déplacez le personnage à l'endroit où se trouvait la porte, la collision avec le tracé se traduira par 0.

```
'----- cette partie se trouve à l'intérieur de la boucle
logique ---- 6410 |PRINTSPALL,1,0
6411 |COLSP,@cs%,0:IF cs%<32 THEN IF cs%>=15 then gosub 6500 (
plus d'instructions . . . .)

'----- routine d'ouverture de la porte -----
6499 ' vérifiez que votre collision se fait avec la clé, qui est le sprite 16
6500 delete$="MM":spaces$=" ":" le sprite de suppression a été défini
comme "M" (M est le sprite 18 dans la "langue" de la commande |LAYOUT)
6501 if cs%=16 then |LAYOUT,10,12,@delete$ : |LAYOUT,10,12,@spaces$ :
|SETUPSP,16,0,0,0
6502 retour
```

10.4 Un jeu de puzzle : LAYOUT avec un arrière-plan

Ensuite, nous allons voir un exemple qui utilise la disposition et l'écrasement, pour lequel il établit un seuil ASCII qui permet à la commande |COLAY de ne pas envisager de collision avec les éléments d'arrière-plan. Concrètement, l'élément de fond est la lettre "Y", qui correspond au sprite id= 30, et l'ASCII du "Y" est 89.



Fig. 63 Mise en page avec motif d'arrière-plan et écrasement

Comme vous pouvez le voir dans l'exemple, il n'est nécessaire d'invoquer COLAY avec le paramètre seuil qu'une seule fois, puisque les invocations successives prennent déjà en compte ce seuil.

Un autre aspect intéressant est la gestion du clavier de cet exemple. Elle est optimale pour exécuter le plus petit nombre d'opérations possible tout en donnant une sensation très agréable lorsqu'on se déplace dans un couloir et qu'on se connecte à un autre couloir en appuyant sur deux touches en même temps.

```

10 MÉMOIRE 23999
20 MODE 0 : DEFINT A-Z : CALL &6B78:' install RSX
21 on break gosub 5000
25 call &bc02 : "restore default palette just in case".
26 gosub 2300:' palette avec écrasement
30 FOR j=0 TO 31:|SETUPSP,j,0,&X0:NEXT:'réinitialiser les sprites
40 |SETLIMITS,0,80,0,200 : ' limites de l'écran de jeu
45 |SETUPSP,30,9,&84d0:' grille de fond ("Y")
46 |SETUPSP,31,9,&84f2:' brique ("Z")
50 dim c$(25):for i=0 to 24:c$(i)="":next
100 c$(1)= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ".
110 c$(2)= "ZYYYYYYYYYYYYYYYYYYYYYYYZ".
120 c$(3)= "ZYYYYYYYYYYYYYYYYYYYYYYYZ".
125 c$(4)= "ZYZZZYZZZZZZZZZZZZZZZZZZYYZ".
130 c$(5)= "ZYZZZYZZZZZZZZZZZZZZZZZZZZYYZ".
140 c$(6)= "ZYYYYYYYYYYYYYYYYYYYYYYYZ".
150 c$(7)= "ZYYYYYYYYYYYYYYYYYYYYYYYZ".
160 c$(8)= "ZYZZZZZZZZZZZZZZZZZZZZZZYYZ".
170 c$(9)= "ZYZ      ZYZ      ZYZ"
190 c$(10)="ZYZ      ZYZ      ZYZ"
195 c$(11)="ZYZZZZZZZZZZZZZZZZZZZZYYZ"
.
200 c$(12)="ZYYYYYYYYYYYYYYYYYYYZ".
210 c$(13)="ZYYYYYYYYYYYYYYYYYYYZ".
220 c$(14)="ZYZZZYZZZZZZZZZZZZZZZZZZ
      YZ".
230     c$(15)="ZYYYYYYYYYYYZ
      ZYYYYYYYYYYZ"
240     c$(16)="ZYYYYYYYYYYYZ
      ZYYYYYYYYYYZ"

```

c\$(17)="ZYZZZZZZZZYZZZZZZZZZ
ZZYZ".
260 c\$(18)="YYYYYYYYYYYYYYYYYYYYYZ".
270 c\$(19)="YYYYYYYYYYYYYYYYYYYYYZ".

```

271 c$(20)="ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ".
272 c$(21)=""
273 c$(22)=""
274 c$(23)=""
    nous imprimons la mise en page
310 FOR i=0 TO 20:|LAYOUT,i,0,@c$(i):NEXT
311 locate 1,1:pen 9:print "DEMO OVERWRITE".
312 locate 3,23:pen 11:print "BASIC using 8BP".
320 |SETUPSP,0,0,&x01000111:' détection des collisions des sprites et de la
    mise en page
330 |SETUPSP,0,7,1:dir=1 : ' séquence = 1 (noix de coco à droite)
340 xa=20*2:xn=xa:ya=12*8:yn=ya:' coordonnées du caractère
350 |LOCATESP,0,ya,xa : 'place le caractère (sans l'imprimer)
360 |PRINTSPALL,0,1,0:' imprime les sprites
361 cl%=0:' variable de collision
362 |COLAY,89,cl%,0:'seuil chr$("Y") est 89
    LE JEU COMMENCE
401 |MUSIC,0,0,5
402 ' lecture du clavier et collisions. si nous allons dans la direction H (ou
    p), nous commençons par
    vérifier si la touche de direction V (q a) est enfoncée et vice versa.
404 if dirn <3 then gosub 450 : gosub 410 else gosub 410:gosub 450
405 |LOCATESP,0,yn,xn:|PRINTSPALL
406 ya=yn:xa=xn
407 goto 404

409 ' direction horizontale du clavier ---
410 si INKEY(27)<0 alors 430
420 xn=xa+1poke 27003,xn:|COLAY : IF cl%=0 then if dir<>>1 then
    |DIR=1:xn=xa:return else dirn=1:return
421 xn=xa:poke 27003,xn:return:'il y a collision
430 si INKEY(34)<0 alors retour
440 xn=xa-1:poke 27003,xn:|COLAY : IF cl%=0 then if dir<>>2 then
    |SETUPSP,0,7,2:DIR=2:xn=xa:return else dirn=2:return
441 xn=xa:poke 27003,xn:'il y a collision
442 retour
449 direction verticale du clavier
450 si INKEY(67)<0 alors 480
460 yn=ya-2:poke 27001,yn:|COLAY : IF cl%=0 then if dir<>>3 then
    |SETUPSP,0,7,3:DIR=3:yn=ya:return else dirn=3:return
461 yn=ya:poke 27001,yn:'il y a collision
    if INKEY(69)<0 then return
490 yn=ya+2:poke 27001,yn:|COLAY : IF cl%=0 then if dir<>>4 then
    |SETUPSP,0,7,4:DIR=4:yn=ya:return else dirn=4:return
491 yn=ya:poke 27001,yn:'il y a collision
492 retour

2300 REM ----- PALETA sprites transparents MODE 0-----
2301 INK 0,11 : REM bleu clair
2302 INK 1.15 : REM orange
2303 INK 2,0 : INK 3,0 : REM noir
2305 INK 4,26 : INK 5,26 : REM blanc
2307 INK 6,6 : INK 7,6 : REM rouge
2309 INK 8,18 : INK 9,18 : REM vert
2311 INK 10.24 : INK 11.24 : REM jaune
2313 INK 12,4 : INK 13,4 :REM magenta
2315 INK 14,16 : INK 15, 16:orange REM
2317 JAUNE=10
2420 RETOUR
    |MUSIQUE

```

5010 fin

10.5 Comment économiser de la mémoire dans vos mises en page

Si votre jeu comporte de nombreux écrans et que vous devez économiser de l'espace, vous pouvez utiliser plusieurs techniques simples pour économiser de la mémoire. Un écran consomme environ 0,5 Ko, il est donc important d'utiliser des méthodes pour réduire sa taille.

La façon la plus simple de procéder est d'éditer les écrans comme s'il s'agissait de sprites. Vous les éditez avec SPEDIT (par exemple) et chaque pixel représentera un élément de la mise en page. Selon la couleur, il représentera des rochers, des briques, de l'espace vide, de l'eau, de la terre, etc. Comme il y a 16 couleurs en mode 0 disponibles, vous aurez 16 types de briques. Le sprite que vous générerez devra être stocké sous forme d'image et, avant de l'afficher à l'écran, il faudra le convertir en layout, en le scannant pixel par pixel et en le transformant en code ASCII dont vous avez besoin (vous devrez le programmer en BASIC ou ce que vous voulez). Si nous considérons que vous utiliserez 5 lignes de caractères pour les marqueurs du jeu, un écran consommera 20x20 pixels = 400 pixels = 200 octets. Par conséquent, 1KB peut contenir 5 écrans et 10KB peuvent contenir 50 écrans. Vous aurez utilisé 4 bits par élément de mise en page.

L'édition d'écrans comme s'il s'agissait de sprites est très "visuelle", bien que vous puissiez décider d'utiliser moins de bits par élément de mise en page. Si vous n'utilisez que 2 bits, vous aurez 4 types d'éléments et vous pourrez également éditer des écrans comme s'il s'agissait d'images, en utilisant le MODE 1. Dans ce cas, vous pourrez faire tenir 100 écrans dans 10 Ko. Si vous utilisez un nombre différent de bits par brique, les choses se compliquent car vous ne pouvez pas dessiner les écrans comme s'il s'agissait de sprites, mais vous pouvez programmer quelque chose qui convertit une image que vous dessinez en bits dont vous avez besoin.

Une autre solution simple et efficace consiste à définir chaque disposition avec de grands blocs, par exemple 8x16 pixels. Et de construire les écrans en les définissant avec des caractères que l'on peut stocker en mémoire. Dans le jeu vidéo "**Happy Monty**", le premier écran de "Mutant Monty" est recréé avec une matrice de 16x10 caractères = 160 octets, de sorte que 25 écrans occupent 4 Ko. Il est vrai que cette disposition n'a que 16 blocs de large, au lieu des 20 possibles, et que seuls 10 blocs sont définis verticalement, au lieu des 25 possibles, mais elle occupe ainsi très peu de mémoire et nous pouvons créer de nombreux écrans.



Fig. 64 : une mise en page définie avec 160 caractères

11 Programmation avancée et "bulk logics".

11.1 Mesure de la vitesse des commandes

L'interpréteur BASIC est très lourd dans son exécution car il ne se contente pas d'exécuter chaque commande, il analyse également le numéro de ligne, analyse la commande saisie, valide son existence, le nombre et le type de paramètres, que leurs valeurs sont dans des plages valides (par exemple, PEN 40 est illégal) et bien d'autres choses encore. C'est l'analyse syntaxique et sémantique de chaque commande qui pèse vraiment, et pas tant son exécution. Le cas des commandes RSX ne fait pas exception. L'interpréteur BASIC vérifie leur syntaxe et cela pèse beaucoup, même s'il s'agit de routines écrites en ASM, car avant de les invoquer, l'interpréteur BASIC a déjà fait beaucoup de choses.

Il faut donc économiser les exécutions de commandes en programmant intelligemment pour que la logique du programme passe par le moins d'instructions possible, quitte à en écrire parfois davantage. Une pratique indispensable consiste à utiliser des instructions qui déplacent ou affectent un groupe de sprites, comme COLSPALL,

|L'utilisation de boucles avec des instructions qui n'affectent qu'un seul sprite peut être évitée grâce à l'utilisation de l'option AUTOALL ou |MOVERALL.

Le nombre de paramètres est un facteur décisif lors de l'invocation d'une commande. Plus il y a de paramètres, plus l'interprétation par BASIC est coûteuse, même s'il s'agit d'une routine ASM invoquée par CALL, car la commande CALL est toujours BASIC et avant d'accéder à la routine dans ASM, le nombre et le type de paramètres sont inévitablement analysés.

Pour évaluer le coût d'exécution d'une commande, vous pouvez utiliser le programme suivant. Vous pouvez également l'utiliser pour évaluer les performances des nouvelles fonctions assembleur que vous incorporez dans la bibliothèque 8BP si vous le souhaitez.

```
1 appel &6b78
10 MÉMOIRE 23499
11 DEFINT a-z
12 c%=0 : a=2
30 FOR i=0 TO 31:|SETUPSP,i,0,0,0:NEXT:'reset
31 itérations=1000
40 a!= TEMPS
50 FOR i=1 TO iterations
60 <i ci vous mettez une commande, par exemple PRINT "A">
70 NEXT
80 b!=TIME
90 PRINT (b!-a !): rem ce qu'il faut en unités de temps cpc. (1/300
secondes)
100 c!=((b!-a !)*1/300)/itérations : rem c!= durée de chaque itération en
secondes
120 d!=(1/50)/c !
130 PRINT "vous pouvez exécuter ",d !, "des commandes par balayage (1/50
sec)".
140 PRINT "la commande prend " ;(c!*1000 -0.47) ; "millisecondes" ; "la
commande prend " ;(c!*1000 -0.47) ; "millisecondes".
```

Note : Pour les experts en langage d'assemblage, vous devez noter que si vous avez l'intention de mesurer le temps d'exécution d'une routine qui désactive les interruptions en interne (en utilisant les instructions DI, EI), le temps écoulé pendant la désactivation n'est pas mesurable avec ce programme BASIC. Les commandes 8BP ne désactivent pas

les interruptions et sont toutes mesurables.

Voyons ci-dessous les performances de certaines commandes (mesurées avec le programme ci-dessus). Force est de constater qu'il est plus rapide d'exécuter un appel direct à l'adresse mémoire (un CALL &XXXX) que d'invoquer la commande RSX correspondante. Dans le tableau suivant, il est évident que plus le résultat est faible (exprimé en millisecondes), plus la commande est rapide. Le tableau présenté ici doit être gardé à l'esprit à tout moment et vous devez prendre vos décisions de programmation sur la base de ce tableau. Il s'agit d'un tableau de mesures des commandes BASIC et des commandes 8BP.

Commandement	ms	Commentaire
IMPRIMER "A"	3.63	Très lent. Ne pensez même pas à l'utiliser, sauf occasionnellement pour changer le nombre de vies, mais n'imprimez pas de score dans un jeu pour chaque ennemi que vous tuez.
LOCATE 1,1 : Points d'impression	24.8 + 7	Placer le curseur de texte avec LOCATE et imprimer la variable de valeur lunaire "points" est très coûteux. Si vous mettez à jour les points, ne le faites que de temps en temps et pas à chaque cycle de jeu.
C\$=str\$(points) PRINTAT,0, y, x, @c\$	10	L'impression des points à l'aide de PRINTAT est beaucoup plus efficace que l'utilisation de LOCATE + PRINT (=32 ms), mais reste coûteuse. Utilisez PRINTAT avec parcimonie.
REM hello	0.20	Les commentaires consomment
' bonjour	0.25	Vous économisez 2 octets de mémoire, mais c'est plus lent !
GOTO 60	0.19	Très rapide ! Encore plus rapide que REM. Utilisez cette commande sans pitié, utilisez-la !!!
A = 3	0.55	Une simple mission coûte. Tout coûte, chaque instruction doit être réfléchie.
A = B	0.72	Assigner la valeur d'une variable à une autre est plus coûteux que d'assigner une valeur. Et affecter la valeur d'un tableau est encore plus coûteux, car l'accès au tableau coûte. Et si le tableau est bidimensionnel, cela coûte encore plus cher.
A = miarray(x)	1.33	
A= miarray(x,y)	1.84	
 LOCATESP,i,10,20	2.8	Si vous n'utilisez pas de coordonnées négatives, il est préférable d'utiliser la commande BASIC POKE pour définir les coordonnées.
 LOCATESP,i,y,x	3.22	Si les coordonnées sont variables, cela prend plus de temps.
CALL &XXXX,i,x,y	1.81	L'équivalent de l'APPEL est beaucoup plus rapide.
 MOVER,31,1,1	3.23	Il est un peu lent et doit donc être utilisé avec parcimonie.
CALL &XXXX,31,1,1	1.77	L'appel équivalent est beaucoup plus rapide

POKE &XXXX, valeur	0.71	Très rapide ! Utilisez-le pour mettre à jour les coordonnées du sprite (si elles sont positives). POKE n'accepte pas les nombres négatifs, mais vous pouvez utiliser la formule 255+x+1 si vous voulez entrer un nombre négatif. Par exemple, pour entrer un -4, vous devez entrer 255-4+1=252 Une autre façon simple d'entrer des nombres positifs et négatifs est d'utiliser l'adresse POKE, x et 255.
POKE dir,data	0.85	Très rapide si l'on considère qu'il doit également traduire la variable "dir".
 POKE,&xxxx,valeur	2.5	Il autorise les nombres négatifs et, si vous ne mettez à jour qu'une seule coordonnée (X ou Y), il est meilleur que LOCATESP.
X=PEEK(&xxxx)	0.93	Très rapide ! Selon le type de jeu, il peut être une alternative à COLSP, en regardant la couleur d'une adresse mémoire de l'écran. Dans l'annexe sur la mémoire vidéo, j'explique comment procéder.
X=INKEY(27)	1.12	Très rapide. Convient aux jeux vidéo, bien qu'il faille l'utiliser intelligemment comme recommandé dans ce livre.
SI x>50 ALORS x=0	1.42	Chaque IF pèse, il faut essayer de les sauver car une logique de jeu va avoir de nombreuses conséquences.
IF A=value THEN GOTO 100 Vs SI A=valeur ALORS 100	1.24 Vs 1.18	Les deux phrases sont équivalentes mais la seconde prend moins de temps.
SI inkey(27)=0 alors x=5	1.75	C'est acceptable. C'est plus rapide que de faire b=INKEY(27) et ensuite IF...THEN
10 Si inkey(27) puis 30 20 x=5 30 <instructions>.	1.0	Une façon beaucoup plus efficace de faire la même chose
SI x>0 alors Vs SI x alors	1.3 Vs 0,8	En BASIC, il est possible d'économiser 0,5 ms en considérant que toute valeur non nulle signifie VRAI. Si nous voulons contrôler une valeur spécifique, nous le ferons : 10 SI x-20 ALORS 30 20 <choses à faire si x=20> 30 ... L'utilisation de cette technique est fortement recommandée pour la lecture au clavier.

A=A+1 : SI A>4 alors A=0	2.6	Il est de loin préférable d'utiliser la deuxième option (utilisation du MOD). D'un autre côté, l'utilisation du MOD doit se faire avec prudence. Si nous le faisons : A=(A+1) MOD 3
Vs	Vs	
A=A MOD 3 +1	1.7	Cela nous coûte 2 ms parce que les parenthèses sont très chères et pourtant nous obtenons la même chose. Si nous voulons compter à partir d'un nombre différent de 1, nous introduisons n'importe quel nombre impair et cela suffit.
A=A MOD 3 + <impair>		Il existe un moyen encore plus rapide d'y parvenir, avec l'opérateur binaire ET, que nous allons maintenant étudier.
A=1+A ET 7 (valeur initiale 0 Valeur finale 7)	1.6	Cela vous permet de faire varier une variable de manière cyclique entre N valeurs afin de choisir un ID de sprite pour votre nouveau tir ou pour un ennemi qui entre à l'écran.
A=20+A MOD 7 (valeur initiale 0 Valeur finale 26)	1.88	Il est préférable d'utiliser AND plutôt que MOD, car AND est une opération binaire rapide et MOD implique une division, ce qui est très coûteux pour notre cher microprocesseur Z80. Cependant, si nous devons utiliser des sprites ID qui ne commencent pas par 1, nous aurons besoin de parenthèses car l'opérateur "+" est prioritaire sur "AND" et l'avantage de la rapidité de AND est perdu. Dans ce cas, l'opérateur
A=21 + (A et 7) (valeur initiale 21 Valeur finale 28)	1.95	
		est le meilleur MOD. Quoi qu'il en soit, essayez toujours et choisissez, car en fonction du nombre initial à ajouter, vous obtiendrez des temps différents. Incroyable mais vrai 20+A mod 7 → prend 1,88 29+A mod 7 → prend 1,94
Si A <0 alors A=15	1.71	Vérifier qu'un nombre n'est pas négatif
A=A ET 15	1.24	Le test peut être simplifié car un nombre négatif est en fait un nombre qui a un "1" dans le bit le plus significatif.
:	0.05	Cela ne permet pas d'économiser beaucoup, mais il est plus rapide d'utiliser ":" au lieu d'un nouveau numéro de ligne, et si vous appliquez cela plusieurs fois, vous finirez par économiser de manière significative. Deux instructions sur deux lignes prendront 0,03 ms de plus que si les deux sont sur la même ligne, séparés par ":".
 PRINTSP,0,10,10	5.3	Un seul sprite 14 x 24 (7 octets x 24 lignes) Si vous devez imprimer plusieurs sprites, il est préférable d'imprimer tous les sprites en une seule fois avec PRINTSPALL.
CALL &xxxx,0,10,10	3.5	Équivalent de PRINTSP, donc plus rapide, mais moins lisible.

 PRINTSPALL (32 sprites 8x16 du mode 0, c'est-à-dire 4 octets x 16 lignes)	<p>55.4 Cela représente environ 18 fps à pleine charge de sprites. Ce qu'il faut, c'est</p> $T = 3,25 + N \times 1,7$ <p>Soit 1,7 ms par sprite et un coût fixe de 3 ms. Ce coût fixe est le coût de l'analyse BASIC plus le coût de la recherche des sprites à imprimer dans la table des sprites. Si vous omettez les paramètres (c'est possible et vous prendrez les valeurs de la dernière invocation), vous économisez 0,6 ms dans la partie fixe, c'est-à-dire 1,7 ms par sprite :</p> $T = 2,6 + N \times 1,1$ <p>Si l'impression est une surimpression et/ou une impression à plat, elle est plus coûteuse. Les coûts relatifs de chaque type d'impression sont indiqués ci-dessous :</p> <p>Impression normale : 100%. Impression avec écrasement : 164% Impression retournée : 179% Impression avec écrasement : 164% Impression retournée : 179% Impression retournée : 179% Impression retournée avec écrasement : 220%.</p>
 PRINTSPALL,N,0,0 (pas de sprite actif) N=0 N=10 N=31	<p>Coût de la commande de sprites :</p> <p>Lorsque N=0, sans sprites à imprimer, la fonction doit parcourir la table des sprites séquentiellement. Mais parcourir la table dans l'ordre est plus coûteux, comme le montre le temps consommé lorsque N augmente. La différence de temps (5,9 -2,6 =2,5ms) correspond à ce qu'il en coûte pour trier tous les sprites.</p>
 COLAY,@x%,0	<p>3.0 A n'utiliser que sur le personnage, pas sur les ennemis, sinon le jeu sera ralenti. Si le personnage est un multiple de 8, le jeu est plus rapide. Dans cet exemple, la taille du personnage était de 14x24 et il est logique qu'elle soit de 14x24.</p>
 COLAY	<p>2.4 n'est pas un multiple de 8. Plus le sprite est grand, plus cela prend de temps. Si vous invoquez la commande sans paramètres, elle est beaucoup plus rapide (vous gagnez 0,6 ms) !</p>
 COLAY vs APPELER &XXXX	<p>2.4 vs 2.0 L'utilisation de CALL comme d'habitude est plus rapide, mais moins lisible.</p>
GOSUB / RETURN	<p>0.56 Rapidité acceptable. La mesure a été effectuée à l'aide d'une routine qui ne fait que le retour.</p>
 SETUPSP, id, paramètre, valeur	<p>2.7 Acceptable, bien que POKE soit bien meilleur pour certains paramètres. Certains paramètres peuvent être définis avec POKE, comme l'état, mais pas d'autres (comme un itinéraire). Voir le guide de référence</p>

FOR / NEXT	0.6	Vous pouvez l'utiliser pour traverser plusieurs ennemis et faire en sorte que chacun d'entre eux se déplace selon la même règle. Vous devriez vous demander si vous pouvez utiliser AUTOALL ou MOVEALL pour vos besoins, car une seule commande déplacera tous les ennemis que vous voulez, ce qui est bien mieux qu'une boucle.
 COLSP,31, @c%	5.5	La durée est à peu près la même quel que soit le nombre de sprites actifs. Évitez d'appeler toujours la variable de collision pour accélérer le processus à 4,3 ms.
 COLSP,31	4.3	Si vous avez un navire ou un personnage et plusieurs tirs, il est beaucoup plus efficace d'invoquer COLSPALL plutôt que d'invoquer COLSP plusieurs fois.
 ANIMALL (cette commande n'est disponible qu'avec un paramètre de PRINTSPALL, elle n'est pas disponible à partir de peut être invoqué directement)	3.5	Elle est coûteuse mais il existe un moyen de l'invoquer en même temps que PRINTSPALL , par un paramètre qui fait que cette fonction est invoquée avant l'impression des sprites. Cela permet d'économiser la couche BASIC, c'est-à-dire le temps nécessaire à l'envoi de la commande, qui est >1ms. On peut donc dire que cette commande consommera normalement un peu moins de 2ms.
 AUTOALL	2.76	Il est peu coûteux et peut déplacer les 32 sprites à la fois.
 MOVEALL,1,1	3.4	Il n'est pas très coûteux et peut déplacer les 32 sprites en même temps.
SON	10	La commande de son est "bloquée" dès que le tampon de 5 notes est plein. Cela signifie que votre logique BASIC ne doit pas enchaîner plus de 5 commandes SOUND sous peine de s'arrêter jusqu'à ce qu'une note se termine... Si vous décidez de l'utiliser, vous devez être très prudent car elle est très gourmande. le temps d'exécution (10 ms est très long)
SI a>1 ET a>2 ALORS a=2	2.52	Un moyen simple d'économiser 0,13 ms
Versus	Vs	Dans tout ce que vous programmez, gardez ces détails à l'esprit, chaque économie est importante.
SI a>1 ALORS SI a>2 ALORS a=2	2.39	
A=RND*10	4.2	La fonction BASIC RND est très coûteuse. Vous pouvez l'utiliser, mais pas à chaque cycle de jeu, seulement éventuellement, par exemple, lorsqu'un nouveau cycle de jeu est lancé. ennemi ou autre. Une autre solution simple consiste à stocker
		10 nombres aléatoires dans un tableau et les utiliser au lieu d'invoquer RND

Bordure <x>	0.75	Assez rapide. Il est utile de l'utiliser en combinaison avec une sorte de collision de sprites, pour renforcer l'effet explosif.
SI a ET 7 alors 30	1.19	J'ai indiqué le temps d'exécution lorsque la condition est remplie. Les deux cas sont assez rapides.
SI A MOD 8 alors 30	1.29	
ON x GOTO L1,L2,L3,L4 Vs 60 si x >2 alors 63 61 si x=1 alors 70 62 goto 70 63 si x=3 alors 70 64 goto 70	3.67 Vs 4.8	Une commande ON GOTO est en moyenne 1 ms plus rapide que son équivalent avec les commandes "IF", bien que cela dépende également de la probabilité d'occurrence de chacune des 4 valeurs. Si la probabilité des 4 valeurs est la même, on peut gagner 1 ms en utilisant ON GOTO Il peut être optimisé comme suit 10ON X GOTO 30,40,50 20 <instructions case x=4> : GOTO 60 30 <instructions case x=1> : GOTO 60 40 <instructions case x=2> : GOTO 60 50 <instructions case x=3> : GOTO 60 60 poursuite du programme Avec cette stratégie, nous sommes descendus à 3,54 ms.

Tableau 5 Liste des temps d'exécution de certaines instructions

Recommandations importantes :

- Utilisez le **DEFINT A-Z au début du programme**. Les performances s'en trouveront considérablement améliorées. Cette commande est presque obligatoire. Cette commande supprime toutes les variables qui existaient auparavant et force toutes les nouvelles variables à être des nombres entiers, sauf indication contraire par des modificateurs tels que "\$" ou "!" (voir le guide de référence du programmeur Amstrad BASIC). Notez que lorsque vous utilisez DEFINT, si vous voulez associer un nombre supérieur à 32768, vous devrez le faire en hexadécimal.
- Si vous pouvez éviter de passer par un IF en insérant un GOTO, il sera toujours préférable de
- Lorsque vous manquez de vitesse et que vous avez besoin d'un peu plus de vitesse, utilisez l'**ACBD**.
<adresse> au lieu de RSX. Dans ce cas, vous devez transmettre des paramètres contenant des nombres négatifs au format hexadécimal.
- Ne synchronisez pas la commande **|PRINTSPALL** avec le balayage de l'écran à moins que votre jeu ne soit très rapide. La synchronisation peut réduire le nombre d'images par seconde. En général, tant que vous obtenez 12 FPS, votre jeu est "jouable".

- **Éliminez les espaces blancs.** Chaque espace vide dans votre liste BASIC consomme 0,01ms d'exécution.
- **Raccourcissez le nom de vos variables.** Plus ils sont longs, plus leur accès est coûteux.

fonctionnement	Météo
A=A+1	Une lettre, prend 1,18 ms
HO=HO+1	Deux lettres : 1,2 ms (2 % de plus)
HELLO=HELLOA+1	5 lettres, 1,25 ms (6 % de plus)
HELLOFRIENDS=HELLOFRIENDS +1	10 lettres 1,34 ms (13% de plus)

- **Réduire le nombre de variables.** S'il y a beaucoup de variables, les accès en lecture et en écriture sont plus lents.
- Une fois que vous avez invoqué avec des paramètres la commande **|STARS** ou la commande **|La bibliothèque 8BP** a une "mémoire" et utilisera les derniers paramètres que vous avez utilisés. La bibliothèque 8BP a de la "mémoire" et utilisera les derniers paramètres que vous avez utilisés. Cela permet d'économiser des millisecondes en passant par la couche d'analyse de l'interpréteur BASIC.
- Gardez toujours à l'esprit qu'une expression non nulle est VRAIE. Cela vous permettra d'économiser 0,5 ms sur chaque IF et peut être utilisé pour la lecture du clavier et le contrôle des variables.

Mauvaise option	Bon choix (gain de 0,5 ms)
SI x<>>0 ALORS <instructions>	SI x ALORS <instructions>
SI x=20 ALORS...	10 SI x-20 ALORS 30 20 <instructions>. 30
SI INKEY(34)=0 ALORS <instructions>.	10 IF INKEY(34) THEN 30 20 <instructions>. 30

- Dans les jeux de vaisseaux où vous n'utilisez pas l'écrasement, assurez-vous que votre vaisseau est un sprite 31, afin qu'il passe "par-dessus" les sprites qui prétendent être l'arrière-plan, car votre vaisseau sera imprimé par la suite.
- Essayez d'autres versions de la même opération

A=A+1:IF A>4 then A=0 : REM this consumes 2.6ms A=A
MOD 3 +1 : REM this consumes 1.84 ms
A=1 + A AND 3 : REM cela consomme 1,6 ms
- Évitez d'utiliser des coordonnées négatives. Cela vous permettra d'utiliser POKE pour mettre à jour la position de votre personnage. La commande POKE (celle de BASIC) est très rapide mais ne supporte que les nombres positifs, tout comme PEEK. Si vous utilisez des coordonnées natives, utilisez **|POKE** et **|PEEK** (commandes 8BP). Réservez l'utilisation de **|LOCATESP** aux cas où vous allez modifier les deux coordonnées en même temps et où elles peuvent être positives et/ou négatives. Rappelez-vous également qu'un POKE d'une valeur x négative peut être effectué en utilisant l'adresse **POKE, 255+x+1**. Si vous souhaitez utiliser des coordonnées négatives pour montrer comment les ennemis entrent lentement

dans l'écran par le côté gauche (écrêtage), vous pouvez éviter les coordonnées suivantes

L'utilisation d'un **|SETLIMITS** et ainsi produire le même effet avec des coordonnées commençant à zéro et un écran de jeu légèrement plus petit.

- Si vous devez vérifier quelque chose, ne le faites pas à chaque cycle de jeu. Il peut être suffisant de vérifier ce "quelque chose" tous les 2 ou 3 cycles, sans avoir besoin de le vérifier à chaque cycle. Pour pouvoir choisir quand exécuter quelque chose, utilisez l'"arithmétique modulaire". En BASIC, l'instruction MOD est un excellent outil. Par exemple, pour exécuter une fois sur 5, vous pouvez faire : **IF cycle MOD 5 = 0 ALORS** ... bien qu'il soit préférable d'utiliser les opérations **AND** plutôt que les opérations **MOD**.
- Utilisez les "**séquences de mort**". Cela vous permettra d'enregistrer des instructions pour vérifier si un sprite en train d'explorer a atteint sa dernière image d'animation afin de le désactiver.
- L'écrasement est coûteux : si vous pouvez créer votre jeu sans écrasement, vous économiserez des millisecondes et gagnerez en couleur. Utilisez-la quand vous en avez besoin, mais pas sans raison.
- Les macros d'animation vous permettent d'économiser des lignes de BASIC car vous n'avez pas besoin de vérifier la direction du mouvement du sprite. Utilisez-les chaque fois que vous le pouvez.

11.2 Notre Amstrad n'a que 64 Ko et si vous ne tenez pas compte de la mémoire vidéo, de la mémoire pour vos sprites, de votre musique et de la bibliothèque 8BP, il vous reste 24 Ko de BASIC à utiliser.

très bien. Si chaque écran a son propre "programme" dans votre jeu, vous pourrez à peine faire un jeu de 10 écrans.

Il y a deux choses que vous devriez essayer de faire pour réduire l'utilisation de la mémoire

- Créer des écrans à faible nombre d'octets
- Créez une logique unique qui régit tous les écrans, c'est-à-dire que le même cycle de jeu doit être exécuté sur tous les écrans du jeu.

Dans les jeux à passage d'écran qui utilisent la mise en page, chaque écran peut occuper 20x25 octets, soit 500 octets. En utilisant certaines "astuces" expliquées au chapitre 8, il est possible de réduire cette mémoire. Dans le jeu vidéo "**Happy Monty**", 25 écrans sont construits avec seulement 160 octets chacun et il y a une seule logique de cycle de jeu pour tous les écrans.

Il est très important que vous programmiez une seule logique de cycle de jeu et que vous l'appliquiez à tous les écrans. Si vous programmez une logique de cycle de jeu pour chaque écran, le code source de votre programme sera énorme et vous ne pourrez programmer que peu d'écrans car vous manquerez rapidement de mémoire.

Vous pouvez également surmonter les limitations de mémoire en utilisant des algorithmes qui génèrent des labyrinthes ou des écrans sans avoir besoin de les stocker. Vous pouvez ainsi créer beaucoup plus d'écrans. Cela demande de la créativité, bien sûr, mais c'est possible. Un algorithme

occupe toujours moins d'espace que les données qu'il génère, même si, logiquement, son exécution prend plus de temps que la simple lecture des données stockées.

Vous pouvez réutiliser la logique de l'ennemi d'un écran à l'autre, ce qui permet d'économiser des lignes de code. Pour cela, utilisez le mécanisme GOSUB/RETURN. Il est également très utile d'utiliser des itinéraires pour les ennemis. **Avec le mécanisme de parcours, l'ennemi se déplace sans exécuter de logique BASIC** et fonctionne très rapidement. Il suffit d'assigner un chemin à un ennemi pour qu'il l'exécute encore et encore sans avoir besoin de coûteuses instructions IF, d'affectations, etc.

Vous pouvez également créer des jeux qui se chargent par étapes, de sorte que vous n'avez pas tout le jeu en mémoire en même temps. C'est un peu gênant pour l'utilisateur de bandes (CPC464) mais pas pour l'utilisateur de disques (CPC6128).

Utilisez le **retournement de sprites** pour économiser de la mémoire sur vos sprites

11.3 Technique de la "logique de masse"

Vous devrez souvent déplacer un grand nombre de sprites, en particulier dans les jeux d'arcade spatiale ou de style "commando" (le classique de Capcom de 1985).

Vous pourriez agir séparément sur les coordonnées de tous les sprites et les mettre à jour à l'aide de POKE, mais ce serait très lent et irréalisable si vous voulez que les mouvements soient fluides. Le meilleur moyen (et le plus simple) est d'utiliser conjointement les fonctions de mouvement automatique et de mouvement relatif, qui sont respectivement |AUTOALL et |MOVERALL.

La clé pour atteindre la vitesse dans de nombreux sprites est d'utiliser la technique que j'ai appelée "logique massive". Cette technique consiste essentiellement à exécuter moins de logique par cycle de jeu (ce que l'on appelle "réduire la complexité de calcul") et il existe plusieurs options pour y parvenir :

- Utiliser **une logique unique** qui affecte plusieurs sprites à la fois (en utilisant les drapeaux de mouvement automatique et/ou relatif).
- Exécuter plusieurs tâches, mais seulement une ou quelques-unes d'entre elles dans chaque cycle de jeu, en utilisant **l'arithmétique modulaire (ou les opérations binaires) en cascade**.
- Introduire **dans le jeu des limitations qui ne sont pas importantes** ou qui n'affectent pas le gameplay, afin de réduire le nombre de tâches qui sont exécutées dans chaque cycle de jeu ou de simplifier les tâches afin qu'elles soient exécutées plus rapidement.
- En règle générale, réduisez le nombre d'instructions que votre programme exécute à chaque cycle de jeu, en remplaçant parfois les algorithmes par des précalculs ou en insérant davantage d'instructions de manière à ce que (paradoxalement) moins d'instructions soient exécutées à chaque cycle.

Ces idées ont le même objectif : **exécuter moins de logique à chaque cycle**, ce qui permet à tous les sprites de se déplacer en même temps, tout en prenant moins de décisions à chaque cycle du jeu. C'est ce qu'on appelle "**réduire la complexité informatique**", en transformant un problème d'ordre N (N sprites) en un problème d'ordre 1 (une seule logique à exécuter à chaque image).

La clé est de déterminer quelle logique ou quelles logiques doivent être exécutées à chaque cycle. Dans le cas le plus simple, si nous avons N sprites, nous exécuterons simplement l'une des N logiques. Mais dans les cas plus complexes, nous devrons faire preuve d'astuce pour déterminer les logiques à exécuter.

11.3.1 Déplace 32 sprites avec des logiques massives

Voyons maintenant un exemple simple de déplacement simultané et fluide de 32 sprites (à 14 images par seconde). C'est parfaitement possible. Un seul fantôme prendra des décisions à chaque cycle, mais tous les fantômes se déplaceront dans tous les cycles. Nous pouvons également les animer tous (en les associant à une séquence d'animation et en utilisant la fonction

|PRINTSPALL,1,0) et il sera toujours lisse, mais il donnera l'impression qu'il y a plus de mouvement, car le battement des ailes d'une mouche (par exemple) génère une grande sensation de mouvement.

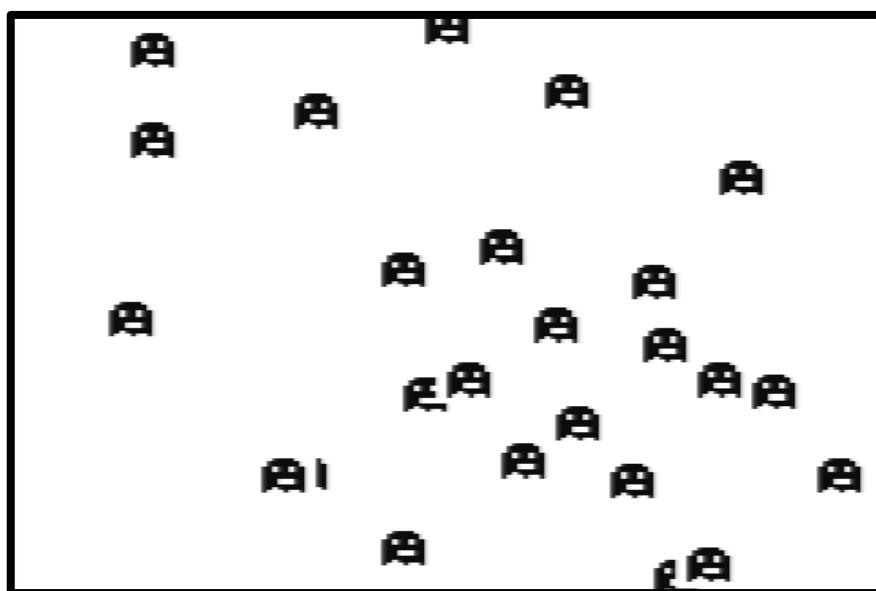


Fig. 65 Avec les logiques massives, vous pouvez déplacer 32 sprites simultanément.

Ce que nous avons fait, c'est réduire la complexité informatique. Nous avons commencé par un problème d'"ordre N", où N est le nombre de sprites. En supposant que la logique de chaque sprite nécessite 3 instructions BASIC, en principe, $N \times 3$ instructions devraient être exécutées à chaque cycle. Avec la technique de "bulk logic", nous transformons le problème d'"ordre N" en un problème d'"ordre 1". Un problème "d'ordre 1" est un problème qui implique un nombre constant d'opérations quelle que soit la taille du problème. Dans ce cas, nous sommes passés de $N \times 3 = 32 \times 3 = 96$ opérations BASIC à seulement 3 opérations BASIC. Cette réduction de la complexité est la clé de la haute performance de la technique de la logique en bloc.

```

1 MODE 0
10 MÉMOIRE 24999 : APPEL &6B78
20 DEFINT a-z
25 ' Réinitialisation des ennemis
30 FOR j=0 TO 31:|SETUPSP,j,0,&X0:NEXT
35 ' num ennemis 12 x 16 (6bytes de large x 16 lignes)
36 num=32 : x%=0:y%=0
40 FOR i=0 TO num-1:|SETUPSP,i,9,&8ee2 : |SETUPSP,i,0,&X1111 :
41 |LOCATESP,i,rnd*200,rnd*80
42 suivant

```

```

43 i=0
45 gosub 100
46 i=i+1 : si i=num alors
i=0
50 |PRINTSPALL,0,0
60 |AUTOALL
70 goto 45
100 |peek,27001+i*16,@y%
110 |peek,27003+i*16,@x%
120 if y%<=0 then |SETUPSP,i,5,2:|SETUPSP,i,6,0 : return
130 if y%>=190 then |SETUPSP,i,5,-2:|SETUPSP,i,6,0 : return
140 if x%<=0 then |SETUPSP,i,5,0:|SETUPSP,i,6,1 : return
150 if x%>=76 then |SETUPSP,i,5,0:|SETUPSP,i,6,-1 : return

160 chance=rnd*3
170 if random=0 then |SETUPSP,i,5,2 :
|SETUPSP,i,6,0:return
180 if random=1 then |SETUPSP,i,5,-
2:|SETUPSP,i,6,0:return
190 if random=2 then
|SETUPSP,i,5,0:|SETUPSP,i,6,1:return
200 if random=3 then |SETUPSP,i,5,0:|SETUPSP,i,6,1:
1:return

```

~~4.3.2 Execution en cascade, alternée et périodique~~

Il n'est pas nécessaire d'effectuer toutes les tâches à chaque cycle de jeu. Par exemple, si vous voulez vérifier si le tir a quitté l'écran, vous pouvez le faire tous les deux ou trois cycles, au lieu de le faire à chaque cycle.

Vous pouvez également faire en sorte que les ennemis tirent tous les quelques cycles et non pas tous les cycles (sinon, ils vous tireraient beaucoup dessus ! !).

En bref, il y a des choses que vous n'avez pas besoin de faire à chaque cycle et vous pouvez économiser l'exécution d'instructions par cycle, ce qui vous permettra d'atteindre une vitesse plus élevée. C'est en fait le fondement de la technique de "bulk logic".

Il existe deux techniques de base pour y parvenir : l'utilisation de l'arithmétique modulaire et des opérations binaires **ET**. Les opérations **ET** binaires sont plus rapides que les opérations **MOD**.

Technique	Temps consommé
A = A+1 : si A=5 alors A=0 : GOSUB <routine>.	2,6 ms
SI cycle MOD 5 =0 ALORS gosub routine	1,84ms, en supposant que vous ayez déjà une variable appelée cycle qui est mise à jour

L'opération MOD est quelque peu coûteuse et, par conséquent, une opération binaire est parfois préférable.

En supposant que la variable cycle soit mise à jour à chaque fois, nous pouvons effectuer une opération binaire pour voir quand un groupe de bits donne une certaine valeur. Par exemple, si nous regardons les 4 bits les moins significatifs de la variable cycle, ils iront toujours de 0000 à 1111 et vice-versa. Si nous faisons un AND 15 avec cette variable, nous pouvons faire la même chose qu'avec MOD 15. Le nombre 15 en

binaire est 1111 et donc un AND révèle la valeur de ces 4 bits.

Technique	Temps consommé
Si cycle ET 15=0 alors gosub routine	1,6 ms (exécute une fois sur 16)
Si cycle AND 1=0 alors gosub routine	1,6 ms (une fois toutes les deux fois)

Si vous devez exécuter plusieurs opérations périodiques, vous pouvez procéder de la manière suivante :

c=cycle AND 15 : rem 15 est en binaire 1111
IF c=0 ALORS GOSUB <routine1> (la routine1 est exécutée une fois toutes les 16 fois) IF c=8 ALORS GOSUB <routine2>... (la routine2 est exécutée une fois toutes les 16 fois, mais loin dans le temps de l'exécution de la routine1)

De cette manière, vous répartissez le temps sur différentes tâches, de sorte qu'à chaque cycle, vous n'effectuez qu'une seule tâche, mais qu'après plusieurs cycles, vous avez accompli toutes les tâches.

Pour vérifier la variable de cycle et décider d'exécuter une tâche, il existe une meilleure façon d'exécuter les opérations binaires, qui est la suivante :

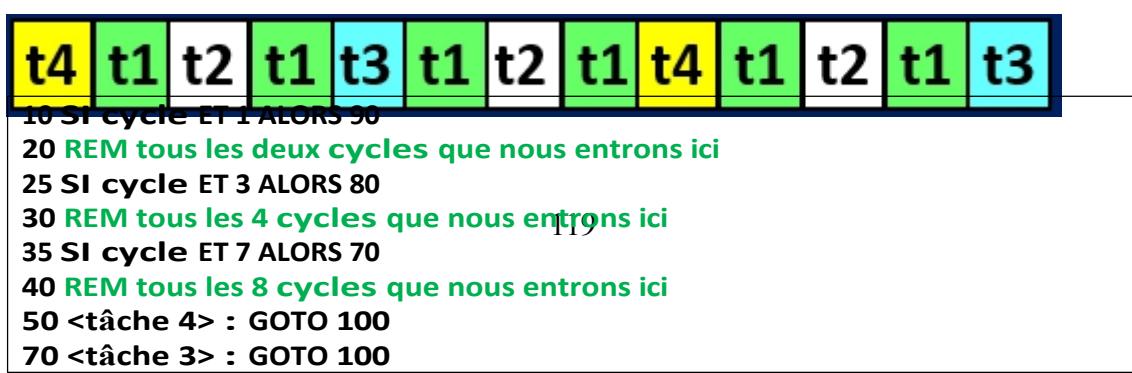
Technique	Temps consommé
10 Si cycle et 7 alors 30	1,18 ms. Il s'agit sans aucun doute de la meilleure stratégie pour l'exécution de tâches périodiques.
20 <instructions qui sont exécutées tous les 8 cycles>	
30 <poursuite du programme>	

L'application de la même stratégie à MOD augmente également la vitesse, bien que moins qu'avec AND. Cependant, elle est très bonne car elle fonctionne même si la période n'est pas un multiple de 2 (on peut mettre MOD 7, MOD 5, MOD 10, etc.).

Technique	Temps consommé
10 Si cycle MOD 8 puis 30	1,29 ms. Presque aussi bien que l'ET et la meilleure stratégie lorsque nous avons besoin d'une période qui n'est pas un multiple de 2.
20 <instructions qui sont exécutées tous les 8 cycles>	
30 <poursuite du programme>	

Comme vous pouvez le constater, pour chaque tâche que nous voulons introduire dans la logique, nous devons introduire un "IF". Cependant, il est encore possible d'améliorer la logique et de la rendre plus efficace en utilisant des intervalles de temps d'exécution des tâches qui sont des multiples. Si ces intervalles sont multiples, le "FI" de la tâche 2 peut être exécuté dans le cadre de la tâche 1, et le "FI" de la tâche 3 dans le cadre de la tâche 2, en "cascade". Cela réduit considérablement le nombre de FI que nous exécutons à chaque cycle, puisqu'il s'agit dans de nombreux cas d'un seul FI.

Examinons un exemple complet, qui permet de réaliser 4 tâches différentes en multitâche, tout en réduisant le nombre de tâches exécutées en même temps. La séquence suivante représente l'ordre d'exécution des tâches puis du code source.




```

80 <tâche 2> : GOTO 100
90 <tâche 1>.
100 REM --- fin des tâches ---

```

Dans cet exemple, nous avons choisi les intervalles 2, 4 et 8.
 ET 1 : cela me donne un intervalle de 2 car il est nul tous les 2 cycles
 ET 3 : il est nul tous les 4 cycles
 AND 7 : est égal à zéro tous les 8 cycles

Comme nous avons choisi des opérations à intervalles multiplo, les FI sont exécutés "en cascade" : nous n'entrions dans un FI que si nous sommes entrés dans le précédent :

- La moitié des cycles exécutent un seul FI (ligne 10).
- La moitié des cycles exécute deux instructions IF (lignes 10 et 25), dont l'autre moitié (c'est-à-dire 25 %) exécute trois instructions IF (lignes 10, 25 et 35).

En moyenne, $1*50\%+2*25\%+3*25\% = 1,75$ instructions IF sont exécutées par cycle.

Grâce à cette stratégie d'**utilisation de l'arithmétique modulaire avec des opérations binaires dans des intervalles multiples pour les mettre en cascade**, nous pouvons réduire le nombre d'opérations "IF" à un minimum et en même temps réduire la complexité de calcul de l'ordre N (n tâches) à l'ordre 1 (une tâche par cycle). Cela accélère considérablement vos jeux.

11.3.3 Exemple simple de logique de masse

Dans le jeu vidéo "Mutante Montoya", les sprites ennemis courrent à tour de rôle à travers les différents cycles du jeu. **Lorsque j'ai programmé ce jeu, je n'avais pas encore programmé le mécanisme |ROUTEALL qui permet d'assigner des trajectoires fixes aux sprites, mais j'ai pu le résoudre avec des logiques massives.** Dans le cas où vous voudriez faire un jeu où les ennemis ont une "intelligence", une trajectoire fixe ne fonctionnerait pas, donc même si vous avez la commande ROUTEALL, vous devriez faire tourner la logique du sprite comme décrit ci-dessous, donc cet exemple est intéressant.

Supposons que nous ayons trois soldats ennemis se déplaçant de droite à gauche et de gauche à droite. Pour gagner en rapidité, nous n'exécuterons la logique que d'un seul soldat par cycle de jeu.

Afin que la coordonnée x de chaque soldat continue d'avancer malgré tout, nous utiliserons l'indicateur de mouvement automatique au lieu de le mettre à jour nous-mêmes.

```

10 MÉMOIRE 24999
20 MODE O : DEFINT A-Z : CALL &6B78:' install RSX
25 FOR j=0 TO 31:|SETUPSP,j,0,&X0:NEXT:'réinitialiser les sprites
26 |SETLIMITS,0,80,0,0,200
30 "paramétrage de 3 soldats
40 dim x(3):x%=0
50 x(1)=10:xmin(1)=10:xmax(1)=60:
y(1)=60:dirección(1)=0:|SETUPSP,1,7,9 :|SETUPSP,1,0,&x1111:
|SETUPSP,1,5,0 :|SETUPSP,1,6,1
60 x(2)=20:xmin(1)=15:xmax(2)=40:
y(2)=100:dirección(2)=1:|SETUPSP,2,7,10 :|SETUPSP,2,0,&x1111:
|SETUPSP,2,5,0 :|SETUPSP,2,6,-1

```

```

70 x(3)=30:xmin(1)=5:xmax(3)=50:
y(3)=130:direccion(3)=0:|SETUPSP,3,7,9 :|SETUPSP,3,0,&x1111:
|SETUPSP,3,5,0 :|SETUPSP,3,6,1
80 for i=1 to 3:|LOCATESP,i,y(i),x(i):next : 'nous plaçons les sprites
81 i=0
89 ----- BOUCLE PRINCIPALE DU JEU (CYCLE DU JEU) -----
90 i=i+1:gosub 100
92 si i=3 alors i=0
93 |AUTOALL
94 |PRINTSPALL,1,0 : ' anime et imprime les 3 soldats
95 goto 90
96 ----- FIN DU CYCLE DE JEU -----
99 '----- routine soldat -----
100 |PEEK,27003+i*16,@x% : x(i)=x%
101 SI adresse(i)=0 ALORS SI x(i)>=xmax(i) ALORS
adresse(i)=1:|SETUPSP,i,7,10 :|SETUPSP,i,6,-1 ELSE return
110 SI x(i)<=xmin(i) ALORS adresse(i)=0:|SETUPSP,i,7,9 :
|SETUPSP,i,6,1
120 retour

```

Chaque soldat a sa propre logique, mais nous n'en exécutons qu'un seul par cycle de jeu, ce qui allège considérablement ce dernier.

La seule limite est qu'en exécutant la logique de chaque soldat une fois sur trois, la coordonnée pourrait dépasser la limite que nous avons fixée pour deux cycles. Cela nous oblige à être plus prudents lorsque nous fixons la limite, en nous assurant que lorsque nous l'exécutons, il n'envahit et n'efface jamais un mur de notre labyrinthe d'écran, par exemple. Je vais essayer d'expliquer ce problème plus précisément :

Supposons que nous ayons 8 sprites et que notre sprite se déplace à chaque cycle, mais que nous n'exécutons sa logique qu'une fois sur 8. Imaginons un sprite qui se trouve à la position x=20 et que nous voulons qu'il se déplace jusqu'à la position x=30 et qu'il fasse demi-tour. Considérons que le sprite a un mouvement automatique avec Vx=1. Dans ce cas, nous vérifierons sa position à x=20, x=28, x=36. Lorsque nous atteindrons 36, nous nous rendrons compte que nous sommes allés trop loin !!! et nous changerons la vitesse du sprite en Vx=-1.

Comme vous pouvez le constater, le contrôle des limites de la trajectoire n'est pas précis, à moins que nous ne tenions compte de cette circonstance et que nous fixions la limite à un niveau que nous pouvons contrôler, ce qui sera Xfinal = Xinitial + n*8.

Cette limitation est minuscule comparée à l'avantage de déplacer de nombreux sprites à grande vitesse. Avec un peu d'astuce, nous pouvons même exécuter la logique moins de fois, de sorte que la logique de l'ennemi ne soit exécutée que toutes les deux secondes.

11.3.4 Bloquer" le mouvement des escadrons

Si vous souhaitez simplement déplacer un escadron dans une direction à la fois, l'une des deux fonctions suivantes de la bibliothèque 8BP fonctionnera :

- Si vous utilisez la commande **AUTOALL**, vous devez faire passer les sprites en vitesse automatique dans la direction souhaitée (en Vx, en Vy ou les deux) et bien sûr activer le bit 4 de l'octet de statut. La commande AUTOALL possède un paramètre optionnel pour invoquer en interne |ROUTEALL avant de déplacer

les sprites.

- Si vous utilisez **|MOVERALL**, vous devez attribuer le bit 5 de l'octet de statut aux sprites que vous allez déplacer. Cette commande requiert comme paramètres l'ampleur du mouvement relatif en Y et en X que vous souhaitez.

De cette façon, avec une seule instruction, vous déplacez plusieurs sprites en même temps. Si chacun de vos sprites doit se déplacer indépendamment et avec une logique indépendante, comme c'est le cas dans des jeux comme "pacman", vous devrez être plus astucieux, comme je vous le dirai plus loin.

11.3.5 Technique de logique massive dans les jeux de type "pacman"

Si vous avez beaucoup d'ennemis et qu'ils doivent prendre des décisions à chaque fourche d'un labyrinthe, ce n'est pas une bonne stratégie que d'exécuter la logique de l'ennemi à tour de rôle à chaque cycle. Il est préférable d'exécuter la logique lorsque celle-ci doit prendre une décision. Dans les jeux de type Pacman, cela se produit lorsqu'un fantôme atteint une fourche où il peut prendre une nouvelle direction de mouvement en fonction de son intelligence. Cela peut se faire grâce à une simple "astuce". Il s'agit simplement de placer des ennemis dans des positions bien choisies au début du jeu.

Supposons que vous ayez 4 ennemis et que les bifurcations du labyrinthe se produisent par multiples de 4. Si le premier ennemi se trouve dans une position multiple de 4, c'est à son tour d'exécuter sa logique. Le deuxième ennemi doit exécuter sa logique de décision au cours du cycle suivant. S'il ne se trouve pas dans une position de bifurcation du labyrinthe, il ne peut pas changer de cap.

Afin de faire correspondre sa position à un multiple de 4 et de pouvoir ainsi décider du chemin à prendre à la bifurcation, nous commençons simplement le jeu avec ce deuxième ennemi placé à un multiple de 4 moins un. En considérant des coordonnées qui commencent à zéro, les multiples de 4 sont :

Premier ennemi : position 0 ou 4 ou 8 ou 12 ou 16 ou 20 ou 24 ou XX (sur l'axe x ou y, peu importe)
 Deuxième ennemi : position 1 ou 5 ou 9 ...

Troisième ennemi : position 2 ou 3 ou 10 ...

Et ainsi de suite. Vous placez vos ennemis selon cette règle :

Position = multiple de 4 - n, où n est le numéro du sprite

Et chaque fois que c'est au tour d'un ennemi d'exécuter sa logique, il peut se trouver à une bifurcation. Si un fantôme doit prendre une décision à l'instant "i", il prendra une décision à $t=i+4$, et la décision suivante sera à $t=i+4+4$, et ainsi de suite.

Voyons un exemple graphique dans lequel j'ai mis en évidence avec un rectangle que l'ennemi va prendre une décision parce qu'il se trouve dans une fourche. Comme vous pouvez le constater, une seule logique de bifurcation est exécutée par cycle de jeu.

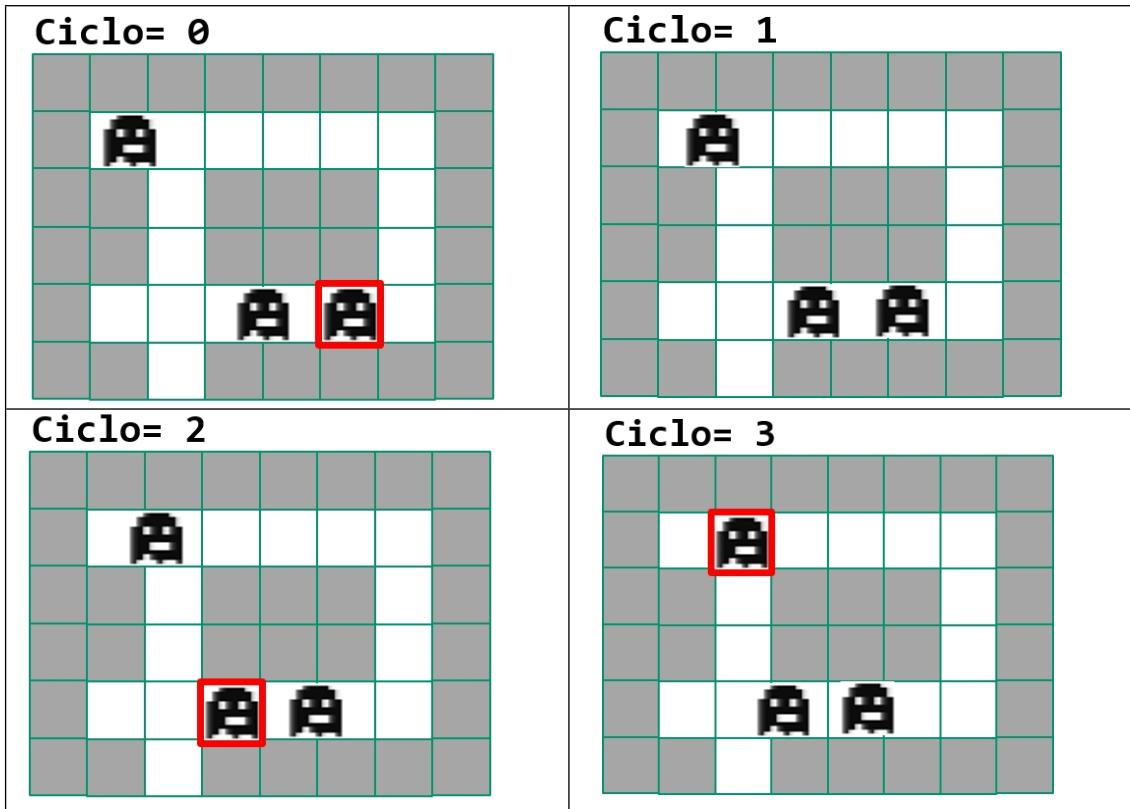


Fig. 66 Logiques massives dans les jeux de type PACMAN

Lorsque c'est à votre tour d'exécuter votre logique, vous devez vérifier que vous ne vous trouvez pas dans une position au milieu d'un couloir sans embranchements. Pour ce faire, vous devez utiliser la commande **|COLAY**.

Le jeu vidéo "Paco, the man" utilise cette technique. En effet, il exécute la logique de 4 fantômes, chacun dans un cycle de jeu différent, tout en répartissant le reste des tâches entre les différents cycles. C'est un magnifique exemple de logique massive, je vous recommande donc de lire le "making of" du jeu vidéo, que vous trouverez dans la documentation du 8BP.

	cycles			
Tâche	0	1	2	3
Lecture du clavier	X			
Détection collision avec fantômes y avec sprites invisible		X		
Détection collision avec la mise en page y relocalisation si Paco s'écrase			X	
Détection de points (noix de coco)				X
Logique fantôme 1	X			
Phantom Logic 2		X		
Phantom Logic 3			X	

Le jeu vidéo "**Paco the man**" répartit les tâches en 4 cycles de jeu, ce qui permet d'atteindre près de 20 FPS en BASIC.



Phantom Logic 4				X	
------------------------	--	--	--	---	--

11.3.6 Réduire le nombre d'instructions du cycle de jeu Réduire le nombre d'instructions du cycle de jeu est essentiel pour accélérer votre programme. Parfois, cela signifie que le programme aura plus de lignes, même si en et moins de lignes sont exécutées à chaque cycle. D'autres fois, vous pouvez introduire des limitations "indétectables" qui accélèrent votre jeu sans que le joueur ne remarque la différence. Voici quelques exemples

11.3.6.1 Gestion du clavier avec moins d'instructions

Gérez le clavier (et en général cela s'applique à tout ce que vous faites) en exécutant le moins d'instructions possible. Voici un exemple (d'abord mal fait, puis bien fait), où l'on passe par au plus 4 opérations INKEYS avec leurs IFs correspondants. Exécutez-le mentalement et vous verrez ce que je veux dire. Le deuxième exemple est beaucoup plus rapide

Mauvais exemple (pire des cas = 8 exécutions "IF INKEY") :

```
1000 rem routine clavier inefficace
1010 SI INKEY(27)=0 et INKEY(67)=0 ALORS <instructions>:RETURN
1020 SI INKEY(27)=0 et INKEY(69)=0 ALORS <instructions>:RETURN
1030 SI INKEY(34)=0 et INKEY(67)=0 ALORS <instructions>:RETURN
1040 SI INKEY(34)=0 et INKEY(69)=0 ALORS <instructions>:RETURN
1050 IF INKEY(27)=0 THEN <instructions>:RETURN
1060 IF INKEY(34)=0 THEN <instructions>:RETURN
1070 IF INKEY(67)=0 THEN <instructions>:RETURN
1080 IF INKEY(69)=0 THEN <instructions>:RETURN
1080 IF INKEY(69)=0 THEN <instructions>:RETURN
```

Voici l'exemple d'une lecture de frappe, mais bien faite (avec dans le pire des cas = 4 exécutions de "IF INKEY"). Il a également été tenu compte du fait que, dans un IF, les expressions non nulles sont VRAIES. Les instructions à exécuter dans votre jeu peuvent être différentes, mais le schéma de lecture du clavier devrait être le même en cas de gestion des diagonales (vers le haut et vers la droite en même temps, par exemple). Cela peut sembler plus long, mais c'est beaucoup plus rapide que l'exemple précédent.

```
REM routine clavier efficace 'saut à
1550 si "P" n'a pas été appuyé 1510 si
inkey(27) ALORS 1550 : 'touche P 1520 si
inkey(67) ALORS 1530 : 'touche Q

1525 <instructions en cas d'appui simultané sur les touches "P" et "Q
time>:RETURN
1530 if inkey(69) THEN 1540 : Clé "A

1535 <instructions en cas d'appui simultané sur les touches "P" et "A
time>:RETURN

1540 <instructions au cas où vous auriez appuyé sur "P"
uniquement>:RETURN 1550 if inkey(34) THEN 1590 : Touche "O
```

1560 if INKEY(67) THEN 1570 : Touche "Q

1565 <instructions en cas d'appui simultané sur "O" et "Q time>:RETURN

1570 if INKEY(69) THEN 1580 : 'clé A

1575 <instructions si vous avez appuyé sur "O" et "A" en même temps> : RETOUR

1580 <instructions si vous avez appuyé sur "O" uniquement>:RETURN

1590 IF INKEY(67) THEN 1600 : Touche "Q

1595 <instructions si "Q"> a été appuyé : RETURN

1600 IF INKEY(69) THEN return : 'A' key

1610 <instructions si "A" a été appuyé seulement> : RETOUR

Une autre chose que vous devriez faire pour accélérer votre jeu est d'utiliser une tâche périodique pour analyser les touches "secondaires" telles que les touches pour activer/désactiver la musique, les touches pour passer à un menu ou afficher quelque chose de spécial, etc. Il s'agit de touches que vous pouvez analyser périodiquement et non à chaque cycle. Ce sont des touches que vous pouvez analyser périodiquement et non à chaque cycle. Cependant, vous devez tenir compte du coût de l'analyse, qui n'est pas très élevé (1 ms).

10 if inkey(47) then 30 : ' ceci coûte 1,0 ms

20 <instructions si vous appuyez sur la touche 47>.

30 rem vous arrivez ici si vous n'avez pas cliqué dessus

L'analyse d'une variable avec AND pour une tâche à effectuer tous les (par exemple) 4 cycles coûte 1,18 ms, ce qui nous coûtera donc

1,18 ms x 4 cycles (évaluation du cycle) + 1,0 ms (**touche**) = 5,72 ms

Si, au lieu de cela, nous avions exécuté l'**inkey** à chaque cycle, nous n'aurions passé que 4 ms. Par conséquent, le balayage de certaines touches en fonction du cycle de jeu n'a de sens que si nous évitons au moins de balayer deux touches dans certains cycles.

Supposons le programme suivant :

10 if cycle and 3 then 50 : ' il coûte 1,18 ms

20 if inkey(47) ... ' ceci coûte 1 ms

30 if inkey(35) ...' this costs 1 ms

50 <plus d'instructions>.

En l'état, le programme évalue les touches 47 et 35 tous les 4 cycles. Lorsqu'il les évalue, il passe $1,18 + 1 + 1 + 1 = 3,8$ ms, tandis que lorsqu'il ne les évalue pas, il passe 1,18 ms. Par conséquent, le temps passé en 4 cycles est de

Temps $3 * 1,18 + 3,8 = 6,72$ ms

En revanche, si nous avions évalué les touches à chaque cycle, nous aurions passé $4 * 2$ ms= 8 ms. Il y a donc une économie de $8 - 6,7 = 1,3$ ms pour 4 cycles, soit environ 0,3 ms par cycle de jeu.

Selon le type de jeu, il est possible de scanner certaines touches sur les cycles pairs et

les autres sur les cycles impairs. Par exemple, dans un jeu qui utilise des touches QAOP, vous pouvez scanner QA sur les cycles pairs et OP sur les cycles impairs, ou vice versa.

De cette façon, vous pouvez obtenir plus de vitesse avec une limitation que le joueur ne remarquera probablement pas. C'est le genre de limitation qu'il vaut parfois la peine d'introduire, mais cela dépend du jeu.

11.3.6.2 Éviter de passer par des IF inutiles

Nous examinerons deux façons de procéder :

```
10 SI A=1 ALORS <instructions lorsque A=1> 10 SI A=1 ALORS <
instructions lorsque A=1> 10 SI A=1 ALORS <instructions lorsque A=1>
20 SI A=2 ALORS <instructions lorsque A=2> 20 SI A=2 ALORS <
instructions lorsque A=2> 20 SI A=2 ALORS <instructions lorsque A=2>
30 SI A=3 ALORS <instructions lorsque A=3>.
40 <plus d'instructions>
```

Si vous pouvez éviter de passer par un IF en insérant un GOTO, c'est toujours préférable. Le GOTO est un grand allié de la technique de logique massive.

```
10 IF A=1 THEN <instructions lorsque A=1> : GOTO 40
20 IF A=2 THEN <instructions lorsque A=2> : GOTO 40
30 <instructions lorsque A=3> : rem si A n'est ni 1 ni 2 alors c'est 3
40 <plus instructions>
```

Une autre façon de procéder consiste à utiliser l'instruction ON <variable> GOTO ou l'instruction ON <variable> GOSUB.

Comme je l'ai présenté dans le tableau de 11.1, vous pouvez gagner plus de 1 ms en utilisant ON GOTO.

```
10 sur A GOTO 30,40,50
20 goto 60 : rem nous arrivons ici si A=4
30 rem nous arrivons ici si A=1
35 goto 60
40 rem nous arrivons ici si A=2
45 goto 60
50 rem nous arrivons ici si A=3
60 rem ici la logique continue
```

11.3.6.3 Remplace les algorithmes par des pré-calculs

Pensez à une balle qui rebondit. Au lieu d'utiliser les équations du mouvement accéléré, construisez une trajectoire qui déplace un sprite vers le bas avec un incrément de coordonnée Y plus important à chaque étape, puis vers le haut avec un incrément de plus en plus petit au fur et à mesure qu'il touche le sol. Il n'y a pas d'équations complexes à exécuter et pourtant l'effet visuel est le même. C'est ainsi que j'ai réalisé les sauts dans le jeu "**Fresh fruits & vegetables**". Il est possible de programmer de cette manière parce que **dans le jeu, l'univers est "déterministe"**. C'est-à-dire que l'on peut prédire à chaque instant ce qui va se passer, quelle que soit la complexité des équations qui régissent le saut d'un personnage ou le mouvement d'une escouade.

Dans les jeux où la logique de l'ennemi nécessite l'utilisation d'une fonction de calcul (comme le cosinus), précalculez tout et stockez-le dans un tableau que vous utiliserez pendant l'exécution de la logique. Le calcul pendant l'exécution de la logique du jeu est

trop coûteux.

La logique complexe est une logique lente. Si vous voulez faire quelque chose de compliqué, une trajectoire complexe, un mécanisme d'intelligence artificielle... ne le faites pas, essayez de le "simuler" avec un modèle comportemental plus simple qui produit le même effet visuel. Par exemple, un fantôme intelligent qui vous poursuit, au lieu de lui faire prendre des décisions intelligentes, faites-lui prendre la même direction que votre personnage, sans aucune logique. Si vous ne pouvez pas simplifier de cette manière, pensez que même l'intelligence artificielle peut devenir "déterministe". Si un ennemi prend une décision sur la façon de se déplacer en fonction de votre position et de votre vitesse, nous pourrions stocker le résultat de cet algorithme lourd dans un tableau et éviter tous les calculs.

```
10 Rem Vx, Vy, X, Y sont la vitesse et la position de mon personnage.
20 rem supposons que j'ai précalculé des décisions pour 3 vitesses, 10 emplacements pour les coordonnées X et 10 emplacements pour les coordonnées Y. Cela représente moins de 1 Ko
30 DIM poursuivre(3,3,10,100)
40 rem ' charger les valeurs dans le tableau après les avoir lentement calculées
50 newaddress=pursue(Vx,Vy,x,y) : le mécanisme rem en action
```

L'univers que vous programmez est "déterministe". Quelle que soit la complexité du comportement des ennemis et des éléments de l'écran, si leur comportement ne dépend pas de votre interaction, alors il existe une position pour chaque chose donnée à chaque instant donné. Une position qui pourrait être précalculée pour éviter tout algorithme comportemental complexe et le résultat serait le même.

11.3.6.4 Ne pas exécuter les commentaires

Supprimez tous les commentaires dans la logique du jeu et si vous en laissez qui sont REM (plus rapides), n'utilisez pas les guillemets. Si vous utilisez les guillemets, c'est pour économiser 2 octets de mémoire, et ils conviennent pour commenter le reste du programme (initialisations et autres). Si vous voulez commenter des parties de la logique, vous pouvez faire ce qui suit :

```
Si x>23 gosub 500
...
499 rem n'est pas transmis sur cette ligne et je commente donc cette routine.
500 si x > 50 ALORS ...
...
550 RETOUR
```

Chaque commentaire que vous exécutez consomme 0,20 ms et il est très facile de sauvegarder son exécution sans laisser de commentaires. Il y a des moments où l'on peut mettre des commentaires sur des lignes à condition qu'il y ait des sauts (GOTO et GOSUB/RETURN) sans craindre de perdre du temps, voyons quelques exemples :

```
10 goto 50 : rem ce commentaire n'est pas chronophage
20 gosub 200 : rem ce commentaire ne prend pas de temps
200 return : rem ce commentaire n'est pas chronophage
```

11.3.6.5 Un seul sprite meurt par cycle

La commande 8BP |COLSPALL est conçue dans une optique de "logique massive". Cela signifie qu'elle détecte potentiellement la collision de tous les sprites, mais dès qu'elle détecte une collision, elle revient en indiquant quel sprite est celui qui est entré en

collision et quel sprite est celui qui est entré en collision. À ce moment-là, vous pouvez ignorer le sprite qui entre en collision (en désactivant sa capacité à entrer en collision dans le bit 1 de son octet de statut) et réexécuter la commande, jusqu'à ce qu'il n'y ait plus de collisions (elle renvoie un 32 sur le collisionneur et le collisionné). Cependant,

Il est plus efficace de ne pas redéclencher la commande avant le prochain cycle de jeu. Cela signifie qu'un seul ennemi peut mourir par image, mais il s'agit d'une limitation imperceptible qui accélérera considérablement vos parties.

11.3.7 Routes qui accélèrent le jeu en manipulant l'état

Vous pouvez créer des chemins qui activent et désactivent alternativement le collider ou le drapeau collidable dans vos sprites. Selon le type d'ennemi, cela peut vous donner une vitesse supplémentaire dans la commande de collision. Les chemins sont expliqués dans un chapitre ultérieur, il est donc préférable de se familiariser avec eux avant de lire ce chapitre.

Par exemple, si vous avez 8 ennemis à l'écran, vous pouvez créer 4 ennemis collidables dans les cycles pairs et 4 ennemis collidables dans les cycles impairs :

```
ROUTE1 ;  
-----  
db 1,0,1  
db 255,128+8+2+1,0 ; changement d'état à colliderable  
db 1,0,1  
db 255,128+8+1,0 ; changement d'état vers db 0 non  
collisionnable
```

Ce chemin modifie l'état de votre sprite, tout en le déplaçant vers la droite. Si ce chemin est assigné à un sprite, celui-ci se déplacera vers la droite et sera alternativement collisionnable et non collisionnable.

Vous pouvez assigner le chemin à 8 sprites et exécuter la commande suivante sur 4 des sprites :

```
|ROUTESP, <sprite_id>, 1
```

Ainsi, la commande de collision |COLSPALL sera plus rapide car elle n'aura à détecter que les collisions avec 4 ennemis par image. Cette astuce accélère un peu votre jeu et, avec d'autres astuces, vous pouvez enfin atteindre le FPS dont vous avez besoin.

La même astuce vous permet d'activer et de désactiver le drapeau d'impression (bit 0 du statut) et dans une trajectoire où l'ennemi passe quelques frames sans bouger, vous pouvez désactiver le drapeau d'impression à travers la trajectoire et ainsi accélérer la commande PRINTSPALL.

11.3.8 Routage des sprites avec des "logiques massives".

Pour déplacer des sprites à travers des chemins, il existe une commande appelée |ROUTEALL qui le fait très efficacement, mais **en tant qu'exercice pour comprendre la philosophie de la logique de masse, il est très intéressant d'étudier ce cas difficile mais emblématique de la logique de masse**. Pour programmer le jeu vidéo "Anunnaki", j'ai utilisé la technique que je vais décrire ci-dessous, car je n'avais pas encore programmé la capacité de routage des sprites dans 8BP. En gros, j'ai utilisé des logiques massives dans le routage des vaisseaux ennemis.

Les navires passeront un par un par une série de "nœuds de contrôle", qui sont des endroits dans l'espace où ils doivent changer leur direction, définie par leurs vitesses en X et Y, c'est-à-dire (Vx,Vy).

Une façon de contrôler que les 8 navires changent de direction à ces endroits serait de comparer leurs coordonnées X,Y avec celles de chacun des nœuds de contrôle et si elles coïncident avec l'un d'entre eux, alors nous appliquons les nouvelles vitesses

associées aux nœuds de contrôle.

changement dans ce nœud. Puisque nous parlons de 2 coordonnées, de 8 navires et de 4 nœuds, nous cherchons :

2 x 8 x 4 = 64 contrôles sur chaque cadre

Cela n'est pas possible si nous voulons que BASIC soit rapide, car ce n'est pas une stratégie efficace du point de vue du calcul. Puisque nous avons affaire à un scénario "déterministe", nous pouvons être sûrs à chaque instant de l'emplacement de chacun des navires et, par conséquent, au lieu de vérifier dans l'espace, nous pouvons **nous concentrer uniquement sur la coordonnée temporelle** (qui est le numéro de l'image de jeu ou le numéro de ce que l'on appelle le "cycle de jeu"). Ne considérez pas le temps en secondes, mais en images.

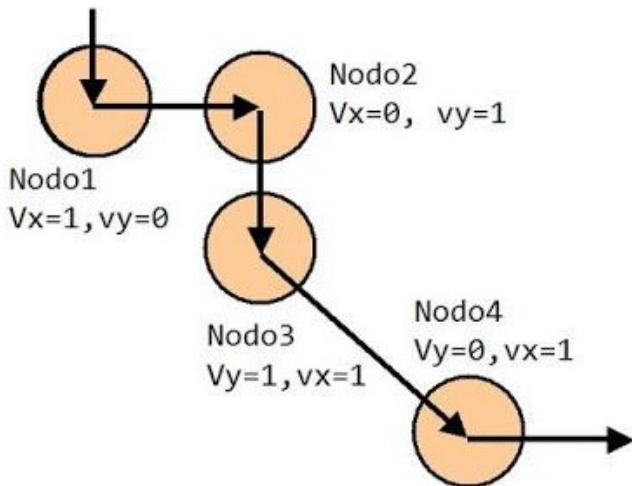


Fig. 67 Trajectoire définie avec des "nœuds de contrôle".

Puisque nous connaissons la vitesse à laquelle les navires se déplacent, nous pouvons savoir quand le premier navire passera le premier nœud. Nous appellerons cet instant $t(1)$. Nous supposerons également qu'en raison de la séparation entre les navires, le deuxième navire passera le nœud à l'instant $t(1)+10$. Le troisième à l'instant $t(1)+20$ et le huitième à l'instant $t(1)+70$. Au lieu d'utiliser des trames comme unités de temps, utilisons des dizaines de trames : dans ce cas, les instants seront $t(1)$, $(1)+1$, $t(1)+2$, etc.

Sachant cela, nous pouvons contrôler le temps avec deux variables : l'une comptera les dizaines (i) et l'autre les unités (j). Pour contrôler le changement des 8 navires dans le premier nœud, nous pouvons écrire :

```
j=j+1 : SI j=10 ALORS j=0 : i=i+1 : SI i>=t(1) ET i<=t(1)+8 ALORS
[met à jour la vitesse du navire     i-t(1) avec les valeurs de vitesse
du nœud 1].
```

Comme nous pouvons le voir, avec une seule ligne, nous pouvons changer les vitesses de chaque vaisseau lorsqu'ils passent par le nœud 1. Chaque fois que "j" devient zéro, nous augmentons la variable "i" et mettons à jour l'un des vaisseaux. Pendant les 80 premiers instants du temps (8 en dizaines d'images), chacun des 8 vaisseaux est mis à jour, juste au moment où il passe par le nœud de contrôle, c'est-à-dire qu'à l'instant $t(1)$, le Sprite 0 est mis à jour, à $t(1)+1$ le Sprite 1 est mis à jour, à $t(1)+2$ le Sprite 2 est mis à jour, et ainsi de suite.

Le numéro du Sprite qui apparaît sur la ligne est $i-t(1)$, donc si $t(1)=4$ alors quand "i" est 4 commencera à mettre à jour le Sprite 0, et quand "i" est 11 mettra à jour le Sprite 7 (8 vaisseaux au total).

Appliquons maintenant la même chose aux 4 nœuds. Nous pourrions effectuer 4 vérifications au lieu d'une, mais ce serait inefficace. De plus, si nous avions beaucoup de nœuds, cela signifierait beaucoup de vérifications. Nous pouvons le faire avec une seule, en tenant compte du fait que le premier navire passe par un nœud à un instant $t(n)$ et que le huitième navire passe par ce nœud à $t(n)+7$.

Lorsque le premier navire passe par le premier nœud, il est logique d'envisager de commencer à vérifier le nœud 2, mais pas le nœud 3 ou le nœud 4.

En ce qui concerne le plus petit nœud, nous pouvons supposer que, même si nous avons 20 nœuds, ils sont suffisamment éloignés les uns des autres pour qu'aucun navire ne traverse plus de 3 nœuds à la fois (c'est ce que nous supposerons et nous utiliserons ce "3" comme paramètre). Par conséquent, le plus petit nœud à vérifier est le plus grand - 3. Nous appellerons le plus petit nœud "nmin" et le plus grand "nmax" ($nmin = nmax - 3$). Si nous voulons être totalement libres de définir n'importe quelle trajectoire, nmin doit être égal à nmax moins le nombre de navires dans la rangée.

```
10 j=j+1 : SI j=10 ALORS j=0 : i=i+1 : n=nmax  
20 IF n<nmin THEN 50: ' plus aucun navire à mettre à jour  
30 SI i>=t(n) ET i<=t(n)+8 ALORS [mise à jour du navire i-t(n) avec les  
vitesses des nœuds n]:SI i-t(n)=0 ALORS nmax=nmax+1 : nmin=nmax-3  
40 n=n-1
```

50 ' plus game instructions

Comme vous pouvez le voir, lorsque la décennie de temps (variable "i") est incrémentée de 1, elle commence à vérifier s'il y a un navire dans l'un des nœuds de "nmax" à "nmin", en ne mettant à jour qu'un seul navire à chaque image. Si le navire mis à jour est zéro, le nœud maximum est incrémenté, car ce navire est en route vers le nœud suivant.

Pour la trame suivante, le nombre de nœuds est réduit (instruction $n = n-1$) de sorte que nous vérifions s'il y a un navire dans le nœud précédent, et ainsi de suite jusqu'à nmin, en ne vérifiant toujours qu'un seul navire par trame.

En bref, nous avons transformé 64 vérifications en une seule, en utilisant la "logique de masse". Et si le chemin comportait 40 nœuds au lieu de 4, nous aurions transformé 640 opérations en une seule !

Le jeu vidéo "**Annunaki**" utilise cette technique pour gérer les trajectoires de deux rangées symétriques de 6 vaisseaux chacune. C'est compliqué, mais comme vous pouvez le constater, à partir de BASIC, vous pouvez prendre le contrôle de 12 vaisseaux et leur faire suivre des trajectoires fantaisistes, en utilisant plus d'intelligence que de puissance, grâce à la technique de la logique massive.

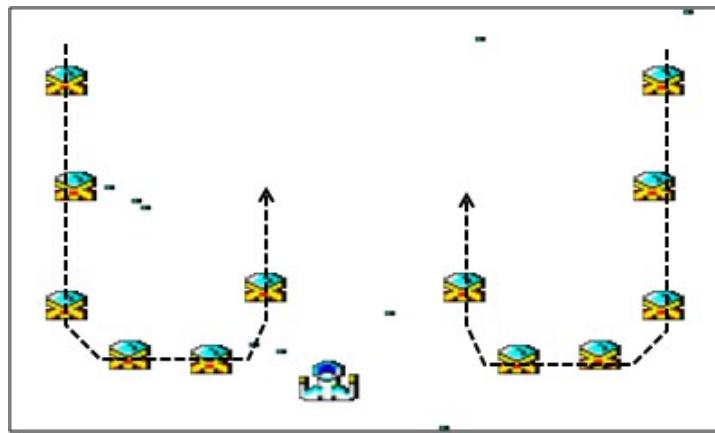


Fig. 68 Deux lignes avec des logiques massives

12 Trajectoires complexes : commande ROUTEALL

Il s'agit d'une commande "avancée" disponible depuis la version V25 de la bibliothèque 8BP. Elle simplifie grandement la programmation car elle permet de définir un chemin et de le faire parcourir pas à pas par un sprite à l'aide de la commande ROUTEALL.

Tout d'abord, vous devez créer une route. Pour cela, vous devez l'éditer dans le fichier routes_yourgame.asm.

Chaque route a un nombre indéterminé de segments (bien que la longueur maximale d'une route soit de 255 octets) et chaque segment a trois paramètres :

- Le nombre de pas que nous allons faire dans ce segment (entre 1 et 250)
- Quelle vitesse Vy doit être maintenue pendant le segment ($-127 \leq Vy \leq 127$) ?
- Vitesse Vx à maintenir pendant le segment ($-127 \leq Vx \leq 127$)

Étant donné qu'un itinéraire peut avoir une longueur maximale de 255 octets et qu'un segment occupe 3 octets, un itinéraire peut comporter au maximum 84 segments.

À la fin de la spécification du segment, nous devons mettre un zéro pour indiquer que le chemin a été complété et que le sprite doit commencer à parcourir le chemin depuis le début.

Prenons un exemple :

; LISTE DES ITINÉRAIRES

=====

**Mettez ici les noms de tous les itinéraires que vous
avez créés ROUTE_LIST**

dw ROUTE0
dw ROUTE1
dw ROUTE2
dw ROUTE3
dw ROUTE4
dw ROUTE4

DÉFINITION DE CHAQUE ITINÉRAIRE

=====

ROUTE0 ; un cercle

**db 5,2,0 ; cinq étapes avec Vy=2
db 5,2,-1 ; cinq étapes avec Vy=2, Vx=-1
db 5,0,-1
db 5,-2,-1
db 5,-2,0
db 5,-2,1
db 5,0,1
db 5,2,1
db 0**

ROUTE1 ; gauche-droite

**db 10,0,-1
db 10,0,1
db 0**

ROUTE2 ; haut-bas

**db 10,-2,0
db 10,2,0**

```
db 0
```

ROUTE3 ; un huit

```
;-----  
db 15,2,0  
db 5,2,-1  
db 5,0,-1  
db 25,-2,-1  
db 5,0,-1  
db 5,2,-1  
db 15,2,0  
db 5,2,1  
db 5,0,1  
db 25,-2,1  
db 5,0,1  
db 5,2,1  
db 0
```

ROUTE4 ; une boucle et va vers la gauche

```
;-----  
db 120,0,-1  
db 10,-2,-1  
db 20,-2,0  
db 10,-2,1  
db 5,0,1  
db 10,2,1  
db 20,2,0  
db 10,2,-1  
db 80,0,-1  
db 0
```

Maintenant, pour utiliser les routes à partir de BASIC, il suffit d'assigner la route à un sprite avec la commande SETUPSP, en indiquant que l'on veut modifier le paramètre 15, qui est celui qui indique la route. En outre, nous devons activer le drapeau de la route (bit 7) dans l'octet d'état du sprite et nous l'associerons au drapeau de mouvement automatique et aux drapeaux d'animation et d'impression.

```
10 MÉMOIRE 24999  
11 ON BREAK GOSUB 280  
20 MODE 0:INK 0,0  
21 LOCATE 1,20:PRINT "commande |ROUTEALL et macroséquences  
d'animation".  
30 APPEL &6B78:DEFINT a-z  
31 |SETLIMITS,0,80,0,200  
40 FOR i=0 TO 31:|SETUPSP,i,0,0,0:NEXT  
41 x=10  
50 FOR i=1 TO 8  
51 x=x+20:SI x>=80 ALORS x=10:y=y+24  
60 |SETUPSP,i,0,143 : rem avec ceci active le drapeau de route  
70 |SETUPSP,i,7,2:|SETUPSP,i,7,33 : rem séquence d'animation macro  
71 |SETUPSP,i,15,3 : l'itinéraire numéro 3 est réattribué  
80 |LOCATESP,i,30,70  
82 FOR t=1 TO 10:|ROUTEALL:|AUTOALL,0:|PRINTSPALL,1,0:NEXT  
91 NEXT  
100 |AUTOALL,1:|PRINTSPALL,1,0 : rem ici AUTOALL invoque déjà ROUTEALL  
120 GOTO 100
```

280 |MUSIQUE:MODE 1 : ENCRE 0,0:STYLO 1

Nous avons tout ce qu'il faut. Cette technique avancée simplifiera grandement votre programmation avec des résultats spectaculaires.



Fig. 69 un itinéraire en forme de 8

Comme vous l'avez vu, la commande ne modifie pas les coordonnées des sprites, qui doivent donc être déplacés avec **|AUTOALL** et imprimés (et animés) avec **|PRINTSPALL**. C'est pourquoi vous avez un paramètre optionnel dans **|AUTOALL**, de sorte que **|AUTOALL,1** invoque en interne **|ROUTEALL** avant de déplacer le sprite, vous épargnant ainsi une invocation BASIC qui prendra toujours une précieuse milliseconde.

12.1 Place un Sprite au milieu d'une route : ROUTESP

Si vous assignez plusieurs sprites à la même route et que vous voulez qu'ils aillent tous dans un seul fichier, comme dans l'exemple ci-dessus, vous devez vous assurer que chaque sprite se trouve à un point différent de la route. Il y a deux façons de procéder

La première consiste à assigner progressivement la route aux sprites. De cette façon, le sprite en tête est le premier à être acheminé et après quelques cycles de jeu, vous acheminez le sprite suivant, puis le suivant, et ainsi de suite. Dans chaque cycle, vous exécutez **|AUTOALL,1** et cela route les sprites qui ont déjà une route assignée à eux, qui seront en avance dans le nombre d'étapes par rapport aux sprites qui n'ont pas encore de route assignée.

La deuxième option consiste à utiliser la commande
|ROUTESP.

|ROUTESP, <spriteid>, <steps>.

Cette commande déplace un sprite du nombre de pas souhaité (**jusqu'à 255 pas**) le long du chemin qui lui a été assigné. Dans l'exemple, j'ai surligné en rouge l'assignation du chemin 8 et les commandes **|ROUTESP** qui positionnent chacun des sprites le long du chemin.

```
10 mémoire 24999
10 ON BREAK GOSUB 12
11 GOTO 20
12 |MUSIC:CALL &BC02:PAPER 0:OPEN 1:MODE 1:END
20 CALL &BC02:DEFINT A-Z:MODE 0
50 APPEL &6B78
```

70 ' tous les navires placés à la même coordonnée initiale
75 ' et avec le même itinéraire mais avec un nombre initial d'étapes différent.

```

76 locate 2,3 : Imprimer "commande |ROUTEESP "
80 pour s=16 à 21 : |SETUPSP,s,9,33 : |SETUPSP,s,0,137 :
|SETUPSP,s,15,8:|LOCATESP,s,20,100:next
90 s=21:|ROUTEESP,s,40 : "tête de navire
100 s=20:|ROUTEESP,s,30
110 s=19:|ROUTEESP,s,20
120 s=18:|ROUTEESP,s,10
130 s=17:|ROUTEESP,s,0
131 |PRINTSPALL,0,0,0,0,0
140 --- cycle de jeu ---
150 |AUTOALL,1:|PRINTSPALL
160 goto 150

```

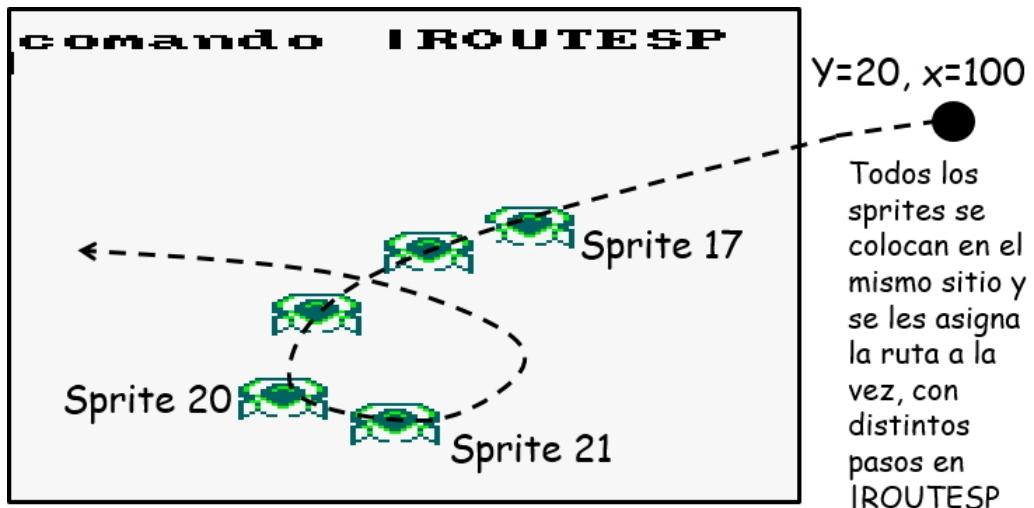


Fig. 70 Maisons en rangée grâce à l'utilisation de ROUTESP

La route 8 serait définie comme suit dans le fichier routes_mygame.asm :

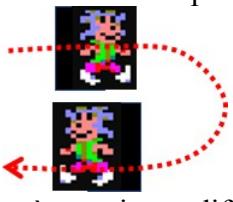
```

ROUTE8 ;
    db 255, 128+64+32+32+8+1,0 ; changement d'état
    db 70,1,-1
    db 10,3,-1
    db 5,3,0
    db 5,3,1
    db 5,0,1
    db 20,-3,1
    db 5,0,1
    db 5,3,1
    db 5,3,0
    db 10,3,-1
    db 5,0,-1
    db 20,-3,-1
    db 40,-1,-1
    repositionne
    ment db
    1,0,115
    db 1,-30,0
    db 120,0,0
    db 120,0,0
    db 0

```

12.2 Création avancée d'itinéraires

Il existe 4 fonctionnalités que vous pouvez utiliser au milieu de n'importe quel itinéraire (**attention, au milieu, pas à la fin**), en utilisant un code d'échappement comme valeur du nombre d'étapes d'un segment :

Code d'évasion n	Description	Exemple
255	Changement d'état du sprite.	DB 255, 3, 0 Le zéro à la fin est un bouche-trou.
254	Modification de la séquence d'animation du sprite  Après avoir modifié la séquence, si vous souhaitez que l'image change également, vous devez utiliser le code 251	DB 254, 10, 0 La séquence 10 est associée. Le zéro est un bouche-trou Si la séquence assignée est celle que le sprite possède déjà, l'opération est inoffensive (l'identifiant de l'image n'est pas réinitialisé). Si vous souhaitez réinitialiser l'identifiant du cadre, le troisième paramètre doit être un 1, par exemple : DB 254, 10, 1
253	Changement d'image 	DB 253 DW new_img L'image "new_img" est associée, qui doit être une adresse mémoire.
252	Changement d'itinéraire	DB 252, 2, 0 La route 2 est associée
251	Aller à jusqu'à suivant cadre de l'animation. 	DB 251, 0, 0 Le Sprite est animé. Les deux zéros sont des éléments de remplissage

IMPORTANT : faites très attention à écrire DB et DW là où ils doivent être utilisés, c'est-à-dire, par exemple, si vous changez d'image, vous devez faire précéder l'image de DW et non de DB. Si vous faites une telle erreur, votre itinéraire ne fonctionnera pas.

IMPORTANT : les codes d'échappement peuvent être utilisés au milieu d'un itinéraire, mais le dernier segment ne peut pas être un code d'échappement, il doit s'agir d'un mouvement, même s'il est immobile, quelque chose comme "DB 1,0,0".

12.2.1 Changements d'état forcés des itinéraires

Cette capacité est très intéressante pour accélérer vos parties et est disponible depuis la V27. Elle permet de forcer un changement d'état au milieu d'un parcours. Pour ce faire, nous indiquons que nous voulons un changement d'état en indiquant le nombre de pas dans le segment sous la forme d'une valeur = 255.

Le changement d'état est un segment supplémentaire et il est important de conserver le même nombre de paramètres par segment, c'est-à-dire 3 octets. Un changement d'état à l'état=13 pourrait s'écrire comme suit :

255,13,0

Le troisième paramètre (le zéro) ne signifie rien, c'est juste un "bouche-trou" pour que le segment mesure 3 octets, mais il est essentiel.

La valeur 255 indique à la commande **|ROUTEALL** qu'elle doit cette fois-ci changer l'état du sprite, en lui attribuant l'état indiqué ci-dessous. Le changement d'état est exécuté sans consommer de pas, de sorte que le pas suivant le changement d'état sera toujours exécuté. Si nous ne voulons plus que le sprite bouge, nous définissons simplement un segment d'un pas sans mouvement en X ou en Y juste après le changement d'état. Voyons un exemple :

```
ROUTE3 ; fire_dere
;-----
db 40,0,2 ; quarante pas vers la droite avec Vx=2 db
255,0,0 ; changement d'état à zéro
db 1,0,0 ; toujours Vy=0, Vx=0
db 0

ROUTE4 ; trigger_izq
;-----
db 40,0,-2 ; quarante pas vers la gauche avec Vx=-2 db
255,0,0 ; changement d'état à zéro
db 1,0,0 ; toujours Vy=0, Vx=0
db 0
```

Nous allons utiliser ces deux routes pour tirer avec notre personnage. Le premier, après avoir parcouru 40 pas au cours desquels il avance de 2 octets en X, subit un changement d'état et le sprite passe à l'état 0, c'est-à-dire désactivé. Le segment suivant n'a qu'un seul pas et aucun mouvement (vy=0, vx=0).

Grâce à ce mécanisme, nous pouvons déclencher et faire en sorte que les déclencheurs se désactivent d'eux-mêmes lorsqu'ils sortent de l'écran. Cela permet d'économiser la logique BASIC et d'accélérer nos jeux. Dans l'exemple suivant, l'image 26 est le déclencheur.



Fig. 71 Déclenchements avec changement de statut en route

```
10 MÉMOIRE 24999
20 MODE 0 : DEFINT A-Z : CALL &6B78:' install RSX
25 ON BREAK GOSUB 280
30 CALL &BC02 : "restaurer la palette par défaut au cas où".
40 INK 0,0 : "fond noir"
50 FOR j=0 TO 31:|SETUPSP,j,0,&X0:NEXT:'réinitialiser les sprites
80 |SETLIMITS,12,80,0,186 : ' définit les limites de l'écran de jeu
90 x=40:y=100:' coordonnées du caractère
100 |SETUPSP,0,0,1:' statut du caractère
110 |SETUPSP,0,7,1:dir=1:' séquence d'animation assignée au démarrage
120 |LOCATESP,0,y,x:'place le sprite (sans l'imprimer)
```

```

125 |MUSIC,0,0,6
126 for i=1 to 4:|SETUPSP,10+i,9,26:next:'shots
130 cycle de jeu---
150 |AUTOALL,1:|PRINTSPALL,0,0
170 ' character movement routine -----
    IF INKEY(27)=0 THEN IF dir=2 THEN dir=1:|SETUPSP,0,7,dir ELSE
    |ANIMA,0:x=x+1:GOTO 191
190 SI INKEY(34)=0 ALORS SI dir=1 ALORS dir=2:|SETUPSP,0,7,dir ELSE
    |ANIMA,0:x=x-1
191 IF wait<cycle-10 then if INKEY(47)=0 THEN wait=cycle:disp= 1+ disp
mod 4 :|LOCATESP,10+disp,y+8,x : |SETUPSP,10+disp,0,137 :
    |SETUPSP,10+disp,15,2+dir
200 |LOCATESP,0,y,x
201 cycle=cycle+1
210 goto 150
280 |MUSIQUE:MODE 1 : ENCRE 0,0:STYLO 1

```

Les changements d'état peuvent être forcés sur n'importe quel segment de l'itinéraire, pas nécessairement à la fin, bien que dans le cas d'un déclencheur, il est très logique de le faire à la fin de l'itinéraire.

12.2.2 Changements de séquence forcés à partir des itinéraires

Nous pouvons changer la séquence d'animation d'un sprite en utilisant un segment spécial. En mettant 254 dans la valeur du nombre de pas, la commande ROUTEALL interprétera qu'un changement de séquence d'animation doit être effectué sur le sprite. Exemple :

254,10,0

Ce segment modifie la séquence d'animation du sprite, en lui attribuant le numéro de séquence 10. Le troisième paramètre (le zéro) signifie que la séquence ne sera pas redémarrée, donc si le sprite se trouve dans la frame 5 d'une autre séquence, la nouvelle séquence sera assignée et la disco frame sera maintenue, bien qu'elle corresponde maintenant à une autre image. Ceci est très utile car nous pouvons assigner une séquence à un sprite à l'intérieur d'un chemin et s'il l'avait déjà, c'est inoffensif. Pour réinitialiser la séquence assignée (à la frame 0), il faut mettre un 1 dans le troisième paramètre :

254,10,1

Dans le cas où vous assignez une séquence et souhaitez qu'elle redémarre, il est conseillé d'utiliser le code d'animation, le 251 que nous verrons plus tard. Ainsi, c'est l'image associée au sprite qui changera dans la table des sprites et pas seulement l'**identifiant de la frame**.

Comme pour le changement d'état, le changement de séquence est exécuté sans consommer d'étape, de sorte que l'étape suivante du changement de séquence sera toujours exécutée.

12.2.3 Changements d'image forcés à partir des itinéraires

Nous avons vu comment router les sprites avec ROUTEALL ou, mieux encore, avec AUTOALL,1

Souvent, nous ne voulons pas faire suivre une trajectoire à un sprite, mais quelque chose de plus quotidien : sauter avec un personnage. Dans l'exemple du chapitre 9, nous avons vu comment faire cela avec un tableau BASIC contenant les mouvements relatifs de la coordonnée Y. Dans ce cas, nous allons faire la même chose avec une trajectoire, pour obtenir un mouvement plus rapide. Dans ce cas, nous allons faire la même chose avec une trajectoire, pour obtenir un mouvement plus rapide.

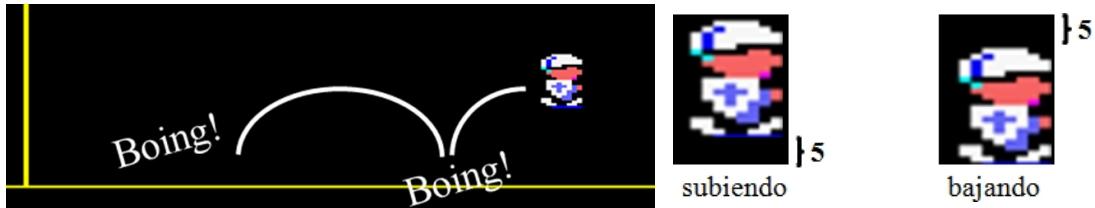


Fig. 72 Sauter un itinéraire

Pour éviter de devoir vérifier si le caractère a atteint le point zénithal du saut, nous pouvons utiliser un segment spécial qui indique le changement d'image. Comme tout segment, il consomme 3 octets, mais dans ce cas, le premier est l'indicateur de changement d'image (une valeur de 253) et les deux suivants correspondent à l'adresse mémoire de l'image. **ATTENTION**, vous devez utiliser "**dw**" avant le nom de l'image que vous voulez allouer, vous devrez donc écrire ce segment de changement d'image en deux lignes. Un "**db**" pour le 253 et un "**dw**" pour l'adresse mémoire de l'image.

```
db 253
dw SOLDIER_R1_UP
```

Dans l'exemple suivant, nous avons un mannequin qui saute. En montant, le mannequin s'efface en bas, tandis qu'en descendant, il s'efface en haut. Pour qu'il n'y ait pas de discontinuité dans le mouvement, il est nécessaire, au moment de remplacer une image par une autre, de réaligner le soldat verticalement, en élevant l'image descendante d'exactement 5 lignes, pour qu'elle coïncide avec le soldat qui montait.

Il s'agirait du fichier sequences_mygame.asm

jusqu'à 31 séquences d'animation
doit être un tableau fixe et non un tableau variable
chaque séquence contient les adresses des images d'animation cycliques
Chaque séquence correspond à 8 adresses de mémoire image.
nombre pair car les animations sont généralement un nombre pair
un zéro signifie la fin de la séquence, bien que 8 mots soient toujours utilisés.
Lorsqu'un zéro est trouvé, un nouveau départ est effectué.
s'il n'y a pas de zéro, il recommence après la huitième image.

La séquence zéro, c'est qu'il n'y a pas de séquence.

Nous commençons par la séquence 1

;-----animation sequences -----
LISTE DES SÉQUENCES
dw SOLDIER_R0,SOLDIER_R2,SOLDIER_R1,SOLDIER_R2,0,0,0,0,0 ; 1
dw SOLD_L0,SOLD_L2,SOLD_L1,SOLD_L2,0,0,0,0 ;2 dw
SOLD_R1_UP,0,0,0,0,0,0;3
dw
SOLDIER_R1_DOWN,0,0,0,0,0,0,0;4
dw SOLDIER_L1_UP,0,0,0,0,0,0,0;5
dw
SOLDIER_L1_DOWN,0,0,0,0,0,0,0;6

SÉQUENCES MACRO

;-----SÉQUENCES MACRO -----

sont des groupes de séquences, un pour chaque direction.

; le sens est :

; encore, gauche, droite, haut, haut-gauche, haut-droite, bas, bas-gauche, bas-droite

Les numéros sont numérotés à partir de 32

db 0,2,1,3,5,3,3,4,6,4 ; 32 --> séquences de soldats, id=32. la suivante serait 33

Nous utiliserons deux routes, l'une pour sauter à droite et l'autre pour sauter à gauche. Ce sera le fichier routes_mygame.asm.

; LISTE DES ITINÉRAIRES

;=====

; mettre ici les noms de tous les itinéraires que vous

faites ROUTE_LIST

dw ROUTE0
dw ROUTE1
dw ROUTE2
dw ROUTE3
dw ROUTE4
dw ROUTE4

DÉFINITION DE CHAQUE ITINÉRAIRE

;=====

ROUTE0 ; jump_right

db 253
dw SOLDIER_R1_UP
db 1,-5,1
db 2,-4,1
db 2,-3,1
db 2,-2,1
db 2,-1,1
db 253
dw SOLDIER_R1_DOWN
db 1,-5,1 ; up so UP et down fit db 2,1,1
db 2,2,1
db 2,3,1
db 2,4,1
db 1,5,1
db 253
dw SOLDIER_R1
db 1,5,1 ; descendre encore d'un cran, car R1 n'a pas de noir au sommet
db 255,13,0 ; nouvel état, sans drapeau de chemin et avec drapeau
d'animation db 254,32,0 ; séquence macro 32
db 1,0,0 ; quietooo. !!!!
db 0

ROUTE1 ; jump_left

db 253
dw SOLDADO_L1_UP
db 1,-5,-1
db 2,-4,-1
db 2,-3,-1
db 2,-2,-1
db 2,-1,-1
db 253
dw SOLDIER_L1_DOWN
db 1,-5,-1 ; soulever pour s'adapter à UP et down
db 2,1,-1

```

db 2,2,-1
db 2,3,-1
db 2,4,-1
db 1,5,-1
db 253
dw SOLDIER_L1
db 1,5,-1 ; descente d'une unité supplémentaire
db 255,13,0 ; nouvel état, sans drapeau de chemin et avec drapeau
d'animation db 254,32,0 ; séquence macro 32
db 1,0,0 ; quietooo. !!!!
db 0

ROUTE2 ; oiseau
db 30,0,-1
db 10,0,0
db 20,2,-1
db 20,-2,-1
db 0

ROUTE3 ; fire_dere
;-----
db 40,0,2
db 255.0
db 1,0,0
db 0

ROUTE4 ; trigger_izq
;-----
db 40,0,-2
db 255.0
db 1,0,0
db 0

```

Il s'agit du programme d'exemple. Comparez son exécution avec celle du chapitre 7 pour voir la différence de vitesse. Vous remarquerez une grande différence

```

10 MÉMOIRE 24999
20 MODE O : DEFINT A-Z : CALL &6B78:' install RSX
25 À LA PAUSE GOSUB 2800
30 CALL &BC02:ink 0,0:'rétablit la palette par défaut
50 FOR j=0 TO 31:|SETUPSP,j,0,0:NEXT:'réinitialiser les sprites
80 |SETLIMITS,12,80,0,186 : ' définit les limites de l'écran de jeu
90 x=40:y=100:jump=0:cycle=40:' coordonnées du personnage
100 |SETUPSP,0,0,13:|SETUPSP,0,5,0,0:' statut des caractères
110 |SETUPSP,0,7,1:|SETUPSP,0,7,32:'séquence d'animation assignée au
démarrage

120 |LOCATESP,0,y,x:'nous plaçons le sprite      l'imprimer)
(sans
123 locate 1,1 : print "press Q" : print "stop" skip" : print "exemple
: print "for
avec itinéraire"
124 print "press SPACE to fire" print "press
SPACE to fire" print "press SPACE to fire"
print "press SPACE to fire" print "press
SPACE to fire" print
125 PARCELLE 1,150:DESSIN 640,150 : PARCELLE      92.400 : "sol et mur
92,150:DESSIN
126 for i=1 to 4:|SETUPSP,10+i,9,26:next:'shots

```

|MUSIC,0,0,5 : 'la musique commence à jouer
130 ----- "cycle de jeu".
150 |AUTOALL,1:|PRINTSPALL,0,1,0
170 ' character movement routine -----

```

172 IF INKEY(47)=0 THEN if wait<cycle-10 then wait=cycle:disp= 1+ disp
mod
4:|LOCATESP,10+disp,peek(27001)+8,peek(27003):|SETUPSP,10+disp,0,137:|
SETUPSP,10+disp,15,3+dir
173 if peek(27000)>128 then 193 else |SETUPSP,0,6,0 : ' if state is
>128 c'est que je saute (a route)
174 SI INKEY(67)=0 ALORS |SETUPSP,0,0,137:|SETUPSP,0,15,dir:'skip
180 IF INKEY(27)=0 THEN dir=0:|SETUPSP,0,6,1:'ir derecha
190 SI INKEY(34)=0 ALORS dir=1:|SETUPSP,0,6,-1:'ir gauche
193 cycle=cycle+1
310 goto 150
2800 |MUSIQUE:MODE 1 : ENCRE 0,0:STYLO 1

```

Pour vérifier que le mannequin saute, j'interroge simplement le statut par peek (27000). De cette façon, nous saurons si le drapeau de routage est actif et si c'est le cas, nous ne passerons pas par les lignes qui le déplacent de gauche à droite.

12.2.4 Reroutage forcé à partir d'itinéraires

Nous pouvons changer le chemin d'un sprite en utilisant un segment spécial. Lorsque nous mettons 252 dans la valeur du nombre de pas, la commande ROUTEALL interprétera qu'un changement de chemin doit être apporté au sprite. Exemple :

252,2,0

Ce segment modifie le chemin du sprite, en lui attribuant le numéro de chemin 2. Le troisième paramètre (le zéro) ne signifie rien, c'est juste un "bouche-trou" pour que le segment mesure 3 octets, mais il est essentiel.

Comme pour le changement d'état, le changement d'itinéraire est exécuté sans consommer d'étape, de sorte que l'étape suivante après le changement d'itinéraire sera toujours exécutée, c'est-à-dire la première étape du premier segment du nouvel itinéraire.

Étant donné qu'un itinéraire peut avoir une longueur maximale de 255 octets et qu'un segment a une longueur de 3 octets, un itinéraire peut avoir une longueur maximale de 84 segments. Vous pouvez avoir besoin de construire un itinéraire encore plus long, et dans ce cas, vous pouvez le faire en concaténant la fin d'un itinéraire avec un changement d'itinéraire vers un autre itinéraire, et vous pouvez concaténer autant d'itinéraires que vous le souhaitez.

12.2.5 Changements de chemin forcés à partir de BASIC

Supposons que vous vouliez que votre personnage saute vers la droite et qu'au milieu du saut, vous vouliez que le personnage puisse changer vers la gauche, en continuant le saut. Ce que nous proposons peut être représenté par ce dessin :

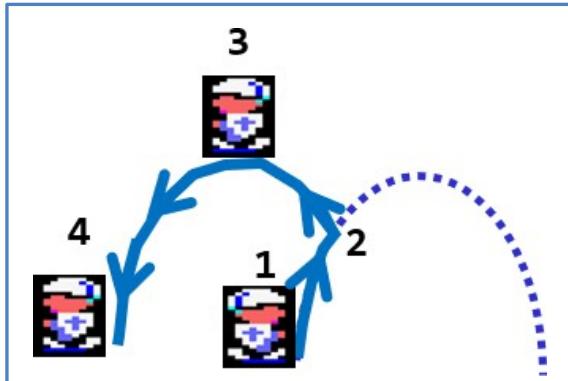


Fig. 73 Changement d'itinéraire au milieu d'un saut

Dans cet exemple, notre personnage saute et au milieu de la trajectoire de saut vers la droite, il change de direction au point 2, continuant la trajectoire de saut vers la gauche, mais sans commencer un nouveau saut, continuant simplement le saut et atteignant la même hauteur (point 3) qu'il atteindrait avec le saut vers la droite pour finalement terminer le saut vers la gauche au point 4.

Pour ce faire, nous devons changer le chemin de notre personnage depuis BASIC au point 2, mais nous ne pouvons pas utiliser la commande **|SETUPSP** car cela initialiserait le chemin, ce qui entraînerait un saut beaucoup plus important, à partir du point 2. Ce que nous pouvons faire dans ce cas, c'est simplement **POKE** à l'adresse mémoire qui stocke le chemin du Sprite, qui est l'adresse de son état +15, comme indiqué dans la table d'attributs du Sprite. Ce **POKE** modifie le chemin, mais conserve la position intacte (numéro de segment et position où se trouve le caractère). ATTENTION : les deux chemins doivent avoir les mêmes segments et la même longueur, car si vous sautez sur un chemin plus court et que vous vous retrouvez sur un segment qui n'existe pas sur ce chemin, un effet imprévisible peut se produire.

En supposant que nous ayons deux chemins de saut (le chemin 0 sautant à droite et le chemin 1 sautant à gauche), l'exemple suivant illustre le concept :

```

130 ----- "cycle de jeu".
150 cycle=cycle+1:AUTOALL,1:|PRINTSPALL,0,1,0
170 ' character movement routine -----
173 SI PEEK(27000)<128 ALORS 178
174 nous sommes en plein saut
175 IF INKEY(27)=0 THEN POKE 27015,0:GOTO 180:'change path to the right
    IF INKEY(34)=0 THEN POKE 27015,1:GOTO 180:'change path to left
177 GOTO 150
178 IF INKEY(67)=0 THEN |SETUPSP,0,0,137:|SETUPSP,0,15,dir:'skip
    IF INKEY(27)=0 THEN dir=0:|SETUPSP,0,6,1:'ir derecha
190 IF INKEY(34)=0 THEN dir=1:|SETUPSP,0,6,-1:'ir gauche
310 GOTO 150

```

La seule limitation de l'utilisation de **POKE** à cette fin est que l'image du personnage ne change pas tant qu'il ne rencontre pas un code de changement d'image au milieu du chemin. Changer l'image en utilisant **|SETUPSP** est possible mais dangereux, car vous ne savez pas si le personnage monte (effacé avec les lignes du bas) ou descend (effacé avec les lignes du haut). Il est donc préférable d'assigner simplement le chemin et de faire en sorte que le chemin lui-même modifie l'image dès que possible. Vous pouvez même placer les changements d'image au milieu de l'itinéraire, même s'ils ne sont pas nécessaires au cas où cette circonstance se produirait. Ce dont il faut s'assurer, c'est que l'image de saut comporte des octets de suppression des deux côtés, car si ce n'est pas le

cas et qu'elle change de direction, elle laissera une trace.

12.2.6 Animation forcée des itinéraires

Nous pouvons laisser le drapeau d'animation d'un Sprite inactif et ne l'animer qu'à certains moments de l'itinéraire en utilisant le code 251 :

251,0,0

Cela peut être très utile pour donner l'impression qu'un sprite approche dans les jeux qui utilisent des techniques de pseudo-3D. Par exemple, dans le cas d'une météorite en approche, nous voulons changer les images pour qu'elle paraisse plus grande. La météorite se déplacera plusieurs fois avant de passer à la taille suivante. Ce mécanisme est très similaire au mécanisme de changement d'image, sauf qu'il permet de définir le changement d'image sans spécifier explicitement l'image, mais en indiquant simplement un changement d'image dans la séquence d'animation assignée au sprite.



Fig. 74 Animation forcée de l'itinéraire

Avec ce drapeau, vous pouvez utiliser la même route pour l'approche d'un oiseau de l'espace ou d'une météorite. En n'indiquant pas l'image, c'est l'image correspondant à la séquence de chaque sprite qui sera appliquée dans chaque cas.

12.2.7 Comment construire des itinéraires "dynamiques" (non prédefinis) ?

Une route dynamique est une route dont le chemin est décidé à l'exécution dans votre programme BASIC. Elle est utile lorsque votre programme génère dynamiquement des labyrinthes ou des pistes de course qu'un ennemi doit traverser et qui ne sont pas connus a priori et ne peuvent donc pas être définis dans le fichier "routes_mygame.asm".

Pour créer une telle route et l'assigner à un Sprite, nous devons créer une route "vide" dans le fichier "routes_mygame.asm", dans ce cas la route 2.

; LISTE DES ITINÉRAIRES

=====

**Mettez ici les noms de tous les itinéraires que vous
avez créés ROUTE_LIST**

dw ROUTE0

dw ROUTE1

dw ROUTE2

dw ROUTE3

dw ROUTE4

dw ROUTE4

DÉFINITION DE CHAQUE ITINÉRAIRE

=====

```
ROUTE0 ; right left db 10,0,1
db 10,0,-1
db 0
```

```
ROUTE1 ; up down db
10,-1,0
db 10,1,0
db 0
```

```
ROUTE2 ; routage
dynamique ds 100
```

Nous avons créé le chemin 2 avec 100 octets libres à remplir à partir de BASIC. Il se peut que le chemin que nous construisons prenne moins de place et, dans ce cas, il suffira de réserver moins d'octets.

Une fois la bibliothèque et les graphiques assemblés, nous devons rechercher l'adresse mémoire de l'étiquette "ROUTE2" dans la fenêtre de symboles de winape. Lorsque nous l'aurons trouvée, à partir de BASIC, nous programmerons la route en utilisant POKE à partir de cette adresse mémoire et des suivantes

POKE place un octet dans une adresse mémoire. Nos nombres doivent appartenir à l'intervalle -127..128 et POKE ne permet pas de mettre des nombres négatifs. Pour ce faire, vous devez utiliser la valeur positive avec laquelle l'Amstrad représente les négatifs en interne (c'est-à-dire le complément à deux). Pour ce faire, il suffit d'effectuer une opération AND 255

```
10 A=-10
```

```
20 PRINT A : REM ceci imprime un 10
```

```
30 PRINT A AND 255 : REM ceci imprime un 246 qui est le même -10
```

En résumé, si vous voulez insérer un -10, vous devez insérer un 246 et utiliser la même stratégie pour tout nombre négatif. N'oubliez pas que le dernier POKE de l'itinéraire doit être l'insertion d'un zéro, ce qui signifie la fin de l'itinéraire.

12.2.8 Programmation d'itinéraires, y compris de schémas

Supposons que vous souhaitez créer un chemin pour qu'un ennemi traverse l'écran de droite à gauche, encore et encore, en partant d'une position initiale où le sprite se trouve à l'extrême droite de l'écran.

```
ROUTE0 ; route unique
```

```
DB 80,0,-1 ; 80 étapes. A chaque étape, 1 octet est déplacé
```

```
DB 1,0,80 ; repositionne le sprite à sa position d'origine
```

```
DB 0
```

Ce chemin déplace un sprite avec un pas de 1 octet à chaque image. Si nous voulions le ralentir, en déplaçant un octet toutes les deux images, nous pourrions faire ce qui suit :

```
ROUTE0 ; route pas si simple DB
```

```
1,0,-1
```

```
DB 1,0,0
```

```
DB 0
```

Cette route nous permet de déplacer un sprite plus lentement, mais nous ne pouvons pas repositionner le sprite à sa position d'origine. En effet, l'écran ayant une largeur de 80 octets, nous avons besoin de 160 segments, car le sprite avance d'un octet tous les deux segments. Le chemin résultant serait très long et il faudrait en fait concaténer 2 chemins car un chemin ne peut mesurer que 255 octets (84 segments).

La solution optimale consiste à définir un chemin court sans repositionner le sprite et à le repositionner à partir de BASIC, en utilisant quelque chose comme ceci :

```
80 |SETUPSP,31, 0, 128+16+1 : REM rotatif, mov automatique, imprimable  
85 |LOCATESP,31,100,80:' placé à droite de l'écran.  
Cycle de jeu de 90 rem  
100 cycle=cycle +1  
110 |AUTOALL,1 : |PRINTSPALL  
120 if MOD cycle 160=0 THEN |LOCATESP,31,100,80 : ' repositionnement  
130 goto 100
```

Ce type de stratégie est utile lorsque nous ne voulons pas répéter le même mouvement à chaque image, mais que nous voulons définir un modèle répétitif dans lequel, dans certaines images, le sprite se déplace dans une direction et dans d'autres, il se déplace dans une autre direction ou même pas du tout. Prenons un autre exemple, celui d'une trajectoire inclinée dans laquelle, tous les 3 mouvements verticaux, nous effectuons un mouvement horizontal.

```
ROUTEO ; route inclinée  
DB 2,1,0  
DB 1,1,1,1  
DB 0
```

12.2.9 Typologie des itinéraires

Compte tenu de tout ce que nous avons vu, nous pouvons classer les itinéraires selon les types suivants :

- **Chemins cycliques sans fin** : ils repositionnent le sprite ou se terminent simplement aux mêmes coordonnées que celles de leur point de départ.
- **Chemins sans fin non cycliques** : ils avancent indéfiniment et ne repositionnent pas le sprite, de sorte qu'ils peuvent s'éloigner infiniment de l'aire de jeu, à moins que nous ne repositionnions le sprite à partir de BASIC.
- **Routes avec fin** : dans la dernière étape, changer l'état du sprite en désactivant le drapeau de routage.
- **Itinéraires enchaînés** : à partir d'un itinéraire, vous pouvez passer à un autre itinéraire et ce deuxième itinéraire peut être cyclique ou non cyclique, avoir une fin, ou même se terminer par un saut vers un troisième itinéraire.

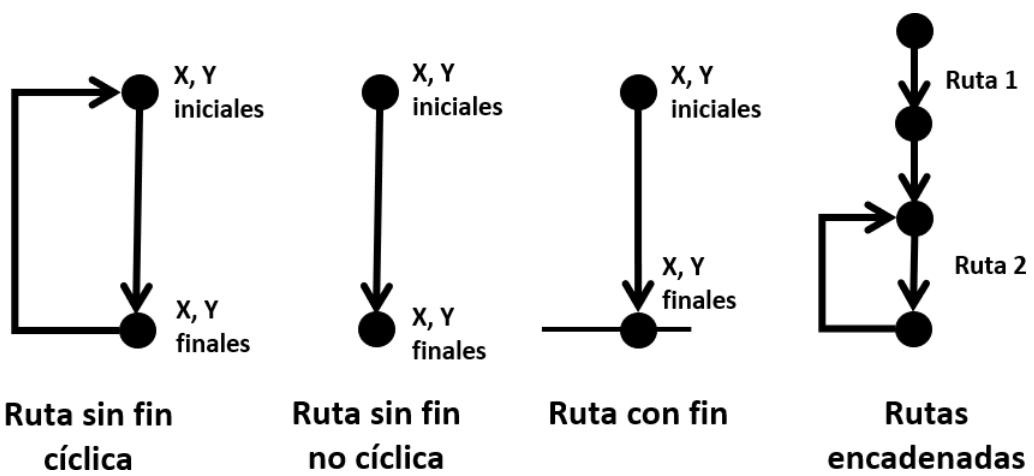
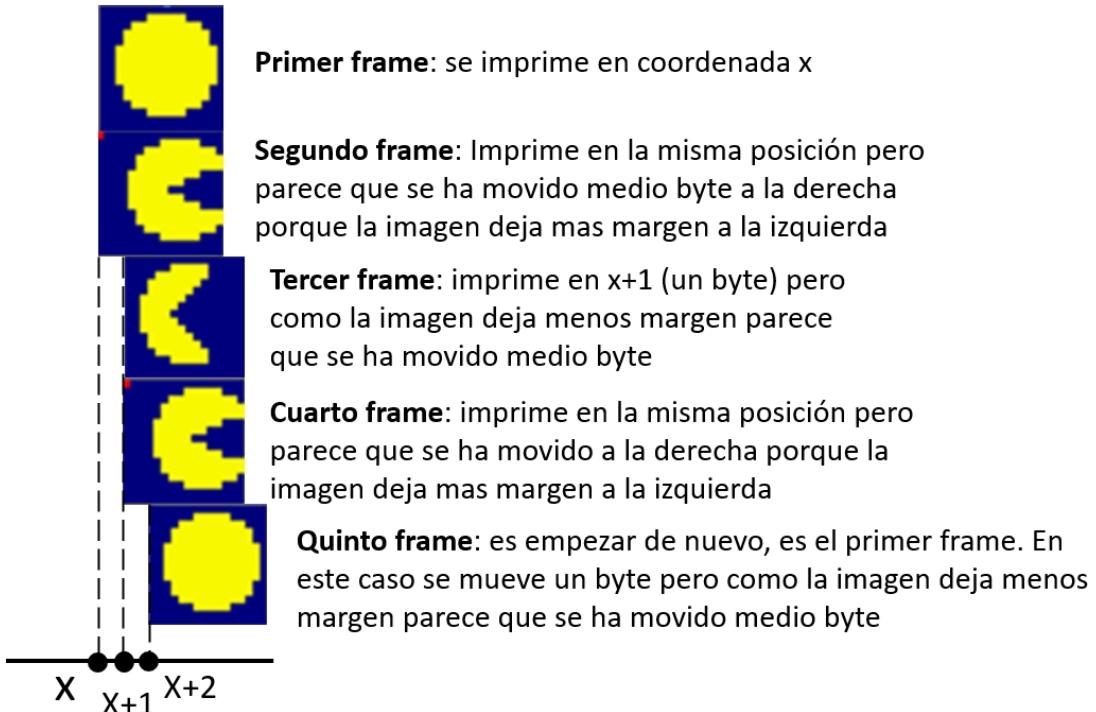


Fig. 75 Types d'itinéraires

13 Mouvement doux d'un demi-octet

Le 8BP déplace les sprites octet par octet à l'aide de commandes telles que MOVE, et son système de coordonnées est constitué d'octets, et non de pixels. Il s'agit donc de 80 positions sur l'axe horizontal et de 200 sur l'axe vertical.

Un octet contient 2 pixels en mode 0, ou 4 pixels en mode 1. Nous pouvons souhaiter un mouvement plus fluide (pixel par pixel en mode 0 ou deux pixels en mode 1). Il existe une astuce simple pour y parvenir. Il s'agit d'avoir une image du caractère décalée d'un demi-octet et de l'assigner simplement au Sprite. Même si elle est imprimée à la même coordonnée, elle aura l'air d'avoir bougé.

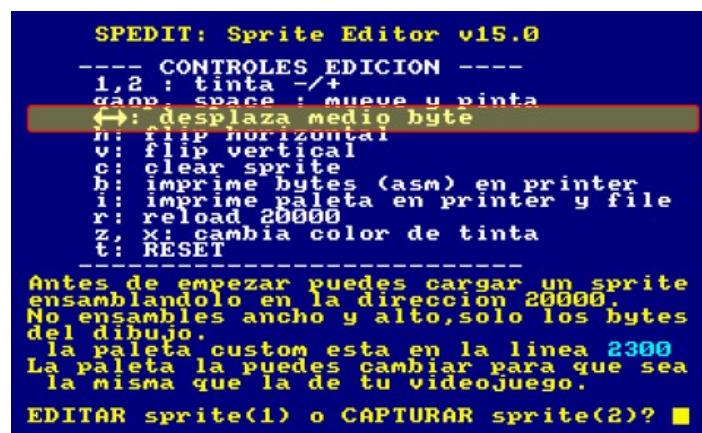


Dans cette séquence d'animation, il y a des moments où le Pac-Man ne bouge pas, mais lorsque nous changeons l'image, c'est comme s'il se déplaçait de 2 pixels (un demi-octet). Lorsqu'il se déplace d'un octet, l'image apparaît décalée d'un demi-octet vers la gauche, de sorte que le résultat "net" est également comme s'il se déplaçait de 2 pixels (un demi-octet). Pour définir le mouvement de ce Pac-Man, nous pouvons utiliser le mécanisme de trajectoire 8BP. Voici un exemple de trajectoire de déplacement vers la droite :

```
ROUTE0 ; droite db
```

```
253  
dw COCO_R1  
db 1,0,0  
db 253  
dw COCO_R2  
db 1,0,1  
db 253  
dw COCO_R1  
db 1,0,0  
db 253  
dw COCO_R0  
db 1,0,1  
db 0
```

Dessiner des images déplacées peut être un peu fastidieux. C'est pourquoi, depuis la version **V15** de **SPEDIT**, vous disposez d'un mécanisme permettant de décaler une image d'un demi-octet (2 pixels en mode 1 ou un pixel en mode 0) vers la droite ou vers la gauche. Comme vous pouvez le voir dans le menu SPEDIT, une nouvelle option apparaît : avec les flèches du curseur, vous pouvez décaler l'image que vous avez dessinée.



```
SPEDIT: Sprite Editor v15.0
---- CONTROLES EDICION ----
1,2 : tinta -/+      g:bor: space : mueve u pinta
f: desplaza medio byte
n: flip horizontal
v: flip vertical
c: clear sprite
b: imprime bytes (asm) en printer
i: imprime paleta en printer y file
r: reload 20000
z: x: cambia color de tinta
t: RESET

Antes de empezar puedes cargar un sprite
ensamblandolo en la direccion 20000.
No ensambles ancho y alto,solo los bytes
del dibujo.
la paleta custom esta en la linea 2300
La paleta la puedes cambiar para que sea
la misma que la de tu videojuego.

EDITAR sprite(1) o CAPTURAR sprite(2)? ■
```

Cela nous permet de déplacer facilement une image afin d'appliquer la technique décrite de déplacement en douceur.

14 Jeux à défilement

La bibliothèque 8BP vous permet de faire défiler les images de différentes manières qui peuvent être combinées simultanément, bien que la méthode la plus importante soit basée sur la commande **|MAP2SP**. Les techniques disponibles sont résumées ci-dessous :

- **Utilisation des commandes de déplacement des blocs de sprites** : Une façon simple de faire défiler des images avec le 8BP est de créer des sprites décoratifs dont l'état est réglé pour être déplacé par les commandes **|MOVERALL** et/ou **|IMOVERALL**.
|L'UTILISATEUR A LE DROIT D'ÊTRE INFORMÉ DE L'ÉVOLUTION DE LA SITUATION ET DE L'ÉVOLUTION DE LA SITUATION.
- **Utilisation de |MAP2SP** : L'idée derrière le défilement multidirectionnel fourni par **|MAP2SP** dans 8BP est simple : tous les éléments représentés à l'écran sont des sprites, donc les éléments du monde que nous allons imprimer et déplacer à l'écran sont des sprites dont les images associées seront des montagnes, des maisons, des arbres ou tout ce dont vous avez besoin pour construire votre "monde". Pour sélectionner une partie du monde et la transformer en une liste de sprites, la fonction **|MAP2SP** est utilisée. La fonction **|UMAP** vous permet de mettre à jour le monde avec une partie d'un monde plus grand.
- **En utilisant la commande |STARS, qui** vous permettra de réaliser un défilement multidirectionnel d'une banque de 40 pixels que vous pourrez placer où vous le souhaitez et que vous pourrez déplacer dans différents plans et à différentes vitesses.
- La **commande |RINK** vous permet de faire pivoter un motif d'encre, ce qui donne une impression de mouvement vers l'avant que vous pouvez utiliser dans certains types de défilement, tels que le mouvement d'un sol en briques, de l'eau, etc.

14.1 ETOILES : Parchemin d'étoiles ou terre mouchetée

Dans la bibliothèque 8BP, vous disposez d'une fonction très facile à utiliser pour créer un effet d'arrière-plan d'étoiles en mouvement, donnant la sensation de défilement. Voici la fonction

|STARS. Cette fonction permet de déplacer jusqu'à 40 étoiles simultanément sans altérer vos sprites, comme si elles passaient "en dessous".

**|STARS,<étoile
d'étoiles>,<couleur>,<dy>,<dx>,<dx>.** **initiale>,<nombre**

Vous disposez d'une banque d'étoiles et pouvez combiner plusieurs commandes STARS pour travailler avec des groupes d'étoiles à des vitesses différentes, ce qui donne une impression de plans de différentes profondeurs.

La banque d'étoiles se compose de 40 paires d'octets représentant les coordonnées (y,x). Elles occupent les adresses 42540 à 42619 (80 octets au total). Une façon de générer 40

étoiles aléatoires serait (notez que si nous avons déjà exécuté DEFINT A-Z le nombre 42540 doit être mis en hexadécimal parce qu'il est plus grand que 32768).

```
FOR dir=42540 TO 42618 STEP 2 : POKE dir,RND*200 : POKE  
dir+1,RND*80:NEXT
```

Pour une description détaillée de la voir le chapitre "guide de référence". Dans ce chapitre, vous trouverez différents exemples pour simuler des étoiles, de la terre, des étoiles avec deux plans de profondeur, de la pluie ou même de la neige. Avec de l'imagination, il est probablement possible de simuler plus de choses avec cette même fonction. Par exemple, en plaçant les étoiles par séquences de 2 ou 3 pixels en diagonale, au lieu de les répartir aléatoirement, on peut obtenir un déplacement de type "segment", ce qui pourrait être idéal.

pour simuler la pluie.

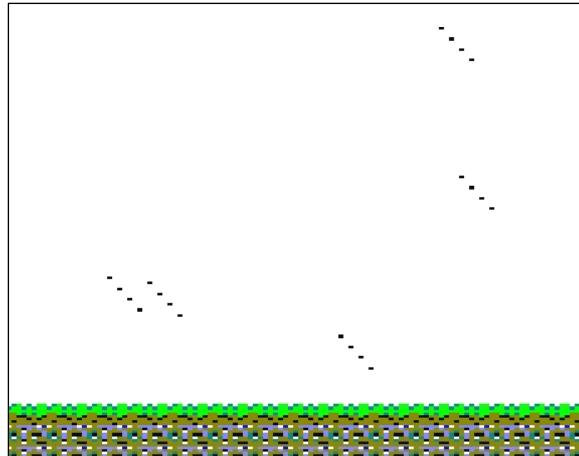


Fig. 76 Effet de pluie avec STARS

```

10 MÉMOIRE 24999
20 CALL &6B78: ' installer RSX
40 mode 0:CALL &BC02 : "restore default palette just in case".
50 bank=42540
60 FOR dir=bank TO bank+40*2 STEP 8 :
70 y=INT(RND*190):x=INT(RND*60)+4
80 POKE dir,y:POKE dir+1,x :
90 POKE dir+2,(y+4):POKE dir+3,x-1
100 POKE dir+4,(y+8):POKE dir+5,x-2
110 POKE dir+6,(y+12):POKE dir+7,x-3
120 NEXT

140 "SCÉNARIO DE PLUIE
141 '-----
150 |SETLIMITS,0,80,50,200 : ' limites de l'écran de jeu
151 grass=&84d0:|SETUPSP,30,9,grass:'La lettre Y est le sprite 31
152 rocks=&84f2:|SETUPSP,21,9,rocks : 'la lettre P est le sprite 21
160
string$="YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY"
170 |LAYOUT,22,0,@cadena$: 'ceci peint l'herbe
180 string$="PPPPPPPPPPPPPPPP"
190 |LAYOUT,23,0,@string$ : 'peint une rangée de rochers
200 |LAYOUT,24,0,@string$: 'peint une autre rangée de pierres
210 '----- cycle de jeu-----
211 defint a-z
220 LOCATE 1,10:PRINT "RAIN DEMO".
221 LOCATE 1,11:PRINT "press ENTER".
230 |ÉTOILES,0,40,4,4,2,-1
240 SI INKEY(18)=0 ALORS 300
250 GOTO 230

```

Comme l'exemple d'un double plan d'étoiles se trouve dans le chapitre de référence de la bibliothèque, nous verrons ici un exemple de vaisseau spatial survolant une planète terrestre tachetée, avec une sensation de défilement vertical.

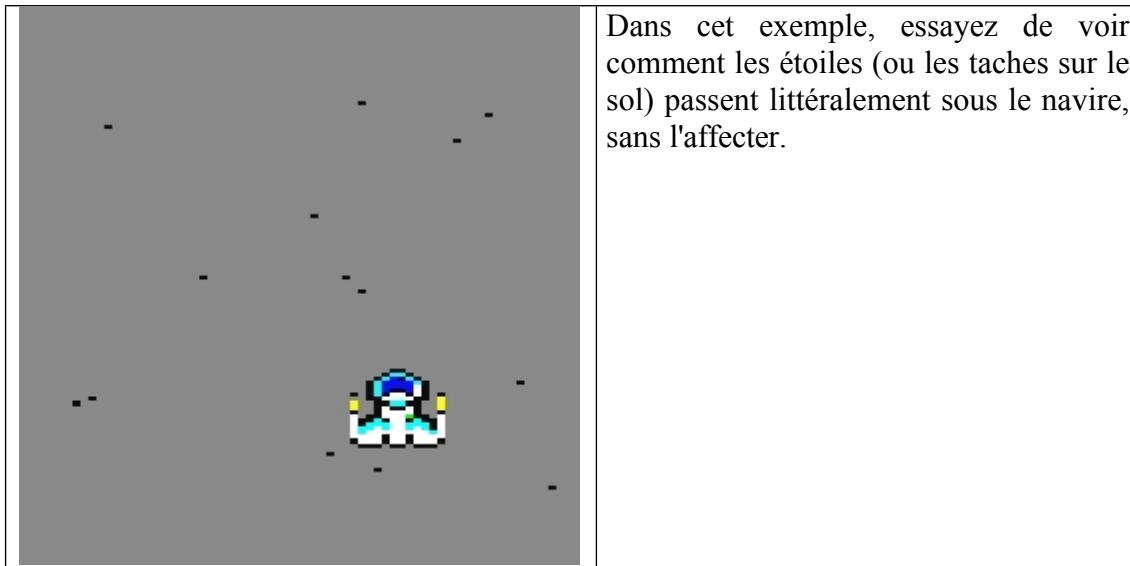


Fig. 77 Effet de sol tacheté avec STARS

Il existe un moyen d'invoquer la commande STARS de manière optimisée et il consiste simplement à l'invoquer la première fois avec des paramètres et les fois suivantes sans paramètres. La commande supposera que les valeurs des paramètres sont les mêmes que celles de la dernière invocation avec paramètres et cela économise le temps que l'interpréteur BASIC passe à traiter les paramètres, jusqu'à 1,7ms.

```

10 MÉMOIRE 24999
11 "J'ai mis des étoiles au hasard
12 FOR dir=42540 TO 42618 STEP 2 : POKE dir,RND*200:POKE dir+1,RND*80:NEXT
20 MODE 0 : DEFINT A-Z : CALL &6B78:' install RSX
25 call &bc02 : "restore default palette just in case".
26 encre 0.13 : "fond gris
30 FOR j=0 TO 31:|SETUPSP,j,0,&X0:NEXT:'réinitialiser les sprites
40 |SETLIMITS,12,80,0,186 : 'limites de l'écran de jeu

41 ' nous allons créer un navire dans le sprite 31
42 |SETUPSP,31,0,&1:' statut
43 ship = &a2f8 : |SETUPSP,31,9,ship:' assigne une image au sprite 31
44 x=40:y=150 : ' coordonnées du navire

49 '----- cycle de jeu-----
50 |STARS,0,20,5,1,0:' étoiles noires sur fond gris
55 gosub 100:' mouvement du navire
60 |PRINTSPALL,0,0
70 goto 50

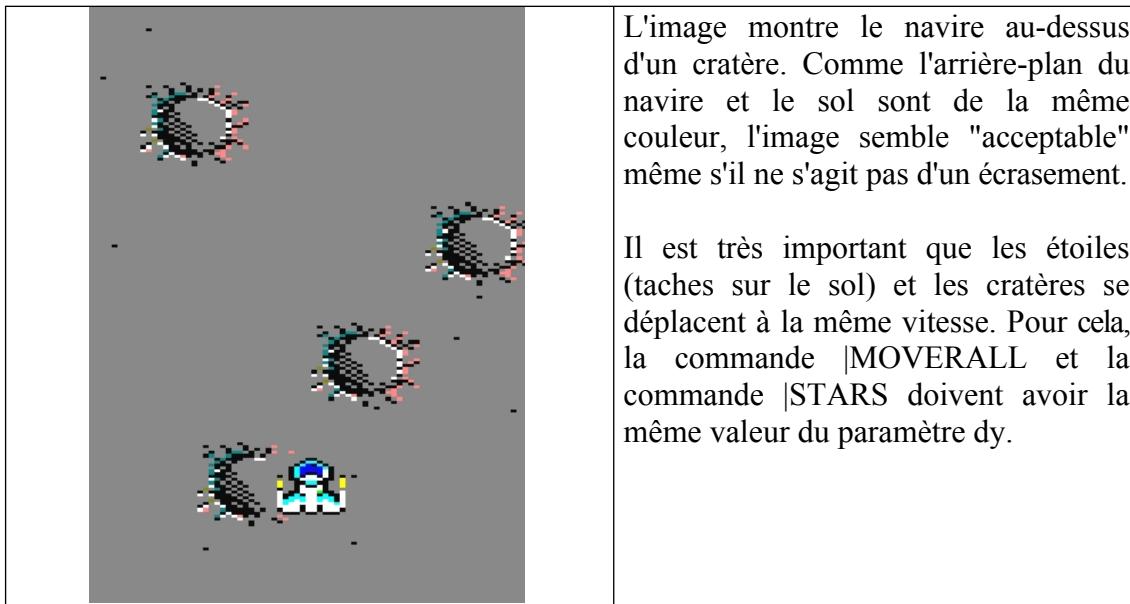
99 ' mouvement de routine navire -----
100 IF INKEY(27)=0 THEN x=x+1:GOTO 120
110 SI INKEY(34)=0 ALORS x=x-1
120 |LOCATESP,31,y,x
130 RETOUR

```

14.2 Défilement à l'aide de MOVERALL et/ou AUTOALL

En combinant l'utilisation du mouvement relatif, du défilement en étoile et de l'ordre d'impression des sprites, nous allons étudier un exemple de simulation d'un vaisseau spatial survolant un paysage lunaire.

Tout d'abord, nous avons choisi le sprite 31 pour notre navire, car il sera imprimé en dernier. Les sprites sont imprimés dans l'ordre, en commençant par zéro et en terminant par 31. Si un cratère est un sprite inférieur à 31, il sera imprimé avant le vaisseau et le vaisseau sera "au-dessus", donnant l'impression qu'il le survole.



L'image montre le navire au-dessus d'un cratère. Comme l'arrière-plan du navire et le sol sont de la même couleur, l'image semble "acceptable" même s'il ne s'agit pas d'un écrasement.

Il est très important que les étoiles (taches sur le sol) et les cratères se déplacent à la même vitesse. Pour cela, la commande |MOVERALL et la commande |STARS doivent avoir la même valeur du paramètre dy.

Fig. 78 : survol de la lune

Il s'agit du code BASIC :

```
10 MÉMOIRE 24999
11 "J'ai mis des étoiles au hasard"
12 FOR dir=42540 TO 42618 STEP 2 : POKE dir,RND*200:POKE
dir+1,RND*80:NEXT
20 MODE 0 : DEFINT A-Z : CALL &6B78:' install RSX
25 call &bc02 : "restore default palette just in case".
26 encre 0.13 : "fond gris"
30 FOR j=0 TO 31:|SETUPSP,j,0,&X0:NEXT:'réinitialiser les sprites
40 |SETLIMITS,12,80,0,186 : ' limites de l'écran de jeu

41 ' créons un navire dans le sprite 31
42 |SETUPSP,31,0,&1:' statut
43 ship = &a2f8 : |SETUPSP,31,9,ship:' assigne une image au sprite 31
45 x=40:y=150 : ' coordonnées du navire

46 ' maintenant les cratères
47 crater=&a39a : cy%=0
48 for i=0 to 3 : |SETUPSP,i,9,crater :
49 |SETUPSP,i,0,&x10001 : ' impression et mouvement relatif
50 x(i)=rnd*40+20:y(i)=i*40
60 |locatesp,i,y(i),x(i)
70 suivant
```

71 t=0

80 '----- cycle de jeu-----

81 |STARS,0,20,5,3,0:' mouvement des étoiles noires

82 gosub 100:' mouvement du navire

83 |MOVERALL,3,0 : "mouvement du cratère".

84 t=t+1 : if t> 10 then t=0:gosub 200:' contrôle du cratère

90 |PRINTSPALL,0,0:' impression du vaisseau et du cratère

91 goto 81

99 ' mouvement de routine navire -----

100 IF INKEY(27)=0 THEN x=x+1:GOTO 120

110 SI INKEY(34)=0 ALORS x=x-1

120 |LOCATESP,31,y,x

130 RETOUR

199 ' surveillance de la rentrée des cratères

200 c=c+1 : si c=6 alors c=0

220 |PEEK,27001+c*16,@cy% 220 |PEEK,27001+c*16,@cy%

230 if cy%>200 then |POKE,27001+c*16,-20

240 retour

Voyons un dernier exemple qui utilise le mouvement relatif pour donner la sensation de défilement, en utilisant des sprites avec des dessins de maisons, un sol tacheté et un personnage situé au centre qui, selon la direction dans laquelle il se déplace, fait bouger tout ce qui est autour de lui. C'est un exemple très basique, mais qui donne une idée du potentiel de ces fonctions - ici, c'est toute la ville qui bouge !



Fig. 79 Tout le village se déplace

10 MÉMOIRE 24999

20 MODE 0 : appel &b78

30 DEFINT a-z

240 INK 0,12

241 bordure 7

250 FOR i=0 TO 31

260 |SETUPSP,i,0,&X0

270 SUIVANT

280 FOR i=0 TO 3

```

|SETUPSP,i,0,&X10001
|SETUPSP,i,9,&A01c:rem maisons
301 |LOCATESP,i,RND*150+50,rnd*60+10
310 SUIVANT
320 |SETUPSP,31,7,6 : caractère rem
330 |LOCATESP,31,90,38
340 |SETUPSP,31,0,0,&X1111
xa=0:ya=0
410 SI INKEY(27)=0 ALORS xa=-1 :
420 SI INKEY(34)=0 ALORS xa=+1 :
430 SI INKEY(67)=0 ALORS ya=+2
440 SI INKEY(69)=0 ALORS ya=-2
450 |MONDIAL,ya,xa
460 |PRINTSPALL,1,0
470 | Les résultats de l'enquête ont été publiés dans le rapport annuel
de la Commission européenne sur la situation des droits de
l'homme.
GOTO 410

```

14.3 Technique du "repérage"

La technique pour peindre les montagnes dans les jeux à défilement horizontal et les lacs dans les jeux à défilement vertical est la même. Ce que nous allons faire, c'est peindre seulement le début de la montagne, en utilisant un sprite pour peindre son côté gauche. Nous en mettrons autant que nous le souhaitons. Dans ce cas, j'en ai mis trois

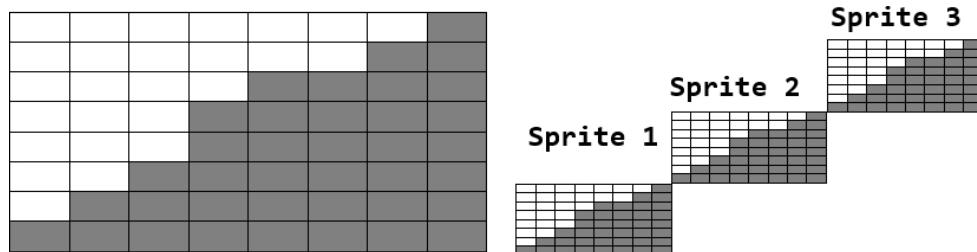


Fig. 80 : Définition de la pente d'une montagne avec plusieurs sprites

Nous faisons de même avec une image miroir que nous associerons à 3 autres sprites, et nous les placerons sur la droite, en construisant le côté droit de la montagne. Assurez-vous que l'image miroir a au moins les deux dernières colonnes de pixels à zéro. Cela lui permettra de s'effacer au fur et à mesure qu'elle se déplace vers la gauche. Notez qu'en 8BP, les sprites se déplacent d'un octet à l'autre (deux pixels à la fois).

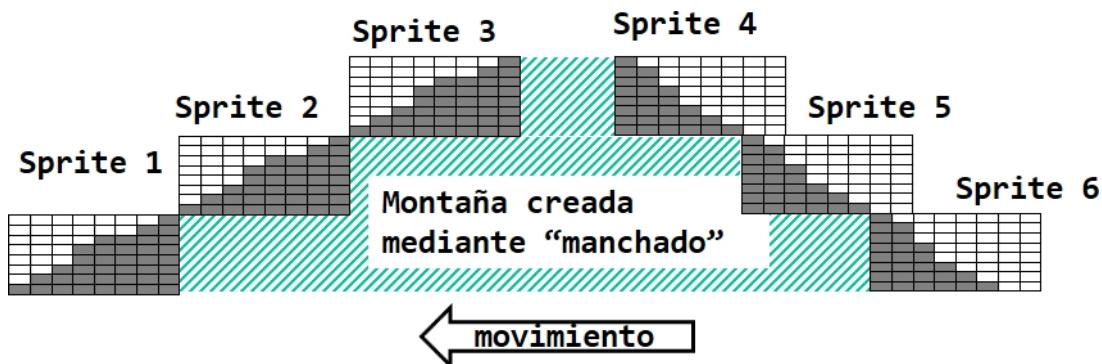


Fig. 81 Montagne créée avec 6 sprites et la technique du "spotting".

En déplaçant tous les sprites vers la gauche par un mouvement automatique ou relatif, les sprites de gauche commenceront à "salir" l'arrière-plan et par conséquent

"En même temps, les sprites de droite commencent à nettoyer la montagne. Si la montagne apparaît lentement en entrant dans l'écran, elle ressemblera à un énorme sprite de montagne, alors qu'il ne s'agit en fait que de 6 petits sprites.

Le jeu vidéo "Nibiru" utilise la technique du "smearing" pour dessiner les montagnes et autres grands éléments, en combinaison avec la commande MAP2SP que nous verrons plus loin.

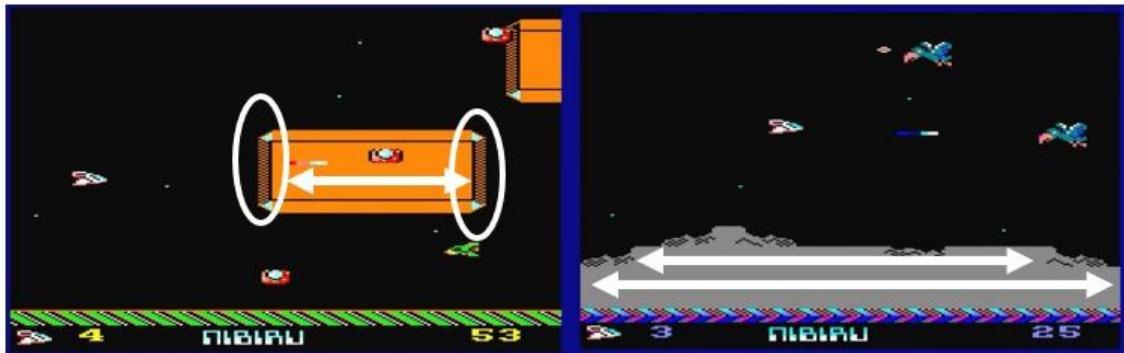


Fig. 82 : Exemples de techniques de coloration

J'ai également utilisé la technique du smudging dans le jeu vidéo "Eridu", où d'immenses montagnes se déplacent doucement à travers l'écran.



Fig. 83 : Technique de coloration dans "Eridu".

Dans le cas d'un jeu à défilement vertical, si nous voulons peindre un lac sur un terrain marron, nous ferons de même, certains sprites "tacheront" le terrain et d'autres plus éloignés le "nettoieront", le faisant ressembler à un immense lac, d'un seul tenant.

Il n'y a qu'une seule précaution à prendre, c'est que les navires ne survolent pas le lac, sinon votre "astuce" sera dévoilée ! Si vous voulez qu'il soit possible de survoler le lac, vous devez utiliser l'écrasement de sprites, comme dans l'étape 2 du jeu vidéo "Nibiru", où votre vaisseau et les vaisseaux ennemis peuvent passer au-dessus de grands rectangles de couleur sans les détruire.

14.4 MAP2SP : Défilement basé sur une carte du monde

Toutes les techniques précédentes sont parfaitement valables pour le défilement, et même compatibles avec ce que nous allons voir maintenant, à savoir la technique fondamentale qui vous permettra de concevoir une "carte du monde" et d'y faire défiler votre personnage ou votre navire, avec une seule ligne de code.

Pour utiliser la commande **|MAP2SP**, il est nécessaire de sélectionner l'"**option d'assemblage**" 2, ce qui nous permet de disposer de 24,8 KB pour le listing BASIC.

L'idée est simple : nous allons créer une liste d'éléments qui composent la carte du monde (jusqu'à 82 éléments que nous appellerons "éléments de la carte"). Chaque élément est décrit par les coordonnées Y,X où il se trouve et l'adresse mémoire où se trouve l'image de l'élément en question (une maison, un arbre, etc.). L'image associée à un élément de la carte peut être de la taille que vous souhaitez. Les coordonnées de chaque élément seront un nombre entier positif, de 0 à 32000.

Une fois la carte créée, nous invoquerons la fonction :

|MAP2SP, Yo, Xo

Cette fonction analyse la liste des éléments du monde et détermine lesquels sont affichés si le monde est visualisé en plaçant le coin inférieur de l'écran aux coordonnées (I, Xo). La fonction transforme les "éléments de la carte" en sprites, occupant les positions dans la table des sprites à partir de zéro. Cela peut consommer beaucoup ou peu de sprites, en fonction de la densité des éléments de la carte que vous avez. Lors d'une invocation ultérieure de la même fonction, les éléments de la carte qui ne sont plus présents dans la scène ne consommeront pas de sprites dans la table, et d'autres éléments de la carte prendront le relais. Cela signifie que **la fonction MAP2SP consomme un nombre variable et indéterminé de sprites, en fonction du nombre d'éléments de carte visibles à l'écran à un moment donné**. Dans l'exemple ci-dessous, vous utiliserez 3 sprites en invoquant MAP2SP aux coordonnées indiquées.

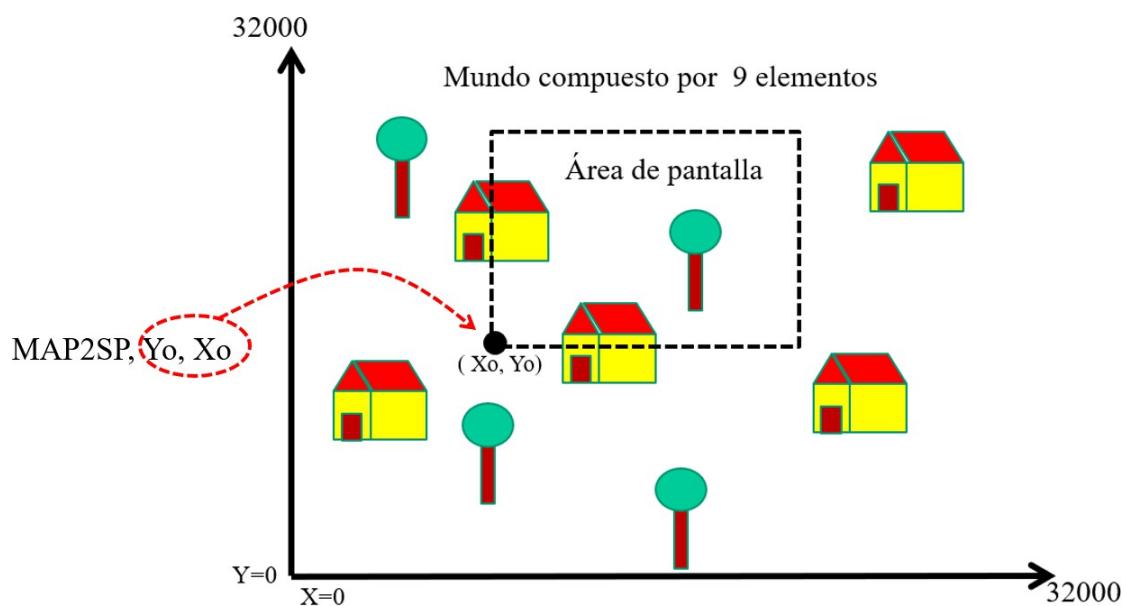


Fig. 84 Carte du monde et MAP2SP

Si vous utilisez ce mécanisme, votre personnage et vos ennemis doivent utiliser les sprites à partir de 31, évitant ainsi les conflits possibles entre les sprites utilisés par le mécanisme de défilement et vos personnages. Si, par hasard, MAP2SP rencontre plus de 32 éléments à traduire en sprites, il ignorera ceux qui dépassent 32.

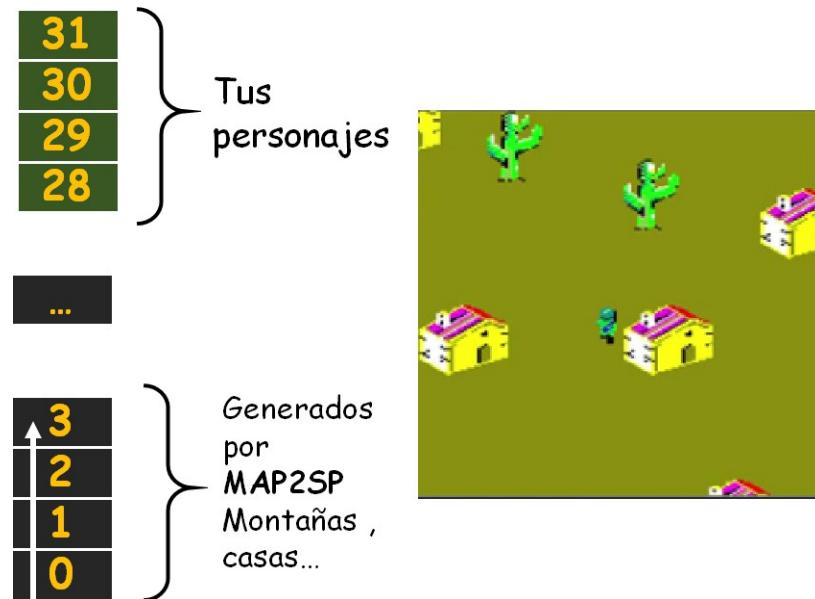


Fig. 85 sprites consommés par MAP2SP

Vous devez invoquer MAP2SP à chaque cycle de jeu ou au moins à chaque fois que vous changez les coordonnées du point de vue à partir duquel vous voulez voir le monde. La fenêtre "coulissante" avec laquelle vous chassez les images qui sont transformées en sprites mesure toujours ce que l'écran mesure (80 octets x 200 lignes), quel que soit le réglage de la commande SETLIMITS. En d'autres termes, **même si SETLIMITS définit une zone de peinture très petite, tout ce que la fenêtre de 80 octets x 200 lignes a "chassé" sera transformé en sprites.**

Les sprites créés par MAP2SP sont créés par défaut avec l'état 3, c'est-à-dire avec le drapeau d'impression actif (PRINTSPALL l'imprime) et avec le drapeau de collision actif (COLSP entre en collision avec lui). Si vous souhaitez que les sprites soient créés dans un autre état, il vous suffit d'invoquer une fois la commande MAP2SP avec un seul paramètre indiquant l'état dans lequel les sprites doivent être créés. Avec une seule invocation de ce type, la commande est configurée pour les invocations ultérieures avec des coordonnées.

|MAP2SP, <status>, <status>, <status>, <status>, <status>, <status>, <status>, <status>.

Exemple :

|**Ceci configure la commande MAP2SP pour qu'elle soit imprimée mais non collimable.**

Après avoir clarifié le concept, examinons en détail comment la carte du monde est spécifiée et un exemple d'utilisation de la fonction MAP2SP.

14.4.1 Carte du monde (Tableau des cartes)

La table dans laquelle nous enregistrerons tous les éléments de la carte s'appelle MAP_TABLE et est spécifiée dans un fichier .asm appelé **map_table_tujuego.asm**.

Cette table contient les éléments qui définissent les images de la carte du monde pour vos jeux à défilement. Le tableau est assemblé à la même adresse mémoire que LAYOUT, c'est-à-dire à l'adresse 42040. Cela signifie que la disposition et la carte du monde ne peuvent pas être utilisées simultanément, mais ce n'est pas un problème puisqu'un jeu à défilement n'utilisera pas la disposition et vice versa. En outre, la restriction porte uniquement sur le fait qu'ils ne peuvent pas être utilisés en même temps, mais un jeu pourrait avoir une phase où il utilise la disposition et une autre où il défile sur la base de la carte du monde.

La table d'entrée de la carte du monde commence par 3 paramètres globaux (occupant 5 octets au total) et une liste d'"éléments de la carte", qui sont décrits par 3 paramètres chacun (x, y, direction de l'image).

La liste peut contenir jusqu'à 82 éléments, mais le nombre d'éléments peut être limité par l'un des paramètres globaux. La liste, au maximum, occupe les 5 octets initiaux + 82 éléments x 6 octets = $5+492=497$ octets. Si on mettait un élément de plus, on dépasserait les 500 octets réservés à la carte (qui sont les mêmes que ceux réservés à la présentation).

Le tableau commence par 3 paramètres :

- Hauteur maximale d'un élément de la carte
- Largeur maximale d'un élément de la carte (à exprimer sous forme de nombre négatif)
- Nombre d'articles (maximum 82)

Les deux premiers paramètres sont importants pour vérifier si un sprite apparaît partiellement à l'écran, car la fonction MAP2SP ne connaît pas la largeur et la hauteur de chaque image. Elle sait seulement où se trouve l'élément de la carte et, en supposant la largeur et la hauteur maximales, elle vérifie si cet élément peut entrer dans l'écran. Si c'est le cas, un sprite est créé à partir de l'élément de la carte. Si ces deux paramètres sont fixés à zéro, le coin supérieur gauche de l'élément de la carte doit se trouver à l'intérieur de l'écran pour que cet élément soit transformé en sprite.

Chaque élément est un tuple de 3 paramètres précédé du mnémonique "DW" :

DW Y, X, <image>

Examinons un exemple de fichier appelé map_table_tujuego.asm

TABLEAU DE LA CARTE

3 paramètres avant la liste des "éléments de la carte".

dw 50 ; hauteur maximale d'un sprite au cas où il passerait par le haut et qu'une partie devrait être peinte.

dw -40 ; largeur maximale d'un sprite dans le cas d'un sprite à gauche (nombre négatif)

db 82 ; nombre d'éléments de la carte. au maximum 82

et à partir de là, les éléments dw

100,10,HOUSE ; 1

dw 50,-10,CACTUS;2

dw 210,0,HOME;3

```

dw 200,20,CACTUS;4
dw 100,40,HOUSE;5
dw 160,60,HOUSE;6
dw 70,70,HOUSE;7
dw 175,40,CACTUS;8
dw 10,50,HOUSE;9
dw 250,50,HOUSE;10
dw 260,70,HOUSE;11
dw 260,70,HOUSE;11
dw 290,60,CACTUS;12
dw 180,90,HOUSE;13
dw 60,100,HOUSE;14
dw 60,100,HOUSE;14

```

...

Pour concevoir votre monde, je vous recommande de prendre un cahier à carreaux et d'y dessiner les éléments que vous souhaitez voir figurer dans votre monde. Chaque carré du carnet peut représenter une quantité fixe, par exemple 8 pixels ou 25 pixels. L'important est que vous preniez le temps de dessiner le monde que vous voulez et la façon dont vous voulez le parcourir. Par exemple, il existe des jeux multidirectionnels de type "Gauntlet" et d'autres à défilement vertical comme Commando. C'est à vous de choisir, mais dans tous les cas, faites-le avec du temps et de la patience et le résultat en vaudra la peine.

Chaque phase de votre jeu peut être une carte. Dans 8BP, vous pouvez changer la carte quand vous le souhaitez en utilisant les fonctions POKE. J'utilise normalement 1KB de l'espace mémoire utilisé par 8BP, par exemple de 23000 à 24000, pour stocker toutes les phases (cartes) du jeu et chaque fois que j'entre dans une phase, je charge à l'adresse 42040 la carte correspondante par PEEKing et POKEing. En d'autres termes, je crée mon fichier de cartes à l'adresse 23000 et il occupe 1 Ko, ce qui laisse 23 Ko pour mon programme BASIC. Pour que ces informations ne soient pas écrasées par le programme BASIC, je dois créer une MEMOIRE 22999 au début du jeu.

14.4.2 Utilisation de la fonction MAP2SP

Voyons maintenant un exemple d'utilisation de cette fonction. Fondamentalement, elle doit être invoquée une fois par cycle de jeu avec les nouvelles coordonnées de l'origine à partir de laquelle le monde est observé.

La fonction créera un nombre variable de sprites à partir du sprite 0 et les créera avec leurs coordonnées d'écran adaptées. En d'autres termes, même si un élément de la carte a une coordonnée $x=100$, si l'origine du mobile est située à la position $x=90$, ce sprite sera créé avec la coordonnée d'écran $x'=x-90=10$. La coordonnée de l'axe Y tient compte du fait que l'axe Y de l'Amstrad croît vers le bas, alors que la carte du monde croît vers le haut. La coordonnée Y est donc adaptée à l'aide de l'équation $Y'= 200-(Y-Y_{\text{orig}})$. Mais ne vous inquiétez pas, cette adaptation est déjà effectuée par la fonction MAP2SP. Il suffit de modifier l'origine du déplacement à partir de laquelle la carte du monde doit être affichée.

Dans ce mini-jeu, un monde de maisons et de cactus a été créé et notre personnage se promène entre les éléments. Dans cet exemple, en cas de collision (détectée avec la fonction **|COLSPALL**), le personnage ne pourra pas continuer. Dans un jeu d'aviation où les

éléments de la carte sont "volables", nous pourrions paramétrer la collision pour qu'elle ne détecte que les collisions avec les ennemis et les tirs, et non avec les éléments de l'arrière-plan, en utilisant **|COLSP, 32,**
<sprite_initial>, <sprite_final>.



Fig. 86 Mini jeu à défilement inspiré de "commando".

Comme vous pouvez le voir, il est très petit, mais il a tout : défilement multidirectionnel, lecture du clavier, changement des séquences d'animation des personnages, détection des collisions, musique...

IMPORTANT : notez la commande MEMORY. Nous avons utilisé l'option d'assemblage 2, idéale pour les jeux à défilement, ce qui nous laisse près de 25 Ko libres.

```

10 MÉMOIRE 24799
20 MODE 0
30 ON BREAK GOSUB 280
40 APPEL &6B78
50 DEFINT a-z
60 INK 0,12
70 POUR y=0 à 400 ÉTAPE 2
80 PLOT 0,y,10:DRAW 78,y
90 PLOT 640-80,y,10:DRAW 640,y
100 SUIVANT
110 x=0:y=0
120 |SETUPSP,31,0,&X100001
130 |SETUPSP,31,7,1:dir=1:' direction initiale vers le haut
140 |locationsp,31,100,36
150 |MUSIQUE,0,1,5
160 |SETLIMITS,10,70,0,199 : |PRINTSPALL,0,1,0
170 col%=32:sp%=32:|COLSPALL,@sp%,@col%
180 |COLSP, 34, 0, 0, 0 : collision REM dès qu'il y a un chevauchement minimum
190 "commence le cycle de jeu
200 SI INKEY(27)=0 ALORS x=x+1:SI dir<>>3 ALORS dir=3:|SETUPSP,31,7,3 :
GOTO 220
210 IF INKEY(34)=0 ALORS x=x-1:IF x<0 ALORS x=0:ELSE IF dir<>>4 ALORS
dir=4:|SETUPSP,31,7,4
220 SI INKEY(67)=0 ALORS y=y+2:IF x=xa AND dir <> 1 ALORS
dir=1:|SETUPSP,31,7,1 : GOTO 240
230 IF INKEY(69)=0 THEN y=y-2:IF y<0 THEN y=0:ELSE IF x=xa AND dir <>>2
THEN dir=2:|SETUPSP,31,7,2 :
240 SI xa=x ET ya=y ALORS dir=0 ELSE |ANIMA,31
250 |MAP2SP,y,x:|COLSPALL : IF col<32 THEN x=xa:y=ya:|MAP2SP,y,x ELSE
xa=x:ya=y
260 |PRINTSPALL

```

270 GOTO 200
280 |MUSIQUE:MODE 1 : ENCRE 0,0:STYLO 1

Examinons maintenant un autre exemple de défilement horizontal, où un effet intéressant de carte du monde "infinie" a été obtenu, rendant la fin de la carte égale au début et provoquant un saut abrupt lorsque Xo atteint une certaine valeur. En fait, cette carte du monde ne comporte que 13 éléments.

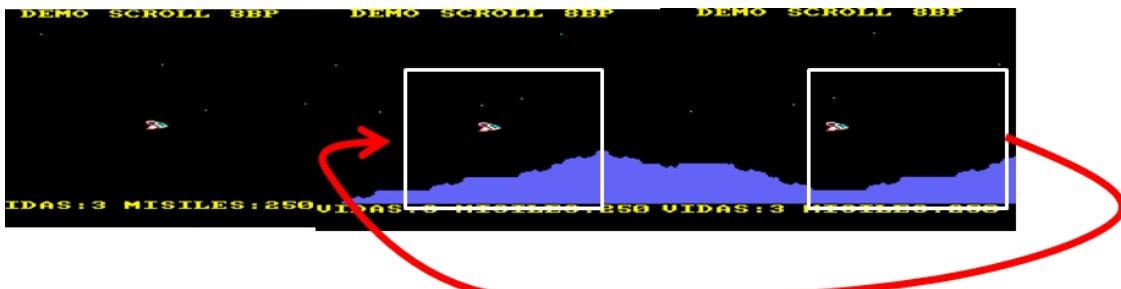


Fig. 87 Carte du monde "infini"

Voici la carte utilisée

TABLEAU_MAP

3 paramètres avant la liste des "éléments de la carte".
dw 50 ; hauteur maximale d'un sprite au cas où il passerait par le haut et qu'une partie devrait être peinte.
dw -18 ; largeur maximale d'un élément de la carte. doit être exprimée sous la forme d'un nombre négatif.
db 13 ; nombre d'éléments de la carte. il devrait être de 82 au maximum

et à partir de là, les articles dw

```
36,80,MONTUP ; 1
dw 48,100,MONTUP;2
dw 60,120,MONTUP;3
dw 72,130,MONTUP;4
dw 72,140,MONTDW;5
dw 60,160,MONTH;6
dw 60,180,MONTDW;7
dw 48,190,MONTDW;8
dw 48,190,MONTDW;8
```

Ici, je répète les éléments pour qu'ils correspondent à la position 100 dw 48,210,MONTUP;9

```
dw 60,230,MONTUP;10
dw 72,240,MONTUP;11
dw 72,250,MONTDW;12
dw 60,270,MONTH;13
dw 60,270,MONTH;13
```

Et voici le programme BASIC, où j'ai mis en évidence la ligne où le monde recule sans que le joueur s'en aperçoive.

10 MÉMOIRE 24799
11 FOR dir=42540 TO 42618 STEP 2 : POKE dir,20+RND*110:POKE
dir+1,RND*80:NEXT
20 MODE 0

```

30 ON BREAK GOSUB 280
40 APPEL &6B78
50 DEFINT a-z
51 INK 0,0
52 |MUSIQUE,0,0,0,5
110 xo=0:yo=0
111 x=36:y=100
120 |SETUPSP,31,0,&X100001
130 |SETUPSP,31,7,1:dir=1:' direction initiale vers le haut
140 |LOCATESP,31,y,x
160 |SETLIMITS,0,80,0,176 : |PRINTSPALL,0,1,0
161 LOCATE 1,23 :PEN 1 : PRINT "LIVES:3 MISSILES:250" PRINT "LIVES:3
MISSILES:250" PRINT "LIVES:3 MISSILES:250" PRINT "LIVES:3 MISSILES:250"
PRINT "LIVES:3 MISSILES:250
162 LOCATE 1,1:PRINT "    DEMO SCROLL 8BP"
170 col%=32:sp%=32:|COLSPALL,@sp%,@col%
180 |COLSP, 34, 0, 0, 0 : collision REM dès qu'il y a un chevauchement minimum
190 "commence le cycle de jeu
200 SI INKEY(27)=0 ALORS x=x+1 : GOTO 220
210 SI INKEY(34)=0 ALORS x=x-1:SI x<0 ALORS x=0
220 SI INKEY(69)=0 ALORS y=y+2 : GOTO 240
230 SI INKEY(67)=0 ALORS y=y-2:SI y<0 ALORS y=0
240 SI xa=x ET ya=y ALORS dir=0 ELSE |ANIMA,31
250 |MAP2SP,yo,xo:|COLSPALL:IF col<32 THEN END END
260 |PRINTSPALL
261 cycle=cycle +1 : SI cycle=2 ALORS |STARS,0,5,2,0,-1:ciclo=0
262 xo=xo+1:SI xo=210 ALORS xo=100
263 |LOCATESP,31,y,x
270 GOTO 200
280 |MUSIQUE:MODE 1 : ENCRE 0,0:STYLO 1

```

14.4.3 Exemple de fichier de phase

Si vous voulez avoir plusieurs étapes dans un jeu à défilement, comme je l'ai dit précédemment, vous pouvez les précharger dans une zone de mémoire. Par exemple, vous pouvez assembler les scènes à l'adresse 23000 et vous disposez ensuite de 1000 octets pour stocker plusieurs cartes du monde, puisque le 8BP commence à l'adresse 24000. Dans ce cas, votre jeu devra commencer avec une MEMOIRE 22999.

Le jeu vidéo "**Nibiru**" fait de même, bien qu'il ait été créé lorsque 8BP a commencé à l'adresse 26000 (il a été créé avec 8BP v26), de sorte qu'il stocke la carte à partir de 25000. Pour charger une phase, il suffit de lire la zone où chaque phase a été assemblée et de l'écrire à l'adresse où la table du monde doit se trouver avant de commencer à jouer dans cette phase. Ces trois lignes BASIC montrent comment copier la phase 1 du jeu (&61a8 = 25000)

```

310 ' pokes from the world map
320 dirmap!=42040:FOR i!=&61A8 TO &620D
330 dato=PEEK(i !):POKE dirmap !,dato:dirmap!=dirmap!+1
340 SUIVANT

```

Il existe un moyen plus rapide de charger les phases, en utilisant la commande |UMAP que j'expliquerai plus loin, mais cette boucle FOR avec POKES est parfaitement valable.

Enfin, je vous montre le fichier des phases du jeu "Nibiru", que nous assemblons à l'adresse 25000 (rappelez-vous que la version de 8bp avec laquelle "Nibiru" a été créé commençait à 26000, donc de 25000 à 26000 il y avait 1000 octets libres. Avec la version actuelle de 8BP, nous assemblerions les phases sans atteindre l'adresse 24800).

```
org 25000
PHASE1
;=====
DÉBUT DE LA PHASE 1
3 paramètres avant la liste des "éléments de la carte".
dw 50 ; hauteur maximale d'un sprite au cas où il passerait par le haut et qu'une partie devrait être peinte.
dw -18 ; largeur maximale d'un élément de la carte. doit être exprimée sous la forme d'un nombre négatif.
db 16 ; num items dw
36,82,MONTUP ; 1 dw
48,104,MONTUP;2 dw
60,126,MONTUP;3 dw
72,138,MONTUP;4 dw
72,150,MONTDW;5 dw
60,172,MONTH;6 dw
60,194,MONTDW;7 dw
48,206,MONTDW;8 dw
60,172,MONTH;6 dw
60 , 194,MONTDW;7
dw 48,206,MONTDW;8
Ici, je répète les éléments pour qu'ils correspondent à la position 100
dw 48,228,MONTUP;9
dw 60,250,MONTUP;10
dw 72,262,MONTUP;11
dw 72,274,MONTDW;12
dw 60,296,MONTH;13
dw 60,320,MONTDW;14
dw 48,350,MONTDW;15
dw 36,380,MONTDW;16
dw 36,380,MONTDW;16
FIN_PHASE1
;=====
PHASE2
;=====

DÉBUT DE LA PHASE 2
dw 50 ; hauteur maximale d'un sprite au cas où il passerait par le haut et qu'une partie devrait être peinte.
dw -6 ; largeur maximale d'un élément de la carte. doit être exprimée en nombre négatif
db 15
dw 128,80,PLACA2_L_OV
dw 128,110,PLACA2_R_OV
dw 192,116,PLACA2_L_OV
dw 192,126,PLACA2_R_OV
dw 92,130,PLACA_L_OV dw
92,150,PLACA_R_OV dw
124,151,PLACA_L_OV dw
124,171,PLACA2_R_OV dw
128,200,PLACA2_L_OV dw
128,210,PLACA2_R_OV dw
92,220,PLACA2_L_OV dw
92,230,PLACA2_R_OV dw
164,240,PLACA2_L_OV dw
164,260,PLACA2_R_OV dw
156,254,PLACA2_R_OV dw
156,254,CUPULA2_OV
```

```

FIN_PHASE2
=====
PHASE3
=====
START_PHASE3
dw 50 ; hauteur maximale d'un sprite au cas où il passerait par le haut et qu'une
partie devrait être peinte.
dw -80 ; largeur maximale d'un élément de la carte. doit être exprimée sous la forme
d'un nombre négatif.
db 4
dw 40,0,MAR ; 1
dw 40,80,SEA ; 2
dw 189,0,CLOUDS ; 2
dw 189,80,CLOUDS ; 2
FIN_PHASE3

```

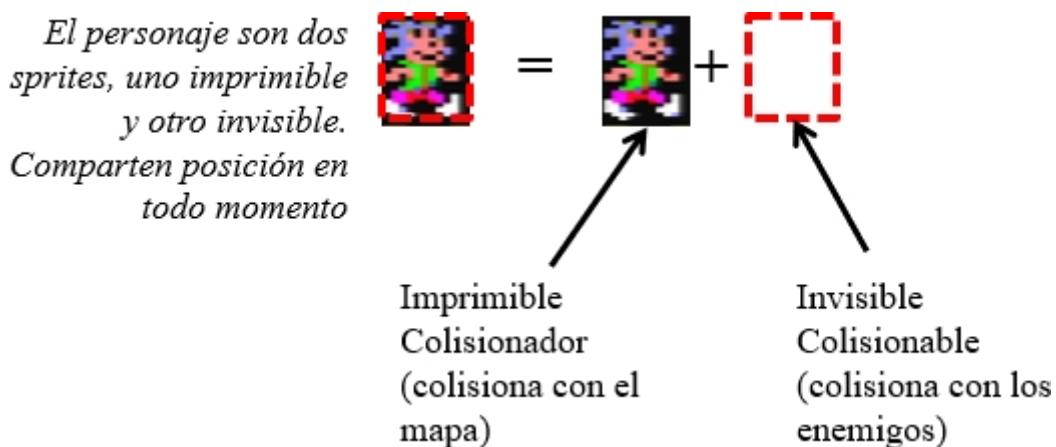
14.4.4 Collision de l'ennemi avec la carte

Vous pouvez vouloir faire un jeu où votre personnage entre en collision avec la carte (en utilisant COLSPALL) et est donc un collisionneur. Disons que vous avez ceci :

- Votre personnage : collider
- Éléments de la carte : colliderable
- Votre tir : collider
- Ennemis : colliderable

Si vos ennemis sont des collisionneurs, ils ne peuvent pas entrer en collision avec la carte, et vous souhaitez peut-être créer un jeu de gauntlet, par exemple. La solution est de faire en sorte que les ennemis soient des collisionneurs. Mais bien sûr, dans ce cas, ils ne peuvent plus vous tuer. Et il y a aussi un problème avec les tirs qui sont des colliders.

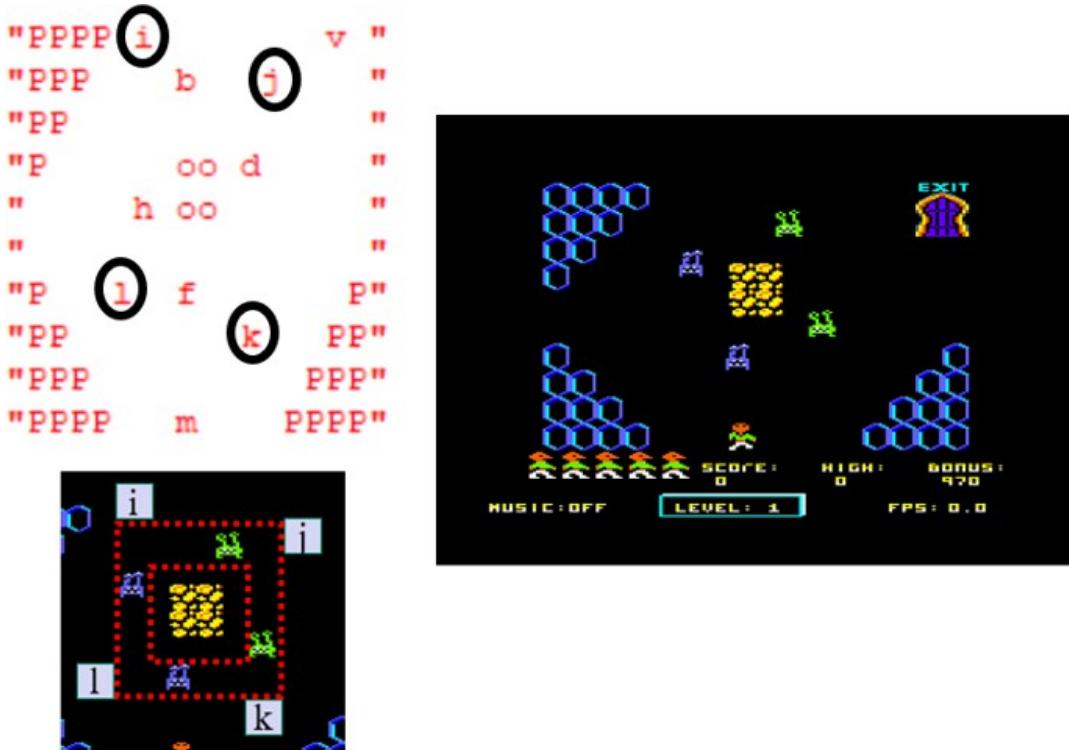
La solution à ce problème repose sur une astuce simple basée sur des sprites "invisibles" non imprimables, qui ne dépendent pas de CPU pour être imprimés mais qui sont bien là. **Votre personnage utilisera deux sprites** : un sprite de collision imprimable et un sprite de collision non imprimable, mais de la même taille. Nous faisons de même avec les tirs, afin qu'ils puissent entrer en collision avec les ennemis et la carte.



Une astuce similaire est utilisée dans le jeu "**Eternal Frogger**". En utilisant des sprites invisibles, la grenouille est tuée en tombant dans la rivière (voir la section expliquant la commande COLSPALL). Cette astuce simple peut également être appliquée aux tirs du

jeu

personnage. Les sprites "invisibles" sont très utiles. Dans le jeu "Happy Monty", ils sont utilisés pour faire changer les ennemis de direction, de sorte que lorsqu'un ennemi entre en collision avec un sprite invisible, il change de direction. Cela signifie qu'il n'était pas nécessaire de définir de nombreux parcours adaptés à chaque écran, mais seulement de placer dans chaque niveau une série de sprites invisibles qui permettaient de modifier les trajectoires des ennemis (voir le document "making off" de **Happy Monty**).



14.4.5 Images d'arrière-plan dans votre défilement

Les images d'arrière-plan sont destinées aux jeux à défilement et sont une fonctionnalité fournie par 8BP à partir de la version V42. Il s'agit d'un type d'impression transparente dans lequel les sprites qui composent l'arrière-plan (la carte du monde) peuvent être imprimés sous votre personnage et vos ennemis sans provoquer de scintillement.

Les images d'arrière-plan, lorsqu'elles sont affectées à un Sprite, sont imprimées avec cette transparence spéciale, que le Sprite ait ou non le drapeau de transparence affecté à son octet de statut. Voir le chapitre 8 pour savoir comment les utiliser.

IMPORTANT : Sur la carte du monde, vous pouvez combiner des images normales avec des "images d'arrière-plan" (section 8.5). Les images d'arrière-plan sont toujours transparentes. Le drapeau de transparence que vous utilisez dans **|MAP2SP, <status>** ne s'appliquera qu'aux images normales.

IMPORTANT : les images d'arrière-plan sont chères à imprimer, et si vous les retournez, elles sont encore plus chères. Si votre jeu dispose d'un défilement, essayez de l'utiliser pour les éléments que votre personnage ou vos ennemis survoleront. Par exemple, pour accélérer le défilement, utilisez

des images normales de maisons ou de rochers apparaissant sur les côtés de votre rouleau, qui ne seront pas souvent chevauchées par les sprites

14.5 Défilement parallaxe

Voyons comment réaliser un défilement parallaxe, c'est-à-dire avec plusieurs "plans" à des vitesses différentes. Peut-être pouvez-vous imaginer une autre façon de faire, celle-ci est juste une façon possible, utilisée dans le jeu "**Nibiru**".

La première chose à savoir est qu'il est beaucoup plus rapide d'imprimer un très long sprite horizontal que plusieurs petits sprites. Ceci est dû aux opérations qui doivent être effectuées après avoir peint chaque ligne de balayage d'un sprite. Pour la même raison, il est beaucoup plus rapide d'imprimer un très grand sprite horizontal qu'un sprite vertical de même taille.

Nous placerons deux sprites géants sur la carte du monde pour créer l'eau. J'en ai fait un de 160 x 8 donc j'en ai mis deux pour que lorsqu'il défile, le suivant commence à apparaître. Lorsque la fonction MAP2SP fait le tour complet de l'écran, elle revient à $x=0$ et tout se répète indéfiniment. Sur la carte du monde, j'ai également mis deux sprites pour les nuages.

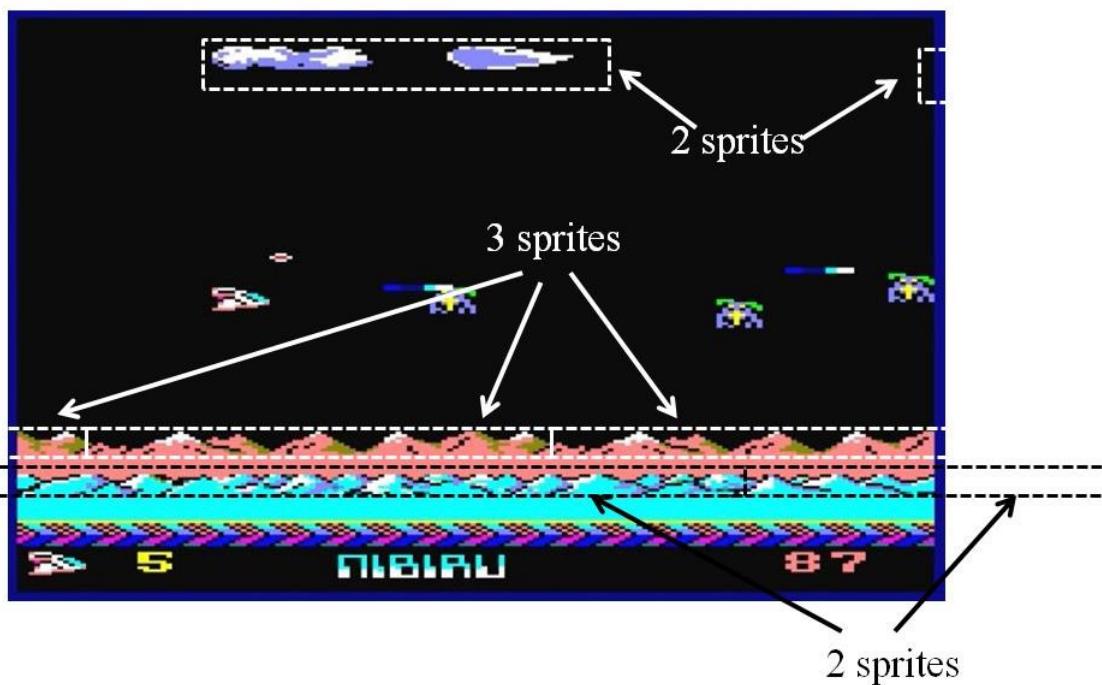


Fig. 88 Défilement parallaxe

Pour les montagnes, j'ai utilisé 3 sprites normaux, en dehors de la carte du monde. Je leur ai donné un mouvement automatique sur leur drapeau d'état et je les ai fait se déplacer vers la gauche. Mais lors des cycles impairs, je désactive à la fois le drapeau d'impression et le drapeau de mouvement automatique, de sorte qu'ils ne se déplacent et n'impriment qu'un cycle de jeu sur deux, atteignant ainsi une vitesse deux fois moindre que celle de l'eau. Les sprites des montagnes sont 16, 17 et 18 et avec ces pokes j'agis sur leur byte d'état.

**mc=cycle AND 1 : IF mc THEN POKE 27256,0 : POKE 27272,0 : POKE 27288,0
ELSE POKE 27256,11 : POKE 27272,11 : POKE 27288,11**

Dans l'exemple, "mc" est la variable qui détermine si le cycle est pair ou impair.

14.6 Mise à jour dynamique de la carte : |UMAP

Peut-être que dans notre jeu, nous avons besoin d'une carte avec plus de 82 éléments. Ou bien nous voulons simplement que la commande `|MAP2SP` s'exécute plus rapidement en utilisant une carte plus petite que nous mettons à jour périodiquement. Ou bien nous voulons les deux !

Pour ce faire, depuis la version 32 du 8BP, il existe la commande **|UMAP** (abréviation de "UPDATE MAP"). Cette commande met à jour la carte avec des informations situées dans une autre zone de mémoire où nous disposons d'une carte plus grande. La commande entraîne la reconstruction complète de la carte, en incluant uniquement les éléments qui répondent à certaines plages de coordonnées X, Y (tous les paramètres sont des nombres de 16 bits).

L'UMAP n'est pas une simple copie d'éléments. Il s'agit d'une copie "sélective". Par exemple, si nous avons une carte située à l'adresse 22500 qui occupe 1500 octets et que nous voulons mettre à jour la carte avec les coordonnées de notre personnage, avec une marge suffisante pour avancer dans la coordonnée Y jusqu'à 100 lignes et dans la coordonnée x jusqu'à 20 octets dans toutes les directions :

| UMAP, 22500,23999, y-100, y+100, x-20, x+20

Cette commande vérifiera les coordonnées des éléments situés sur la carte à l'adresse 22500 et, s'ils se trouvent dans les marges X, Y que nous avons définies, ils seront copiés dans la zone de mémoire que le 8BP utilise pour la commande |MAP2SP, c'est-à-dire qu'il les copiera à partir de l'adresse 42040. Cependant, il ne copiera que ceux qui remplissent la condition. Comme il y a moins d'éléments, la commande |MAP2SP s'exécutera plus rapidement car elle devra lire et vérifier s'il y a moins d'éléments à l'écran.

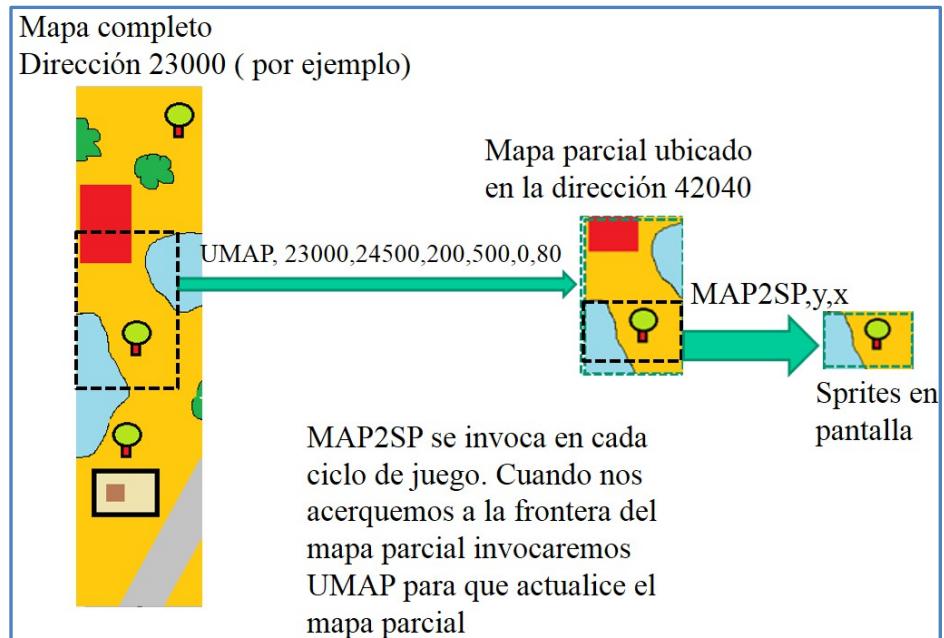


Fig. 89 UMAP

Périodiquement (pas à chaque cycle de jeu), nous pouvons mettre à jour la carte avec UMAP et ainsi créer de très grands mondes avec de nombreux éléments tout en augmentant la vitesse de MAP2SP. La commande UMAP est très rapide, mais l'invoquer à chaque cycle de jeu n'a pas de sens car MAP2SP peut travailler avec une carte beaucoup plus grande que celle qui tient sur l'écran et nous pouvons invoquer UMAP uniquement lorsque nous avons besoin de zones de la carte originale qui ne sont pas présentes dans la carte partielle. Je l'ai utilisé dans le jeu de course automobile "**3D Racing one**", car la carte de la piste était très grande (plus de 82 éléments) et ce que j'ai fait, c'est d'invoquer l'UMAP périodiquement au fur et à mesure de la progression de la voiture.

Dans l'adresse de la carte complète (dans l'exemple 22500), nous n'aurons qu'une liste d'éléments. **Nous n'aurons pas les 3 données initiales** contenues dans la carte 42040 (je veux dire la hauteur maximale d'un élément de la carte, la largeur maximale d'un élément de la carte et le nombre d'éléments). Le nombre d'éléments est mis à jour par **|UMAP** (en fonction du nombre d'éléments respectant les marges imposées). Les deux autres paramètres sont définis par vous dans le fichier "map_table_votre_jeu.asm".

La commande **|UMAP** place les éléments dans la carte partielle dans un ordre destiné à trier la carte partielle en fonction de la coordonnée Y de l'écran. Normalement, vous éditez votre carte globale dans un fichier appelé "misupermapa.asm" ou quelque chose comme ça. Et vous l'assemblez dans le 22500 (par exemple). Dans ce fichier, vous écrirez un par un les éléments de votre carte, dans l'ordre croissant de la coordonnée Y. Pour obtenir les sprites déjà triés par coordonnées Y (dans l'ordre croissant des coordonnées de l'écran), la commande **|UMAP** les lit de la fin vers le début. De cette façon, les sprites générés plus tard avec **|MAP2SP** seront triés par coordonnées Y de l'écran. Rappelez-vous que l'écran utilise un système de coordonnées inversé par rapport à la carte, c'est-à-dire que la 150ème coordonnée de l'écran est la 50ème coordonnée de la carte (dans le cas de **|MAP2SP,0,0**). Si vous ne comprenez pas très bien cela, ne vous inquiétez pas. Il s'agit d'une partie du fonctionnement interne de **|UMAP** et de **|MAP2SP** afin de le rendre plus efficace. Dans la figure suivante, j'ai représenté la carte et l'écran du CPC. Comme vous pouvez le voir

la carte peut être très grande, mais ses coordonnées augmentent alors que celles de l'écran diminuent.

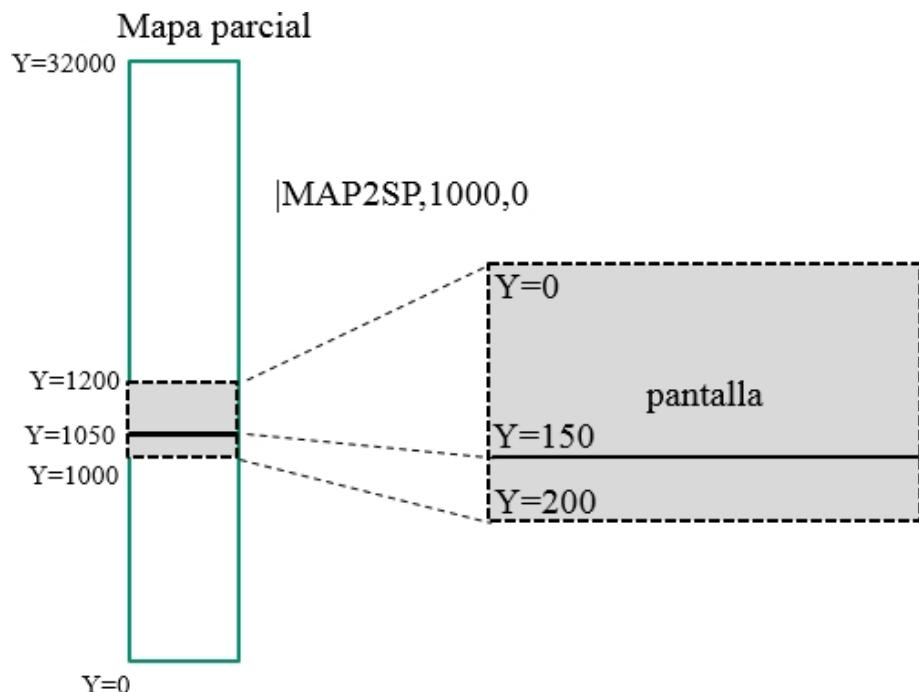


Fig. 90 MAP et affichage

14.7 Animation et défilement de l'encre : commande RINK

Certains jeux nécessitent le déplacement de grands blocs de mémoire d'écran pour donner une impression de mouvement, comme les barres latérales dans les jeux de course ou de grands blocs de briques ou de terre. La commande MAP2SP vous permet de faire tout cela, mais la vitesse n'est pas fulgurante car elle utilise beaucoup de CPU pour déplacer les sprites. L'animation par teinture est le complément parfait dans ces cas.

Sur des ordinateurs comme l'AMSTRAD avec sa puissante palette de 16 couleurs simultanées, de nombreux jeux font appel à l'animation d'encre. Un exemple clair est celui de certains jeux de voiture dont les bandes routières se prêtent à ce type d'animation.

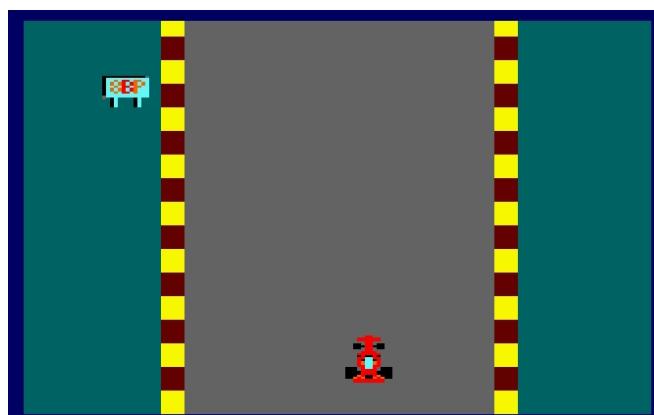


Fig. 91 Bandes animées à l'encre

L'animation par encres consiste à définir un ensemble d'encres sur lesquelles on fera tourner un ensemble de couleurs. Voyons un exemple avec des bandes blanches/grises et 8 encres :

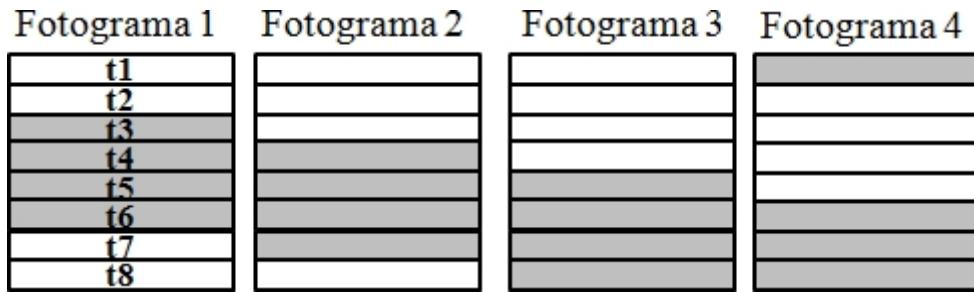


Fig. 92 Animation de l'encre

En principe, pour donner une impression de mouvement, la première chose à faire est d'attribuer les couleurs aux encres qui vont tourner. Dans le cas présent, il s'agit des couleurs blanc (26) et gris (26) et gris (26).

(13) sont attribuées aux encres t1..t8. Supposons que l'encre t1 soit 8, l'encre t8 sera donc 15. Les autres encres (0 à 7) seront utilisées pour les sprites. A chaque instant, il faudra réaffecter les valeurs des 8 encres pour donner la sensation de rotation. C'est ce que fait la commande |RINK (abréviation de "Rotate INK").

RINK vous permet de définir un modèle de couleurs à faire tourner sur un ensemble d'encres. Une encre n'est pas une couleur. Une encre est un identifiant dans l'intervalle [0..15] qui identifie une couleur dans l'intervalle [0..26]. Pour définir le motif de couleurs à faire tourner, nous utiliserons la commande RINK comme suit :

RINK, <initial_ink>, <colour1>,<colour2>, <colour3>, ... ,<colourN>, <colourN>, ... ,<colourN>, <colourN>, <colourN>, <colourN>.

Cela indique qu'ils feront tourner N encres en commençant par l'encre initiale et en utilisant le modèle de couleur indiqué. Notez que si vous utilisez beaucoup d'encres pour réaliser une animation, il vous restera moins d'encres pour vos sprites.

Une fois le motif établi, on peut faire tourner les encres plus ou moins rapidement avec

RINK, <pas>

La valeur du pas est le nombre de déplacements que les encres du pochoir vont subir et une valeur plus élevée donne donc un effet de déplacement plus rapide.

Recommandation : en raison de l'utilisation des interruptions RINK, RINK "bégaié" parfois lorsqu'il est utilisé en même temps que la commande |MUSIC à la vitesse 6. Si vous souhaitez utiliser les deux en même temps sans interférence, utilisez une autre vitesse pour la musique (vous pouvez utiliser la vitesse 5 ou 7, les deux fonctionneront parfaitement).

14.7.1 Course de voitures en 2D

L'exemple du jeu de voiture se trouve dans la liste suivante. Le son du moteur est destiné à donner une sensation d'accélération. Pour les panneaux latéraux, un sprite de mouvement relatif a été utilisé, qui se déplace à la même vitesse que le pas des bandes.

Le motif de l'encre est défini à la ligne 100.

| RINK,1,3,3,3,3,3,24,24,24,24 : enlever les bandes jaunes et rouges

```
1 MEMOIRE 24999
2 APPEL &6B78
3 FOR i=0 TO 31:|SETUPSP,i,0,0:NEXT:|AUTOALL,0:|PRINTSPALL,0,0,0
4 |SETUPSP,31,9,16 : |SETUPSP,31,0,1 : vy=0
10 MODE 0
20 DEFINT a-z
31 |LOCATESP,31,160,40 : x=40
32 |SETUPSP,30,9,17 : |SETUPSP,30,0,17:|LOCATESP,30,-20,10
40 APPELER &BC02 : "palette par défaut
50 GOSUB 430
60 INK 0,13
70 INK 14,10
80 linestinta=3
90 rangotintas=8
91 ' établissement du motif de l'encre
100 |RINK,1,3,3,3,3,3,24,24,24,24,24,24 : enlever les bandes jaunes et rouges
101 |RINK,0
110 y=400
120 ' PAINT ROAD -----
121 tini=1
130 POUR strips=1 A 10
140 FOR t=tini TO rangotintas+tini-1
150 FOR j=1 TO linestinta
160 PLOT 0,y,14:DRAW 136,y
170 PLOT 140,y,t:DRAW 160,y
180 PLOT 480,y,t:DRAW 500,y
190 PLOT 504,y,14:DRAW 640,y
200 y=y-2
210 NEXT j
220 SUIVANT
240 bandes NEXT
250 skipb=-16:xc=65 : cosa=0
270 REM cycle de jeu -----
293 SI saltob=-16 ALORS 296
294 IF jumpb>0 THEN thing=-jump ELSE chose=chose-1
295 IF thing<0 THEN |RINK,thing:vy=3*thing:posv=posv-3*thing:|MOVERALL,-vy,0:IF
saltob <=0 THEN thing=2-saltob
296 |PRINTSPALL
351 cycle=cycle+1:SI posv>240 ALORS posv=-30:|LOCATESP,30,posv,xc:SI xc=10
ALORS xc=65 ELSE xc=10
361 SI INKEY(27)=0 ALORS SI x<52 ALORS x=x+1:POKE 27499,x:GOTO 370
362 SI INKEY(34)=0 ALORS SI x>21 ALORS x=x-1:POKE 27499,x
370 SI INKEY(67)=0 ALORS SI jumpb<16 ALORS jumpb=jumpb+1:jump=jumpb/4
380 SI INKEY(69)=0 ALORS SI jumpb>-16 ALORS jumpb=jumpb-1:jump=jumpb/4
390 SON 1 ,6000/(saut+17),1,15
      GOTO 270
421 REM PALETA
430 INK 0 ,
440 INK 1 , 5
450 INK ,
460 INK ,
470 INK , 26
      INK 5 , 0
490 INK ,
500 INK , 8
510 INK 8 , 10
```

```

520 INK  9 , 12
530 INK  10 ,
      INK
      ,
550 INK  , 0
560 INK  , 23
570 INK  , 0
580 INK  ,
590 RETOUR

```

14.7.2 Parchemin de brique

Dans cet exemple, nous allons combiner l'utilisation de l'animation à l'encre avec le défilement basé sur **|MAP2SP**. En utilisant l'animation à l'encre, nous déplacerons un motif de briques que nous avons dessiné avec un sprite répétitif tandis que le château et l'arbre seront déplacés par **|MAP2SP**.

Le déplacement des briques serait un travail énorme s'il était effectué par l'unité centrale, et cette technique permet donc de réaliser l'"impossible". C'est une technique très puissante si vous l'utilisez avec ingéniosité.



Fig. 93 Défilement utilisant simultanément l'animation d'encre et MAP2SP

La brique utilisée est en fait un sprite à 8 couleurs, dont le dessin est illustré ci-dessous :

0							
8	9	10	11	12	13	14	15

Et le motif d'encre est 0,6,3,3,3,3,3,3,3,3,3,3,3,3,3. Pour le créer, il suffit d'exécuter la commande :

|RINK,8,0,6,3,3,3,3,3,3,3,3,3,3,3,3,3

Le personnage (sprite 31) reste au centre de l'écran et peut sauter et se déplacer dans les deux sens. Pour synchroniser le mouvement des encres et de MAP2SP, un pas=2 est défini pour la commande **|RINK**, car un octet correspond à deux pixels et MAP2SP se déplace au niveau de l'octet.

Il est intéressant de noter que l'oiseau (sprite 30) doit également être affecté par le mouvement puisque sa vitesse doit être ajoutée ou soustraite à celle du personnage.

En ce qui concerne la couleur, étant donné qu'un motif de 8 couleurs est utilisé et que l'écrasement est également utilisé, nous nous retrouvons avec 2 couleurs pour l'arrière-plan (château, branches) et 3 couleurs pour les sprites (personnage et oiseau). Il est facile de modifier le programme pour utiliser un motif de rotation à 4 couleurs et ainsi avoir 5 couleurs pour les sprites, ce qui est plus raisonnable.

La liste complète se trouve ci-dessous.

```
10 MÉMOIRE 24999
11 ON BREAK GOSUB 5000
20 APPELER &6B78
30 FOR i=0 TO 31:|SETUPSP,i,0,0,0:NEXT:|AUTOALL,1:|PRINTSPALL,0,1,0
40 |SETUPSP,31,7,1:|SETUPSP,31,0,65:|LOCATESP,31,130,36:'caractère
50 |SETUPSP,30,7,7:|SETUPSP,30,0,157:|LOCATESP,30,50,80:
|SETUPSP,30,15,2 : 'oiseau
60 MODE 0:DEFINT a-z
80 CALL &BC02 : "palette par défaut
90 BORDER 10
100 ENCRE 6,15:ENCRE 7,15
110 ENCRE 4,26:ENCRE 5,26
120 ENCRE 2.0:ENCRE 3.0
130 INK 1,14
140 ' établissement du motif de l'encre
170 tini=8:|RINK,tini,0,6,3,3,3,3,3,3
200 |RINK,0
210 LOCATE 1,1:pen 4:PRINT "DEMO |RINK et |MAP2SP".
220 ' PEINTURE murale -----
230 |SETUPSP,29,9,23 : "brique
231 y=152
232 FOR row=1 TO 6
240 FOR brick=xini to 42 step 2:|PRINTSP,29,y,brick*2:next
241 xini=(xini-1) mod 2 : y=y+8
242 suivant
390 dir=1:x=0:xp=80 : cycle=40 : stepy=2
400 |MUSIQUE,0,0,0,7
409 ' cycle de jeu -----
410 |AUTOALL:|PRINTSPALL
450 IF INKEY(27)=0 THEN |RINK, stepy:x=x+1:|MAP2SP,0,x:|MOVER,30,0,-
1:si saut=0 alors IF dir=2 ALORS dir=1:|SETUPSP,31,7,dir ELSE |ANIMA,31
460 SI INKEY(34)=0 ALORS |RINK,-stepy::x=x-1:|MAP2SP,0,x:if jump=0 then IF
dir=1 THEN dir=2:|SETUPSP,31,7,dir ELSE |ANIMA,31
471 IF INKEY(67)+jump=0 THEN
jump=cycle:|SETUPSP,31,0,205:|SETUPSP,31,15,dir-1:|SETUPSP,31,7,31+dir
472 SI cycle-saut=20 ALORS saut=0:|SETUPSP,31,7,dir:|MOVER,31,5,0 :
490 |PEEK,27483,@xp:IF xp<-20 THEN |LOCATESP,30,50,80:'oiseau recommencer
501 cycle=cycle+1
502 SI xant=x ALORS |MAP2SP,0,32000
503 xant=x:' IF still THEN Je n'imprime pas le château pour qu'il ne
clignote pas
510 GOTO 410
5000 |MUSIC:CALL &BC02:pen 1:MODE 2:END
```

15 Jeux de plateforme

La difficulté de la programmation d'un jeu de plateforme réside principalement dans la gestion correcte de la physique des sauts et des collisions entre plateformes. Quelque chose que vous pouvez trouver dans le jeu vidéo : "**Fresh fruits & vegetables**" disponible chez 8BP.



Fig. 94 Fruits et légumes frais

Sauts :

En gros, pour la physique des sauts, au lieu d'utiliser l'équation de Newton, j'ai défini une trajectoire où Vy commence à -5 et diminue image par image. Lorsque la position zénithale est atteinte, l'image est modifiée de sorte qu'elle s'efface en haut et que la vitesse Vy devient positive et augmente progressivement.

Cela revient à appliquer l'équation de Newton, mais sans les calculs.

Contrôle du sol

Pendant que le personnage marche, je vérifie chaque image pour voir s'il y a un sol. Comme les plates-formes appartiennent à la carte du monde, elles utilisent des identifiants de sprites bas (<10), donc si avec COLSPALL je détecte un nombre <10, il y a un sol. Si le résultat de la détection est un 32 (les sprites vont de 0 à 31), il n'y a rien et le personnage doit commencer à tomber. Les ennemis ne détectent rien, ils marchent simplement le long d'itinéraires prédéfinis, où il est indiqué combien de pas ils doivent faire dans chaque direction, puis recommencer. A partir de là, le sprite parcourt l'itinéraire pas à pas en invoquant **|AUTOALL** (qui en interne invoque déjà **|ROUTEALL**).

Collisions de plates-formes :

lorsque le personnage est sur la trajectoire de chute, je le déplace (sans imprimer) de 5 unités vers le bas et je détecte la collision du sprite. Si la collision est de 32, il n'y a pas de collision et je continue à faire tomber le personnage. Si la collision est inférieure à 10, le personnage est entré en collision avec une plateforme. Dans ce cas, je déplace le personnage pour qu'il s'ajuste parfaitement au début de la plateforme, donc : position de la tête de la poupée est "posy" position des pieds est posy+26 car le personnage mesure 21 et j'ajoute 5 car il tombe (il y a 5 d'autodestruction), donc en réalité il mesure 26. Comme les plateformes ont été placées en multiples de 8, je garde simplement le reste de la division entière, que je peux obtenir avec un ET 7 bref, deux instructions très bien pensées, mais seulement deux :

```
dy = (posy%+26) AND 7  
| MOVER,31,5-dy,0
```

J'assigne ensuite la séquence d'animation de marche, qui n'est pas la séquence de chute et dont les images ont une hauteur de 21 images.

16 Hordes d'ennemis dans les jeux à défilement

Les jeux à défilement se déroulent généralement sous la forme d'une succession de hordes d'ennemis de types et de trajectoires différents, au fur et à mesure que vous avancez verticalement ou horizontalement.

Si vous vouliez que votre carte comporte 10 hordes à différents moments de la progression de la carte (ou du cycle de jeu), vous pourriez utiliser 10 instructions IF, mais l'exécution de chaque cycle serait très lente (chaque vérification IF coûte une milliseconde).

La meilleure solution consiste à conserver deux tableaux, l'un pour la position de la horde et l'autre pour le type de horde.

Index (Numéro d'ordre séquentiel)	Nexthorda (Cycle dans lequel il doit apparaître)	Tipohorda (Type d'ennemis de la horde)
0	100	1 - avion
1	200	2 - missiles
		4 - OVNI
	320	3 - dragons

```
10 dim tipohorda(10)
20 typeforda(0)=1 : typeforda(1)=2 : typeforda(2)=4 :
t y p e f o r d a (3)=3 :
30 dim nexthorda(0)
40 nexthorda(0)=100:nexthorda(1)=200 : nexthorda(2)=250 ...
50 index=0
```

Cycle de jeu de 100 rem

```
110 cycle=cycle +1
120 SI cycle = nexthorda(index) ALORS GOSUB 500
```

...

```
500   création d'une    horde
      routine rem
```

```
510   sur tipohorda(index)  goto 600,700,800
```

```
520   rem routin création horde type 4.  Al final    nous    goto
          e                      ferons
```

```
600   rem routin création horde type 1.  Al final    nous    goto
          e                      ferons
```

```
1000  rem routin création horde type 2.  Al final    nous    goto
          Index=index+1           ferons
```

```
1000  RETOUR
800   rem routin création horde type 3.  Al final    nous    goto
          e                      ferons
```

...

Au lieu d'utiliser le cycle dans lequel il devrait apparaître, vous pouvez également faire la comparaison avec la position de la carte dans laquelle vous vous trouvez, cela dépendra de la façon dont vous voulez le faire, mais avec cette idée des deux tableaux,

vous avez déjà l'approche générale pour organiser vos hordes d'ennemis.

17 Mini-caractères redéfinissables : PRINTAT

Le jeu de caractères Amstrad est agréable et bien construit. Cependant, en mode 0, nous n'avons que 20 caractères de large par ligne et ils apparaissent trop "larges", de sorte qu'ils ne conviennent pas toujours à l'affichage de certains textes ou marqueurs dans un jeu. De plus, la commande PRINT est très lente, il n'est donc pas recommandé de mettre à jour les marqueurs fréquemment, car le jeu "s'arrête" pendant qu'il imprime, ce n'est que quelques millisecondes, mais c'est perceptible.

C'est pourquoi, à partir de la version v31 de 8BP, la commande PRINTAT a été ajoutée, qui permet d'imprimer une chaîne de caractères en utilisant un nouvel ensemble de caractères plus petits (que j'appelle "mini-caractères"). Cette nouvelle commande vous permet d'utiliser le mécanisme de transparence des sprites, de sorte que vous pouvez imprimer des caractères en respectant l'arrière-plan. Elle fonctionne comme suit :

|PRINTAT, <flag transparence>, y, x, @string

Exemple :

```
cad$= "Hello".
|PRINTAT, 0, 100, 10, @cad$
```



Fig. 95 PRINTAT

La commande |PRINTAT imprime des chaînes de caractères et non des variables numériques. Par conséquent, si vous souhaitez imprimer un nombre (par exemple, les points sur le tableau d'affichage de votre jeu vidéo), vous devez le faire :

```
points=points+1
cad$= str$(points)
|PRINTAT,0,100,10, @cad$
```

La commande |PRINTAT n'est pas affectée par les limites d'écrêtage définies avec la commande |SETLIMITS. Ceci est très logique puisque vous utiliserez normalement PRINTAT pour imprimer les scores sur vos marqueurs, qui seront en dehors de la zone délimitée par |SETLIMITS.

Contrairement à la commande BASIC PRINT, la commande |PRINTAT est assez rapide et peut être utilisée pour mettre à jour fréquemment vos marqueurs de jeu.

PRINTAT utilise un alphabet redéfini, qui peut contenir une version réduite ou différente des caractères "officiels" d'Amstrad. Par défaut, le 8BP fournit un petit alphabet composé de chiffres, de lettres majuscules, d'espaces blancs et de quelques symboles. Vous ne pourrez pas utiliser un caractère qui ne fait pas partie de cet ensemble, comme les lettres minuscules. Les caractères de cet alphabet ont tous la même taille : 4 pixels de large x 5 pixels de haut, soit 2 octets x 5 lignes.

Cette chaîne contient tous les caractères que vous pouvez utiliser avec l'alphabet série 8BP. Notez qu'il n'y a pas de lettres minuscules et que de nombreux symboles sont absents, bien que vous puissiez créer votre propre alphabet qui les contienne. Le dernier symbole est le ". " "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ: ! ,."

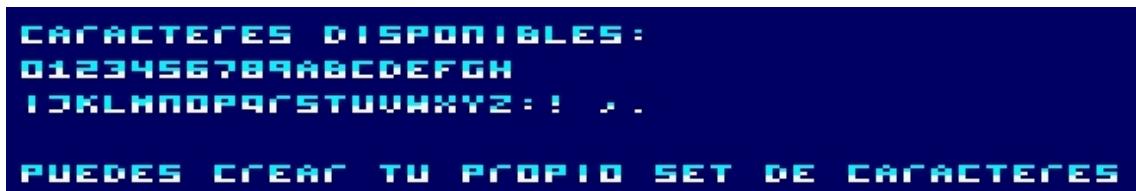


Fig. 96 Jeu de caractères par défaut disponible en 8BP pour l'utilisation de PRINTAT

IMPORTANT : Si vous essayez d'imprimer avec PRINTAT un caractère qui n'existe pas dans l'alphabet créé, le dernier caractère de la liste de caractères sera imprimé, qui dans l'alphabet par défaut est le point "".

J'ai créé les caractères en utilisant les encres 2 et 4, afin de pouvoir utiliser la surimpression, car les couleurs de fond sont 0 et 1 et, en utilisant la surimpression, l'encre 2 doit être égale à 3 et l'encre 4 doit être égale à 5 (voir le chapitre où j'explique la surimpression). Pour utiliser la surimpression, il suffit de mettre le drapeau de transparence de la commande PRINTAT à "1".

17.1 Créez votre propre mini-alphabet

Les caractères à utiliser avec la commande PRINTAT sont définis dans un fichier du répertoire ASM et importés du fichier "images.asm".

Regardons un fragment de "images.asm"

Nous y trouvons ces trois lignes :

```
si vous n'utilisez pas la commande |PRINTAT, mais uniquement les caractères Amstrad
Vous pouvez ensuite commenter les 3 lignes suivantes
ALPHABET DE DÉBUT
lire "alphabet_default.asm"
FIN DE L'ALPHABET
```

L'alphabet est constitué de quelques données et des images de chaque caractère. Tout cela est rassemblé dans la zone de mémoire d'image 8BP. L'alphabet par défaut fait un peu plus de 400 octets.

Le fichier alphabet_default.asm contient l'alphabet standard du 8BP. Vous pouvez créer votre propre fichier d'alphabet. Ce fichier contient 3 variables indiquant la taille des caractères, que vous pouvez dessiner dans la taille que vous voulez. Par défaut, j'ai créé un alphabet de 2 octets de large par 5 lignes de haut, mais vous pouvez décider d'utiliser une autre taille, voire de créer des caractères géants. Si vous modifiez la largeur ou la hauteur, vous devez également modifier la variable ALPHA_SIZE dans votre fichier alphabet.asm en conséquence.

```
ALPHA_width db 2 ; largeur de l'alphabet.toutes les lettres mesurent
la même chose ALPHA_height      db 5 ; hauteur de
l'alphabet.toutes les lettres sont identiques ALPHA_span db 2*5
```

Ensuite, nous trouvons la chaîne indiquant les caractères valides à utiliser avec la commande PRINTAT. Ces caractères sont valables parce qu'ils ont un dessin associé. Après cette chaîne, les dessins de tous ces caractères sont trouvés un par un, dans l'ordre où ils apparaissent dans la chaîne de texte ALPHA_LIST

```
ALPHA_LIST
"0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ: ! ,." ; caractères créés
db 0 ;l'octet zéro indique la fin de la chaîne de la liste Alpha
```

Les images de chaque caractère de la chaîne ALPHA_LIST sont présentées ci-dessous. Je les ai créées avec l'outil SPEDIT. Le premier d'entre eux doit être le premier caractère de la chaîne ALPHA_LIST, c'est-à-dire "0".

Les octets correspondant à ce caractère sont indiqués ici :

; caractère 0 db 12 , 8 db 8 , 8 db 8 , 8 db 32 , 32 db 48 , 32	
--	--

Ensuite, nous trouverons un par un le reste des lettres, chiffres et symboles définis. Avec un peu de patience, vous pouvez créer votre propre série de mini-personnages, qui donneront plus de personnalité à votre jeu vidéo. Vous ne devez créer que ceux que vous allez utiliser, et moins il y en a, moins vous utiliserez de mémoire dans la zone d'image.

Rappelez-vous que si vous essayez d'imprimer un caractère que vous n'avez pas créé, le dernier des caractères définis dans la liste ALPHA_LIST sera imprimé.

17.2 Alphabet par défaut pour le MODE 1

L'alphabet par défaut de 8BP est créé en MODE 0 et si vous essayez de l'utiliser en MODE 1, il ne fonctionnera pas correctement, car il s'agit de dessins réalisés en mode 0. Par défaut, 8BP est également livré avec un alphabet qui fonctionne en MODE 1. Il suffit de modifier une ligne du fichier images.asm. Cet alphabet est inspiré d'une police appelée "5th agent".

```
ALPHABET DE DÉBUT
lire "alphabet_default_mode1.asm"
FIN DE L'ALPHABET
```

```
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ : ! , .
```


18 Pseudo 3D

Dans les années 1980, les jeux de voiture de type "Pole position" étaient très populaires. Ils donnaient une impression de 3D, mais ne faisaient pas de calculs en 3D, se contentant du plan de route, et parfois même pas de celui-ci, car il s'agissait d'approximations qui donnaient une bonne impression de vitesse.

Dans les machines d'arcade, des puces spécifiques étaient utilisées pour effectuer la "mise à l'échelle des sprites", ce qui permettait d'augmenter la taille des sprites en douceur, et le calcul de la route au moyen d'une puce spécifique (telle que la "sega Road chip") qui était exclusivement dédiée à la peinture de la route et de ses bandes. La carte de la route était ensuite ajoutée à la carte des sprites et l'image finale était composée. Ces puces ont été utilisées dans des machines d'arcade telles que "Pole Position" et "Space Harrier".



Fig. 97 Position de pôle et course de sortie (machines d'arcade)

La caractéristique commune des techniques logicielles (utilisées sur les ordinateurs 8bit) et matérielles (utilisées sur les machines d'arcade) est que les courbes sont une illusion : la route est déformée tandis que les montagnes à l'horizon sont déplacées pour donner la sensation d'un virage, mais il n'y a pas de virage du tout. Les résultats étaient souvent très convaincants, mais les courbes n'étaient qu'une illusion.

Les techniques utilisées sur les ordinateurs 8 bits étaient très variées. Tout était acceptable tant que le joueur avait l'impression d'être sur une piste de course. Dans de nombreux cas, la rotation de l'encre était utilisée pour donner une impression de vitesse. De nombreux jeux souffraient d'un faible taux de rafraîchissement, inférieur à 5 images par seconde. Parmi les meilleurs jeux pour Amstrad, on peut citer "**3D grand Prix**" (qui utilise un logiciel de mise à l'échelle des sprites combiné à une animation à l'encre), "**Buggy boy**" basé sur une technique de programmation très avancée d'effet de trame et quelques autres comme "**Crazy cars**" ou "**Chase HQ**".

Depuis la version V32 de 8BP, vous disposez de la capacité Pseudo-3D, utilisée dans le jeu de démonstration "3D Racing one". Elle est très facile à utiliser et vous permet de créer vos propres jeux de course, jeux de chars, jeux de bateaux, etc.

Pour utiliser Pseudo-3D en 8BP, vous devez utiliser l'"**option d'assemblage**" 3, ce qui laisse 24 KB libres pour votre programme BASIC.

Les possibilités offertes par 8BP pour rendre la pseudo-3D disponible sont les suivantes :

- 1) **Projection 3D** : il s'agit de projeter une carte du monde en 2D en 3D, de sorte que, même si nous la parcourons en 2D, elle nous donne la sensation de la voir en 3D. Pour ce faire, il suffit d'invoquer la commande |3D pour configurer les

commandes PRINTSPALL.

et PRINTSP pour qu'ils projettent en 3D avant de peindre sur l'écran. Il ne sera pas possible de tourner dans le plan, nous regarderons toujours vers l'avant, mais l'effet combiné d'une courbe et de maisons ou de montagnes en mouvement à l'horizon simulera que nous prenons une courbe.

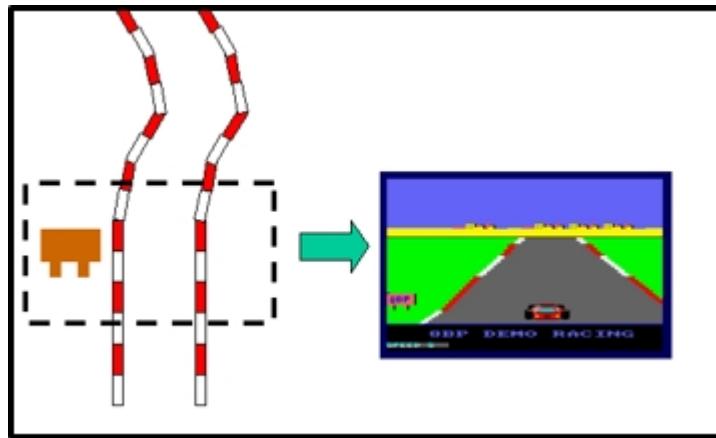


Fig. 98 Projection 3D

- 2) **Zoom** : il s'agit de pouvoir utiliser différentes versions de la même image d'un objet pour donner un effet de zoom au fur et à mesure que l'on s'en approche. Cela se fait simplement dans le fichier image, en définissant 3 versions de la même image et en les regroupant dans une structure. L'image montre l'exemple typique d'un zoom sur l'affiche. Les commandes |PRINTSPALL et |PRINTSP choisiront la version la plus appropriée de l'image en fonction de la distance à laquelle elle se trouve, sans qu'il soit nécessaire de faire quoi que ce soit d'autre que de définir l'image comme une image de type "Zoom".



Fig. 99 Images zoom

- 3) **Segments** : il s'agit de pouvoir disposer de sprites de type "segment", définis par une seule ligne de balayage horizontale (ils prennent donc très peu de place), auxquels on peut associer une longueur et une pente. Avec eux, on peut construire des routes, des rivières, etc. et des bords de route. Cela se fait simplement dans le fichier image en définissant un type d'image qui a non seulement une largeur et une hauteur mais aussi une pente. Nous utiliserons ces segments dans la construction de la carte du monde.



Fig. 100 segments avec différentes inclinaisons

Les segments utilisés dans le jeu "3D Racing one" sont rouges ou blancs et bien qu'ils soient longs, ils ne sont définis que par une ligne de balayage. Par exemple, le segment blanc gauche est composé de quelques octets verts pour l'herbe, de quelques octets blancs et de quelques octets gris pour la route. Au moment de l'impression, le segment ainsi défini est reproduit à la longueur souhaitée et imprimé en perspective, de sorte que même s'il s'agit d'un segment droit, il apparaîtra de travers.

Enfin, il convient de mentionner que dans de nombreuses occasions, il sera nécessaire de mettre à jour dynamiquement la carte (commande **|UMAP**). Grâce à cette commande, la carte peut être très grande (ce qui est nécessaire si nous créons un circuit avec de nombreux segments). Cette commande est décrite dans le chapitre sur le défilement. Dans ce qui suit, nous allons examiner en détail ces trois caractéristiques et la manière de les utiliser dans vos programmes.

18.1 Projection 3D

Pour projeter, nous disposons de la

commande **|3D Use**

|3D, 1, <sprite_fin>, <offsety> : REM active la projection 3D.

|3D, 0 REM désactive la projection 3D

Cette commande active la projection 3D dans la commande **|PRINTSP** et dans **|PRINTSPALL**. Cela signifie qu'avant d'imprimer à l'écran, les coordonnées "projétées" seront calculées puis imprimées à l'écran. Les coordonnées des sprites ne sont pas affectées, c'est-à-dire qu'elles restent les mêmes, mais dans le monde 2D. À l'écran, nous avons maintenant une vue en 3D et les coordonnées où ils seront imprimés seront le résultat de l'exécution de la fonction de projection sur leurs coordonnées en 2D.

Les sprites concernés vont du sprite 0 au **<sprite_fin>**. Les autres sprites ne seront pas projetés, ils seront simplement imprimés à l'écran dans leurs coordonnées 2D.

Cette commande **n'affecte pas les mécanismes de collision**, c'est-à-dire que si nous utilisons COLSPALL et détectons une collision entre des sprites projetés, la collision se produira dans le plan 2D. Dans certains cas, deux sprites projetés peuvent se chevaucher

partiellement dans le plan 2D.

L'un peut être légèrement en avance sur l'autre et ils peuvent se chevaucher lorsqu'ils sont projetés, mais il ne s'agit pas d'une véritable collision. Les collisions sont détectées dans le plan 2D.

Quant au dernier paramètre `<offsety>`, il s'agit de projeter plus ou moins haut, afin de pouvoir placer les marqueurs de jeu où l'on veut. Lorsque l'on projette l'écran, qui fait 200 pixels de haut, il devient 100 pixels de haut, nous pouvons donc choisir à quelle hauteur nous plaçons la projection. Si un Sprite n'est pas affecté par la projection parce qu'il est plus haut que `<Sprite_fin>`, il n'est pas non plus affecté par `<offsety>`. C'est le cas, par exemple, des nuages et des maisons à l'horizon dans le jeu "3D Racing one".

Pour comprendre les coordonnées de l'écran sur lesquelles vos sprites projetés apparaissent finalement, je vous recommande de lire la section mathématique très simple suivante pour bien comprendre. La figure suivante représente les coordonnées de la carte du monde qui sont projetées sur certains points représentatifs de l'écran lorsque MAP2SP est invoqué avec ($yo=0$, $xo=0$). La coordonnée Z représente la distance à laquelle les objets sont situés, également appelée "profondeur".

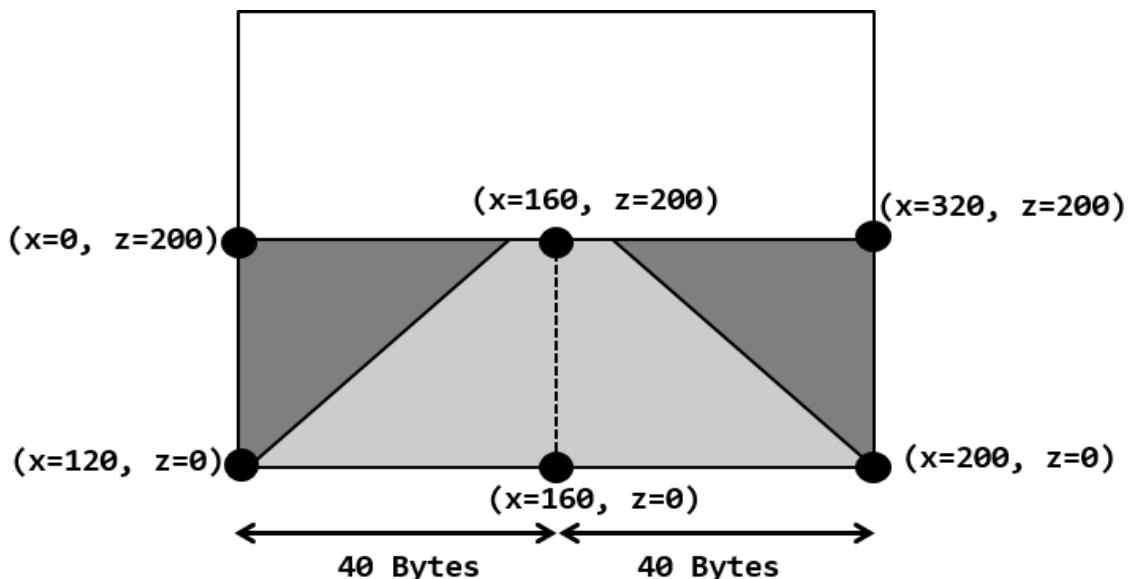


Fig. 101 Coordonnées mondiales projetées

Si, au lieu de ($xo=0$, $yo=0$), nous utilisons une autre coordonnée pour MAP2SP, les coordonnées 2D du monde correspondant aux points référencés dans l'image seront décalées de (x , z) comme indiqué par (xo , yo).

18.1.1 Mathématiques de la pseudo-projection 3D

La projection pseudo-3D consiste à projeter sur l'écran le plan du sol, où se trouve notre carte du monde en 2D.

Calcul de la coordonnée Y

Notre sol peut être infini, mais nous ne projetterons que 200 pixels de long. Cette longueur est appelée "profondeur" ou coordonnée "Z". Ces 200 pixels de profondeur, une fois projetés, sont comprimés en 100 lignes de hauteur (coordonnée Y projetée). Les pixels

Les objets projetés les plus éloignés constituent la ligne d'horizon. Notez que nous ne verrons pas les objets très éloignés sur l'horizon, mais seulement ceux qui se trouvent à une profondeur de 200 au maximum.

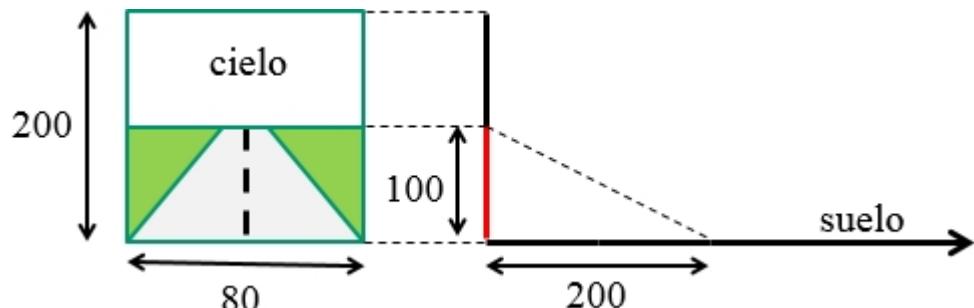


Fig. 102 projection du sol sur l'écran

Au fur et à mesure que les objets s'éloignent, ils deviennent de plus en plus petits. Il ne s'agit pas d'une relation linéaire entre l'écran et le sol, c'est-à-dire que pour savoir à quelle hauteur imprimer un pixel qui se trouve à une certaine distance, il est incorrect de penser que comme la profondeur couverte est de 200 et qu'elle est projetée sur 100 lignes de haut, il suffit de diviser par deux. Dans une fonction linéaire (telle que $y = f(z) = A \cdot z + B$) la dérivée (A) est constante, mais dans la projection, ce qui est constant est la "dérivée seconde" de la fonction $f(z)$. Nous allons maintenant examiner cela en détail.

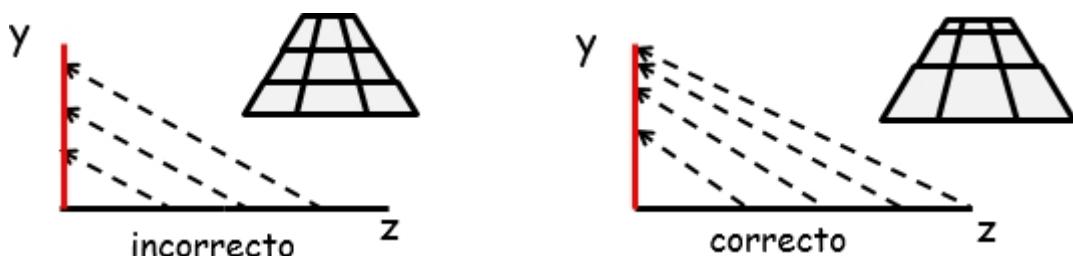


Fig. 103 La projection correcte n'est pas linéaire

Supposons que nous partions de " $Z=0$ " et que nous voulions savoir combien vaut " Y ". C'est très simple, à la ligne de base ($z=0$), la coordonnée y est également nulle.

Si nous incrémentons z avec une petite variation " dz ", le " y " vaudra le " y " précédent (zéro) plus un incrément dy , qui sera initialement égal à l'incrément dz puisque l'incrément a été faible et que nous sommes proches de l'horizon.

$$\boxed{dy \text{ (initial)} = dz}$$

Maintenant, si nous nous éloignons à nouveau de dz , l'incrément dy doit diminuer, et si nous nous éloignons à nouveau de dz , le dy à ajouter sera de plus en plus petit. En d'autres termes :

Chaque fois que nous nous éloignons de dz , nous ajoutons un incrément à y que chaque fois que nous nous éloignons de dz , nous ajoutons un incrément à y que chaque fois que nous nous éloignons de dz , nous ajoutons un incrément à y que chaque fois que nous nous éloignons de dz , nous ajoutons un incrément à y .

le temps est plus

court $z=z+dz$

$dy = dy + ddy$, où ddy négatif $y = y + dy$

L'augmentation de dy est constante. Nous avons appelé cette augmentation ddy . Eh bien, dy est la "dérivée" de y , tandis que " ddy " est ce que l'on appelle la "dérivée seconde". Ici, la dérivée n'est pas constante, mais la dérivée seconde l'est.

La valeur constante que nous attribuons à " ddy " produira des projections plus ou moins exagérées. Dans le 8BP, la valeur ddy est négative, environ -0,005, ce qui fait que la valeur dy à la ligne de base est presque égale à 1, alors qu'à la ligne d'horizon, l'accumulation de 200 ddy fait que la valeur dy finit par être égale à zéro.

Malgré la simplicité de ces calculs, les effectuer en temps réel est coûteux pour notre bien-aimé Amstrad CPC, c'est pourquoi le 8BP fait une approximation pour éviter les calculs et traduire " z " en " y " d'une manière beaucoup plus simple et avec un résultat très similaire.

Si $0 \leq z < 50$, alors $dy = dz$, donc $y = z$

Si $50 \leq z < 110$, alors $dy = dz/2$, donc $y = 50 + (z-50)/2$

Si $110 \leq z \leq 200$, alors $dy = dz/4$, donc $y = 50 + (110-50)/2 + (z-110)/2$

En d'autres termes, nous avons divisé l'écran en trois bandes et chaque bande est traitée comme une zone "linéaire" (puisque " dy " est constant) mais avec une valeur différente de " dy " par rapport aux autres bandes. Cette approximation donne des résultats visuellement très similaires à ceux qui sont mathématiquement corrects.

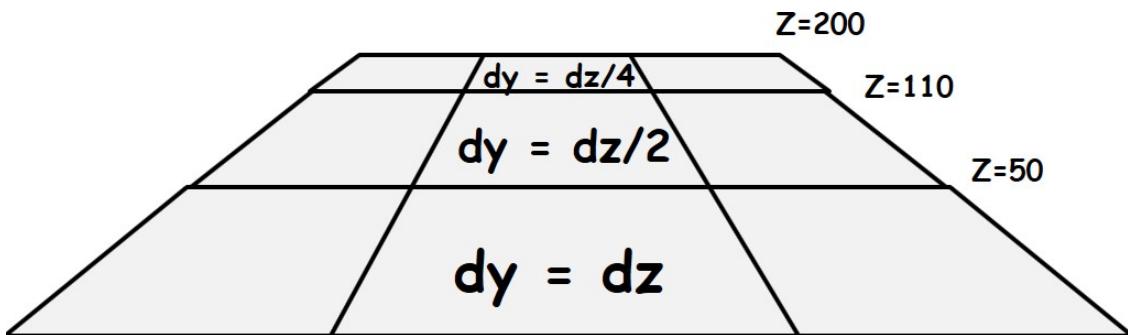


Fig. 104 Approche du 8BP

Les équations utilisées dans le 8BP tiennent également compte du fait que les coordonnées de l'écran sont en fait "inversées", c'est-à-dire que la coordonnée zéro est la coordonnée supérieure et la coordonnée 200 est la coordonnée inférieure, mais il s'agit simplement d'un ajustement final très simple.

Passons maintenant au calcul de la coordonnée "X" :

Il existe une différence fondamentale entre la ligne d'horizon et la ligne de sol. La ligne au sol mesure 80 octets, mais la ligne d'horizon représente une plus grande largeur, parce que la route est droite et que ce qui mesure 80 au sol ne mesure qu'une fraction dans la ligne d'horizon, et plus précisément dans le 8BP, il mesure 4 fois moins. La route se rétrécit à l'horizon, parce que l'horizon représente beaucoup plus. Dans la projection appliquée par le 8BP, il représente 320 octets.

Et si l'horizon mesure 320 et le sol 80, c'est parce que la surface réelle totale que nous pouvons voir sur l'écran une fois la projection effectuée est une surface trapézoïdale. Ce n'est PAS que le sol est un trapèze, mais que la surface du sol que nous pouvons voir sur l'écran est de forme trapézoïdale.

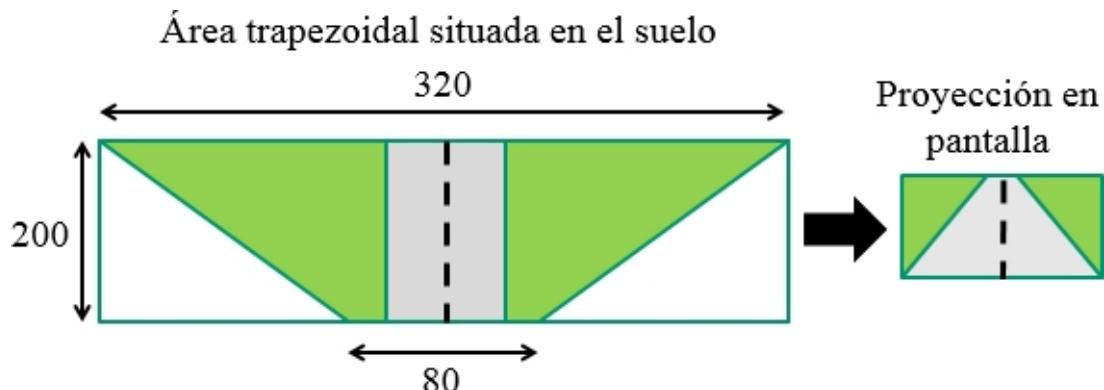


Fig. 105 Zone trapézoïdale affichée

Intuitivement et sans montrer d'équations, ce que nous voulons faire et comment fonctionne la projection est déjà clair. Examinons maintenant les équations.

Par essence, les mathématiques de la coordonnée "X" sont les mêmes que celles de la coordonnée "Y". En effet, une fois la coordonnée "Y" calculée, nous pouvons établir une équation linéaire $x=f(y)$ car, en raison de la non-linéarité de "Y" par rapport à "Z", cela revient à créer une relation non linéaire entre "X" et "Z". Nous avons dit plus haut que l'horizon a une longueur de 320 octets et que le sol a une longueur de 80 octets. Cela signifie que le sol est 4 fois plus petit que l'horizon. Dans la distance lointaine, nous devons diviser par 4 (la division est coûteuse, nous préférions donc multiplier par un facteur 0,25), tandis que dans la distance proche, nous devons multiplier par un facteur = 1.

Il s'agit de centrer la coordonnée X de l'objet à projeter par rapport au centre de l'écran. Nous multiplierons ensuite par un facteur qui dépendra de la coordonnée "Y" projetée. Cela permet d'établir une relation non linéaire entre "X" et "Z".

Nous devons construire une équation qui renvoie un résultat 4 fois plus petit à l'horizon, par exemple :

Facteur = ((100-y)+ 32) / 2
x= (x - centre) * Facteur + centre
si z=200 alors y= 100, et ensuite facteur =16
si z=0 alors y= 0, et alors facteur =64 (4 fois plus)

L'équation choisie est intéressante parce que la division par 2 est quelque chose que l'Amstrad peut faire en tournant un bit en binaire. C'est-à-dire qu'il peut le faire en quelques cycles d'horloge.

Comme le montre l'équation, pour projeter la coordonnée "X", il suffit de la centrer et de la multiplier par le facteur obtenu. Le nombre ainsi obtenu est très grand, car dans le cas de la ligne d'horizon, nous serons multipliés par 16 et dans le cas de la ligne de sol par 64, c'est-à-dire que nous aurons multiplié la coordonnée x par un facteur dont la valeur est comprise entre 16 et 64. Entre l'horizon et le sol, nous aurons les 48 valeurs possibles pour le facteur, de sorte que, même si le facteur est un nombre entier, il évoluera en douceur.

On divise ensuite par 64, ce qui revient à tourner 6 fois en binaire, et c'est tout. Ce dernier résultat équivaudra à multiplier par 0,25 l'horizon, par 1 le sol et par la valeur décimale correspondant à toute hauteur intermédiaire entre le sol et l'horizon, mais nous l'avons fait avec des nombres entiers, que l'Amstrad peut traiter rapidement. Cette technique est appelée "arithmétique à virgule fixe".

En tenant compte de tout cela, et en supposant que nous demandions à MAP2SP de générer les sprites de la carte du monde à partir des coordonnées ($yo=0$, $xo=0$), ce que nous verrons à l'écran aura les coordonnées suivantes du monde. Je suis sûr qu'il est facile de comprendre la figure maintenant.

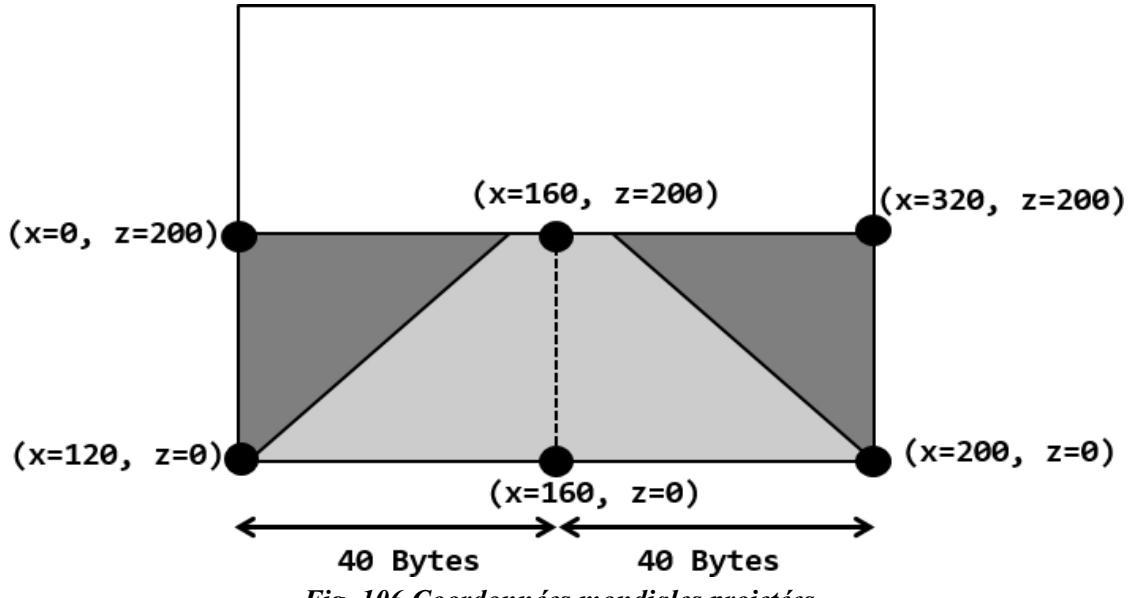


Fig. 106 Coordonnées mondiales projetées

Comme vous pouvez le déduire, le point ($x=0$, $y=0$) de la carte du monde est projeté hors de l'écran, il n'est pas affiché. Si au lieu de ($xo=0$, $yo=0$) nous utilisons une autre coordonnée pour MAP2SP, les coordonnées 2D du monde correspondant aux points référencés dans l'image seront décalées de (x , z) comme indiqué par (xo , yo).

18.1.2 Courbes

Comme je l'ai mentionné au début de ce chapitre, la projection 3D utilisée par 8BP ne permet pas de faire des rotations du plan, donc pour simuler une courbe, nous devrons faire un petit tour de passe-passe. Il s'agit de faire tourner la route vers la droite ou la gauche, tout en déplaçant des sprites tels que des maisons ou des montagnes à l'horizon. Ces sprites ne seront pas projetés et pour éviter cela, nous utiliserons des identifiants supérieurs à `<Sprite_fin>`. Rappelez-vous que pour activer la projection 3D, nous devons faire :

| La liste de ces éléments est la suivante : <3D, 1, <Sprite_fin>, <offsety>, <offsety>, <offsety>, <offsety>, <offsety>.

Par conséquent, un circuit apparent avec des courbes sera représenté sur notre carte 2D comme une route avec des pentes vers la droite et vers la gauche, tout simplement.

Cette stratégie permet de créer la sensation d'un pilote de course sur un circuit avec de vraies courbes, et de donner l'impression que sa voiture tourne dans les courbes au moyen de maisons ou de montagnes à l'horizon qui se déplacent dans la direction opposée à la coordonnée X. Si vous êtes un bon artiste et que vous tordez progressivement les différents segments de façon plus ou moins prononcée, vous donnerez l'impression que les courbes sont très réalistes. Si vous êtes un bon artiste et que vous tordez progressivement les différents segments de manière plus ou moins prononcée, vous donnerez l'impression de courbes très réalistes.

Cette stratégie est très efficace sur le plan informatique et suffisamment réaliste. Si nous

voulions faire pivoter le plan du sol pour de vrai, nous devrions appliquer un calcul matriciel pour effectuer la rotation, avec de nombreuses opérations très coûteuses et, en outre, les textures des sprites sur le sol devraient également pivoter, de sorte que le coût de calcul serait énorme.

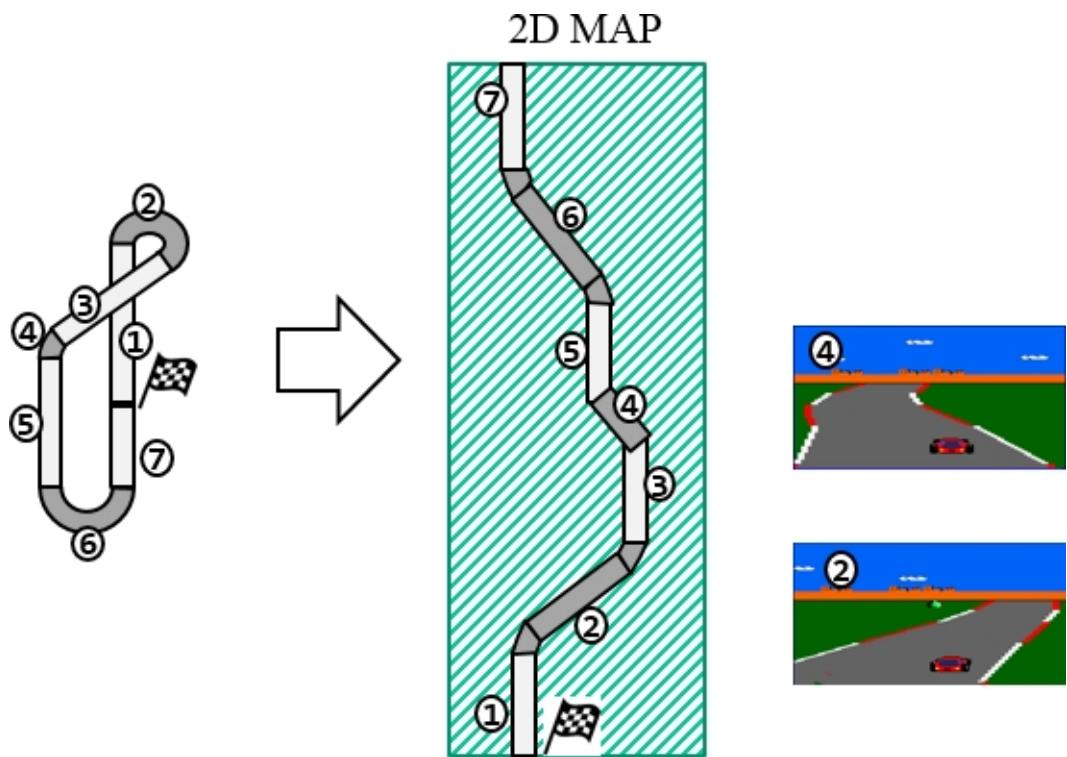


Fig. 107 Circuit imaginaire et carte du monde en 2D

Les ordinateurs d'aujourd'hui sont si puissants que les stratégies présentées ici sont dénuées de sens, mais elles sont supérieures en élégance et en ingéniosité à la "force brute" d'aujourd'hui. N'oubliez pas que "les limitations ne sont pas un problème, mais une source d'inspiration".

Vous devrez utiliser la commande **|UMAP** pour parcourir de grandes cartes de circuits, mais n'oubliez pas d'invoquer **|UMAP** non pas à chaque cycle, mais seulement de temps en temps.

18.2 Zoom sur les images

Pour définir une "image zoom", il suffit de créer les différentes versions de l'image (3 versions). Ensuite, dans le fichier image "images_mygame.asm", nous chercherons une balise appelée "**_3D_ZOOM_IMAGES**".

Nous le trouverons :

```
IMAGES_3D_ZOOM_IMAGES
;=====
; limites applicables à toutes les images agrandies
L'horizon pour ces limites est considéré comme étant 0 et croissant
vers le bas jusqu'à 200
ZOOM_LIMIT_A
db 120 ; entre 200 (sol) et limitA est fixé à l'image 3
ZOOM_LIMIT_B
db 50
entre cette limite et la limite A, l'image 2 est définie.
plus proche de l'horizon que la limite B est fixée à l'image 1
=====
```

```

CARTEL_ZOOM
db 1 ; largeur
symbolique db 1 ;
hauteur symbolique
dw POSTER1, POSTER2, POSTER3

```

FIN_3D_ZOOM_IMAGES

Toutes les images ZOOM doivent être définies après la balise "**_BEGIN_3D_ZOOM_IMAGES**". Il s'agit d'images qui ont une largeur et une hauteur symboliques, car en réalité, un sprite auquel l'une de ces images est attribuée utilisera la largeur et la hauteur de la version de l'image qui est automatiquement choisie en fonction de sa coordonnée Y. Autrement dit, "CARTEL1" est une image normale, préalablement définie, dont la largeur, la hauteur et les octets contiennent le dessin. Il en va de même pour "CARTEL2" et "CARTEL3". Si nous associons l'image "CARTEL_ZOOM" à un Sprite (cela peut se faire en associant un identifiant à cette image au début du fichier image), il se produira que, selon la position du Sprite sur l'écran, l'une ou l'autre version de l'image sera imprimée.

Pour la sélection automatique de l'image, des seuils de coordonnées y sont définis. Vous pouvez modifier ces seuils, mais les seuils par défaut fonctionnent bien et sont les suivants :

- entre l'horizon =0 et 50, la première image est choisie (dans l'exemple, "CARTEL1").
- entre 50 et 120, la deuxième image est choisie (dans l'exemple, "CARTEL2").
- entre 120 et 200, la troisième image est choisie (dans l'exemple, "CARTEL3").

Le choix de ces limites pour délimiter les bords de l'écran est configurable et vous pouvez en définir autant que vous le souhaitez. Notez que le choix est fait en fonction de la coordonnée Y du Sprite non projeté. Une fois projeté, sa coordonnée Y varie beaucoup, mais ce n'est pas le "Y" projeté qui est utilisé pour délimiter les 3 bandes, mais le "Y" du Sprite en 2D. Lorsque le Sprite se déplace d'une bande à l'autre, son image change automatiquement. Si vous préférez qu'une image apparaisse en premier, vous pouvez modifier les seuils.

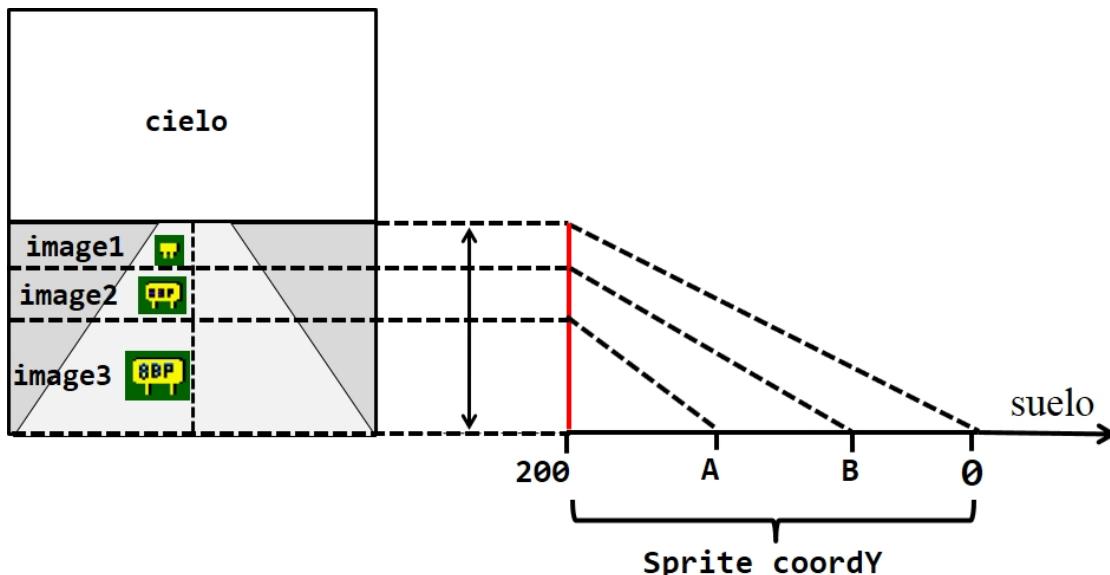


Fig. 108 Plages d'utilisation des 3 versions de l'image ZOOM

Par exemple :

```
REM supposons que CARTEL_ZOOM ait id=16 dans le fichier image  
|REM associe CARTEL_ZOOM au Sprite 20.  
|LOCATESP, 20,100, 160 : REM Sprite à au centre du "trapèze  
(coordy=100)  
|3D,1,31,0 : REM tous les sprites seront projetés.  
|PRINTSP,20 : REM la version 2 de l'affiche est imprimée.
```

18.3 Utilisation des segments

Maintenant que vous savez comment construire une carte 2D qui "simule" un circuit avec des courbes, voici un moyen puissant de construire les segments avec différents degrés d'inclinaison dont vous avez besoin pour construire des circuits de course ou des routes.

Un segment est une image d'une seule ligne en hauteur qui, par répétition, peut atteindre la longueur que l'on souhaite. Outre la longueur, il possède un autre paramètre, qui est l'inclinaison totale du segment, en octets. Cette inclinaison peut être négative (inclinaison vers la droite) ou positive (inclinaison vers la gauche).

Pour définir ce type d'images, vous devez les créer dans le fichier image de votre jeu "images_mygame.asm", après la balise "_BEGIN_3D_SEGMENTS".

```
;=====  
DÉBUT DES SEGMENTS 3D  
=====  
Il pourrait y avoir une définition différente des segments.  
; la largeur est la largeur de la ligne de balayage  
le point culminant est le point culminant 2D du segment  
; puis vient le dx, qui peut être positif (incliné à gauche) ou négatif  
(incliné à droite).  
db 0 ; il s'agit de la première image de type segment à être >  
_3D_SEGMENTS  
  
----- SEGMENT_EDGE_LWI10 -----  
SEGMENT_EDGE_LWI10  
db 22 ; largeur  
db 50 ; élevé  
db 10 ; dx  
db 192,192,192,192,192,192,192,192,192,192,192,192,192,192,240, 240  
,0,0,0,0,0,0,0,0,0  
;
```

Dans l'exemple, nous avons défini un segment comportant de l'herbe à gauche (octets verts), puis deux octets blancs et enfin des octets gris (route) à droite. La couleur verte ou grise dépend des couleurs que nous avons associées aux encres.

Il s'agit d'un segment incliné vers la gauche de 10 octets. Si nous avions mis un zéro dans l'inclinaison (dx), il s'agirait d'un segment droit, non incliné. Il a une hauteur de 50 lignes, mais lorsqu'il est projeté, il est moins grand, à moins qu'il ne soit très proche.

Les points du segment qui sont projetés ne sont que deux. Une fois projetées, les lignes de balayage sont peintes une à une avec un certain déplacement horizontal de sorte que la dernière ligne commence au point final. Notez que, bien que le segment soit peint en perspective, sa largeur est constante. Vous ne peignez pas la partie supérieure plus étroite (plus éloignée) et la partie inférieure plus large (plus proche), mais vous peignez chaque ligne de balayage exactement comme elle a été définie dans le fichier image.

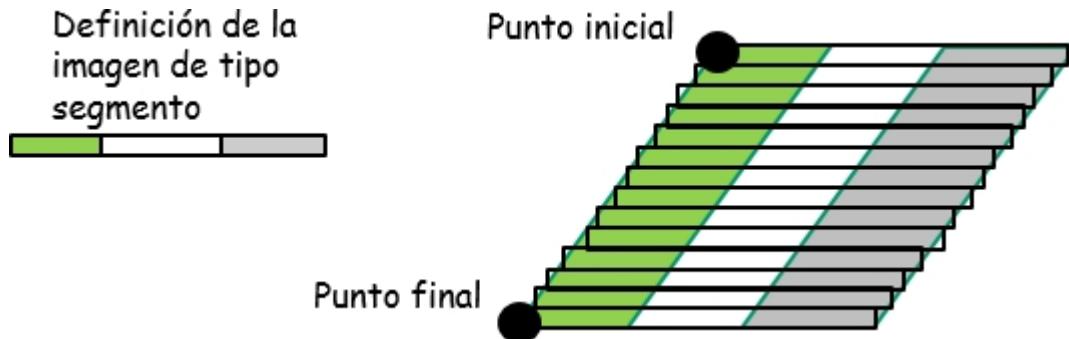


Fig. 109 2 points sont projetés sur un segment

Comme vous pouvez le voir, j'ai utilisé beaucoup d'octets d'herbe et de route. Ceci afin que le segment "s'efface" au fur et à mesure qu'il avance, bien que si la voiture va trop vite, des traces peuvent subsister. Une fois que votre jeu fonctionnera, vous pourrez vérifier si la vitesse est trop élevée et s'il reste des traces.



Fig. 110 Relation entre le degré d'inclinaison d'un segment et la vitesse maximale

Comme le montre la figure, la vitesse d'avance maximale d'un segment auto-grippant peut être plus élevée si le degré d'inclinaison est modéré. S'il est très croche, la vitesse ne peut pas être aussi élevée, sinon des traces subsisteront. Une fois le segment projeté, il sera plus court en raison de l'effet de perspective et, selon l'endroit où il se trouve, il peut être encore plus ou moins tordu. Il est conseillé de les rendre larges pour qu'ils s'effacent plus sûrement.

Dans le jeu "**3D Racing one**", il y a une étape appelée "superfast" dans laquelle, en utilisant des segments moins courbés, j'ai augmenté la vitesse de la voiture sans que les segments ne laissent de traces. Un "truc" simple et très efficace. Si le jeu est lent (par exemple un jeu de tanks, qui est censé être lent), les segments peuvent être très courbés car ils ne laisseront pas de trace en raison de l'effet de vitesse.

En résumé, pour augmenter la vitesse, vous avez deux options :

- Utiliser des segments moins tordus
- Utiliser des segments plus larges (avec plus de marge pour s'effacer).

19 Musique

Les outils dont je vais parler dans cette section ne sont pas programmés par moi, mais ils sont intégrés dans 8BP et ils sont vraiment bons.

19.1 Editer de la musique avec le tracker WYZ

Cet outil est un séquenceur musical pour la puce sonore AY3-8912. La musique qu'il génère peut être exportée et les résultats dans deux fichiers

- Un fichier d'instrument ".mus.asm".
- Un fichier de notes de musique ".mus".

Vous pouvez composer des chansons avec cet outil et la seule limitation est que toutes les chansons que vous intégrez dans votre jeu doivent partager le même fichier d'instrument.

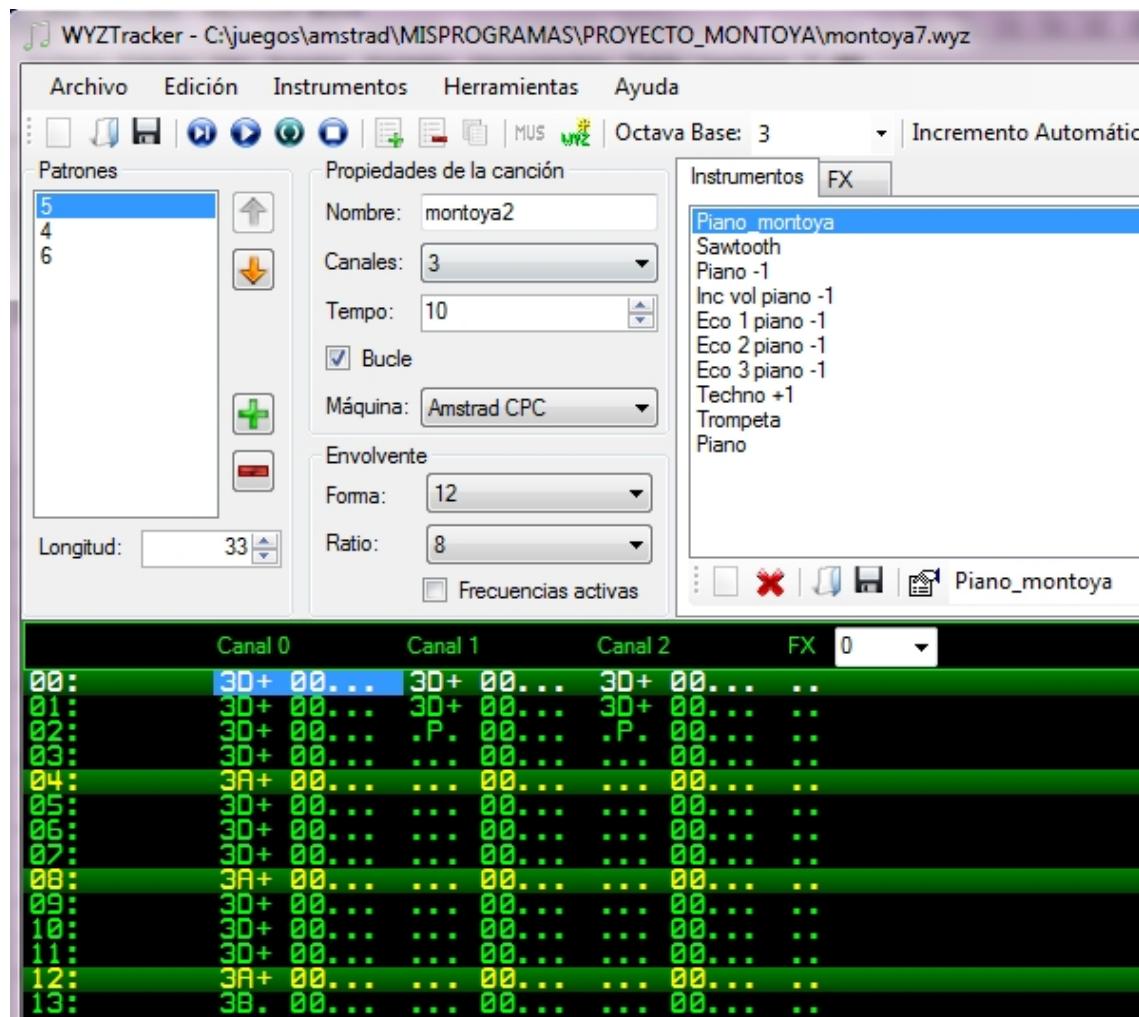


Fig. 111 WYZTracker

Depuis la version V41, vous pouvez utiliser le canal "FX" pour incorporer des effets sonores. Jusqu'à cette version, ce pseudo canal était ignoré dans le lecteur 8BP. Attention lors de l'édition de la musique car, bien que le canal FX soit toujours associé à un canal, dans le tracker WYZ il semble indépendant et le résultat peut être différent de ce que l'on entend dans le tracker WYZ.

Si le FX est associé au canal 0 et que vous placez un FX à un moment où le canal 0 ne joue pas, rien ne sera entendu dans l'amstrad même si votre effet est entendu dans le WYZtracker.

Ce séquenceur musical est complété par le WYZplayer qui est intégré dans la bibliothèque 8BP.

Depuis la version V26 de 8BP, la musique peut être composée avec WYZtracker version 2.0.1.0 et cela fonctionne très bien. Jusqu'à la version V25 de 8BP, la compatibilité se faisait avec WYZtracker 0.5.07 et il y avait quelques petits problèmes, mais tout cela a disparu avec WYZtracker 2.0.1.0.

IMPORTANT : n'utilisez pas l'octave 7 même si le tracker vous permet de l'utiliser. Cette octave ne peut pas être jouée et des problèmes de synchronisation peuvent survenir pendant la lecture de musique sur l'Amstrad.

Une recommandation importante lors de la création de vos chansons avec WyZtracker est de supprimer les instruments que vous n'allez pas utiliser. De cette façon, le fichier d'instruments (qui se termine par ".mus.asm") prendra moins de place, et comme 8BP ne réserve que **1.500 octets** pour la musique, chaque octet est important.

IMPORTANT : si vous éditez la musique avec une version de WYZtracker postérieure à la version

2.0.1.0 peut ne pas fonctionner correctement et vous pouvez rencontrer des effets étranges tels qu'un canal qui ne joue pas ou des notes qui ne sont pas synchronisées. Si cela vous arrive, la solution est de créer un fichier à partir de zéro avec WYZtracker 2.0.1.0 et de copier manuellement note par note la musique qui ne fonctionne pas.

19.2 Assemblage des chansons

Une fois que vous avez composé votre chanson et que vous disposez des deux fichiers, il vous suffit de modifier le fichier make_music.asm et d'y inclure vos fichiers musicaux comme suit :

```
après l'assemblage, sauvegardez-le avec save "musica.bin",b,32200,1400

org 32200
;-----MUSICA-----
La limite est que vous ne pouvez inclure qu'un seul fichier de
; instruments pour toutes les chansons
La limitation est résolue en mettant simplement tous les
dans un seul fichier.

fichier de l'instrument. NOTEZ QU'IL NE DOIT Y AVOIR
QU'UN SEUL fichier lire      "instruments.mus.asm"

; fichiers musicaux SONG_0 :
INCBIN      "micancion.mus" ;
SONG_0_END :

SONG_1 :
```

```

INCBIN      "another_song.mus"
; SONG_1_END :

SONG_2 :
INCBIN      "third_song.mus"      ;
SONG_2_END :

CHANSO
N_3      :
CHANSO
N_4      :
CHANSO
N_5      :
CHANSO
N_6      :
CHANSO
N_7 :

```

Enfin, vous réassemblez la bibliothèque 8BP pour que le lecteur de musique (qui est intégré dans la bibliothèque) connaisse les paramètres des instruments et l'endroit où les chansons ont été assemblées.

Pour ce faire, il suffit d'assembler le fichier make_all.asm, qui ressemble à ceci

```

Makefile pour les jeux vidéo utilisant la puissance 8bit
si vous ne modifiez qu'une seule pièce, vous n'avez qu'à assembler la
pièce correspondante
par exemple, vous pouvez assembler le make_graphics si vous changez de
dessin
;-----CODIGO -----
;comprend la bibliothèque 8bp et le lecteur de
musiqueWYZ lire "make_codigo.asm".

;-----MUSICA-----
;inclus les chansons. lire
"make_musica.asm"

; ----- GRAPHIQUES -----
;Cette partie comprend des images et des séquences d'animation.
;et la table des sprites initialisée avec ces images et séquences lues dans
"make_graphics.asm".

```

Et voilà, tout est assemblé. Vous devez maintenant générer votre bibliothèque 8BP comme suit :

SAVE "8BP.LIB", b, 24000, 8200

Et la musique :

SAVE "music.bin", b, 32200, 1400

19.3 Que faire si vous ne pouvez pas faire tenir de la musique dans 1400 octets ?

Il est possible que 1400 octets ne suffisent pas pour vos chansons. Si une chanson ne

rentre pas (et vous le saurez en vérifiant où la balise "_END_MUSIC" est assemblée), vous pouvez spécifier une adresse d'assemblage différente pour cette chanson et les suivantes. Dans le cas du jeu vidéo "Nibiru", vous faites cela avec la troisième chanson, en l'assemblant à une adresse inférieure à celle de la bibliothèque 8BP (par exemple 23000). Dans ce cas, votre jeu BASIC devra commencer par un

La commande MEMORY spécifiant une adresse inférieure à cette nouvelle limite, par exemple MEMORY 22999, fonctionnera.

À partir de la version 34 du 8BP, la bibliothèque est assemblée à partir de l'adresse 24000. Si vous souhaitez utiliser de l'espace supplémentaire pour la musique, celle-ci doit occuper des adresses inférieures à 24000. Par exemple, de 23500 à 23999.

Il s'agit de la

```
après l'assemblage, sauvegardez-le avec save
"musica.bin",b,32200,1400
org 32200
;-----MUSICA-----
est limité à la possibilité d'inclure un seul fichier de ; instruments
pour les
La limitation est résolue en insérant simplement toutes les chansons.
dans un seul fichier.

fichier de l'instrument. NOTEZ QU'IL NE DOIT Y AVOIR
QU'UN SEUL fichier lire      ".../MUSIC/nibiru5.mus.asm"
;

; fichiers musicaux SONG_0 :
INCBIN      ".../MUSIC/atack5.mus"
; SONG_0_END :

SONG_1 :
INCBIN      ".../MUSIC/nibiru5.mus" ;
SONG_1_END :

org 23500 ; J'utilise cette ligne parce que je ne peux pas mettre la troisième
chanson ! !
SONG_2 :
INCBIN      ".../MUSIC/gorgo3.mus" ;

SONG_3 :
CHANSO
N_4   :
CHANSO
N_5   :
CHANSO
N_6   :
CHANSO
N_7 :
FIN_MUSIQU
E
```

Le même type de solution s'applique dans le cas où vous ne pouvez pas faire tenir tous vos graphiques dans la zone réservée par 8BP, bien que vous disposiez de 8540 octets pour les graphiques et que vous soyez moins susceptible de rencontrer ce problème.

20 Programmation en C avec 8BP

Au début de ce manuel, je vous ai recommandé de ne pas utiliser les compilateurs BASIC comme Fabacom ou CPC BASIC 3 à cause des limitations de mémoire qu'ils imposent (vous perdez environ 20 Ko). Si ce que vous voulez, c'est augmenter la vitesse de vos jeux, à partir de 8BP v40 vous avez à votre disposition un wrapper de la bibliothèque 8BP à utiliser à partir du C, et un petit ensemble de commandes BASIC invocables à partir du C (que j'appelle "minibasic") de sorte que vous pouvez traduire votre programme BASIC presque immédiatement et obtenir le résultat rapide que vous recherchez.

Cette nouvelle fonctionnalité vous offre trois possibilités :

- 1) **Réalisez votre programme à 100% en BASIC** (c'est-à-dire n'utilisez pas les fonctionnalités). Cette option simplifie grandement la tâche de programmation, mais elle est moins rapide.
- 2) **Réaliser votre programme à 100% en C**. Cette option est complexe car la programmation, la compilation, la recherche et la correction des erreurs est une tâche beaucoup plus lente que la programmation en BASIC.
- 3) **Réalisez votre programme en BASIC et, à la fin, traduisez uniquement le cycle de jeu en C**. Cette option est aussi simple que la première, à l'exception de la dernière phase de traduction en C.

Je vais vous expliquer la troisième option, qui est très intéressante parce qu'elle vous permet de programmer en BASIC avec toutes les commodités qu'il offre (facilité de programmation, de détection et de correction des erreurs, etc.) Si vous voulez programmer votre jeu entièrement en C, cette explication vous servira exactement de la même façon, alors n'ayez pas peur et continuez à lire. Un exemple commercial de l'option 3 est le célèbre et mythique jeu "galactic plague", une production d'Indescomp créée par l'excellent programmeur Paco Suárez en 1984. Nous devons de nombreuses heures de divertissement au grand Paco Suárez.

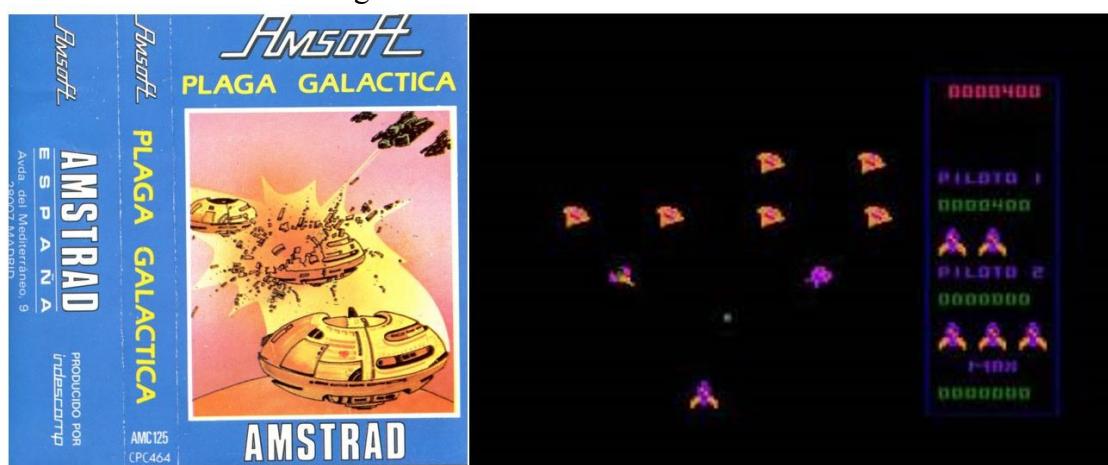


Fig. 112 La "peste galactique".

Pour pouvoir programmer en C, nous avons besoin de quelques outils, mais ne vous inquiétez pas, vous n'avez pas besoin d'apprendre à les utiliser car 8BP le fait pour vous, grâce à un .bat qui s'occupe de tout comme vous le verrez bientôt. Les outils sont :

- **compila.bat** : c'est le fichier .bat qui appelle les différents outils de cascade et qui, à partir d'un fichier .c, produit un fichier .dsk contenant un binaire. Vous trouverez cet outil dans le sous-répertoire "C" de 8BP.

- **SDCC** (Small Device C compiler) : vous devez le télécharger et l'installer. Vous le trouverez à l'[adresse http://sdcc.sourceforge.net/](http://sdcc.sourceforge.net/). Il s'agit probablement du meilleur compilateur C pour le Z80 (bien que SDCC prenne également en charge d'autres microprocesseurs). Il en existe un autre

J'ai choisi le Z88dk mais j'ai préféré choisir le SDCC car certains tests révèlent que le programme compilé avec le SDCC est plus rapide que l'équivalent en z88dk.

- **Hex2bin.exe** : nous allons l'utiliser (à partir du .bat) pour convertir le fichier que nous avons généré en hexadécimal avec SDCC en binaire. Pas besoin de le télécharger, il est petit et vous le trouverez dans le sous-répertoire "C" de 8BP.
- **ManageDsk.exe** : nous l'utiliserons (à partir du fichier .bat) pour mettre notre binaire compilé dans un fichier .dsk. Vous n'avez pas besoin de le télécharger. Il est petit et vous le trouverez dans le sous-répertoire "C" de 8BP.

Passons en revue les étapes nécessaires à l'aide d'un exemple, puis je détaillerai comment invoquer chacune des fonctions 8BP à partir du C, ainsi que les nouvelles commandes "minibasic" que le 8BPV40 met à votre disposition pour programmer en C comme si vous étiez en BASIC.

20.1 Première étape : programmer votre jeu **BASIC**

Nous allons utiliser le programme d'exemple fourni avec la bibliothèque 8BP. Il s'agit d'un jeu très simple dans lequel vous incarnez un soldat qui doit esquiver des balles tombant du ciel dans différentes directions. Chaque fois qu'une balle vous touche, vous perdez des points, qui augmentent avec le temps.



Fig. 113 L'ensemble d'exemples

La liste du jeu est la suivante, dans laquelle j'ai souligné en rouge la partie qui correspond au "cycle du jeu".

```

10 MÉMOIRE 19999
11 LOAD "loop.bin",20000 : REM charge la boucle de jeu compilée
20 MODE 0 : DEFINT A-Z : CALL &6B78:' install RSX
30 ON BREAK GOSUB 320
40 CALL &BC02 : "restaurer la palette par défaut au cas où".
50 INK 0,0 : "fond noir"
60 FOR j=0 TO 31:|SETUPSP,j,0,0:NEXT:|3D,0:'reset sprites
70 |SETLIMITS,0,80,0,124 : ' définit les limites de l'écran de jeu
80 PLOT 0,74*2:DRAW 640,74*2
90 x=40:y=100:' coordonnées du caractère
100 PRINT "SCORE :      FPS :"

110 |SETUPSP,31,0,1+32:' statut du caractère
    |SETUPSP,31,7,1 "Séquence d'animation assignée au démarrage
130 |LOCATESP,31,y,x:'place le sprite (sans l'imprimer)
140 |MUSIC,0,0,0,0,5 : points=0

```

```

150 cor=32:cod=32:|COLSPALL,@cor,@cod:' définir la commande de collision
    LOCATE 1,20:INPUT "basic(1) or C(2)", a : IF a=1 THEN 160 ELSE CALL &56b0
    GOTO 320
160 |PRINTSPALL,0,0,0,0 : 'configurer la commande d'impression
161 POKE &B8B4,0 : POKE &B8B5,0 : POKE &B8B6,0 : POKE &B8B7,0 : 'reset timer
cpc6128. il est nécessaire parce que TIME peut renvoyer un très grand nombre et peut
don débordement avec DEDFINT
ner t1=TIME
162
170 cycle de jeu. C'est la partie qui a été traduite en C ---  

    c=c+1
190 ' lit le clavier et positionne le caractère
191 IF INKEY(27)=0 THEN IF dir<>>0 THEN |SETUPSP,31,7,1:dir=0 ELSE
|ANIMA,31:x=x+1:GOTO 195
192 SI INKEY(34)=0 ALORS SI dir<>>1 ALORS |SETUPSP,31,7,2:dir=1 ELSE
|ANIMA,31:x=x-1
195 |LOCATESP,31,y,x
200 |AUTOALL:|PRINTSPALL
210 |COLSPALL
220 IF cod<32 THEN BORDER 7:dots=dots-1:LOCATE 7,1:PRINT points:GOTO
221 REM pour calculer le FPS nous prenons en compte que TIME me donne en unités 1/300
secondes et que je vais mesurer tous les 20 cycles. Donc fps= 20 cycles x 300
/ dt, où dt= t2-t1
230 SI c MOD 20=0 ALORS points=points+10 :LOCATE 7,1:PRINT points :
t2=t1:t1=TIME:fps=6000/(t1-t2):LOCATE 17,1:PRINT fps
240 SI c MOD 5=0 ALORS |SETUPSP,i,9,9,19:|SETUPSP,i,5,4,RND*3-
1:|SETUPSP,i,0,11:|LOCATESP,i,10,RND*80 : i=i+1:IF i=30 ALORS i=0
    IF c<500 THEN GOTO 180
251 '--- fin du cycle de jeu ---
252 |POKE,42038,points
310 fin du jeu
320 |MUSIC:INK 0,0:PEN 1:BORDER 0:|PEEK,42038,@points
330 LOCATE 3,10:PRINT "SCORE FINAL :" ;dots

```

20.2 Étape 2 : traduire votre cycle de jeu BASIC en C

Pour traduire le cycle du jeu en C, nous devons écrire un programme C, que nous appellerons cycle.c.

Allez dans le sous-dossier "C" de 8BP. Vous y trouverez tout ce dont vous avez besoin et ce même exemple, avec le fichier ciclo.c.

La première chose que vous devez savoir lorsque vous programmez le cycle du jeu en C, c'est que vous devez inclure deux petites bibliothèques : le 8BP wrapper (8BP.h) et le minibasic (minibasic.h). Le 8BP.h se trouve dans le sous-répertoire "8BP_wrapper" et le minibasic.h dans le sous-répertoire "mini_basic".

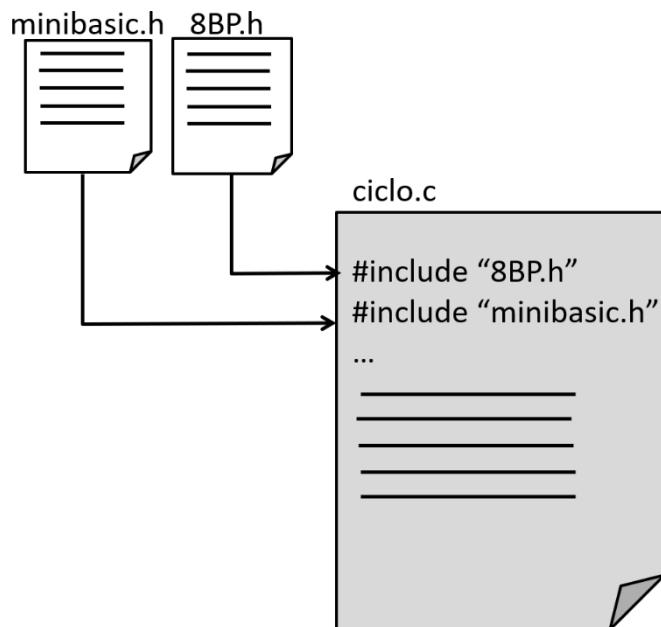


Fig. 114 fichiers à compiler

Voici le listing C qui, comme vous pouvez le voir, correspond directement au morceau de listing BASIC **correspondant au cycle de jeu**, pratiquement une traduction littérale. Vous verrez quelques étiquettes comme "label_195" qui servent de numéros de ligne pour pouvoir sauter avec GOTO. Il s'agit pratiquement de la liste BASIC traduite instruction par instruction, sans qu'il soit nécessaire de repenser la façon de la programmer. Dans ce cas simple, il n'y a qu'une seule fonction (la fonction principale) et elle revient lorsque le temps est écoulé.

Pour franchir cette étape vous avez le "minibasic" pour vous aider dans la traduction du BASIC vers le C, mais si vous êtes un expert en C et que vous connaissez le firmware AMSTRAD ou que vous avez une autre bibliothèque d'aide vous pouvez faire le cycle de jeu directement en C, sans l'avoir préalablement programmé et validé en BASIC. La méthode que je vous propose est simple et puissante, votre programme tournera comme un lèvrier, mais ici vous êtes libre de faire ce que vous voulez.

IMPORTANT : n'oubliez pas, lorsque vous programmez en C, que la notation des nombres décimaux, hexadécimaux et binaires est la suivante :

mivariable = 165 ; //notation décimale
mivariable = 0xA5 ; //notation hexadécimale
mivariable = 0b10100101 ; //notation binaire

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <stdio.h>

#include "8BP.h"
#include "minibasic.h"
#include "8BP.h"
#include "minibasic.h"

//déclarer les variables toutes globales afin de pouvoir
//accéder à eux à partir de n'importe quelle fonction, comme en BASIC
//bien qu'ils ne soient pas initialisés ici
//-----
int c;
```

```

char dir
; int x ;
int y ;
int cod ;
int cor ;
int i ;
int points ;
int t1 ;
int t2 ;
int fps ;
/*****************/
MAIN
/*****************/
int main()
{
    //initialise les variables c=0 ;
    dir=0 ;
    x=40 ;
    y=100
    ;
    cod=32
    ;
    cor=32
    ; i=0 ;
    points=0 ;
    fps=0 ;
    t1=_basic_time() ;

    //configurer les commandes
    _8BP_printspall_4(0,0,0,0);
    _8BP_colspall_2(&cor,&cod) ;

    //cycle de jeu
    //-----
    label_CICLE :
    c=c+1 ;

    if (_basic_inkey(27)==0) {
        if (dir !=0) {
            _8BP_setupsp_3(31,7,1) ;
            dir=0 ;
        }
        else {
            _8BP_anima_1(31) ;
            x=x+1 ;
            goto label_195 ;
        }
    }
    if (_basic_inkey(34)==0) {
        if (dir !=1) {

```

```

    _8BP_setupsp_3(31,7,2) ;
    dir=1 ;
}
else {
    _8BP_anima_1(31)
    ; x=x-1 ;
}
}

label_195 :
//-----
_8BP_locatesp_3(31,y,x);
_8BP_autoall() ;
_8BP_printspall() ;
_8BP_colspall() ;

if (cod<32) {
    La frontière de base (7) ;
    _basic_sound(1,100,14,0,0,1,0);
    puntos=puntos-1;
    _8BP_setupsp_3(cod,0,9);
    La valeur de l'argent est de l'ordre d'un million d'euros, mais
    la valeur de l'argent est de l'ordre d'un million d'euros ;
    _basic_print(_basic_str(dots)) ; goto
    _label_250 ;

}
else _basic_border(0)
;if (c %20 ==0) {
    points=points+10 ;
    La valeur de l'argent est de l'ordre d'un million d'euros, mais
    la valeur de l'argent est de l'ordre d'un million d'euros ;
    _basic_print(_basic_str(points)) ;
    t2=t1;t1=_basic_time() ;
    fps=6000/(t1-t2) ;
    _basic_locate(17,1);_basic_print(_basic_str(fps)) ;
}

if (c %5 ==0){
    _8BP_setupsp_3(i,9,19);
    _8BP_setupsp_4(i,5,4,_basic_rnd(3)-1);

    _8BP_setupsp_3(i,0,11);_8BP_locatesp_3(i,10,_basic_rnd(80)) ;
    i=i+1;if (i==30) i=0 ;
}

_label_250 :
if (c<500 goto label_CICLE ;

_8BP_poke_2(42038,points) ;

```

```
return 0 ;
```

}

Comme vous pouvez le voir, les fonctions 8BP sont invoquées comme suit :

8BP<fonction>_<N>(paramètres)

N étant le nombre de paramètres de la fonction, car il existe des versions de chaque fonction avec un nombre différent de paramètres (comme c'est le cas dans les versions RSX des commandes).

Les fonctions de base qui ne sont pas disponibles en C mais que nous avons créées dans minibasic sont invoquées comme suit :

basic<function>(parameters)

Comme vous pouvez le voir, en connaissant la traduction de chaque commande, il est très facile de traduire un listing BASIC de votre cycle de jeu en un listing cycle.c.

Une chose nécessaire à faire est de communiquer le LOCOMOTIVE BASIC avec le C. Par exemple, il y a des données que vous pouvez vouloir transmettre au programme C, comme le nombre de vies qu'il vous reste ou les points que vous avez accumulés. Pour cela, vous pouvez utiliser les fonctions :

- En LOCOMOTIVE BASIC, vous disposez de **PEEK**, **POKE**, **|PEEK** et **|POKE**.
- En C, vous avez **_basic_peek()**, **_basic_poke()**, **_8BP.Peek_2()**, **_8BP.Poke_2()**.

Avec ces fonctions, vous pouvez réserver une adresse mémoire pour stocker les vies, les points, la phase, etc. et ainsi invoquer le cycle de jeu à chaque fois que vous êtes tué avec tout le contexte du jeu dans quelques variables qui peuvent être lues et modifiées à la fois par BASIC et C. Dans l'exemple, vous pouvez voir comment cela est fait avec la variable "points".

IMPORTANT : même si tout votre programme est en C, vous devez initialiser 8BP avec un **CALL &6b78**, car en plus d'installer les commandes RSX, cet appel initialise également les tables de la bibliothèque interne.

20.2.1 GOSUB et RETURN en C

Les GOSUB doivent être traduits en fonctions car vous pouvez accéder à une routine GOSUB à partir de n'importe quel endroit du programme et vous devez pouvoir revenir. En BASIC, vous retournez avec RETURN à l'endroit où vous avez invoqué GOSUB. En langage C, la chose naturelle à faire est de traduire cette routine en fonction, afin de pouvoir retourner à l'endroit du programme d'où vous l'avez appelée.

Prenons un exemple :

BASIC

C

<pre> 10 a=5 20 GOSUB 100 30 PRINT "PEPE" 40 fin 100 REM ROUTINE 110 PRINT STR\$(a) 120 RETOUR </pre>	<pre> #include <stdlib.h> #include <string.h> #include <stdio.h> #include <stdio.h> #include "8BP.h" #include "minibasic.h" #include "8BP.h" #include "minibasic.h" </pre>
	<pre> void mifucion(int id) ; int a ; int main() { a=5 ; mifucion(a) ; _basic_print(_"PEPE") ; retour 0 ; } void mifucion(int a) { _basic_print(_basic_str(a)) ; _basic_print("\r") ; } </pre>

20.2.2 Communication entre BASIC et C avec des variables BASIC

Si vous commencez à utiliser le C avec le 8BP, vous pouvez passer à l'étape suivante. Il s'agit d'une section "avancée" qui vous apprend à communiquer entre BASIC et C avec des variables BASIC au lieu de PEEK/POKE, mais elle n'est pas essentielle.

Pour communiquer BASIC avec C, au lieu d'une adresse mémoire et de PEEK/POKE, vous pouvez également utiliser une variable qui existe en BASIC. C'est un peu plus complexe, mais c'est possible. Nous allons voir comment le faire avec une variable simple, puis nous verrons comment le faire avec des variables de type tableau.

La première chose que vous devez savoir est comment trouver l'endroit où BASIC stocke une variable. Pour cela, nous disposons de l'opérateur "@". Voyons un exemple simple :

10 DEFINT A-Z : "important que le type de données soit int 20 a=5 30 print @a : 'imprime l'adresse mémoire où est stocké a 40 poke @a,7 : 'c'est la même chose que de faire a=7 50 PRINT a : ' ceci imprime un 7

Il y a une petite "erreur" dans le programme. Il s'agit de l'instruction POKE @a,7 parce qu'elle ne stocke qu'un octet et que la variable "a" a 2 octets parce qu'il s'agit d'un entier. Ce cas fonctionne parce que l'octet le plus significatif de "a" est zéro, mais si "a" avait une valeur supérieure à 255, son octet le plus significatif ne serait pas zéro et l'instruction POKE ne modifierait que l'octet le moins significatif. Un POKE 8BP fonctionnerait toujours car il s'agit de 16 bits :

| **POKE,@a,7 : 'met un 7 dans la variable "a".**

```
a=1000
Ready
print a
1000
Ready
print Ca
432
Ready
poke Ca,7
Ready
print a
775
Ready
```

Sachant cela, il ne nous reste plus qu'à transmettre au C l'adresse où est stockée notre variable BASIC. Pour cela, nous disposons de l'instruction |POKE de 8BP qui fonctionne sur 16 bits (rappelons qu'une adresse mémoire occupe 16 bits). Nous allons passer l'adresse de la variable "a" à l'adresse 40000, bien que nous puissions utiliser n'importe quelle autre adresse libre.

|POKE, 40000, @a:**nous laissons @a à l'adresse 40000**

Une autre méthode en BASIC consiste à utiliser deux POKE :

dir=@a :

**POKE 40001, INT (dir/256) POKE
40000, INT (dir MOD 256)**

Ce que nous avons écrit à l'adresse 40000 est l'adresse mémoire où la variable a est stockée.

IMPORTANT : BASIC peut relocaliser une variable lorsque de nouvelles variables sont créées. Cela signifie que si vous transmettez une adresse mémoire à une routine C via une commande |POKE et que vous créez ensuite de nouvelles variables BASIC, l'adresse mémoire de la variable que vous avez transmise peut avoir changé. **Elle n'est garantie de ne pas changer que si aucune nouvelle variable n'est créée.** L'exemple suivant l'illustre très bien, il y a 2 changements d'emplacement

<pre>10 DIM a(100) : i=0 20 PRINT @a(0) 30 b=2:'nouvelle variable relocalise a() 40 PRINT @a(0) 50 c=2:'nouvelle variable relocalise a() 60 PRINT @a(0) 70 goto 20</pre>	
--	--

La solution à ce problème consiste à déclarer toutes les variables au début du programme.

<pre>10 dim a(100) 20 b=2 30 c=3 40 print @a(0) 70 goto 40</pre>	
--	--

En C, nous pouvons maintenant accéder à la variable "a" de deux manières, bien que la seconde (au moyen d'une variable C "mappée") soit la plus intéressante.

<pre>// variables globales int dir_a ; int data ; int main() { //stockons dans dir_a l'adresse de la variable a _8BP_peek_2(40000, &dir_a) ; _8BP_peek_2(dir_a, &data) ; //cela met la valeur de la variable BASIC "a" //dans la variable C "data". C'est une façon de lire la valeur de "a". _8BP_poke_2(dir_a,7) ; //cela met un 7 dans la variable BASIC "a". retour 0 ; }</pre>
--

Voyons le même exemple d'une autre manière, avec une variable C qui est "mappée" à la variable de base. Pour cela, nous devons utiliser la notation avec "*".

```
// variables globales
int dir ;
int *a ;

int main()
{
//stockons dans dir_a l'adresse de la variable a
_8BP_peek_2(40000, &dir) ;
a=dir ; //pour éviter l'avertissement de compilation, utiliser a=(int*)dir
*a=5 ; //cela met un 5 dans la variable BASIC "a".
retour 0 ;
}
```

Nous avons vu comment cela fonctionne avec des variables simples. Nous allons maintenant voir comment fonctionnent les tableaux BASIC afin d'y accéder à partir du langage C. La première chose à faire est de comprendre comment BASIC stocke les données des tableaux en mémoire.

```
1 DEFINT a-z
2 MODE 2
10 a=5
20 PRINT "le dir de a est @a=";@a
25 PRINT "-----"
30 DIM b(5)
40 POUR i=0 à 5
50 PRINT "le dir de b(";i ;")
est ";@b(i)
60 NEXT
70 PRINT "-----"
80 DIM c(3,4)
90 FOR j=0 TO 4:FOR i=0 TO 3
100 PRINT "le dir de c(";i ;",";j ;")
est ";@c(i,j)
110 NEXT:NEXT

120 |POKE,40000,@c(0,0)
```

Avec la ligne 120, nous avons stocké à l'adresse 40000, l'adresse mémoire où sont stockées les premières données du tableau bidimensionnel.

Nous pourrions stocker celle du tableau unidimensionnel avec
|POKE,40000, @b(0)

```
la dir de a es @a= 681
-----
la dir de b( 0 ) es 698
la dir de b( 1 ) es 700
la dir de b( 2 ) es 702
la dir de b( 3 ) es 704
la dir de b( 4 ) es 706
la dir de b( 5 ) es 708
-----
la dir de c( 0 , 0 ) es 727
la dir de c( 1 , 0 ) es 729
la dir de c( 2 , 0 ) es 731
la dir de c( 3 , 0 ) es 733
la dir de c( 0 , 1 ) es 735
la dir de c( 1 , 1 ) es 737
la dir de c( 2 , 1 ) es 739
la dir de c( 3 , 1 ) es 741
la dir de c( 0 , 2 ) es 743
la dir de c( 1 , 2 ) es 745
la dir de c( 2 , 2 ) es 747
la dir de c( 3 , 2 ) es 749
la dir de c( 0 , 3 ) es 751
la dir de c( 1 , 3 ) es 753
la dir de c( 2 , 3 ) es 755
la dir de c( 3 , 3 ) es 757
la dir de c( 0 , 4 ) es 759
la dir de c( 1 , 4 ) es 761
la dir de c( 2 , 4 ) es 763
la dir de c( 3 , 4 ) es 765
Ready
```

Le programme ci-dessus nous apprend comment BASIC stocke les variables.

- Les données d'un tableau unidimensionnel sont stockées sur une seule ligne. Chaque donnée occupe 2 octets car nous travaillons avec des nombres entiers.

Par exemple, en BASIC, vous tapez :

```
DIM b(20)
|POKE,40000,@b
CALL <routine C>:'adresse où main() a été assemblé
```

PRINT b(8)

et de C que vous écrivez :

```
int dir;
int *b; //variable avec laquelle nous allons accéder au
tableau BASIC int main(){
_8BP.Peek_2(40000, &dir);
b= dir; //b est un pointeur et *b[] sont des variables
*b[8]=5; //ajoute un 5 à la variable BASIC b(8)
return 0;
}
```

- Les données des tableaux bidimensionnels sont stockées consécutivement et les variations de la première dimension produisent des données consécutives. Dans l'exemple avec **DIM(3,4)**, la donnée (2,3) suit la donnée (1,3) mais la donnée (2,3) est $4 \times 2 = 8$ octets plus loin que la donnée (2,2), parce que la première dimension du tableau est 4. **Un DIM (3,4) est un tableau de dimensions (4, 5) parce que le zéro compte aussi.** Vous pouvez le vérifier en imprimant @c(3,2) et @c(2,2), vous verrez qu'il y a une différence de 8. Pour accéder aux données à partir de C, on peut utiliser un simple pointeur en tenant compte de la première dimension lors de l'accès. Dans l'exemple suivant, j'ai créé un tableau c(12,16) et pour accéder à la position (2,7) à partir de C, j'utilise $2 + 7 \times 13$, puisque la première dimension est $12 + 1 = 13$.

En BASIC, vous écrivez :

```
DIM c(12,16)
|POKE, 40000, @c
CALL <routine C>;' adresse de la routine assemblée
PRINT c(2,7)
```

et de C que vous écrivez :

```
int dir;
int *b; //variable avec laquelle nous allons accéder au tableau
BASIC int main(){
_8BP.Peek_2(40000, &dir); //lire le dossier 40000 et écrire
dans le dossier c= dossier; //c est un pointeur et *c[] sont des
variables c[2+7*13]=5; //ajoute un 5 à la variable BASIC c(2,7)
return 0;
}
```

Si vous êtes un programmeur C, vous savez qu'il existe en C des pointeurs doubles exprimés par un double astérisque (par exemple `**c`). A priori, ils semblent convenir car on peut référencer des données avec `c[x][y]`. Cependant, ils ne servent qu'à stocker dynamiquement de la mémoire réservée avec les instructions "malloc" et "calloc" et nécessitent de la mémoire à la fois pour les données et les pointeurs dans chaque ligne. Cela signifie que vous pouvez les utiliser, mais que vous devez réserver de la mémoire pour les pointeurs. Je ne les recommande pas.

20.2.3 Chaînes de texte BASIC et C

Vous devez savoir qu'une chaîne en C est un simple pointeur sur un caractère, alors qu'une variable de chaîne en BASIC (par exemple `mivar$="hello"`) consiste en un descripteur de 3 octets contenant l'adresse mémoire où se trouve la chaîne et la longueur de la chaîne. En d'autres termes, **une chaîne de caractères en C et une chaîne de caractères en BASIC sont deux choses différentes.**

Ainsi, si vous voulez imprimer une variable contenant une chaîne de texte, vous ne pouvez pas la passer du BASIC au C car ce ne sont pas les mêmes choses. Mais vous pouvez imprimer des chaînes de texte sans problème si, au lieu de les passer de BASIC, vous les définissez en C, par exemple :

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <stdio.h>
#include "8BP.h"
#include "minibasic.h"
#include "8BP.h"
#include "minibasic.h"

//----globalvariables
char* cad ; //pourrait être initialisé ici

//----fonctions-----
int main(){
    cad="vous avez échoué dans votre mission" ; //nous l'initialisons avec une
    phrase
    _basic_print(cad) ; //imprimer dans le style BASIC
    _8BP_printat4(0,0,60,cad) ; //imprime le style 8BP printat
    return 0 ;
}
```

20.3 Troisième étape : Compiler en utilisant "compila.bat".

Nous sommes sur le point de passer à la grande étape : la compilation pour générer un binaire AMSTRAD. Pour cette étape, vous disposez d'un fichier d'aide .bat, appelé "compila.bat". **Avant de réaliser cette étape, vous devez avoir installé le compilateur SDCC sur votre ordinateur, sinon l'opération échouera.** Vous pouvez le télécharger à partir de <http://sdcc.sourceforge.net/>.

Note : Sous Windows 10, l'exécution de SDCC a échoué parce qu'il est installé dans "Program files" et que les espaces blancs ne semblent pas lui convenir. Si c'est votre cas, installez-le dans un répertoire dont le nom ne comporte pas d'espace blanc.

Une fois SDCC installé, ouvrez une fenêtre de commande (le script ses.bat fait cela) et tapez compila.bat. L'écran changera de couleur : vert si tout s'est bien passé et rouge s'il y a des problèmes. Le script "compila.bat" exécute les étapes suivantes :

1. Supprime les fichiers de sortie du script lui-même (ciclo.dsk et ciclo.bin entre autres).
2. Invitez SDCC à compiler votre programme
3. Traduire la sortie SDCC en binaire à l'aide de l'outil hex2bin
4. Insérez le binaire généré dans un disque que vous appelez cycle.dsk, à l'aide de l'outil manageDsk.

Après avoir exécuté "compila.bat", vous obtiendrez un écran rouge dans les cas suivants :

- Erreurs de syntaxe dans votre programme C
- Le fichier ciclo.dsk que compila.bat va générer est ouvert par winape. Dans ce cas, vous devez connecter un autre disque à winape pour que compila.bat puisse

le recréer.

- Erreurs de compilation telles qu'une bibliothèque que vous avez essayé d'utiliser et que vous n'avez pas incluse, etc.

En cas d'erreurs de syntaxe C, ou d'autres erreurs C, ou si le fichier ciclo.dsk qui génère compila.bat est ouvert par winape, vous obtiendrez un écran d'erreur rouge comme celui-ci :

```
8888  BBBB  PPPPPP
88 88  BB  BB  PP  PP
88 88  BB  BB  PP  PP
8888  BBBB  PPPPP
88 88  BB  BB  PP
88 88  BB  BB  PP
8888  BBBB  PPPP
8 bits de poder . Un tributo al AMSTRAD CPC
Jose Javier Garcia Aranda 2016-2020

*****
*      compilacion con SDCC      *
*****
borramos los ficheros de compilacion anterior
*****



main.c : compilamos y linkamos, generando un main.ihx
*****
sdcc -mz80 --verbose --code-loc 20000 --data-loc 0 --no-std-crt0 --
BP_wrapper -Imini_BASIC ciclo.c
sdcc: Calling preprocessor...
sdcc: sdcpp -nostdinc -Wall -std=c11 -I"8BP_wrapper" -I"mini_BASIC"
SDCC_CHAR_UNSIGNED -D__SDCC_INT_LONG_REENT -D__SDCC_FLOAT_REENT -D_
__SDCC_VERSION_MINOR=0 -D__SDCC_VERSION_PATCH=0 -D__SDCC_REVISION=1
=1 -D__STDC_NO_THREADS_=1 -D__STDC_NO_ATOMICS_=1 -D__STDC_NO_VLA_
DC_UTF_16_=1 -D__STDC_UTF_32_=1 -isystem "C:\proyectos\proyectos0
" -isystem "C:\proyectos\proyectos09\_personal\8BP\SDCC\bin..\incl
sdcc: Generating code...
ciclo.c:36: syntax error: token -> 'puntos' ; column 8
ciclo.c:37: error 1: Syntax error, declaration ignored at 'fps'
ciclo.c:38: error 1: Syntax error, declaration ignored at 't1'
ciclo.c:41: syntax error: token -> '0' ; column 21
.
"+-----+
 "| HAY ERRORES DE COMPILACION! |
"+-----+"

C:\proyectos\proyectos09\_personal\8BP\V40\PROYECTO_V40_clean\C>
```

Fig. 115 Compila.bat se plaint d'erreurs C

Si tout se passe bien, le résultat sera le suivant

```

transformamos el .ihx en un .bin
*****
hex2bin -output\ciclo.ihx
hex2bin v1.0.1, Copyright (C) 1999 Jacques Pelletier
Lowest address = 00004E20
Highest address = 00005A41

metemos el .bin en un disco de amstrad cpc
*****
managedisk -C -S"output\ciclo.dsk"
managedisk -L"output\ciclo.dsk" -I"output\ciclo.bin"/CICLO.BIN/BIN/20

*****
**          FIN DEL PROCESO
**  ASEGUrate DE QUE NO EXCEDES LA DIRECCION 24000
** es la (highest address) de la transformacion ihx en bin
** 
** se ha generado ciclo.dsk y dentro esta ciclo.bin
** 
** Pasos para cargarlo en el amstrad
** 1) carga o ensambla 8BP, con tus graficos, musica etc
** 2) carga tu juego BASIC
** 3) ejecuta LOAD "ciclo.bin", 20000
** para invocar a tu programa o rutina simplemente:
** call <direccion de main en fichero ciclo.map>
** 
** Para mover ciclo.bin de ciclo.dsk a otro disco debes
** conocer su longitud:
**   longitud=Highest address - Lowest address
**   lo cargas desde ciclo.dsk
**   LOAD "ciclo.bin", 20000
**   Y salvas en el disco donde esta tu juego
**   SAVE "ciclo.bin",b,20000,longitud
*****
C:\proyectos\proyectos09\_personal\8BP\V40\PROYECTO_V40_clean\C>
```

Fig. 116 Compila.bat produit un écran vert : tout s'est bien passé.

Si vous obtenez un écran vert, cela signifie que toutes les étapes de compila.bat se sont bien déroulées (borSDCC, vous aurez des informations précieuses à l'écran et dans le sous-répertoire de sortie vous trouverez les fichiers dont vous avez besoin :

- Cycle.dsk
- Cycle.map

20.4 Quatrième étape : vérifier les limites de la mémoire

Le script compila.bat affiche sur son écran vert deux informations très précieuses : "l'adresse la plus basse" et "l'adresse la plus haute". Il s'agit des adresses mémoire où le binaire généré commence et se termine. Le script "compila.bat" utilise l'adresse 20000 comme adresse de compilation dans l'invocation SDCC et si le fichier binaire résultant est très grand, il pourrait dépasser l'adresse 24000, endommageant ainsi la bibliothèque 8BP. Vous devez vous assurer que la "plus haute adresse" est inférieure à 24000. Si elle n'est pas inférieure, **vous devez modifier l'invocation SDCC dans le script "compila.bat"**.

à compiler en lui assignant une adresse inférieure à 20000. De cette façon, votre nouveau binaire et la bibliothèque 8BP ne se chevaucheront pas. Vous devez modifier deux lignes du script compila.bat. La première est celle qui invoque SDCC. C'est une très longue ligne. Vous devez remplacer le paramètre 20000 par la nouvelle adresse

```
sdcc -mz80 --verbose --code-loc 20000 --data-loc 0 --no-std-crt0 --fomit-frame-pointer --opt-code-size -l8BP_wrapper -lmini_BASIC -o output/cycle.c
```

La deuxième ligne que vous devez modifier est celle qui invoque managedsk pour qu'elle soit cohérente avec la nouvelle adresse mémoire. Voici cette ligne et, comme vous pouvez le voir, l'adresse 20000 apparaît également.

```
managedsk -L "output "cycle.dsk" -l "output "cycle.bin"/CYCLE.BIN/BIN/2000000 -I "output\cycle.bin"/CICLO.BIN/BIN/20000 -S "output\ciclo.dsk" -l "output\ciclo.bin"/CICLO.BIN/BIN/20000
```

Il est évident que si votre binaire commence à 20000, votre programme BASIC devra incorporer une MEMOIRE de 19999. Cela soustraira 4KB de votre espace libre mais vous sauvegarderez aussi les lignes BASIC correspondant au cycle du jeu, une chose pour une autre et c'est comme si vous n'aviez rien perdu.

Si l'adresse la plus élevée est inférieure à 24000, **vous devez l'ajuster autant que possible, c'est-à-dire la rapprocher le plus possible de 24000, afin de ne pas gaspiller de mémoire.** Cela peut impliquer (par exemple) l'utilisation de l'adresse 21000 lors de l'invocation du SDCC. Faites cela afin de disposer d'autant de mémoire que possible pour BASIC. Vous devez placer une MEMOIRE correspondant à cette adresse dans votre programme BASIC. Par exemple, si le binaire commence à 21000, vous devrez définir une MEMOIRE de 20999.

En fin de compte, vous devrez peut-être modifier deux lignes du script "compila.bat" pour ajuster l'adresse de démarrage de la compilation, que j'ai initialement fixée à 20000.

20.5 Étape 5 : Localisez l'adresse de la fonction à invoquer à partir de BASIC.

Après avoir compilé avec "compila.bat", vous aurez obtenu dans le sous-répertoire "output" un fichier appelé ciclo.map. Dans ce fichier, vous devez trouver l'adresse mémoire de la ou des fonctions que vous avez l'intention d'invoquer à partir de BASIC. Dans cet exemple, nous allons seulement invoquer la fonction main(), qui est située à &56b0 comme on peut le voir dans ce fragment du fichier ciclo.map

000056A0	_basic_paper
0000566B	_basic_plot
00005682	_basic_move
00005699	_basic_draw
000056B0	main
00005920	_abs
0000592C	_strlen
0000593B	_modschar
00005948	_modsint
00005954	_moduchar

Fig. 117 l'adresse de chaque fonction se trouve dans ciclo.map

Cela signifie que pour invoquer la fonction main() à partir de BASIC, il suffit de faire :

APPEL &56B0

Attention car si vous faites des changements dans la phase de compilation (modification du cycle .c ou changement des adresses mémoire du script compila.bat) l'adresse de chaque fonction peut changer lors de la recompilation.

20.6 Étape 6 : Inclure le nouveau binaire dans votre jeu .dsk

Vous avez déjà généré un fichier cycle.dsk contenant le fichier cycle.bin que vous devez charger pour invoquer la fonction principale (et/ou toute autre fonction que vous souhaitez). Pour obtenir ce fichier et votre jeu sur le même disque, vous devez sélectionner cycle.dsk dans winape et charger simplement le fichier cycle.bin.

LOAD "CYCLE.BIN",20000

Ensuite, dans le menu winape, vous sélectionnez votre disque (où se trouve votre jeu) et vous sauvegardez ce binaire.

SAVE "CICLO.BIN", b, 20000, <longueur>.

où longueur = adresse la plus haute - adresse la plus basse +1

Maintenant, dans votre fichier loader.bas, vous devez charger ce nouveau binaire supplémentaire et l'invoquer à partir de votre listing BASIC avec CALL <address>.

```
10 MÉMOIRE 24999
15 LOAD "!pant.scr",&c000 : 'seulement si votre jeu a un écran de chargement'
20 LOAD " yourgame.bin"
25 LOAD "cycle.bin", 20000
50 RUN " !yourgame.bas"
```

C'est tout. Vous pouvez maintenant programmer en C avec 8BP !

20.7 Référence de la fonction 8BP en C

RSX	C prototype
3D, 0 3D, <flag>, #, offsety	void _8BP_3D_1(int flag) ; void _8BP_3D_3(int flag, int sp_fin, int offsety) ;
ANIMA, #	void _8BP_anima_1(int sp) ;
ANIMALL	void _8BP_animall() ;
AUTO, #	void _8BP_auto_1(int sp) ;
AUTOALL, <flag routed>, <flag routed>, <flag routed>, <flag routed>, <flag routed>.	void _8BP_autoall() ; void _8BP_autoall_1(int flag) ;
COLAY, threshold_ascii, @collision, # COLAY, @collision, # COLAY, # COLAY	void _8BP_colay_3(int threshold, int* collision, int sp) ; void _8BP_colay_2(int* collision, int sp) ; void _8BP_colay_1(int sp) ; void _8BP_colay() ;
COLSP, #, @collided%, @collided%, @COLSP, #, @collided%, @collided%. COLSP, 32, ini, end COLSP, 33 ans, @collided% COLSP, # COLSP, 34, dy, dx	/* opération 32, ini, fin ou opération 34, dy, dx */ void _8BP_colsp_3(int operation, int a, int b) ; /* opération 33 ou sp */ void _8BP_colsp_2(int sp, int* collision) ; void _8BP_colsp_1(int sp) ;
COLSPALL, @qui%, @qui%, @avecqu i% COLSPALL, collisionneur COLSPALL	void _8BP_colspall_2(int* collider, int* collided) ; void _8BP_colspall_1(int collider_ini) ; void _8BP_colspall() ;
LAYOUT, y, x, @string\$, @string\$, @string\$, @string\$, @string\$, @string\$.	void _8BP_layout_3(int y, int x, char* cad) ;
LOCATESP, #, y, x	void _8BP_locatesp_3(char sp, int y, int x) ;
MAP2SP, y, x MAP2SP, statut	void _8BP_map2sp_2(int y, int x) ; void _8BP_map2sp_1(unsigned char status) ;
MOVER, #, dy, dx	void _8BP_mover_3(int sp, int dy, int dx) ; void _8BP_mover_1(int sp) ;
dy,dx, dy,dx, dy,dx	void _8BP_moverall_2(int dy, int dx) ; void _8BP_moverall() ;
MUSIQUE, C, drapeau, chanson, vitesse MUSIQUE, drapeau, chanson, vitesse MUSIQUE	void _8BP_music_4(int flag_c, int flag_repetition, int song, int speed) ; void _8BP_music() ;
PEEK, dir, @variable%	void _8BP.Peek_2(int address, int* data) ;
POKE, dir, value	void _8BP.poke_2(int address, int data) ;
PRINTAT, flag, y, x, @string	void _8BP.printat_4(int flag, int y, int x, char* cad) ;
PRINTSP, #, y, x PRINTSP, # PRINTSP, 32, bits	void _8BP.printsp_1(int sp) ; void _8BP.printsp_2(int sp, int bits_background) ; void _8BP.printsp_3(int sp, int y, int x) ;
Le projet de loi a été adopté à l'unanimité par l'Assemblée nationale. L'impression de l'étiquette, le mode d'emploi, le mode d'emploi PRINTSPALL	void _8BP.printspall_4(int ini, int fin, int flag_anima, int flag_sync) ; void _8BP.printspall_1(int order_type) ; void _8BP.printspall() ;
RINK,tini,color1,color1,color2,...,color N RINK, sauter	void _8BP_rink_N(int num_params, int* ink_list) ; void _8BP_rink_1(int step) ;
ROUTESP, #, étapes	void _8BP_routesp_2(int sp, int steps) ; void _8BP_routesp_1(int sp) ;
ROUTEALL	void _8BP_routeall() ;
SETLIMITS, xmin, xmax, ymin, ymax	Void _8BP_setlimits_4 (int xmin, int xmax, int ymin, int ymax)

SETUPSP, #, nombre_de_paramètres, valeur SETUPSP, #, 5, Vy, Vx	void _8BP_setupsp_3(int sp, int param, int value) ; void _8BP_setupsp_4(int sp, int param, int value1,int value2) ;
STARS, initstar, num, colour, dy, dx	void _8BP_stars_5(int star_ini, int num_stars,int colour, int dy, int dx) ; void _8BP_stars() ;
UMAP,adr_ini, adr_end, yini, yfin, xini, xfin	void _8BP_umap_6(int map_ini, int map_fin, int y_ini, int y_fin, int x_ini, int x_fin) ;

Voici quelques exemples d'utilisation, qui complètent l'exemple utilisé dans la traduction de BASIC en C.

RSX	 3D, <flag>, #, offset
	10 3D,1,10,200
C	void _8BP_3D_3(int flag, int sp_fin, int offset) ; _8BP_3D_3(1, 10, 200) ;

RSX	 COLSPALL,@qui%,@qui%,@avecqui%
	10 collider%=0 : collided%=0 20 COLSPALL, @collider%, @colliderd%, @colliderd%
C	void _8BP_colspall_2(int* collider, int* collided) ; Int cor=0 ; Inr cod=0 ; _8BP_colspall_2 (&cor, &cod) ;

RSX	 RINK,tini,color1,color1,color2,...,colorN RINK, sauter
	10 BOIRE,1,2,2,2,3,3 20 RINK,1
C	void _8BP_rink_N(int num_params,int* ink_list) ; void _8BP_rink_1(int step) ; Int inks[5]={1,2,2,3,3} ; _8BP_rink_N(5,encres) ; _8BP_rink_1(1) ;

RSX	 LAYOUT, y, x, @string\$, @string\$, @string\$, @string\$, @string\$, @string\$. 10 cad\$="AA YYYY YYYY Y YY Y" 20 LAYOUT,10,1,@cad\$
C	void _8BP_layout_3(int y, int x, char* cad) ; _8BP_layout_3(10,1," YYYY YYYY Y YY Y" ; Ou bien : char* cad=" AA YYYY YYYY Y YY Y" ; _8BP_layout_3(10,1,cad)

RSX	 PRINTAT, flag, y, x, @string
	10 cad\$=str\$(125) 20 PRINTAT,0,100,40,@cad\$ 20 PRINTAT,0,100,40,@cad\$ 20
C	void _8BP_printat_4(int flg,int y,int x,char* cad) char* cad=_basic_str(125) ; _8BP_printat_4(0,100,40, cad) ;

20.8 Référence à une fonction BASIC en C ("minibasic")

Il s'agit d'un petit ensemble d'instructions de type BASIC que 8BP met à votre disposition par le biais de la bibliothèque minibasic.h, pour traduire facilement votre listing BASIC en C. Si vous essayez de traduire uniquement le cycle du jeu, cet ensemble d'instructions suffira.

BASIC	C prototype
BORDER	<code>void _basic_border(char colour) ; //exemple _basic_border(7)</code>
APPEL	<code>void _basic_call(unsigned int address) ; // exemple _basic_call(0xbd19)</code>
TIRAGE	<code>void _basic_draw(int x, int y) ;</code>
INK	<code>void _basic_ink(char ink1,char ink2) ;</code>
INKEY	<code>char _basic_inkey(char key) ; //prend environ 0,3 ms. lent mais simple</code>
LOCALISER	<code>void _basic_locate(unsigned int x, unsigned int y) ; // exemple : _basic_locate(2,25);_basic_print("TEST") ;</code>
DÉPLACER	<code>void _basic_move(int x, int y) ;</code>
PAPIER	<code>void _basic_paper(char ink) ;</code>
PEEK	<code>char _basic.Peek(unsigned int address) ;</code>
GRAPHIQUES PEN	<code>void _basic_pen_graph(char ink) ;</code>
PEN	<code>void _basic_pen_txt(char ink) ;</code>
POKE	<code>void _basic_poke(unsigned int address, unsigned char données) ;</code>
PLOT	<code>void _basic_plot(int x, int y) ;</code>
IMPRIMER	<code>void _basic_print(char *cad) ; //exemple : _basic_print("Hello")</code>
RND	<code>unsigned int _basic_rnd(int max) ; //exemple : num=_basic_rnd(50)</code>
SON	<code>void _basic_sound(unsigned char nChannelStatus, int nTonePeriod, int nDuration, unsigned char nVolume, char nVolumeEnvelope, char nToneEnvelope, unsigned char nNoisePeriod) ;</code>
STR\$	<code>char* _basic_str(int num) ; //similaire à STR\$ //exemple : _basic_print(_basic_str(num))</code>
TEMPS	<code>unsigned int _basic_time() ; //retourne un int non signé,(0..65535). En tant qu'entier, lorsque //atteindre 32768 aller à -32768</code>

21 Guide de référence de la bibliothèque 8BP

21.1 Fonctions de la bibliothèque

21.1.1 |3D

Cette commande active la projection 3D dans les commandes PRINTSP et PRINTSPALL Pour projeter, nous avons la commande |3D

Utilisation

Pour activer la projection 3D :

| La liste de ces éléments est la suivante : <3D, 1, <Sprite_fin>, <offsety>, <offsety>, <offsety>, <offsety>, <offsety>.

Pour le désactiver :

|3D, 0

Les sprites concernés vont de Sprite 0 à <Sprite_fin>. Cette commande active la projection 3D dans la commande |PRINTSP et dans |PRINTSPALL. Cela signifie qu'avant d'imprimer à l'écran, les coordonnées "projétées" seront calculées puis imprimées à l'écran. Les coordonnées des sprites ne sont pas affectées, c'est-à-dire que les coordonnées 2D dans la table des sprites restent les mêmes.

Cette commande **n'affecte pas les mécanismes de collision**, c'est-à-dire que si nous utilisons COLSPALL et détectons une collision entre des sprites projetés, la collision se produit dans le plan 2D.

Quant au dernier paramètre <offsety>, il s'agit de projeter plus ou moins haut, afin de pouvoir placer les marqueurs de jeu où l'on veut. Lorsque l'on projette l'écran, qui fait 200 pixels de haut, il devient 100 pixels de haut, nous pouvons donc choisir à quelle hauteur nous plaçons la projection. Si un sprite n'est pas affecté par la projection parce qu'il est plus haut que <Sprite_fin>, il n'est pas non plus affecté par <offsety>.

La figure suivante représente les coordonnées de la carte du monde qui sont projetées à certains points représentatifs de l'écran lorsque |MAP2SP est invoqué avec (yo=0, xo=0).

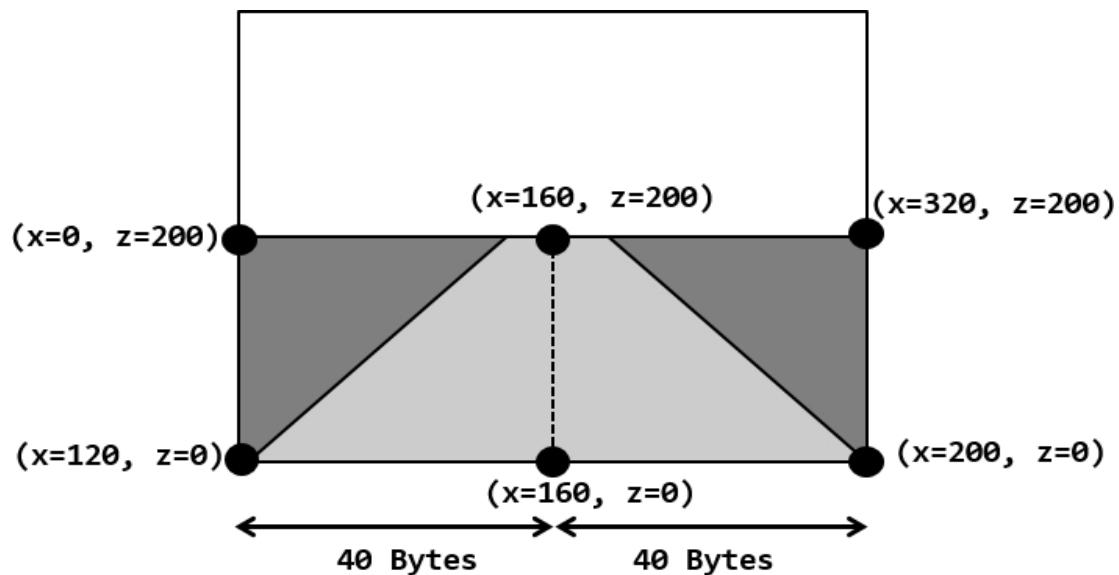


Fig. 118 Coordonnées mondiales projetées

Si, au lieu de ($x_0=0$, $y_0=0$), nous utilisons une autre coordonnée pour MAP2SP, les coordonnées 2D du monde correspondant aux points référencés dans l'image seront décalées de (x , z) comme indiqué par x_0 et y_0 .

21.1.2 |ANIMA

Cette commande modifie la trame d'animation d'un sprite, en tenant compte de la séquence d'animation qui lui a été attribuée.

Utiliser :

|ANIMA, <nombre de caractères>, <nombre de caractères>, <nombre de caractères>, <nombre de caractères>, <nombre de caractères>.

Exemple :

|ANIMA,3

La commande interroge la séquence d'animation du sprite, et si elle est non nulle, elle passe à la table des séquences d'animation (la première séquence valide est 1 et la dernière est 31). Elle choisit l'image dont la position est la plus proche de l'image courante et met à jour le champ frame de la table d'attributs du sprite.

Si la trame suivante de la séquence est nulle, elle est cyclée, c'est-à-dire que la première trame de la séquence est choisie.

Outre la modification du champ de la trame, le champ de l'image est modifié et l'adresse de la mémoire où la nouvelle trame est stockée est attribuée.

|ANIMA n'imprime pas le sprite, mais le laisse prêt à être imprimé, de sorte que la prochaine image de sa séquence soit imprimée.

|Le sprite doit être animé au moment où il se déplace et non pas au moment où il s'imprime, car l'ANIMA ne vérifie pas que le drapeau d'animation est actif dans l'octet de statut du sprite. En effet, notre personnage ne voudra normalement être animé que

lorsqu'il se déplace et pas toujours lorsqu'il est imprimé.

Si la séquence d'animation est une "séquence de mort" (elle comporte un "1" dans sa dernière image), le sprite devient inactif lorsqu'il atteint l'image dont l'adresse de la mémoire image est 1.

La bibliothèque 8BP vous permet de créer des "séquences de mort", c'est-à-dire des séquences qui, une fois terminées, mettent le sprite dans un état inactif. Cet état est indiqué par un simple "1" comme valeur de l'adresse mémoire de la dernière image. Ces séquences sont très utiles pour définir les explosions des ennemis qui sont animés avec |ANIMA ou

|ANIMALL. Après les avoir touchés avec votre tir, vous pouvez leur associer une séquence d'animation de mort et dans les cycles de jeu suivants, ils passeront par les différentes phases d'animation de l'explosion, et lorsqu'ils atteindront la dernière, ils passeront à l'état inactif, n'imprimant plus rien. Cet état inactif se fait automatiquement, donc ce que vous avez à faire est simplement de vérifier la collision de votre tir avec les ennemis et s'il entre en collision avec l'un d'entre eux, vous changez l'état avec |SETUPSP pour qu'il ne puisse plus entrer en collision et vous lui assignez la séquence d'animation de la mort, également avec |SETUPSP.

Si vous utilisez une séquence de mort, n'oubliez pas de vous assurer que la dernière image avant de trouver le "1" est complètement vide, afin qu'il n'y ait aucune trace de l'explosion.

Exemple de séquence de mort

```
dw EXPLOSION_1,EXPLOSION_2,ExpLOSION_3,1,0,0,0,0,0,0
```

21.1.3 |ANIMALL

Cette commande anime tous les sprites dont l'octet de statut contient le drapeau d'animation. Cette commande n'a pas de paramètres

Utilisation

|ANIMALL

IMPORTANT : depuis la version v37 de la bibliothèque, cette commande n'est accessible que via CALL (voir tableau des correspondances en annexe), et non plus via la commande RSX. En la supprimant de la liste, on a économisé quelques octets de mémoire et elle peut toujours être utilisée soit à partir d'un paramètre dans PRINTSPALL, soit à partir d'un appel CALL.

Elle est recommandée si vous devez animer un grand nombre de sprites, car elle est beaucoup plus rapide que d'invoquer plusieurs fois la commande **|ANIMA**.

Comme vous voudrez normalement invoquer **|ANIMALL** à chaque cycle de jeu, avant d'imprimer les sprites, il y a une façon plus efficace de l'invoquer, et c'est de mettre le paramètre correspondant de la commande **|PRINTSPALL** à "1", c'est à dire

|PRINTSPALL,1,0

Cette fonction invoque en interne **|ANIMALL** avant d'imprimer les sprites, ce qui permet d'économiser de l'argent.

1,17ms par rapport au temps qu'il faudrait pour invoquer séparément **|ANIMALL** et
|PRINTSPALL

21.1.4 |AUTO

Cette commande déplace un sprite (modifie ses coordonnées) en fonction de ses attributs de vitesse Vy,Vx. Ces attributs sont ceux que le sprite possède dans la table des sprites.

Utiliser :

|AUTO, <numéro d'emplacement>, <numéro d'emplacement>, <numéro d'emplacement>, <numéro d'emplacement>, <numéro d'emplacement>.

Exemple :

|AUTO, 5

Cette commande met à jour les coordonnées dans la table des sprites, en ajoutant la vitesse à la coordonnée actuelle.

Les nouvelles coordonnées sont
nouveau X = coordonnée X actuelle
+ Vx nouveau Y = coordonnée Y
actuelle + Vy

Il n'est pas nécessaire que le sprite ait le drapeau de mouvement automatique actif dans le champ d'état.

21.1.5 |AUTOALL

Cette commande déplace tous les sprites dont le drapeau de mouvement automatique est actif, en fonction de leurs attributs de vitesse Vy, Vx.

Utiliser :

|Les informations suivantes sont disponibles sur le site web de l'entreprise : <AUTOALL>, <routing flag>, <routing flag>.

Exemple

|AUTOALL,1 invoque **|ROUTEALL** avant de déplacer les sprites

|L'UTILISATION DE L'OPTION AUTOALL,0 n'invoque pas **|ROUTEALL**

|AUTOALL la dernière valeur utilisée est utilisée comme paramètre (a de la mémoire)

Le drapeau de routage est optionnel. Puisque la commande **|ROUTEALL** ne modifie pas les coordonnées des sprites, ils doivent être déplacés avec **|AUTOALL** et imprimés (et animés) avec **|PRINTSPALL**. C'est pourquoi vous avez un paramètre optionnel dans **|AUTOALL, de** sorte que **|AUTOALL,1** invoque en interne **|ROUTEALL** avant de déplacer le sprite, vous épargnant ainsi une invocation BASIC qui prendra toujours une précieuse milliseconde.

21.1.6 |COLAY

Déetecte la collision d'un sprite avec la carte d'écran (la disposition). Il prend en compte la taille du sprite pour déterminer s'il entre en collision, et considère que les éléments de la disposition sont tous des pixels 8x8 en mode 0 (c'est-à-dire 4 octets x 8 lignes). Il peut être invoqué avec 3,2,1 ou sans paramètres. S'il est invoqué sans paramètres, les valeurs

du dernier appel avec paramètres seront utilisées et il est beaucoup plus rapide.

Utiliser :

| Les données de l'interface utilisateur peuvent être utilisées pour l'analyse des données de l'interface utilisateur et pour l'évaluation de la qualité des données.

| COLAY, @collision%,<numéro d'inscription>

| COLAY, <num_sprite>, <num_sprite>.

| COLAY

Le paramètre facultatif <Seuil ASCII> ne doit être utilisé que lors d'une première invocation pour définir le seuil de collision dans la commande |COLAY|. Ce seuil représente le plus grand code ASCII de l'élément de mise en page considéré comme "sans collision". La valeur par défaut est 32 (l'espace blanc). Pour définir le seuil que vous souhaitez, reportez-vous à la table ASCII de la commande |LAYOUT|.

Exemple :

| COLAY, 65, @col%,31 : rem sprite est 31, seuil est 65

La variable que vous utilisez pour la collision peut être appelée comme vous le souhaitez. J'ai mis "col".

Cette routine modifie la variable collision (qui doit être un entier, d'où le "%") en la mettant à 1 s'il y a une collision du sprite indiqué avec la disposition. S'il n'y a pas de collision, le résultat est 0.

```
10 xprevious=x
20 x=x+1
30 |LOCATESP,0,y,x : ' positionne le sprite dans la nouvelle position
40 |COLAY,@collision%,0 : 'vérification de la collision
```

Nous vérifions maintenant la collision et, si c'est le cas, nous laissons l'objet à son emplacement précédent.

50 si collision%=1 alors x=précédent : LOCATESP,0,y,x

Vous pouvez également utiliser la commande |MOVER pour positionner le sprite et effectuer la vérification.

```
10 |COLAY,65,@col,31 : 'configuration. Nous ne le faisons qu'une fois
20 |MOVER,31,1,1 : ' nous le déplaçons vers la droite et vers le bas
30 |COLAY:' nous l'invoquons sans paramètres (plus rapide)
30 if col THEN MOVER,31,-1,-1 : le rem est entré en collision et je le
laisse donc là où il était.
```

21.1.7 |COLSP

Cette commande permet de détecter la collision d'un sprite avec les autres sprites dont le drapeau de collision est actif.

Utilisation :

Pour configurer :

| **COLSP, 32, <sprite initial>, <sprite final>.**
| **COLSP, 33 ans, @collision%**
| **COLSP, 34, dy, dx**

Pour la détection des collisions :

|COLSP,<numéro d'immatriculation>, @colsp%.

Exemple :

col%=0

|COLSP,0,@col%

La fonction renvoie dans la variable que nous passons en paramètre, le numéro du sprite avec lequel elle entre en collision, ou s'il n'y a pas de collision elle renvoie un 32 car le sprite 32 n'existe pas (il n'y en a que de 0 à 31).

IMPORTANT : La variable de collision dans la commande COLSP n'est pas celle utilisée dans la commande COLSPALL. Il s'agit de variables différentes (à moins que vous ne transmettiez aux deux commandes la même variable pour agir sur elle).

Comme pour l'impression de sprites avec PRINTSPALL, la fonction COLSP vérifie les sprites commençant par 31 et se terminant par zéro. S'ils ont un drapeau de collision de sprite actif (bit 2 de l'octet de statut), la collision est vérifiée. Si deux sprites entrent en collision en même temps avec notre sprite, le plus grand numéro de sprite est renvoyé car c'est celui qui est vérifié en premier.

Invocations pour configurer la commande :

Il est possible de configurer COLSP pour qu'il fasse moins de travail en vérifiant les collisions avec moins de sprites afin d'économiser du temps d'exécution. La configuration sera indiquée par l'utilisation du sprite 32 (qui n'existe pas).

|COLSP, 32, <sprite initial à vérifier>, <sprite final à vérifier>.

Si par exemple les ennemis de notre personnage sont les sprites 25 à 30, et que nous les configurons comme des colliders (pas des collisionneurs), nous pouvons invoquer (une seule fois) la commande comme suit :

|COLSP, 32, 25, 30

Cela signifie que toute invocation ultérieure de la commande |COLSP ne doit vérifier que la collision des sprites 25 à 30 (pour autant que le drapeau "collided" soit actif).

Par exemple, si nous ne devons vérifier que 6 ennemis, en préconfigurant la commande pour qu'elle ne vérifie qu'à partir de 25, nous pouvons économiser jusqu'à 2,5 ms à chaque exécution. Cela devient particulièrement important dans les jeux où le personnage peut tirer, puisque dans chaque cycle de jeu, au moins la collision entre le personnage et les tirs devra être vérifiée.

Une autre optimisation intéressante, capable d'économiser 1,1 milliseconde à chaque invocation, consiste à indiquer à la commande de toujours utiliser la même variable BASIC pour laisser le résultat de la collision. Pour ce faire, nous l'indiquerons en utilisant 33 comme sprite, qui n'existe pas non plus.

col%=0

|COLSP, 33, @col%, @col%, @COLSP, 33, @col%, @COLSP, 33, @col%, @COLSP, 33, @col%

Une fois ces deux lignes exécutées, les invocations suivantes de COLSP laisseront le résultat dans la variable col, sans qu'il soit nécessaire de l'indiquer, par exemple :

|COLSP, 23

Enfin, il est possible d'ajuster la sensibilité de la commande COLSP, en décidant si le chevauchement entre les sprites doit être de plusieurs pixels ou d'un seul, pour considérer qu'une collision s'est produite.

Pour ce faire, il faut définir le nombre requis de pixels qui se chevauchent dans les directions Y et X, en utilisant la commande COLSP et en spécifiant le sprite 34 (qui n'existe pas).

|COLSP, 34, dy, dx

Les valeurs par défaut de dy et dx sont respectivement 2 et 1. Notez que dans la direction y, ils sont considérés comme des pixels, mais dans la direction x, ils sont considérés comme des octets (un octet correspond à deux pixels en mode 0).

Pour une détection avec un chevauchement minimal (un pixel verticalement et/ou un octet horizontalement), vous devez procéder comme suit :

|COLSP, 34, 0, 0

21.1.8 |COLSPALL

Utiliser :

Pour configurer :

**|COLSPALL,@collider%, @collider%, @collider%, @collider%,
@collider%, @collider%, @collider%.**

Pour vérifier s'il y a des collisions

**|COLSPALL
|COLSPALL, <Collisionneur initial>.**

Cette fonction vérifie qui est entré en collision (parmi le groupe de sprites dont l'indicateur de collision de l'octet de statut est fixé à "1") et avec qui il est entré en collision (parmi le groupe de sprites dont l'indicateur de collision de l'octet de statut est fixé à "1").

Il s'agit d'une fonctionnalité fortement recommandée lorsque vous devez gérer les collisions de votre personnage et les tirs multiples, car elle permet d'économiser les invocations à |COLSP et donc d'accélérer votre jeu.

Important : les colliders (bit d'état 5) sont vérifiés de 31 à 0. Pour chaque collider, les colliderables (bit d'état 1) sont également vérifiés de 31 à 0.

Dans le cas où COLSPALL est invoqué avec un seul paramètre,

|COLSPALL, <Collisionneur initial>.

Les collisionneurs seront analysés à partir du collisionneur -1 indiqué jusqu'au sprite zéro, dans l'ordre décroissant. De cette façon, si vous avez besoin de détecter plus d'une collision par cycle de

vous pouvez le faire en invoquant successivement **COLSPALL**, <collider> jusqu'à ce que la variable collider prenne la valeur 32

Exemple :

|COLSPALL, 7 : recherche de collisions à partir du collisionneur 6

21.1.9 |LAYOUT

Utiliser :

|LAYOUT, <y>, <x>, <@string\$>, <@string\$>, <@string\$>, <@string\$>, <@string\$>, <@string\$>, <@string\$>.

Exemple :

```
string$ = "XYZZZZ      ZZ"  
|LAYOUT, 0,1, @chaîne$
```

Notez que l'utilisation de |LAYOUT, 0,1, "XYZZZZZZ ZZ" serait incorrecte sur un CPC464 alors qu'elle fonctionne sur un CPC6128. De même, sur un CPC6128, vous pouvez ignorer l'utilisation du "@", mais sur un CPC464, il est obligatoire.

Cette routine imprime une rangée de sprites pour construire la disposition ou le "labyrinthe" pour chaque écran. En plus de dessiner le labyrinthe, ou tout autre graphique à l'écran construit avec de petits sprites 8x8, vous pouvez également détecter les collisions d'un sprite avec la disposition, à l'aide de la commande **|COLAY**.

Les sprites à imprimer sont définis par une chaîne de caractères, dont les caractères (32 possibles) représentent l'un des sprites suivant cette règle simple, où la seule exception est l'espace vide représentant l'absence de sprite.

Caractère	Identifiant du sprite	Code ASCII
" "	AUCUN	
" ;"	0	59
"<"	1	
"=="		
">"		
" ?"		63
"@"	5	
"A"		65
"B"		
"C"	8	67
"D"		
"E"	10	69
"F"		70
"G"		71
"H"		
"T"		
"J"		
"K"		75

"L"		
"M"		
"N"		78
"O"		79
"P"	21	80
"Q"		81
"R"	23	82
"S"		
"T"	25	84
"U"	26	85
"V"		86
"W"		87
"X"	29	88
"Y"	30	
"Z"	31	90

Tableau 6 Correspondance des caractères et des sprites pour la commande |LAYOUT

IMPORTANT : Après avoir imprimé la mise en page, vous pouvez changer les sprites en caractères, de sorte que vous disposerez toujours des 32 sprites.

Les coordonnées y,x sont transmises sous forme de caractères. La bibliothèque gère en interne une carte de 20x25 caractères, de sorte que les coordonnées prennent les valeurs suivantes :

y prend les valeurs
[0,24] x prend les
valeurs [0,19].

Les sprites à imprimer doivent avoir une taille de 8x8 pixels. Ce sont des "briques", également appelées "tuiles", qui sont souvent utilisées de la même manière.

Si vous utilisez d'autres tailles de sprites, cette fonction ne fonctionnera pas bien. Elle imprimera effectivement les sprites, mais si un sprite est grand, vous devrez placer des blancs pour lui faire de la place.

La bibliothèque maintient une carte de disposition interne et cette fonction met à jour les données de la carte de disposition interne afin qu'il soit possible de détecter les collisions. Cette carte est un tableau de 20x25 caractères, où chaque caractère correspond à un sprite.

Vous ne pouvez pas passer la chaîne directement, bien que dans le CPC6128 le passage des paramètres le permette, mais cela serait incompatible avec le CPC464.

Précautions :

La fonction ne valide pas la chaîne de caractères que vous lui transmettez. Si elle contient des lettres minuscules ou tout autre caractère différent de ceux autorisés, elle peut provoquer des effets indésirables, tels qu'un redémarrage ou un plantage de l'ordinateur. Il ne peut pas s'agir d'une chaîne vide non plus !

Les limites fixées par SETLIMITS devraient vous permettre d'imprimer là où vous le souhaitez. Si vous souhaitez par la suite découper une zone plus petite, vous pouvez invoquer à nouveau SETLIMITS lorsque toute la mise en page est imprimée.

Exemple :

<pre> 2070 SETLIMITS,0,80,0,200 2090 c\$(1)= " PPPPPPPPPPPPPPPPPPPPPPPPPPPP". P" 2100 c\$(2)= "PU P" 2110 c\$(3)= "P P" 2120 c\$(4)= "P P" 2130 c\$(5)= "P TPPPPPPU TPPPPPPPPPPPPP" TP" 2140 c\$(6)= "P P" 2150 c\$(7)= "P P" 2160 c\$(8)= "P P" 2170 c\$(9)= "P YYYYYYYYYYYYYYY YYYYYYYYY P" 2190 c\$(10)= "P TPPPPPPPPPPPPU P" 2195 c\$(11)= "P P" 2200 c\$(12)= "P P" 2210 c\$(13)= "P P" 2220 c\$(14)= "YYYYYYYYYYYYYP PYYYYYYYYYYY " 2230 c\$(15)= "RRRRRRRRRRRRRRR RRRRRRRRR" 2240 c\$(16)= "PPPPPPPPPPPPPPPP PPPPPPPPPP P" 2250 c\$(17)= "PU TP PU TP" 2260 c\$(18)= "P T U P" 2270 c\$(19)= "P P" 2271 c\$(20)= "P P" 2272 c\$(21)= "P W P" 2273 c\$(22)= "PP W PP" 2274 c\$(23)= "PPPPPPPPPPPPPPPPPP". 2280 for i=0 to 23 2281 LAYOUT,i,0,@c\$(i) 2282 suivant </pre>	<p>Cet exemple utilise plusieurs briques qui ont été préalablement créées avec l'outil "SPEDIT".</p> <p>; sprite 20 --> O bush</p> <p>; sprite 21 --> P rock</p> <p>; sprite 22 --> nuage Q</p> <p>; sprite 23 --> R eau</p> <p>; sprite 24 --> fenêtre S</p> <p>; sprite 25 --> T arche du pont de droite</p> <p>; sprite 26 --> arche U du pont gauche</p> <p>; sprite 27 --> drapeau V</p> <p>; sprite 28 --> W plant</p> <p>; sprite 29 --> X tower spike</p> <p>; sprite 30 --> Et herbe</p> <p>; sprite 31 --> Z brick</p>
--	--

21.1.10 |LOCATESP

Cette commande modifie les coordonnées d'un sprite dans la table d'attributs des sprites.

Utilisation

|LOCATESP, <numéro d'immatriculation>, <y>, <x>

Exemple

|LOCATESP,0,10,20

Une alternative à cette commande, si l'on ne souhaite modifier qu'une seule coordonnée, est d'utiliser la commande BASIC POKE, en insérant la valeur souhaitée à l'adresse mémoire occupée par la coordonnée X ou Y. Si nous voulons introduire une coordonnée négative, la commande |POKE est nécessaire, car elle serait illégale avec la commande POKE BASIC.

La commande **LOCALISER** n'imprime pas le sprite, elle le positionne seulement pour qu'il

soit imprimé.

21.1.11 |MAP2SP

Cette fonction parcourt la carte du monde décrite dans le fichier map_table.asm et transforme en sprites les éléments de la carte qui peuvent entrer partiellement ou totalement dans l'écran.

Utilisation

|MAP2SP, <y>, <x>

|MAP2SP, <status>, <status>, <status>, <status>, <status>, <status>, <status>, <status>.

Exemple

|MAP2SP, 1500, 2500

Les sprites créés par MAP2SP sont créés par défaut avec l'état 3, c'est-à-dire avec le drapeau d'impression actif (**|PRINTSPALL** l'imprime) et avec le drapeau de collision actif (**|COLSP** va entrer en collision avec lui). Si vous souhaitez que les sprites soient créés dans un autre état, il vous suffit d'invoquer une fois la commande **|MAP2SP** avec un seul paramètre indiquant l'état dans lequel les sprites doivent être créés.

Si, par hasard, **|MAP2SP** rencontre plus de 32 éléments à traduire en sprites, il ignorerá ceux qui dépassent 32.

|MAP2SP, <status>, <status>, <status>, <status>, <status>, <status>, <status>, <status>.

| Ceci configure la commande MAP2SP pour qu'elle soit imprimée mais non collisionnelle.

IMPORTANT : Sur la carte du monde, vous pouvez combiner des images normales avec des "images d'arrière-plan" (section 8.5). Les images d'arrière-plan sont toujours transparentes. Le drapeau de transparence que vous utilisez dans **|MAP2SP, <status>** ne s'appliquera qu'aux images normales.

Les paramètres **<y>, <x>** de la fonction sont l'origine du déplacement à partir duquel le monde est affiché à l'écran. Trois autres paramètres se trouvent dans **MAP_TABLE**, la table à partir de laquelle le monde est défini. Ces paramètres sont la hauteur maximale, la largeur maximale (en négatif) et le nombre d'éléments dans le monde (maximum 82).

Chaque élément est un tuple de 3 paramètres précédé du mnémonique "DW" :
DW Y, X, <image>

TABLEAU DE LA CARTE

;

3 paramètres avant la liste des "éléments de la carte".

dw 50 ; hauteur maximale d'un sprite au cas où il passerait par le haut et qu'une partie devrait être peinte.

dw -40 ; largeur maximale d'un sprite dans le cas d'un sprite gaucher (nombre négatif)

db 64 ; nombre d'éléments de la carte à prendre en compte. il devrait être au maximum de 82

et à partir de là, les éléments dw

100,10,HOUSE ; 1

dw 50,-10,CACTUS;2

dw 210,0,HOUSE;3

dw 200,20,CACTUS;4

dw 100,40,HOUSE;5

dw 160,60,HOUSE;6

dw 70,70,HOUSE;7

dw 175,40,CACTUS;8

dw 10,50,HOUSE;9

dw

250,50,HOUSE;10

dw

260,70,HOUSE;11

dw

260,70,HOUSE;11

dw 290,60,CACTUS;12

dw 180,90,HOUSE;13

dw 60,100,HOUSE;14

dw 60,100,HOUSE;14

...

21.1.12 |MOVER

Cette commande permet de déplacer un sprite de manière relative, c'est-à-dire en ajoutant des quantités relatives à ses coordonnées.

Utiliser :

|MOVER,<numéro d'ordre>, <dy>, <dx>, <dx>, <dx>, <dx>, <dx>, <dx>, <dx>, <dx>.

Exemple :

|MOVER,0,1,-1

L'exemple déplace le sprite 0 vers le bas et vers la droite en même temps. Le sprite n'a pas besoin d'avoir le drapeau de mouvement relatif activé.

Il existe un moyen d'utiliser **|MOVER** sans spécifier ni "dy" ni "dx". Pour ce faire, nous spécifierons le sprite 32, qui n'existe pas, et nous mettrons en paramètre les adresses mémoire des variables que nous voulons utiliser pour stocker à la fois "dy" et "dx".

L'adresse mémoire d'une variable est obtenue en la faisant précédé du symbole "@".

Exemple :

dy%= 5

dx%= 2

|MOVER,32, @dy, @dx

A partir de ce moment, nous pourrons utiliser :

|MOVER, <id>

Ainsi, le sprite "id" se déplacera comme indiqué par les variables dy, dx. Ce mécanisme fonctionne également avec **|MOVERALL**

21.1.13 |GLOBAL

Cette commande déplace relativement tous les sprites dont le drapeau de mouvement relatif est activé.

Utilisation

|MOVERALL, <dy>, <dx>, <dx>, <dx>, <dx>, <dx>, <dx>, <dx>, <dx>.

Exemple

|MOVERALL,2,1

L'exemple déplace tous les sprites ayant un drapeau de mouvement relatif vers le bas (2 lignes) et 1 octet vers la droite.

Si aucun paramètre n'est spécifié, les variables spécifiées dans l'invocation du MOVER avec le sprite 32 seront utilisées, c'est-à-dire

|MOVER,32, @dy, @dx

|GLOBAL

Equivalent à **|MOVERALL, dy, dx**

Cette utilisation "avancée" de la commande évite le passage de paramètres à chaque invocation et est donc plus rapide, ce qui est essentiel dans nos programmes BASIC.

21.1.14 |MUSIQUE

Cette commande permet à une mélodie de commencer à jouer Usage :

|MUSIC,<flag_channel_C>,<flag_repeat>,<melody_number>,<speed>. |La musique peut être répétée à l'infini : <MUSIC>, <flag_repetition>, <melody_number>, <speed>.

|MUSIC : ' sans paramètres la musique se termine

La valeur 1 de l'indicateur C-channel permet de libérer le troisième canal sonore afin qu'il puisse être utilisé pour produire des effets sonores (déclenchements, etc.) à l'aide de la commande

SON 4,<note>,<durée>, ...

Notez que vous devez utiliser le canal 4, car en BASIC les canaux sont tapés comme A=1, B=2, C=4.

Si l'indicateur de canal est omis ou mis à zéro, la musique utilisera les 3 canaux et vous ne pourrez pas utiliser la commande SOUND en même temps (vous pouvez le faire mais avec des effets bizarres).

L'indicateur de répétition doit être égal à zéro pour que la musique soit jouée en boucle. Si vous souhaitez que la musique ne soit jouée qu'une seule fois, vous devez utiliser la valeur 1.

Le numéro de la mélodie doit être compris entre 0 et 7.

La vitesse "normale" est de 6. Si nous utilisons un chiffre plus élevé, la lecture sera plus lente et si le chiffre est plus bas, elle sera plus rapide.

La commande |MUSIC invoquée sans paramètres désactive l'interruption de la musique et arrête la lecture de toute mélodie.

Exemples :

|MUSIQUE,0,0,0,6
|MUSIQUE,1,0,0,6
|MUSIC,0,0,6
|MUSIQUE

En interne, cette commande installe une interruption qui est déclenchée 300 fois par seconde. Si nous fixons la vitesse à 6, la fonction de lecture de musique est exécutée une fois sur six.

Comme elle est basée sur les interruptions, il faut qu'un programme soit en cours d'exécution pour que la musique soit jouée, car lorsque l'interpréteur BASIC attend des commandes, ces interruptions ne sont pas activées. Si vous exécutez simplement la commande |MUSIC, vous n'entendrez rien, mais si vous l'exécutez à l'intérieur d'un programme comme celui présenté ci-dessous, la musique sera jouée.

10 MUSIQUE,0,0,0,5 20 goto 20 : ' boucle infinie. Lors de l'exécution, la musique est jouée

21.1.15 |PEEK

Cette commande permet de lire une valeur de données de 16 bits à partir d'une adresse mémoire donnée. Elle est destinée à interroger les coordonnées des sprites qui se déplacent de manière automatique ou relative.

Utilisation

|PEEK,<adresse>, @data%.

Exemple

data%=0

|PEEK, 27001, @dato%, @dato%, @dato%, @dato%, @dato%, @pEEK

Si les coordonnées ne sont que positives et inférieures à 255, vous pouvez utiliser la commande BASIC PEEK, qui est un peu plus rapide.

21.1.16 |POKE

Cette commande insère une donnée de 16 bits (positive ou négative) dans une adresse mémoire. Elle est destinée à modifier les coordonnées d'un sprite, car la commande POKE ne peut pas gérer les coordonnées négatives ou supérieures à 255, puisque POKE travaille avec des octets alors que **|POKE** est une commande de 16 bits.

Utiliser :

|POKE, <adresse>, <valeur>.

Exemple :

|POKE, 27003, 23

Cet exemple place la valeur 23 à la coordonnée x du sprite 0.

Il s'agit d'une fonction très rapide, bien que si vous ne manipulez que des coordonnées positives, il est préférable d'utiliser POKE, qui est encore plus rapide.

21.1.17 |PRINTAT

|PRINTAT permet d'imprimer une chaîne de caractères en utilisant un nouveau jeu de caractères plus petit (que j'appelle "mini-caractères"). Cette nouvelle commande vous permet d'utiliser le mécanisme de transparence des sprites, de sorte que vous pouvez imprimer des caractères tout en respectant l'arrière-plan. Elle fonctionne de la manière suivante :

Utiliser :

|PRINTAT,<flag transparence>, y,x,@string

Exemple :

cad\$= "Hello".

|PRINTAT,0,100,10, @cad\$

La commande **|PRINTAT** imprime des chaînes de caractères et non des variables numériques. Par conséquent, si vous souhaitez imprimer un nombre (par exemple, les points sur le tableau d'affichage de votre jeu vidéo), vous devez le faire :

```
points=points+1  
cad$= str$(points)  
|PRINTAT,0,100,10, @cad$
```

La commande |PRINTAT n'est pas affectée par les limites d'écrêtage définies avec la commande

|SETLIMITS. Ceci est très logique puisque vous utiliserez normalement PRINTAT pour imprimer les scores sur vos marqueurs, qui seront en dehors de la zone délimitée par |SETLIMITS.

Contrairement à la commande BASIC PRINT, la commande |PRINTAT est assez rapide et peut être utilisée pour mettre à jour fréquemment vos marqueurs de jeu.

PRINTAT utilise un alphabet redéfini, qui peut contenir une version réduite ou différente des caractères "officiels" d'Amstrad. Par défaut, le 8BP fournit un petit alphabet composé de chiffres, de lettres majuscules et de quelques symboles. Il se présente comme suit :

"0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ: ! ,."

Vous ne pourrez pas utiliser un caractère qui ne fait pas partie de cet ensemble, comme les lettres minuscules. Si vous essayez de le faire, le dernier caractère défini dans la chaîne (dans ce cas, l'espace) sera imprimé.

Les caractères de cet alphabet ont tous la même taille : 4 pixels de large x 5 pixels de haut, soit 2 octets x 5 lignes.

L'alphabet par défaut ne contient pas de lettres minuscules et de nombreux symboles sont absents, bien que vous puissiez créer votre propre alphabet qui les contienne.

21.1.18 |PRINTSP

Utiliser :

|PRINTSP, <identifiant du preneur d'empreinte>, <y>,<x>, <x>.

|PRINTSP, <identifiant du preneur d'otages>.

|Les données de l'écran d'affichage peuvent être modifiées en fonction de l'évolution de l'environnement.

Exemple :

Imprimer le sprite 23 aux coordonnées y=100, x=40 (mise à jour de ses coordonnées).

|PRINTSP, 23,100,40

Exemple :

Imprime le sprite 23 aux coordonnées déjà assignées dans la table des sprites :

|PRINTSP, 23

Si le sprite 32 (qui n'existe pas) est spécifié, le paramètre suivant est utilisé pour spécifier le nombre de bits d'arrière-plan dans l'impression transparente. Si 1 bit est utilisé, 2 couleurs d'arrière-plan peuvent être utilisées. Si 2 bits sont spécifiés, 4 couleurs d'arrière-plan peuvent être utilisées.

Si un sprite_id <32 est spécifié, la commande imprime un sprite à l'écran, et si des coordonnées sont spécifiées, elle les met également à jour.

Les coordonnées prises en compte sont les suivantes :

- Nombre de lignes verticales [-32768..32768]. Les lignes correspondantes à l'intérieur de l'écran sont [0..199].
- Nombre d'octets en horizontal [-32768..32768]. Ceux correspondant à l'intérieur

de l'écran sont [0..79].

Normalement, dans la logique d'un jeu vidéo, vous utiliserez **|PRINTSPALL**, car il est plus rapide de les imprimer tous en même temps. Cependant, à d'autres moments du jeu, vous pouvez vouloir imprimer les sprites séparément. Cet exemple montre la descente d'un "rideau", en utilisant un seul sprite qui se répète horizontalement et qui, en descendant, "teinte" l'écran en rouge, donnant la sensation d'un rideau qui descend.

```

7089 telon=&8ec0
7090 |setupsp,1,9,curtain
7100 pour y=8 à 168 étape 4
7110 pour x=12 à 64 étape 4
7111 |PRINTSP,1,y,x
7112 suivant
7113 suivant

```



Fig. 119 Exemple d'utilisation de PRINTSP

21.1.19 |PRINTSPALL

Cette routine imprime en une seule fois tous les sprites dont le bit d'état0 est activé.

Utiliser :

| Les données relatives à l'état d'avancement de la mise en œuvre de la politique de l'UE sont les suivantes : <PRINTSPALL>, <ordenini>, <ordenfin>, <flag anima>, <flag sync>.

| Les données relatives à l'ordre et au type d'ordre ne sont pas disponibles pour le moment,

Exemple :

Avec les valeurs suivantes, la commande imprime tous les sprites en les animant en premier et de manière non synchronisée avec le balayage, et sans tri :

|PRINTSPALL, 0, 0, 0, 1, 0

| Si la commande est omise, elle prend la dernière valeur attribuée, ou zéro si elle n'a jamais été attribuée.

Si la valeur 1 est attribuée, avant l'impression des sprites, le cadre est modifié dans sa séquence d'animation, à condition que le bit d'état 3 soit activé pour les sprites. IMPORTANT : l'animation est effectuée avant l'impression, et non après. Cela signifie que si vous venez d'attribuer une séquence d'animation, vous ne verrez pas la première image de cette séquence.

Le <flag sync> est un drapeau de synchronisation avec le balayage de l'écran. Il peut être égal à 1 ou 0. La synchronisation n'a de sens que si vous compilez le programme avec un compilateur comme "Fabacom". La logique BASIC fonctionne lentement et la synchronisation avec le balayage produit de petites attentes supplémentaires dans chaque cycle du jeu, ce qui n'est pas pratique.

En règle générale, cela ne convient que si votre jeu est capable de générer 50 images par

seconde, ou en d'autres termes, un cycle de jeu complet toutes les 20 millisecondes. Si vous compilez votre jeu avec un compilateur tel que "fabacom", il est recommandé de le synchroniser avec le balayage de l'écran, car vous atteindrez presque certainement les 50 images par seconde et vous pourrez obtenir un cycle de jeu complet toutes les 20 millisecondes.

Si vous les dépassiez, votre jeu produira plus d'images que l'écran ne peut en afficher, certaines ne seront pas affichées et les mouvements ne seront pas fluides.

Plus il y a de sprites à imprimer à l'écran, plus la commande prendra de temps, bien qu'elle soit très rapide. De nombreux sprites peuvent apparaître à l'écran, mais n'ont pas besoin d'être imprimés (ils peuvent avoir le bit d'état 0 désactivé), comme les fruits, les pièces de monnaie, les éléments de bonus en général et/ou les personnages qui ne bougent pas et n'ont pas d'animation. Même s'ils ne sont pas imprimés, ils peuvent avoir le bit de collision activé et donc affecter la routine.

|COLSP et |COLSPALL

Les paramètres d'ordre ("ordenini", "ordenfin") indiquent les sprites initiaux et finaux qui définissent le groupe de sprites ordonnés par la coordonnée "Y" que nous allons imprimer. Par exemple, si nous attribuons les valeurs 0,0, ils seront imprimés séquentiellement du sprite 0 au sprite 31. Si nous attribuons 0,8, ils seront imprimés de 0 à 8 dans l'ordre (9 sprites) et de 10 à 31 séquentiellement. Si nous attribuons 0,31, tous les sprites seront imprimés dans l'ordre. Si nous attribuons 10,20, les sprites seront imprimés séquentiellement de 0 à 9, puis ils seront imprimés dans l'ordre de 10 à 20 et enfin ils seront imprimés séquentiellement de 21 à 31.

La commande est très utile pour réaliser des jeux de type "Renegade" ou "Golden AXE", où il est nécessaire de donner un effet de profondeur.

Si le paramètre "ordenini" est omis, c'est la dernière valeur attribuée qui est prise en compte, ou zéro si aucune valeur n'a jamais été attribuée. Par ailleurs, si vous avez l'intention de modifier l'un des deux paramètres de tri, il est conseillé d'exécuter d'abord PRINTSPALL,0,0,0,0,0 pour que les sprites soient d'abord réordonnés séquentiellement avant d'être triés avec une nouvelle configuration.

L'impression dans l'ordre est plus coûteuse en termes de calcul que l'impression séquentielle. Si vous n'avez que 5 sprites à trier, passez un 4 comme paramètre de tri, ne passez pas un 31. Le tri de tous les sprites prend environ 2,5 ms, mais si vous ne triez que 5 sprites, vous pouvez gagner 2 ms. Il se peut que vous ayez beaucoup de sprites et qu'il ne soit pas utile de trier certains d'entre eux, comme les plans ou les sprites dont vous savez qu'ils ne se chevauchent pas.

Le tri de |PRINTSPALL est partiel, c'est-à-dire qu'une seule paire de sprites non ordonnés est triée à chaque invocation. Vous pouvez parfois vouloir que le tri soit complet pour chaque image. En d'autres termes, il ne s'agit pas de trier une paire de sprites à chaque invocation de |PRINTSPALL, mais de s'assurer qu'ils sont tous triés. La bibliothèque 8BP vous permet de le faire grâce à ses quatre modes de tri, que vous pouvez définir en invoquant la commande |PRINTSPALL avec un seul paramètre (il suffit de l'exécuter une fois pour définir le mode de tri) :

PRINTSPALL,0 : tri partiel en utilisant Ymin	PRINTSPALL,1 : tri complet en utilisant Ymin	PRINTSPALL,2 : tri partiel en utilisant Ymax	PRINTSPALL,3 : tri complet en utilisant Ymax
---	---	---	---

Le tri utilisant Ymax est basé sur la plus grande coordonnée Y des sprites, c'est-à-dire l'endroit où se trouvent leurs pieds plutôt que leur tête. Si les sprites sont de la même taille, un tri basé sur Ymin peut fonctionner, mais si les sprites sont de hauteurs différentes, vous pouvez vouloir trier en fonction de l'emplacement des pieds de chaque personnage, et pour cela vous devrez utiliser le mode de tri 2 ou 3.

Les modes de tri Ymax sont plus lents, environ 0,128 ms par sprite, et ne sont donc utilisés que lorsque vous en avez vraiment besoin.

Le tri complet consomme très peu plus que le tri partiel (environ 0,3 ms). Cela s'explique par le fait que les sprites ne sont pratiquement jamais mélangés d'une image à l'autre, mais même ces 0,3 ms valent la peine d'être économisées si possible.

Il existe un comportement très intéressant de cette fonction pour gagner 1ms dans son exécution. Il consiste à l'invoquer avec des paramètres une fois et à l'invoquer sans paramètres les fois suivantes. Dans ce cas, il sera supposé que, même si aucun paramètre n'est passé, leurs valeurs sont égales aux dernières passées. De cette manière, l'analyseur syntaxique travaille moins et réduit le temps d'exécution.

21.1.20 |RINK

Cette commande permet d'effectuer une animation par encres. Utilisation :

|RINK, <initial_ink>, <colour1>, <colour2>, . . . , <colourN>, <colourN>, <colourN>, <colourN>, <colourN>, <colourN>, <colourN>, <step>

RINK fait tourner un ensemble d'encres à partir de l'encre initiale N encres (n'importe quel nombre d'encres), en fonction de la taille du motif de couleur défini.

La vitesse de rotation peut être contrôlée par le paramètre step, qui indique le nombre de sauts de couleur effectués par chaque encre lors d'une invocation.

Recommandation : en raison de l'utilisation d'interruptions, **|RINK** provoque des "bégaiements" dans certains cas lorsqu'il est utilisé en même temps que la commande **|MUSIC** à la vitesse 6. Si vous souhaitez utiliser les deux en même temps sans interférence, utilisez une autre vitesse pour la musique (vous pouvez utiliser la vitesse 5 ou 7, les deux fonctionneront parfaitement).

Exemples :

Cette commande définit un motif de 4 rouges (couleur =3) et 4 jaunes (couleur =24) à faire tourner de l'encre 8 à l'encre 15.

|LA COMMISSION EUROPÉENNE A DÉCIDÉ DE METTRE EN PLACE UN SYSTÈME DE GESTION DE L'INFORMATION ET DE LA COMMUNICATION (GIS).

Cette commande définit un motif de 2 blancs (couleur =26) et de 2 gris (couleur=13) à faire pivoter de l'encre 3 à l'encre 6.

|RINK, 3, 26, 26, 13, 13, 13, 13

Rotation d'une couleur du motif toutes les encres

|RINK, 1

Dans un motif à 8 couleurs, la commande suivante laisse tout en l'état

|RINK, 8

Cette commande ne ferait rien d'autre que de forcer les encres à adopter la couleur du motif.

| RINK, 0

21.1.21 |ROUTEALL

Cette commande permet d'acheminer tous les sprites dont l'octet d'état contient l'indicateur d'itinéraire par l'itinéraire qui leur a été assigné (paramètre 15 de |SETUPSP).

Utiliser :

|ROUTEALL

Elle n'a pas de paramètres et est donc très facile à invoquer. Ce que cette commande fait en interne, c'est de garder un compte de pas pour le segment que chaque sprite parcourt, de sorte que si le segment s'épuise, elle modifie la vitesse du sprite.

La commande ne modifie pas les coordonnées des sprites, ils doivent donc être déplacés avec |AUTOALL et imprimés (et animés) avec |PRINTSPALL. C'est pourquoi vous avez un paramètre optionnel dans |AUTOALL, de sorte que |AUTOALL,1 invoque en interne |AUTOALL,1.

|ROUTEALL avant de déplacer le sprite, vous évitant ainsi une invocation BASIC qui prendra toujours une précieuse milliseconde.

Les routes sont définies dans le fichier routes routes_yourgame.asm

DÉFINITION DE CHAQUE ITINÉRAIRE

=====

ROUTE0 ; un cercle

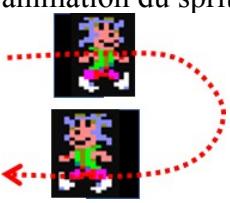
```
db 5,2,0
db 5,2,-1
db 5,0,-1
db 5,-2,-1
db 5,-2,0
db 5,-2,1
db 5,0,1
db 5,2,1
db 0
```

Le dernier segment est égal à zéro, ce qui indique que le chemin est terminé et que le sprite doit repartir du début. Assurez-vous que le nombre de pas dans chaque segment ne dépasse pas 250 et que Vy et Vx sont tous deux compris entre -127 et 127.

Pour attribuer un chemin à un sprite, vous devez utiliser la commande SETUPSP en spécifiant le paramètre 15. L'exemple suivant associe le chemin 3 au sprite 31

|SETUPSP, 31, 15, 3

Il existe 4 fonctionnalités que vous pouvez utiliser au milieu (**et non à la fin**) d'un itinéraire :

Code d'échappement (champ "nombre de pas")	Description	Exemple
255	Changement d'état du sprite.	DB 255, 3, 0 L'État passe à la valeur 3. Le zéro à la fin est un bouchetrou.
254	Modification de la séquence d'animation du sprite  Après avoir modifié la séquence, si vous souhaitez que l'image change également, vous devez utiliser le code 251	DB 254, 10, 0 La séquence 10 est associée. Le zéro est un remplissage. Si la séquence assignée est celle que le sprite possède déjà, l'opération est inoffensive (l'identifiant de l'image n'est pas réinitialisé). Si vous souhaitez réinitialiser l'identifiant du cadre, le troisième paramètre doit être un 1, par exemple DB 254, 10, 1.
253	Changement d'image 	DB 253 DW new_img L'image "new_img" est associée, qui doit être une adresse mémoire.
252	Changement d'itinéraire	DB 252, 2, 0 La route 2 est associée
251	Aller à jusqu'à suivant cadre de l'animation. 	DB 251, 0, 0 Le Sprite est animé. Les deux zéros sont des éléments de remplissage

IMPORTANT : faites très attention à écrire DB et DW là où ils doivent être utilisés, c'est-à-dire, par exemple, si vous changez d'image, vous devez faire précéder l'image de DW et non de DB. Si vous faites une telle erreur, votre itinéraire ne fonctionnera pas.

IMPORTANT : Un itinéraire peut avoir une longueur maximale de 255 octets et un segment a une longueur de 3 octets, de sorte qu'un itinéraire peut avoir une longueur maximale de 84 segments. Vous pouvez avoir besoin de construire un itinéraire encore plus long et, dans ce cas, vous pouvez le faire en concaténant la fin d'un itinéraire avec un changement d'itinéraire vers un autre itinéraire (code 252), et vous pouvez concaténer autant d'itinéraires que vous le souhaitez.

IMPORTANT : les codes d'échappement peuvent être utilisés au milieu d'un itinéraire, mais le dernier segment ne peut pas être un code d'échappement, il doit s'agir d'un mouvement, même s'il est immobile, quelque chose comme "DB 1,0,0".

Dans ces cas, |ROUTEALL interprétera qu'un changement d'état, de séquence, d'image, de chemin de sprite ou d'animation doit être forcé et exécutera également l'étape suivante. Les changements peuvent être forcés à n'importe quel segment du chemin, pas nécessairement à la fin, et vous pouvez forcer autant de changements que vous le souhaitez.

ROUTE0 ; un coup à gauche

```
-----
;-----  

db 100,0,-1 ; il reste cent pas à la vitesse Vx=-1 db 255,0,0  

;désactivation du sprite avec status=0  

db 1,0,0 ; ne rien déplacer à ce stade db 0  

ROUTE1 ; un saut à droite db  

253  

dw SOLDIER_R1_UP  

db 1,-5,1  

db 2,-4,1  

db 2,-3,1  

db 2,-2,1  

db 2,-1,1  

db 253  

dw SOLDIER_R1_DOWN  

db 1,-5,1 ; vers le haut pour que UP et down  

correspondent à db 2,1,1  

db 2,2,1  

db 2,3,1  

db 2,4,1  

db 1,5,1  

db 253  

dw SOLDIER_R1  

db 1,5,1 ; descendez une fois de plus  

db 255,13,0 ; nouvel état, sans drapeau de chemin et avec  

drapeau d'animation db 254,32,0 ; séquence macro 32  

db 1,0,0 ; quietooo. !!!!  

db 0
```

21.1.22 |ROUTESP

Cette commande permet d'acheminer un Sprite unique dont l'octet d'état contient l'indicateur d'acheminement actif par l'itinéraire qui lui a été assigné (paramètre 15 de SETUPSP).

Utiliser :

|ROUTESP, <spriteid>, <steps>.

|Cette commande déplace un sprite d'un nombre quelconque de pas (**jusqu'à 255**) le long du chemin qui lui a été assigné. La commande déplace le sprite à travers tous les pas indiqués, le laissant finalement à la même position qu'il aurait eu si nous avions exécuté |AUTOALL,1 un nombre de fois égal au nombre de pas.

IMPORTANT : les étapes ne peuvent pas prendre une valeur supérieure à 255.

21.1.23 |SETLIMITS

Cette commande définit les limites de la zone où les sprites ou les étoiles peuvent être imprimés.

Utiliser :

Exemple qui définit l'ensemble de l'écran comme la zone autorisée

| SETLIMITS,0,80,0,200

En dehors de ces limites, les sprites sont coupés, de sorte que si un sprite est partiellement à l'extérieur de la zone autorisée, les fonctions |PRINTSP y |PRINTSPALL n'imprimera que la partie située à l'intérieur de la zone autorisée.

21.1.24 |SETUPSP

Cette commande charge les données d'un sprite dans la table d'utilisation SPRITES_TABLE :

|SETUPSP, <id_sprite>, <nombre_de_paramètres>, <valeur>.

Exemple :

|SETUPSP, 3, 7, 2

Permet par exemple d'assigner une nouvelle séquence d'animation lorsque le sprite change de direction, ou simplement de modifier son registre de drapeaux d'état.

Cette fonction permet de modifier n'importe quel paramètre d'un sprite, à l'exception de X, Y (ce qui se fait avec LOCATE_SPRITE).

On ne peut modifier qu'un seul paramètre à la fois. Le paramètre à modifier est spécifié avec `param_number`. Le numéro de paramètre est en fait la position relative du paramètre dans la table SPRITES TABLE.

Numéro de paramètre	Action	Utilisation possible de POKE ou POKE comme alternative
0	modifie l'état (occupe 1 octet)	OUI
5	modifie Vy (occupe 1 octet, valeur en lignes verticales). Vous pouvez également modifier	
	Vx en même temps si nous l'ajoutons à la fin en tant que paramètre	
	change Vx (occupe 1 octet, valeur en octets horizontaux)	OUI
	séquence de changement (occupe 1 octet, prend les valeurs 0...31)	NON, car SETUPSP réinitialise également l'identifiant de la trame et vous donne également attribue l'adresse de la

		première image.
8	change frame_id (occupe 1 octet, prend les valeurs 0...7)	OUI
	changer l'image du répertoire (occupe 2 octets). L'image spécifiée peut faire partie de la liste initiale d'images du fichier images_mygame.asm,	Il n'en va pas de même si une image <255 est utilisée. Si une adresse mémoire est utilisée, POKE peut être utilisé comme alternative.
	changer le chemin (prend 1 octet)	NON, parce que SETUPSP fait plus de choses dans les tables internes pour que l'itinéraire fonctionne.

Exemple :

Dans cet exemple, nous avons donné au sprite 31 l'image d'un bateau qui est assemblé à l'adresse &a2f8.

```
navire = &a2f8
|SETUPSP, 31, 9, nef
```

Il existe un moyen plus simple de spécifier l'image du sprite en utilisant la liste des images (IMAGE_LIST) dans le fichier images_yourgame.asm. Si nous avons le NAVE dans la liste des images, nous pouvons lui associer un identifiant compris entre 16 et 255.

|SETUPSP,31, 9, 16 : rem le 16 est l'identifiant du NAVE dans la liste des images (IMAGE_LIST)

LISTE D'IMAGES

;

Nous mettrons ici une liste des images que nous voulons utiliser sans spécifier l'adresse mémoire de base.

La commande |SETUPSP,<id>,9,<adresse> devient |SETUPSP,<id>,9,<numéro> ; ainsi la commande

|SETUPSP,<id>,9,<adresse> devient |SETUPSP,<id>,9,<numéro>.

L'avantage de ne pas utiliser les adresses mémoire en BASIC est que si nous agrandissons les graphiques ou les réassemblons en

Le numéro que nous attribuons ne changera pas.

Ils ne doivent PAS tous avoir un numéro, mais seulement ceux que nous allons utiliser avec |setupsp, id, 9,<num>.

La numérotation commence à 16

Nous pouvons utiliser jusqu'à 255 images spécifiées de cette manière.

La liste n'a pas besoin d'être composée de 255 éléments, elle est de longueur variable et peut même être vide.

;

DW NAVE ; 16

DW OTHER_SHIP ; 17

----- DÉBUT DE L'IMAGE -----

NEF

db 7 ; largeur

db 92, 0, 0, 0, 0, 0, 0

db 0, 0, 0, 0, 0, 0

db 0, , 48, 0, 0, 0, 0

db 0, , 240, 48, 0, 0, 0

db 0 , , 207 , 112 , 12 , 0 , 0
db 0 , 84 , 240 , 48 , 164 , 8 , 0
db 0 , 0 , 48 , 176 , 112 , 12 , 0
db 0 , 69 , 48 , 112 , 48 , 101 , 0
db 0 , 16 , 48 , 207 , 207 , 0 , 0
db 0 , , , 80 , 0 , 0 , 0
db 0 , 0 , 0 , 0 , 0 , 0 , 0
db 0 , 0 , 0 , 0 , 0 , 0 , 0

Dans le cas de param_number=5, nous pouvons inclure Vx comme paramètre à la fin :
|SETUPSP, 31, 5, Vy, Vx

De cette façon, nous mettrons à jour les deux vitesses en une seule commande, ce qui coûte 3,73 ms au lieu des 6,8 ms qu'il faudrait pour invoquer les deux commandes séparément.

Dans le cas de l'utilisation de param_number=7, en plus de changer la séquence d'animation, la commande met automatiquement à jour la frame (frame id), en la plaçant dans la frame initiale (le zéro) et l'adresse de l'image est mise à jour, de sorte que vous n'avez pas besoin d'invoquer param_number=9 pour changer l'image du sprite à la première image de la nouvelle séquence assignée. Si vous utilisez |ANIMALL avant l'impression ou

|Même si SETUPSP place l'animation à l'image zéro, vous sauterez à l'image 1 avant d'imprimer. Ce n'est normalement pas un problème, mais dans le cas d'une séquence de mort où, par exemple, la première image consiste à effacer le sprite, vous ne voudrez peut-être pas sauter directement à l'image 1. Dans ce cas, une astuce simple consiste à répéter l'image zéro dans la définition de la séquence de mort. De cette manière, vous vous assurez que l'image est visible. Une autre option est de supprimer le drapeau d'animation et de l'animer avec ANIMASP après l'impression.

Dans le cas du paramètre numéro=15, outre l'affectation du chemin au sprite dans la table d'attributs, la commande effectue une réinitialisation interne des données de sorte que le sprite commence à parcourir ce chemin à partir du premier segment du chemin en question.

21.1.25 |LES ÉTOILES DE L'EUROPE

Déplace une banque de 40 étoiles maximum sur l'écran (dans les limites fixées par |SETLIMITS), sans peindre sur d'autres sprites déjà imprimés.

**| STARS,<étoile
d'étoiles>,<couleur>,<dy>,<dx>,<dx>.** **initiale>,<nombre**

Exemple :

|STARS, 0, 15, 3, 1, 0

L'exemple déplace 15 étoiles de couleur 3 (rouge) d'un pixel verticalement (puisque dy=1 et dx=0). L'invocation répétée donne la sensation d'un fond d'étoiles défilant. Lorsqu'une étoile quitte la limite de l'écran ou celle fixée par |SETLIMITS, elle réapparaît de l'autre côté, ce qui donne une impression de continuité dans le flux d'étoiles.

La banque d'étoiles est située à l'adresse 42540 (=&A62C) et a une capacité de 40 étoiles jusqu'à l'adresse 42619. Chaque étoile consomme 2 octets, un pour la coordonnée Y et un pour la coordonnée X.

Les groupes d'étoiles peuvent être déplacés séparément, en commençant par l'étoile de votre choix. Les coordonnées initiales des étoiles doivent être initialisées par le programmeur.

Exemple d'initialisation et d'utilisation dans un rouleau de quatre plans en étoile pour

donner une impression de profondeur. Chaque avion se déplace à une vitesse différente

1 MEMOIRE 24999

10 CALL &6b78 : rem installe les commandes RSX

```

20 bank=42540
30 FOR star=0 TO 39:' boucle pour créer 40 étoiles
40 POKE banc+étoile*2,RND*200
50 POKE banc+étoile*2+1,RND*80
60 NEXT
70 MODE 0
80 REM nous peindrons et déplacerons 4 avions en étoile de 10 étoiles
chacun.
90 |STARS,0,10,3,0,-1 : ' 3 est rouge. Les plus éloignés se déplacent
davantage
lentement
91 |STARS,10,10,2,0,-2 : ' 2 est bleu
92 |STARS,20,10,1,0,-3 : ' 1 est jaune
93 |STARS,30,10,4,0,-4 : ' le 4 est blanc. Les plus proches vont plus
rapidement
95 goto 90

```

Les utilisations de cette commande peuvent être très diverses.

- L'utilisation simultanée de plusieurs bancs d'étoiles, avec des vitesses et des couleurs différentes, peut donner une impression de profondeur.
- Si la direction des étoiles est diagonale, vous pouvez obtenir un "effet de pluie".
- Si la couleur est noire et le fond brun ou orange, vous pouvez donner l'impression d'avancer sur un territoire sablonneux.
- Si le mouvement est un mouvement de roulement et que la couleur des étoiles est blanche, vous pouvez donner l'impression de neige. Le mouvement de roulement peut être obtenu par un zigzag en X tout en conservant la vitesse en Y, ou même en utilisant des fonctions trigonométriques telles que le cosinus. Évidemment, si vous utilisez le cosinus dans la logique du jeu, ce sera très lent, mais vous pouvez stocker la valeur précalculée du cosinus dans un tableau.

Exemple d'effet de neige :

```

1 MÉMOIRE 23999 : MODE 0
30 ' Initialisation de la banque de 40 étoiles
40 FOR dir=42540 TO 42619 STEP 2
45 POKE dir,RND*200:POKE dir+1,RND*80
48 NEXT
50 |STARS,0,20,4,2,dx1
60 |STARS,20,20,4,1,dx2
61 dx1=1*COS(i):dx2=SIN(i)
69 i=i+1 : SI i=359 ALORS i=0
70 GOTO 50

```

Il existe un moyen d'accélérer l'exécution, qui consiste à éviter de passer des paramètres. Tout au long de ce livre, nous avons vu que le passage de paramètres est coûteux, même si la commande invoquée ne fait rien. Dans le cas présent, il s'agit d'une commande qui nécessite 5 paramètres, ce qui la rend particulièrement coûteuse. Si nous voulons réduire le temps nécessaire à BASIC pour interpréter les paramètres, nous pouvons simplement invoquer la commande une fois avec des paramètres et les fois suivantes sans passer de paramètres.

|STARS,0,10,1,1,5,0

|STARS : cette invocation sans paramètre prend les mêmes valeurs que la dernière invocation.

Cette possibilité est particulièrement utile dans les jeux où l'on souhaite invoquer la commande à chaque cycle de jeu pour déplacer les étoiles, car elle permet d'économiser environ 1,7 ms.

IMPORTANT : La commande STARS est affectée par les limites de **|SETLIMITS**, mais seulement si Vx ou Vy sont différents de zéro. Si les deux sont à zéro, **|SETLIMITS** n'est pas affecté et des étoiles peuvent être peintes sur tout l'écran.

21.1.26 | UMAP

Cette commande met à jour la carte avec des informations situées dans une autre zone de mémoire où nous disposons d'une carte plus grande. La commande entraîne la reconstruction de l'ensemble de la carte, en incluant uniquement les éléments qui répondent à certaines plages de coordonnées X, Y (tous les paramètres sont des nombres de 16 bits).

Utiliser :

Par exemple, si nous avons une carte située à l'adresse 22.000 qui occupe 1500 octets et que nous voulons mettre à jour la carte avec les coordonnées de notre personnage, avec une marge suffisante pour avancer dans la coordonnée Y jusqu'à 100 lignes et dans la coordonnée x jusqu'à 20 octets dans toutes les directions :

| UMAP, 22000, 23500, y-100, y+100, x-20, x+20

Cette commande vérifiera les coordonnées des éléments situés sur la carte à l'adresse 23000 et s'ils se trouvent dans les marges X, Y que nous avons définies, ils seront copiés dans la zone de mémoire que le 8BP utilise pour la commande |MAP2SP, c'est-à-dire qu'il les copiera à partir de l'adresse 42040. Cependant, il ne copiera que ceux qui remplissent la condition. Comme il y a moins d'éléments, la commande |MAP2SP s'exécutera plus rapidement car elle devra lire et vérifier s'il y a moins d'éléments à l'écran.

IMPORTANT : la carte à copier ne doit PAS inclure les 3 paramètres de chaque carte :

<Max high> (qui est un DW, c'est-à-dire 2 octets)

`<max_width>` (qui est un DW, c'est-à-dire 2 octets)

<Num_items> (qui est un DB, c'est-à-dire 1 octet)

C'est-à-dire qu'il y a 5 octets qui ne doivent pas être inclus dans la carte à copier.

22 Comment fabriquer un tableau d'affichage

Dans de nombreux jeux, il est intéressant de disposer d'un mécanisme de tableau d'affichage (également appelé "Hall of fame") qui stocke les meilleurs scores de manière ordonnée à partir de différents jeux. Il peut être programmé en BASIC au moyen d'un tableau qui stocke le score de chaque partie, mais chaque fois que l'on arrête le jeu et que l'on exécute RUN, l'interpréteur BASIC exécute en interne un CLEAR et toutes les valeurs de ce tableau sont effacées. Une façon d'éviter cela est d'empêcher l'utilisateur d'interrompre le programme en appuyant deux fois sur la touche ESC à l'aide de la routine CALL &bb48 du micrologiciel. Une autre option consiste à stocker les points dans un tableau en mémoire et à le lire et le modifier à partir de BASIC. Il existe de nombreuses façons de programmer cette opération, dont voici un exemple. Les étapes à suivre sont les suivantes :

Inclure dans make_all.asm une lecture du fichier "score_table.asm".

```
;----- CODIGO -----  
;comprend la bibliothèque 8bp et le lecteur de  
musiqueWYZ lire "make_codigo_mygame.asm".  
;----- MUSICA -----  
lire "make_musica_mygame.asm" ;  
comprend les chansons.  
; ----- GRAPHIQUES -----  
; cette partie comprend des images et des séquences  
d'animation, lisez "make_graficos_mygame.asm".  
lire "score_table.asm"
```

Ensuite, nous devons créer un fichier score_table.asm avec des exemples de données. J'en ai créé un avec des noms de déesses sumériennes et des scores de 10 à 1. Chaque nom prend 8 caractères. Une chose très importante est l'**org _end_graph**. Avec cette commande, nous indiquons que le tableau va être assemblé après les graphiques, dans les adresses mémoire qui suivent.

```
org _end_graph  
TABLEAU DE  
DÉOUISHTAIRE "  
dw 10 db  
"ANTU"  
dw 9  
db "INANNA"  
dw 8  
db "NIMUG"  
dw 7  
db "NIMBARA"  
dw 6  
db "ASTA"  
dw 5  
db "DAMKINA"  
dw 4  
db "NEBAT"  
dw 3  
db "NISABA"  
dw 2  
db "NINSUB"  
dw 1  
TABLE_DE_FIN_DE_S  
CORE
```

Maintenant vient la partie BASIC : Nous allons assembler avec winape et ensuite nous allons regarder (avec winape, option assemble->symbols) dans quelle adresse mémoire la balise end_graph a été assemblée. Dans mon cas, il s'agit de &9685. Dans notre programme BASIC nous allons nous prendrons en compte (je la stocke dans la variable "dir")

```

190 ' --- hall of fame
200 DIM pts(11) : DIM name$(11) : "scores
210 GOSUB 2040 : 'lire le tableau des scores
220 INK 3,7:PEN 3:LOCATE 15,12:PRINT " Hall of fame ":LOCATE 15,13:PRINT
" -----
230 p=1:FOR i=0 TO 9:LOCATE 1,i+14:PEN p :PRINT , name$(i), pts(i) :
p=1+(p MOD 3):NEXT
240 b$=INKEY$:IF b$="" THEN 240 ELSE 250

```

Examinons la routine qui lit le tableau des scores à la ligne 2040.

```

2040 '--- LIRE LE TABLEAU DES SCORES
2050 dir=&9685 : FOR i=0 TO 9 : name$(i)=""
2070 FOR j=dir TO dir +7:'lee character a character les 8 lettres
2080 letter=PEEK (j) :
name$(i)=name$(i)+CHR$(letter)
2090 SUIVANT j : dir=dir+8:'après les 8 lettres il y points (un nombre
a les entier)
2100 pts(i)=0:| PEEK,dir,@pts(i):dir=dir+2:'un entier est de 2 octets
2110 NEXT i
2120 RETOUR

```

Enfin, à chaque fin de partie, nous vérifions si le score (variable "score" dans l'exemple) est supérieur à un enregistrement de la table des scores (tableau "pts") et si c'est le cas, nous modifions la table. En modifiant le tableau dans les adresses mémoire, nous ne perdrons pas les valeurs, même en cas de RUN.

```

1800 '--- TERMINER LE JEU ET VÉRIFIER LE MEILLEUR SCORE ---
1810 ENCRE 0,0:BORDURE 5 : ENCRE 2,15:ENCRE 1,20:| MUSIQUE
1820 j=10:FOR i=9 TO 0 STEP -1:IF score>pts(i) THEN j=i:NEXT
1830 IF j=10 THEN RUN:'end game & start
1831 déplace tous les scores inférieurs d'une position.
1840 FOR i=8 TO j STEP -1 : pts(i+1)=pts(i) : name$(i+1)=name$(i) : NEXT
1850 b$=INKEY$:IF b$<>"" THEN 1850:'clean buffer keyboard
1860 MODE 1 : BORDER 5 : INK 3,8 : LOCATE 6,8 : PEN 3 : PRINT
"CONGRATULATIONS ! NOUVEAU MEILLEUR SCORE".
1880 LOCALISER 14,10:PEN 2:IMPRIMER "ENTREZ VOTRE NOM".
1900 LOCALISER 15,12:PEN 1:INPUT nom$(j) : nom$(j)=MID$(nom$(j),1,8)
1910 pts(j)=score
'--- ÉCRIRE LE TABLEAU SUR LA MÉMOIRE --
1930 dir=&9685 : FOR i=0 TO 9 : k=1
1950 FOR j=dir TO dir +7:'nous écrivons caractère par caractère, les 8
lettres
1960 dato$=MID$(name$(i),k,1) : IF dato$="" ALORS dato$=" "
1970 dato=ASC(dato$)
POKE j,data:k=k+1:NEXT j
1990 dir=dir+8 : 'nous écrivons la ponctuation après le nom (8 lettres)
2000 |POKE,dir,pts(i)

```

2010 dir=dir+2:'le score est un entier = 2 octets

2020 NEXT i

2030 RUN

23 Améliorations futures possibles de la bibliothèque

La bibliothèque 8BP peut être améliorée par l'ajout de nouvelles fonctions qui pourraient ouvrir de nouvelles possibilités au programmeur. Voici quelques suggestions à cet effet

23.1 Mémoire pour la localisation de nouvelles fonctions

Actuellement, grâce au mécanisme des "**options d'assemblage**", la bibliothèque vous laisse une quantité de mémoire libre pour votre listing BASIC qui dépend de chaque option.

- Option 0 : 23.5 KB gratuit (ne devrait être utilisé que pour tester des choses)
- Option 1 : 25 KB gratuits (jeux de labyrinthe)
- Option 2 : 24.8 KB libre (jeux avec défilement)
- Option 3 : 24 KB gratuits (jeux avec pseudo 3D)

La bibliothèque pourrait encore s'agrandir, grâce à une option 4 qui étend les capacités de l'option 1, par exemple grâce à des capacités de "filmation". Cette option 4 pourrait fournir de telles capacités en utilisant 1 Ko et en laissant 24 Ko libres à l'utilisateur.

23.2 Impression à résolution de pixels

Actuellement, le 8BP utilise la résolution et les coordonnées par octet, qui sont 2 pixels du mode 0. En fait, une façon de contourner cette limitation est de définir 2 images pour le même sprite qui sont décalées d'un seul pixel. Lorsque vous déplacez le sprite sur l'écran, vous pouvez alterner entre le simple remplacement de l'image par l'image décalée et le déplacement du sprite d'un octet. Vous obtenez ainsi un déplacement pixel par pixel. Cette technique est décrite en détail au chapitre 13

23.3 Layout de mode 1

La présentation actuelle fonctionne comme un tampon de caractères de $20 \times 25 = 500$ octets. Il peut être utilisé sans problème dans les jeux en mode 1, mais il y aura des choses que nous ne pourrons pas faire, comme définir une partie qui prend 3 caractères de largeur en mode 1, car les caractères en mode 0 prennent deux fois la largeur des caractères en mode 1. Ce n'est pas un problème, mais c'est une limitation.

Une disposition en mode 1 occuperait 1 Ko, soit $40 \times 25 = 1000$. Étant donné que la disposition en mode 0 et la disposition en mode 1 ne seraient pas utilisées simultanément, elles pourraient se chevaucher dans la mémoire et, compte tenu du fait que la disposition en mode 0 se trouve entre 42000 et 42500, nous placerions simplement la disposition en mode 1 entre 41500 et 42500, "volant" 500 octets de la mémoire des sprites de 8 Ko, située entre 34000 et 42000.

Les changements apportés à cette amélioration sont minimes et ne concernent que deux fonctionnalités

Le layout0/layout1 et **|LAYOUT** et **|COLAY** devraient être conscients du mode de l'écran, au moyen d'une variable qui agit comme un drapeau (layout0/layout1). Cette modification serait très bien, mais même sans elle, nous pouvons toujours utiliser les layouts dans les jeux en mode 1 sans problème.

23.4 Capacité de filmation

Il pourrait être intéressant de créer un mode "filmation" pour créer des jeux de type

"knight lore". En utilisant les fonctions existantes de la bibliothèque, avec un peu de code supplémentaire, cette capacité intéressante pourrait être mise en œuvre. Il est possible que cette fonctionnalité soit ajoutée prochainement.

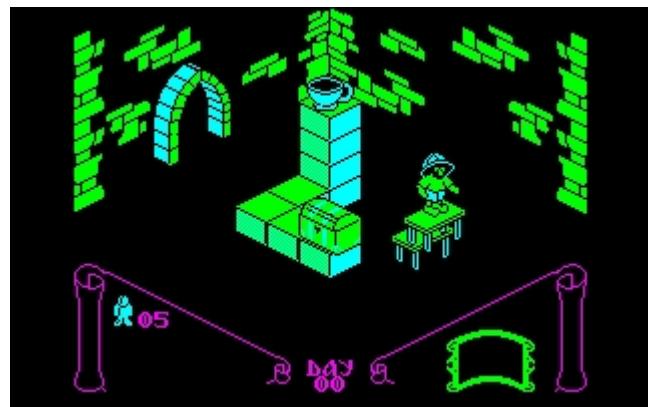


Fig. 120 : le mythique "Knight Lore".

23.5 Fonctions de défilement du matériel

Il existe quelques jeux Amstrad CPC avec un défilement fluide de qualité, programmés en utilisant les capacités du contrôleur vidéo M6845. Actuellement, la bibliothèque dispose d'un mécanisme de défilement basé sur le CPU (non basé sur le matériel, mais efficace et polyvalent pour les jeux avec défilement dans n'importe quelle direction de mouvement).

Le fait qu'il n'y ait pas beaucoup de jeux de ce type est dû au fait que dans les années 1980, les programmeurs de jeux vidéo n'avaient pas beaucoup d'informations et, dans de nombreux cas, étaient des amateurs.

Parmi les rares jeux à défilement fluide, on trouve 2 jeux de Firebird :

- "Mission Genocide" (tiré de Firebird, 1987, par Paul Shirley, un excellent programmeur qui a également inventé une technique d'écrasement ultra-rapide sans utiliser de masques).
- "Warhawk" (par Firebird, 1987)



Fig. 121 Jeux Firebird avec défilement rapide et fluide

La technique de défilement de ces deux jeux est la même, appelée "défilement vertical". La technique de défilement consiste à contrôler exactement l'instant où le défilement de l'écran se produit. A cet instant, nous trompons le CTRC 6845 en lui disant que l'écran se termine plus tôt que d'habitude. Cependant, avant de terminer cette section de l'écran, nous lui demandons d'incorporer moins de lignes de balayage que ce qui correspond à une section de cette taille. Ensuite, à un moment très précis que nous devons contrôler à la microseconde près, nous demandons à la puce de commencer un nouvel écran, sans

avoir produit le signal de synchronisation verticale. Cela nous permet de dessiner un deuxième

(les marqueurs, par exemple) et compenser le nombre de lignes de balayage de la première section. Si le mécanisme de compensation des lignes de balayage est correct, il est possible de faire bouger l'une des deux sections de l'écran de manière extraordinairement fluide. Le problème de l'application de cette technique à une commande à utiliser à partir de BASIC est que le contrôle des interruptions est imprécis en raison de l'exécution de l'interpréteur, et que nous avons besoin ici d'un contrôle très, très précis.

Le problème du défilement matériel (qui affecte également le défilement logiciel qui déplace tout l'écran) est qu'il "entraîne" les sprites présents avec lui, de sorte que lorsque vous les repositionnez, vous remarquerez une vibration indésirable chez les ennemis et/ou chez votre personnage. Pour résoudre ce problème, vous pouvez utiliser le double buffering et basculer entre deux blocs de 16Ko à chaque fois qu'une image est prête. Cela vous empêchera de voir "comment chaque image est faite". Dans le 8BP, j'ai abandonné le double buffering afin de laisser un bon espace de RAM au programmeur et c'est pourquoi ces techniques n'ont pas été implémentées.

Pour toutes ces raisons, le défilement dans 8BP est basé sur une carte du monde qui ne fait pas glisser les sprites lorsqu'ils se déplacent et qui est donc plus efficace car elle déplace moins de mémoire tout en permettant un mouvement multidirectionnel.

23.6 Migration de la bibliothèque 8BP vers d'autres micro-ordinateurs

Cette bibliothèque serait facilement transférable à d'autres micro-ordinateurs basés sur le Z80, tels que le Sinclair ZX Spectrum. Dans le cas du ZX Spectrum, il serait nécessaire de réécrire les routines qui peignent sur l'écran, car la mémoire vidéo est gérée différemment. La migration du ZX est déjà un projet ferme, après avoir reçu de nombreuses demandes de la part des utilisateurs du ZX.

La migration de la bibliothèque vers un Commodore 64 serait également possible, bien que le code d'assemblage ne puisse pas être réutilisé, puisqu'il est basé sur un autre microprocesseur. En outre, dans le cas du Commodore 64, la migration de la bibliothèque 8BP devrait tirer parti des caractéristiques propres à la machine, telles que ses 8 sprites matériels, de sorte que ce que la bibliothèque 8BP devrait incorporer en interne serait un multiplexeur de sprites, offrant 32 sprites, mais utilisant en interne les 8 sprites matériels.



Fig. 122 Sinclair ZX et Commodore 64, deux classiques

24 Quelques jeux réalisés avec 8BP

Dans ce chapitre, je vais décrire comment sont réalisés certains jeux que vous pouvez trouver sur le site <https://github.com/jjaranda13/8BP> réalisés avec 8BP (du plus récent au plus ancien) :

- **Paco the man** : un jeu de type puzzle, qui utilise la technique du mouvement doux (demi-octet) et des logiques de masse avancées.
- **NOMWARS** : un jeu de type "commando".
- **Blaster pilot** : un jeu de défilement multidirectionnel inspiré de jeux tels que "Time Pilot" ou "Asteroids".
- **Happy Monty** : jeu de style mutant Monty au rythme effréné
- **Eridu** : un jeu classique de type "scramble" avec défilement horizontal.
- **Space phantom** : inspiré du space harrier
- **Eternal Frogger** : un remake du classique Frogger, présenté à la célèbre foire "eternal amstrad".
- **3D Racing one** : premier jeu de course à utiliser la pseudo 3D
- **Fresh Fruits & vegetables** : un jeu de plateforme utilisant le défilement horizontal et une gestion avancée des trajectoires des sprites.
- **Nibiru** : un jeu de vaisseau à défilement horizontal, utilisant des fonctionnalités avancées de 8BP
- **Anunnaki** : un jeu de bateau, genre arcade
- **Mutante Montoya** : un jeu à défilement d'écran. On pourrait le classer dans la catégorie des jeux de plates-formes. C'est le premier jeu que j'ai réalisé avec 8BP.
- **Mini-jeux** : il s'agit de jeux courts, simples et didactiques destinés à vous familiariser avec la programmation avec le 8BP. Il existe une version du classique "pong", appelée "Mini-pong", et une version du classique "Space Invaders", appelée "mini-invaders".

24.1 Mutant Montoya

Un premier hommage à l'Amstrad CPC, avec un titre inspiré du classique "mutant monty".

Il s'agit d'un jeu simple à 5 niveaux. Il est basé sur l'utilisation de la disposition 8BP pour construire chaque écran.





24.2 Les Anunnaki, notre passé extraterrestre

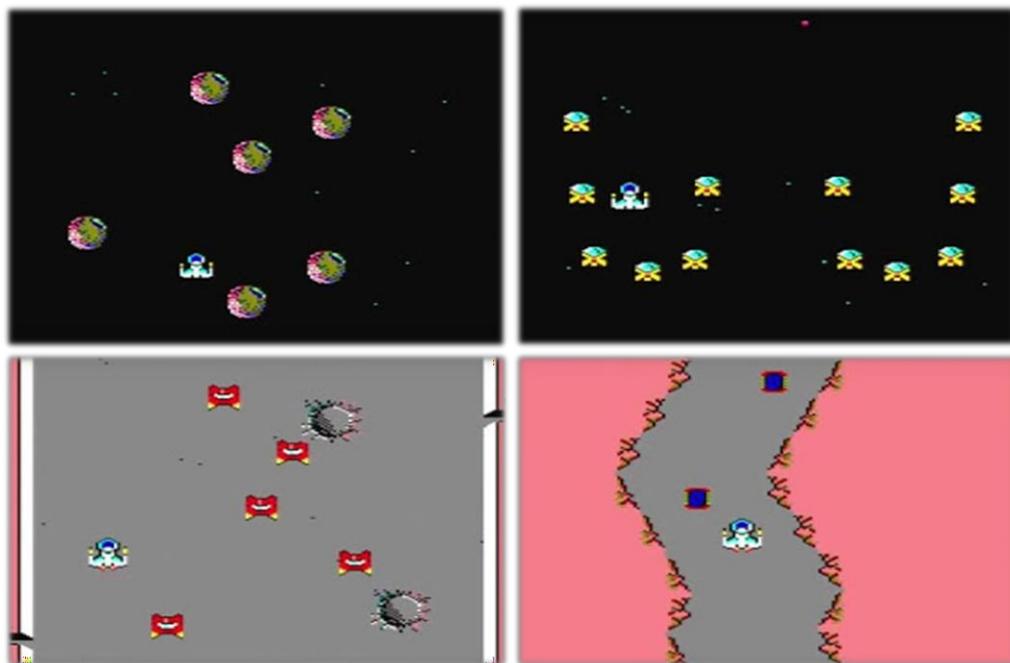
Il s'agit d'un jeu vidéo d'arcade très intéressant pour analyser et apprendre la technique de programmation par "logique massive". Lorsqu'il a été programmé, la bibliothèque 8BP ne disposait pas encore de commande de défilement ni de routage des sprites. C'est pourquoi la programmation de ce jeu est si intéressante, car tout y est réalisé par le biais de la logique massive.

Contrairement à "Mutant Montoya", le jeu vidéo "Anunnaki" n'utilise pas la disposition, car il s'agit d'un jeu de progression et de destruction de navires ennemis, et non d'un jeu de labyrinthes ou de passage d'écrans. Ce jeu utilise également de défilement "simulé", très intéressant.

Vous incarnez Enki, un commandant Anunnaki qui affronte des races extraterrestres afin de conquérir la planète Terre et de soumettre les humains à sa volonté.

Le jeu se compose de 2 niveaux, bien que si vous perdez une vie, vous continuez au point du niveau où vous êtes, vous ne revenez pas au début du niveau.

Le premier niveau est une étape dans l'espace interstellaire, où vous devez esquiver des météores et tuer des hordes de vaisseaux et d'oiseaux de l'espace. Le deuxième niveau se déroule sur la lune, où vous devez détruire des hordes de vaisseaux, après quoi vous devez traverser un tunnel truffé de mines jusqu'à ce que vous rentriez trois "boss" que vous devez détruire.



24.3 Nibiru

Il s'agit d'un jeu qui teste de nombreuses caractéristiques du 8BP et de la technique de programmation "logique massive", et qui comporte des détails tels qu'un graphique de charge fantaisiste et trois musiques en jeu, ainsi qu'un tableau des scores qui ne se perd pas même si vous redémarrez le jeu et d'autres aspects techniques avancés tels que le défilement parallaxique, les itinéraires, les macros, etc. Le listing BASIC fait un peu plus de 16KB.

Vous êtes le pilote d'un vaisseau destroyer et devez vaincre la planète Nibiru et son chef, "Gorgo", un ancien reptile presque invincible. Vous devez détruire les oiseaux galactiques qui vivent sur ses lunes et, une fois que vous aurez atteint la planète, vous devrez affronter ses dangers avant de pouvoir combattre Gorgo.

Le jeu se compose de trois phases et utilise le mécanisme de défilement de 8BP basé sur la commande MAP2SP, ainsi que le routage des sprites et les macros d'animation - le tout en BASIC ! grâce à 8BP et à la technique de la "logique massive".



Le jeu maintient un tableau des scores qui n'est pas effacé, même si vous arrêtez le jeu. Pour ce faire, il le stocke en RAM avec des pokes, au lieu de le stocker dans des variables BASIC.

24.4 Fruits et légumes frais

Il s'agit d'un jeu de plateforme, où votre mission est de collecter tous les fruits pour laisser la population sans rien à manger, de sorte qu'elle doive sacrifier un pauvre cochon pour se nourrir.

Sa principale nouveauté est l'utilisation avancée des itinéraires, concaténés les uns aux autres pour passer de la "chute" à la "marche", et l'utilisation de RINK combinée à MAP2SP comme technique de défilement.



Les briques bleues de la présentation du jeu utilisent une impression synchronisée des sprites (PRINTSPALL,0,0,1) alors que pendant le jeu, l'impression des sprites n'est pas synchronisée (PRINTSPALL,0,0,0). L'effet de la synchronisation sur la présentation est que tout semble synchronisé, y compris l'animation de l'encre RINK (utilisée sur les briques bleues). Cela donne un effet de défilement fluide. À mon avis, il ne faut pas synchroniser un jeu car, bien que vous obteniez plus de fluidité, la vitesse du jeu diminue également. Selon le type de jeu auquel vous jouez, vous pouvez être intéressé ou non.

24.5 "3D Racing one

Il s'agit du premier jeu réalisé en utilisant la capacité pseudo3D de 8BP. Il comporte un graphique de chargement fantaisiste et 4 pistes : une piste d'entraînement avec des flaques d'eau sur la route, une piste où nous mesurons à 4 autres voitures, une piste où la vitesse est doublée, en utilisant des segments de pente plus faible, et une étape finale de nuit. Le jeu utilise également la commande PRINTAT et son propre jeu de caractères. Il dispose en outre d'un tableau d'affichage, d'une musique principale, de deux musiques secondaires et d'effets sonores.

Pour la présentation, une carte 2D remplie de balles est utilisée et la commande MAP2SP avec écrasement est configurée. Les boules sont des "images zoom".

Le premier circuit a été construit avec un fichier dans lequel nous avons placé un par un tous les éléments (segments, panneaux, arbres, flaques d'eau...) mais le reste des circuits est créé dynamiquement à partir de BASIC, en poke l'adresse de la carte du monde.

Pour détecter les collisions et les sorties de route, la commande PEEK est utilisée à la

place de COLSPALL, de sorte que dès qu'elle détecte un octet sur la voiture d'une couleur différente de celle de la route, vous êtes considéré comme étant en collision et votre moteur est endommagé.

Une autre nouveauté intéressante est l'utilisation d'itinéraires dynamiques. Le circuit de compétition et les itinéraires des voitures sont créés à partir de BASIC, en suivant la procédure expliquée dans ce manuel.



24.6 Fantôme de l'espace

Il s'agit d'un jeu réalisé avec la version v35 de la bibliothèque. Il utilise les capacités de la pseudo-3D pour la présentation des titres dans le style "Star Wars". Il utilise également l'impression transparente avec des sprites (pièces) passant derrière l'arrière-plan (le tableau d'affichage).

Le jeu est inspiré du classique "Space Harrier" et vous contrôlez un héros équipé d'un jet-pack qui vole dans l'espace, tuant des météores, des ovnis, des oiseaux de l'espace et un dragon comme ennemi final. Le jeu se compose de trois étapes et d'un final épique. Le jeu de caractères est interne, bien que seuls les chiffres aient été définis, dans le style d'une "horloge casio" pour les marqueurs.

Dans la première phase, les routes sont utilisées pour les étoiles, qui sont des sprites, ainsi que pour les météores et les oiseaux.

Le second utilise l'écrasement de sprites avec un arrière-plan de 2 bits (4 couleurs) et l'animation d'encre à l'aide de RINK. Les vaisseaux d'une rangée sont construits en les plaçant à l'aide de la commande ROUTESP améliorée, disponible dans la version 35.

Bien que le jeu simule 3 dimensions, il n'utilise pas de projection pseudo-3D, mais plutôt des chemins de sprites dans lesquels la version de l'ennemi est changée en une version plus grande pour donner l'impression qu'il s'approche. Ainsi, une collision avec un ennemi éloigné ne nous affecte pas,

un registre d'état des drapeaux inutilisé est utilisé pour marquer les ennemis éloignés comme inoffensifs.

Dans la troisième phase, le mouvement horizontal relatif des pierres sur le sol a été utilisé en combinaison avec une trajectoire de mouvement vertical accéléré.



24.7 Frogger éternel

Le jeu "Frogger Eternal" a été réalisé avec la version V36 du 8BP, et son titre évoque à la fois le classique "frogger" de konami sorti en 1981 et la foire "Amstrad Eternal" qui s'est tenue en 2019, l'événement où ce jeu a fait son apparition.

Il s'agit d'un jeu programmé en mode 1, utilisant LAYOUT, l'impression transparente sur la grenouille et des parcours de nature variée pour les sprites. Certains sprites sont invisibles, mais collisibles, comme 4 rivières "invisibles" sous des troncs, des nénuphars et des tortues que la grenouille doit enjamber, comme des "murs invisibles" sur les côtés de la rivière, pour que la grenouille ne puisse pas s'enfuir.



24.8 Eridu : Le port spatial

Ce jeu est un clone du classique "Scramble" de Konami créé en 1981. Il présente de nombreuses différences par rapport à l'original, mais il s'inspire essentiellement du jeu classique, car il intègre la nécessité de faire le plein, ce qui oblige le joueur à prendre des risques pour détruire les réservoirs de carburant afin de ne pas perdre de vie.

Il est assez rapide malgré l'utilisation d'un puissant scroller, affichant 32 sprites à l'écran à de nombreux endroits. Il atteint jusqu'à 18 fps

Le jeu comporte 5 étapes et différentes musiques, ainsi qu'une présentation graphique très sophistiquée.

Eridu est un jeu vidéo qui renvoie à une ancienne histoire "interdite" de l'humanité. Eridu fut la première ville du monde, créée par les "Anunnaki" il y a 400 000 ans, une race extraterrestre selon les tablettes sumériennes trouvées dans le désert d'Irak. Ils y ont établi un port spatial appelé "Terre 1".



Les cartes des différentes phases sont chargées dans différentes banques de mémoire, chacune occupant 500 octets de données mondiales et 200 octets pour la description des emplacements des ennemis. Le défilement se fait logiquement avec la commande **|MAP2SP.**

24.9 Happy Monty

Il s'agit d'un jeu de balayage d'écran rapide, qui imite presque parfaitement le classique Mutant Monty. Il "clone" même son écran initial. Ce jeu est réalisé avec la version 37 de la bibliothèque et atteint 25 FPS.

Il fait un usage intensif de la disposition et, bien sûr, des logiques massives. Il parvient à stocker 25 niveaux en utilisant une technique simple pour compacter les dispositions (chaque disposition n'occupe que 160 octets) expliquée dans ce livre.



Une technique originale utilisée dans ce jeu permet aux ennemis de changer d'itinéraire. Il s'agit de sprites invisibles "inverseurs". Lorsqu'un ennemi entre en collision avec un sprite inverseur, il change de chemin et se voit attribuer le chemin opposé ou même un chemin perpendiculaire, ce qui permet de construire des chemins de n'importe quelle longueur sans avoir à définir plus d'un chemin vertical et d'un chemin horizontal. Il est même possible de faire des boucles.

Cela simplifie également la création de la carte (tracé + ennemis) de chaque niveau, car pour localiser un ennemi il suffit d'utiliser un code (un caractère) qui indique la vitesse et la direction de l'ennemi (6 lettres sont utilisées pour tous les types d'ennemis et changent périodiquement la "palette" d'ennemis, les poupées associées à ces 6 lettres).

24.10 Pilote de blaster

Il s'agit d'un jeu créé avec la version v39 de 8BP, dans lequel il est possible d'avoir des effets sonores (créés avec SOUND) en même temps que la musique est jouée en utilisant la commande MUSIC. Dans ce cas, la commande SOUND est utilisée pour les coups de feu et les explosions et utilise le troisième canal alors que la musique utilise les deux premiers canaux.



Le jeu dispose d'un défilement multidirectionnel et s'inspire du style de jeux tels que "Time Pilot" ou "Asteroids". À côté du panneau des scores et des vies, il y a un radar avec lequel vous pouvez vous orienter dans l'espace pour localiser 3 acteursnauts perdus que vous devez sauver. Le jeu est intéressant dans la façon dont la programmation du mouvement du vaisseau dans 12 directions possibles est gérée aussi efficacement que possible.

En plus des 6 niveaux, il dispose d'une phase de bonus à la fin de chaque niveau, qui permet d'accumuler des points et de récupérer une vie.

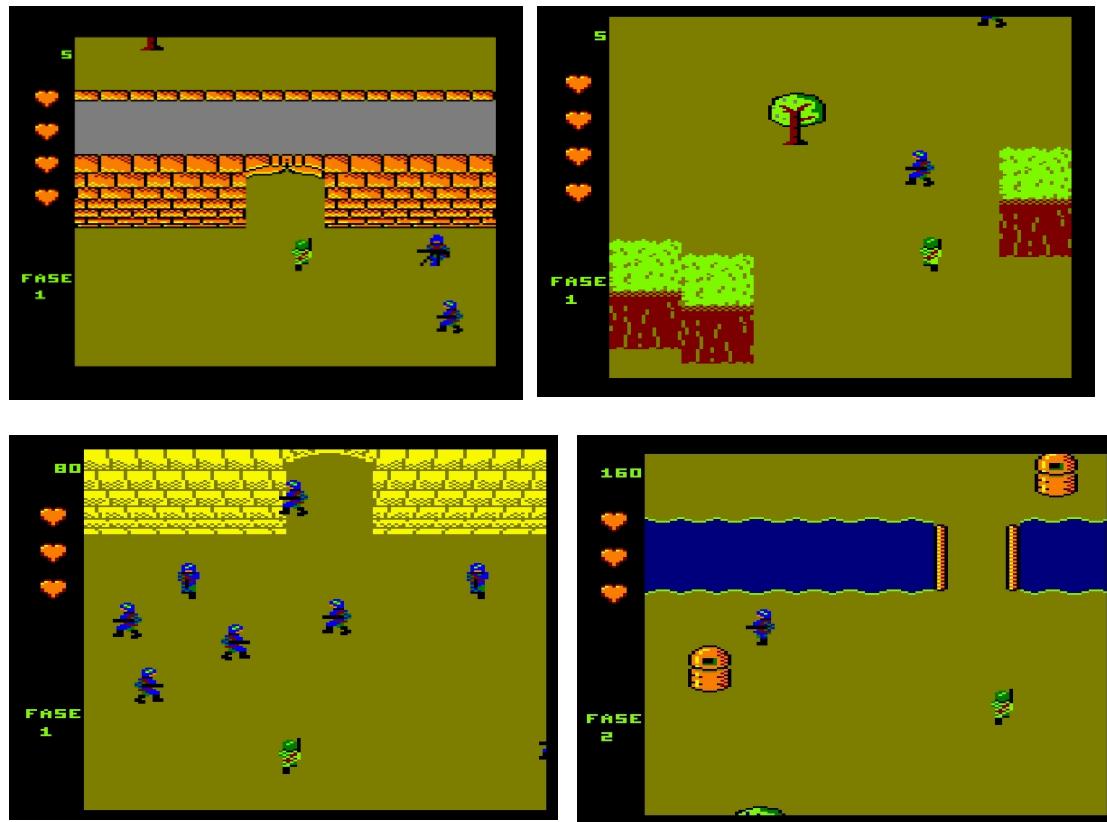
24.11 NOMWARS

Il s'agit d'un jeu dans le style du classique "Commando" créé en 1985 par Capcom. Un shoot'em up classique à défilement vertical.

Le jeu se compose de 4 étapes et d'une introduction "fantaisiste", avec une histoire sur la guerre du nouvel ordre mondial. Il a été publié au format DES.

Cette version exploite la capacité de défilement de 8BP (commande MAP2SP) et inclut le célèbre pont sous lequel Joe passe en utilisant une technique basée sur la commande SETLIMITS de 8BP.

Le jeu est proposé en deux versions : la version BASIC pure et la version cycle compilée (cycle traduit en langage C à l'aide du wrapper 8BP et du minibasic 8BP). Les deux versions sont identiques, la traduction en C étant une réplique totale, presque un miroir de la version BASIC. En fait, le jeu a été programmé en BASIC et, le dernier jour, il a été traduit en C avec les facilités offertes par 8BP.

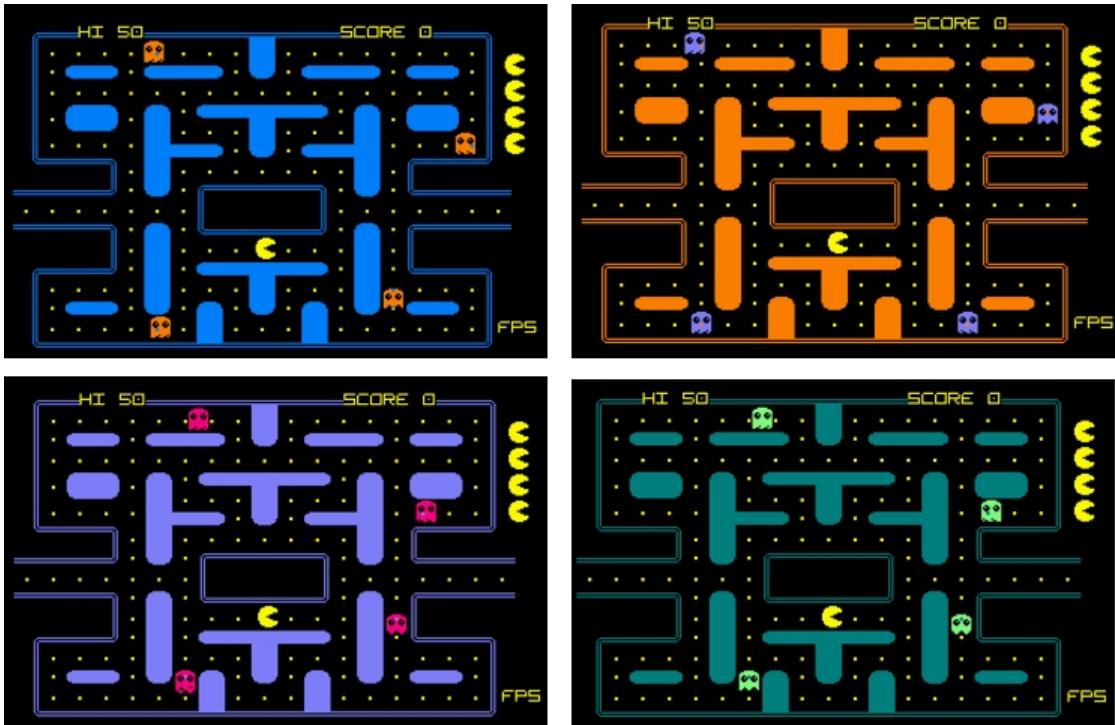


24.12 Paco, l'homme

Un jeu dans le plus pur style Pac-Man. Le jeu contient deux phases en BASIC et deux en cycle compilé. En BASIC, il atteint 19 FPS avec la logique du labyrinthe, des collisions et des 4 fantômes, et en C, il atteint 33 FPS.



Chaque niveau de Paco est identifié par ses couleurs. Les deux premiers fonctionnent en BASIC et utilisent des décisions fantômes "précalculées" pour atteindre 19 fps. Les troisième et quatrième niveaux utilisent le langage C et atteignent 33 images par seconde.



24.13 Mini-jeux

Il s'agit de jeux conçus à des fins éducatives. Simples à comprendre et courts, ils aident les programmeurs débutants à développer leurs propres jeux.

24.13.1 Mini-pong

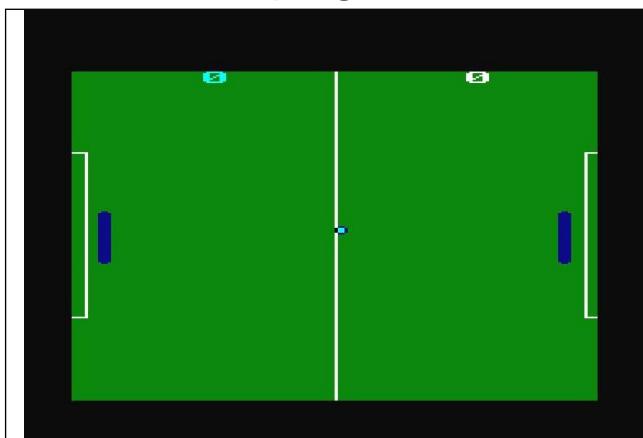


Fig. 123 Jeu vidéo "Mini-pong"

Il s'agit d'un jeu vidéo très simple et didactique. Il est basé sur le classique "Pong" d'Atari (1972). La barre de l'adversaire (l'ordinateur) commence à prendre des décisions lorsque le ballon dépasse la moitié du terrain, il est donc possible de le battre. Si nous l'obligeons à prendre des décisions plus tôt, il arrive un moment où il est impossible de gagner.

Quelques détails sur le jeu :

- Utilisation de **|COLSPALL** : pour détecter les collisions entre le "collider" (la balle) et les "colliders" (les barres). Dans l'octet d'état des sprites, le drapeau de collision est activé pour la balle (sprite 29) et le drapeau de collision est activé sur 31 (notre barre) et 30 (la barre de l'adversaire).
- Utilisez overwrite sur le ballon pour respecter la bande blanche sur le terrain et ne pas l'effacer lors d'une passe. Pour ce faire, utilisez une palette avec overwrite et activez le drapeau overwrite dans l'octet de statut du sprite du ballon (le 29e).
- il n'y a que deux images (la balle=17 et la barre=16) qui sont assignées aux 2 sprites (les sprites 30 et 31 sont assignés à l'image 16 et le sprite 29 est assigné à l'image 17).
- Les sprites utilisent le mouvement automatique. Pour cela, le drapeau de mouvement automatique est activé et la commande **|AUTOALL** les déplace (modifie leurs coordonnées) en fonction de leur vitesse.
- Tous les sprites sont imprimés avec **|PRINTSPALL** à chaque cycle de jeu.

24.13.2 Mini-Invaders

Comme "Mini-pong", il s'agit d'un jeu à but éducatif, inspiré du classique "Space Invaders" de Taito (1978).



Fig. 124 Jeu vidéo "mini-invaders".

Quelques conseils sur la manière de procéder :

- Le jeu utilise 32 sprites
- Le navire est le sprite 31
- Les tirs que vous pouvez effectuer avec le navire sont 29 et 30.
- Les envahisseurs tirent en utilisant le sprite 28
- Les envahisseurs utilisent les sprites 0 à 27 (28 envahisseurs au total).
- Les sprites 31, 30 et 29 ont un drapeau de collision actif.
- Les autres sprites sont "collés" et ont un drapeau de collision actif.
- Les envahisseurs ont un drapeau de mouvement automatique actif et sont associés à la route "0" qui les déplace de droite à gauche et vers le bas, ce qui est typique des envahisseurs.
- Les déclencheurs du navire et de l'envahisseur utilisent une caractéristique du V27, ils traversent l'écran et sont automatiquement désactivés à la fin de leur parcours avec un changement d'état défini, ce qui simplifie la logique BASIC et accélère donc le jeu.

25 ANNEXE I : Organisation de la mémoire vidéo

25.1 L'œil humain et la résolution du CPC

La mémoire vidéo de l'Amstrad CPC a 3 modes de fonctionnement. Le mode le plus utilisé pour les jeux est le mode 0 (160x200) parce qu'il a plus de couleurs, mais le mode 1 (320x200) a également été beaucoup utilisé pour les jeux de programmation. Le mode 2 (640x200) a été rarement ou jamais utilisé pour les jeux en raison de sa limitation à 2 couleurs.

Comme la quantité de mémoire vidéo est la même, la résolution est sacrifiée au profit du nombre de couleurs, mais curieusement, à l'horizontale, qui est le côté le plus long de l'écran, la résolution est inférieure à celle de la verticale (160 à l'horizontale et 200 à la verticale). On peut se demander pourquoi. D'ailleurs, ce n'est pas le seul micro-ordinateur qui a fait cela, beaucoup d'autres ordinateurs ont également utilisé la même stratégie avec le côté horizontal.

La raison est liée au fonctionnement du système visuel humain. L'œil perçoit plus de détails verticalement qu'horizontalement, de sorte qu'endommager la résolution sur l'axe horizontal n'est pas aussi grave que de l'endommager verticalement. Subjectivement, le résultat est plus acceptable. Le système visuel humain est en mode 0, c'est-à-dire avec des pixels très larges. C'est pourquoi le champ de vision horizontal est plus grand que le champ de vision vertical.

25.2 Mémoire vidéo

Les informations les plus complètes et les plus claires se trouvent dans le manuel du firmware Amstrad. Ces informations seront utiles si vous voulez construire un éditeur de sprites amélioré ou si vous voulez entrer dans l'assembleur et programmer des routines d'écrasement d'impression ou tout autre chose.

25.2.1 Mode 2

En mode 2, chaque pixel est représenté par un bit. Un octet représente donc 8 pixels. Si nous prenons n'importe quel octet de la mémoire vidéo, sa correspondance avec les pixels est de 1 bit pour chaque pixel, dans ce tableau les bits sont représentés et à quels pixels (p) ils correspondent.

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
p0	p1	p2	p3	p4	p5	p6	p7

Dans un octet, le bit 7 est numéroté comme étant le bit le plus à gauche. Le pixel 0 est également le pixel le plus à gauche, c'est-à-dire qu'il n'y a rien "à l'envers" ici. Tout est correct

25.2.2 Mode 1

En mode 1, nous avons 4 couleurs (représentées par 2 bits). Un octet représente donc 4 pixels. La correspondance entre pixels et bits est un peu plus complexe. Le pixel 0, par exemple, est codé avec les bits 7 et 3.

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
p0(0)	p1(0)	p2(0)	p3(0)	p0(1)	p1(1)	p2(1)	p3(1)

25.2.3 Mode 0

Ici, c'est un peu le bazar. Chaque octet ne représente que deux pixels, dont la correspondance avec les bits de l'octet est la suivante : Le pixel 0 est codé avec les bits 7,5,3 et 1.

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
p0(0)	p1(0)	p0(2)	p1(2)	p0(1)	p1(1)	p0(3)	p1(3)

L'image suivante vous aidera certainement à y voir plus clair :

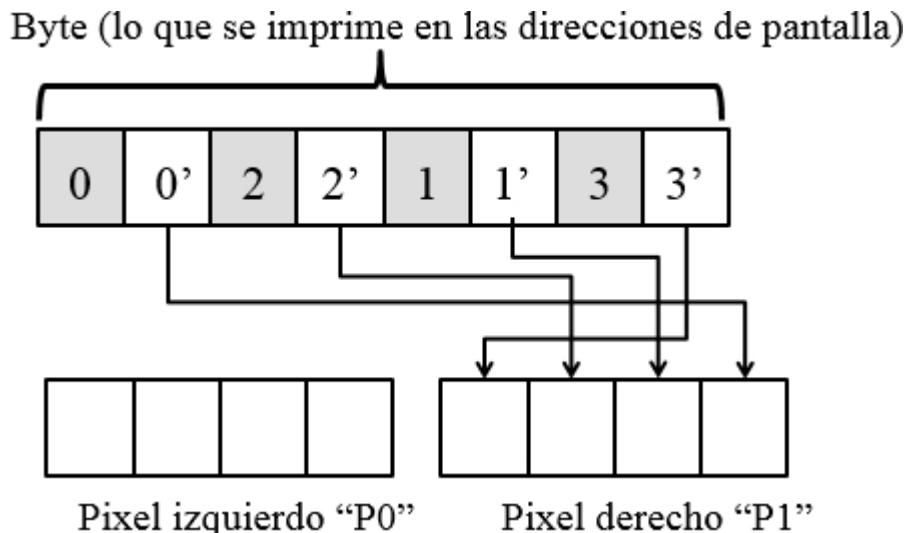


Fig. 125 pixels et bits en mode 0

Je ne peux pas vous dire quelle est la raison obscure pour laquelle la mémoire a été organisée de cette manière, mais j'imagine que la cause se trouve dans le GATE ARRAY, la puce qui traduit ces bits en un signal vidéo. J'imagine que le concepteur a réduit les circuits avec cette conception tordue.

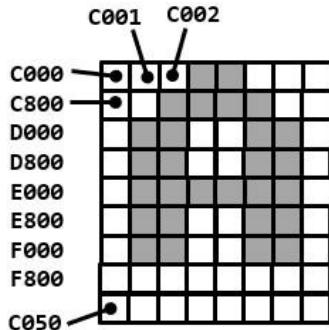
25.2.4 Mémoire d'affichage

Les pixels de l'écran appartenant à une même ligne sont encodés dans des octets également contigus. Cependant, d'une ligne à l'autre, il y a des sauts.

Si nous avançons dans les adresses mémoire, lorsque nous atteignons la fin d'une ligne, nous sautons à une ligne qui se trouve 8 lignes plus loin. Et si nous voulons continuer sur la ligne suivante, nous devons sauter dans les adresses mémoire à 2048 positions.

Le tableau suivant montre la mémoire vidéo. A gauche se trouve la ligne de caractères (de 1 à 25) et pour chaque ligne, l'adresse de départ de chacune des 8 lignes de balayage qui la composent (nommées ROW0 ...ROW7).

CHARACTER LINE	R0W0	R0W1	R0W2	R0W3	R0W4	R0W5	R0W6	R0W7
1	C000	C800	D000	D800	E000	E800	F000	F800
2	C050	C850	D050	D850	E050	E850	F050	F850
3	C0A0	C8A0	D0A0	D8A0	E0A0	E8A0	F0A0	F8A0
4	C0F0	C8F0	D0F0	D8F0	E0F0	E8F0	F0F0	F8F0
5	C140	C940	D140	D940	E140	E940	F140	F940
6	C190	C990	D190	D990	E190	E990	F190	F990
7	C1E0	C9E0	D1E0	D9E0	E1E0	E9E0	F1E0	F9E0
8	C230	CA30	D230	DA30	E230	EA30	F230	FA30
9	C280	CA80	D280	DA80	E280	EA80	F280	FA80
10	C2D0	CAD0	D2D0	DAD0	E2D0	EAD0	F2D0	FAD0
11	C320	CB20	D320	DB20	E320	EB20	F320	FB20
12	C370	CB70	D370	DB70	E370	EB70	F370	FB70
13	C3C0	CBC0	D3C0	DBC0	E3C0	EBC0	F3C0	FBC0
14	C410	CC10	D410	DC10	E410	EC10	F410	FC10
15	C460	CC60	D460	DC60	E460	EC60	F460	FC60
16	C4B0	CCB0	D4B0	DCB0	E4B0	ECB0	F4B0	FCB0
17	C500	CD00	D500	DD00	E500	ED00	F500	FD00
18	C550	CD50	D550	DD50	E550	ED50	F550	FD50
19	C5A0	CDA0	D5A0	DDA0	E5A0	EDA0	F5A0	FDA0
20	C5F0	CDF0	D5F0	DDF0	E5F0	ED50	F550	FD50
21	C640	CE40	D640	DE40	E640	EE40	F640	FE40
22	C690	CE90	D690	DE90	E690	EE90	F690	FE90
23	C6E0	CEE0	D6E0	DEE0	E6E0	EEE0	F6E0	FEE0
24	C730	CF30	D730	DF30	E730	EF30	F730	FF30
25	C780	CF80	D780	DF80	E780	EF80	F780	FF80
spare start	C7D0	CFD0	D7D0	DFD0	E7D0	EFD0	F7D0	FFD0
spare end	C7FF	CFFF	D7FF	DFFF	E7FF	EFFF	F7FF	FFFF



*Direcciones de la
esquina superior
izquierda de la
pantalla*

C000 = comienzo de pantalla
= 49152 , es decir 48KB

FFFF= fin de pantalla
= 65535

La pantalla mide:
65535 – 49152 = 16384 =16KB

Fig. 126 Carte mémoire de l'écran

L'écran de l'Amstrad est composé de 200 lignes de 80 octets de large chacune, de sorte que la mémoire de l'écran affiché est de $200 \times 80 = 16\ 000$ octets. Cependant, la mémoire vidéo est de 16384 octets. Il y a 384 octets "cachés" dans 8 segments de 48 octets chacun, qui ne sont pas affichés à l'écran, bien qu'ils fassent partie de la mémoire vidéo. Ces 8 segments sont appelés "spares" dans le tableau ci-dessus. Chaque segment a une longueur de 48 octets parce que, comme je l'ai dit plus haut, pour passer d'une ligne à l'autre, il faut ajouter 2048 octets, mais en réalité les 25 lignes de mémoire contiguës qui les séparent n'occupent que 25×80 octets =2000 octets.

De &C7D0 à C7FF tous les deux inclus De &CFD0 à CFFF tous les deux inclus De &D7D0 à D7FF tous les deux inclus De &DFD0 à DFFF tous les deux inclus De &E7D0 à E7FF tous les deux inclus De &EFD0 à EFFF tous les deux inclus De &F7D0 à F7FF tous les deux inclus De &FFD0 à FFFF tous les deux inclus De &FFD0 à FFFF tous les deux inclus De &FFD0 à FFFF tous les deux inclus

Vous pouvez le vérifier en faisant un POKE sur ces adresses mémoire, et vous verrez que vous ne modifierez pas le contenu de l'écran.

Il est tentant de penser à utiliser ces zones "cachées" de la mémoire pour stocker de petites

routines d'assemblage ou des variables. Mais c'est dangereux car un

MODE exécutée à partir de BASIC efface complètement ces segments de mémoire, donc si vous l'utilisez, vous devez en être conscient. Dans la bibliothèque 8BP, ces segments sont utilisés pour stocker les variables locales de certaines fonctions, dont la valeur peut être effacée en toute sécurité.

25.3 Calcul d'une adresse d'écran

Si vous souhaitez connaître l'adresse mémoire à laquelle correspondent des coordonnées 8BP spécifiques, vous devez effectuer l'opération suivante

$$\text{Dir} = \&C000 + \text{INT}(y/8)*80 + (y \bmod 8)*2048 + x$$

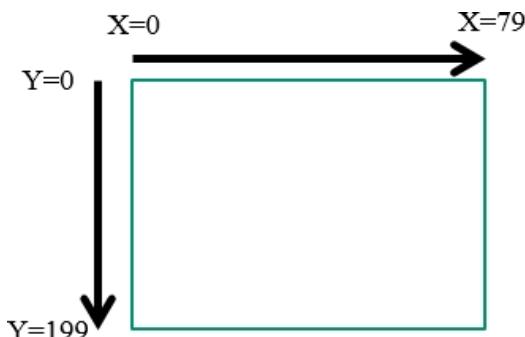


Fig. 127 Coordonnées de l'écran en 8BP

C'est très utile si, par exemple, vous voulez utiliser PEEK pour savoir s'il y a un certain élément ou une certaine couleur dans un octet particulier de l'écran et l'utiliser comme mécanisme de détection des collisions. Cette technique est utilisée dans le jeu vidéo "3D-Racing One".

Si vous voulez connaître la direction de certaines coordonnées graphiques (celles utilisées par la commande BASIC PLOT), vous devez d'abord faire $y2=(200-y/2)$, $x2=x/8$

$$\text{Dir} = \&C000 + \text{INT}(y2/8)*80 + (y2 \bmod 8)*2048 + x2$$

25.4 Balayages d'écran

L'Amstrad génère 50 images par seconde. Cela signifie que toutes les 20 ms environ, un nouveau balayage de l'écran doit être produit.

On pourrait penser que le balayage de l'écran, qui consiste à peindre l'écran, consomme une fraction de ces 20 ms, mais ce n'est pas le cas. La peinture de l'écran prend à l'Amstrad la totalité des 20 ms, de sorte que même si vous synchronisez l'impression de votre sprite avec le balayage de l'écran, il est très probable qu'il vous rattrape, ce qui donne lieu à deux effets bien connus :

- **Le scintillement** se produit lorsque vous effacez le sprite avant de l'imprimer dans sa nouvelle position. Pour éviter cela, il existe une solution très simple : ne pas l'effacer. Il suffit de faire en sorte que le sprite efface sa propre trace, en laissant une bordure sur le sprite pour remplir cette fonction. Le sprite est plus grand mais il ne scintillera pas, même s'il est pris au milieu, car il ne disparaît pas.

- **Déchirure** : se produit lorsque nous sommes pris par le balayage au milieu du sprite. La moitié est imprimée avec la nouvelle position (tête et tronc) et l'autre moitié ne l'est pas.

donne du temps (les jambes). Le sprite est alors imprimé "de travers", bien qu'il soit corrigé dans l'image suivante, mais pendant un instant, on a l'impression qu'il est déformé ou cassé. La déchirure est un mauvais effet, mais beaucoup plus acceptable que le scintillement. La solution parfaite est de contrôler chaque milliseconde où se trouve le scintillement afin d'imprimer chaque sprite sans qu'il ne nous rattrape.

Une recommandation typique est d'imprimer les sprites de bas en haut, pour minimiser ces effets. De cette façon, il n'est possible d'obtenir le scan qu'une seule fois sur l'un des sprites, alors qu'en imprimant de haut en bas, vous pouvez obtenir le scan sur plusieurs sprites car les deux (CPU et rayon cathodique) travaillent dans la même direction. Malheureusement, la chose la plus intéressante à faire est de trier de haut en bas pour donner un effet de profondeur aux sprites (utile dans certains jeux comme "Golden axe", "Double dragon", "Renegade", etc.)

Les temps consommés par l'écran sont les suivants. Notez qu'à partir de l'interruption du balayage, vous avez 3,5 ms pour peindre sans qu'il soit possible de vous faire prendre. Mais dans ce laps de temps, vous pouvez imprimer au maximum 2 petits sprites.

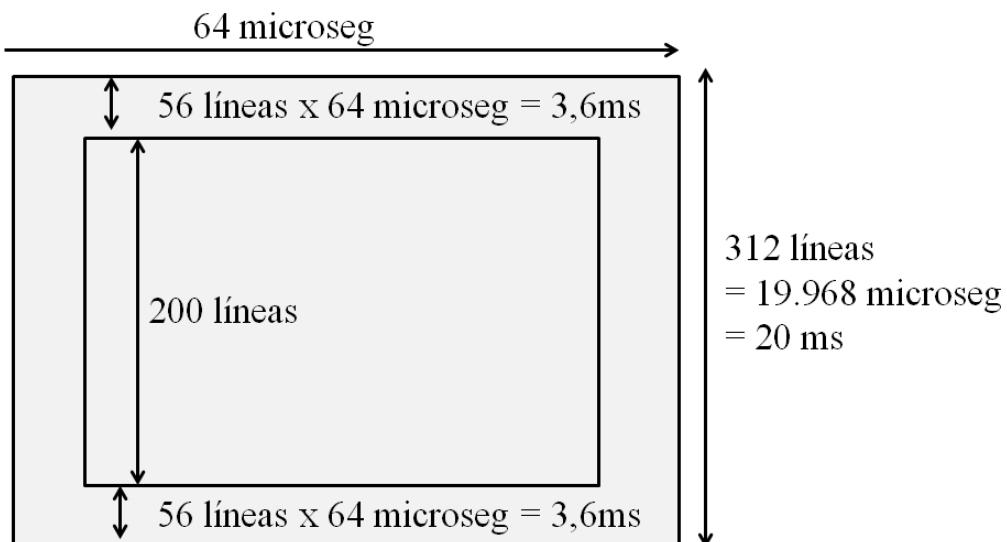


Fig. 128 fois dans un balayage d'écran

25.5 Comment créer un écran de chargement pour votre jeu

Il y a plusieurs façons de procéder. Une façon très simple est de construire un graphique avec le programme SPEDIT modifié pour vous permettre de peindre sur tout l'écran sans afficher les menus et, à la fin, d'appuyer sur une touche pour lancer une commande SAUVEGARDER comme celle-ci

SAVE "mipantalla.bin", b, &C000, 16384

Comme vous pouvez le voir, la commande enregistre 16 Ko à partir de l'adresse de départ de l'écran, qui est &C000.

La façon de le charger serait la suivante

LOAD " mipantalla.bin", &C000

Si vous n'utilisez pas la palette par défaut, vous devez d'abord exécuter les commandes INK correspondant à la palette que vous avez utilisée, avant de charger l'écran. Lors du

chargement de l'écran, vous verrez que l'écran se dessine lentement sur l'écran au fur et à mesure qu'il se charge, puisque c'est précisément là que vous le chargez, dans la mémoire vidéo.

Une autre méthode consiste à générer une mise en page bien travaillée et à l'enregistrer à l'aide de la commande SAUVEGARDER ci-dessus. Le but est de faire un dessin et, comme vous pouvez le voir, il y a de nombreuses façons de le faire.

Enfin, vous pouvez utiliser un outil comme ConvImgCPC (il en existe d'autres), un convertisseur/éditeur d'images qui fonctionne sous Windows. Cet outil vous permet de transformer n'importe quelle image (qui peut être un scan d'un de vos dessins) en un fichier binaire (avec l'extension .scr) adapté au CPC. Cet outil vous permet également de l'édition pixel par pixel et de la retoucher jusqu'à ce qu'elle soit parfaite. Vous pouvez même l'édition à partir de zéro, sans rien scanner. À mon avis, c'est la meilleure option.

Pour placer ce fichier sur un disque (dans un fichier .dsk), vous devez utiliser CPCDiskXP, un autre outil qui vous permet de placer des fichiers dans des fichiers .dsk.

Une fois dans le .dsk, vous pouvez le charger avec LOAD "mipantalla.bin",&C000 Cependant, les couleurs ne seront pas correctes car ConvImgCPC ajuste la palette pour qu'elle soit aussi proche que possible des couleurs d'origine. Pour les voir correctement, vous devez exécuter la routine où ConvImg place la routine de changement de palette, c'est-à-dire CALL &C7D0.

Cette routine est "cachée" dans le premier des 8 segments cachés de la mémoire vidéo, de sorte que l'image .scr n'occupe pas plus de place parce qu'elle contient cette routine. Le problème est que tant que vous ne chargez pas l'écran, vous ne pouvez pas l'exécuter et donc vous verrez comment il charge l'image avec de mauvaises couleurs et à la fin vous pourrez invoquer cet appel, en changeant les couleurs. Ce que vous pouvez faire, c'est préparer un fichier de palette spécial. Voici ce qu'il faut faire :

Charger "image.scr", &c000

Enregistrer "palette.bin", b, &c7d0, 48, &c7d0

Vous avez maintenant un fichier de 48 octets contenant la palette. Dans votre chargeur de jeu, vous devez faire ceci :

Charger "!palette.bin"

Appeler &c7d0

Charger " !image.scr", &c000

Je vous recommande de placer simplement quelques commandes INK avant la commande LOAD qui charge l'image.

Charger "image.scr", &c000

Et juste après cela, avant d'utiliser 8BP pour imprimer des sprites, etc. il est pratique de supprimer le segment caché où ConvImg quitte la routine, parce que c'est un espace que 8BP utilise pour les variables et si elles ne sont pas initialement mises à zéro, elles peuvent interférer

for i = &c000+2000 to &c000+2000+48 : poke i,0:NEXT

en bref :

10 <plusieurs commandes INK>

20 Charger " !image.scr", &c000

30 for i = &c000+2000 to &c000+2000+48 : poke i,0:NEXT

Pour laisser la palette à ses valeurs par défaut, utilisez CALL &BC02, qui est une routine firmware.

Et pour sauvegarder l'écran sur une cassette ?

Nous avons vu comment le faire sur disque mais CPCDiskXP ne laisse pas le fichier .scr sur la bande, nous devons donc faire quelque chose comme ceci :

```
|DISC  
MEMOIRE 15999  
LOAD "image.scr", 16000  
|TAPE  
VITESSE D'ECRITURE 1  
SAVE "imagen.scr",b,16000,16384
```

Et lorsque nous le chargeons, nous le faisons de la même manière que sur le disque :

Charger " !image.scr", &c000

26 ANNEXE II : La palette

Le tableau suivant présente la palette AMSTRAD. Dans chaque couleur et entre parenthèses se trouve le numéro d'encre attribué à cette couleur dans la palette par défaut. Les 27 couleurs sont les suivantes

0 - Negro (5)	1 - Azul (0,14)	2 - Azul claro (6)	3 - Rojo	4 - Magenta	5 - Violeta	6 - Rojo claro (3)	7 - Púrpura	8 - Magenta claro (7)
9 - Verde	10 - Cyan (8)	11 - Azul cielo (15)	12 - Amarillo (9)	13 - Gris	14 - Azul pálido (10)	15 - Anaranjado	16 - Rosa (11)	17 - Magenta pálido
18 - Verde claro (12)	19 - Verde mar	20 - Cyan claro (2)	21 - Verde lima	22 - Verde pálido (13)	23 - Cyan pálido	24 - Amarillo claro (1)	25 - Amarillo pálido	26 - Blanco (4)

Les valeurs par défaut de la palette dans chaque mode sont les suivantes

Modo 2:

0: Azul (paleta 1)	1: Amarillo intenso (paleta 24)
--------------------	---------------------------------

Modo 1:

0: Azul (paleta 1)	1: Amarillo intenso (paleta 24)
2: Cyan claro (paleta 20)	3: Rojo claro (paleta 6)

Modo 0:

0: Azul (paleta 1)	1: Amarillo intenso (paleta 24)	2: Cyan claro (paleta 20)	3: Rojo claro (paleta 6)
4: Blanco (paleta 26)	5: Negro (paleta 0)	6: Azul claro (paleta 2)	7: Magenta claro (paleta 8)
8: Cyan (paleta 10)	9: Amarillo (paleta 12)	10: Azul pálido (paleta 14)	11: Rosa (paleta 16)
12: Verde claro (paleta 18)	13: Verde pálido (paleta 22)	14: Parpadeo Azul/Amarillo	15: Parpadeo azul cielo/Rosa

Les valeurs de la palette dans chaque mode sont gérées par la commande INK, voir le manuel de référence de l'Amstrad BASIC pour plus d'informations. Par exemple, pour mettre l'encre zéro en rouge, on consulte la palette du 27, on voit que le rouge est la sixième couleur et on écrit

INK 0,6

Et nous avons déjà configuré l'encre zéro pour qu'elle soit rouge. Comme vous pouvez le constater, une encre n'est pas une couleur spécifique, mais peut être configurée pour être de la couleur de votre choix.

La raison pour laquelle la palette Amstrad est si bonne est que les 27 couleurs offrent de nombreuses possibilités, même si seulement 16 peuvent être utilisées à la fois. Les couleurs sont classées par ordre de luminosité.

Si nous représentons les couleurs de l'Amstrad dans l'échelle RVB 24 bits (8 bits par composante), nous constatons qu'il y a 3 valeurs de rouge, 3 de vert et 3 de bleu, (ces valeurs sont 0,127 et 255), soit le nombre de combinaisons $3 \times 3 \times 3 = 27$. La couleur grise se trouve au centre de la palette, là où les valeurs de R, V et B sont égales.

(R=127,G=127,B=127). Les 3 composantes sont également les mêmes en blanc (R=255,G=255,B=255) et en noir (R=0,G=0,B=0).

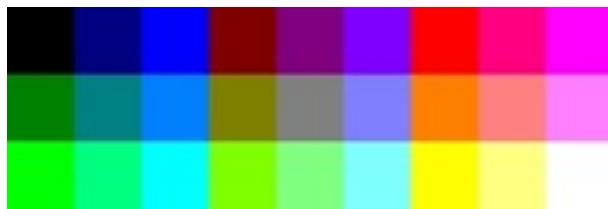


Fig. 129 Palette Amstrad

Une palette de 27 couleurs signifie que, bien que nous ne puissions en choisir que 16, il y a toujours des couleurs parmi lesquelles choisir, ce qui nous permet de créer des fondus et des mélanges. Cependant, d'autres ordinateurs de l'époque, comme le C64 (une machine formidable), avaient 16 couleurs sur une palette de 16. Le C64 avait 3 nuances de gris dans une palette aussi réduite, ce qui, bien que cela ait été critiqué, n'est pas une mauvaise chose car, comme il ne s'agit pas de couleurs très saturées, elles se combinent bien avec le gris, et elles se combinent également bien entre elles, c'est-à-dire qu'il s'agit de 16 couleurs qui sont "proches" les unes des autres.

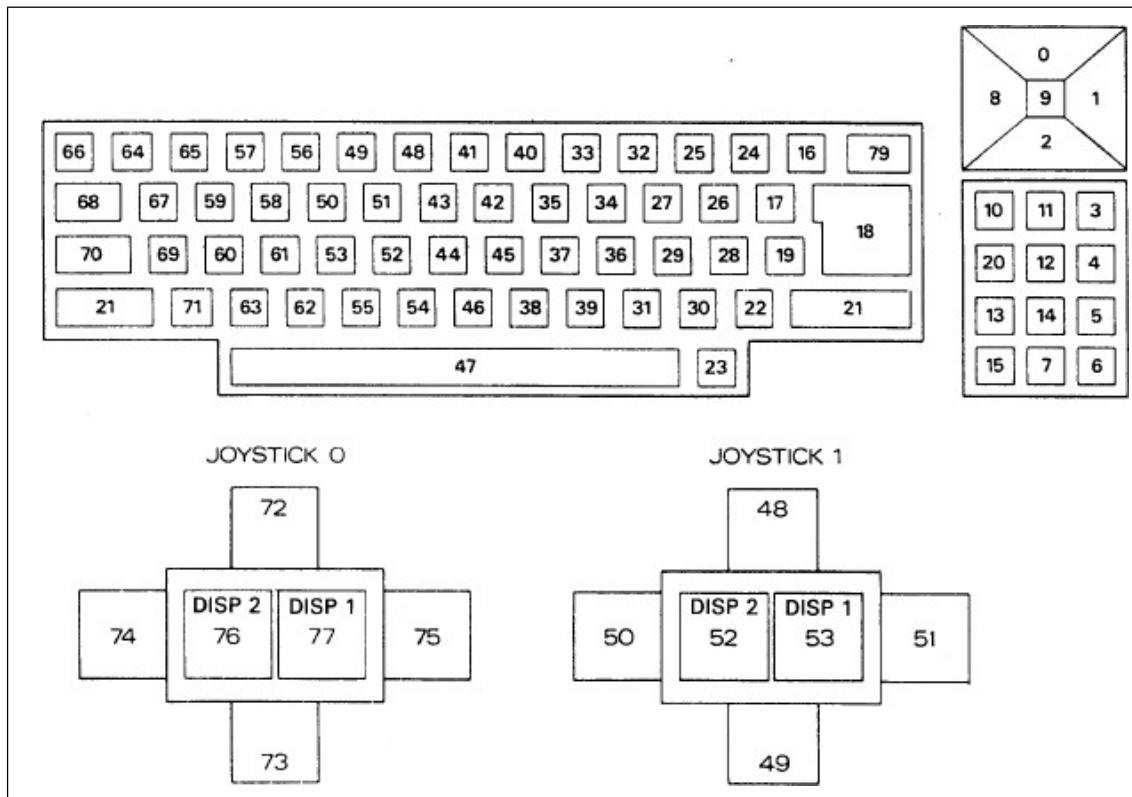


Fig. 130 Palette C64

En bref, sur Amstrad, vous pouvez choisir parmi un plus grand nombre de couleurs et donc toujours trouver la bonne couleur pour estomper, ombrer ou simplement trouver le ton de couleur dont vous avez besoin. Le grand succès d'Amstrad a été de créer une palette de 27 couleurs, bien que vous ne puissiez utiliser que 16 couleurs à la fois. Vous pouvez également choisir 16 couleurs qui ne s'accordent pas bien entre elles et obtenir des graphismes de très mauvais goût (ce qui est impossible sur le C64, car vous ne pouvez pas choisir).

La palette de 27 permettait à de nombreux graphiques de chargements Amstrad d'être de véritables œuvres d'art.

27 ANNEXE III : Codes INKEY



Chaque fois que vous voulez lire le clavier, essayez d'abord de vider le tampon de lecture des dernières frappes. Il est très fréquent que dans un écran où l'on demande le nom de l'utilisateur qui a obtenu un score élevé, les dernières frappes du jeu (mouvements et tirs) "se faufilent", en affichant des choses comme "OPPPOQQAAA" dans la commande INPUT. Pour éviter cela, exécutez quelque chose comme :

**10 B\$=INKEY\$: si B\$<>"" ALORS 10
20 INPUT "name :";\$name**

En plus de cette recommandation, rappelez-vous que la manière la plus rapide de traiter le clavier est la suivante :

**10 IF INKEY(keycode) THEN 30 : REM saute à 30 si la touche n'est pas enfoulée
20 <instructions en cas d'appui sur le clavier> 30**

29 ANNEXE IV : Tableau ASCII de l'AMSTRAD CPC

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	□	□	P	^	p	.	^	α	/	—	Ω	↑				
1	Γ	Φ	!	1	A	Q	a	q	■	!	β	^	I	Ω	↓	
2	└	Ω	"	2	B	R	b	r	■	-	„	„	—	Φ	←	
3	└	Ω	#	3	C	S	c	s	■	£	€	„		♦	→	
4	◊	Φ	*	4	D	T	d	t	■	,	®	€	^	♥	▲	
5	⊗	×	5	E	U	e	u	■		π	Θ)	▼	♣	▼	
6	√	Π	&	6	F	V	f	v	■	r	§	λ	▼	△	○	▶
7	Ω	—	'	7	G	W	g	w	■	†	‘	μ	◀	◀	●	◀
8	←	X	(8	H	X	h	x	■	-	14	π	✓	❖	□	✗
9	→	†)	9	I	Y	i	y	■	12	σ	ς	▀	▀	▀	▀
A	↓	Ω	*	:	J	Z	j	z	■	-	34	♂	○	❖	♂	✗
B	↑	Θ	+	;	K	[k	€	■	±	♂	✗	▀	♀	✗	✗
C	Ψ	¶	,	<	L	\	l	l	■	¬	÷	X	/	▀	J	✗
D	←	▀	-	=	M]	m	»	■	†	¬	ω	＼	▀	▀	✗
E	Ω	▀	.	>	N	†	n	~	■	†	♂	Σ	☒	☒	☒	☒
F	Θ	¶	/	?	O	_	o	☒	■	+	i	Ω	☒	☒	☒	☒

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240
1	1	17	33	49	65	81	97	113	129	145	161	177	193	209	225	241
2	2	18	34	50	66	82	98	114	130	146	162	178	194	210	226	242
3	3	19	35	51	67	83	99	115	131	147	163	179	195	211	227	243
4	4	20	36	52	68	84	100	116	132	148	164	180	196	212	228	244
5	5	21	37	53	69	85	101	117	133	149	165	181	197	213	229	245
6	6	22	38	54	70	86	102	118	134	150	166	182	198	214	230	246
7	7	23	39	55	71	87	103	119	135	151	167	183	199	215	231	247
8	8	24	40	56	72	88	104	120	136	152	168	184	200	216	232	248
9	9	25	41	57	73	89	105	121	137	153	169	185	201	217	233	249
A	10	26	42	58	74	90	106	122	138	154	170	186	202	218	234	250
B	11	27	43	59	75	91	107	123	139	155	171	187	203	219	235	251
C	12	28	44	60	76	92	108	124	140	156	172	188	204	220	236	252
D	13	29	45	61	77	93	109	125	141	157	173	189	205	221	237	253
E	14	30	46	62	78	94	110	126	142	158	174	190	206	222	238	254
F	15	31	47	63	79	95	111	127	143	159	175	191	207	223	239	255

30 ANNEXE V : quelques effets sonores

Tout d'abord, il faut savoir que l'Amstrad possède 3 canaux et que leurs identifiants sont 1, 2 et 4. Pour faire sonner deux canaux ou les 3 en même temps, il suffit de les additionner.

Utilisation de la commande SOUND :

SOUND canal, hauteur, durée, volume, hauteur de l'enveloppe, volume de l'enveloppe, bruit

IMPORTANT : Le volume va de 0 à 7 sur le CPC464 et de 0 à 15 sur le 6128. Sur le CPC464, les valeurs 8..15 peuvent être utilisées mais elles sont une répétition des valeurs 0..7 (c'est-à-dire que 8 signifie volume 0). Il s'agit d'une différence fondamentale entre les deux modèles. Par conséquent, un volume 10 sur le CPC6128 est élevé alors que sur le 464 il est très bas.

Le paramètre de bruit est compris entre 0 et 31

Exemples :

SOUND 1,2000,10,7 : son canal 1

SOUND 1+2,2000,10,7 : les canaux 1 et 2 sont en train de sonner.

Voici quelques exemples à utiliser directement dans vos programmes ou pour vous inspirer dans la création d'autres sons.

Collectez un diamant ou une pièce ENT 1,10,-100,3 : son 1,638,30,30,15,15,0,1	vous avez été frappé par une pierre ou un projectile ENT 1.10, 100.3 : son 1,638,30,30,15,15,0,1
Boing ENV 1,1,15,1,15,-1,1 : SON 1,638,0,0,0,1	Boing 2 ENT 2,20,-125,1 : SON 1,1500,10,12,12,,2
La mort ENT 3,100,5,3 : SON 1,142,100,15,0,3	
Explosion SON 7,1000,20,20,15,,,15	Explosion 2 ENV 1,11,-1,25:ENT 1,9,49,5,9,-10,15 SON 3,145,300,12,1,1,1,12
Tir ENT -5,7,10,1,1,7,-10,1 : SON 1,25,20,12,12,,5	

31 ANNEXE VI : Routines de microprogrammation intéressantes

Dans cette section, je vais inclure quelques routines de microprogrammes qui peuvent être invoquées à partir de BASIC et qui peuvent être intéressantes dans vos programmes.

CALL 0 : réinitialise l'ordinateur

CALL &bc02 : initialise la palette à sa valeur par défaut. Il est conseillé de l'appeler au début de votre programme au cas où elle serait modifiée.

CALL &bd19 : synchronisation avec le balayage de l'écran. Si vous manipulez très peu de sprites, vous pouvez obtenir un mouvement plus fluide, mais sachez que cette instruction ralentira considérablement votre programme.

CALL &bb48 : désactive le mécanisme BREAK, empêchant le programme de s'arrêter s'il est en cours d'exécution.

CALL &bd21 , &bd22, &bd23, &bd24, &bd25 : produit un effet de clignotement sur l'écran

Pour réinitialiser la minuterie de l'AMSTRAD :

Sur un 6128

POKE &b8b4,0 : POKE &b8b5,0 : POKE &b8b6,0 : POKE &b8b7,0

Dans une 464

POKE &b187,0 : POKE &b188,0 : POKE &b189,0 : POKE &b18a,0

Pour différencier la machine sur laquelle se trouve votre programme, vous devez désactiver la musique et consulter une adresse avec PEEK.

**| MUSIC : Si peek(&39)=57 alors modèle=464 sinon modèle=6128
Si modèle=464 alors ...**

CALL &bca7 : arrêter la sonnerie de tout son en cours de diffusion

La commande GRAPHICS PAPER existe dans le CPC6128 mais pas dans le CPC464. Cependant, il existe un moyen de l'obtenir, en utilisant le **CALL &BBE4** et autant de paramètres avec la valeur 1 comme la couleur d'encre que nous voulons, par exemple :

CALL &BBE4,1,1: identique à "GRAPHICS PAPER 2" mais fonctionne sur cpc464

CALL &BB18 : attend que vous appuyiez sur une touche

32 ANNEXE VII : Tableau des attributs des sprites

Le tableau suivant contient les adresses mémoire où sont stockés les attributs de chaque sprite, ainsi que la longueur en octets de chacun d'entre eux.

	1byte	2 bytes	2 bytes	1byte	1byte	1byte	1byte	2 bytes	1byte
sprite	status	coordy	coordx	vy	vx	seq	frame	imagen	ruta
0	27000	27001	27003	27005	27006	27007	27008	27009	27015
1	27016	27017	27019	27021	27022	27023	27024	27025	27031
2	27032	27033	27035	27037	27038	27039	27040	27041	27047
3	27048	27049	27051	27053	27054	27055	27056	27057	27063
4	27064	27065	27067	27069	27070	27071	27072	27073	27079
5	27080	27081	27083	27085	27086	27087	27088	27089	27095
6	27096	27097	27099	27101	27102	27103	27104	27105	27111
7	27112	27113	27115	27117	27118	27119	27120	27121	27127
8	27128	27129	27131	27133	27134	27135	27136	27137	27143
9	27144	27145	27147	27149	27150	27151	27152	27153	27159
10	27160	27161	27163	27165	27166	27167	27168	27169	27175
11	27176	27177	27179	27181	27182	27183	27184	27185	27191
12	27192	27193	27195	27197	27198	27199	27200	27201	27207
13	27208	27209	27211	27213	27214	27215	27216	27217	27223
14	27224	27225	27227	27229	27230	27231	27232	27233	27239
15	27240	27241	27243	27245	27246	27247	27248	27249	27255
16	27256	27257	27259	27261	27262	27263	27264	27265	27271
17	27272	27273	27275	27277	27278	27279	27280	27281	27287
18	27288	27289	27291	27293	27294	27295	27296	27297	27303
19	27304	27305	27307	27309	27310	27311	27312	27313	27319
20	27320	27321	27323	27325	27326	27327	27328	27329	27335
21	27336	27337	27339	27341	27342	27343	27344	27345	27351
22	27352	27353	27355	27357	27358	27359	27360	27361	27367
23	27368	27369	27371	27373	27374	27375	27376	27377	27383
24	27384	27385	27387	27389	27390	27391	27392	27393	27399
25	27400	27401	27403	27405	27406	27407	27408	27409	27415
26	27416	27417	27419	27421	27422	27423	27424	27425	27431
27	27432	27433	27435	27437	27438	27439	27440	27441	27447
28	27448	27449	27451	27453	27454	27455	27456	27457	27463
29	27464	27465	27467	27469	27470	27471	27472	27473	27479
30	27480	27481	27483	27485	27486	27487	27488	27489	27495
31	27496	27497	27499	27501	27502	27503	27504	27505	27511

Tableau 7 Adresses des attributs des sprites

7 ROUTEALL lo ruta	6 Sobre- escritura	5 COLSPALL collider	4 MOVERALL lo mueve	3 AUTOALL lo mueve	2 ANIMALL lo anima	1 COLSP collided	0 PRINTSPALL lo imprime
--------------------------	--------------------------	---------------------------	---------------------------	--------------------------	--------------------------	------------------------	-------------------------------

Tableau 8 Drapeaux dans l'octet de statut

33 ANNEXE VIII : Carte mémoire du 8BP

```
AMSTRAD CPC464 MEMORY MAP de 8BP
;
; &FFFF +-----+
; | affichage + 8 segments cachés de 48 octets chacun
; &C000 +-----+
; | système (symboles redéfinissables, pointeur de pile,
etc.) -----
; 42619 +
; | -banque-de 40 étoiles (de 42540 à 42619 = 80bytes)
; 42540 +
; | Carte de disposition des caractères (25x20 =500 octets)
; | et carte du monde (jusqu'à 82 éléments peuvent tenir dans 500
octets)
; | -Les-deux-sont stockés dans la même zone de mémoire.
; | sprites (jusqu'à 185,8Ko d'images)
; 42040 dessins avez 8440 octets s'il n'y a pas de séquences et
; | pas de routes)
; +-----Les images de l'alphabet sont également stockées ici.
; | définitions d'itinéraires (d'une longueur variable)
; +-----+
; | Séquences d'animation de 8 images (16 octets chacune)
; | les séquences d'animation et les groupes de séquences
; 33600 d'animation (macro-séquences)
; | chansons
; | (1500 Bytes pour la musique éditée avec WYZtracker
; 32100 +-----2.0.1.0)
; | Routines 8BP(8100 octets ou 7100 octets)
; | voici toutes les routines et la table des sprites
; | inclut le lecteur de musique "wyz" 2.0.1.0
; 25000 +-----+
; |
; | VOTRE LISTE DE BASE ou C
; | 24 Ko, 24,8 Ko ou jusqu'à 25 Ko libres pour BASIC ou C
; | en fonction de l'option d'assemblage que vous utilisez
; | pour 8BP
; |
; 0 +-----+
```


34 ANNEXE IX : Commandes disponibles 8BP

Liste des commandes disponibles par ordre alphabétique :

3D, <flag>, #, offsety 3D, 0	Active le mode de projection pseudo 3D.
ANIMA, #	Change l'image d'un sprite en fonction de sa séquence
ANIMALL	Modifie l'image des sprites dont le drapeau d'animation est activé (il n'est pas nécessaire de l'invoquer, un drapeau dans l'instruction PRINTSPALL suffit à l'invoquer).
AUTO, #	Déplacement automatique d'un sprite en fonction de ses Vy,Vx
AUTOALL, <flag routed>, <flag routed>, <flag routed>, <flag routed>, <flag routed>.	Mouvement de tous les sprites dont le drapeau de mouvement automatique est actif
COLAY, threshold_ascii, @collision, # COLAY, @collision, # COLAY, # COLAY	Déetecte les collisions avec la disposition et renvoie 1 s'il y a collision. Accepte un nombre variable de paramètres (toujours dans le même ordre) de 4 à aucun.
COLSP, #, @collided%, @collided%, @COLSP, #, @collided%, @collided%. COLSP, 32, ini, end COLSP, 33 ans, @collided% COLSP, 34, dy, dx COLSP, #	Renvoie le premier sprite avec lequel # entre en collision. La commande peut être configurée avec les codes 32, 33 et 34.
COLSPALL, @who%, @who%, @withwho%. COLSPALL, collisionneur COLSPALL	Retourne qui est entré en collision (collider) et avec qui il est entré en collision (collided).
LAYOUT, y, x, @string\$, @string\$, @string\$, @string\$, @string\$.	Imprime une bande d'image 8x8 et remplit la carte.
LOCATESP, #, y, x	Modifie les coordonnées d'un sprite (sans l'imprimer)
MAP2SP, y, x MAP2SP, statut	Crée des sprites pour peindre le monde dans les jeux à défilement. Les sprites sont créés avec state = status
MOVER, #, dy, dx	mouvement relatif d'un seul sprite
dy,dx, dy,dx, dy,dx GLOBAL	Mouvement relatif de tous les sprites dont le drapeau de mouvement relatif est actif
MUSIQUE, C, drapeau, chanson, vitesse MUSIQUE, drapeau, chanson, vitesse MUSIQUE	Une mélodie commence à être jouée. Le canal C peut être désactivé pour l'utilisation d'effets FX si vous le souhaitez. Aucun paramètre ne permet d'arrêter la sonnerie
PEEK, dir, @variable%	Lit une donnée de 16 bits (peut être négative)
POKE, dir, value	entrer une donnée de 16 bits (qui peut être négative)
PRINTAT, flag, y, x, @string	Imprime une chaîne de "mini-caractères" redéfinissables.
PRINTSP, #, y, x PRINTSP, # PRINTSP,32, bits	imprime un seul sprite (# est son numéro) indépendamment de l'octet de statut. Si 32 est spécifié, les bits d'arrière-plan sont activés
Le projet de loi a été adopté à l'unanimité par l'Assemblée nationale. L'impression de l'étiquette, le mode d'emploi, le mode d'emploi PRINTSPALL	Imprime tous les sprites dont le drapeau d'impression est actif. S'il est invoqué avec un seul paramètre, le mode de commande est défini.
RINK,tini,color1,color1,color2,...,colorN RINK, sauter	Fait tourner un ensemble d'encre selon un modèle définissable composé d'un nombre quelconque d'encre.

ROUTEESP, #, étapes	Vous fait passer par N étapes de la route des sprites en une seule fois.
ROUTEALL	Modifier la vitesse du sprite avec le drapeau de chemin (pas besoin de l'invoquer, il suffit de mettre le drapeau dans AUTOALL).
SETLIMITS, xmin, xmax, ymin, ymax	Définit la fenêtre de jeu dans laquelle le découpage est effectué.
SETUPSP, #, nombre_de_paramètres, valeur	Modifie un paramètre d'un sprite. Si le paramètre 5 est spécifié, Vx peut être fourni en option.
SETUPSP, #, 5, Vy, Vx	
STARS, initstar, num, colour, dy, dx	Parchemin d'un ensemble d'étoiles
UMAP,adr_ini, adr_end, yini, yfin, xini, xfin	Met à jour les éléments de la carte du monde avec un sous-ensemble d'éléments d'une carte plus grande.

35 ANNEXE X : Options d'assemblage du 8BP

Depuis la version V42, la bibliothèque 8BP dispose de plusieurs options d'assemblage qui vous permettent de choisir les capacités que vous souhaitez pour votre jeu et d'avoir ainsi plus de mémoire disponible pour votre listing de jeu.

L'option d'assemblage doit être spécifiée dans le fichier **Make_all_mygame.asm**, qui comporte une ligne spécifique pour assigner la valeur du paramètre "**ASSEMBLING_OPTION**".

Option	Description de l'option	Exemple de jeu typique
0	<p>Vous pouvez faire n'importe quel jeu Toutes les commandes sont disponibles Vous devez utiliser MEMOIRE 23499</p> <p>Pour sauvegarder la bibliothèque + les graphiques + la musique :</p> <p>SAVE "8BP0.bin",b,23500,19119</p>	n'importe qui
1	<p>jeux de labyrinthe ou de passage d'écran Vous devez utiliser la MEMOIRE 24999</p> <p>Non disponible dans ce mode :</p> <p> MAP2SP, UMAP, 3D</p> <p>Pour sauvegarder la bibliothèque + les graphiques + la musique :</p> <p>SAVE "8BP1.bin",b,25000,17619</p>	
	<p>Pour les jeux à défilement</p> <p>Vous devez utiliser MEMOIRE 24799</p> <p>Non disponible dans ce mode :</p> <p> LAYOUT, COLAY, 3D</p> <p>Pour sauvegarder la bibliothèque + les graphiques + la musique :</p> <p>SAVE "8BP2.bin", b,24800,17819</p>	
	<p>Pour les jeux en pseudo 3D, vous devez utiliser MEMOIRE 23999</p> <p>Non disponible dans ce mode :</p> <p> LAYOUT, COLAY</p> <p>Pour sauvegarder la bibliothèque + les graphiques + la musique :</p> <p>SAVE "8BP3.bin", b,24000,18619</p>	

36 ANNEXE XI : Correspondance RSX/CALL

Liste des commandes disponibles par ordre alphabétique et leur adresse associée pour utiliser l'invocation CALL &XXXX lorsque cela est nécessaire pour augmenter la vitesse. Je n'ai donné que quelques exemples car l'utilisation est identique à celle de la commande RSX, sauf que vous devez remplacer la commande par CALL <adresse>.

IMPORTANT : Cette liste d'adresses peut varier d'une version à l'autre du 8BP. Assurez-vous que vous utilisez la dernière version du 8BP si vous comptez utiliser les adresses de ce tableau.

COMMANDÉ	ADRESSE	EXAMPLE
3D	&6BDE	
ANIMA	&6BB7	
ANIMALL	&7479	
AUTO	&6BC9	
AUTOALL	&6B9C	APPELER &6B9C,1
COLAY	&6BA8	
COLSP	&6BBA	
COLSPALL	&6B99	
LAYOUT	&6BD5	
LOCATESP	&6BAE	
MAP2SP	&6BA2	
MOVER	&6BC0	
GLOBAL	&6B9F	
MUSIQUE	&6BD8	CALL &6BD8,0,0,0,0,6
PEEK	&6BB1	CALL &6BB1,dir,@var
POKE	&6BB4	
PRINTAT	&6BC6	CALL &6BC6,0,y,x,@c\$ CALL &6BC6,0,y,x,@c\$
PRINTSP	&6BC3	APPELER &6BC3,31
PRINTSPALL	&6B96	CALL &62A6,0,0,0,0,0,0
RINK	&6BBD	
ROUTESP	&6BCC	
ROUTEALL	&6BD2	
SETLIMITS	&6BDB	
SETUPSP	&6BAB	
LES ÉTOILES DE L'EUROPE	&6BA5	
UMAP	&6BCF	

37 ANNEXE XII : Fonctions 8BP en C

RSX	C prototype
3D, 0 3D, <flag>, #, offsety	void _8BP_3D_1(int flag) ; void _8BP_3D_3(int flag, int sp_fin, int offsety) ;
ANIMA, #	void _8BP_anima_1(int sp) ;
ANIMALL	void _8BP_animall() ;
AUTO, #	void _8BP_auto_1(int sp) ;
AUTOALL, <flag routed>, <flag routed>, <flag routed>, <flag routed>, <flag routed>.	void _8BP_autoall() ; void _8BP_autoall_1(int flag) ;
COLAY, threshold_ascii, @collision, # COLAY, @collision, # COLAY, # COLAY	void _8BP_colay_3(int threshold, int* collision, int sp) ; void _8BP_colay_2(int* collision, int sp) ; void _8BP_colay_1(int sp) ; void _8BP_colay() ;
COLSP, #, @collided%, @collided%, @COLSP, #, @collided%, @collided%. COLSP, 32, ini, end COLSP, 33 ans, @collided% COLSP, # COLSP, 34, dy, dx	/* opération 32, ini,fin ou opération 34,dy,dx*/ void _8BP_colsp_3(int operation, int a, int b) ; /*opération 33 ou sp*/ void _8BP_colsp_2(int sp, int* collision) ; void _8BP_colsp_1(int sp) ;
COLSPALL,@qui%,@qui%,@avecqu i% COLSPALL, collisionneur COLSPALL	void _8BP_colspall_2(int* collider, int* collided) ; void _8BP_colspall_1(int collider_ini) ; void _8BP_colspall() ;
LAYOUT, y, x, @string\$, @string\$, @string\$, @string\$, @string\$, @string\$.	void _8BP_layout_3(int y, int x, char* cad) ;
LOCATESP, #, y, x	void _8BP_locatesp_3(char sp, int y, int x) ;
MAP2SP, y, x MAP2SP, statut	void _8BP_map2sp_2(int y, int x) ; void _8BP_map2sp_1(unsigned char status) ;
MOVER, #, dy, dx	void _8BP_mover_3(int sp, int dy,int dx) ; void _8BP_mover_1(int sp) ;
dy,dx, dy,dx, dy,dx	void _8BP_moverall_2(int dy, int dx) ; void _8BP_moverall() ;
MUSIQUE, C, drapeau, chanson, vitesse MUSIQUE, drapeau, chanson, vitesse MUSIQUE	void _8BP_music_4(int flag_c, int flag_repetition,int song, int speed) ; void _8BP_music() ;
PEEK, dir, @variable%	void _8BP.Peek_2(int address, int* data) ;
POKE, dir, value	void _8BP.poke_2(int address, int data) ;
PRINTAT, flag, y, x, @string	void _8BP.printat_4(int flag,int y,int x,char* cad) ;
PRINTSP, #, y, x PRINTSP, # PRINTSP,32, bits	void _8BP.printsp_1(int sp) ; void _8BP.printsp_2(int sp, int bits_background) ; void _8BP.printsp_3(int sp,int y,int x) ;
Le projet de loi a été adopté à l'unanimité par l'Assemblée nationale. L'impression de l'étiquette, le mode d'emploi, le mode d'emploi PRINTSPALL	void _8BP.printspall_4(int ini, int fin, int flag_anima, int flag_sync) ; void _8BP.printspall_1(int order_type) ; void _8BP.printspall() ;
RINK,tini,color1,color1,color2,...,color N RINK, sauter	void _8BP_rink_N(int num_params,int* ink_list) ; void _8BP_rink_1(int step) ;
ROUTESP, #, étapes	void _8BP_routesp_2(int sp, int steps) ; void _8BP_routesp_1(int sp) ;
ROUTEALL	void _8BP_routeall() ;

SETLIMITS, xmin, xmax, ymin, ymax	Void _8BP_setlimits_4(int xmin, int xmax, int ymin, int ymax)
SETUPSP, #, nombre_de_paramètres, valeur SETUPSP, #, 5, Vy, Vx	void _8BP_setupsp_3(int sp, int param, int value) ; void _8BP_setupsp_4(int sp, int param, int value1,int value2) ;
STARS, initstar, num, colour, dy, dx	void _8BP_stars_5(int star_ini, int num_stars,int colour, int dy, int dx) ; void _8BP_stars() ;
UMAP,adr_ini, adr_end, yini, yfin, xini, xfin	void _8BP_umap_6(int map_ini, int map_fin, int y_ini, int y_fin, int x_ini, int x_fin) ;

38 ANNEXE XIII : MiniBASIC en C

BASIC	C prototype
BORDER	<code>void _basic_border(char colour) ; //exemple _basic_border(7)</code>
APPEL	<code>void _basic_call(unsigned int address) ; // exemple _basic_call(0xbd19)</code>
TIRAGE	<code>void _basic_draw(int x, int y) ;</code>
INK	<code>void _basic_ink(char ink1,char ink2) ;</code>
INKEY	<code>char _basic_inkey(char key) ; //prend environ 0,3 ms. lent mais simple</code>
LOCALISER	<code>void _basic_locate(unsigned int x, unsigned int y) ; // exemple : _basic_locate(2,25);_basic_print("TEST") ;</code>
DÉPLACER	<code>void _basic_move(int x, int y) ;</code>
PAPIER	<code>void _basic_paper(char ink) ;</code>
PEEK	<code>char _basic_peek(unsigned int address) ;</code>
GRAPHIQUES PEN	<code>void _basic_pen_graph(char ink) ;</code>
PEN	<code>void _basic_pen_txt(char ink) ;</code>
POKE	<code>void _basic_poke(unsigned int address, unsigned char données) ;</code>
PLOT	<code>void _basic_plot(int x, int y) ;</code>
IMPRIMER	<code>void _basic_print(char *cad) ; //exemple : _basic_print("Hello")</code>
RND	<code>unsigned int _basic_rnd(int max) ; //exemple : num=_basic_rnd(50)</code>
SON	<code>void _basic_sound(unsigned char nChannelStatus, int nTonePeriod, int nDuration, unsigned char nVolume, char nVolumeEnvelope, char nToneEnvelope, unsigned char nNoisePeriod) ;</code>
STR\$	<code>char* _basic_str(int num) ; //similaire à STR\$ //exemple : _basic_print(_basic_str(num))</code>
TEMPS	<code>unsigned int _basic_time() ; //retourne un int non signé,(0..65535). En tant qu'entier, lorsque // atteindre 32768 aller à -32768</code>