

# 8 BITS DE PODER

En tu AMSTRAD CPC



“Una guía para programadores de 8 bit en el siglo XXI”

V37\_01

Jose Javier García Aranda



## INDICE

<b>1</b>	<b>¿POR QUÉ PROGRAMAR HOY UNA MAQUINA DE 1984? .....</b>	<b>7</b>
<b>2</b>	<b>FUNCIONES DE 8BP Y USO DE LA MEMORIA .....</b>	<b>9</b>
2.1	¿QUÉ ES UNA LIBRERÍA RSX? .....	10
2.2	FUNCIONES DE 8BP .....	11
2.3	ARQUITECTURA DEL AMSTRAD CPC .....	12
2.4	USO DE LA MEMORIA DE 8BP .....	14
<b>3</b>	<b>HERRAMIENTAS NECESARIAS .....</b>	<b>17</b>
<b>4</b>	<b>PASOS QUE DEBES DAR PARA HACER UN JUEGO.....</b>	<b>19</b>
4.1	ESTRUCTURA EN DIRECTORIOS DE TU PROYECTO.....	19
4.2	TU JUEGO EN 6 FICHEROS .....	19
4.3	METODO ABREVIADO: SOLO 3 FICHEROS.....	21
4.4	CREAR UN DISCO O UNA CINTA CON TU JUEGO .....	22
4.4.1	<i>Hacer un disco.....</i>	22
4.4.2	<i>Hacer una cinta .....</i>	23
4.4.3	<i>Solución de problemas con LOAD y MEMORY.....</i>	25
<b>5</b>	<b>CICLO DE JUEGO.....</b>	<b>27</b>
5.1	COMO MEDIR LOS FPS DE TU CICLO DE JUEGO.....	27
<b>6</b>	<b>SPRITES .....</b>	<b>29</b>
6.1	EDITAR SPRITES CON SPEDIT Y ENSAMBLARLOS.....	29
6.2	SPRITE FLIPPING .....	34
6.3	SPRITES CON SOBREESCRITURA .....	36
6.3.1	<i>Quieres más colores y usar sobreescritura .....</i>	40
6.3.2	<i>Sobreescritura con 4 colores de fondo .....</i>	41
6.3.3	<i>Sobreescritura en MODE 1 .....</i>	42
6.3.4	<i>Cómo pintar sprites “por detrás” del fondo .....</i>	43
6.4	TABLA DE ATRIBUTOS DE SPRITES .....	43
6.5	IMPRESIÓN DE TODOS LOS SPRITES Y ORDENADOS .....	48
6.6	COLISIONES ENTRE SPRITES .....	50
6.7	AJUSTE DE LA SENSIBILIDAD DE LA COLISIÓN DE SPRITES .....	52
6.8	QUIÉN COLISIONA Y CON QUIÉN: COLSPALL.....	53
6.8.1	<i>Cómo programar un disparo múltiple usando COLSPALL.....</i>	55
6.8.2	<i>Quien colisiona cuando hay varios solapes .....</i>	56
6.8.3	<i>Uso avanzado del byte de status en colisiones .....</i>	57
6.9	TABLA DE SECUENCIAS DE ANIMACIÓN .....	58
6.10	SECUENCIAS DE MUERTE .....	60
6.11	MACROSECUENCIAS DE ANIMACIÓN .....	60
<b>7</b>	<b>TU PRIMER JUEGO SENCILLO .....</b>	<b>63</b>
7.1	AHORA, A SALTAR! BOING, BOING!! .....	63
<b>8</b>	<b>JUEGOS DE PANTALLAS: LAYOUT O “TILE MAP” .....</b>	<b>67</b>
8.1	DEFINICIÓN Y USO DEL LAYOUT .....	67
8.2	EJEMPLO DE JUEGO CON LAYOUT .....	70
8.3	CÓMO ABRIR UNA COMPUERTA EN EL LAYOUT .....	72
8.4	UN COMECOCOS: LAYOUT CON FONDO .....	73

8.5	CÓMO AHORRAR MEMORIA EN TUS LAYOUTS .....	75
<b>9</b>	<b>PROGRAMACIÓN AVANZADA Y “LÓGICAS MASIVAS” .....</b>	<b>77</b>
9.1	MEDICIÓN DE LA VELOCIDAD DE LOS COMANDOS.....	77
9.2	HAZ UNA SOLA LÓGICA PARA GOBERNAR TODAS TUS PANTALLAS .....	83
9.3	TÉCNICA DE “LÓGICAS MASIVAS” .....	84
9.3.1	<i>Mueve 32 sprites con lógicas masivas.....</i>	85
9.3.2	<i>Ejecución alternada y periódica en cascada.....</i>	86
9.3.3	<i>Ejemplo sencillo de lógica masiva .....</i>	88
9.3.4	<i>Movimiento “en bloque” de escuadrones .....</i>	89
9.3.5	<i>Técnica de lógicas masivas en juegos tipo “pacman” .....</i>	90
9.3.6	<i>Reducción del número de instrucciones en el ciclo de juego .....</i>	91
9.3.7	<i>Enrutando sprites con “lógicas masivas ” .....</i>	95
<b>10</b>	<b>TRAYECTORIAS COMPLEJAS: COMANDO ROUTEALL.....</b>	<b>99</b>
10.1	COLOCA A UN SPRITE EN MITAD DE UNA RUTA : ROUTESP.....	101
10.2	CREACIÓN DE RUTAS AVANZADAS .....	103
10.2.1	<i>Cambios de estado forzados desde rutas.....</i>	103
10.2.2	<i>Cambios de secuencia forzados desde rutas .....</i>	105
10.2.3	<i>Cambios de imagen forzados desde rutas .....</i>	105
10.2.4	<i>Cambios de ruta forzados desde rutas .....</i>	109
10.2.5	<i>Animación forzada desde rutas .....</i>	109
10.2.6	<i>Como construir rutas “dinámicas” (no predefinidas) .....</i>	110
10.2.7	<i>Programación de rutas que incluyen patrones.....</i>	111
10.2.8	<i>Tipología de rutas.....</i>	112
<b>11</b>	<b>JUEGOS CON SCROLL.....</b>	<b>113</b>
11.1	STARS: SCROLL DE ESTRELLAS O TIERRA MOTEADA .....	113
11.2	SCROLL USANDO MOVERALL y/o AUTOALL .....	116
11.3	TÉCNICA DEL “MANCHADO” .....	118
11.4	MAP2SP: SCROLL BASADO EN UN MAPA DEL MUNDO.....	121
11.4.1	<i>Mapa del mundo (Map Table) .....</i>	122
11.4.2	<i>Uso de la función MAP2SP .....</i>	124
11.4.3	<i>Ejemplo de fichero de fases .....</i>	127
11.5	SCROLL PARALLAX .....	128
11.6	ACTUALIZACIÓN DINÁMICA DEL MAPA:  UMAP .....	129
11.7	ANIMACIÓN Y SCROLL POR TINTAS: COMANDO RINK .....	131
11.7.1	<i>Carreras de coches 2D .....</i>	133
11.7.2	<i>Scroll de Ladrillos .....</i>	134
<b>12</b>	<b>JUEGOS DE PLATAFORMAS.....</b>	<b>137</b>
<b>13</b>	<b>MINICARACTERES REDEFINIBLES: PRINTAT.....</b>	<b>139</b>
13.1	CREA TU PROPIO ALFABETO DE MINICARACTERES .....	140
<b>14</b>	<b>PSEUDO 3D .....</b>	<b>143</b>
14.1	PROYECCION 3D .....	145
14.1.1	<i>Matemáticas de la proyección pseudo 3D.....</i>	146
14.1.2	<i>Curvas.....</i>	150
14.2	ZOOM IMAGES .....	151
14.3	USO DE SEGMENTOS .....	153

<b>15 MÚSICA.....</b>	<b>157</b>
15.1 EDITAR MÚSICA CON WYZ TRACKER.....	157
15.2 ENSAMBLAR LAS CANCIONES .....	158
15.3 QUÉ HACER SI NO TE CABE LA MÚSICA EN 1400 BYTES .....	159
<b>16 GUÍA DE REFERENCIA DE LA LIBRERÍA 8BP.....</b>	<b>161</b>
16.1 FUNCIONES DE LA LIBRERÍA .....	161
16.1.1 /3D .....	161
16.1.2 /ANIMA.....	162
16.1.3 /ANIMALL.....	163
16.1.4 /AUTO.....	164
16.1.5 /AUTOALL.....	164
16.1.6 /COLAY.....	164
16.1.7 /COLSP.....	165
16.1.8 /COLSPALL.....	167
16.1.9 /LAYOUT.....	168
16.1.10 /LOCATESP.....	170
16.1.11 /MAP2SP .....	171
16.1.12 /MOVER.....	172
16.1.13 /MOVERALL.....	173
16.1.14 /MUSIC.....	173
16.1.15 /MUSICOFF.....	174
16.1.16 /PEEK .....	174
16.1.17 /POKE.....	174
16.1.18 /PRINTAT .....	174
16.1.19 /PRINTSP.....	175
16.1.20 /PRINTSPALL.....	176
16.1.21 /RINK.....	178
16.1.22 /ROUTEALL .....	179
16.1.23 /ROUTESP.....	181
16.1.24 /SETLIMITS.....	181
16.1.25 /SETUPSP.....	182
16.1.26 /STARS.....	184
16.1.27 /UMAP .....	185
<b>17 ENSAMBLADO DE LA LIBRERÍA, MÚSICA Y GRÁFICOS .....</b>	<b>187</b>
17.1 MAKEALL.ASM .....	189
17.2 ESTRUCTURA DEL FICHERO DE IMAGÉNES .....	190
17.3 ESTRUCTURA DEL FICHERO DE SECUENCIAS DE ANIMACIÓN .....	190
17.4 ESTRUCTURA DEL FICHERO DE RUTAS .....	191
17.5 ESTRUCTURA DEL FICHERO DE MAPA DEL MUNDO .....	191
<b>18 COMO HACER UNA TABLA DE PUNTUACIONES.....</b>	<b>193</b>
<b>19 POSIBLES MEJORAS FUTURAS A LA LIBRERÍA .....</b>	<b>195</b>
19.1 MEMORIA PARA UBICAR NUEVAS FUNCIONES .....	195
19.2 IMPRESIÓN A RESOLUCIÓN DE PÍXEL .....	195
19.3 LAYOUT DE MODE 1 .....	195
19.4 CAPACIDAD FILMATION .....	195
19.5 FUNCIONES DE SCROLL POR HARDWARE .....	196
19.6 MIGRAR LA LIBRERÍA 8BP A OTROS MICROORDENADORES.....	197

<b>20 ALGUNOS JUEGOS HECHOS CON 8BP .....</b>	<b>199</b>
20.1 MUTANTE MONTOYA .....	199
20.2 ANUNNAKI, NUESTRO PASADO ALIEN .....	200
20.3 NIBIRU .....	200
20.4 FRESH FRUITS & VEGETABLES.....	201
20.5 “3D RACING ONE” .....	202
20.6 SPACE PHANTOM .....	203
20.7 FROGGER ETERNO .....	204
20.8 ERIDU: THE SPACE PORT .....	205
20.9 HAPPY MONTY .....	205
20.10 MINI JUEGOS .....	206
20.10.1 <i>Mini-pong</i> .....	206
20.10.2 <i>Mini-Invaders</i> .....	207
<b>21 APENDICE I: ORGANIZACIÓN DE LA MEMORIA DE VIDEO .....</b>	<b>209</b>
21.1 EL OJO HUMANO Y LA RESOLUCIÓN DEL CPC .....	209
21.2 LA MEMORIA DE VIDEO.....	209
21.2.1 <i>Mode 2</i> .....	209
21.2.2 <i>Mode 1</i> .....	209
21.2.3 <i>Mode 0</i> .....	210
21.2.4 <i>Memoria de la pantalla</i> .....	210
21.3 CÁLCULO DE UNA DIRECCIÓN DE PANTALLA .....	212
21.4 BARRIDOS DE PANTALLA .....	212
21.5 CÓMO HACER UNA PANTALLA DE CARGA PARA TU JUEGO .....	213
<b>22 APENDICE II: LA PALETA .....</b>	<b>215</b>
<b>23 APENDICE III: INKEY CODES .....</b>	<b>217</b>
<b>25 APENDICE IV: TABLA ASCII DEL AMSTRAD CPC .....</b>	<b>219</b>
<b>26 APENDICE V: ALGUNOS EFECTOS DE SONIDO .....</b>	<b>221</b>
<b>27 APENDICE VI: RUTINAS INTERESANTES DEL FIRMWARE .....</b>	<b>223</b>
<b>28 APENDICE VII: TABLA DE ATRIBUTOS DE SPRITES.....</b>	<b>225</b>
<b>29 APENDICE VIII: MAPA DE MEMORIA DE 8BP .....</b>	<b>227</b>
<b>30 APENDICE IX: COMANDOS DISPONIBLES 8BP.....</b>	<b>229</b>
<b>31 APENDICE X: CORRESPONDENCIAS RSX/CALL.....</b>	<b>231</b>

# 1 ¿Por qué programar hoy una maquina de 1984?

Porque las limitaciones no son un problema sino una fuente de inspiración.

Las limitaciones, ya sean de una maquina o de un ser humano, o en general de cualquier recurso disponible estimulan nuestra imaginación para poder superarlas. El AMSTRAD, una maquina de 1984 basada en el microprocesador Z80, posee una reducida memoria (64KB) y una reducida capacidad de procesamiento, aunque sólo si lo comparamos con los ordenadores actuales. Esta máquina es en realidad un millón de veces más rápida que la que construyó Alan Turing para descifrar los mensajes de la maquina enigma en 1944.

Como todos los ordenadores de los años 80, el AMSTRAD CPC arrancaba en menos de un segundo, con el intérprete BASIC dispuesto a recibir comandos de usuario, siendo el BASIC el lenguaje con el que los programadores aprendían y hacían sus primeros desarrollos. El BASIC del AMSTRAD era particularmente rápido en comparación al de sus competidores. ¡Y estéticamente era un ordenador muy atractivo!



*Fig. 1. El mítico AMSTRAD modelo CPC464*

En cuanto al microprocesador Z80 ni siquiera es capaz de multiplicar (en BASIC puedes multiplicar, pero eso se basa en un programa interno que implementa la multiplicación mediante sumas o desplazamientos de registros), tan solo puede hacer sumas, restas y operaciones lógicas. A pesar de ello era la mejor CPU de 8 bit y tan sólo constaba de 8500 transistores, a diferencia de otros procesadores como el M68000 cuyo nombre precisamente le viene de tener 68000 transistores.

CPU	Número de transistores	MIPS (millones de instrucciones por segundo)	Ordenadores y consolas que lo incorporan
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 por core)	Raspberry pi 2, Nintendo 3DS, Samsung galaxy, ...
Intel i7 (5 <sup>a</sup> generación)	2.600.000.000	238310 @ 3Ghz (¡casi 500.000 veces más rápido que un Z80 !)	PC

*Tabla 1Comparativa de MIPS*

Ello hace que programarlo sea extremadamente interesante y estimulante para lograr resultados satisfactorios. Toda nuestra programación debe ir orientada a reducir complejidad computacional espacial (memoria) y temporal (operaciones), obligándonos a inventar trucos, artimañas, algoritmos, etc., y haciendo de la programación una aventura apasionante. Es por ello, que la programación de máquinas de baja capacidad de procesamiento es un concepto atemporal, no sujeto a modas ni condicionado por la evolución de la tecnología.

Todo el código de este libro, incluida la librería para que hagas tus propios juegos o para que hagas contribuciones a la librería, lo puedes encontrar en el proyecto GitHub “8BP”, en esta URL. Basta con que te descargas el zip

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

También existe un blog con mucha información en:

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

Y un canal de youtube:

[https://www.youtube.com/channel/UCThves0T-jLU\\_s8a5Z0jBFA](https://www.youtube.com/channel/UCThves0T-jLU_s8a5Z0jBFA)

## 2 Funciones de 8BP y uso de la memoria

La librería 8BP no es un “motor de juegos”. Es algo intermedio entre una simple extensión de comandos BASIC y un motor de juegos.

Los motores de juegos como el game-maker, el AGD (Arcade Game Designer), el Unity, y muchos otros, limitan en cierta medida la imaginación del programador, obligándole a usar unas determinadas estructuras, a programar en lenguaje limitado de script la lógica de un enemigo, a definir y enlazar pantallas de juego, etc.



Fig. 2 Motores de juego versus 8BP

La librería 8BP es diferente. Es una librería capaz de ejecutar deprisa aquello que el BASIC no puede hacer. Cosas como imprimir sprites a toda velocidad, o mover bancos de estrellas por la pantalla, son cosas que el BASIC no puede hacer y 8BP lo consigue. ¡Y ocupa sólo 8 KB!!

BASIC es un lenguaje interpretado. Eso significa que cada vez que el ordenador ejecuta una línea de programa debe primero verificar que se trata de un comando válido, comparando la cadena de caracteres del comando con todas las cadenas de comandos válidos. A continuación, debe validar sintácticamente la expresión, los parámetros del comando e incluso los rangos permitidos para los valores de dichos parámetros. Además, los parámetros los lee en formato texto (ASCII) y debe convertirlos a datos numéricos. Finalizada toda esta labor, procede con la ejecución. Pues bien, toda esa labor que se realiza en cada instrucción es la que diferencia un programa compilado de un programa interpretado como los escritos en BASIC.

Dotando al BASIC de los comandos proporcionados por 8BP, es posible hacer juegos de calidad profesional, ya que la lógica del juego que programes puede ejecutarse en BASIC mientras que las operaciones intensivas en el uso de CPU como imprimir en pantalla o detectar colisiones entre sprites, etc. son llevadas a cabo en código máquina por la librería. Sin embargo, no todo es facilidad y ausencia de problemas. Aunque la librería 8BP te va a proporcionar funciones muy útiles en videojuegos, deberás usarla con cautela pues cada comando que invoques atravesará la capa de análisis sintáctico del BASIC, antes de llegar al inframundo del código máquina donde se encuentra la función, por lo que el rendimiento nunca será el óptimo. Deberás ser astuto y ahorrar instrucciones, medir los tiempos de ejecución de instrucciones y trozos de tu programa y pensar estrategias para ahorrar tiempo de ejecución. Toda una aventura de ingenio y diversión. Aquí aprenderás como hacerlo e incluso te presentaré una técnica a la que he

llamado “lógicas masivas” que te permitirá acelerar tus juegos a límites que quizás considerabas imposibles.

Además de la librería, tienes a tu disposición un sencillo pero completo editor de sprites y gráficos y una serie de herramientas magníficas que te permitirán disfrutar en el siglo XXI de la aventura de programar un microordenador.



Fig. 3 Resumen de 8BP

## 2.1 ¿Qué es una librería RSX?

RSX es el acrónimo de Resident System eXtensions. Las librerías como 8BP que proporcionan comandos para extender el BASIC se les llama librerías RSX.

En el CPC6128, algunos de los comandos que se utilizan para manejar la unidad de disco son comandos RSX que vienen preinstalados, tales como |TAPE, |DISC, |A, |B, |CPM y otros. Si esta funcionalidad no existiese, cada rutina de 8BP habría que invocarla con un CALL <dirección>, por lo que la existencia de RSX hace más entendibles los programas.

No todo es paz y armonía. Usar RSX es más lento que usar CALL directamente y además si declaramos 10 comandos nuevos en una librería, el décimo comando puede tardar 1ms más en empezar a ejecutarse que el primero. La librería 8BP tiene 27 comandos y el último comienza a ejecutarse 2ms mas tarde por encontrarse en el último lugar de la lista. Es uno de los problemas de estar bajo el intérprete BASIC.

8BP compensa este problema creando los comandos de uso más frecuente al principio de la lista, y dejando para el final los menos frecuentes. Como pronto deducirás, el comando más frecuentemente usado en 8BP es |PRINTSPALL, el cual imprime todos los sprites en pantalla. Dicho comando es, por consiguiente, el primero de la lista.

## 2.2 Funciones de 8BP

Tras cargar la librería con el comando: LOAD “8BP.BIN” e invocar desde BASIC la función \_INSTALL\_RSX (definida en código máquina) mediante el comando BASIC:  
**CALL &6b78**

Dispondrás de los siguientes comandos, que aprenderás a usar con este libro

3D, <flag>, #, offsety	Activa el modo de proyección pseudo 3D
ANIMA, #	cambia el fotograma de un sprite según su secuencia
ANIMALL	cambia el fotograma de los sprites con flag animación activado (no hace falta invocarla, basta con un flag en la instrucción PRINTSPALL para que sea invocada)
AUTO, #	movimiento automático de un sprite de acuerdo a su velocidad en la tabla de sprites
AUTOALL, <flag enrutado>	movimiento de todos los sprites con flag de movimiento automático activo
COLAY, umbral_ascii, @colision, #  COLAY, @colision, #  COLAY, #  COLAY	detecta la colisión con el layout y retorna 1 si hay colisión. Acepta un número variable de parámetros (siempre en el mismo orden) desde 4 hasta ninguno (tiene memoria)
COLSP, #, @collided%  COLSP, 32, ini, fin  COLSP, 33, @collided%  COLSP, 34, dy, dx	retorna primer sprite con el que colisiona #. Se puede configurar el comando con los códigos 32,33 y 34
COLSPALL, @quien%, @conquien%  COLSPALL, colisionador  COLSPALL	Retorna quien ha colisionado (collider) y con quién ha colisionado (collided)
LAYOUT, y, x, @string\$	imprime un layout de imágenes de 8x8 y rellena map layout
LOCATESP, #, y, x	cambia las coordenadas de un sprite (sin imprimirla)
MAP2SP, y, x  MAP2SP, status	crea sprites para pintar el mundo en juegos con scroll. Los sprites se crean con estado = status
MOVER, #, dy, dx	movimiento relativo de un solo sprite
MOVERALL, dy,dx	movimiento relativo de todos los sprites con flag de movimiento relativo activo
MUSIC, cancion, speed	comienza a sonar una melodía
MUSICOFF	deja de sonar la melodía
PEEK, dir, @variable%	lee un dato 16bit (puede ser negativo)
POKE, dir, valor	introduce un dato 16bit (que puede ser negativo)
PRINTAT, flag, y, x, @string	Imprime una cadena de “minicaracteres” redefinibles
PRINTSP, #, y, x  PRINTSP, #  PRINTSP,32, bits	imprime un solo sprite (# es su número) sin tener en cuenta byte de status. Si se especifica 32 entonces establecemos bits fondo
PRINTSPALL, ini, fin, anima, sync  PRINTSPALL, ordermode	imprime todos los sprites con flag de impresión activo. Si se invoca con un solo parámetro, se establece el modo de ordenamiento
RINK,tini,color1,color2,...,colorN  RINK, salto	Rota un conjunto de tintas de acuerdo a un patrón definible compuesto por cualquier número de tintas
ROUTESP, #, pasos	Hace recorrer de golpe N pasos en la ruta asignada a un sprite,
ROUTEALL	Modifica la velocidad de los sprites con flag de ruta (no hace falta invocarla, basta con un flag en la instrucción AUTOALL para que sea invocada)
SETLIMITS, xmin, xmax, ymin, ymax	define la ventana de juego, donde se hace clipping
SETUPSP, #, param_number, valor	modifica un parámetro de un sprite
STARS, initstar, num, color, dy, dx	scroll de un conjunto de estrellas
UMAP,adr_ini, adr_end, yini, yfin, xini, xfin	Actualiza los ítems del mapa del mundo con un subconjunto de elementos de un mapa mayor

Tabla 2 Comandos disponibles en la librería 8BP

Fíjate que aparece una barra vertical al principio de cada uno por ser “extensiones” del BASIC.

Adicionalmente dispones de un comando experimental:

|RETROTIME, fecha

Este comando permite transformar tu CPC en una máquina del tiempo, con solo introducir la fecha de destino deseada. La única limitación del comando es que debes introducir una fecha igual o posterior a la del nacimiento del AMSTRAD CPC, Abril de 1984,

|RETROTIME, “01/04/1984”

Por favor, utiliza esta funcionalidad con precaución. Podrías crear una paradoja temporal y destruir el mundo.

Aunque de momento puedes tener cierto escepticismo respecto lo que puedes llegar a hacer con la librería 8BP, pronto descubrirás que el uso de esta librería junto con técnicas de programación avanzadas que aprenderás en este libro te permitirá hacer juegos profesionales en BASIC, algo que quizás creías imposible.

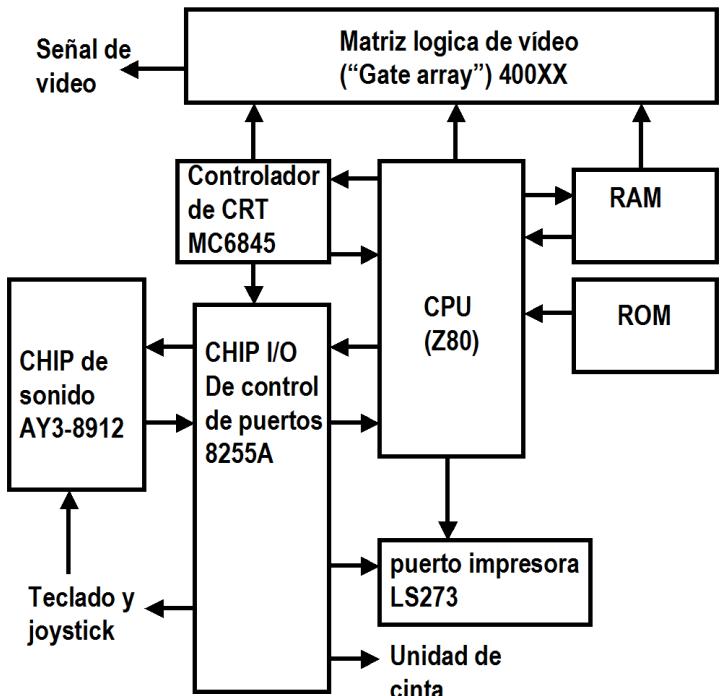
#### **Nota importante para el programador:**

La librería 8BP está optimizada para ser muy rápida. Es por ello que no chequea que hayas colocado correctamente los parámetros de cada comando, ni que tengan un valor adecuado. Si algún parámetro está mal puesto, es muy posible que el ordenador se cuelgue al ejecutar el comando. Chequear estas cosas lleva tiempo de ejecución y el tiempo es un recurso que no se puede desperdiciar, ni un milisegundo.

### **2.3 Arquitectura del AMSTRAD CPC**

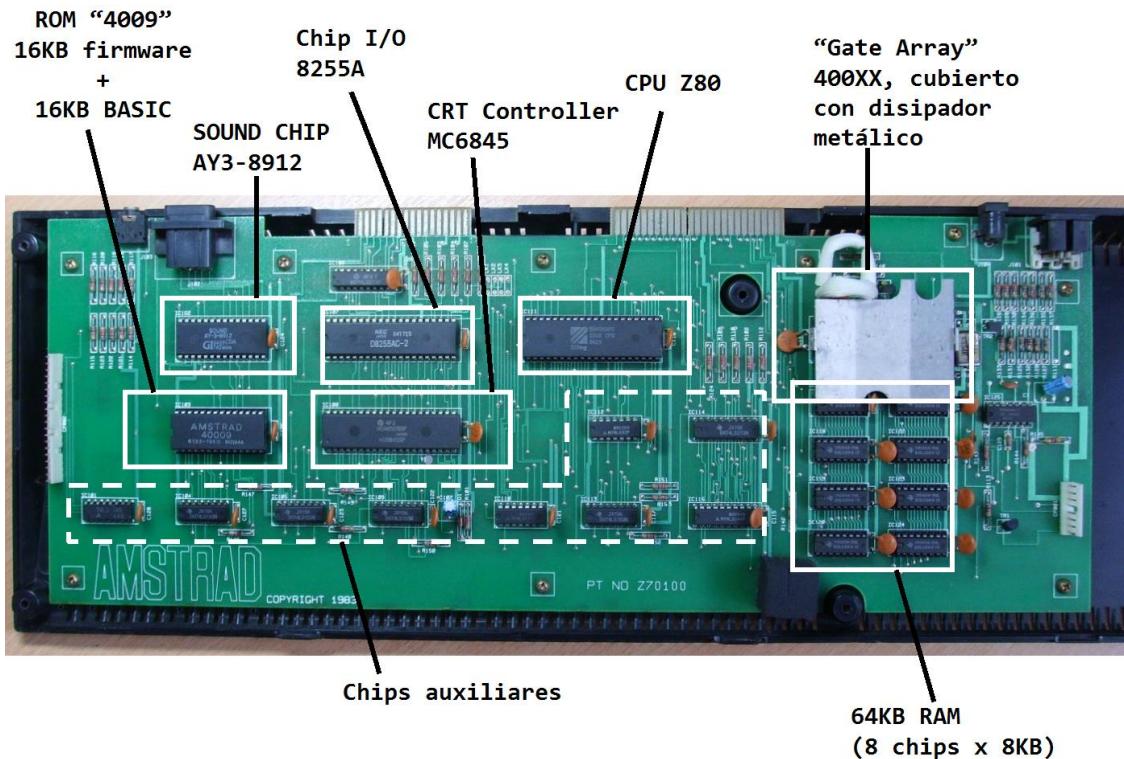
Este apartado es útil para comprender posteriormente como usa la librería 8BP la memoria.

El AMSTRAD es una computadora basada en el microprocesador Z80, funcionando a 4MHz. Como se aprecia en su diagrama de arquitectura, tanto la CPU como la matriz lógica de video (llamada “gate array”) acceden a la memoria RAM, por lo que para poder “turnarse”, los accesos a la memoria desde la CPU son retrasados, dando como resultado una velocidad efectiva de 3.3Mhz. Esto sigue siendo bastante potencia.



*Fig. 4 Arquitectura del AMSTRAD*

La memoria RAM de video es accedida por el gate array 50 veces por segundo para poder enviar una imagen a la pantalla. En ordenadores más antiguos (como el Sinclair ZX81) esta labor era encomendada al procesador, restándole aun más potencia.



*Fig. 5 Identificación de componentes en la placa*

El Z80 posee un bus de direcciones de 16bit, por lo que no es capaz de direccionar más de 64KB. Sin embargo, el Amstrad posee 64kB RAM y 32kB ROM. Para poder direccionarlas, el AMSTRAD es capaz de “conmutar” entre unos bancos y otros, de modo que, por ejemplo, si se invoca a un comando BASIC, se conmuta al banco de

La memoria de video, lo que vemos en la pantalla, es parte de las 64KB de memoria RAM, en concreto son las 16KB situadas en la zona superior de la memoria. La memoria se numera desde 0 hasta 65535 bytes. Pues bien, los 16KB comprendidos entre la dirección 49152 y 65535 es la memoria de video. En hexadecimal se representa como &C000 hasta &FFFF.

ROM donde se almacena el intérprete BASIC, que está solapado con las 16KB de pantalla. Este mecanismo es sencillo y efectivo.

Además de la ROM que contiene el intérprete BASIC de 16KB situado en la zona de memoria alta, hay otras 16KB de ROM situadas en la memoria baja, donde se encuentran las rutinas del firmware (lo que podría considerarse el sistema operativo de esta máquina). En total (interpretar BASIC y firmware) suman 32KB

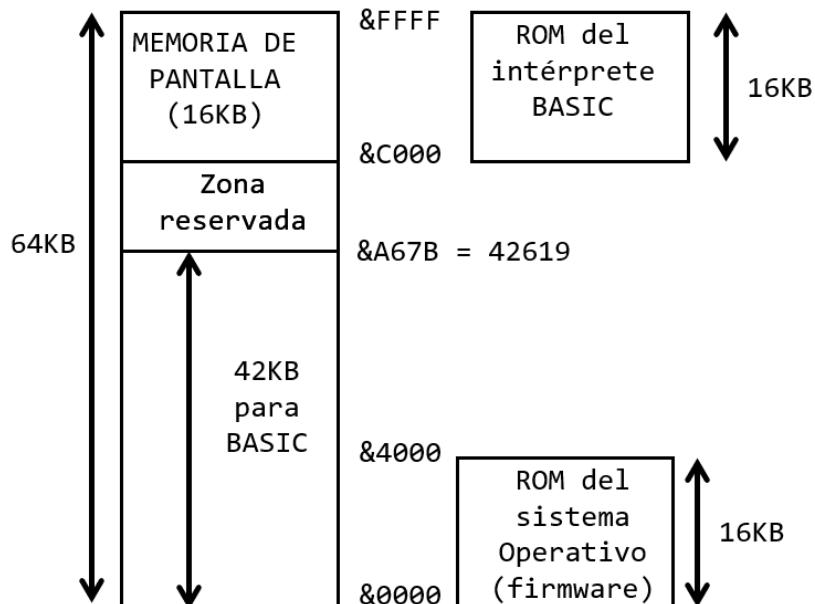


Fig. 6 Memoria del AMSTRAD

Como se aprecia en el mapa de memoria, de los 64KB de RAM, 16KB (desde &C000 hasta &FFFF) son la memoria de vídeo. Los programas en BASIC pueden ocupar desde la posición &40 (dirección 64) hasta la 42619, pues más allá hay variables del sistema. Es decir, que se dispone de unas 42KB para BASIC, tal y como podemos comprobar al imprimir la variable del sistema HIMEM (abreviatura de "High Memory").

```
print HIMEM
42619
Ready
```

Fig. 7 Variable de sistema HIMEM

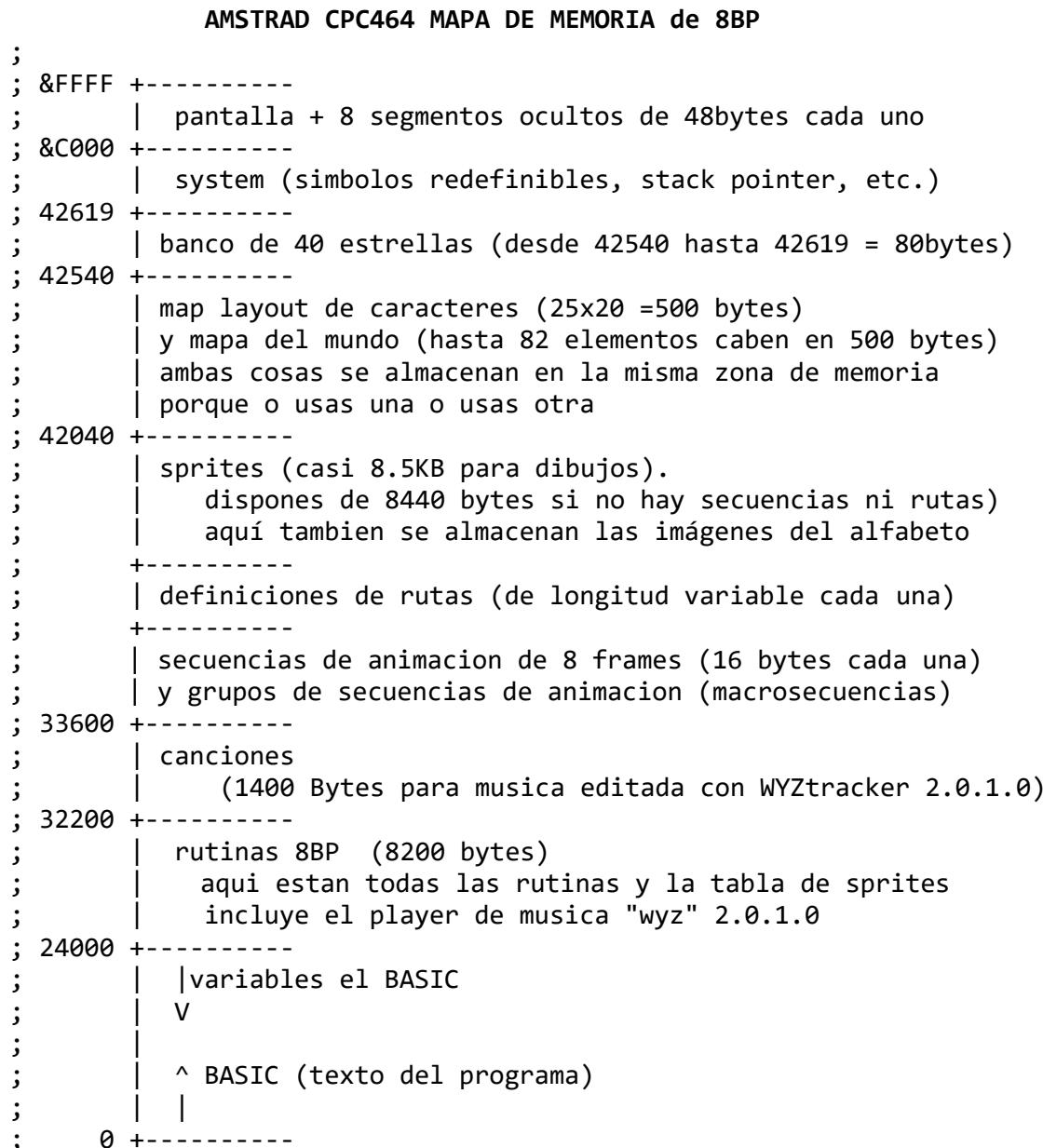
El funcionamiento del BASIC tiene en cuenta el almacenamiento del programa en direcciones crecientes desde la posición &40, mientras que una vez en ejecución, las variables que se declaran deben ocupar espacio para almacenar los valores que toman y puesto que no pueden ocupar la misma zona donde se almacena el programa, sencillamente se empiezan a almacenar en la dirección más alta posible, la 42619 y a medida que se usan más variables se consumen direcciones de memoria decrecientes. En el AMSTRAD cada variable numérica de tipo entero ocupará 2 bytes de memoria.

## 2.4 Uso de la memoria de 8BP

La librería 8BP se carga en la zona de memoria alta disponible. Es importante entender cómo funciona el BASIC, para poder usar la librería.

El texto de un programa escrito en BASIC se almacena a partir de la dirección &40 (en decimal es 64) pero una vez que empieza a ejecutarse, los valores que van tomando las variables de programa se almacenan ocupando posiciones decrecientes desde la 42619, de modo que si un programa es grande, podrían llegar a “chocar” con el propio texto del programa, destruyendo parte del mismo. Esto normalmente no va a ocurrir, no te preocunes.

La librería se carga a partir de la dirección 24000, destinando la memoria a las funciones, el player de música, las canciones, el mapa del mundo y los dibujos, tal y como se muestra en el siguiente diagrama



*Fig. 8 Memoria usando 8BP*

Antes incluso de cargar la librería, deberás ejecutar el comando:

**MEMORY 23999**

Para limitar el espacio ocupado por el BASIC. De esta manera, las variables de BASIC empezaran a ocupar espacio en ejecución desde la dirección 23999 hacia abajo. Tus programas pueden disponer de 24 KB de memoria, aunque como las variables ocupan espacio, lo normal es que el tamaño máximo de tu programa sea inferior. No obstante, hablamos de una cantidad de memoria muy respetable. Te costará mucho hacer un juego de 24 KB, te lo aseguro.

En todo ordenador existe un área de memoria llamada “pila” o “stack”. Es una zona utilizada para almacenar la dirección de memoria donde se encuentra un programa antes de saltar a una rutina, de modo que pueda volver por donde estaba cuando dicha rutina termine. Se utiliza en bajo nivel (lenguaje ensamblador). En el Amstrad es una zona ubicada en lo que hemos denominado área “system” de 8BP (área desde la dirección &C000 (en decimal 49152) hasta la 42619. La pila crece hacia direcciones mas bajas a medida que almacena direcciones y al retornar de una rutina vuelve a decrecer. Podríamos llegar a pensar que, si una rutina llama a otra y a su vez esta llama a otra y asi sucesivamente, la pila crecería tanto que se saldría de la zona “system” e invadiría el banco de estrellas, pero no te preocupes, 8BP mantiene la pila bajo control.

### 3 Herramientas necesarias

**Winape:** emulador para S.O. windows con editor para editar y probar tu programa BASIC. Y también para ensamblar los gráficos y las músicas

**SPEDIT:** (“Simple Sprite Editor”) herramienta BASIC para editar tus gráficos. El resultado de spedit es código en ensamblador que se envía a la impresora del Amstrad CPC. Ejecutando la herramienta dentro de Winape, la impresora se redirige a un fichero de texto de modo que tus gráficos se almacenarán en un fichero txt. Esta herramienta ha sido creada para complementar a la librería 8BP y los graficos de todos los juegos que he creado los he hecho con SPEDIT

**Wyztracker:** para componer música, bajo windows. El programa capaz de tocar las melodías compuestas por Wyztracker es el Wyzplayer, el cual está integrado dentro de 8BP. Tras ensamblar la música podrás hacerla sonar con un sencillo comando |MUSIC

**Librería 8BP:** instala nuevos comandos accesibles desde BASIC para tu programa. Como comprobarás, esto va a ser el “corazón” que mueva la maquinaria que construyas.

**CPCDiskXP :** te permite grabar un disquete de 3.5” que luego podrás insertar en tu CPC6128 si dispones de un cable para conectar una disquetera. Si quieres hacer una cinta de audio para CPC464 esta herramienta no la necesitas

**2CDT:** imprescindible herramienta para crear ficheros .cdt. Yo normalmente extraigo los ficheros de un .dsk al disco de windows usando CPCDiskXP y después uso 2cdt para crear el archivo cdt

Opcionalmente:

**ConvImgCPC:** editor de imágenes de carga para tus juegos. También convierte desde BMP. Programado por Ludovic Deplanque (“DEMONIAK”)

**RGAS:** (Retro Game Asset Studio) potente editor de sprites, evolucionado del a herramienta AMSprite, creado por Lachlan Keown. Este editor de sprites es compatible con 8BP y corre bajo Windows. Cuando Spedit se te quede pequeño, esta puede ser la mejor opción.

**Fabacom:** compilador ejecutable dentro del AMSTRAD CPC 6128 o desde el emulador Winape para compilar tu programa BASIC y hacerlo ejecutar más rápido. Es compatible con las llamadas a los comandos de la librería 8BP. Sin embargo, no es recomendable por varios motivos:

- Tu programa ocupará mucho mas pues fabacom necesita 10KB adicionales para sus librerías, y además, una vez que compila tu programa sigue ocupando lo mismo, de modo que un programa de 10KB se transforma en uno de 20KB.
- Hay documentados algunos problemas de incompatibilidad de este compilador con algunas instrucciones de BASIC.
- Además, como verás a lo largo de este libro, puedes lograr una velocidad muy alta sin necesidad de compilar.

**CPC BASIC compiler:** compilador ejecutable para windows. Es compatible con las llamadas a los comandos de la librería 8BP. A diferencia de fabacom, el programa compilado solo ocupa unos 5KB extra, sin embargo, reserva 16KB de trabajo para funcionar de modo que apenas te deja espacio para tu programa, ya que en total "roba" 20 KB. Y además no es 100% compatible locomotive BASIC.

La ganancia en velocidad puede llegar a un 50%, dependiendo del juego. Es decir, que es como pasar de 100Km/h a 150Km/h. Esto no está mal, pero piensa que hemos pasado del BASIC interpretado al código máquina y normalmente se dice que la velocidad se debe multiplicar al menos por 100 (hablaríamos de un incremento del 10000%). Sin embargo, sólo hemos ganado un 50%. El motivo de tan "pobre" ganancia es que las instrucciones de 8BP ya hacen todo el trabajo duro y en realidad el compilador sólo traduce a código máquina la parte menos pesada, la lógica del juego.

## 4 Pasos que debes dar para hacer un juego

### 4.1 Estructura en directorios de tu proyecto

Lo más recomendable a la hora de programar tu juego es que estructures los diferentes ficheros en 7 carpetas, en función del tipo de fichero del que se trata.

Es perfectamente posible meterlo todo en el mismo directorio y trabajar sin carpetas, pero es mas “limpio” hacerlo como te voy a presentar a continuación



Fig. 9 estructura de directorios

- **ASM:** aquí meterás archivos de texto escritos en ensamblador (.asm), como es la propia librería 8BP, los sprites generados con el editor de sprites SPEDIT, y algunos ficheros auxiliares.
- **BASIC:** aquí meterás tu juego y utilidades como el SPEDIT y el cargador (Loader).
- **Dsk:** aquí meterás el archivo .dsk listo para ejecutar en un Amstrad CPC. En su interior deberás ubicar 5 ficheros de los que hablaremos en el siguiente apartado
- **Music:** con el secuenciador de música WYZtracker, podrás crear tus canciones y almacenarlas en formato .wyz en este directorio. Una vez que las “exportes”, se generarán un archivo .asm que tendrás que guardar en la carpeta ASM y un archivo binario que también almacenarás en la carpeta ASM (también los puedes dejar en esta carpeta, con tal de que los referencies adecuadamente en el fichero make\_musica.asm del directorio ASM).
- **Output\_spedit:** en esta carpeta puedes almacenar el fichero de texto que genera spedit. SPEDIT lo que hace es mandar a la impresora los sprites en formato ensamblador y el emulador winape puede recoger la salida de la impresora del Amstrad en un fichero. Aquí lo ubicaremos
- **Tape:** aquí puedes almacenar el fichero .wav si deseas hacer una cinta para cargar en el Amstrad CPC464, o el fichero .cdt

### 4.2 Tu juego en 6 ficheros

En este apartado vamos a ver los pasos que debes dar. No es algo secuencial, puedes ir haciendo gráficos a medida que programas e igual ocurre con las músicas. No te preocupes si ahora no entiendes con precisión cada paso. A lo largo del libro irás comprendiendo exactamente lo que significan con precisión. Y al final tienes un apéndice con información detallada a este respecto.

Lo que de momento debes entender es que tu juego se debe componer de 6 ficheros: 4 ficheros binarios y 2 ficheros BASIC. Existe la posibilidad de compactar todos los ficheros binarios en uno solo, más adelante te explicaré como se hace esta simplificación.

Los 4 ficheros binarios son:

- Librería 8BP (es un fichero binario), incluye la tabla de atributos de sprites
- Fichero binario de música con las melodías e instrumentos de tu juego
- Fichero binario de imágenes de sprites, incluyendo la tabla de secuencias de animación y el alfabeto para el comando PRINTAT
- Opcionalmente, si tu juego usa scroll y posee un mapa del mundo predefinido o si usa layout y tiene un layout predefinido, necesitarás un fichero de mapa de ese mundo (500 bytes a partir de la dirección 42040). Yo recomiendo utilizar 1KB desde la 23000 hasta la 24000, para almacenar todas las fases (mapas) del juego, y cada vez que entras en una fase cargas en la dirección 42040 el mapa correspondiente haciendo PEEK y POKE, o bien usas |UMAP para actualizar el mapa. Es decir, que mi fichero de mapa lo construyo en la dirección 23000 y ocupa 1000 bytes, aunque es solo una recomendación.

Y dos ficheros BASIC:

- Cargador (carga la librería, música y sprites y por último tu juego). Si además deseas hacer una pantalla de presentación que se muestre mientras se carga el juego, será lo primero que cargue este cargador
- Programa BASIC (tu juego).

Para hacer estos 5 ficheros debes dar estos pasos

### PASO 1

Editar gráficos con SPEDIT

Ensamblar los gráficos con winape

Salvar los gráficos con el comando **SAVE “sprites.bin”, b, 33600, <tamaño>**

Si desconoces el tamaño, usa lo máximo (8440)

### PASO 2

Editar la música con WYZtracker

Ensamblar la música con winape. Las melodías se ensamblarán una detrás de otra, de modo que cada una comenzará en una dirección de memoria diferente que dependerá del tamaño que ocupen.

Salvar la música con el comando **SAVE “music.bin”, b, 32200, 1400**

### PASO 3

Re-ensamblar la librería 8BP, de modo que la parte de la librería que selecciona las melodías (el player wyz) pueda conocer en qué direcciones de memoria se han ensamblado (hay más dependencias, pero esa es una de ellas). Una vez re-ensamblada, tendrás que salvarla con el comando:

**SAVE “8BP.LIB”, b, 24000, 8200, &6b78**

Esta será una versión de la librería específica para tu juego. Por ejemplo, el comando |MUSIC,3,6 hará sonar la melodía número 3 que tú mismo has compuesto. La melodía número 3 puede ser completamente diferente en otro juego.

## PASO 4

Cargar todo con un loader , que deberás hacer en BASIC. Por ejemplo:

```
10 MEMORY 23999
15 LOAD "!pant.scr",&c000: REM esto solo si tu juego tiene pantalla de carga
20 LOAD "!8bp.lib"
30 LOAD "!music.bin"
40 LOAD "!sprites.bin"
45 LOAD "!map.bin": REM esto solo si tu juego tiene un mapa
50 RUN "!tujuego.bas"
```

En el ejemplo que acabas de ver he puesto una carga de pantalla inicial en la dirección inicial de memoria de pantalla (&C000). Al final de este manual encontraras una de las muchas formas de hacer una pantalla de carga. Es algo opcional. Puedes hacer un juego con pantalla de carga o sin ella.

## PASO 5

Programar tu juego, el cual debe primeramente ejecutar la llamada para instalar los comandos RSX, es decir CALL &6b78. Y algo muy importante: no te olvides de incluir MEMORY 23999 al principio, para evitar que el BASIC en ejecución guarde variables por encima de la dirección 24000, que es donde comienza 8BP.

Tu juego lo puedes programar usando el editor de winape, mucho más versátil que el editor del AMSTRAD y sirve tanto para editar ensamblador (.asm) como para editar BASIC (.bas). El editor de winape es sensible a las palabras clave y las cambia de color automáticamente, facilitando la labor de programar. Tras escribir un programa BASIC hay que copiar/pegar en la ventana de CPC del winape. Para hacerlo más deprisa puedes activar la opción “High Speed” de winape durante el pegado, de ese modo ese proceso será inmediato.

Opcionalmente puedes compilar tu juego con una herramienta llamada fabacom y usar la versión compilada, pero no es necesario, ya que con 8BP tus juegos funcionarán muy rápido.

## PASO 6

Crear una cinta o un disco con tu juego

### 4.3 **Metodo abreviado: solo 3 ficheros**

Existe una forma mas sencilla de generar tu juego. Se trata de almacenar todos los binarios juntos en un solo fichero, así:

**SAVE “tujuego.bin”,b,24000, 18540, &6b78**

Siendo la logitud la dirección final menos la inicial. La dirección final incluyendo música y graficos y el mapa, es 42540, de modo que la longitud es 42540-24000 =18540. Si además quieres incluir el banco de estrellas (que por defecto las he pre-generado aleatoriamente) entonces la longitud son 80 bytes adicionales

**SAVE “tujuego.bin”,b,24000, 18620, &6b78**

Y el segundo fichero es tu juego basic

**SAVE “tujuego.bas”**

El cargador simplemente será:

```
10 MEMORY 23999
15 LOAD "!paint.scr",&c000: 'solo si tu juego tiene pantalla de carga
20 LOAD "tujuego.bin"
50 RUN "!tujuego.bas"
```

Este método “abreviado” es útil sobre todo si ocupas casi toda la memoria disponible para gráficos. En juegos de cassette cargar 18KB puede llevar un tiempo y si no usas los 8.5KB de graficos, quizás sea mejor que cargues por separado los distintos ficheros y así ahorres tiempo de carga. Por ejemplo, si usas solo 2KB de gráficos, con el método abreviado cargarías 8.5KB de graficos, es decir 6.5KB extra vacios. Esto en tiempo de carga en cinta pueden suponer casi dos minutos extra de tiempo. En disco (CPC 6128) da lo mismo porque no tarda nada en cargarlos.

## 4.4 Crear un disco o una cinta con tu juego

### 4.4.1 Hacer un disco

Para crear un nuevo disco desde winape hacemos

File->drive A-> new blank disk

Con ello te aparecerá una ventana de administración de archivos para que le des nombre al nuevo fichero .dsk

Una vez creado ya puedes guardar ficheros con el comando SAVE. Para borrar un archivo se utiliza el comando “|ERA” (abreviatura de ERASE), que solo existe en CPC 6128 como parte del sistema operativo “AMSDOS” (esto en CPC464 no existe pues funcionaba con cinta de cassette).

```
|ERA,"juego.*"
```

Y se borrarán

Para cargar el juego necesitas un cargador que cargue uno por uno los ficheros necesarios. Algo como:

```
10 MEMORY 23999
20 LOAD "!8bp.lib"
30 LOAD "!music.bin"
40 LOAD "!sprites.bin"
45 LOAD "!map.bin": REM esto solo si tu juego tiene un mapa
50 RUN "!tujuego.bas"
```

Para salvar cada uno de los ficheros debes usar el comando SAVE con los parámetros necesarios, por ejemplo, si te decides por los 4 ficheros binarios:

```
SAVE "LOADER.BAS"
SAVE "8BP.LIB", b, 24000, 8200, &6b78
SAVE "MUSIC.BIN", b, 32200, 1400
```

```
SAVE "SPRITES.BIN", b, 33600, 8440  
SAVE "MAP.BIN", b, 42040, 500  
SAVE "tujuego.BAS"
```

O bien, si te decides por usar un unico binario:

```
SAVE "LOADER.BAS"
```

Luego tienes estas dos opciones:

```
SAVE "tujuego.BIN", b, 24000, 18540, &6b78 (sin el banco de estrellas)  
SAVE "tujuego.BIN", b, 24000, 18620, &6b78 (con estrellas)
```

Y por ultimo:

```
SAVE "tujuego.BAS"
```

Si quieras grabar el .dsk en un disquete de 3.5" y conectarlo a una disquetera externa de tu AMSTRAD CPC 6128, necesitarás el programa CPCDiskXP, muy sencillo de usar. A partir de un .dsk puede grabar un disquete de 3.5" en doble densidad (no olvides tapar el agujero del disquete para "engaños" al PC).

#### 4.4.2 Hacer una cinta

Lo más importante al crear una cinta es guardar en ella los ficheros en el orden en el que van a ser cargados por el ordenador. Una cinta no es como un disco en el que puedes cargar cualquier fichero almacenado, sino que los ficheros se encuentran uno detrás de otro, por lo que debes poner especial cuidado en este punto.

Si tu cargador de juego es así:

```
10 MEMORY 23999  
20 LOAD "!8bp.lib"  
30 LOAD "!music.bin"  
40 LOAD "!sprites.bin"  
45 LOAD "!map.bin": REM esto solo si tu juego tiene un mapa  
50 RUN "!tujuego.bas"
```

O bien, si prefieres usar un unico binario:

```
10 MEMORY 23999  
20 LOAD "!tujuego.bin"  
50 RUN "!tujuego.bas"
```

Es muy importante la exclamación "!" para que el Amstrad no saque el mensaje "press play then any key" al ejecutar cada LOAD

Primero debes guardar el cargador (supongamos que se llama "loader.bas"), después el fichero "8BP.LIB", después "MUSIC.BIN", después "SPRITES.BIN", después "MAPA.BIN" y por último "tujuego.BAS".

Para crear un ".wav" o desde winape

file->tape->press record

En ese momento te saldrá un menú de administración de archivos para que podamos decidir qué nombre le damos al fichero “.wav”

Si estás en modo CPC 6128, entonces a continuación debes ejecutar desde BASIC

| **TAPE**

Y luego

**SPEED WRITE 1**

Con este comando lo que habremos hecho es decirle al AMSTRAD que grabe a 2000 baudios. Así la carga durará menos. Si no ejecutas ese comando, la grabación se realizará a 1000 baudios, más segura pero mucho más lenta

**SAVE "LOADER.BAS"**

Saldrá un mensaje indicando que presiones rec&play, y luego pulsas “ENTER”.

Después grabar todos y cada uno de los ficheros:

```
SAVE "8BP.LIB", b, 24000, 8200, &6b78
SAVE "MUSIC.BIN", b, 32200, 1400
SAVE "SPRITES.BIN", b, 33600, 8440
SAVE "MAP.BIN", b, 42040, 500
SAVE "tujuego.BAS"
```

O bien, si prefieres un único binario:

```
SAVE "tujuego.BIN",b,24000,18620, &6b78
SAVE "tujuego.BAS"
```

Por último, debemos hacer una última operación para que winape cierre el fichero.

file->remove tape

Tras hacer el remove tape, el fichero adquirirá su tamaño (si no lo haces puedes ver que en el disco de tu PC el fichero no crece y es debido a que no se ha volcado al disco)

Para cargar el juego, si estas en un CPC6128

| **TAPE**

RUN “”

Para volver a usar el disco

| **DISC**

Si quieras salvar una pantalla de carga consulta el apéndice I sobre organización de la memoria de vídeo, ahí te explico como se hace.

#### 4.4.3 Solución de problemas con LOAD y MEMORY

Antes de cargar un fichero BASIC, el Amstrad se asegura que va a tener espacio disponible para ejecutarlo. Puede que el programa BASIC solo use 1KB de variables, pero eso el Amstrad no lo sabe, de modo que es más conservador y exige que tengas 5KB extra adicionales vacíos en la memoria. Estos 5KB pueden parecer excesivos, pero el Amstrad no sabe a priori cuantas variables vas a declarar en tu programa y prefiere tener mucho espacio libre para almacenar variables que quedarse corto y que el programa falle.

Esto significa que si previamente has cargado uno o varios ficheros binarios (con graficos, librería 8BP o lo que sea) y has dejado libres 20 KB, entonces no podrás cargar un juego BASIC de 20 KB sino como mucho un juego de 15 KB. Pero no te preocupes, hay dos modos de solventarlo. Empecemos por el método “difícil”

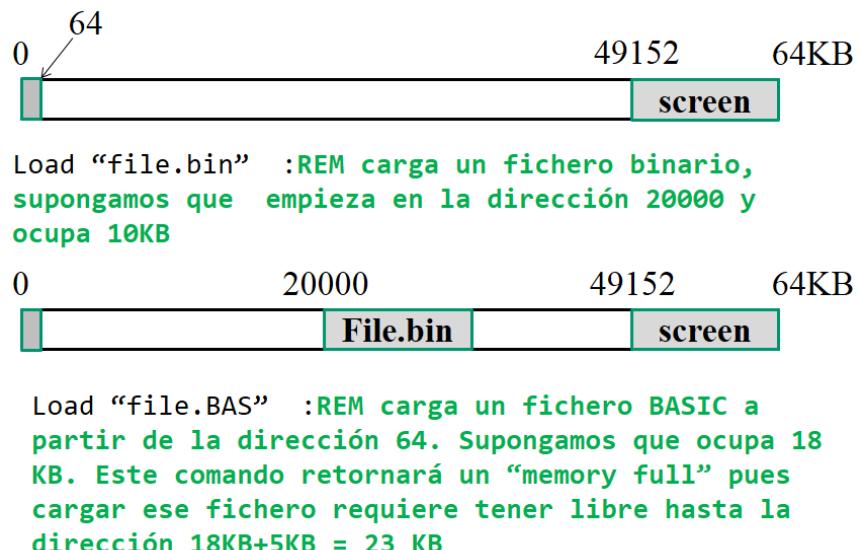


Fig. 10 El problema de LOAD

La solución consiste en cargar primero el programa BASIC. A continuación, ejecutar MEMORY 20000 y después cargar el fichero binario. Con esta estrategia quedan 2KB (20KB-18KB) para las variables del programa BASIC, lo cual puede ser mas que suficiente, dependiendo del programa. 2KB de variables es bastante.

En el caso en el que quieras que esto se haga automáticamente en la carga de un videojuego, lo puedes hacer así:

RUN "!mijuego.BAS"

Y en las primeras líneas del juego cargas el fichero binario, para lo cual necesitas una “marca” que permita cargar el binario una sola vez. Yo he elegido meter un numero concreto (144) en una posición de memoria (19999):

```
10 MEMORY 19998
20 IF PEEK (19999)=144 THEN 40:Rem el 144 es una marca. La primera vez
la marca no esta puesta y por lo tanto pasamos a la linea 30
30 LOAD "file.bin":'carga el fichero binario que comienza en la 20000
40 POKE 19999,144:'en vez de 144 podria haber usado cualquier numero
50 resto del juego
```

Esta técnica la he empleado en el videojuego “3D Racing one”, el cual ocupa casi 20 KB de BASIC y en la dirección 22000 empezaba un binario con datos de circuitos de carreras. Al haber solo 2KB de margen (menos de 5KB), tuve que aplicar esta solución.

Fíjate que en el ejemplo he puesto MEMORY 19998. Eso es para que el Basic no llegue a la 19999 que es donde he puesto la marca. Como el binario del ejemplo se carga a partir de la 20000, ni la carga del binario ni el BASIC van a afectar a la marca.

Otro de los problemas típicos relacionados con el error MEMORY FULL ocurre cuando cargamos un programa y en mitad de la ejecución lo paramos (pulsando ESC dos veces) Es posible que nos de memory full al tratar de hacer cosas como acceder al disco con un comando CAT.

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

Esto es debido a que nuestro programa BASIC puede consumir mucha memoria RAM de variables. Haberlo parado no significa que hayan desaparecido las variables del programa, de hecho, continúan existiendo e incluso puedes imprimirlas para ver su valor. Lo que haremos en este caso es simplemente ejecutar el comando CLEAR, el cual libera la memoria de dichas variables y a continuación nuestro comando CAT (o el que queramos).

```
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
```

Por ultimo, al principio te dije que había dos modos de resolver el problema. El modo fácil es hacer un “loader.bas” que cambie el memory. Supongamos que tienes datos binarios a partir de la dirección 20.000 y que tu programa ocupa 16KB, dejando menos de 5KB de margen. Lo único que tienes que hacer es:

```
10 MEMORY 19999
20 LOAD "!juego.bin": rem carga datos a partir de la 20000
30 CLEAR: MEMORY 25000 : rem asi le damos mas margen, por ejemplo.
40 RUN "!juego.bas": rem la primera linea de juego.bas debe ser
memory 19999
```

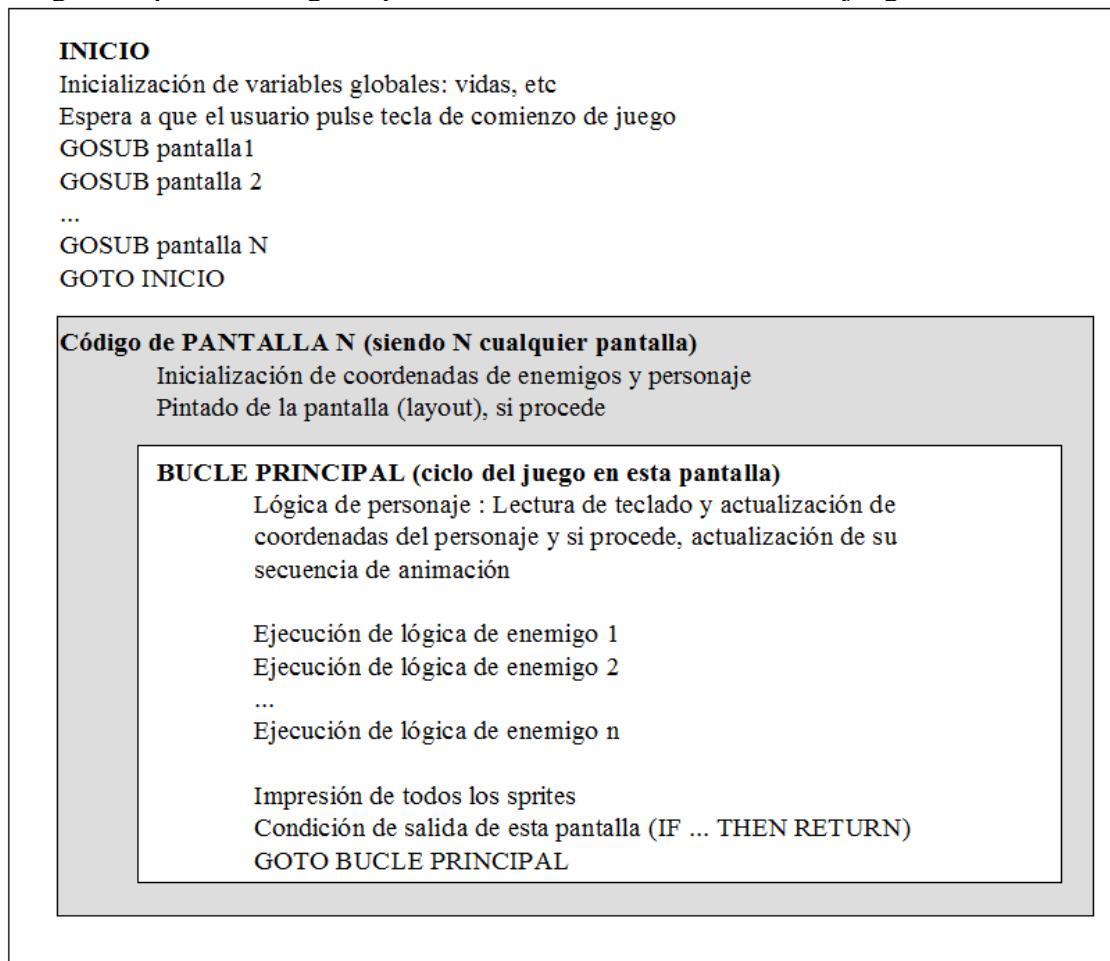
Este método es muy simple y 100% fiable porque aunque dejas desprotegidos parte de los datos binarios durante la carga del BASIC, lo primero que haces en BASIC es ejecutar el MEMORY, con lo que vuelve a quedar protegido

## 5 Ciclo de juego

Un videojuego de arcade, plataformas, aventuras, generalmente tiene un tipo de estructura similar, en la que unas ciertas operaciones se van a repetir cíclicamente en lo que denominaremos “ciclo de juego”.

En cada ciclo de juego actualizaremos posiciones de sprites e imprimiremos en pantalla los sprites, de modo que el número de ciclos de juego que se ejecutan por segundo equivale a los “fotogramas por segundo” (FPS) del juego.

El siguiente pseudo código esquematiza la estructura básica de un juego



*Fig. 11 Estructura básica de un juego*

Si la lógica de los enemigos es muy pesada por haber muchos enemigos o por ser muy compleja, esto consumirá más tiempo en cada ciclo del juego y por lo tanto el número de ciclos por segundo se verá reducido. Intenta no bajar de 10fps para que el juego mantenga un nivel de acción aceptable.

### 5.1 Como medir los FPS de tu ciclo de juego

Para saber si tu juego tiene un nivel de acción aceptable, nada mejor que jugar a él y si te gusta, pues estará bien. Sin embargo, quizás quieras medir exactamente cuántos frames por segundo es capaz de generar tu videojuego, porque así puedes tomar decisiones en la lógica de tu programa y medir cuánto perjudican o benefician esas decisiones de programación.

Lo que haremos para medir es simplemente tomar nota del instante de tiempo antes de empezar el primer ciclo de juego, en el principio del “código de la pantalla N”. Luego tomaremos nota del tiempo tras unos cuantos ciclos de juego y haremos una sencilla división. Vamos a verlo paso a paso:

**A=TIME : rem esta línea almacena en la variable A el tiempo en 1/300 fracciones de segundo**

El número que se va a almacenar en A puede ser un número muy grande, de hecho, puede ser mayor que lo que es capaz de almacenar una variable entera como es “A”. Para que la asignación no produzca error, es conveniente resetear el temporizador del AMSTRAD, que se inicia cada vez que arrancamos la maquina. Para resetearlo, antes de asignar la variable “A”, simplemente ejecuta:

**En un CPC 6128**

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

**En un CPC 464**

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

Con ello habrás puesto a cero las direcciones de memoria donde el AMSTRAD almacena el temporizador.

A continuación, ejecutamos tantos ciclos de juego como queramos, y controlamos en que ciclo estamos con la variable “ciclo”, que incrementaremos en una unidad en cada ciclo. Tras la salida de esa fase o pantalla, ejecutamos

**FPS= ciclo \* 300/ (TIME - A)**

Y ya tenemos los FPS de nuestro juego. Te lo pongo todo en orden a continuación:

**Rem suponemos que estamos en un 6128**

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

**A=TIME**

**<aquí va el programa que ejecuta tu ciclo de juego, incluyendo ciclo=ciclo+1 >**

**Llegaremos aquí tras la condición de salida de la pantalla**

**FPS= ciclo \* 300/ (TIME - A)**

**PRINT "FPS =" ;FPS**

Para que quede claro: El tiempo transcurrido desde que empezó el programa hasta que ha terminado es TIME-A expresado en 1/300 fracciones de segundo. Para pasarlo a segundos hay que dividirlo por 300

**Segundos= (TIME -A) /300**

Si en dichos segundos se han ejecutado n ciclos (por ejemplo), entonces un ciclo ha tardado: **Tc= 300\*(TIME-A)/n**

Y el numero de ciclos que se pueden ejecutar en un segundo (los FPS) es la inversa, es decir

**FPS = 1/Tc = n\*300/(TIME-A)**

## 6 Sprites

### 6.1 Editar sprites con spedit y ensamblarlos

Spedit (Simple Sprite Editor) es una herramienta que te va a permitir crear tus imágenes de personajes y enemigos y usarlos en tus programas BASIC

Spedit está hecha en BASIC, y es muy sencilla, de modo que puedes modificarla para que haga cosas que no están contempladas y te interesen. Se ejecuta en el Amstrad CPC, aunque está pensada para que la utilices desde el emulador winape.

Lo primero que debes hacer es configurar winape para que la salida de la impresora la saque a un fichero. En este ejemplo he puesto la salida de la impresora al fichero printer5.txt

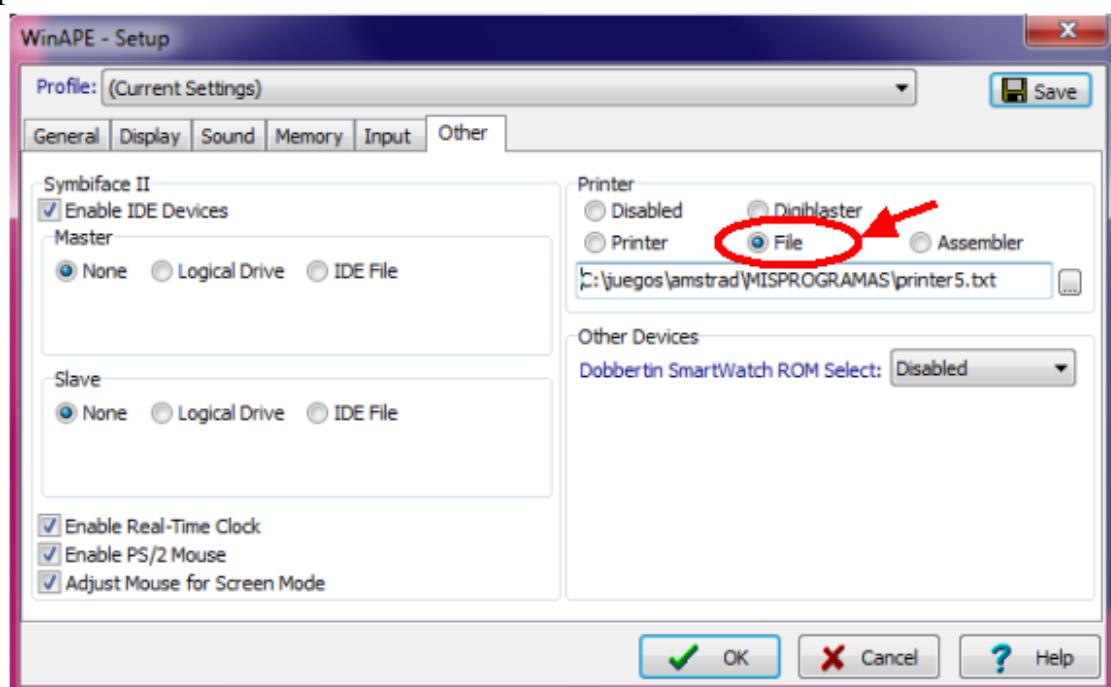


Fig. 12 Redirección de la impresora del CPC a un fichero con Winape

Cuando ejecutes SPEDIT te aparecerá el siguiente menú, donde puedes elegir si vas a usar la paleta por defecto o bien una tuya que quieras definir. Si decides definir tu propia paleta, deberás reprogramar las líneas de BASIC donde se define la paleta alternativa, que es una subrutina a la que se invoca con GOSUB cuando pulsas “2” en la respuesta a la pregunta sobre si quieres usar la paleta por defecto.

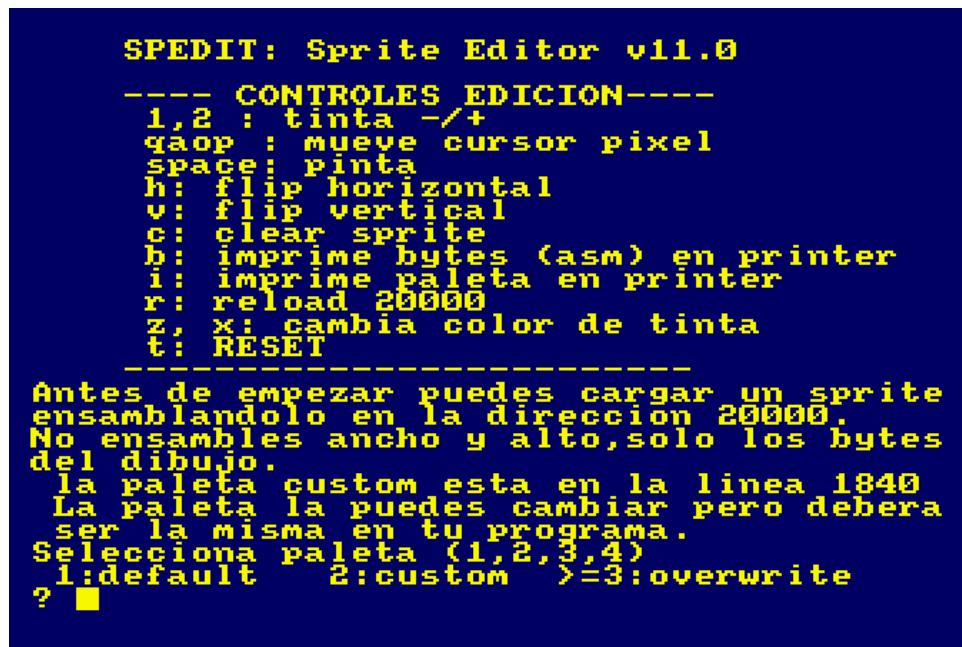


Fig. 13 Pantalla inicial de SPEDIT

Suponiendo que eliges usar la paleta por defecto, la herramienta se pone en mode 0 y te permite editar dibujos, con la ayuda en la pantalla. Manejas un píxel que parpadea y en la parte inferior se muestra las coordenadas donde te encuentras y el valor del byte en el que te encuentras.



Fig. 14 Pantalla de edición de SPEDIT

SPEDIT te permite “espejar” tu imagen para hacer el mismo muñeco caminando hacia la izquierda sin esfuerzo, basta con pulsar H (flip horizontal) y lo mismo se puede hacer en vertical. También te permite “espejar la imagen” respecto de un eje imaginario situado en el centro del personaje, tanto en vertical como en horizontal. Esto es muy útil para personajes simétricos o casi simétricos, donde una ayuda al dibujarlo siempre viene bien.



*Fig. 15 sprites simétricos con SPEDIT*

Desde la versión 11 de SPEDIT, el modo 1 de AMSTRAD está soportado, de modo que puedes editar sprites en mode 1 sin problema, y usar también el mecanismo de espejado.



*Fig. 16 edición de sprites en MODE 1 con SPEDIT*

Una vez que has definido tu muñeco, para extraer el código ensamblador deberás pulsar la “b”. Esto mandará a la impresora (al fichero que hayamos definido como salida) un texto como el siguiente, al que puedes añadir un nombre, yo le he llamado “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 -----

```



Fíjate como he dejado siempre un byte a la izquierda a cero. Lo he hecho para que, al mover el soldado hacia la derecha, se “borre a sí mismo”, ya que de lo contrario dejaría un rastro, “manchando” la pantalla mientras avanza.

*Fig. 17 Soldado en formato .asm*

Una vez que has hecho el primer fotograma de tu soldado puedes dejar el trabajo y continuar otro día. Para partir del soldado que has dibujado y continuar retocándolo o bien modificarlo para construir otro fotograma, puedes ensamblar el soldado en la dirección **20000**, quitando el ancho y el alto. Una vez ensamblado desde winape, le dices a SPEDIT que vas a editar un sprite del mismo tamaño y una vez estés en la pantalla de edición pulsas “r” (reload). El sprite se cargará desde la dirección **20000**, que es donde lo has “ensamblado”.

Gran parte del atractivo de un juego son sus sprites. No escatimes tiempo en esto, hazlo despacio y con gusto y tu juego parecerá mucho mejor.

```

.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 -----

```

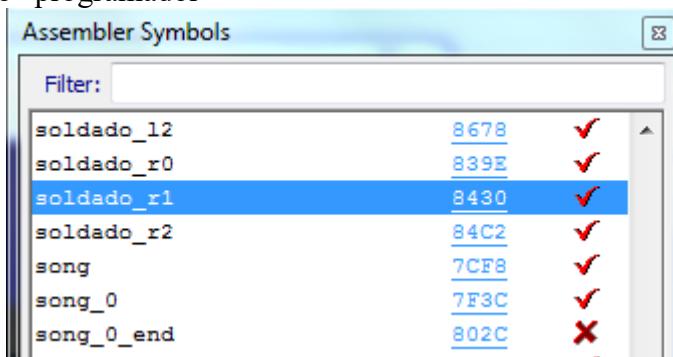
Con esto ya sabes lo que significa “ensamblar” un sprite. Es simplemente meter los bytes de datos que lo constituyen en direcciones de memoria consecutivas, en este caso comenzando por la **20000**

SPEDIT ocupa muy poca memoria y esa dirección está muy lejos del programa de modo que no hay problema de que al ensamblarlo estemos dañando” el programa SPEDIT.

*Fig. 18 Ensamblado de gráficos*

Para saber en qué dirección de memoria se ha ensamblado cada imagen, utiliza desde el menú de winape: Assemble->symbols

Con ello veras una relación de las etiquetas que has definido, como “SOLDADO\_R1” y la dirección de memoria (en hexadecimal) a partir de la cual se ha ensamblado. Para transformar las direcciones de hexadecimal a decimal puedes usar la calculadora de windows en modo “programador”



*Fig. 19 detalle de lo que muestra symbols en Winape*

Una vez que hayas hecho las diferentes fases de animación de tu soldado, puedes agruparlas en una “secuencia” de animación. Las secuencias de animación son listas de imágenes y no se definen con SPEDIT. Con SPEDIT simplemente editas los

“fotogramas”. En un apartado posterior te explicaré como decirle a la librería 8BP qué conjunto de imágenes constituyen una secuencia de animación.

Las imágenes que vayas haciendo para tu juego ve guardándolas todas en un único fichero, que se titule “images\_mijuego.asm”, por ejemplo. Dicho fichero comienza con una lista de imágenes que puedes referenciar en los comandos 8BP en BASIC con un índice, con independencia de la dirección en la que se ensamblen, por ejemplo:

```
IMAGE_LIST  
; la primera imagen siempre se asignara al índice 16, la siguiente al  
17 y asi sucesivamente  
-----  
dw SOLDADO_R0; 16  
dw SOLDADO_R1; 17  
dw SOLDADO_R2; 18
```

Una vez que estén todas las imágenes hechas podrás ensamblar ese fichero en la dirección 33600 y lo salvarás en un fichero binario desde el Amstrad con el comando SAVE. Por ejemplo, si has hecho 2500 bytes de imágenes, tras ensamblarlas en 33600 ejecuta desde el BASIC del CPC en el emulador:

**SAVE “sprites.bin”, b, 33600,2500**

Con esto habrás salvado en disco tu fichero de imágenes, así como las descripciones de secuencias de animación de las que hablaremos más adelante. En este ejemplo he puesto una longitud de 2500 pero si has dibujado muchos sprites puede que tengas que poner hasta 8440. No te olvides de la letra “b”, que sirve para especificar que se trata de un fichero binario. Si no estás seguro de cuanto ocupan tus imágenes, pon lo máximo, que es 8540 bytes.

**MUY IMPORTANTE:** asegúrate de no exceder los 8440 bytes de gráficos. Para ello comprueba donde se ha ensamblado la etiqueta “\_END\_GRAPH”, la cual debe ser inferior a 42040 (ya que  $42040 - 33600 = 8440$  bytes). Si se ensambla en una dirección superior entonces estás “machacando” direcciones que necesita el interprete BASIC y el ordenador podrá bloquearse. En caso de necesitar más memoria para gráficos deberás ensamblar los gráficos “extra” en una zona de memoria no ocupada, por ejemplo, en la 22000, y usar en tu programa un MEMORY 21999, reduciendo la memoria disponible para el BASIC

En resumen, si quieres salvar todas las imágenes y las secuencias de animación simplemente ejecuta:

**SAVE “sprites.bin”, b, 33600,8440**

Y para cargar tus sprites en memoria RAM, simplemente ejecuta:

**LOAD “sprites.bin”**

Para evitar problemas con la longitud, yo recomiendo grabar la librería, graficos y música todo junto en un solo fichero binario

## 6.2 Sprite flipping

En muchas ocasiones necesitarás dibujar personajes que caminan en diferentes direcciones, con diferentes imágenes para cada caso. La imagen del sprite en dirección

izquierda será la imagen especular de la dirección derecha. Se pueden definir dos imágenes y almacenarlas en memoria, pero desde la V33, hay una forma de evitar el consumo de memoria RAM para estas imágenes. Se trata de las imágenes “flipeadas”.



*Fig. 20 ejemplo de imágenes flipeadas*

Una imagen flipeada es la imagen especular de otra imagen que se ha creado e incluido en el fichero de imágenes. Definiendo una imagen de esta manera, se evita tener que almacenarla. Para hacerlo, simplemente debes incluir una lista de imágenes flipeadas dentro del fichero de imágenes (al que normalmente llamo “`images_mygame.asm`”). Encontrarás al principio del fichero una sección delimitada por las etiquetas “`BEGIN FLIP IMAGES`” y “`END FLIP IMAGES`” destinada a este propósito.

```

;-----
; BEGIN_FLIP_IMAGES
; aqui pon las imagenes que se definen como otras existentes pero
; flipeadas horizontalmente.
JOE_LEFT dw JOE_RIGHT; joe_left sera la version flipeada de joe_right

; los frames del soldado a la izquierda los defino como flipeados
SOLDADO_L0 dw SOLDADO_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

_END_FLIP_IMAGES
;-----
```

Las imágenes flipeadas las puedes usar igual que si fuesen imágenes normales. Mas adelante encontrarás como crear secuencias de animación, las cuales puedes construir con imágenes flipeadas y no flipeadas. A todos los efectos, es como si una imagen “flipeada” fuese real, aunque se trata de una imagen “virtual”, que no está almacenada y que al imprimirla se calcula como la imagen especular de otra que sí existe. Las imágenes flipeadas se soportan tanto en mode 0 como en mode 1.

El inconveniente de las imágenes flipeadas es que su impresión tiene mayor coste, concretamente consume un 1.8 veces el tiempo que consume una impresión normal, lo cual se podría traducir en una menor velocidad de tu juego. Si tu juego es un arcade (un “shoot’em up”) en el que necesitas la máxima velocidad, mi recomendación es no usar imágenes flipeadas masivamente. Sin embargo, en juegos de aventuras, de pasar pantallas, de laberintos, etc, es una excelente opción. De todos modos, prueba en tu arcade a usarlas pues si no hay muchas flipeadas a la vez, la velocidad resultante puede ser muy aceptable.

He hecho el flipping horizontal y no el vertical porque normalmente un personaje que camina a la izquierda es la imagen especular del mismo caminando hacia la derecha, mientras que si sube muestra la espalda y al bajar muestra el pecho y la cara. Por lo

tanto, el flipping vertical no es tan útil como el horizontal, y en aras de reducir el tamaño de 8BP, no lo he incluido entre sus capacidades.

**IMPORTANTE:** el flipping no es aplicable a las imágenes de tipo segmento que se pueden usar en el modo pseudo-3D de 8BP

### 6.3 Sprites con sobreescritura

Desde la versión v22 de 8BP es posible editar sprites transparentes, es decir, que pueden sobrevolar un fondo y lo restablecen al pasar. Para ello los sprites que disfrutan de esta posibilidad deben ser configurados con un “1” en el flag de sobreescritura del byte de estado (bit 6). En el siguiente apartado se detallará debidamente el byte de status. Veamos cómo se edita un sprite con esta capacidad con SPEDIT.

Muchos juegos utilizan una técnica llamada “doble buffer” para poder restablecer el fondo cuando un sprite se mueve por la pantalla. Se basa en tener una copia de la pantalla (o del área de juego) en otra zona de memoria, de modo que, aunque nuestros sprites destruyan el fondo, siempre podemos consultar en dicha área que había debajo y así restablecerlo. En realidad, ese es el principio básico, pero es algo más complejo. Se imprime en el doble buffer (también llamado “backbuffer”) y cuando ya está todo impreso, se vuelca a la pantalla o bien se hace comutar la dirección de comienzo de la memoria de video desde la dirección original de pantalla a la nueva, la del doble buffer. La conmutación es instantánea. Para construir el siguiente fotograma se usa la dirección de pantalla original donde ahora ya no está apuntando la memoria de video. Allí se construye el nuevo fotograma y se vuelve a comutar, alternativamente, en cada fotograma. Estas técnicas, aunque funcionan muy bien, tienen un par de desventajas para nuestros propósitos: llevan más tiempo de CPU y consumen mucha más memoria (hasta 16KB adicionales), dejándonos muy poca memoria para nuestro programa BASIC. Si un juego se desarrolla enteramente en ensamblador, esto no es tan grave porque 10KB de ensamblador dan para mucho, pero 10KB de BASIC es poco. Algunos videojuegos reducen el área de juego para no gastar tanta memoria, pero eso los hace algo mas pobres.

La solución adoptada en 8BP está inspirada en el programador Paul Shirley (autor de “misión Genocide”, pero es ligeramente diferente. Contaré directamente la de 8BP:



Fig. 21 Sprites con sobreescritura en 8BP

La idea consiste en que el fondo nunca es destruido por los sprites que pasan por encima, por lo que no es necesario guardarlos. Esta aparente “magia” tiene su lógica: consiste en “esconder” el color de fondo en el color del sprite que se pinta sobre él.

En el AMSTRAD un pixel de mode 0 es representado con 4 bits, por lo que son posibles hasta 16 tintas diferentes de una paleta de 27 colores. Pues bien, si usamos un bit para el color de fondo y 3 para los colores de los sprites, tendremos un total de 2 colores de fondo + 7 colores + 1 color para indicar transparencia = 9 colores en total. Esto nos va a permitir “esconder” el color de fondo en el color del sprite, aunque pagamos el precio de reducir el número de colores de 16 a tan sólo 9. Además, el fondo solo podrá ser de dos colores. Sin embargo, ciertos elementos ornamentales de la pantalla de juego pueden tener más color, pues los sprites no pasarán por encima (como las hojas de los árboles o el tejado del ejemplo siguiente), de modo que podemos conseguir cierta dosis de colorido en nuestro juego.

Para editar este tipo de sprites debemos usar una paleta adecuada, de 9 colores, donde para cada color de sprites se usan dos códigos binarios (los correspondientes al 0 y 1 del bit de fondo). En el SPEDIT hay dos paletas así definidas que puedes usar seleccionando la opción “3” o “4” al escoger la paleta. Se ha construido así:

```

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. 22 Paleta ejemplo de sobreescritura*

Como ves, tras el color 0 y 1, todos los colores se repiten dos veces. Tú puedes construir tu propia paleta de este modo. Puedes ayudarte consultando el apéndice de este manual dedicado a la paleta de color.

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

La técnica se podría resumir diciendo que en realidad el fondo nunca es destruido por los sprites, sino que se “esconde” en los propios sprites que se imprimen sobre el fondo

*Fig. 23 Mecanismo de sobreescritura en 8BP*

Con el editor SPEDIT puedes modificar la paleta a tu gusto sin necesidad de editar manualmente con comandos INK, y permite exportarla para copiarla en nuestros programas BASIC. La exportación se realiza mandando a la impresora los comandos INK que conforman la paleta (la impresora la redirigimos a un fichero desde winape). Disponemos de las teclas z/x para alterar la paleta y de la opción "i" para exportarla al fichero de salida. Este es un ejemplo de lo que exporta (es una paleta sin sobreescritura):

```
' ----- BEGIN PALETA -----
INK  0 , 1
INK  1 , 24
INK  2 , 20
INK  3 , 6
INK  4 , 26
INK  5 , 0
INK  6 , 2
INK  7 , 8
INK  8 , 10
INK  9 , 12
INK  10 , 14
INK  11 , 16
INK  12 , 18
INK  13 , 22
INK  14 , 0
INK  15 , 11
' ----- END PALETA -----
```

Los sprites que uses para construir los dibujos del fondo sólo podrán tener los colores 0 y 1 pero los sprites que uses para ornamentar, por donde no vayan a pasar los sprites en movimiento pueden usar los 9 colores.

También puedes aumentar el colorido de los decorados con elementos que sean sprites en lugar de fondos, como el caldero verde del ejemplo anterior. De este modo podrás tener resultados muy coloristas.

La tinta 0001 tiene un uso “especial”. Si editas un sprite que no use el flag de sobreescritura, la tinta 1 será simplemente un color. Pero si editas un sprite con flag de sobreescritura activo en su byte de status, al imprimirse se dejarán sin pintar esos píxeles, respetando lo que hubiese debajo. Eso permite que las colisiones entre sprites no sean “rectangulares”, sino que conserven la forma del sprite.

code	Significado
0000	Color 1 de fondo. Si un sprite lo usa y le activas el flag de sobreescritura, significa "transparencia", es decir, imprimir restableciendo el fondo
0001	Color 2 de fondo. Si un sprite lo usa y le activas el flag de sobreescritura, deja de significar un color para significar "no imprimir". Se respeta lo que haya en ese pixel, por ejemplo, un pixel coloreado por otro sprite anteriormente impreso con el que nos estamos solapando.
0010	color 1 de sprite
0011	color 2 de sprite
0100	color 3 de sprite
0101	color 4 de sprite

code	significado
0110	color 5 de sprite
0111	color 6 de sprite
1000	color 7 de sprite
1001	
1010	
1011	
1100	
1101	
1110	
1111	

9 colores en total:

- 2 de fondo
- 7 para sprites (en realidad 8 pero uno -000- significa transparencia)
- Los elementos ornamentales pueden usar los 9.

A continuación, voy a mostrarte un sprite donde he pintado de tinta 0001 lo que no se va a pintar, es decir, donde ni siquiera se va a restablecer el fondo, ya que con el resto de píxeles a 0000 ya es suficiente para borrar el rastro del sprite mientras se desplaza.



Fig. 24 sprite y paleta diseñados para sobreescritura

Como podrás imaginar, en el caso del caldero, al ser un sprite que no se desplaza y por lo tanto no se borra a sí mismo, todo su contorno está pintado con la tinta 0001. Ello permite colisiones perfectas, sin formas rectangulares que evidencian que en realidad los sprites son rectángulos. El resultado final es el que se muestra a continuación en una colisión múltiple.



Como habrás podido adivinar, la colisión además de ser perfecta, evidencia que los sprites han sido ordenados según su coordenada Y, de modo que el último en imprimirse es el ubicado en la posición más inferior. Esto se hace con un simple parámetro al imprimir los sprites con el comando |PRINTSPALL, que veremos más adelante.

*Fig. 25 Colisión múltiple, efecto de la tinta 0001*

Las operaciones de impresión con este mecanismo son muy rápidas, sin necesidad de definir lo que se conoce como “máscaras de sprites”. Las máscaras son mapas de bits del tamaño de un sprite que sirven para acelerar las operaciones de impresión. En este caso no son necesarias. La siguiente figura representa una típica máscara asociada a un sprite. Primero se suele hacer la operación AND entre el fondo y la máscara y después se hace el OR con el sprite. En 8BP es más rápido, pues el sprite no toca el bit destinado al fondo, de modo que la operación OR entre el fondo y el sprite respeta el fondo a la vez que pinta el sprite. Si no entiendes esto muy bien, no te preocupes, no es importante entenderlo pues no es necesario en 8BP.

sprite	mask		
0	2	2	0
2	3	3	2
0	2	2	0
0	2	2	0
0	0	0	0

				Metodo convencional:
				Se imprime
				Fondo AND mask OR sprite
				Metodo 8BP:
				Imprimimos
				Fondo OR sprite

*Fig. 26 En 8BP no son necesarias máscaras*

La impresión de sprites con el flag de sobreescritura activo es más costosa que la impresión sin sobreescritura. A pesar de no requerir máscara y ser muy rápida, esta impresión consume aproximadamente 1.6 veces el tiempo que consume la impresión de un sprite sin sobreescritura. Por ese motivo, úsala cuando sea necesario, y no la uses si tu juego no va a tener un dibujo de fondo que los sprites deban respetar. La combinación de sobreescritura y flipping es aun mas costosa (consume 2.2 veces el tiempo de una impresión normal sin sobreescritura ni flipping) de modo que tenlo en cuenta en tus juegos.

### 6.3.1 Quieres más colores y usar sobreescritura

Si has hecho tus primeras pruebas con sobreescritura y necesitas mas colores en tu videojuego, hay una forma de lograrlo, pero necesitas entender bien el método de 8BP.

Suponte que solo uno de tus sprites requiere sobreescritura y consume 3 colores. Eso significa que debes destinar 6 tintas a este Sprite. Sin embargo, si el resto de sprites no requieren sobreescritura, puedes imprimirlos sin sobreescritura y usar mas colores en ellos, es decir:

- 2 tintas para el fondo (2 colores)
- 6 tintas para el Sprite con sobreescrituras (3 colores)
- 8 tintas para los demás sprites (8 colores)

¡En este caso, en total podrás usar  $2+3+8 = 13$  colores!!!! Simplemente tienes que activar el flag de sobreescritura en el Sprite que lo necesita y dejarlo inactivo en los otros sprites. En los sprites que no usen sobreescritura podrás usar los 13 colores, en el Sprite con sobreescritura usarías 3 y en el fondo 2.

Otros ejemplos son posibles. Por ejemplo, si los sprites con sobreescritura necesitan 4 colores, entonces gastarán 8 tintas. Aparte tendremos 2 tintas para el fondo y las 6 tintas restantes pueden identificar 6 colores diferentes, es decir que podremos usar un total de 12 colores.

### 6.3.2 Sobreescritura con 4 colores de fondo

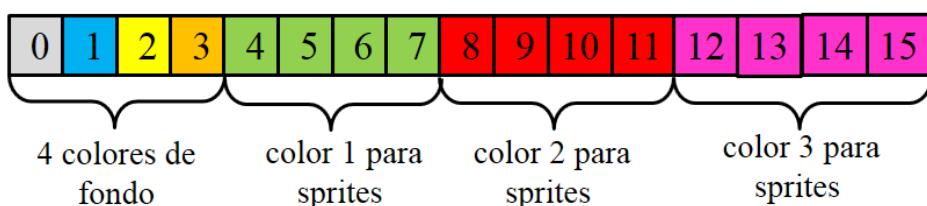
Desde la versión V33 es posible elegir el número de bits que se usan para el fondo mediante una invocación al comando |PRINTSP especificando el Sprite 32, el cual no existe. Se pueden elegir 1 o 2 bit para el fondo, permitiendo 2 y 4 colores de fondo respectivamente.

**|PRINTSP, 32, <num bits fondo>**

Ejemplos:

```
|PRINTSP, 32, 1 : ' con 1 bit de fondo tenemos 2 colores para el fondo
|PRINTSP, 32, 2 : ' con 2 bit de fondo tenemos 4 colores para el fondo
```

Una vez que invocamos este comando, la librería 8BP se configura para tener en cuenta el numero de bits que se van a usar como bits de fondo. Si configuramos 2 bit de fondo, nuestra paleta de color tendrá que ser coherente con esta circunstancia. A continuación, se muestra un ejemplo



*Fig. 27 Ejemplo de paleta con cuatro colores de fondo (2 bit de fondo)*

En este ejemplo tienes 4 colores de fondo y tres colores para los sprites. Al igual que cuando se usa 1 bit para el fondo, al definir tus sprites debes tener en cuenta que la tinta 0 significa transparencia y la 1 significa no reestablecer el fondo, permitiendo formas no rectangulares de sprites. En el siguiente ejemplo los 3 colores escogidos para los sprites son el negro, el verde claro y el blanco.



*Fig. 28 ejemplo de juego con paleta con hasta cuatro colores de fondo (2 bit)*

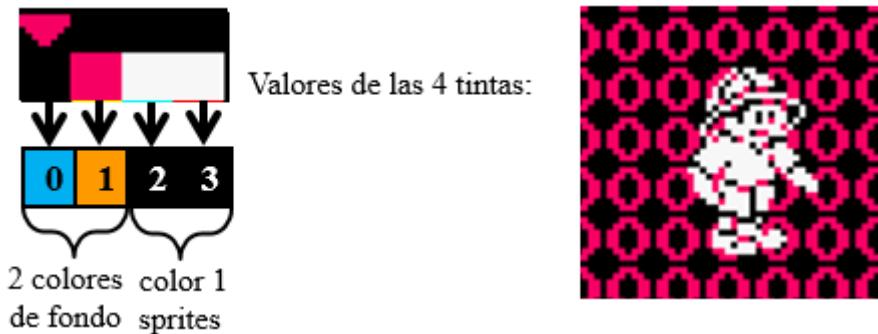
Aunque con dos bits para el fondo puedes conseguir decorados mas bonitos, la desventaja es que solo te quedan 3 colores para los sprites, mientras que si usas 1 solo bit para el fondo, tienes hasta 7 colores para tus sprites.



*Fig. 29 Ejemplo de paleta con dos colores de fondo (1 bit de fondo)*

### 6.3.3 Sobreescritura en MODE 1

Desde la versión V34 de 8BP, es posible usar sprites con sobreescritura en MODE 1. Aquí nos encontramos con una limitación muy fuerte porque, aunque tenemos dos colores para el fondo, tan solo disponemos de un color para los sprites.



*Fig. 30 Ejemplo de paleta de mode 1*

Se puede caer fácilmente en el error de pensar que los sprites disponen del color que sea y además, del negro. Eso no es así. El negro es un color más y como tal debe consumir dos tintas con este mecanismo. Solo disponemos de dos tintas para el Sprite y las hemos usado en el blanco (en este ejemplo). Como puedes apreciar, el personaje donde no es blanco es transparente y no negro.

A pesar de esta estricta limitación, si te esmeras puedes hacer unos sprites muy vistosos en MODE 1, y si en cada pantalla cambias los colores de fondo, y usas mezclas (entramados) de color en los marcadores del juego, puedes conseguir un resultado muy satisfactorio.

### 6.3.4 Cómo pintar sprites “por detrás” del fondo

El mismo mecanismo para imprimir sprites por encima del fondo puede servir para pintar por detrás del fondo.

Como ya hemos visto, para imprimir por delante del fondo usamos bits que no se usan en el fondo, de modo que, aunque reducimos el numero de colores, no dañamos el bit o los bits de fondo. Si usamos un bit para el fondo, tendremos que usar dos tintas para representar el mismo color de sprite: una con el bit de fondo a cero y otra con el bit de fondo a 1.

Ahora bien, si en lugar de asignar el mismo color a estas dos tintas, asignamos el mismo color que el fondo a la que tiene el bit de fondo a 1, entonces ante un solape del sprite con el fondo, se verá el color del fondo, dando la sensación de que el sprite pasa por detrás. Esto funciona tanto en MODE 0 como en MODE 1.

En el siguiente ejemplo se usa un bit para el fondo, el cual consiste en unas letras amarillas sobre un fondo negro. Los sprites son unas “monedas” que como se puede apreciar se han pintado aparentemente detrás del fondo.



*Fig. 31 Sprites impresos “detrás” de las letras*

### 6.4 Tabla de atributos de sprites

Los sprites se almacenan en una tabla que contiene un total de 32 sprites.

Cada entrada de la tabla contiene todos los atributos del sprite y ocupa 16 bytes por razones de rendimiento, ya que 16 es múltiplo de 2 y ello permite acceder a cualquier sprite con una multiplicación muy poco costosa. La tabla se encuentra ubicada en la dirección de memoria 27000, de modo que se puede acceder desde BASIC con PEEK y POKE, aunque también disponemos de comandos RSX para manipular los datos de esta tabla, tales como |SETUPSP o |LOCATESP

Los sprites tienen un conjunto de parámetros, de los que el primero de ellos es el byte de flags de status. En este byte, cada bit representa un flag y cada flag significa una cosa, concretamente representan si el sprite se toma en consideración al ejecutar ciertas funciones. En la siguiente tabla se resume lo que ocurre si están activos (a “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. 32 flags en el byte de estado*

Para entender la potencia de estos flags vamos a ver algunos ejemplos:

- Bit 0: flag de impresión: nuestro personaje o las naves enemigas lo tendrán activado y en cada ciclo del juego invocaremos a |PRINTSPALL y se imprimirán todos a la vez
- bit 1: flag de colisión: una fruta o moneda por ejemplo pueden no tener flag de impresión, pero tener el de colisión. Este flag significa que un sprite “colisionador” se puede colisionar con el sprite que tenga este flag activo.
- bit 2: flag de animación automática: se tiene en cuenta en |ANIMALL. En el caso del personaje, recomiendo desactivarlo, ya que si me quedo quieto no hay que cambiar el fotograma.
- bit 3: flag de movimiento automático. Se mueve solo al invocar |AUTOALL teniendo en cuenta su velocidad. útil en meteoritos y guardias que van y vienen.
- bit 4: flag de movimiento relativo. Todos los sprites que tengan este flag se mueven a la vez al invocar “[MOVEALL, dy, dx]” muy útil en naves en formación y llegadas a planetas. También sirve para simular un scroll si dejas tu personaje en el centro y al pulsar los controles se desplazan casas o elementos de alrededor. Parecerá que es tu personaje el que avanza por un territorio.
- bit 5: flag de colisionador. Todos los sprites con este flag activo son considerados por la función |COLSPALL, a la hora de detectar su posible colisión con el resto de sprites.
- Bit 6: flag de sobreescritura: si este flag está activo, el sprite se podrá desplazar por encima de un fondo, restableciéndolo al pasar. Esta es una opción avanzada e implica el uso de una paleta de color especial, de 9 colores. La sobreescritura tiene este “precio”.
- Bit 7: flag de ruta: si este flag esta activo, el comando |ROUTEALL te permitirá mover un sprite cíclicamente a través de una trayectoria que tu definas, definida por una serie de segmentos. El comando |ROUTEALL sabe en qué segmento y posición se encuentra cada sprite y si llega a un cambio de segmento, modifica la velocidad del sprite de acuerdo a las condiciones del siguiente segmento. |ROUTEALL no mueve al sprite, solo modifica su velocidad. Para moverlo hay que usarlo conjuntamente con |AUTOALL.

Ejemplos de asignación del valor del byte de status:

Típico enemigo: un sprite que se debe imprimir en cada ciclo, con detección de colisión con otros sprites y animación debe tener:

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

Una casa que se desplaza al movernos: un sprite que se imprime en cada ciclo, pero sin detección de colisión con otros y movimiento relativo

$$\text{status} = 1(\text{bit 0}) + 0 (\text{bit1}) + 0 (\text{bit 2}) + 0 (\text{bit 3}) + 16 (\text{bit 4}) = 17 = \&x10001$$

Una fruta que nos da bonus: es un sprite que no se imprime en cada ciclo, pero tiene detección de colisión

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

Una nave que va a seguir una trayectoria predefinida. Va a requerir del flag de ruta, el de movimiento automático, el de animación, el de colisión y el de impresión. Esta vez te lo voy a poner en binario directamente. Como ves el bit 7 lo he puesto a 1, después hay 3 ceros porque no he puesto el de sobreescritura ni el de colisionador ni el de movimiento relativo y por último he puesto 4 flags activos correspondientes respectivamente a los de movimiento automático, animación, colisión e impresión

$$\text{status}=10001111$$

La tabla de atributos de sprites se compone de 32 entradas de 16 bytes cada una, comenzando en la dirección 27000.

El motivo de tener 16 bytes no es otro que el del rendimiento, ya que calcular la dirección del sprite N implica multiplicar por 16, lo cual, al ser un múltiplo de 2, se puede hacer con un desplazamiento. Esto es útil en operaciones que involucran un único sprite. Para operaciones que recorren la tabla de sprites (como |PRINTSPALL o |COLSP), internamente se recorre la tabla con un índice al que se le suma 16 para pasar de un sprite al siguiente. La suma es lo más rápido en ese caso.

Los atributos que tiene cada sprite son:

atributo	Byte	Longitud (bytes)	Significado
status	0	1	Byte que contiene los flags de status para las operaciones PRINTSPALL, COLSP, ANIMALL, AUTOALL , MOVERALL, COLSPALL y ROUTEALL
Y	1	2	Coordenada Y [-32768..32768] los valores correspondientes al interior de la pantalla son [0..199]
X	3	2	Coordenada X en bytes [-32768..32768] los valores correspondientes al interior de la pantalla son [0..79]
Vy	5	1	Paso a dar en el movimiento automático
Vx	6	1	Paso a dar en el movimiento automático
Secuencia	7	1	Identificador de la secuencia de animación [0..31]. Si no posee secuencia se debe asignar un cero
Fotograma	8	1	Numero de frame en la secuencia [0..7]
Imagen	9	2	Dirección de memoria donde está la imagen
Sprite anterior	10	2	Uso interno para el mecanismo de ordenación de sprites
Sprite siguiente	12	2	Uso interno para el mecanismo de ordenación de sprites
Ruta	15	1	Identificador de ruta que debe seguir el sprite

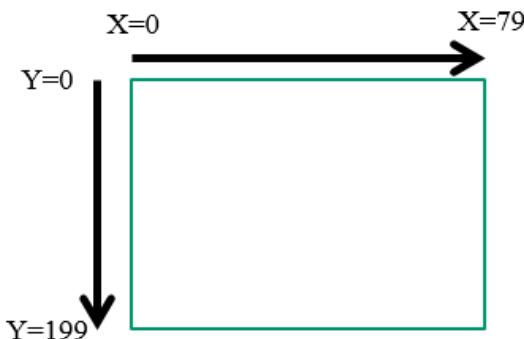
La dirección de memoria donde se almacenan las coordenadas de cada sprite se pueden calcular así:

**Dirección coordenada Y =** $27000 + 16*N + 1$

**Dirección coordenada X =** $27000 + 16*N + 3$

Accediendo con POKE a esas direcciones podemos modificar su valor, aunque también dispones de |LOCATESP.

La librería 8BP no usa “pixels” en la coordenada X, sino bytes, de modo que la coordenada X que cae dentro de la pantalla se encuentra en el rango [0..79]. La coordenada Y se representa en líneas de modo que el rango representable en pantalla es [0..200]. si ubicas un sprite fuera de esos rangos, pero parte del sprite se encuentra en la pantalla, la librería hará el “clipping” y pintará el trozo que se tenga que ver.



*Fig. 33 coordenadas de pantalla*

Las direcciones de los atributos de los 32 sprites se pueden manejar con PEEK y POKE, aunque la asignación de secuencia de animación y de ruta implican más operaciones y si quieras cambiarlas no basta con un POKE, sino que hay que usar |SETUPSP. Aquí tienes la lista de direcciones de todos los atributos de los 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

Tabla 3 Direcciones de atributos de los 32 sprites

El espacio ocupado por cada sprite en la tabla es de 16 bytes. Como ves las coordenadas X e Y son números de 2bytes. Los sprites aceptan coordenadas negativas por lo que puedes imprimir parcialmente un sprite en la pantalla, dando la sensación de que va entrando poco a poco. No podrás establecer coordenadas negativas con POKE, pero si podrás hacerlo con |LOCATESP y también con |POKE, que es una versión del comando POKE de BASIC pero que acepta números negativos (y números de 16 bit).

Es una buena práctica ubicar al personaje o nave espacial en la posición 31 (hay 32 sprites numerados del 0 al 31). Si tu nave tiene la posición 31 se imprimirá la última, encima del resto de sprites en caso de solape.

## 6.5 Impresión de todos los sprites y ordenados

En la librería 8BP dispone de un comando que imprime a la vez todos los sprites que tengan el flag de impresión activo. Se trata del comando |PRINTSPALL

Este comando tiene 4 parámetros, aunque solo necesitas rellenarlos la primera vez que lo invoques, pues las siguientes veces, se acordará de los parámetros, y solo deberás rellenarlos de nuevo si deseas cambiar alguno de ellos. Esto es útil porque el paso de parámetros consume mucho tiempo (te puedes ahorrar más de 1ms evitando el paso de parámetros).

Los parámetros son:

|PRINTSPALL, <ordenini>, <ordenfin>, <animar>, <sincronismo>

- **El parámetro de sincronismo** puede tomar los valores 0 o 1 e indica que se esperará a una interrupción de barrido de pantalla para imprimir. Si deseas velocidad no te lo recomiendo. Si deseas más suavidad, quizás sí.
- **El parámetro de animación** puede tomar los valores 0 o 1. En caso de estar activo, antes de imprimir cada sprite se comprobará su flag de animación en el byte de status y si lo tiene activo, entonces se cambiará de fotograma antes de imprimirlo. Es muy útil con los enemigos, pero con tu personaje normalmente no, pues solo desearás animarle al moverle y no en cada fotograma.
- **Los parámetros de orden (“ordenini”, “ordenfin”)** indican los sprites inicial y final que definen el grupo de sprites ordenados por coordenada “Y” que vamos a imprimir. Por ejemplo, si asignamos los valores 0,0 entonces se imprimirán secuencialmente desde el sprite 0 hasta el sprite 31. Si asignamos 0,8 se imprimirán del 0 al 8 ordenados (9 sprites) y del 10 al 31 de modo secuencial. Si ponemos un 0,31 se imprimirán todos los sprites ordenados. Si ponemos un 10,20 se imprimirán secuencialmente los sprites del 0 al 9, luego se imprimirán ordenados del 10 al 20 y finalmente se imprimirán secuencialmente del 21 al 31

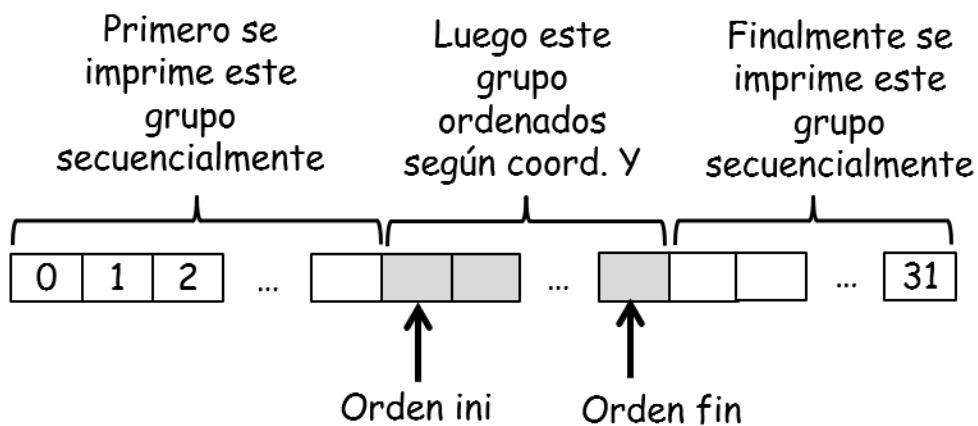


Fig. 34 Grupos de sprites secuenciales y ordenados

El ordenamiento es muy útil para hacer juegos tipo “Renegade” o “Golden AXE”, donde es necesario dar un efecto de profundidad. El ordenamiento se aprecia cuando hay solapamientos entre sprites.



Fig. 35 Efecto del ordenamiento de sprites

```
|PRINTSPALL, 0,0,1,0 : imprime de forma secuencial todos los sprites  
|PRINTSPALL, 0,31,1,0 : imprime de forma ordenada todos los sprites  
|PRINTSPALL, 0,7,1,0: imprime 8 ordenados y el resto secuencial  
|PRINTSPALL, 16,24,1,0: 16 secuenciales, 9 ordenados y 7 secuenciales
```

Si el parámetro “ordenini” se omite, se considera el ultimo valor asignado, o bien cero si nunca se ha asignado un valor. Además, si vas a modificar alguno de los dos parámetros de ordenamiento, conviene primero ejecutar PRINTSPALL,0,0,0,0 para que primero se reordenen los sprites secuencialmente antes de ordenarlos con una nueva configuración.

Imprimir de forma ordenada es más costoso computacionalmente que imprimir de forma secuencial. Si solo tienes 5 sprites que deben ser ordenados, pasa por ejemplo un 0,4 como parámetros de ordenamiento, no pases un 0,31. Ordenar todos los sprites lleva unos 2.5 ms pero si ordenas solo 5 te puedes ahorrar 2ms. Quizás tengas muchos sprites y no merezca la pena ordenar algunos, como los disparos o sprites que sabes que no se van a solapar.

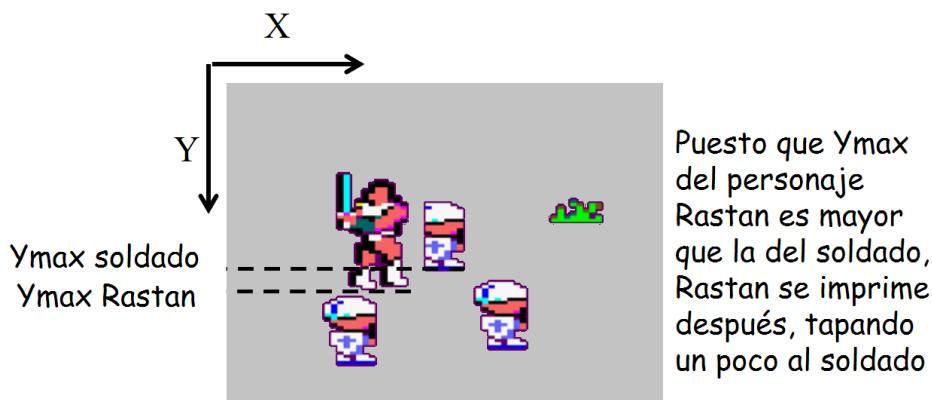
El algoritmo que se usa para ordenar los sprites es una variante del algoritmo conocido como “burbuja”. Aunque encontrarás en la literatura que el algoritmo llamado “burbuja” es muy poco eficiente, eso lo dicen los que hablan pensando en una lista de números aleatorios a ordenar. Nos encontramos ante un caso donde normalmente los sprites están casi ordenados y de un fotograma a otro solo se desordena uno o dos sprites, no más, pues sus coordenadas evolucionan “suavemente”. Por ello, el algoritmo recorre la lista de sprites y cuando encuentra un par de sprites desordenados, les da la vuelta y deja de seguir ordenando. Es tremadamente rápido, y aunque solo sea capaz de ordenar un par de sprites cada vez, es ideal para este caso de uso. Solo en el caso de que haya 2 sprites desordenados y que además se estén solapando, habrá un fotograma en el que veamos uno de ellos imprimiéndose en desorden, pero quedará corregido en el siguiente fotograma. Es imperceptible.

Puede que alguna vez deseas que la ordenación sea completa en cada fotograma. Es decir, que no se ordene un par de sprites en cada invocación a PRINTSPALL, sino tener la seguridad de que todos estan ordenados. La librería 8BP te lo permite mediante sus cuatro modos de ordenamiento, que puedes establecer mediante la invocación del comando PRINTSPALL con un solo parámetro (basta con ejecutarlo una vez para fijar el modo de ordenación):

<b>PRINTSPALL,0</b> : ordenamiento parcial usando Ymin
<b>PRINTSPALL,1</b> : ordenamiento completo usando Ymin
<b>PRINTSPALL,2</b> : ordenamiento parcial usando Ymax
<b>PRINTSPALL,3</b> : ordenamiento completo usando Ymax

Los ordenamientos que usan Ymax se basan en la coordenada Y mayor de los sprites, es decir, donde se encuentran sus pies en lugar de su cabeza. Si los sprites son del mismo tamaño, un ordenamiento basado en Ymin te puede servir, pero si los sprites tienen diferente altura puede que desees ordenar según donde se encuentren los pies de cada personaje y para ello tendrás que usar el modo de ordenar 2 o el 3.

Los modos de ordenar con Ymax son mas lentos, aproximadamente 0.128 ms por sprite, de modo que úsalos cuando los necesites realmente.



*Fig. 36 Ordenamiento según Ymax*

La ordenación completa consume muy poco mas que la parcial (aproximadamente 0.3ms). Esto es debido a que los sprites apenas se desordenan de un fotograma al siguiente, pero incluso esos 0.3ms merece la pena ahorrarlos si es posible.

Recuerda que el comando PRINTSPALL tiene “memoria”, de modo que basta con invocar la primera vez con parámetros y a partir de ese momento podemos invocar a PRINTSPALL sin parámetros pues el comando “conserva” los valores de los parámetros con los que fue invocado y no hace falta pasárselos a menos que cambien. Esto permite ahorrar mas de 1ms, ya que el analizador sintáctico trabaja menos.

## 6.6 Colisiones entre sprites

Para comprobar si tu personaje o tu disparo han colisionado con otros sprites dispones del comando

```
|COLSP, <sprite_number>, @colision%
```

Donde sprite number es el sprite que quieras comprobar (tu personaje o tu disparo) y la variable “colision” es una variable entera que previamente ha tenido que ser definida, asignando un valor inicial, por ejemplo:

```
colision%=0
|COLSP, 1, @colision%
```

La variable “colision” se llenará con el primer identificador de sprite que se detecte que ha colisionado con tu sprite, aunque podría ocurrir una colisión múltiple, pero el comando solo te entrega un resultado.

Internamente la librería 8BP recorre los sprites colisionables desde el 31 hasta el 0 (los recorre en orden inverso), y si tienen el flag de colisión activo (bit 1 del byte de status) entonces se comprueba si colisiona con tu sprite. Si no hay colisión con ninguno, la variable colision% queda con valor 32. En caso de haberla retornará el número de sprite que esté colisionando con tu sprite. Si por ejemplo colisionan el 4 y el 12, la función retornará un 12 pues comprueba antes el 12 que el 4.

**¡Ni tu personaje ni tu disparo deben tener el flag de colisión de sprites activo, ya que de lo contrario siempre colisionarán...consigo mismos!. Es decir un sprite no puede tener los bits 1 y 5 activos a la vez**

La colisión entre sprites es una tarea costosa. Internamente la librería necesita calcular la intersección entre los rectángulos que contiene cada sprite para determinar si hay solape entre ellos. Para ahorrar cálculos, lo mejor es ubicar a los enemigos en posiciones consecutivas de sprites. Si por ejemplo los enemigos con los que podemos chocar son los sprites del 15 hasta el 25, podemos configurar la colisión para que sólo compruebe esos sprites. Para ello invocaremos la colisión sobre el sprite 32 que no existe. Eso le indicará a la librería 8BP que se trata de información de configuración para el comando, indicando el **rango de sprites colisionables que se va a explorar por cada colisionador:**

|COLSP, 32, <sprite inicial>, <sprite final>

Ejemplo:

|COLSP, 32, 15, 25

Esta optimización si bien no es muy significativa, lo empieza a ser cuando se invoca varias veces a COLSP o se usa el comando |COLSPALL que internamente invoca varias veces a COLSP.

Otra interesante optimización, capaz de ahorrar 1 milisegundo en cada invocación, es decirle al comando que siempre use la misma variable BASIC para dejar el resultado de la colisión. Para ello se lo indicaremos usando como sprite el 33, que tampoco existe

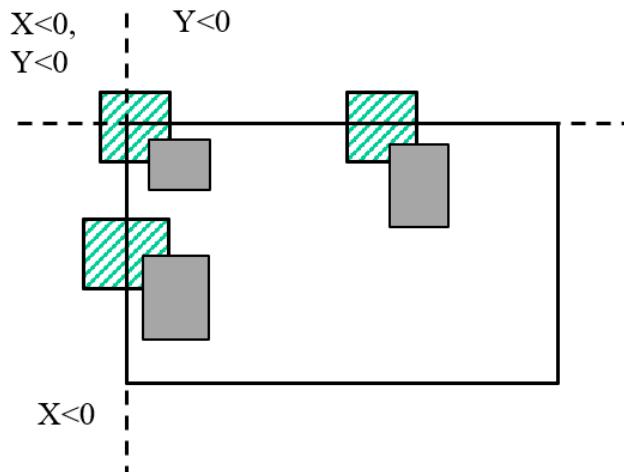
col%-0

|COLSP, 33, @col%

Una vez ejecutadas estas dos líneas, las siguientes invocaciones a COLSP, dejarán el resultado en la variable col, sin necesidad de indicarlo, por ejemplo:

|COLSP, 23 : REM esta invocación es equivalente a |COLSP, 23, @col%

**IMPORTANTE:** no se detectan colisiones con sprites que tengan alguna de sus dos coordenadas negativas con sprites que las tengan positivas



*Fig. 37 colisiones no detectables*

**IMPORTANTE:** la variable de colisión del comando COLSP no es la que se usa en el comando COLSPALL. Son variables diferentes (a menos que le pases a ambos comandos la misma variable para que actúen sobre ella)

## 6.7 Ajuste de la sensibilidad de la colisión de sprites

Es posible ajustar la sensibilidad del comando COLSP, decidiendo si el solape entre sprites debe ser de varios pixels o de uno solo, para considerar que ha habido colisión.

Para ello se puede configurar el número de pixels (pixels en dirección Y, bytes en dirección X) de solape necesario tanto en la dirección Y como en la dirección X, usando el comando COLSP y especificando el sprite 34 (que no existe)

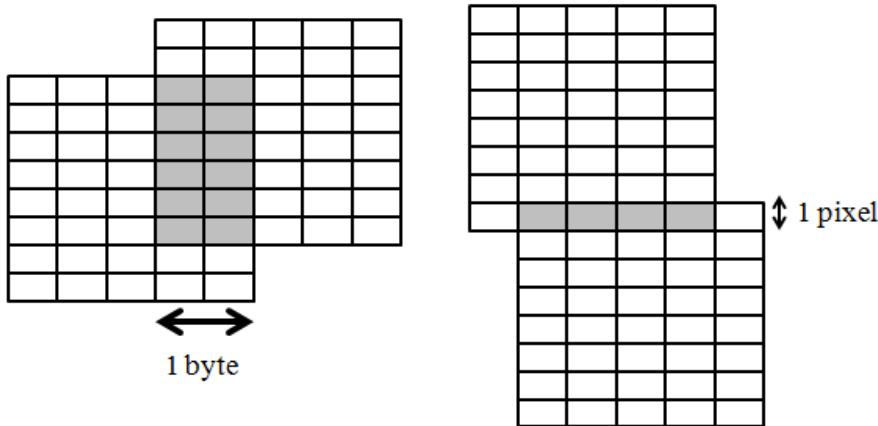
|COLSP, 34, <dy>, <dx>

La librería 8BP no usa “pixels” en la coordenada X, sino bytes, de modo que debes tener en cuenta que una colisión de 1 byte, en realidad son 2 pixels y esa es la mínima colisión posible cuando ajustas dx=0.

En la coordenada Y, la librería trabaja con líneas de modo que dy=0 significa una colisión de un solo pixel.

Una colisión estricta, útil para disparos sería aquella que no tolera ningún margen, considerando colisión en cuanto hay un mínimo solape entre sprites (1 pixel en dirección Y o un byte en dirección X)

|COLSP, 34, 0, 0: rem colision en cuanto hay un mínimo solape



*Fig. 38 Colisión estricta con COLSP, 34, 0, 0*

Sin embargo, si estamos haciendo un juego en MODE 0, donde los pixels son más anchos que altos, es quizás más adecuado dar algo de margen en Y y nada en X. Por ejemplo:

**|COLSP, 34, 2, 0 : rem colision con 3 pix en Y y 1 byte en X**

Mi recomendación es que, si hay disparos estrechos o pequeños, ajustes la colisión con (dy=1, dx=0) mientras que si solo hay personajes grandes puedes dejarla con mas margen (dy=2, dx=1). También debes considerar que, si tus sprites tienen un “margen” de borrado alrededor para desplazarse borrándose a sí mismos, dicho margen no debería formar parte de la consideración de colisión por lo que tiene sentido que tanto dy como dx no sean cero. En cualquier caso, es algo que decidirás en función del tipo de juego que hagas.

## 6.8 Quién colisiona y con quién: COLSPALL

Con la función COLSP que hemos visto hasta ahora, es posible la detección de colisión de un sprite con todos los demás. Sin embargo, si tenemos un disparo múltiple, donde por ejemplo nuestra nave puede disparar hasta 3 disparos simultáneamente, tendríamos que detectar la colisión de cada uno de ellos y adicionalmente la de nuestra nave, resultando en 4 invocaciones a COLSP.

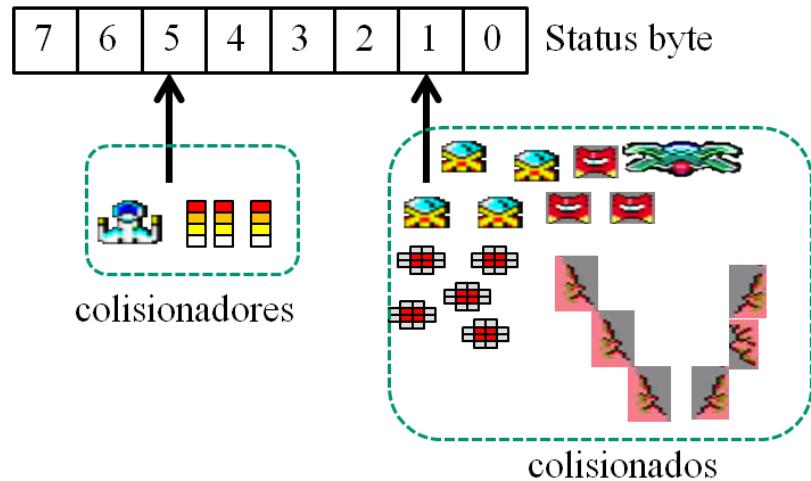
Debemos tener presente que cada invocación atraviesa la capa de análisis sintáctico, por lo que cuatro invocaciones resultan costosas. Para ello disponemos de un comando muy potente: |COLSPALL.

Esta función funciona en dos pasos: primero debemos especificar que variables van a almacenar el sprite colisionador y el colisionado. La siguiente instrucción la ejecutaremos una sola vez, y sirve para definir las variables sobre las obtendremos los resultados, las cuales deben existir previamente:

**|COLSPALL, @colisionador%, @colisionado%**

Y posteriormente, en cada ciclo de juego simplemente invitamos la función |COLSPALL sin parámetros.

La función va a considerar como sprites “colisionadores” aquellos que tengan el flag de colisionador a “1” en el byte de estado (es el bit 5), y como “colisionados” aquellos sprites que tengan a “1” el flag de colisión (bit 1) del byte de estado. Los sprites colisionadores deberán ser nuestra nave y nuestros disparos y los colisionados todos aquellos con los que nos podamos chocar: naves y disparos enemigos, montañas, etc. Como anteriormente dije, un sprite no debe tener ambos bits activos (=1) a la vez.



*Fig. 39 Colisionadores versus colisionados*

La función |COLSPALL empieza comprobando el sprite 31 (si es colisionador) y va descendiendo hasta el sprite 0, invocando internamente a |COLSP para cada sprite colisionador. Para cada colisionador, los sprites colisionables también se recorren en orden decreciente (desde 31 hasta 0). En cuanto detecta una colisión, interrumpe su ejecución y retorna el valor del colisionador y el colisionado. Por ello es importante que nuestra nave tenga un sprite superior a nuestros disparos. De ese modo, si nos alcanzan lo detectaremos, aunque hayamos alcanzado a un enemigo con un disparo en el mismo instante.

En cada ciclo de juego sólo se podrá detectar una colisión, pero es suficiente. No es una limitación importante que en cada fotograma solo pueda empezar a “explotar” un enemigo. Si, por ejemplo, tiras una granada y hay un grupo de 5 soldados afectados, cada soldado comenzará a morir en un fotograma distinto, y al cabo de 5 fotogramas estarán todos explotando. Usando |COLSPALL no explotarán todos a la vez, pero tu juego será más rápido y en un arcade es algo muy importante.

En caso de invocar a COLSPALL con un único parámetro,  
**|COLSPALL, <colisionador inicial>**

Se explorarán los colisionadores desde el colisionador indicado hasta el sprite cero, en orden descendente. De este modo si necesitas detectar mas de una colisión por ciclo de juego, podrás hacerlo invocando sucesivamente a COLSPALL hasta que la variable colisionador tome el valor 32

**IMPORTANTE:** al igual que ocurre con COLSP, no se detectan colisiones con sprites que tengan alguna de sus dos coordenadas negativas con sprites que las tengan positivas

### 6.8.1 Cómo programar un disparo múltiple usando COLSPALL

Lo primero que debes hacer es decidir el número de disparos activos que van a poder existir. Si decides que tu nave puede disparar 3 proyectiles a la vez, entonces debes reservar 3 identificadores de sprite para disparar. A continuación, debes ajustar el retardo entre un disparo y el siguiente para evitar que dos proyectiles salgan casi pegados si se dispara muy rápido. Esto lo puedes hacer definiendo una demora mínima entre disparo y disparo.

En el siguiente ejemplo he establecido una demora entre disparos de 10 ciclos de juego, para ello, al pulsar la barra espaciadora (tecla 47) se comprueba si han transcurrido 10 al menos 10 ciclos desde el ultimo disparo. De no ser así, no disparará.

```
130 'ciclo de juego -----
150 |AUTOALL,1:|PRINTSPALL,0,1,0
170 ' rutina movimiento personaje -----
172 IF INKEY(47)=0 THEN IF demora<ciclo-10 THEN demora=ciclo:disp= 1+
disp MOD 4:|LOCATESP,10+disp,PEEK(27001)+8,PEEK(27003):
|SETUPSP,10+disp,0,137: |SETUPSP,10+disp,15,3+dir
180 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 izquierda
193 ciclo=ciclo+1
310 GOTO 150
```

Y Ahora vamos con las colisiones y la ventaja de utilizar COLSPALL, mucho más rápido que invocar múltiples veces a COLSP. Las únicas recomendaciones importantes son:

- Que nuestro <sprite number> sea superior a nuestros disparos, para que |COLSPALL lo compruebe antes que a los disparos.
- Que tengamos configurado |COLSP para solo comprobar la lista de sprites que son enemigos y son necesarios de colisionar, mediante el uso de |COLSP 32, <inicio>, <fin>

Antes de comenzar el ciclo de juego definimos nuestras variables:

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

En el ciclo de juego pondremos:

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

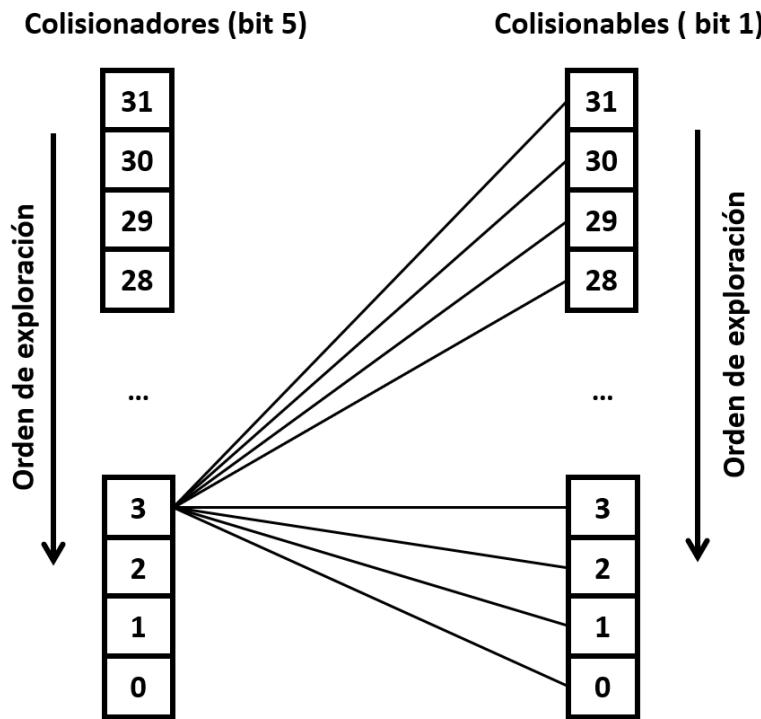
Con esta línea ya sabemos si hay colisión, pues entonces la variable “collider” será <32. Además, si es igual a 31 entonces es nuestra nave (nos han dado) y si no, entonces seguro que uno de nuestros disparos ha alcanzado a una nave enemiga e iremos a la rutina ubicada en la línea 770, donde se encontrará algo como esto:

```
769' --- rutina colision disparo -----
770 |SETUPSP, collider, 9, imgborrado:'asociamos imagen borrado al
disparo
772 |PRINTSP, collider: 'borramos el disparo
775 |SETUPSP, collider, 0, 0: 'desactivamos el disparo
777 if collided>=duros then return:'enemigo indestructible
778 ' la secuencia 4 es una secuencia de animación de "Muerte", una
explosion
780 |SETUPSP, collided, 7, 4:|SETUPSP, collided, 0, &x101: return
```

En resumen, con una sola invocación a COLSPALL ya sabemos quién ha colisionado (“collider”), y con quién ha colisionado (“collided”).

### 6.8.2 Quien colisiona cuando hay varios solapes

Es muy importante tener en cuenta que 8BP recorre los colisionadores desde el 31 hasta el 0 y para cada uno de ellos, recorre los colisionables desde el 31 hasta el 0. Debemos asociar los Sprite ID a nuestros sprites en función de como queremos que colisionen.



*Fig. 40 orden de comprobación de colisiones*

Si nuestro sprite (colisionador) colisiona con dos sprites en la misma área, podemos saber a priori con cual de ellos vamos a colisionar, lo cual es muy útil para ciertos juegos.

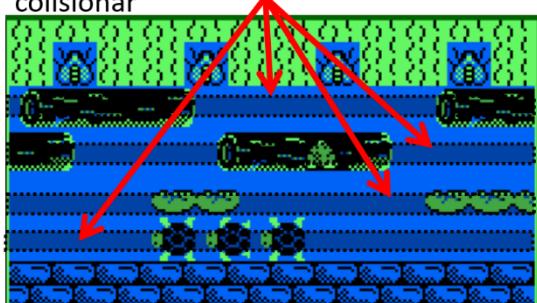
Supongamos el juego “frogger”, en el que una rana debe cruzar un rio saltando sobre los troncos. Si la colisión es sobre un tronco, no moriremos, pero si la colisión es sobre un río, moriremos.

Para programarlo podemos poner 4 ríos (4 sprites alargados inmóviles) y sobre ellos mover unos sprites que son troncos. Podriamos plantear el siguiente reparto:

- La rana es el sprite 31 (supongamos que es colisionador)
- Los troncos son los sprites 4 , 5 , 6, 7 (colisionables)
- Los ríos son los sprites 0, 1, 2 ,3 (colisionables)

Los ríos pueden tener el flag de impresión desactivado, de modo que pueden colisionar con la rana (flag de collided) sin necesidad de que se impriman. Es decir, les pondríamos status =2

Ríos (rectángulos largos) que no se imprimen pero que están ahí y pueden colisionar



Se detecta la colisión con el tronco antes que con el río por que el río (`collided`) tiene un Sprite ID menor que el tronco



*Fig. 41 en caso de solape nos interesa que colisione el tronco*

Pues bien, como los troncos y el río se solapan, en el momento en que la rana se sube a un tronco colisiona con ambos, pero se comprueba primero el tronco pues tiene un sprite ID mayor. El comando de colisión detectará únicamente la colisión con el tronco. Por el contrario, si la rana salta sobre el agua, entonces el comando de colisión detectará al río y tras evaluar desde BASIC la variable `collided` y ver que es un río, determinaríamos que nuestra rana debe morir

### 6.8.3 Uso avanzado del byte de status en colisiones

En ocasiones puedes querer que un enemigo no te mate al colisionar con tu personaje debido a que se encuentra en un estado especial, o porque simplemente se encuentra lejos en un juego que pretende simular que los enemigos se están acercando.

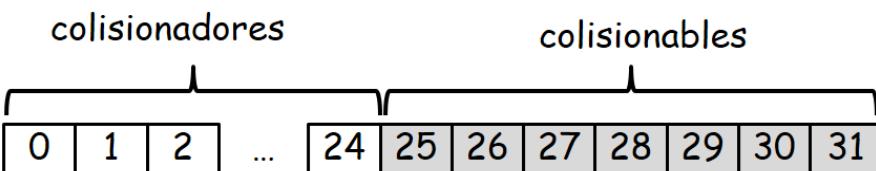
Algunas circunstancias especiales pueden requerir el uso de una “marca” en el sprite para indicar que, aunque haya habido colisión, no debe morir, o no debe matarte.

Para ello te puedes valer de los flags que no uses en el sprite y chequearlos en tu rutina de colisión.

Vamos a ver un ejemplo para un enemigo que queremos hacer inofensivo por encontrarse lejos (simulando 3D) o con poca energía pero que colisiona con nosotros.

Supongamos que tu personaje es `collided` y el enemigo es `collider`. Puedes forzar que solo se detecte colisión con sprites mayores que 25 y si el enemigo es el número 20 (por ejemplo) no podrá colisionar consigo mismo.

|COLSP,32,25,31



*Fig. 42 efecto de COLSP,32,25,31*

Como indicador de “inofensivo” vamos a usar el flag de `collided`, poniéndolo a 1. De modo que pondremos al enemigo (sprite 20) dicho flag a 1 en su byte de status

Ahora supongamos que el comando |COLSPALL detecta una colision, y deja el resultado en collider y collided

```
|COLSPALL
If collider<32 then GOSUB 100
<instrucciones>

100 rem rutina de colision
110 dir=27000 + collider*16 :rem dirección de byte status de collider
120 if PEEK (dir) and 2 THEN RETURN: rem inofensivo si bit collided=1
130 <ha colisionado un enemigo que no es inofensivo>
```

La instrucción “**if PEEK (dir) and 2**” es la que comprueba el bit collided ya que 2 en binario es 00000010, justo la posición de ese bit en el estado del sprite.

Cuando el enemigo deje de ser inofensivo simplemente ponemos su flag de collided a 0 y ante una nueva colision, nos matará.

Esta técnica es prefectamente valida usando cualquier otro flag que no se use.

## 6.9 Tabla de secuencias de animación

Las animaciones suelen componerse de un número par de fotogramas, aunque esto no es una regla estricta. Piensa por ejemplo en la animación simple de un personaje con solo dos fotogramas: piernas abiertas y cerradas. Son dos fotogramas. Ahora piensa en una animación mejorada, con una fase de movimiento intermedia. Esto supone crear la secuencia: cerradas-intermedia-abiertas-intermedia- y vuelta a empezar. Como ves es número par, son 4

Las secuencias de animación de 8BP son listas de 8 fotogramas, no pueden tener más, aunque siempre puedes hacer secuencias más cortas.

Los fotogramas de una secuencia de animación son las direcciones de memoria donde están ensambladas las imágenes de las que se componen, pudiendo ser diferentes en tamaño, aunque lo normal es que sean iguales. Si a mitad de la secuencia introduces un cero, el significado es que la secuencia ha terminado.

Aunque antes de la versión V33 existía un comando RSX llamado SETUPSQ para crear secuencias desde BASIC, lo he eliminado en la V33, debido a que su uso es mas complejo que definir secuencias desde el fichero sequences.asm y, de hecho, nunca he usado el comando SETUPSQ en ninguno de mis juegos, de modo que decidí sacrificarlo para ahorrar memoria.

Veamos un ejemplo de creacion de una secuencia en el fichero sequences.asm

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

Desde la versión V26 de 8BP, existe la posibilidad de incluir una lista de imágenes (sus etiquetas) en una lista llamada IMAGE\_LIST de tu fichero images\_tujuego.asm. Con ello puedes referenciar las imágenes desde BASIC con un índice en lugar de una dirección de memoria. Así no tendrás que consultar las direcciones de memoria cada vez que ensambles. Esto aplica a la instrucción SETUPSP, #, 9, <dirección>

El ejemplo muestra una secuencia de animación de 3 fotogramas diferentes, pero para que sea fluida antes de volver a empezar hay que pasar por el fotograma “intermedio”

otra vez ( fíjate que el segundo y el cuarto son iguales), de modo que al final son 4 fotogramas:



y vuelta a empezar

*Fig. 43 secuencia de animación*

Si quisieras hacer una secuencia de más de 8 fotogramas podrías simplemente encadenar dos secuencias seguidas y cuando el personaje llegase al último fotograma de la primera secuencia usar el comando |SETUPSP para asignarle la segunda secuencia

Las secuencias de animación se ensamblan a partir de la dirección 33500 y puedes definir hasta 31 secuencias de animación (a partir del número 32 no se consideran secuencias, sino “macrosecuencias”, que es otro concepto). Cada secuencia estará identificada por un número que se encontrará en el rango [1..31]. La secuencia cero no existe, se usa para indicar que un sprite no tiene secuencia.

Para asignar una secuencia a un Sprite usa el comando SETUPSP con parámetro 7:

|**SETUPSP, <sprite\_id>, 7, <sequence number>**

Cada secuencia almacena 8 direcciones de memoria correspondientes a los 8 fotogramas, esto son 16 bytes consumidos por cada secuencia.

Tu fichero de secuencias de animación se puede parecer a esto:

```
org 33500;
;=====
; hasta 31 secuencias de animacion
;=====
; debe ser una tabla fija y no variable
; cada secuencia contiene las direcciones de frames de animacion ciclica
; cada secuencia son 8 direcciones de memoria de imagen
; numero par porque las animaciones suelen ser un numero par
; un cero significa fin de secuencia, aunque siempre se gastan 8 words
; por secuencia
; al encontrar un cero se comienza de nuevo.
; si no hay cero, tras el frame 8 se comienza de nuevo
; si a un Sprite se le asigna la secuencia cero es que no tiene secuencia.
; empezamos desde la secuencia 1

;-----secuencias de animacion del personaje montoya-----
SEQUENCES_LIST
dw MONTOYA_R0,MONTOYA_R1,MONTOYA_R2,MONTOYA_R1,0,0,0,0 ;1
dw MONTOYA_UR0,MONTOYA_UR1,MONTOYA_UR2,MONTOYA_UR1,0,0,0,0 ;2
dw MONTOYA_U0,MONTOYA_U1,MONTOYA_U0,MONTOYA_U2,0,0,0,0 ;3
dw MONTOYA_UL0,MONTOYA_UL1,MONTOYA_UL2,MONTOYA_UL1,0,0,0,0 ;4
dw MONTOYA_L0,MONTOYA_L1,MONTOYA_L2,MONTOYA_L1,0,0,0,0 ;5
dw MONTOYA_DL0,MONTOYA_DL1,MONTOYA_DL2,MONTOYA_DL1,0,0,0,0 ;6
dw MONTOYA_D0,MONTOYA_D1,MONTOYA_D0,MONTOYA_D2,0,0,0,0 ;7
dw MONTOYA_DR0,MONTOYA_DR1,MONTOYA_DR2,MONTOYA_DR1,0,0,0,0 ;8

;-----secuencias de animacion del soldado -----
dw SOLDADO_R0,SOLDADO_R2,SOLDADO_R1,SOLDADO_R2,0,0,0,0 ;9
dw SOLDADO_L0,SOLDADO_L2,SOLDADO_L1,SOLDADO_L2,0,0,0,0 ;10
```

## 6.10 Secuencias de muerte

La librería 8BP te permite hacer “secuencias de muerte”, que son secuencias que al terminar de recorrerlas, el sprite pasa a estado inactivo. Esto se indica con un simple “1” como valor de la dirección de memoria del fotograma final. Estas secuencias son muy útiles para definir explosiones de enemigos que están animados con |ANIMA o |ANIMAALL. Tras alcanzarles con tu disparo, les puedes asociar una secuencia de animación de muerte y en los siguientes ciclos del juego pasarán por las distintas fases de animación de la explosión, y al llegar a la última pasarán a estado inactivo, no imprimiéndose más. Este paso a inactivo se hace automáticamente, de modo que lo que debes hacer es simplemente chequear la colisión de tu disparo con los enemigos y si colisiona con alguno le cambias el estado con SETUPSP para que no pueda colisionar más y le asignas la secuencia de animación de muerte, también con SETUPSP

Si usas una secuencia de muerte, no te olvides de que el último fotograma antes de encontrar el “1” sea uno completamente vacío, de modo que no quede ningún resto de la explosión.

Ejemplo de secuencia de muerte (fíjate que incluye un “1”):

```
dw EXPLOSION_1,EXPLOSION_2,EXPLOSION_3,1,0,0,0,0
```

Un efecto interesante es hacer pasar por varios fotogramas repetidamente antes de terminar con un fotograma negro que sirva para borrar

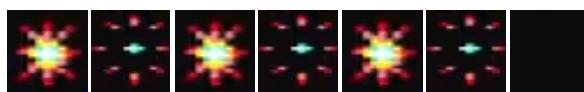


Fig. 44 Secuencia de muerte

Ahora el “1” aparece en octava posición:

```
dw EXPLOSION_1,EXPLOSION_2, EXPLOSION_1,EXPLOSION_2, EXPLOSION_1,EXPLOSION_2,  
EXPLOSION_3,1
```

## 6.11 Macrosecuencias de animación

Esta es una característica “avanzada” disponible a partir de la versión V25 de la librería 8BP. Una “macrosecuencia” es una secuencia formada por secuencias. Cada una de las secuencias de animación constituyentes es la animación que hay que efectuar en una dirección concreta. La dirección viene determinada por los atributos de velocidad del sprite, que están en la tabla de sprites. De este modo, cuando animemos a un sprite con |ANIMALL, automáticamente cambiará su secuencia de animación sin que tengamos que hacer nada (en realidad no hace falta invocar a |ANIMALL porque |PRINTSPALL ya lo hace internamente si se lo indicamos en un parámetro).

Las macrosecuencias se numeran comenzando en la 32. Es muy importante colocar las secuencias dentro de la macrosecuencia en el orden correcto, es decir, la primera secuencia debe ser para cuando el personaje está quieto, la siguiente para cuando va a la izquierda ( $Vx < 0$ ,  $Vy = 0$ ), la siguiente para la derecha ( $Vx > 0$ ,  $Vy = 0$ ), etc, siguiendo el siguiente orden (ten cuidado porque es fácil equivocarse):



Fig. 45 Orden de secuencias en una macrosecuencia

Si la secuencia asignada a la posición quieto es cero, entonces simplemente se anima con la última secuencia asignada.

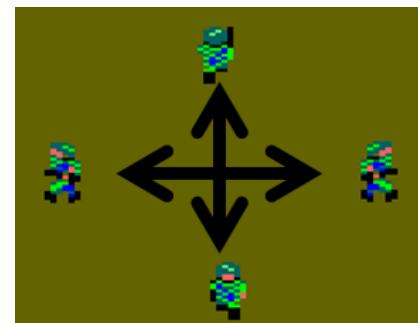
Las macrosecuencias hay que especificarlas en el fichero sequences\_tujuego.asm, del que a continuación tienes un ejemplo:

```
org 33500;
;=====secuencias de animacion=====
;-----secuencias de animacion -----
SEQUENCES_LIST
dw NAVE,0,0,0,0,0,0,0;1
dw JOE1,JOE2,0,0,0,0,0;2 UP JOE
dw JOE7,JOE8,0,0,0,0,0;3 DW JOE
dw JOE3,JOE4,0,0,0,0,0;4 R JOE
dw JOE5,JOE6,0,0,0,0,0;5 L JOE

_MACRO_SEQUENCES
;-----MACRO SECUENCIAS -----
; son grupos de secuencias, una para cada dirección. el significado es:
; still, left, right, up, up-left, up-right, down, down-left, down-right
; se numeran desde 32 en adelante
db 0,5,4,2,5,4,3,5,4;la secuencia 32 contiene las secuencias del soldado Joe
```

Con esa definición de secuencias podemos hacer un sencillo juego que permita mover a “joe” por la pantalla sin controlar su secuencia de animación. Le asignamos la secuencia 32 y alterando la velocidad, el comando |ANIMA (invocado desde dentro de |PRINTSPALL) se encarga de cambiarle la secuencia de animación si su velocidad denota un cambio de dirección. Eso sí, para mover al sprite necesitamos invocar a |AUTOALL, ya que al pulsar los controles no cambiamos sus coordenadas sino su velocidad y |AUTOALL actualizará las coordenadas del sprite de acuerdo a su velocidad.

```
10 MEMORY 23999
20 MODE 0:INK 0,12
30 ON BREAK GOSUB 280
40 CALL &6B78
50 DEFINT a-z
111 x=36:y=100
120 |SETUPSP,31,0,&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 'comienza ciclo de juego
199 vy=0:vx=0
200 IF INKEY(27)=0 THEN vx=1: GOTO 220
210 IF INKEY(34)=0 THEN vx=-1
220 IF INKEY(69)=0 THEN vy=2: GOTO 240
230 IF INKEY(67)=0 THEN vy=-2
240 |SETUPSP,31,5,vy,vx
```



<b>250  AUTOALL: PRINTSPALL</b>	
<b>270 GOTO 199</b>	
<b>280  MUSICOFF:MODE 1: INK 0,0:PEN 1</b>	

Fíjate que no he definido la secuencia para cuando el personaje no se mueve. En dicha posición he puesto un cero en la macrosecuencia. Eso significa que, si el personaje comienza quieto, no se sabe qué secuencia asignar pues no hay una “última” secuencia usada. Es por ello que asigno la secuencia 2 antes de asignar la 32, así me aseguro de que el personaje ya tiene una secuencia, aunque se encuentre quieto.

<b>130  SETUPSP, 31, 7, 2: SETUPSP, 31, 7, 32</b>
---

## 7 Tu primer juego sencillo

Ya tienes los conocimientos para intentar un primer paso en la creación de videojuegos. Para ello vamos a ver un sencillo ejemplo de un soldado al que vas a controlar, haciéndole caminar a derecha e izquierda por la pantalla

Supongamos que hemos editado a un soldado, gracias a SPEDIT. Y hemos construido sus secuencias de animación, las cuales han sido definidas en el fichero “sequences\_mygame.asm” y han quedado con el identificador 9 y 10 para las direcciones de movimiento derecha e izquierda respectivamente.

Las dos secuencias de animación las hemos creado desde el fichero de secuencias.asm

```
10 MEMORY 23999
20 MODE 0: DEFINT A-Z: CALL &6B78:' install RSX
25 call &bc02:'restaura paleta por defecto por si acaso
26 ink 0,0:'fondo negro
30 FOR j=0 TO 31:|SETUPSP,j,0,&X0:NEXT:'reset sprites
40 |SETLIMITS,12,80,0,186: ' establecemos los limites de la pantalla de juego
50 x=40:y=100: ' coordenadas del personaje
51|SETUPSP,0,0,1:' status del personaje
52|SETUPSP,0,7,9:'secuencia de animacion asignada al empezar
53|LOCATESP,0,y,x:'colocamos al sprite (sin imprimirla aun)

60 'ciclo de juego
70 gosub 100
80 |PRINTSPALL,0,0
90 goto 60

99 ' rutina movimiento personaje -----
100 IF INKEY(27)=0 THEN IF dir<>0 THEN |SETUPSP,0,7,9:dir=0:return ELSE
|ANIMA,0:x=x+1:GOTO 120
110 IF INKEY(34)=0 THEN IF dir<>1 THEN |SETUPSP,0,7,10:dir=1:return ELSE
|ANIMA,0:x=x-1
120 |LOCATESP,0,y,x
130 RETURN
```

Con este listado ya tienes un minijuego que te permite controlar un soldado y hacerlo correr horizontalmente. Fíjate que, si al caminar hacia la izquierda sobrepasas el valor mínimo del límite establecido con |SETLIMITS, se producirá el “clipping” del personaje, mostrándose tan solo la parte que queda dentro del área de juego permitida

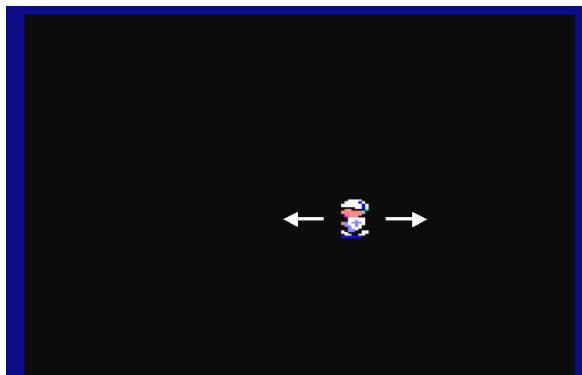


Fig. 46 Un sencillo juego

### 7.1 Ahora, a saltar! Boing, boing!!

En el ejemplo anterior nuestro personaje sólo se mueve de izquierda a derecha. Si queremos programar un salto, podemos hacerlo almacenando la trayectoria vertical en

un array de BASIC. Más adelante veremos una forma mejor de hacerlo (con rutas de 8BP) pero de momento servirá como ejemplo.

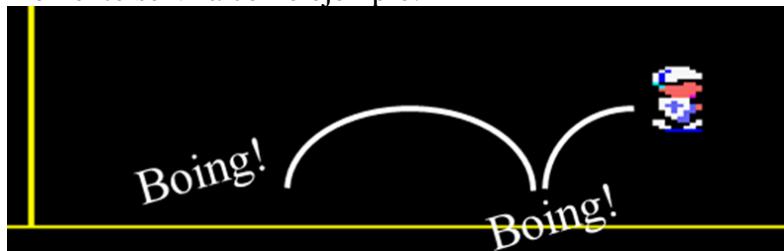


Fig. 47 nuestro personaje puede saltar

La trayectoria de salto se define para la coordenada Y. Primero sube 5 líneas de golpe, luego sube 4, luego sube 3, etc. hasta llegar a cero. En ese momento se invierte la dirección de movimiento y comenzamos a bajar 1 línea, luego 2, luego 3, etc. hasta 5. Para que cuando suba y baje el muñeco no deje rastro, tendremos que tener un dibujo del muñeco subiendo con 5 líneas negras por debajo para borrarse a sí mismo al subir y del mismo modo, otra imagen con 5 líneas negras por encima para bajar. En este caso son las imágenes 22 y 23 para saltar a la derecha y 24,25 para la izquierda

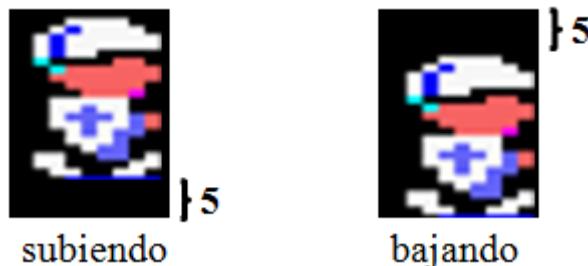


Fig. 48 Imagen de subida y de bajada

En el punto cenital del salto se debe cambiar la imagen de subida por la de bajada, pero antes debemos subir de golpe 5 líneas pues si comparas ambas imágenes te darás cuenta de que si pasas de una a otra directamente es como bajar 5 líneas.

```

10 MEMORY 23999
20 MODE 0: DEFINT A-Z: CALL &6B78:' install RSX
25 ON BREAK GOSUB 2800
30 CALL &BC02:'restaura paleta por defecto por si acaso
40 INK 0,0: 'fondo negro
50 FOR j=0 TO 31:|SETUPSP,j,0,&X0:NEXT:'reset sprites
80 |SETLIMITS,12,80,0,186: ' establecemos los limites de la pantalla
90 x=40:y=100: ' coordenadas del personaje
100 |SETUPSP,0,0,1: ' status del personaje
110 |SETUPSP,0,7,9: 'secuencia de animacion asignada al empezar
120 |LOCATESP,0,y,x: 'colocamos al sprite (sin imprimirlo aun)
121 DIM salto(24): ' datos del salto
122 for i=-5 to 5: k=k+1:salto(k)=i: k=k+1: salto(k)=i: next:
    salto(11)=-5: salto(23)=5
125 PLOT 1,150:DRAW 640,150: plot 92,150:draw 92,400: 'suelo y pared
126 |MUSIC,0,5: 'comienza a sonar la musica
130 'ciclo de juego -----
150 |LOCATESP,0,y,x:|PRINTSPALL,0,0
151 GOSUB 170
160 GOTO 130: ' fin ciclo juego

```

```

170 ' rutina movimiento personaje -----
171 IF jump =0 THEN IF INKEY(67)=0 THEN jump=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 IF INKEY(34)=0 THEN x=x-1:if jump=0 then IF dir<>1 THEN
|SETUPSP,0,7,10:dir=1:x=x+1:RETURN ELSE |ANIMA,0
210 if jump=0 then RETURN
260 'rutina de salto -----
270 IF jump=11 THEN |SETUPSP,0,9,DIR*2+23 ELSE IF jump=23 THEN
y=y+salto(jump):jump=0:|SETUPSP,0,7,DIR+9:return
280 y=y+salto(jump)
300 jump=jump+1
310 return
2800 |MUSICOFF:MODE 1: INK 0,0:PEN 1

```

Como ves en el listado, si pulsas la tecla “Q”, la variable jump se iguala a 1 y en ese momento la lógica del muñeco se complica pues requiere ejecutar una sentencia IF para cambiar la imagen al llegar al punto cenital y además es necesario actualizar la coordenada Y del muñeco y la variable jump.

Más adelante veremos cómo hacer esto mismo con una técnica más avanzada, usando “rutas” de sprites. **Las rutas programables de 8BP proporcionan un método más eficiente de hacer este tipo de cosas, por lo que verás como tu personaje salta mucho más rápido.** Las rutas te van a permitir ejecutar una trayectoria (un salto, un círculo, etc.) sin necesidad de controlar en cada instante las coordenadas. Y además podrás cambiar el estado de un sprite en mitad de una ruta, o cambiarle su imagen asociada, su secuencia o incluso cambiarle de ruta, concatenando diferentes rutas.



## 8 Juegos de pantallas: layout o “tile map”

### 8.1 Definición y Uso del layout

A menudo querrás que tus juegos consistan en un conjunto de pantallas donde el personaje deba recoger tesoros o esquivar enemigos en un laberinto. En esos casos se hace indispensable el uso de una matriz donde definas los bloques constituyentes de cada “laberinto” o también llamado “layout” de la pantalla. A veces a este concepto también se le llama “mapa de tiles” (un “tile” es la palabra en inglés para decir “azulejo”)

En la librería 8BP tienes un mecanismo sencillo para hacerlo, que además de proporciona una función de colisión para que compruebes si tu personaje se ha desplazado a una zona ocupada por un “ladrillo”. Este mecanismo se denomina “layout”. En 8BP un layout se define con una matriz de 20x25 “bloques” de 8x8 pixeles, los cuales pueden estar ocupados o no. Es decir, hay tantos bloques como tiene la pantalla de caracteres en mode 0.

Para imprimir un layout en la pantalla dispones del comando:

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

Esta rutina imprime una fila de sprites para construir el layout o “laberinto” de cada pantalla. La matriz o “mapa del layout” se almacena en una zona de la memoria que maneja 8BP de modo que cuando imprimes bloques en realidad no solo estás imprimiendo en la pantalla, sino que también estas rellenando el área de memoria que ocupa el layout (20x25 bytes) donde cada byte representa un bloque.

Las coordenadas y,x se pasan en formato caracteres, es decir

y toma valores [0,24]

x toma valores [0,19]

Los bloques que imprime la función |LAYOUT se construyen con cadenas de caracteres y cada carácter se corresponde con un sprite que debe existir. De este modo el bloque “Z” se corresponde con la imagen que tenga asignada el sprite 31. El bloque “Y” se corresponde con la imagen que tenga asignada el sprite 30, y así sucesivamente.

Los sprites a imprimir se definen con un string, cuyos caracteres (32 posibles) representan a uno de los sprites siguiendo esta sencilla regla, donde la única excepción es el espacio en blanco que representa la ausencia de sprite.

<b>Caracter</b>	<b>Sprite id</b>	<b>Codigo ASCII</b>
“ “	NINGUNO	32
“;”	0	59
“<”	1	60
“=”	2	61
“>”	3	62
“?”	4	63
“@”	5	64
“A”	6	65
“B”	7	66
“C”	8	67
“D”	9	68
“E”	10	69
“F”	11	70
“G”	12	71
“H”	13	72
“I”	14	73
“J”	15	74
“K”	16	75
“L”	17	76
“M”	18	77
“N”	19	78
“O”	20	79
“P”	21	80
“Q”	22	81
“R”	23	82
“S”	24	83
“T”	25	84
“U”	26	85
“V”	27	86
“W”	28	87
“X”	29	88
“Y”	30	89
“Z”	31	90

*Tabla 4 correspondencia entre caracteres y Sprites para el comando /LAYOUT*

El @string es una variable de tipo cadena. No puedes pasar directamente la cadena. Es decir, sería ilegal algo como:

| LAYOUT, 1, 0, "ZZZ YYY"

Lo correcto es:

Cadena\$ = "ZZZ YYY"  
| LAYOUT, 1, 0, @cadena\$

Ten cuidado de que la cadena no esté vacía, de lo contrario puede bloquearse el ordenador!. Además, debes anteponer el símbolo “@” en la variable de tipo string para

que la librería pueda ir a la dirección de memoria donde se almacena la cadena y así poder recorrerla, imprimiendo uno a uno los sprites correspondientes.

Debes tener en cuenta que los espacios en blanco significan ausencia de sprite, es decir, en las posiciones correspondientes a los espacios no se imprime nada. Si había previamente algo en esa posición, no se borrará. Si deseas borrar necesitas definirte un sprite de borrado de 8x8, donde todo sean ceros.

Aunque usas los sprites para imprimir el layout, justo después de imprimirla puedes redefinir los sprites con |SETUPSP y asignarles imágenes de soldados, monstruos o lo que quieras, es decir, el layout se “apoya” en el mecanismo de sprites para imprimir, pero no te limita el número de sprites, pues dispones de los 32 para que sean lo que tú quieras justo después de imprimir el layout

Para detectar colisiones con el layout dispones de la función COLAY, que se puede usar con un número de parámetros variable.

```
|COLAY,<umbral ASCII>, @colision , <sprite number>
|COLAY, @colision , <sprite number>
|COLAY, <sprite number>
|COLAY
```

Dado un sprite y dependiendo de sus coordenadas y de su tamaño, esta función averiguará si está colisionando con el layout y te avisará a través de la variable colision, la cual debe estar previamente definida.

El parámetro <umbral ASCII> es opcional y sirve para que el comando no considere colisión a aquellos códigos ASCII inferiores a dicho umbral. Por defecto es 32 (que es el correspondiente al espacio en blanco). Para entender esto hay que tener en cuenta la correspondencia entre valores ASCII y Sprites que se ha mostrado en la tabla anterior. Por ejemplo, si ponemos como umbral el 69 (código de la “E”, sprite 10), entonces los sprites 9, 8, 7, 6, 5, 4, 3, 2, 1, y 0 no serán “colisionables”, de modo que si nuestro personaje pasa por encima, simplemente no será detectada la colisión.

Tan solo hace falta invocar a COLAY con el parámetro de umbral una vez, ya que las sucesivas invocaciones tienen ya en cuenta dicho umbral.

Ejemplo de uso:

```
col%=0
|COLAY, @col%, 20: REM este es un ejemplo con spriteID=20
```

Si invocas al comando COLAY sin parámetros, considerará los últimos que usó. De ese modo te puedes ahorrar el paso de parámetros y acelerar el comando 0.5 ms.

Si no hay colisión, la variable tomará el valor cero. Si hay colisión, tomará el valor 1. Hay colisión si el sprite choca con algún elemento del layout diferente del espacio en blanco (“ ”), cuyo código ASCII es 32. En caso de usar el umbral, habría colisión si el elemento del layout tiene un ASCII superior al umbral que se defina.

Vamos a ver un ejemplo de creación de un layout y de movimiento de un personaje dentro del layout, corrigiendo su posición si ha colisionado.

Dos ultimas consideraciones sobre el comando COLAY:

- Al comando COLAY, no le afecta el ajuste de la sensibilidad de colisión entre sprites (configurable con COLSP,34, dy, dx). El ajuste de sensibilidad de colisión solo afecta a los comandos COLSP y COLSPALL
- El comando COLAY no tiene en cuenta el tamaño real de las imágenes usadas como “Tiles” o “bloques” del layout. Es decir que considera que todos los bloques especificados en el comando LAYOUT miden 8x8 pixeles de mode 0 (4 bytes x 8 lineas), aunque tu pongas imágenes mas grandes.

## 8.2 Ejemplo de juego con layout

Vamos a evolucionar un poco el juego presentado en el anterior capítulo, en lo que respecta al control del personaje. Esta vez vamos a usar a Montoya como ejemplo, el cual tiene 8 secuencias de animación, cada una para moverse en una dirección diferente. A las secuencias de animación les hemos asignado un número que va del 1 al 8.

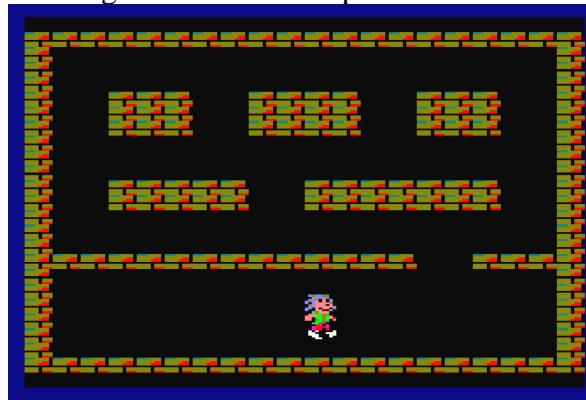
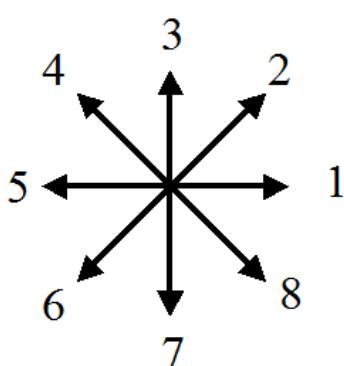


Fig. 49 Uso del layout en un juego

En la rutina de control del personaje hemos incluido colisión con el layout. En función de la dirección en la que avanzamos, modificamos las coordenadas “nuevas” (yn , xn) e invocamos a la función de colisión con layout |COLAY,0 para chequear si el sprite 0 (nuestro personaje) ha colisionado. Si ha colisionado, corregimos las coordenadas (una o las dos) para dejarle en una posición sin colisión antes de imprimirla de nuevo

```

10 MEMORY 23999
20 MODE 0: DEFINT A-Z: CALL &6B78:' install RSX
25 call &bc02:' restaura paleta por defecto por si acaso
26 ink 0,0:' fondo negro
30 FOR j=0 TO 31:|SETUPSP,j,0,&X0:NEXT:'reset sprites
40 |SETLIMITS,0,80,0,200: ' establecemos los limites de la pantalla de juego
50 dim c$(25):for i=0 to 24:c$(i)=" ":next
100 c$(1)= "ZZZZZZZZZZZZZZZZZZ"
110 c$(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$(7)= "Z  ZZZ  ZZZZ  ZZZ  Z"
160 c$(8)= "Z"           Z"

```

```

170 c$(9)="Z" Z"
190 c$(10)="Z" Z"
195 c$(11)="Z ZZZZZ ZZZZZZ Z"
200 c$(12)="Z ZZZZZ ZZZZZZ Z"
210 c$(13)="Z Z"
220 c$(14)="Z Z"
230 c$(15)="Z Z"
240 c$(16)="ZZZZZZZZZZZZZZ ZZZZ" Z"
250 c$(17)="Z Z"
260 c$(18)="Z Z"
270 c$(19)="Z Z"
271 c$(20)="Z Z"
272 c$(21)="Z Z"
273 c$(22)="Z Z"
274 c$(23)="ZZZZZZZZZZZZZZZZ ZZZZ" Z"
300 gosub 550: ' imprime el layout
310 xa=40: xn=xa: ya=150: yn=ya: ' coordenadas del personaje
311 |SETUPSP,0,0,&x111: ' detección de colisión con sprites y layout
312 |SETUPSP,0,7,1: ' secuencia = 1
320 |LOCATESP,0,ya,xa: ' colocamos al personaje (sin imprimirla)
325 c1%=0: 'declaramos la variable de colisión, explicitamente entera (%)

330 '----- ciclo de juego -----
340 gosub 1500: 'rutina de lectura teclado y movimiento de personaje
350 |PRINTSPALL,0,0
360 goto 340

550 'rutina print layout-----
560 FOR i=0 TO 23: |LAYOUT,i,0,@c$(i):NEXT
570 RETURN

1500 ' rutina movimiento personaje -----
1510 IF INKEY(27)<0 GOTO 1520
1511 IF INKEY(67)=0 THEN IF dir<>2 THEN |SETUPSP,0,7,2:dir=2:GOTO 1533 ELSE
|ANIMA,0: xn=xa+1: yn=ya-2:GOTO 1533
1512 IF INKEY(69)=0 THEN IF dir<>8 THEN |SETUPSP,0,7,8:dir=8:GOTO 1533 ELSE
|ANIMA,0: xn=xa+1: yn=ya+2:GOTO 1533
1513 IF dir<>1 THEN |SETUPSP,0,7,1:dir=1:GOTO 1533 ELSE |ANIMA,0: xn=xa+1:GOTO
1533
1520 IF INKEY(34)<0 GOTO 1530
1521 IF INKEY(67)=0 THEN IF dir<>4 THEN |SETUPSP,0,7,4:dir=4:GOTO 1533 ELSE
|ANIMA,0: xn=xa-1: yn=ya-2:GOTO 1533
1522 IF INKEY(69)=0 THEN IF dir<>6 THEN |SETUPSP,0,7,6:dir=6:GOTO 1533 ELSE
|ANIMA,0: xn=xa-1: yn=ya+2:GOTO 1533
1523 IF dir<>5 THEN |SETUPSP,0,7,5:dir=5:GOTO 1533 ELSE |ANIMA,0: xn=xa-1:GOTO
1533
1530 IF INKEY(67)=0 THEN IF dir<>3 THEN |SETUPSP,0,7,3:dir=3:GOTO 1533 ELSE
|ANIMA,0: yn=ya-4:GOTO 1533
1531 IF INKEY(69)=0 THEN IF dir<>7 THEN |SETUPSP,0,7,7:dir=7:GOTO 1533 ELSE
|ANIMA,0: yn=ya+4:GOTO 1533
1532 RETURN
1533 |LOCATESP,0,yn,xn: ynn=yn: |COLAY,@c1%,0:IF c1%=0 THEN 1536
1534 yn=ya: |POKE, 27001,yn: |COLAY,@c1%,0:IF c1%=0 THEN 1536
1535 xn=xa: yn=ynn: |POKE, 27001,yn: |POKE, 27003,xn: |COLAY,@c1%,0:IF c1%=1
THEN yn=ya: |POKE, 27001,yn
1536 ya=yn:xa=xn
1537 RETURN

```

### 8.3 Cómo abrir una compuerta en el layout

Si deseas que tu personaje pueda coger una llave y abrir una compuerta o en general eliminar una parte del layout para permitir el acceso, lo que tienes que hacer son dos pasos:

- 1) Tener definido un sprite de borrado de 8x8 donde todo sean ceros. Usando |LAYOUT lo imprimes en las posiciones que deseas
- 2) A continuación, usando nuevamente |LAYOUT, imprimes espacios donde has borrado. Así el map layout quedará con el carácter “ ” en esas posiciones y la función de colisión con el layout resultará cero.

En el juego “mutante montoya” se utiliza esta técnica para abrir la puerta del castillo, así como para abrir las compuertas que conducen a la princesa



Fig. 50 Modificación del layout al coger la llave

En el siguiente ejemplo se ilustra el concepto, abriendo una compuerta situada en las coordenadas (10, 12) de un tamaño de 2 bloques, al coger una llave que está definida con el sprite 16.

Nada mas coger la llave se abre la compuerta y la llave queda desactivada para no evaluar más veces la colisión con ella, es decir, el comando |COLSP retornará un 32 a partir del momento que cojas la llave si vuelves a colisionar con ella.

Tras abrir la compuerta, si desplazas el personaje hasta el lugar que ocupaba dicha compuerta, la colisión con layout dará como resultado 0.

```
'----- esta parte esta dentro del bucle de logica -----  
6410 |PRINTSPALL,1,0  
6411 |COLSP,@cs%,0:IF cs%<32 THEN IF cs%>=15 then gosub 6500  
(... mas instrucciones . . .)  
  
'----- rutina de apertura de compuerta-----  
6499 ' comprueba que tu colision sea con la llave, que es el sprite 16  
6500 borra$="MM":spaces$=" ":" el sprite de borrado se ha definido  
como "M" (M es el sprite 18 en el "idioma" del comando |LAYOUT)  
6501 if cs%=16 then|LAYOUT,10,12,@borra$ : |LAYOUT,10,12,@spaces$:  
|SETUPSP,16,0,0  
6502 return
```

## 8.4 Un comecocos: LAYOUT con fondo

A continuación, vamos a ver un ejemplo que usa layout y sobreescritura, para lo cual establece un umbral ASCII que permite que el comando |COLAY no considere colisión con los elementos de fondo. Concretamente como elemento de fondo se usa la letra “Y”, la cual se corresponde con el sprite id= 30, y el ASCII de la “Y” es el 89.



Fig. 51 Layout con un patrón de fondo y sobreescritura

Como puedes ver en el ejemplo tan solo hace falta invocar a COLAY con el parámetro de umbral una vez, ya que las sucesivas invocaciones tienen ya en cuenta dicho umbral

Otro de los aspectos interesantes es la gestión del teclado de este ejemplo. Es óptima para ejecutar el menor número de operaciones |COLAY y a la vez da una sensación muy agradable al avanzar por un pasillo y conectar con otro teniendo dos teclas pulsadas a la vez

```

10 MEMORY 23999
20 MODE 0: DEFINT A-Z: CALL &6B78: ' install RSX
21 on break gosub 5000
25 call &bc02: 'restaura paleta por defecto por si acaso
26 gosub 2300: ' paleta con sobreescritura
30 FOR j=0 TO 31:|SETUPSP,j,0,&X0:NEXT: 'reset sprites
40 |SETLIMITS,0,80,0,200: ' limites de la pantalla de juego
45 |SETUPSP,30,9,&84d0: ' rejilla de fondo ("Y")
46 |SETUPSP,31,9,&84f2: ' ladrillo ("Z")
50 dim c$(25):for i=0 to 24:c$(i)=" ":next
100 c$(1)= "ZZZZZZZZZZZZZZZZZZZZ"
110 c$(2)= "YYYYYYYYYYYYYYYYYYYYYZ"
120 c$(3)= "YYYYYYYYYYYYYYYYYYYYYZ"
125 c$(4)= "ZYZZYZZZZZZZZYZZYZ"
130 c$(5)= "ZYZZYZZZZZZZZYZZYZ"
140 c$(6)= "YYYYYYYYYYYYYYYYYYYYYZ"
150 c$(7)= "YYYYYYYYYYYYYYYYYYYYYZ"
160 c$(8)= "ZYZZZZZZYZZZZZZZZY"
170 c$(9)= "ZY"      "YZ"      "ZY"
190 c$(10)= "ZY"      "YZ"      "ZY"
195 c$(11)= "ZYZZZZZZYZZZZZZZZY"
200 c$(12)= "YYYYYYYYYYYYYYYYYYYYYZ"
210 c$(13)= "YYYYYYYYYYYYYYYYYYYYYZ"
220 c$(14)= "ZYZZYZZZZZZYZZYZ"
230 c$(15)= "YYYYYYYYYZ" "YYYYYYYYYZ"
240 c$(16)= "YYYYYYYYYZ" "YYYYYYYYYZ"
250 c$(17)= "ZYZZZZYZZZYZZZZZZY"
260 c$(18)= "YYYYYYYYYYYYYYYYYYYYYZ"
270 c$(19)= "YYYYYYYYYYYYYYYYYYYYYZ"

```

```

271 c$(20)="ZZZZZZZZZZZZZZZZZZZZZ"
272 c$(21)=""
273 c$(22)=""
274 c$(23)=""
300 'imprimimos el layout
310 FOR i=0 TO 20:|LAYOUT,i,0,@c$(i):NEXT
311 locate 1,1:pen 9:print "DEMO SOBREESCRITURA"
312 locate 3,23:pen 11:print "BASIC usando 8BP"
320 |SETUPSP,0,0,&x01000111:' deteccion colision con sprites y layout
330 |SETUPSP,0,7,1:dir=1: ' secuencia = 1 (coco derecha)
340 xa=20*2:xn=xa:ya=12*8:yn=ya:' coordenadas del personaje
350 |LOCATESP,0,ya,xa: 'colocamos al personaje (sin imprimirlo)
360 |PRINTSPALL,0,1,0:' imprime sprites
361 cl%=0:' variable colision
362 |COLAY,89,cl%,0:' umbral chr$("Y") es 89
400 ' COMIENZA EL JUEGO
401 |MUSIC,0,5
402 ' lectura teclado y colisiones. si vamos en direccion H (o p), primero
  chequeamos si hay pulsada tecla direccion V (q a) y viceversa
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 ' teclado direccion horizontal ---
410 if INKEY(27)<0 then 430
420 xn=xa+1poke 27003,xn:|COLAY: IF cl%=0 then if dir<>1 then
  |SETUPSP,0,7,1:DIR=1:xn=xa:return else dirn=1:return
421 xn=xa:poke 27003,xn:return:'hay colision
430 if INKEY(34)<0 then return
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:'hay colision
442 return
449 'teclado direccion vertical
450 if INKEY(67)<0 then 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:'hay colision
480 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:'hay colision
492 return

2300 REM ----- PALETA sprites transparentes MODE 0-----
2301 INK 0,11: REM azul claro
2302 INK 1,15: REM naranja
2303 INK 2,0 : INK 3,0: REM negro
2305 INK 4,26: INK 5,26: REM blanco
2307 INK 6,6: INK 7,6: REM rojo
2309 INK 8,18: INK 9,18: REM verde
2311 INK 10,24: INK 11,24: REM amarillo
2313 INK 12,4: INK 13,4 :REM magenta
2315 INK 14,16 : INK 15, 16:REM naranja
2317 AMARILLO=10
2420 RETURN
5000 |musicoff
5010 end

```

## 8.5 Cómo ahorrar memoria en tus layouts

Si tu juego tiene muchas pantallas y necesitas ahorrar espacio puedes utilizar muchas técnicas sencillas para ahorrar memoria. Una pantalla consume casi 0.5 KB de modo que es importante usar métodos para reducir su tamaño.

La forma mas sencilla de hacerlo es editar las pantallas como si fuesen sprites. Las editas con SPEDIT (por ejemplo) y cada pixel representará un elemento del layout. Dependiendo del color, representará rocas, ladrillos, espacio vacío, agua, tierra, etc. Como hay 16 colores en mode 0 disponibles, tendrás 16 tipos de ladrillo. El sprite que generes lo tendrás que almacenar como una imagen mas y antes de mostralo en pantalla, convertirlo a layout, explorando pixel a pixel y transformandolo en el código ASCII que necesites (esto te lo tendrás que programar en BASIC o como deseas). Si consideramos que usarás 5 líneas de caracteres para los marcadores del juego, una pantalla consumirá 20x20 pixeles = 400 pixeles = 200 Bytes. Por consiguiente, en 1KB te caben 5 pantallas y en 10KB te caben 50 pantallas. Habrás usado 4 bits por elemento de layout.

La edición de pantallas como si fuesen sprites es muy “visual”, aunque puedes decidir usar menos bits por elemento del layout. Si usas solo 2 bits, tendrás 4 tipos de elementos y también podrás editar pantallas como si fuesen imágenes, usando MODE 1. En ese caso te cabrán 100 pantallas en 10KB. Si usas un número diferente de bits por ladrillo la cosa se complica pues no puedes dibujar las pantallas como si fuesen sprites pero te puedes programar algo que te convierta una imagen que dibujes en los bits que necesites.

Otra solución sencilla y eficaz consiste en definir cada layout con bloques grandes, por ejemplo, de 8x16 pixels. Y construir las pantallas definiéndolas con caracteres que podemos almacenar en memoria. En el video juego “happy Monty”, se recrea la primera pantalla del mutant Monty con una matriz de 16x10 caracteres =160 bytes, de modo que 25 pantallas ocupan 4 KB. Es cierto que este layout tiene solo 16 bloques de ancho y no los 20 que puede llegar a tener, y además solo se definen 10 bloques verticalmente, en lugar de los 25 que puede llegar a tener, pero así ocupa muy poca memoria y podemos crear muchas pantallas.



Fig. 52 un layout definido con 160 caracteres



## 9 Programación avanzada y “lógicas masivas”

### 9.1 Medición de la velocidad de los comandos

El interprete BASIC es muy pesado en ejecución debido a que no solo ejecuta cada comando, sino que analiza el número de línea, realiza un análisis sintáctico del comando introducido, valida su existencia, el número y tipo de parámetros, que sus valores que se encuentren en rangos validos (por ejemplo, PEN 40 es ilegal) y muchas más cosas. Es el análisis sintáctico y semántica de cada comando lo que realmente pesa y no tanto su ejecución. El caso de los comandos RSX no es una excepción. El intérprete BASIC comprueba su sintaxis y eso pesa mucho, a pesar de que sean rutinas escritas en ASM, pues antes de invocarlas, el intérprete BASIC ya ha hecho muchas cosas.

Por consiguiente, hay que ahorrar ejecuciones de comandos, programando con astucia para que la lógica del programa pase por el menor número de instrucciones posibles, aunque ello a veces implique escribir más. Una práctica indispensable es usar instrucciones que muevan o afecten a un grupo de sprites, tales como COLSPALL, AUTOALL o MOVEALL, evitando el uso de bucles con instrucciones que afectan a un solo sprite.

Un factor decisivo a la hora de invocar un comando es el paso de parámetros. Cuantos más parámetros tiene, más costoso es su interpretación por parte del BASIC, incluso aunque sea una rutina ASM que se invoque por CALL, pues el comando CALL sigue siendo BASIC y antes de acceder a la rutina en ASM, se analiza el número y tipo de parámetros irremediablemente.

Para evaluar el coste de ejecución de un comando puedes usar el siguiente programa. También te servirá para evaluar el rendimiento de nuevas funciones en ensamblador que incorpores a la librería 8BP si deseas hacerlo.

```
1 call &6b78
10 MEMORY 23999
11 DEFINT a-z
12 c%=0: a=2
30 FOR i=0 TO 31:|SETUPSP,i,0,0:NEXT: 'reset
31 iteraciones=1000
40 a!= TIME
50 FOR i=1 TO iteraciones
60 <aquí pones un comando, por ejemplo, PRINT "A">
70 NEXT
80 b!=TIME
90 PRINT (b!-a!): rem lo que tarda en unidades de tiempo cpc. (1/300
segundos)
100 c!=((b!-a!)*1/300)/iteraciones: rem c! = lo que tarda cada
iteracion en segundos
120 d!=(1/50)/c!
130 PRINT "puedes ejecutar ",d!, "comandos por barrido (1/50 seg)"
140 PRINT "el comando tarda ";(c!*1000 -0.47);"milisegundos"
```

Nota: para los expertos en lenguaje ensamblador, debéis tener en cuenta que si pretendéis medir el tiempo de ejecución de una rutina que internamente desactiva las interrupciones (usa las instrucciones DI, EI) el tiempo que transcurre durante la desactivación no es medible con este programa BASIC. Los comandos de 8BP no desactivan las interrupciones y son todos medibles.

Vamos a ver a continuación el resultado del rendimiento de algunos comandos (medidos con el programa anterior). Hay que decir que es más rápido ejecutar una llamada directa a la dirección de memoria (un CALL &XXXX) que invocar el comando RSX correspondiente. En la siguiente tabla obviamente cuanto menor sea el resultado (expresado en milisegundos), mas rápido es el comando. La tabla que aquí se presenta debes tenerla en todo momento presente y tomar tus decisiones de programación en base a ella. Es una tabla con medidas de comandos BASIC y comandos 8BP

Comando	ms	Comentario
<b>PRINT "A"</b>	3.63	Lentísimo. Ni se te ocurra usarlo, salvo puntualmente para cambiar el número de vidas, pero no imprimas puntuación en un juego por cada enemigo que mates
<b>LOCATE 1,1 PRINT puntos</b>	24.87	Colocar el cursor de texto con LOCATE e imprimir el valor de luna variable “puntos” es costosísimo. Si actualizas puntos hazlo sólo de vez en cuando y no en cada ciclo de juego
<b>C\$=str\$(puntos)  PRINTAT,0, y, x, @c\$</b>	10	Imprimir los puntos usando PRINTAT es mucho mas eficiente que usar PRINT, pero aun asi es costoso. Usa PRINTAT con moderación.
<b>REM hola</b>	0.20	Los comentarios consumen
<b>' hola</b>	0.25	Ahorras 2 bytes de memoria, pero es más lento!!
<b>GOTO 60</b>	0.19	Muy rápido!!! Más rápido incluso que REM. Usa este comando sin piedad, úsallo!!!
<b>A = 3</b>	0.55	Una simple asignación cuesta. Todo cuesta, cada instrucción debe ser pensada.
<b>A = B</b>	0.72	Asignar el valor de una variable a otra es mas costoso que asignar un valor. Y asignar el valor de un array es aun mas costoso, porque acceder al array cuesta. Y si el array es bidimensional aun cuesta mas.
<b>A = miarray(x)</b>	1.33	
<b>A= miarray(x,y)</b>	1.84	
<b> LOCATESP,i,10,20</b>	2.8	Si no usas coordenadas negativas es mejor usar el comando BASIC POKE para establecer coordenadas.
<b> LOCATESP,i,y,x</b>	3.22	Si las coordenadas son variables entonces tarda mas.
<b>CALL &amp;XXXX,i,x,y</b>	1.81	El equivalente CALL es mucho mas rápido.
<b> MOVER,31,1,1</b>	3.23	Es algo lento y por ello debes usarlo con moderación
<b>CALL &amp;XXXX,31,1,1</b>	1.77	El equivalente call es mucho mas rápido
<b>POKE &amp;XXXX, valor</b>	0.71	Muy rápido! Úsalo para actualizar las coordenadas de los sprites (si son positivas) POKE no acepta números negativos, pero puedes usar la formula $255+x+1$ si quieres meter un numero negativo. Por ejemplo, para meter un -4 debes meter $255-4+1=252$ Otra forma sencilla de meter positivos y negativos es usar POKE dirección, x and 255
<b>POKE dir,dato</b>	0.85	Muy rápido teniendo en cuenta que además debe traducir la variable “dir”
<b> POKE,&amp;xxxx,valor</b>	2.5	Permite números negativos y si sólo actualizas una coordenada (X o Y) es mejor que LOCATESP

<code>X=PEEK(&amp;xxxx)</code>	0.93	Muy rápido! Según el tipo de videojuego puede ser una alternativa a COLSP, mirando el color de una dirección de memoria de pantalla. En el apéndice sobre la memoria de video te explico como hacerlo.
<code>X=INKEY(27)</code>	1.12	Muy rápido. Apto para videojuegos, aunque debes usarlo inteligentemente como se recomienda en este libro.
<code>IF x&gt;50 THEN x=0</code>	1.42	Cada IF pesa, hay que tratar de ahorrarlos porque una lógica de juego va a tener muchos
<code>IF A=valor THEN GOTO 100</code>  <code>Vs</code>  <code>IF A=valor THEN 100</code>	1.24  Vs  1.18	Ambas sentencias son equivalentes pero la segunda tarda menos
<code>IF inkey(27)=0 then x=5</code>	1.75	Aceptable. Es más rápido que hacer <code>b=INKEY(27)</code> y después el IF...THEN
<code>10 If inkey(27) then 30 20 x=5 30 &lt;instrucciones&gt;</code>	1.0	Una forma mucho mas eficiente de hacer lo mismo
<code>IF x&gt;0 then</code>  <code>Vs</code>  <code>IF x then</code>	1.3  Vs  0.8	En BASIC es posible ahorrar 0.5ms teniendo en cuenta que cualquier valor distinto de cero significa TRUE. Si queremos controlar un valor concreto haremos:  <code>10 IF x-20 THEN 30 20 &lt;cosas a hacer si x=20&gt; 30 ...</code> El uso de esta técnica es muy recomendable en la lectura de teclado
<code>A=A+1: IF A&gt;4 then A=0</code>  <code>Vs</code>  <code>A=A MOD 3 +1</code>	2.6  Vs  1.7	Este es un ejemplo clarísimo de como debemos programar. Es mucho mejor usar la segunda opción. Por otro lado, el uso de MOD hay que hacerlo con cautela. Si hacemos:  <code>A=(A+1) MOD 3</code> Nos cuesta 2 ms ya que los paréntesis son muy costosos y sin embargo conseguimos lo mismo. Hay una forma mejor de hacerlo, con el operador binario AND
<code>A=1+A AND 7</code>	1.6	Es mejor usar AND que MOD, ya que AND es una operación binaria rápida y MOD implica una división, muy costosa para nuestro querido microprocesador Z80
<code>A=A mod 2 +29</code>  <code>A=(A and 1) +29</code>	2.15  1.9	Estos ejemplos hacen lo msmo: te permiten oscilar la variable “a” entre los valores 29 y 30. La mejor es la segunda opción. Este mecanismo te permite oscilar entre un rango de valores según el operando de la operación “AND” o “MOD”, lo cual es muy útil para elegir el sprite a usar para tus disparos, etc.  <code>10 a=(a and 1)+29 20 print a 30 goto 10</code>

:	0.05	No ahorra mucho, pero es mas rápido usar ":" en lugar de un nuevo número de línea, y si aplicas esto muchas veces acabas ahorrando de forma significativa. Dos instrucciones en dos líneas gastan 0.03ms mas que si ambas estan en la misma línea separadas por ":"
PRINTSP,0,10,10	5.1	Un solo sprite de 14 x 24 (7 bytes x 24 líneas) Ojo, si vas a imprimir varios compensa mucho mas imprimir todos los sprites de golpe con PRINTSPALL
CALL &xxxx,0,10,10	3.5	Equivalente a PRINTSP, así es mas rápido, aunque menos legible
PRINTSPALL  (32 sprites 8x16 de mode 0, es decir 4 bytes x 16 líneas)	57	<p>Esto son unos 17 fps a plena carga de sprites. Lo que tarda es</p> $T = 3.25 + N \times 1.7$ <p>Es decir, 1.7 ms por sprite y un coste fijo de 3 ms. Este coste fijo es el coste del análisis sintáctico de BASIC sumando al de recorrer la tabla de sprites buscando cuales hay que imprimir. Si se omiten los parámetros (es posible y se tomarían los valores de la ultima invocación), se ahorran 0.6ms en la parte fija, es decir:</p> $T = 2.6 + N \times 1.1$ <p>Si la impresión es con sobreescritura y/o flipeada, es mas costosa. A continuación, se muestran los costes relativos de cada tipo de impresión:</p> <p><b>Impresión normal: 100%</b>  <b>Impresión con sobreescritura: 164%</b>  <b>Impresión flipeada: 179%</b>  <b>Impresión flipeada con sobreescritura: 220%</b></p>
PRINTSPALL,N,0,0 (ningún sprite activo)  N=0 N=10 N=31	2.6 4.3 5.9	<p>Coste de ordenar los sprites: Cuando N=0, no habiendo ningún sprite que imprimir, la función debe recorrer la tabla de sprites de forma secuencial. Pero recorrerla de forma ordenada es más costoso, tal como evidencia el tiempo consumido al aumentar N. La diferencia de tiempos (5.9 -2.6 =2.5ms) es lo que cuesta ordenar todos los sprites</p>
COLAY,@x%,0	3.0	Usar solo con el personaje, no con los enemigos o el juego irá lento. Si el personaje mide múltiplos de 8 es más rápido. En este ejemplo era de 14x24 y lógicamente 14 no es múltiplo de 8. cuanto mayor es el sprite más tarda. ¡Si invocas el comando sin parámetros es mucho mas rápido! ( ahorras 0.6 ms)
COLAY	Vs 2.4	
COLAY vs CALL &xxxx	2.4 vs 2.0	Usar CALL como siempre es más rápido, pero menos legible.
GOSUB / RETURN	0.56	Aceptablemente rápido. La medida la he hecho con una rutina que solo hace return.
SETUPSP, id, param, valor	2.7	Aceptable, aunque POKE es mucho mejor para ciertos parámetros. Hay parámetros que se pueden establecer

		con POKE como el estado, pero otros no (como una ruta). Consulta la guía de referencia
<b>FOR / NEXT</b>	0.6	Lo puedes usar para recorrer varios enemigos y que cada uno se mueva de acuerdo a una misma regla. Debes valorar si puedes usar AUTOALL o MOVEALL para tus propósitos ya que en un solo comando moverías a todos los que quieras, lo cual es mucho mejor que un bucle.
<b> COLSP,0, @c%</b>	5.5	Tarda lo mismo con independencia del número de sprites activos. Esta rutina la tendrás que invocar en cada ciclo de la lógica de tu juego, de modo que son casi 5ms que obligatoriamente tienes que destinar a esto. Si tienes una nave o personaje y varios disparos es mucho mas eficiente que invoques a COLSPALL en lugar de invocar varias veces a COLSP
<b> ANIMALL</b>	3.5	Es costoso pero hay una forma de invocarlo conjuntamente al invocar  PRINTSPALL , mediante un parámetro que hace que se invoque a esta función antes de imprimir los sprites. Ello permite ahorrar la capa del BASIC, es decir lo que consume enviar el comando, que es >1ms. Por ello podemos decir que este comando consumirá normalmente algo menos de 2ms
<b>AUTOALL</b>	2.76	No es costosa y puede mover a la vez los 32 sprites
<b>MOVEALL,1,1</b>	3.4	No es muy costosa y puede mover a la vez los 32 sprites
<b>SOUND</b>	10	El comando sound es “bloqueante” en cuanto se llena el buffer de 5 notas. Esto significa que tu lógica de BASIC no debe encadenar más de 5 comandos SOUND o se parará hasta que alguna nota termine..Si decides usarlo debe ser con sumo cuidado ya que consume mucho tiempo su ejecución (10 ms es muchísimo)
<b>IF a&gt;1 AND a&gt;2 THEN a=2</b>  <b>Versus</b>  <b>IF a&gt;1 THEN IF a&gt;2 THEN a=2</b>	2.52  Vs  2.39	Una sencilla forma de ahorrar 0.13 ms  En cada cosa que programes ten en cuenta estos detalles, cada ahorro es importante
<b>A=RND*10</b>	4.2	La función RND de BASIC es muy costosa. Puedes usarla, pero no en cada ciclo de juego sino solo eventualmente, por ejemplo, cuando aparezca un nuevo enemigo o cosas así. Otra solución sencilla es almacenar 10 números aleatorios en un array y utilizarlos en lugar de invocar a RND
<b>Border &lt;x&gt;</b>	0.75	Bastante rápida. Útil para usarla en combinación con algun tipo de colisión de sprites, reforzando el efecto explosivo
<b>IF a AND 7 then 30</b>	1.19	He puesto el tiempo de ejecución cuando se cumple la condición. Ambos casos son bastante rápidos.
<b>IF A MOD 8 then 30</b>	1.29	

*Tabla 5 Relación de tiempos de ejecución de algunas instrucciones*

## Recomendaciones importantes:

- **Usar DEFINT A-Z al principio del programa.** El rendimiento mejorará muchísimo. Esto es casi obligatorio. Este comando borra las variables que existiesen antes y obliga a que todas las nuevas variables sean enteros a menos que se indique lo contrario con modificadores como “\$” o “!” (Consulta la guía de referencia de programador BASIC de Amstrad). Ojo, en cuanto uses DEFINT, si quieras asociar un número mayor que 32768 tendrás que hacerlo en hexadecimal.
- Si puedes evitar pasar por un IF insertando un GOTO, siempre será preferible
- Cuando te falte velocidad y necesites un poquito más de rapidez utiliza CALL <dirección> y en lugar de RSX. En caso de hacer esto, has de pasar los parámetros que contengan números negativos en hexadecimal.
- No sincronices el comando PRINTSPALL con el barrido de pantalla a menos que tu juego funcione muy rápido. Sincronizar puede reducir tus FPS. En general con que consigas 12 FPS tu juego será “jugable”.
- Eliminar espacios en blanco. Cada espacio en blanco en tu listado BASIC consume 0.01ms en ejecución.
- Una vez que hayas invocado con parámetros al comando STARS o al comando PRINTSPALL, o a COLAY o a otros comandos de 8BP, las siguientes veces no lo invoques con parámetros. La librería 8BP tiene “memoria” y usará los últimos parámetros que usaste. Esto ahorra milisegundos al atravesar la capa de análisis sintáctico del intérprete BASIC.
- Ten siempre en cuenta que una expresión diferente de cero es TRUE. Esto te permitirá ahorrar 0.5ms en cada IF y lo puedes usar en la lectura de teclado y en el control de variables

Mala opción	Buena opción (ahorras 0.5ms)
<b>IF x&lt;&gt;0 THEN &lt;instrucciones&gt;</b>	<b>IF x THEN &lt;instrucciones&gt;</b>
<b>IF x=20 THEN...</b>	<b>10 If x-20 THEN 30 20 &lt;instrucciones&gt; 30</b>
<b>IF INKEY(34)=0 THEN &lt;instrucciones&gt;</b>	<b>10 IF INKEY(34) THEN 30 20 &lt;instrucciones&gt; 30</b>

- En juegos de naves donde no uses sobreescritura, procura que tu nave sea el sprite 31, de este modo pasará por “encima” de los sprites que simulan ser el fondo, pues tu nave se imprimirá después.
- Prueba versiones alternativas de una misma operación
 

**A=A+1:IF A>4 then A=0 : REM esto consume 2.6ms**

**A=A MOD 3 +1 : REM esto consume 1.84 ms**

**A=1 + A AND 3 : REM esto consume 1.6 ms**

- Evita el uso de coordenadas negativas. Ello te permitirá usar POKE para actualizar la posición de tu personaje. El comando POKE (el de BASIC) es muy veloz pero solo soporta números positivos, al igual que PEEK. En caso de usar coordenada negativas, usa |POKE y |PEEK (comandos de 8BP). Reserva el uso de |LOCATESP para cuando vayas a modificar ambas coordenadas a la vez y puedan ser positivas y/o negativas. Recuerda también que un POKE de un valor x negativo se puede hacer usando POKE dirección, 255+x+1. En caso de que quieras usar coordenadas negativas para que se vea como poco a poco los enemigos entran a la pantalla por el lateral izquierdo (que se perciba el clipping), puedes evitar las coordenadas negativas usando un SETLIMITS y de esa manera producir el mismo efecto con coordenadas que comienzan en cero y una pantalla de juego ligeramente mas pequeña
- Si necesitas comprobar algo, no lo hagas en todos los ciclos de juego. A lo mejor basta que compruebes ese "algo" cada 2 o 3 ciclos, sin ser necesario que lo compruebes en cada ciclo. Para poder elegir cuando ejecutar algo, haz uso de la "aritmética modular". En BASIC dispones de la instrucción MOD que es una excelente herramienta. Por ejemplo para ejecutar una de cada 5 veces puedes hacer : IF ciclo MOD 5 = 0 THEN ... aunque es mejor que uses operaciones AND que operaciones MOD
- Haz uso de las "secuencias de muerte". Ello te permitirá ahorrar instrucciones para comprobar si un sprite que está explotando ha llegado a su último fotograma de animación para desactivarlo.
- La sobreescritura es costosa: si puedes hacer tu juego sin sobreescritura ahorrarás milisegundos y ganaras colorido. Úsala cuando la necesites, pero no sin motivo
- Las macrosecuencias de animación te ahorran líneas de BASIC ya que no necesitas chequear la dirección de movimiento del sprite. Úsalas siempre que puedas.

## **9.2 Haz una sola lógica para gobernar todas tus pantallas**

Nuestro amstrad solo tiene 64KB y si descontamos la memoria de video, la memoria para tus sprites, tu música y la librería 8BP, te quedan 24KB de BASIC que debes usar muy bien. Si cada pantalla tiene su propio "programa" dentro de tu juego, apenas podrás hacer un juego de 10 pantallas.

Hay dos cosas que debes tratar de hacer para reducir el uso de la memoria

- Crear pantallas con pocos bytes
- Crear una sola lógica que gobierne todas las pantallas, es decir que un mismo ciclo de juego es el que se va a ejecutar en todas las pantallas del juego.

En los juegos de pasar pantallas que usan layout, cada pantalla puede ocupar 20x25 bytes, es decir 500 bytes. Si empleas ciertos "trucos" como los explicados en el capítulo 8, puedes reducir esta memoria. En el videojuego "happy Monty" se

construyen 25 pantallas con solo 160 bytes cada una y hay una única lógica de ciclo de juego para todas las pantallas.

**Es muy importante que programes una única lógica de ciclo de juego y la apliques a todas las pantallas.** Si programas una lógica de ciclo de juego para cada pantalla, el código fuente de tu programa será enorme y podrás programar pocas pantallas pues se te acabará la memoria pronto.

Tambien puedes superar las limitaciones de memoria mediante algoritmos que generen laberintos, o pantallas sin necesidad de almacenarlos. De este modo podrás hacer muchas más pantallas. Esto requiere creatividad, desde luego, pero es posible. Un algoritmo siempre ocupa menos que los datos que genera, aunque lógicamente cuesta mas tiempo ejecutarlo que simplemente leer datos almacenados.

Puedes reutilizar la lógica de enemigos de una pantalla en otra, ahorrando líneas de código. Aprovecha el mecanismo GOSUB/RETURN para ello. Tambien es muy útil usar rutas en los enemigos. **Con el mecanismo de rutas, el enemigo se mueve sin ejecutar lógica BASIC** y funcionará muy rápido. Basta con asignar una ruta a un enemigo para que este la recorra una y otra vez sin que necesitemos costosas instrucciones IF , asignaciones, etc

También puedes hacer juegos que carguen por fases, de modo que no tengas todo el juego en memoria a la vez. Esto es un poco molesto para el usuario de cinta (CPC464) aunque no lo es para el de disco (CPC6128)

Usa sprite flipping para ahorrar memoria en tus sprites

### **9.3 Técnica de “Lógicas masivas”**

A menudo vas a necesitar mover muchos sprites, sobre todo en juegos de arcade del espacio o de estilo “commando” (el clásico de Capcom de 1985).

Podrías actuar por separado en las coordenadas de todos los sprites y actualizarlas usando POKE, pero resultaría muy lento, inviable si quieres fluidez de movimientos. Lo más recomendable (y sencillo) es hacer uso combinado de las funciones de movimiento automático y de movimiento relativo, que son |AUTOALL y |MOVEALL respectivamente.

La clave de lograr velocidad en muchos sprites es utilizar la técnica que he bautizado como “lógicas masivas”. Esta técnica consiste fundamentalmente en ejecutar menos lógica en cada ciclo de juego (lo que se denomina “reducir la complejidad computacional”) y para ello hay varias opciones:

- Usar una sola lógica que afecta a muchos sprites a la vez (usando los flag de movimiento automático y/o relativo)
- Ejecutar varias tareas, pero solo una de ellas o unas pocas en cada ciclo del juego, usando aritmética modular (u operaciones binarias) en cascada.
- Introducir limitaciones en el juego que no sean importantes o no afecten a la jugabilidad, para reducir el numero de tareas que se ejecutan en cada ciclo de juego o simplificar las tareas de modo que se ejecuten mas rápido.

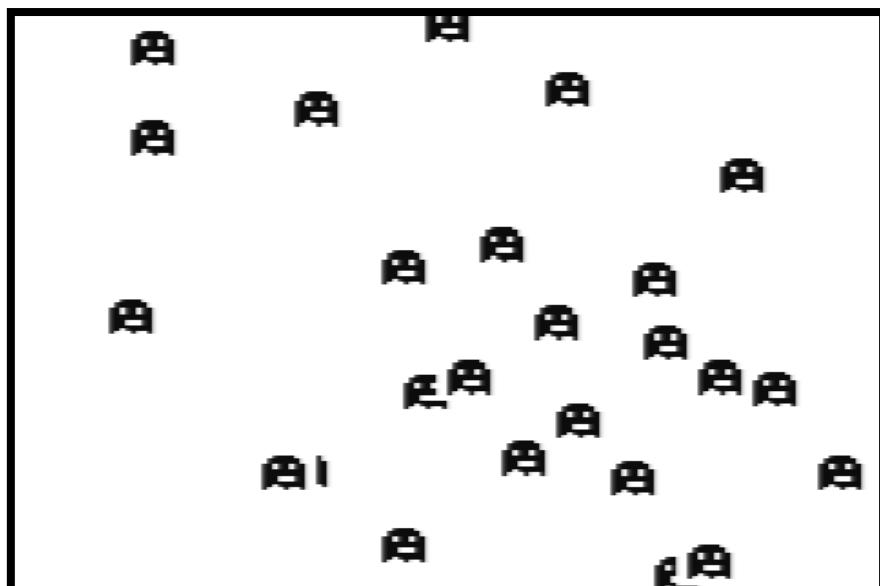
- Como norma general, reduce el numero de instrucciones por las que pasa tu programa en cada ciclo de juego, reemplazando a veces algoritmos por precálculos o poniendo mas instrucciones para lograr (paradójicamente) que se ejecuten menos cada ciclo.

Estas ideas tienen un mismo objetivo: **ejecutar menos lógica en cada ciclo**, permitiendo que todos los sprites se muevan a la vez, pero tomando menos decisiones en cada ciclo del juego. A esto se le llama “**reducir la complejidad computacional**”, **transformando un problema de orden N (N sprites) en un problema de orden 1 (una sola lógica a ejecutar en cada fotograma)**.

La clave está en determinar qué lógica o lógicas ejecutar en cada ciclo. En el caso más sencillo, si tenemos N sprites simplemente ejecutaremos una de las N lógicas. Pero en casos más complejos deberemos ser astutos para determinar que lógicas conviene ejecutar.

### 9.3.1 Mueve 32 sprites con lógicas masivas

Ahora vamos a ver un sencillo ejemplo para mover 32 sprites simultáneamente y suavemente (a 14fps). Es perfectamente posible. Solo un fantasma va a tomar decisiones en cada ciclo, aunque se van a mover todos los fantasmas en todos los ciclos. También podemos animar a todos (asociándoles una secuencia de animación y usando `|PRINTSPALL,1,0`) y seguirá quedando suave, pero aun parecerá que hay mayor movimiento pues el aleteo de las alas de una mosca (por ejemplo) genera mucha sensación de movimiento



*Fig. 53 con lógicas masivas puedes mover 32 sprites simultáneamente*

Lo que hemos hecho ha sido reducir la complejidad computacional. Hemos partido de un problema de “orden N”, siendo N el número de sprites. Suponiendo que cada lógica de sprite requiera 3 instrucciones BASIC, en principio habría que ejecutar  $N \times 3$  instrucciones en cada ciclo. Con la técnica de “lógicas masivas”, transformamos el problema de “orden N” en un problema de “orden 1”. Se llama problemas de “orden 1” a los que involucran un número constante de operaciones independientemente del tamaño del problema. En este caso hemos pasado de  $N \times 3 = 32 \times 3 = 96$  operaciones

BASIC a sólo 3 operaciones BASIC. Esta reducción de complejidad es la clave del alto rendimiento de la técnica de lógicas masivas.

```

1 MODE 0
10 MEMORY 23999: CALL &6B78
20 DEFINT a-z
25 ' reset enemigos
30 FOR j=0 TO 31:|SETUPSP,j,0,&X0:NEXT
35 ' num enemigos de 12 x 16 (6bytes de ancho x 16 lineas)
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 next
43 i=0
45 gosub 100
46 i=i+1: if i=num then 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 azar=rnd*3
170 if azar=0 then |SETUPSP,i,5,2: |SETUPSP,i,6,0:return
180 if azar=1 then |SETUPSP,i,5,-2:|SETUPSP,i,6,0:return
190 if azar=2 then |SETUPSP,i,5,0:|SETUPSP,i,6,1:return
200 if azar=3 then |SETUPSP,i,5,0:|SETUPSP,i,6,-1:return

```

### 9.3.2 Ejecución alternada y periódica en cascada

No es necesario que ejecutes todas las tareas en cada ciclo de juego. Por ejemplo, si quieras comprobar si el disparo ha salido de la pantalla, puedes comprobarlo cada dos o 3 ciclos, en lugar de comprobarlo en cada ciclo.

También puedes hacer que los enemigos disparen cada cierto número de ciclos y no cada ciclo (¡de lo contrario te dispararían muchísimo!!)

En definitiva, hay cosas que no necesitas hacer en cada ciclo y puedes ahorrar ejecución de instrucciones por ciclo y por lo tanto lograrás mayor velocidad. Esto es en realidad el fundamento básico de la técnica de “lógicas masivas”.

Hay dos técnicas básicas para esto: El uso de aritmética modular y las operaciones binarias AND. Las operaciones binarias AND son más rápidas

Técnica	Tiempo consumido
A = A+1: if A =5 then A=0: GOSUB <rutina>	2.6 ms
IF ciclo MOD 5 =0 THEN gosub rutina	1.84ms, suponiendo que ya tienes una variable llamada ciclo que se actualiza

La operación MOD es algo costosa y por ello a veces es mejor una operación binaria.

Suponiendo que tienes la variable ciclo que se actualiza cada vez, podemos hacer una operación binaria para ver cuando un grupo de bits da un determinado valor. Por ejemplo, si observamos los 4 bits menos significativos de la variable ciclo siempre van a ir desde 0000 hasta 1111 y vuelta a empezar. Pues bien, si hacemos un AND 15 con dicha variable podremos hacer lo mismo que con MOD 15. El número 15 en binario es 1111 y por eso un AND nos revela el valor de esos 4 bits.

Técnica	Tiempo consumido
If ciclo AND 15=0 then gosub rutina	1.6 ms (se ejecuta una de cada 16 veces)
If ciclo AND 1=0 then gosub rutina	1.6ms (Se ejecuta una de cada 2 veces)

Si tienes varias cosas periódicas a ejecutar puedes hacerlo así:

```
c=ciclo AND 15 :' rem 15 es en binario 1111
IF c=0 THEN GOSUB <rutina1> (se ejecuta "rutina1" una de cada 16 veces)
iF c=8 THEN GOSUB <rutina2>... ( rutina2 se ejecuta una vez cada 16 veces,
pero alejado en el tiempo de la ejecución de la rutina1)
```

De esta forma estás repartiendo el tiempo en distintas tareas, de forma que en cada ciclo sólo haces una tarea, pero al cabo de varios ciclos has hecho todas las tareas.

Para comprobar la variable ciclo y decidir ejecutar una tarea hay una forma de ejecutar las operaciones binarias mejor y es la siguiente:

Técnica	Tiempo consumido
10 If ciclo and 7 then 30 20 <instrucciones que se ejecutan cada 8 ciclos> 30 <continuación del programa>	<b>1.18 ms.</b> Esta es sin lugar a dudas la mejor estrategia para la ejecución periódica de tareas

Aplicando la misma estrategia a MOD logramos también aumentar la velocidad, aunque menos que con AND. Sin embargo es muy buena porque funciona aunque el periodo no sea múltiplo de 2 ( puedes poner MOD 7, MOD 5, MOD 10, etc.)

Técnica	Tiempo consumido
10 If ciclo MOD 8 then 30 20 <instrucciones que se ejecutan cada 8 ciclos> 30 <continuación del programa>	<b>1.29 ms.</b> Casi tan bueno como AND y la mejor estrategia cuando necesitamos un periodo que no es múltiplo de 2

Como ves, por cada tarea que queramos introducir en la lógica, deberemos introducir un “IF”. **Sin embargo, aun se puede mejorar** y hacerlo más eficiente usando intervalos de tiempo de ejecución de tareas que sean múltiplos. Si son múltiplos, **el “IF” de la tarea 2 se puede hacer dentro de la tarea 1, y el de la tarea 3 dentro de la tarea 2, en “cascada”**. Esto reduce enormemente el número de IF que ejecutamos en cada ciclo, siendo en muchos casos de un solo IF.

Vamos a ver un ejemplo completo, que te permite multitarea de 4 tareas diferentes, pero reduciendo el numero de tareas que se ejecutan a la vez. La siguiente secuencia representa el orden de ejecución de las tareas y a continuación el código fuente.



```

10 IF ciclo AND 1 THEN 90
20 REM cada dos ciclos entramos aquí
25 IF ciclo AND 3 THEN 80
30 REM cada 4 ciclos entramos aquí
35 IF ciclo AND 7 THEN 70
40 REM cada 8 ciclos entramos aquí
50 <tarea 4> : GOTO 100
70 <tarea 3> : GOTO 100
80 <tarea 2> : GOTO 100
90 <tarea 1>
100 REM --- fin de tareas ---

```

Gracias a esta estrategia de **usar aritmética modular con operaciones binarias en intervalos múltiplos para hacerlas en cascada**, podemos reducir el número de operaciones “IF” al mínimo y a la vez reducimos la complejidad computacional de orden N (n tareas) a orden 1 (una sola tarea por ciclo). Esto acelera muchísimo tus juegos

### 9.3.3 Ejemplo sencillo de lógica masiva

En el videojuego “Mutante Montoya”, los sprites enemigos se turnan para ejecutarse en los distintos ciclos del juego. Cuando programé este juego aun no había programado el mecanismo |ROUTEALL que permite asignar trayectorias fijas a los sprites, pero pude solventarlo con lógicas masivas. En caso de que los enemigos tuviesen “inteligencia”, no serviría una ruta fija y tendríamos que turnar la lógica de los sprites tal como se describe a continuación:

Supongamos que tenemos 3 soldados enemigos que se mueven de derecha a izquierda y de izquierda a derecha. Para ganar velocidad vamos a ejecutar sólo la lógica de un soldado en cada ciclo de juego.

Para que, a pesar de ello, la coordenada x de cada soldado siga avanzando, usaremos el flag de movimiento automático, en lugar de actualizarla nosotros.

```

10 MEMORY 23999
20 MODE 0: DEFINT A-Z: CALL &6B78:' install RSX
25 FOR j=0 TO 31:|SETUPSP,j,0,&X0:NEXT:'reset sprites
26 |SETLIMITS,0,80,0,200
30 'parametrizacion de 3 soldados
40 dim x(3):x%=0
50 x(1)=10:xmin(1)=10:xmax(1)=60:
y(1)=60:direccion(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:direccion(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: 'colocamos los sprites
81 i=0
89 ----- BUCLE PRINCIPAL DEL JUEGO (CICLO DE JUEGO) -----
90 i=i+1:gosub 100
92 if i=3 then i=0
93 |AUTOALL
94 |PRINTSPALL,1,0: ' anima e imprime los 3 soldados
95 goto 90
96 ----- FIN DEL CICLO DE JUEGO -----
99 ----- rutina de soldado -----
100 |PEEK,27003+i*16,@x%: x(i)=x%
101 IF direccion(i)=0 THEN IF x(i)>=xmax(i) THEN
direccion(i)=1: |SETUPSP,i,7,10: |SETUPSP,i,6,-1 ELSE return
110 IF x(i)<=xmin(i) THEN direccion(i)=0: |SETUPSP,i,7,9 :
|SETUPSP,i,6,1
120 return

```

Cada soldado tiene su propia lógica, pero solo ejecutamos una en cada ciclo de juego, aligerando muchísimo el ciclo de juego.

La única limitación es que al ejecutar la lógica de cada soldado una de cada 3 veces, la coordenada podría sobrepasar el límite que hemos establecido durante dos ciclos. Eso hace que debamos ser más cuidadosos al fijar el límite, asegurándonos al ejecutarlo que nunca invade y borra un muro de nuestro laberinto de pantalla, por ejemplo. Voy a tratar de explicar este problema con más precisión:

Supongamos que tenemos 8 sprites y nuestro sprite se mueve en todos los ciclos, pero sólo ejecutamos su lógica una de cada 8 veces. Imagínate un sprite que está en la posición x=20 y queremos que se mueva hasta la posición x=30 y dar la vuelta. Consideremos que el sprite tiene un movimiento automático con Vx=1. En ese caso comprobaremos su posición cuando x=20, x=28, x=36. ¡Al llegar a 36 nos daremos cuenta de que nos hemos pasado!!! y cambiaremos la velocidad del sprite a Vx=-1

Como ves el control de los límites de la trayectoria no es preciso, a menos que tengamos en cuenta esta circunstancia y fijemos el límite en algo que podamos controlar, que será Xfinal = Xinicial + n\*8.

Esta limitación es minúscula si la comparamos con la ventaja de mover muchos sprites a gran velocidad. Con algo de astucia podemos incluso ejecutar la lógica menos veces, de modo que solo uno de cada dos ciclos se ejecute algún tipo de lógica de enemigos.

### 9.3.4 Movimiento “en bloque” de escuadrones

Si lo que quieras es simplemente mover a la vez un escuadrón en una dirección, cualquiera de las dos funciones siguientes de la librería 8BP te servirán:

- Si usas |AUTOALL tienes que ponerle velocidad automática a los sprites en la dirección que quieras (en Vx, en Vy o en ambas) y por supuesto activar el bit 4 del byte de estado. El comando AUTOALL tiene un parámetro opcional para invocar internamente a |ROUTEALL antes de mover a los sprites

- Si usas |MOVEALL tienes que activar el bit 5 del byte de estado a los sprites que vayas a mover. Este comando requiere como parámetros cuento movimiento relativo en Y y en X deseas

De esta forma con una sola instrucción estás moviendo muchos sprites a la vez. En caso de que cada uno de tus sprites se deba mover de forma independiente y con una lógica independiente como ourre en juegos tipo “pacman”, habrá que ser mas astutos, como te contare a continuación.

### **9.3.5 Técnica de lógicas masivas en juegos tipo “pacman”**

Si tienes muchos enemigos y deben tomar decisiones en cada bifurcación de un laberinto, no es una buena estrategia simplemente turnar la ejecución de lógicas de enemigos en cada ciclo. Lo que conviene es ejecutar la logica cuando dicha logica deba tomar una decisión. En juegos tipo pacman esto ocurre cuando un fantasma llega a una bifurcación en la que puede tomar una nueva dirección de movimiento en función de su inteligencia. Esto se puede llevar a cabo con un sencillo “truco”. Es simplemente colocar a los enemigos en posiciones bien escogidas al empezar el juego.

Supongamos que tienes 4 enemigos y que las bifurcaciones del laberinto ocurren en múltiplos de 4. Si el primer enemigo está en una posición múltiplo de 4 le tocará ejecutar su lógica. Al segundo enemigo le toca ejecutar su lógica de decisión en el ciclo siguiente. Si no se encuentra en una posición de bifurcación del laberinto no podrá cambiar su rumbo

Para que “encaje” su posición con un múltiplo de 4 y así poder decidir qué camino tomar en la bifurcación, simplemente empezamos el juego con este segundo enemigo colocado en un múltiplo de 4 menos uno. Considerando coordenadas que comienzan en cero, los múltiplos de 4 son:

Primer enemigo: posición 0 o 4 o 8 o 12 o 16 o 20 o 24 o XX (en eje x o y, da igual)

Segundo enemigo: posición 1 o 5 o 9 ...

Tercer enemigo : posición 2 o 3 o 10 ...

Y así sucesivamente. Colocas a tus enemigos siguiendo esta regla:

**Posición = múltiplo de 4 - n, siendo n el numero de sprite**

Y cada vez que le toque a un enemigo ejecutar su lógica, podrá encontrarse en una bifurcación. si un fantasma debe tomar una decisión en el instante “i”, posteriormente lo hará en  $t=i+4$  y la siguiente será en  $t=i+4+4$ , etc.

Vamos a ver un ejemplo grafico, en el que he destacado con un rectángulo que enemigo va a tomar una decisión por encontrarse en bifurcación. Como puedes comprobar solo se ejecuta una lógica de bifurcación en cada ciclo de juego

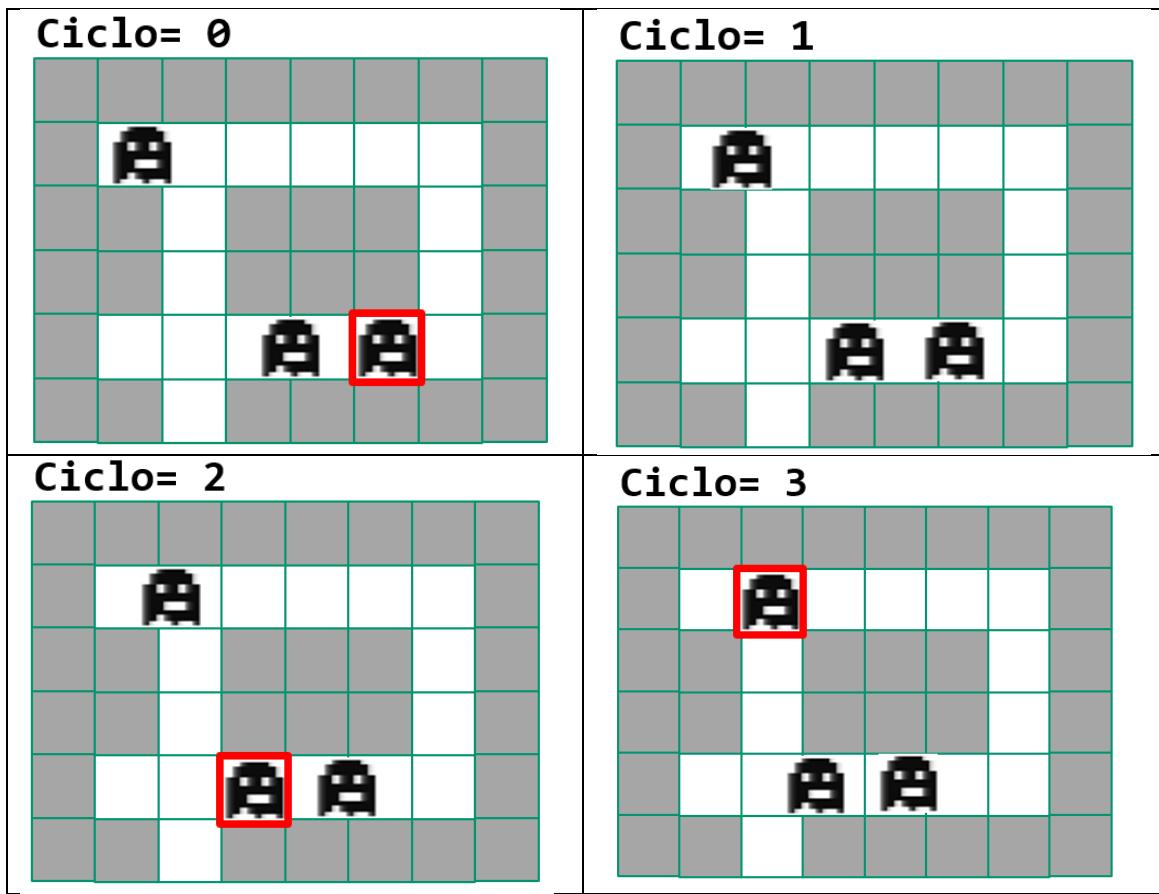


Fig. 54 Lógicas masivas en juegos tipo PACMAN

Cuando le toque ejecutar su lógica deberá comprobar que no se encuentra en una posición en mitad de un pasillo sin bifurcaciones. Debe comprobarlo y para ello puede usar el comando COLAY.

### 9.3.6 Reducción del número de instrucciones en el ciclo de juego

Reducir el numero de instrucciones por las que pasa el ciclo de juego es fundamental para acelerar tu programa. A veces ello implica que el programa tenga mas líneas, aunque al final se ejecuten menos líneas en cada ciclo. Otras veces puedes introducir limitaciones “indetectables” que aceleran tu juego sin que el jugador perciba la diferencia. Vamos a ver algunos ejemplos

#### 9.3.6.1 Gestión de teclado con menos instrucciones

Gestiona el teclado (y en general esto es aplicable a cualquier cosa que hagas) ejecutando el menor número de instrucciones. Aquí tienes un ejemplo (primero mal hecho y luego bien hecho), donde como mucho se pasa por 4 operaciones INKEYS con sus correspondientes IF. Ejecútalo mentalmente y comprobarás lo que digo. Es mucho más rápida la segunda

Ejemplo mal hecho (caso peor = 8 ejecuciones “IF INKEY”):

```
1000 rem rutina de teclado ineficiente
1010 IF INKEY(27)=0 and INKEY(67)=0 THEN <instrucciones>:RETURN
1020 IF INKEY(27)=0 and INKEY(69)=0 THEN <instrucciones>:RETURN
```

```

1030 IF INKEY(34)=0 and INKEY(67)=0 THEN <instrucciones>:RETURN
1040 IF INKEY(34)=0 and INKEY(69)=0 THEN <instrucciones>:RETURN
1050 IF INKEY(27)=0 THEN <instrucciones>:RETURN
1060 IF INKEY(34)=0 THEN <instrucciones>:RETURN
1070 IF INKEY(67)=0 THEN <instrucciones>:RETURN
1080 IF INKEY(69)=0 THEN <instrucciones>:RETURN

```

A continuación, el ejemplo de lectura de teclado, pero bien hecho (con caso peor = 4 ejecuciones “IF INKEY”). Además, se ha tenido en cuenta que, en un IF, las expresiones distintas de cero son TRUE. Las instrucciones a ejecutar en tu juego pueden ser diferentes pero el esquema de lectura de teclado debería ser el mismo en caso de manejar diagonales (arriba y derecha a la vez, por ejemplo). Puede parecer mas largo, pero es mucho mas rápido que el ejemplo anterior.

```

REM rutina del teclado eficiente
'saltar a 1550 si no se ha pulsado "P"
1510 if inkey(27) THEN 1550: 'tecla P
1520 if inkey(67) THEN 1530: 'tecla Q

1525 <instrucciones en caso de haber pulsado "P" y "Q" a la
vez>:RETURN
1530 if inkey(69) THEN 1540:'tecla A

1535 <instrucciones en caso de haber pulsado "P" y "A" a la
vez>:RETURN

1540 <instrucciones en caso de haber pulsado "P" solamente>:RETURN

1550 if inkey(34) THEN 1590:'tecla O
1560 if INKEY(67) THEN 1570:'tecla Q

1565 <instrucciones en caso de haber pulsado "O" y "Q" a la
vez>:RETURN

1570 if INKEY(69) THEN 1580: 'tecla A

1575 <instrucciones en caso de haber pulsado "O" y "A" a la vez>:
RETURN
1580 <instrucciones en caso de haber pulsado "O" solamente>:RETURN

1590 IF INKEY(67) THEN 1600:'tecla Q
1595 <instrucciones si se ha pulsado "Q">: RETURN

1600 IF INKEY(69) THEN return:'tecla A
1610 <instrucciones si se ha pulsado "A" solamente>: RETURN

```

Otra cosa que debes hacer para acelerar tu juego es usar una tarea periodica para explorar teclas “secundarias” tales como teclas para activar/desactivar la música, teclas para pasar a un menú o visualizar algo especial, etc. Son teclas que puedes explorar periódicamente y no cada ciclo. Eso si, debes tener en cuenta lo que cuesta explorarlas, que no es mucho (1 ms)

```

10 if inkey(47) then 30: ' esto cuesta 1.0 ms

```

```

20 <instrucciones si pulsas la tecla 47>
30 rem llegas aquí si no la has pulsado

```

Analizar una variable con AND para que ocurra una tarea cada (por ejemplo) 4 ciclos cuesta 1.18 ms, por lo tanto nos va a costar

1.18 ms x 4 ciclos (la evaluación del ciclo) + 1.0ms ( el inkey) =5.72 ms

si en lugar de hacer eso ejecutamos el inkey en cada ciclo, habriamos gastado solo 4ms por lo tanto explorar ciertas teclas basándonos en el ciclo de juego solo tiene sentido si al menos vamos a evitar explorar en algunos ciclos dos teclas

Supongamos el siguiente programa:

```

10 if ciclo and 3 then 50: ' esto cuesta 1.18 ms
20 if inkey(47) ... ' esto cuesta 1 ms
30 if inkey(35) ...' esto cuesta 1 ms
50 <mas instrucciones>

```

Tal y como está hecho, el programa evalua las teclas 47 y 35 uno de cada 4 ciclos. Cuando las evalua gasta  $1.18 + 1 + 1 = 3.8$  ms mientras que cuando no las evalua gasta 1.18 ms. Por lo tanto, el tiempo gastado en 4 ciclos es

Tiempo  $3 * 1.18 + 3.8 = 6.72$  ms

Mientras que si hubiésemos evaluado las teclas todos los ciclos habriamos gastado  $4 * 2$  ms= 8 ms. Por consiguiente, hay un ahorro de  $8 - 6.7 = 1.3$  ms por cada 4 ciclos, o lo que es lo mismo, aproximadamente 0.3 ms por ciclo de juego.

Una de las opciones que tienes, dependiendo del tipo de juego, es explorar unas teclas en los ciclos pares y las otras en los ciclos impares. Por ejemplo, en un juego que use las teclas QAOP, puedes explorar QA en los ciclos pares y OP en los impares o viceversa. De esa forma puedes obtener mayor velocidad con una limitación que el jugador seguramente no perciba. Este es el tipo de limitaciones que a veces merece la pena introducir, pero depende de cada juego.

### 9.3.6.2 Evitar pasar por IF innecesarios

Vamos a ver dos formas de hacer esto:

```

10 IF A=1 THEN < instrucciones cuando A=1>
20 IF A=2 THEN < instrucciones cuando A=2>
30 IF A=3 THEN < instrucciones cuando A=3>
40 <mas instrucciones >

```

Si puedes evitar pasar por un IF insertando un GOTO, siempre será preferible. El GOTO es un gran aliado de la técnica de logicas masivas.

```

10 IF A=1 THEN < instrucciones cuando A=1> : GOTO 40
20 IF A=2 THEN < instrucciones cuando A=2> : GOTO 40
30 < instrucciones cuando A=3> : rem si A no es 1 ni 2 entonces es 3
40 <mas instrucciones>

```

### 9.3.6.3 Reemplaza algoritmos por precálculos

Piensa en una pelota que bota. En lugar de usar las ecuaciones del movimiento acelerado, construye una ruta que mueve un sprite hacia abajo con un incremento de

coordenada “Y” mayor en cada paso y al tocar el suelo sube con un incremento cada vez menor. No hay ecuaciones complejas que ejecutar y sin embargo el efecto visual es el mismo. Así fue como hice los saltos del juego “fresh fruits & vegetables”. Esto es posible programarlo así porque **dentro del juego el universo es “determinista”**. Es decir, se puede predecir en cada instante de tiempo que va a ocurrir, por muy complejas que sean las ecuaciones que gobiernan un salto de un personaje o el movimiento de un escuadrón.

En juegos donde la lógica de los enemigos requiere del uso del cálculo de alguna función (como el coseno), precalcula todo y guárdalo en un array que uses durante la ejecución de la lógica. Calcular durante la lógica del juego tiene un coste prohibitivo.

Una lógica compleja es una lógica lenta. Si quieres hacer algo complicado, una trayectoria compleja, un mecanismo de inteligencia artificial...no lo hagas, trata de **“simularlo”** con un modelo de comportamiento más sencillo pero que produzca el mismo efecto visual. Por ejemplo, un fantasma que es inteligente y te persigue, en lugar de que tome decisiones inteligentes, haz que trate de tomar la misma dirección que tu personaje, sin ninguna lógica. Si no puedes simplificar de este modo, entonces piensa que incluso la inteligencia artificial se puede transformar en “determinista”. Si un enemigo toma una decisión sobre como moverse en función de tu posición y tu velocidad, podríamos almacenar el resultado de ese pesado algoritmo un array y evitar todos los cálculos.

```
10 Rem Vx, Vy, X, Y son la velocidad y posición de mi personaje
20 rem supongamos que tengo precalculadas las decisiones para 3
velocidades, 10 franjas de coordenadas X y 10 franjas en Y. Esto es
menos de 1KB
30 DIM perseguir(3,3,10,100)
40 rem ' cargaría los valores en el array tras calcularlos despacio
50 nuevadireccion=perseguir(Vx,Vy,x,y): rem mecanismo en acción
```

El universo que estas programando es “determinista”. Por muy complejo que sea el comportamiento de enemigos y elementos de la pantalla, si su comportamiento no depende de tu interacción, entonces hay una posición para cada cosa determinada en cada instante de tiempo. Una posición que podría precalcularse para evitar cualquier algoritmo complejo de comportamiento y el resultado seria el mismo.

### 9.3.6.4 No ejecutar comentarios

Eliminar cualquier comentario en la lógica de juego y si dejas alguno que sea REM (mas rápido), no uses la comilla. Si usas la comilla es para ahorrar 2 bytes de memoria, y es adecuado para comentar el resto del programa (inicializaciones y cosas así). Si quieres comentar partes de lógica puedes hacer lo siguiente:

```
If x>23 gosub 500
...
499 rem por esta linea no se pasa y asi comento esta rutina
500 if x > 50 THEN ...
...
550 RETURN
```

Cada comentario que ejecutes consume 0.20 ms y ahorrar su ejecución es muy fácil sin por ello dejar de poner comentarios. Hay ocasiones en las que puedes poner comentarios

en líneas siempre que haya saltos (GOTO y GOSUB/RETURN) sin miedo a gastar nada de tiempo, veamos algunos ejemplos:

```
10 goto 50 : rem este comentario no consume tiempo  
20 gosub 200: rem este comentario no consume tiempo  
200 return: rem este comentario no consume tiempo
```

### 9.3.6.5 Sólo muere un sprite en cada ciclo

El comando COLSPALL de 8BP esta pensado con “lógica masiva”. Esto significa que potencialmente te detecta la colisión de todos los sprites, pero en cuanto detecta una colisión te retorna indicando quien es el sprite colisionador y quien el colisionado. En ese momento puedes anular al sprite colisionado (desactivando su capacidad de ser colisionado en el bit 1 de su byte de estado) y volver a lanzar el comando, hasta que no haya mas colisiones (te retorne un 32 en el colisionador y el colisionado). Sin embargo, lo mas eficiente es no volver a lanzar el comando hasta el siguiente ciclo de juego. Esto significa que solo un enemigo podrá morir en cada fotograma, pero es una limitación imperceptible que acelerará mucho tus juegos.

### 9.3.7 Enrutando sprites con “lógicas masivas”

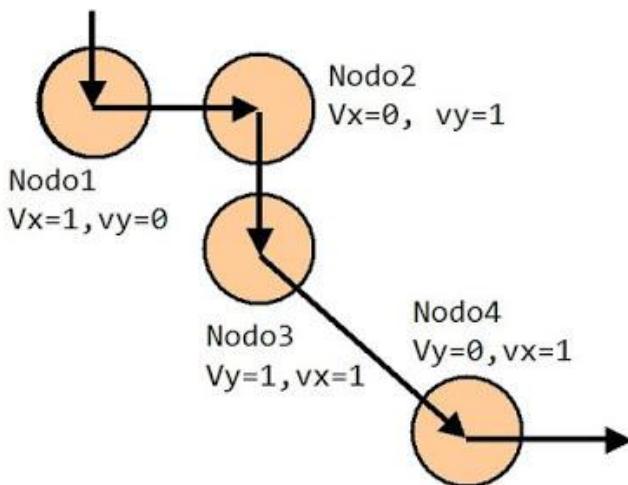
Para mover sprites a través de trayectorias existe un comando llamado |ROUTEALL que lo hace muy eficientemente, pero **como ejercicio para comprender la filosofía de lógicas masivas es muy interesante estudiar este difícil pero emblemático caso de lógicas masivas**. Para programar el videojuego “Anunnaki” empleé la técnica que voy a describir a continuación, ya que aun no había programado la capacidad de enrutamiento de sprites en 8BP. Básicamente empleé lógicas masivas en el enrutamiento de las naves enemigas.

Comencemos imaginando una trayectoria para una hilera de 8 naves enemigas. las naves pasarán una a una por una serie de "nodos de control", que son lugares en el espacio donde deben cambiar su dirección, definida por sus velocidades en X, e Y, es decir (Vx,Vy)

Una forma de controlar que las 8 naves cambien de dirección en dichos lugares sería comparar sus coordenadas X,Y con la de cada uno de los nodos de control y si coinciden con alguno de ellos, entonces aplicamos las velocidades nuevas asociadas al cambio en ese nodo. Puesto que hablamos de 2 coordenadas, 8 naves y 4 nodos, estamos ante:

$$2 \times 8 \times 4 = 64 \text{ comprobaciones en cada fotograma}$$

Esto no es viable si queremos velocidad desde BASIC, pues no es una estrategia computacionalmente eficiente. Puesto que estamos ante un escenario "**determinista**", podemos estar seguros en cada instante de tiempo donde van a encontrarse cada una de las naves y por lo tanto en lugar de hacer comprobaciones en el espacio, podemos únicamente **centrarnos en la coordenada temporal** (que es el número de fotograma del juego o el también llamado número de "ciclo de juego"). No consideres al tiempo en segundos, sino en fotogramas.



*Fig. 55 Trayectoria definida con “nodos de control”*

Puesto que conocemos a la velocidad a la que se mueven las naves, podemos saber cuando la primera de ellas pasará por el primer nodo. A ese instante lo llamaremos  $t(1)$ . También asumiremos que debido a la separación entre las naves, la segunda de las naves pasará por el nodo en el instante  $t(1)+10$ . La tercera en  $t(1)+20$  y la octava en  $t(1)+70$ . En lugar de usar fotogramas como unidades de tiempo, usemos decenas de fotogramas: en ese caso los instantes serán  $t(1), (1)+1, t(1)+2$ , etc.

Sabiendo esto podemos controlar el tiempo con dos variables: una contará las decenas ( $i$ ) y otra las unidades ( $j$ ). Para controlar el cambio de las 8 naves en el primer nodo podemos escribir:

```
j=j+1: IF j=10 THEN j=0: i=i+1: IF i>=t(1) AND i<=t(1) +8 THEN
[actualiza velocidad de nave i-t(1) con los valores de velocidad del
nodo 1]
```

Como vemos con una sola línea podemos ir cambiando las velocidades de cada nave a medida que van pasando cada una de ellas por el nodo 1. Cada vez que “ $j$ ” se hace cero, incrementamos la variable “ $i$ ” y actualizamos una de las naves. Durante los primeros 80 instantes de tiempo (8 en decenas de fotogramas) se van actualizando cada una de las 8 naves, justo cuando pasan por el nodo de control, es decir, en el instante  $t(1)$  se actualiza el Sprite 0, en el  $(t(1)+1)$  se actualiza el Sprite 1, en el  $t(1)+2$  se actualiza el Sprite 2, etc.

El Sprite number que aparece en la línea es  $i-t(1)$ , de ese modo si  $t(1)=4$  queremos que sea el fotograma 40, ( $t(1)=4$ ) entonces cuando “ $i$ ” sea 4 se empezara a actualizar el Sprite 0, y cuando “ $i$ ” valga 11 se actualizará el Sprite 7 (8 naves en total).

Ahora vamos a aplicar lo mismo a los 4 nodos. Podríamos ejecutar 4 comprobaciones en lugar de una, pero sería ineficiente. Además, si tuviésemos muchos nodos esto supondría muchas comprobaciones. Podemos hacerlo solo con una, teniendo en cuenta que la primera nave pasa por un nodo en un instante  $t(n)$  y la octava nave pasa por ese nodo en  $t(n)+7$ .

Cuando la primera nave pasa por el primer nodo, tiene sentido pensar en empezar a comprobar el nodo 2, pero no el nodo 3 ni el 4. Ya tenemos el nodo mayor que vamos a controlar.

En cuanto al nodo menor, podemos asumir que, aunque tengamos 20 nodos, estén lo suficientemente separados como para que no haya naves atravesando más de 3 nodos a la vez (vamos a suponer eso y usaremos ese "3" como parámetro). Por lo tanto, el nodo menor a comprobar es el mayor - 3. Al nodo menor lo vamos a llamar "nmin" y al mayor "nmax". (  $nmin = nmax - 3$  ). El caso que queramos tener plena libertad para poder definir cualquier trayectoria, nmin debe ser nmax menos el número de naves de la hilera.

```

10 j=j+1: IF j=10 THEN j=0: i=i+1: n=nmax
20 IF n<nmin THEN 50: ' no hay que actualizar mas naves
30 IF i>=t(n) AND i<=t(n)+8 THEN [actualiza nave i-t(n) con velocidades
de nodo n]:IF i-t(n)=0 THEN nmax=nmax+1: nmin=nmax-3
40 n=n-1
50 ' mas instrucciones del juego

```

Como ves, cuando se incrementa en 1 la decena de tiempo (variable "i"), se empieza a comprobar si hay alguna nave en uno de los nodos desde "nmax" hasta "nmin", actualizando una sola nave en cada fotograma. Si la nave que se actualiza es la cero, entonces el nodo máximo se incrementa, pues esa nave va camino del siguiente nodo.

Para el siguiente fotograma se decrementa el numero de nodo ( instrucción  $n = n-1$  ) de modo que lo que comprobaremos será si hay una nave en el nodo anterior, así sucesivamente hasta nmin. Siempre, eso si, comprobando una única nave en cada fotograma.

En resumen, hemos transformado 64 comprobaciones en solo 1, usando "Lógicas masivas". Y si la trayectoria tuviese 40 en lugar de 4 nodos, ¡habríamos transformado 640 operaciones en una sola!

El videojuego "annunaki" utiliza esta técnica para manejar las trayectorias de dos hileras simétricas de 6 naves cada una. Es complicado pero como ves desde BASIC puedes tomar el control de 12 naves y hacerlas moverse siguiendo caprichosas trayectorias, usando mas inteligencia que potencia, gracias a la técnica de lógicas masivas.

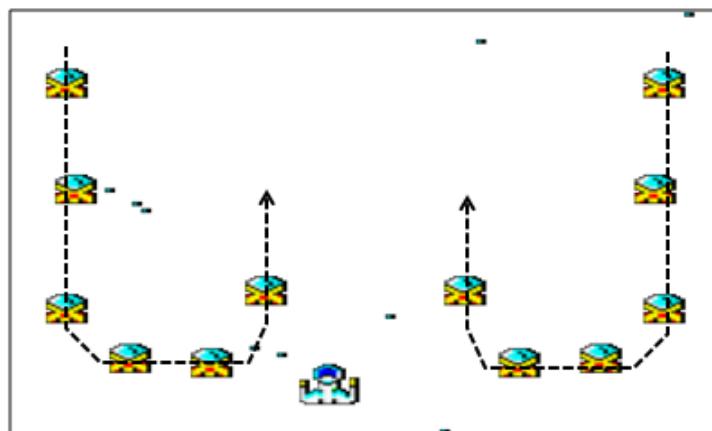


Fig. 56 Dos hileras con lógicas masivas



## 10 Trayectorias complejas: Comando ROUTEALL

Este es un comando “avanzado” disponible desde la versión V25 de la librería 8BP. Simplifica muchísimo la programación porque puedes definir una trayectoria y hacer que un sprite la recorra paso a paso mediante el comando ROUTEALL.

Primeramente, necesitas crear una ruta. Para ello necesitas editarla en el fichero routes\_tujuego.asm.

Cada ruta posee un número indeterminado de segmentos (aunque la longitud máxima de una ruta son 255 bytes) y cada segmento tiene tres parámetros:

- Cuantos pasos vamos a dar en ese segmento (entre 1 y 250)
- Qué velocidad Vy se va a mantener durante el segmento ( $-127 \leq Vy \leq 127$ )
- Qué velocidad Vx se va a mantener durante el segmento ( $-127 \leq Vx \leq 127$ )

Como una ruta puede medir a lo sumo 255 bytes y un segmento ocupa 3 bytes, una ruta puede tener como mucho 84 segmentos.

Al final de la especificación de segmentos debemos poner un cero para indicar que la ruta se ha terminado y que el sprite debe comenzar a recorrer la ruta desde el principio.

Veamos un ejemplo:

```
; LISTA DE RUTAS
;=====
;pon aqui los nombres de todas las rutas que hagas
ROUTE_LIST
    dw ROUTE0
    dw ROUTE1
    dw ROUTE2
    dw ROUTE3
    dw ROUTE4
; DEFINICION DE CADA RUTA
;=====
ROUTE0; un circulo
;-----
    db 5,2,0; cinco pasos con Vy=2
    db 5,2,-1; cinco pasos con 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; izquierda-derecha
;-----
    db 10,0,-1
    db 10,0,1
    db 0

ROUTE2; arriba-abajo
;-----
    db 10,-2,0
```

```

        db 10,2,0
        db 0

ROUTE3; un ocho
;-----
        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; un loop y se va hacia la izquierda
;-----
        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

```

Ahora para usar las rutas desde BASIC, simplemente asignamos la ruta a un sprite con el comando SETUPSP indicando que queremos modificar el parámetro 15, que es el que indica la ruta. Además, hay que activar el flag de ruta (bit 7) en el byte de status del sprite y le pondremos con el flag de movimiento automático y el de animación y el de impresión.

```

10 MEMORY 23999
11 ON BREAK GOSUB 280
20 MODE 0:INK 0,0
21 LOCATE 1,20:PRINT "comando |ROUTEALL y macrosecuencias de
animacion"
30 CALL &6B78:DEFINT a-z
31 |SETLIMITS,0,80,0,200
40 FOR i=0 TO 31:|SETUPSP,i,0,0:NEXT
41 x=10
50 FOR i=1 TO 8
51 x=x+20:IF x>=80 THEN x=10:y=y+24
60 |SETUPSP,i,0,143: rem con esto activo el flag de ruta
70 |SETUPSP,i,7,2:|SETUPSP,i,7,33: rem macrosecuencia de animacion
71 |SETUPSP,i,15,3: rem asigno la ruta numero 3
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 aqui AUTOALL ya invoca a ROUTEALL
120 GOTO 100
280 |MUSICOFF:MODE 1: INK 0,0:PEN 1

```

Ya lo tenemos todo. Esta técnica avanzada te va a simplificar la programación muchísimo con resultados espectaculares.



*Fig. 57 una ruta en forma de 8*

Como has visto, el comando no modifica las coordenadas de los sprites, de modo que deben ser movidos con AUTOALL e impresos (y animados) con PRINTSPALL. Es por ello que dispones de un parámetro opcional en |AUTOALL, de modo que AUTOALL,1 invoca internamente a ROUTEALL antes de mover el sprite, ahorrándote una invocación desde BASIC que siempre va a suponer un precioso milisegundo.

### **10.1 Coloca a un Sprite en mitad de una ruta : ROUTESP**

Si asignas a varios sprites la misma ruta y quieres que vayan todos en fila india como el ejemplo anterior, entonces necesitas ingeníártelas para que cada Sprite se encuentre en un punto diferente de la ruta. Para ello existe pudes usar dos alternativas

La primera consiste en ir asignando poco a poco la ruta a los sprites. De ese modo, el sprite que va en cabeza es el primero en ser enrutado y tras unos cuantos ciclos de juego enrutas al siguiente y mas tarde al siguiente, etc. En cada ciclo ejecutas AUTOALL,1 y eso enruta a los sprites que ya tengan ruta asignada, que se van adelantando en el numero de pasos respecto de los sprites que aun no la tienen asignada.

La segunda opción consiste en usar el el comando ROUTESP

**|ROUTESP, <spriteid>, <pasos>**

Este comando mueve un sprite el numero de pasos que deseas a lo largo de la ruta que tenga asignada. En el ejemplo he destacado en rojo la asignación de la ruta 8 y los comandos ROUTESP que posicionan cada uno de los sprites en la ruta

```

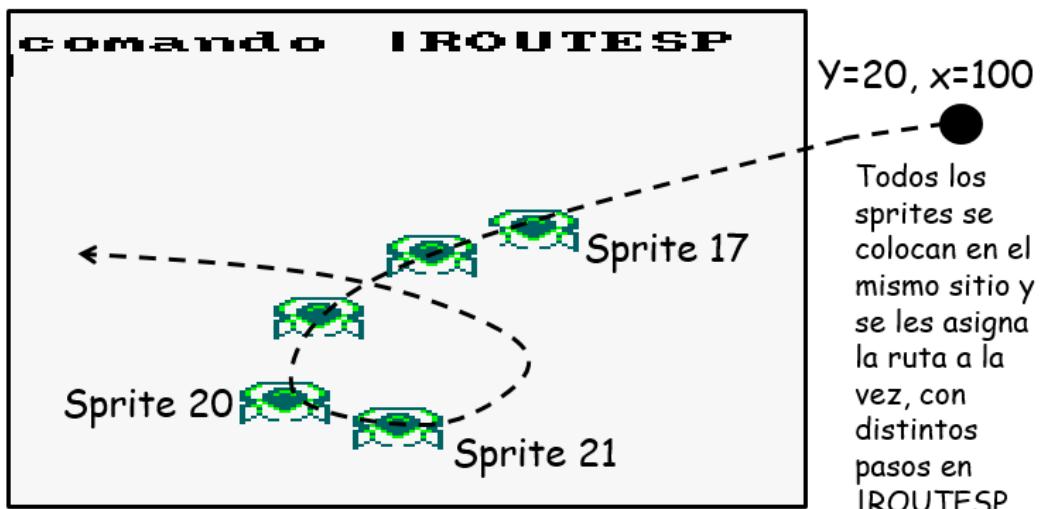
10 memory 23999
10 ON BREAK GOSUB 12
11 GOTO 20
12 |MUSICOFF:CALL &BC02:PAPER 0: PEN 1: MODE 1: END
20 CALL &BC02:DEFINT A-Z: MODE 0
50 CALL &6B78

```

```

70 ' todas las naves colocada en la misma coordenada inicial
75 ' y con la misma ruta pero con un numero inicial de pasos distinto
76 locate 2,3: Print "comando |ROUTE8P "
80 for s=16 to 21: |SETUPSP,s,9,33: |SETUPSP,s,0,137:
|SETUPSP,s,15,8: |LOCATESP,s,20,100:next
90 s=21: |ROUTE8P,s,40: 'nave cabeza
100 s=20: |ROUTE8P,s,30
110 s=19: |ROUTE8P,s,20
120 s=18: |ROUTE8P,s,10
130 s=17: |ROUTE8P,s,0
131 |PRINTSPALL,0,0,0
140 '--- ciclo de juego ---
150 |AUTOALL,1: |PRINTSPALL
160 goto 150

```



*Fig. 58 naves en hilera gracias al uso de ROUTE8P*

La ruta 8 estaría definida en el fichero routes\_mygame.asm tal como sigue:

**ROUTE8;**

```

db 255, 128+64+32+8+1,0; cambio de estado
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
; reubicacion
db 1,0,115
db 1,-30,0
db 120,0,0
db 120,0,0
db 0

```

## 10.2 Creación de Rutas avanzadas

Existen 4 funcionalidades que puedes usar en mitad de cualquier ruta (**ojo, en mitad, no al final**), usando como valor del número de pasos de un segmento un código de escape:

Código de escape (campo “número de pasos”)	Descripción	Ejemplo
255	Cambio de estado del sprite.	DB 255,3,0 Estado pasa a valor 3. El cero del final es de relleno
254	Cambio de secuencia de animación del sprite	DB 254,10,0 Se asocia la secuencia 10. El cero es de relleno Si la secuencia asignada es la que ya tiene el sprite, entonces es inocuo (no se reinicia la secuencia de animación)
253	Cambio de imagen	DB 253 DW new_img Se asocia la imagen “new_img” que debe ser una dirección de memoria
252	Cambio de ruta	DB 252,2,0 Se asocia la ruta 2
251	Pasa al siguiente frame de la animación.	DB 251,0,0 Se anima el Sprite. Los dos ceros son de relleno

**IMPORTANTE:** ten mucho cuidado de escribir DB y DW donde deben usarse, es decir, por ejemplo, si cambias de imagen debes preceder la imagen con DW y no con DB. Si cometes un error de este tipo, tu ruta no funcionará.

**IMPORTANTE:** los codigos de escape puedes emplearlos en mitad de una ruta, pero el ultimo segmento no puede ser un código de escape, debe ser un movimiento aunque sea quedarse quieto, algo como “DB 1,0,0”

### 10.2.1 Cambios de estado forzados desde rutas

Esta capacidad es muy interesante para acelerar tus juegos y está disponible desde la V27. Consiste en que podemos forzar un cambio de estado en mitad de una ruta. Para ello indicaremos que deseamos un cambio de estado indicando como valor de número de pasos del segmento = 255.

El cambio de estado es un segmento más y es importante que mantengas el mismo número de parámetros por segmento, es decir, 3 bytes. Un cambio de estado a status=13 podria escribirse como:

**255,13,0**

El tercer parámetro (el cero) no significa nada, es solo un “relleno” para que el segmento mida 3 bytes, pero es imprescindible.

El valor 255 le indicará al comando ROUTEALL que lo que debe hacer esta vez es cambiar el estado del sprite, asignándole el que se indique a continuación. El cambio de estado se ejecuta sin consumir un paso, por lo que siempre se ejecutará el siguiente paso al cambio de estado. Si no queremos que el sprite se mueva mas, simplemente definiremos un segmento de un paso sin movimiento en X ni en Y justo después del cambio de estado. Veamos un ejemplo:

```
ROUTE3; disparo_dere
;-----
db 40,0,2; cuarenta pasos a la derecha con Vx=2
db 255,0,0; cambio de estado a cero
db 1,0,0; quieto Vy=0, Vx=0
db 0

ROUTE4; disparo_izq
;-----
db 40,0,-2; cuarenta pasos a la izquierda con Vx=-2
db 255,0,0; cambio de estado a cero
db 1,0,0; quieto Vy=0, Vx=0
db 0
```

Estas dos rutas las vamos a usar para disparar con nuestro personaje. La primera de ellas, tras recorrer 40 pasos en los que avanza 2 bytes en X, sufre un cambio de estado y el sprite pasa a estado 0, es decir, desactivado. El siguiente segmento solo tiene un paso y no hay movimiento (vy=0, vx=0).

Con este mecanismo podemos disparar y que los disparos se desactiven solos cuando se salen de la pantalla. Ello ahorra lógica de BASIC y acelera nuestros juegos. En el siguiente ejemplo, la imagen 26 es el disparo.



*Fig. 59 disparos con cambio de estado en ruta*

```
10 MEMORY 23999
20 MODE 0: DEFINT A-Z: CALL &6B78: ' install RSX
25 ON BREAK GOSUB 280
30 CALL &BC02: 'restaura paleta por defecto por si acaso
40 INK 0,0: 'fondo negro
50 FOR j=0 TO 31: |SETUPSP,j,0,&X0:NEXT: 'reset sprites
80 |SETLIMITS,12,80,0,186: ' establecemos los limites de la pantalla
de juego
90 x=40:y=100: ' coordenadas del personaje
100 |SETUPSP,0,0,1: ' status del personaje
110 |SETUPSP,0,7,1:dir=1: 'secuencia de animacion asignada al empezar
120 |LOCATESP,0,y,x: 'colocamos al sprite (sin imprimirlo aun)
125 |MUSIC,0,6
126 for i=1 to 4: |SETUPSP,10+i,9,26:next: 'disparos
130 'ciclo de juego---
150 |AUTOALL,1: |PRINTSPALL,0,0
```

```

170 ' rutina movimiento personaje -----
180 IF INKEY(27)=0 THEN IF dir=2 THEN dir=1:|SETUPSP,0,7,dir ELSE
|ANIMA,0:x=x+1:GOTO 191
190 IF INKEY(34)=0 THEN IF dir=1 THEN dir=2:|SETUPSP,0,7,dir ELSE
|ANIMA,0:x=x-1
191 IF espera<ciclo-10 then if INKEY(47)=0 THEN espera=ciclo: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 ciclo=ciclo+1
210 goto 150
280 |MUSICOFF:MODE 1: INK 0,0:PEN 1

```

Los cambios de estado se pueden forzar en cualquier segmento de la ruta, no necesariamente al final, aunque en el caso de un disparo es muy lógico hacerlo al final de la ruta.

### 10.2.2 Cambios de secuencia forzados desde rutas

Podemos cambiar la secuencia de animación de un sprite usando un segmento especial. Cuando pongamos 254 en el valor del número de pasos, el comando ROUTEALL interpretará que se debe realizar un cambio de secuencia de animación en el sprite. Ejemplo:

**254,10,0**

Este segmento cambia la secuencia de animación del sprite, estableciendo la secuencia número 10. El tercer parámetro (el cero) no significa nada, es solo un “relleno” para que el segmento mida 3 bytes, pero es imprescindible.

Al igual que con el cambio de estado, el cambio de secuencia se ejecuta sin consumir un paso, por lo que siempre se ejecutará el siguiente paso al cambio de secuencia.

### 10.2.3 Cambios de imagen forzados desde rutas

Hemos visto como enrutar sprites con ROUTEALL o aun mejor, con AUTOALL,1 A menudo no queremos enrutar un sprite a través de una trayectoria sino algo más cotidiano: dar un salto con un personaje. En el ejemplo del capítulo 7 vimos como hacerlo con un array de BASIC que contiene los movimientos relativos de la coordenada Y. En este caso vamos a hacer lo mismo con una ruta, logrando un movimiento más veloz.

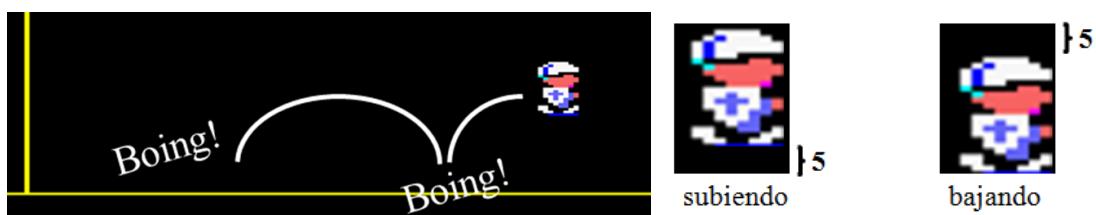


Fig. 60 Saltar usando una ruta

Para no tener que controlar si el personaje ha llegado al punto cenital del salto, podemos usar un segmento especial que indica cambio de imagen. Al igual que cualquier otro segmento, consume 3 bytes, pero en este caso el primero es el indicador de cambio de

imagen (un valor 253) y los dos siguientes se corresponden con la dirección de memoria de la imagen. **MUCHA ATENCIÓN**, deberás usar “dw” antes del nombre de la imagen que quieras asignar, por lo que este segmento de cambio de imagen lo tendrás que escribir en dos líneas. Un “db” para el 253 y un “dw” para la dirección de memoria de la imagen.

```
db 253
dw SOLDADO_R1_UP
```

En el ejemplo siguiente tenemos un muñeco que salta. Al subir el muñeco que se borra a sí mismo por abajo mientras que al bajar se borra a sí mismo por arriba. Para que no haya discontinuidad en el movimiento, justo al cambiar una imagen por otra es necesario realinear verticalmente al soldado, subiendo la imagen de bajada exactamente 5 lineas, para hacerla coincidir con el soldado que estaba subiendo.

Este sería el fichero sequences\_mygame.asm

```
org 33500;
;=====
; hasta 31 secuencias de animacion
;=====
; debe ser una tabla fija y no variable
; cada secuencia contiene las direcciones de frames de animacion ciclica
; cada secuencia son 8 direcciones de memoria de imagen
; numero par porque las animaciones suelen ser un numero par
; un cero significa fin de secuencia, aunque siempre se gastan 8 words
; al encontrar un cero se comienza de nuevo.
; si no hay cero, tras el frame 8 se comienza de nuevo

; la secuencia cero es que no hay secuencia.
; empezamos desde la secuencia 1

;-----secuencias de animacion -----
SEQUENCES_LIST
dw SOLDADO_R0,SOLDADO_R2,SOLDADO_R1,SOLDADO_R2,0,0,0,0 ;1
dw SOLDADO_L0,SOLDADO_L2,SOLDADO_L1,SOLDADO_L2,0,0,0,0 ;2
dw SOLDADO_R1_UP,0,0,0,0,0,0,0;3
dw SOLDADO_R1_DOWN,0,0,0,0,0,0,0;4
dw SOLDADO_L1_UP,0,0,0,0,0,0,0;5
dw SOLDADO_L1_DOWN,0,0,0,0,0,0,0;6

_MACRO_SEQUENCES
;-----MACRO SECUENCIAS -----
; son grupos de secuencias, una para cada direccion
; el significado es:
; still, left, right, up, up-left, up-right, down, down-left, down-right
; se numeran desde 32 en adelante
db 0,2,1,3,5,3,4,6,4; 32 --> secuencias del soldado , id=32. la siguiente
seria la 33
```

Usaremos dos rutas, una para saltar a la derecha y otra a la izquierda. Este sería el fichero routes\_mygame.asm

```
; LISTA DE RUTAS
;=====
; pon aqui los nombres de todas las rutas que hagas
ROUTE_LIST
```

```

dw ROUTE0
dw ROUTE1
dw ROUTE2
dw ROUTE3
dw ROUTE4

; DEFINICION DE CADA RUTA
;=====
ROUTE0; jump_right
db 253
dw SOLDADO_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 SOLDADO_R1_DOWN
db 1,-5,1; subo para que UP y down encajen
db 2,1,1
db 2,2,1
db 2,3,1
db 2,4,1
db 1,5,1
db 253
dw SOLDADO_R1
db 1,5,1; baja una mas
db 255,13,0; nuevo estado, ya sin flag de ruta y con flag de animacion
db 254,32,0; macrosecuencia 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 SOLDADO_L1_DOWN
db 1,-5,-1; subo para que UP y down encajen
db 2,1,-1
db 2,2,-1
db 2,3,-1
db 2,4,-1
db 1,5,-1
db 253
dw SOLDADO_L1
db 1,5,-1; baja una mas
db 255,13,0; nuevo estado, ya sin flag de ruta y con flag de animacion
db 254,32,0; macrosecuencia 32
db 1,0,0; quietooo.!!!!
db 0

ROUTE2; bird
db 30,0,-1
db 10,0,0

```

```

        db 20,2,-1
        db 20,-2,-1
        db 0

ROUTE3; disparo_dere
;-----
        db 40,0,2
        db 255,0
        db 1,0,0
        db 0

ROUTE4; disparo_izq
;-----
        db 40,0,-2
        db 255,0
        db 1,0,0
        db 0

```

Y este sería el ejemplo de programa. Compara su ejecución con la del capítulo 7 para ver la diferencia de velocidad. Notarás una gran diferencia

```

10 MEMORY 23999
20 MODE 0: DEFINT A-Z: CALL &6B78:' install RSX
25 ON BREAK GOSUB 2800
30 CALL &BC02:ink 0,0:'restaura paleta por defecto
50 FOR j=0 TO 31:|SETUPSP,j,0,0:NEXT:'reset sprites
80 |SETLIMITS,12,80,0,186: ' establecemos los limites de la pantalla
de juego
90 x=40:y=100:jump=0:ciclo=40: ' coordenadas del personaje
100 |SETUPSP,0,0,13:|SETUPSP,0,5,0,0:' status del personaje
110 |SETUPSP,0,7,1:|SETUPSP,0,7,32:'secuencia de animacion asignada al
empezar
120 |LOCATESP,0,y,x:'colocamos al sprite (sin imprimirla aun)
123 locate 1,1: print "pulsa Q" : print "para saltar": print "ejemplo
con ruta"
124 print "pulsa SPACE para disparar"
125 PLOT 1,150:DRAW 640,150: PLOT 92,150:DRAW 92,400:'suelo y pared
126 for i=1 to 4:|SETUPSP,10+i,9,26:next:'disparos
127 |MUSIC,0,5: 'comienza a sonar la musica
130 'ciclo de juego -----
150 |AUTOALL,1:|PRINTSPALL,0,1,0
170 ' rutina movimiento personaje -----
172 IF INKEY(47)=0 THEN if espera<ciclo-10 then espera=ciclo: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: ' si estado es
>128 es que estoy saltando (tiene ruta)
174 IF INKEY(67)=0 THEN |SETUPSP,0,0,137:|SETUPSP,0,15,dir:'saltar
180 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 izquierda
193 ciclo=ciclo+1
310 goto 150
2800 |MUSICOFF:MODE 1: INK 0,0:PEN 1

```

Para comprobar que el muñeco se encuentra saltando simplemente consulto el estado mediante un peek (27000). De este modo, sabremos si tiene el flag de enrutamiento activo y en ese caso no pasaremos por las líneas que lo mueven a derecha e izquierda.

#### 10.2.4 Cambios de ruta forzados desde rutas

Podemos cambiar la ruta de un sprite usando un segmento especial. Cuando pongamos 252 en el valor del número de pasos, el comando ROUTEALL interpretará que se debe realizar un cambio de ruta en el sprite. Ejemplo:

**252,2,0**

Este segmento cambia la ruta del sprite, estableciendo la ruta número 2. El tercer parámetro (el cero) no significa nada, es solo un “relleno” para que el segmento mida 3 bytes, pero es imprescindible.

Al igual que con el cambio de estado, el cambio de ruta se ejecuta sin consumir un paso, por lo que siempre se ejecutará el siguiente paso al cambio de ruta, que será el primer paso del primer segmento de la nueva ruta.

Como una ruta puede medir a lo sumo 255 bytes y un segmento ocupa 3 bytes, una ruta puede tener como mucho 84 segmentos. Es posible que necesites construir una ruta aun mas larga y en ese caso podrás hacerlo concatenando el fin de una ruta con un cambio de ruta hacia otra ruta, y puedes concatenar tantas rutas como deseas

#### 10.2.5 Animación forzada desde rutas

Podemos dejar inactivo el flag de animación de un Sprite y animarlo solo en determinados instantes de la ruta usando el código 251. Ejemplo:

**251,0,0**

Esto puede ser muy útil para dar sensación de que un Sprite se acerca en juegos que utilizan técnicas pseudo 3D. Por ejemplo, para un meteorito que se acerca y que queremos que cambie de fotograma para aparecer mas grande. EL meteorito se moverá unas cuantas veces antes de pasar al siguiente tamaño. Este mecanismo es muy similar al de cambio de imagen, solo que te permite definir el cambio de imagen sin especificar explícitamente la imagen, sino simplemente indicando un cambio de frame en la secuencia de animación que tenga asignado el Sprite.



*Fig. 61 Animación forzada desde ruta*

Con este flag puedes usar la misma ruta para un pájaro espacial que se acerca o para un meteorito. Al no indicar la imagen, en cada caso se aplicará la imagen que corresponda a la secuencia que tenga cada sprite.

### 10.2.6 Como construir rutas “dinámicas” (no predefinidas)

Una ruta dinámica es una ruta cuya trayectoria se decide en tiempo de ejecución de tu programa BASIC. Es útil cuando tu programa genera dinámicamente laberintos o circuitos de carreras que un enemigo debe recorrer y que a priori no son conocidas y por lo tanto no pueden ser definidas en el fichero “routes\_mygame.asm”

Para poder hacer una ruta así y asignársela a un Sprite, lo que tenemos que hacer es crear una ruta “vacía” en el fichero “routes\_mygame.asm”, en este caso es la ruta 2

```
; LISTA DE RUTAS
;=====
;pon aqui los nombres de todas las rutas que hagas
ROUTE_LIST
    dw ROUTE0
    dw ROUTE1
    dw ROUTE2
    dw ROUTE3
    dw ROUTE4

; DEFINICION DE CADA RUTA
;=====
ROUTE0; derecha izquierda
    db 10,0,1
    db 10,0,-1
    db 0

ROUTE1; arriba abajo
    db 10,-1,0
    db 10,1,0
    db 0

ROUTE2; ruta dinamica
    ds 100
```

Hemos creado la ruta 2 con 100 bytes libres para llenar desde BASIC. Puede que la ruta que construyamos ocupe menos y en ese caso bastará con reservar menos bytes.

Una vez ensamblada la librería y los graficos debemos buscar la dirección de memoria de la etiqueta “ROUTE2” en la ventana de simbolos de winape. Cuando la tengamos, desde BASIC programaremos la ruta usando POKE a partir de esa dirección de memoria y en las siguientes

POKE mete un byte en una dirección de memoria. Nuestros números deben pertenecer al rango -127..128 y POKE no permite meter números negativos. Para hacerlo debes usar el valor positivo con el que internamente el amstrad representa a los negativos (es decir, el complemento a dos). Para hacerlo basta una operación AND 255

```
10 A=-10
20 PRINT A: REM esto imprime un 10
30 PRINT A AND 255: REM esto imprime un 246 que es el mismo -10
```

En definitiva, si quieras insertar un -10 lo que tienes que instertar es un 246 y usar la misma estrategia para cualquier numero negativo. No te olvides que el ultimo POKE de la ruta debe ser la inserción de un cero, que significa fin de ruta.

### 10.2.7 Programación de rutas que incluyen patrones

Vamos a suponer que quieres hacer una ruta para que un enemigo atraviese la pantalla de derecha a izquierda una y otra vez, suponiendo que partimos de una posición inicial en la que el sprite se encuentra en el extremo derecho de la pantalla

```
ROUTE0; ruta sencilla
    DB 80,0,-1 ; 80 pasos. En cada paso se mueve 1 byte
    DB 1,0,80 ; recolocamos el sprite a su posición original
    DB 0
```

Esta ruta mueve a un sprite dando un paso de 1 byte en cada fotograma. Si quisiésemos que fuese mas despacio, moviéndose 1 byte cada dos fotogramas, podríamos hacer lo siguiente:

```
ROUTE0; ruta no tan sencilla
    DB 1,0,-1
    DB 1,0,0
    DB 0
```

Ahora esta ruta permite mover a un sprite mas despacio, pero no podemos recolocar al sprite a su posición original. Esto es debido a que como la pantalla tiene 80 bytes de ancho, necesitamos 160 segmentos, ya que el sprite avanza 1 byte cada dos segmentos. La ruta resultante sería larguísima y de hecho tendríamos que concatenar 2 rutas pues una ruta solo puede medir 255 bytes (84 segmentos).

La solución optima es definir una ruta corta sin recolocación del sprite y recolocarlo desde BASIC, mediante algo como:

```
80 |SETUPSP,31, 0, 128+16+1: REM rutaible, mov automatico, imprimible
85 |LOCATESP,31,100,80: ' colocado a la derecha de la pantalla.
90 rem ciclo de juego
100 ciclo=ciclo +1
110 |AUTOALL,1: |PRINTSPALL
120 if ciclo MOD 160=0 THEN |LOCATESP,31,100,80: ' recolocacion
130 goto 100
```

Este tipo de estrategias son útiles siempre que no queramos repetir el mismo movimiento en cada fotograma, sino que queremos definir un patrón de repeticion en el que en ciertos fotogramas en sprite se mueve en una dirección y en otros se mueve en otra dirección o incluso no se mueve. Veamos otro ejemplo, una ruta inclinada en la que por cada 3 movimientos en vertical nos movemos uno en horizontal

```
ROUTE0; ruta inclinada
    DB 2,1,0
    DB 1,1,1
    DB 0
```

### 10.2.8 Tipología de rutas

Con todo lo que hemos visto podemos clasificar las rutas en los siguientes tipos:

- **Rutas sin fin cíclicas:** recolocan al sprite o simplemente acaban en las mismas coordenadas donde empezaron
- **Rutas sin fin no cíclicas:** avanzan indefinidamente y no recolocan el sprite por lo que se pueden alejar infinitamente del área de juego, a menos que recoloquemos el sprite desde BASIC
- **Rutas con fin:** en el ultimo paso cambian el estado del sprite desactivando el flag de enrutamiento
- **Rutas encadenadas:** desde una ruta se puede saltar a otra y que esta segunda ruta sea cíclica o no cíclica o tenga fin, o incluso acabe saltando a una tercera ruta.

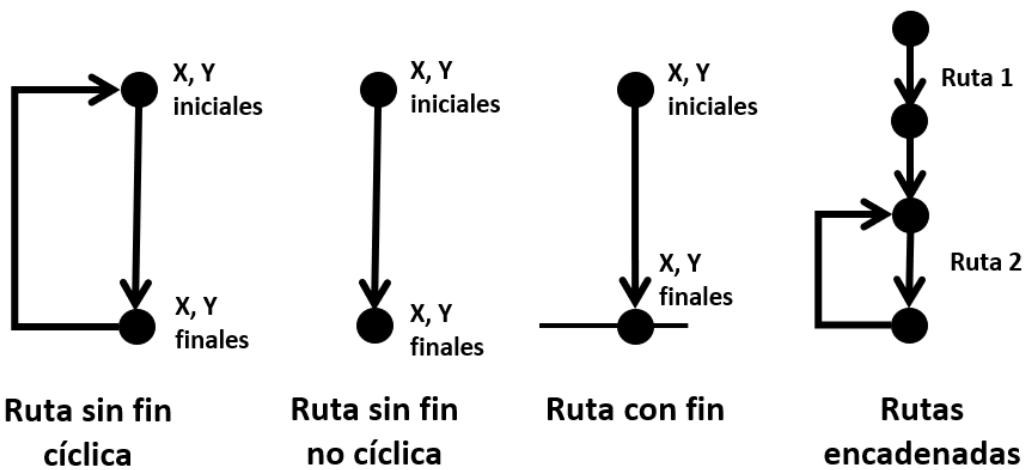


Fig. 62 Tipos de rutas

## 11 Juegos con scroll

La librería 8BP te permite hacer scroll de diferentes modos que se pueden combinar simultáneamente, aunque el método mas importante se basa en el comando |MAP2SP. Las técnicas disponibles te las resumo a continuación:

- **Mediante comandos de movimiento en bloque de sprites:** Una forma sencilla de hacer scroll con 8BP es simplemente crear unos sprites decorativos a los que configuramos su estado para ser movidos por los comandos |MOVERALL y/o |AUTOALL.
- **Mediante MAP2SP:** La idea que subyace al scroll multidireccional proporcionado por MAP2SP en 8BP es sencilla: todos los elementos que se representan en pantalla son sprites, de modo que los elementos del mundo que vamos a imprimir y mover por pantalla son sprites cuyas imágenes asociadas serán montañas, casas, árboles o lo que necesites para construir tu “mundo”. Para seleccionar una porción del mundo y transformarla en una lista de sprites se usa la función |MAP2SP. En cuanto a la función |UMAP permite actualizar el mundo con una porción de un mundo mayor.
- **Mediante el comando |STARS,** el cual te permitirá hacer scroll multidireccional de un banco de 40 pixels que puedes ubicar donde quieras y que puedes mover en diferentes planos y a diferente velocidad.
- **Mediante el comando |RINK,** el cual te permitirá rotar un patrón de tintas, dando sensación de movimiento de avance que puedes utilizar en ciertos tipos de scroll, tales como movimiento de un suelo de ladrillos, agua, etc.

### 11.1 STARS: Scroll de estrellas o tierra moteada

En la librería 8BP dispones de una función muy sencilla de utilizar para crear un efecto de fondo de estrellas que se mueven, dando la sensación de scroll. Se trata de la función |STARS. Esta función es capaz de mover hasta 40 estrellas simultáneamente sin alterar tus sprites, de modo que es como si pasasen “por debajo”

**|STARS,<estrella inicial>,<num estrellas>,<color>,<dy>,<dx>**

Dispones de un banco de estrellas y puedes combinar varios comandos STARS para trabajar con grupos de estrellas a diferente velocidad, dando sensación de planos con distinta profundidad.

El banco de estrellas consiste en 40 pares de bytes representando coordenadas (y,x). Ocupando desde la dirección 42540 hasta 42619 (son 80 bytes en total). Una forma de generar 40 estrellas aleatorias sería (ojo si ya hemos ejecutado DEFINT A-Z el numero 42540 lo debemos poner en hexa porque es mayor de 32768).

```
FOR dir=42540 TO 42618 STEP 2: POKE dir,RND*200: POKE dir+1,RND*80:NEXT
```

Para una descripción detallada del comando, consulta el capítulo de “guía de referencia”. En ese capítulo encontrarás distintos ejemplos para simular estrellas, tierra, estrellas con dos planos de profundidad, lluvia o incluso nieve. Probablemente con imaginación sea posible simular más cosas con esta misma función. Por ejemplo, si colocas las estrellas en secuencias de 2 o tres pixeles en diagonal, en lugar de repartirlas aleatoriamente, podrás conseguir un desplazamiento de movimiento de “segmentos”, algo que podría ser ideal para simular lluvia.

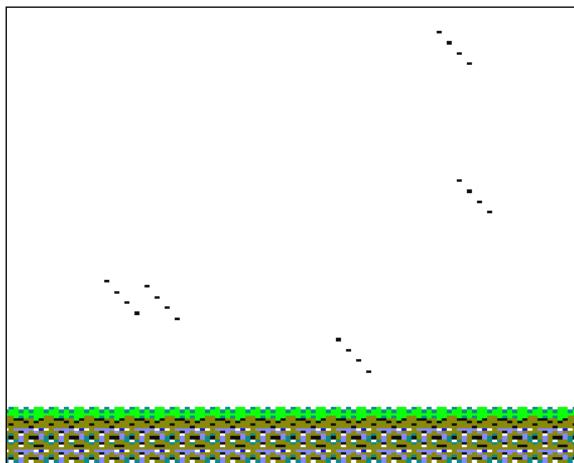


Fig. 63 Efecto de lluvia con STARS

```

10 MEMORY 23999
20 CALL &6B78:' install RSX
40 mode 0:CALL &BC02:'restaura paleta por defecto por si acaso
50 banco=42540
60 FOR dir=banco TO banco+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 'ESCRITURA DE LLUVIA
141 -----
150 |SETLIMITS,0,80,50,200: ' limites de la pantalla de juego
151 cesped=&84d0:|SETUPSP,30,9,cesped:'letra Y es el sprite 31
152 rocas=&84f2:|SETUPSP,21,9,rocas:'letra P es el sprite 21
160 cadena$="YYYYYYYYYYYYYYYYYYYY"
170 |LAYOUT,22,0,@cadena$: 'estos pintan el cesped
180 cadena$="PPPPPPPPPPPPPPPPPPPP"
190 |LAYOUT,23,0,@cadena$: 'pinta una fila de rocas
200 |LAYOUT,24,0,@cadena$: 'pinta otra fila de rocas
210 ----- ciclo de juego-----
211 defint a-z
220 LOCATE 1,10:PRINT "DEMO DE LLUVIA"
221 LOCATE 1,11:PRINT "pulsa ENTER"
230 |STARS,0,40,4,2,-1
240 IF INKEY(18)=0 THEN 300
250 GOTO 230

```

Como el ejemplo de doble plano de estrellas lo tienes en el capítulo de referencia de la librería, aquí vamos a ver un ejemplo en el que una nave espacial sobrevuela un planeta de tierra moteada, con sensación de scroll vertical



*Fig. 64 Efecto de tierra moteada con STARS*

Existe un modo de invocar de forma optimizada el comando STARS y consiste simplemente en invocarlo una primera vez con parámetros y las siguientes veces sin parámetros. El comando asumirá que los valores de los parámetros son los mismos que los de la última invocación con parámetros y ello permite ahorrar tiempo que el interprete BASIC dedica a procesar los parámetros, hasta 1.7ms

```

10 MEMORY 23999
11 'pongo estrellas aleatorias
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:'restaura paleta por defecto por si acaso
26 ink 0,13:'fondo gris
30 FOR j=0 TO 31:|SETUPSP,j,0,&X0:NEXT:'reset sprites
40 |SETLIMITS,12,80,0,186: 'limites de la pantalla de juego

41 ' vamos a crear una nave en el sprite 31
42 |SETUPSP,31,0,&1:' status
43 nave = &a2f8: |SETUPSP,31,9,nave:' asigno imagen al sprite 31
44 x=40:y=150: ' coordenadas de nave

49 '----- ciclo de juego-----
50 |STARS,0,20,5,1,0:' estrellas negras sobre suelo gris
55 gosub 100: ' movimiento de la nave
60 |PRINTSPALL,0,0
70 goto 50

99 ' rutina movimiento nave -----
100 IF INKEY(27)=0 THEN x=x+1:GOTO 120
110 IF INKEY(34)=0 THEN x=x-1
120 |LOCATESP,31,y,x
130 RETURN

```

## 11.2 Scroll usando **MOVEALL** y/o **AUTOALL**

Ahora haciendo uso combinado del movimiento relativo, del scroll de estrellas y del orden en que se imprimen los sprites vamos a ver un ejemplo de cómo simular que una nave sobrevuela un paisaje lunar.

Primeramente, hemos escogido el sprite 31 para nuestra nave, porque eso hará que se imprima la última. Los sprites se imprimen en orden, comenzando por el cero y acabando en el 31. Si un cráter es un sprite inferior a 31 se imprimirá antes que la nave y la nave quedará “por encima”, dando la sensación de que lo está sobrevolando.

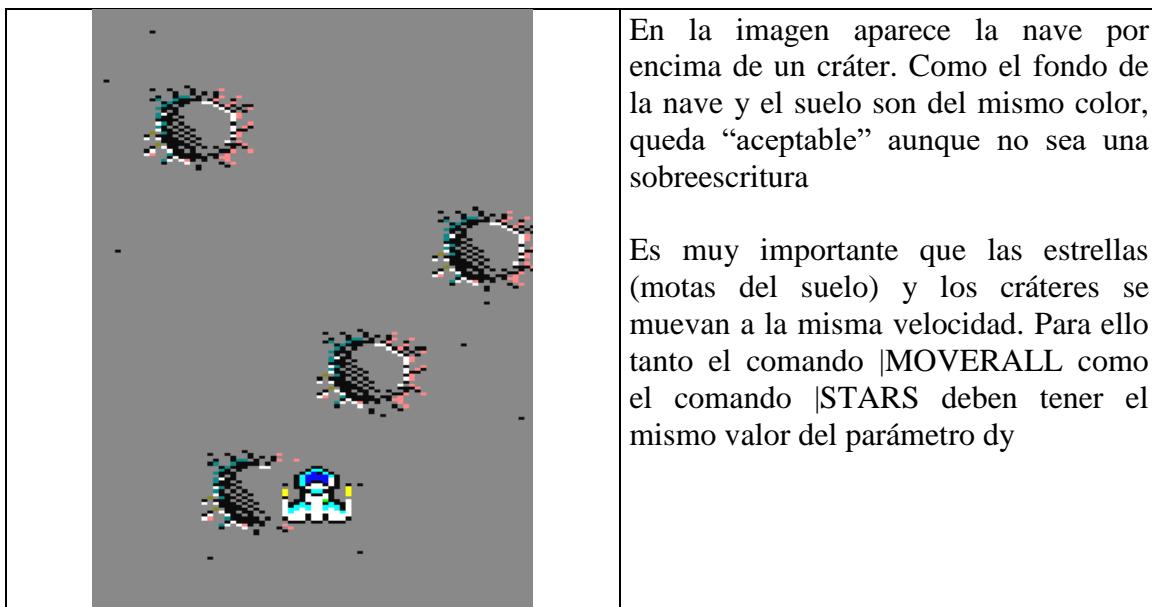


Fig. 65 sobrevolando la luna

Este es el código BASIC :

```
10 MEMORY 23999
11 'pongo estrellas aleatorias
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:'restaura paleta por defecto por si acaso
26 ink 0,13:'fondo gris
30 FOR j=0 TO 31:|SETUPSP,j,0,&x0:NEXT:'reset sprites
40 |SETLIMITS,12,80,0,186: ' limites de la pantalla de juego

41 ' vamos a crear una nave en el sprite 31
42 |SETUPSP,31,0,&1:' status
43 nave = &a2f8: |SETUPSP,31,9,nave:' asigno imagen al sprite 31
45 x=40:y=150: ' coordenadas de nave

46 ' ahora los crateres
47 crater=&a39a: cy%=0
48 for i=0 to 3 : |SETUPSP,i,9,crater:
49 |SETUPSP,i,0,&x10001: ' impresion y movimiento relativo
50 x(i)=rnd*40+20:y(i)=i*40
60 |locatesp,i,y(i),x(i)
70 next
```

```

71 t=0

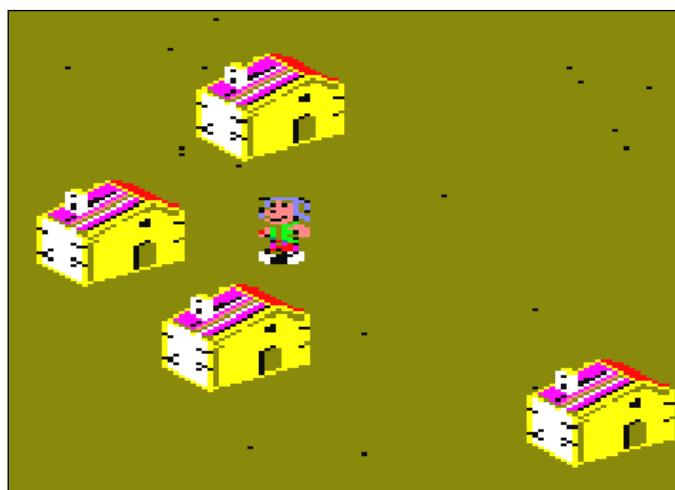
80 ----- ciclo de juego-----
81 |STARS,0,20,5,3,0: ' movimiento estrellas negras
82 gosub 100: ' movimiento de la nave
83 |MOVERALL,3,0: 'movimiento de crateres
84 t=t+1: if t> 10 then t=0:gosub 200:' control de crateres
90 |PRINTSPALL,0,0: ' impresion de nave y crateres
91 goto 81

99 ' rutina movimiento nave -----
100 IF INKEY(27)=0 THEN x=x+1:GOTO 120
110 IF INKEY(34)=0 THEN x=x-1
120 |LOCATESP,31,y,x
130 RETURN

199 ' control de reentrada de crateres
200 c=c+1: if c=6 then c=0
220 |PEEK,27001+c*16,@cy%
230 if cy%>200 then |POKE,27001+c*16,-20
240 return

```

Vamos a ver un último ejemplo que utiliza movimiento relativo para dar la sensación de scroll, usando sprites con dibujos de casas, un suelo moteado y un personaje situado en el centro que según la dirección que tome, hace que todo se mueva entorno a él. Es un ejemplo muy básico, pero te da una idea del potencial de estas funciones. ¡Aquí lo que se mueve es todo el pueblo!



*Fig. 66 El pueblo entero se mueve*

```

10 MEMORY 23999
20 MODE 0: call &6b78
30 DEFINT a-z
240 INK 0,12
241 border 7
250 FOR i=0 TO 31
260 |SETUPSP,i,0,&X0
270 NEXT
280 FOR i=0 TO 3

```

```

290 |SETUPSP,i,0,&X10001
300 |SETUPSP,i,9,&A01c:rem casas
301 |LOCATESP,i,RND*150+50,rnd*60+10
310 NEXT
320 |SETUPSP,31,7,6: rem personaje
330 |LOCATESP,31,90,38
340 |SETUPSP,31,0,&X1111
400 xa=0:ya=0
410 IF INKEY(27)=0 THEN xa=-1:
420 IF INKEY(34)=0 THEN xa+=1:
430 IF INKEY(67)=0 THEN ya+=2
440 IF INKEY(69)=0 THEN ya-=2
450 |MOVERALL,ya,xa
460 |PRINTSPALL,1,0
470 |STARS,1,20,5,ya,xa
480 GOTO 410

```

### 11.3 Técnica del “manchado”

La técnica para pintar montañas en juegos con scroll horizontal y lagos en juegos con scroll vertical es la misma. Lo que haremos es pintar solo el comienzo de la montaña, mediante un sprite que nos sirve para pintar su lado izquierdo. Pondremos tantos como queramos. En este caso yo he puesto tres

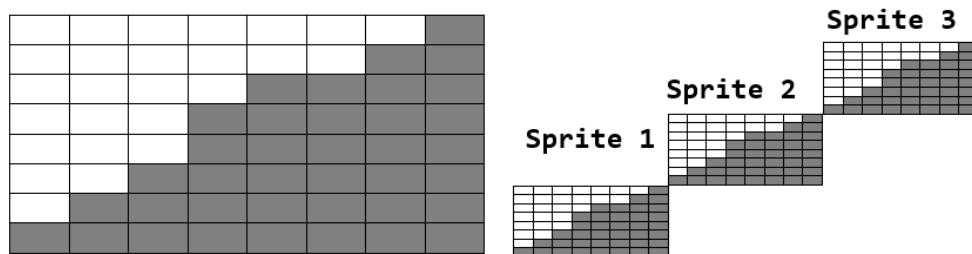


Fig. 67. Definir la ladera de una montaña con varios sprites

Hacemos lo mismo con una imagen espejada que asociaremos a otros 3 sprites, y los situaremos a la derecha, construyendo el lateral derecho de la montaña. Cuidado de que la imagen espejada al menos tenga las dos últimas columnas de pixels a cero. Esto le permitirá borrarse a sí misma al avanzar a la izquierda. Ten en cuenta que en 8BP los sprites se mueven byte a byte (dos pixels a la vez).

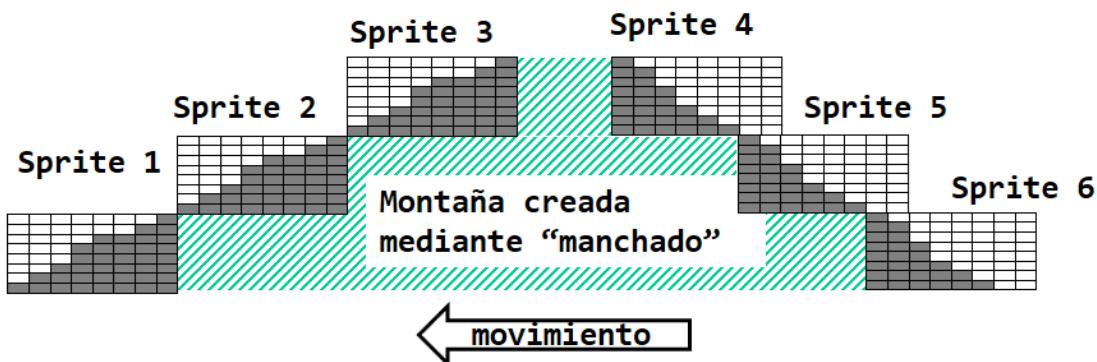
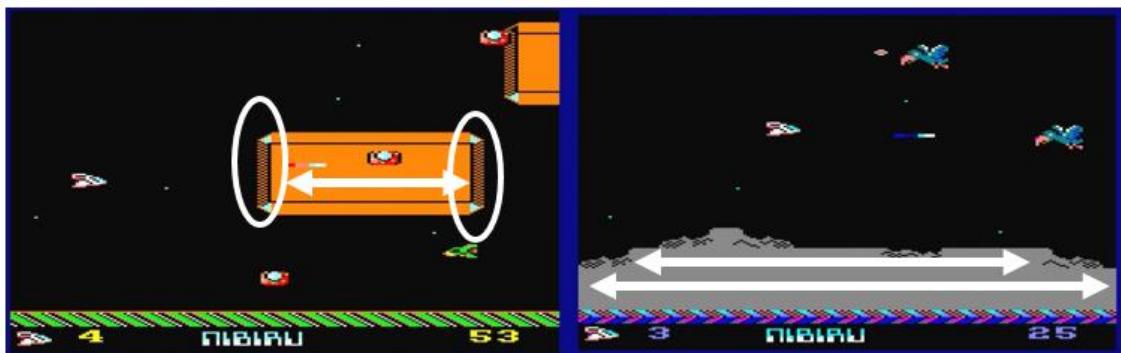


Fig. 68 Montaña creada con 6 sprites y la técnica del “manchado”

Al mover todos los sprites hacia la izquierda mediante movimiento automático o relativo, los sprites de la izquierda empezarán a “manchar” el fondo y por consiguiente

“rellenando” la montaña, al tiempo los sprites de la derecha empezarán a limpiarlo. Si la montaña aparece poco a poco entrando en la pantalla, parecerá un sprite de una montaña enorme, cuando en realidad se trata de 6 pequeños sprites.

El videojuego “Nibiru” hace uso de la técnica del “manchado” para dibujar las montañas y otros elementos grandes, en combinación con el comando MAP2SP que luego veremos.



*Fig. 69. Ejemplos de la técnica del manchado*

También utilicé la técnica del manchado en el videojuego “Eridu”, donde enormes montañas se mueven suavemente por la pantalla.



*Fig. 70. Técnica del manchado en “Eridu”*

En el caso de un juego de scroll vertical, si queremos pintar un lago sobre un terreno marrón, haremos lo mismo, unos sprites que van “manchando” el terreno y otros más lejos que lo van “limpiando”, aparentando que se trata de un lago enorme, de una sola pieza.

Solo debes tener una precaución, y es que las naves no sobrevuelen el lago o ¡tu “truco” quedará al descubierto! En caso de que quieras que sea posible que se sobrevuele el lago, entonces deberás usar sobreescritura en los sprites, tal y como se hace en la fase 2 del videojuego “Nibiru”, donde tu nave y las naves enemigas pueden pasar por encima de grandes rectángulos de color sin destruirlos.



## 11.4 MAP2SP: Scroll basado en un mapa del mundo

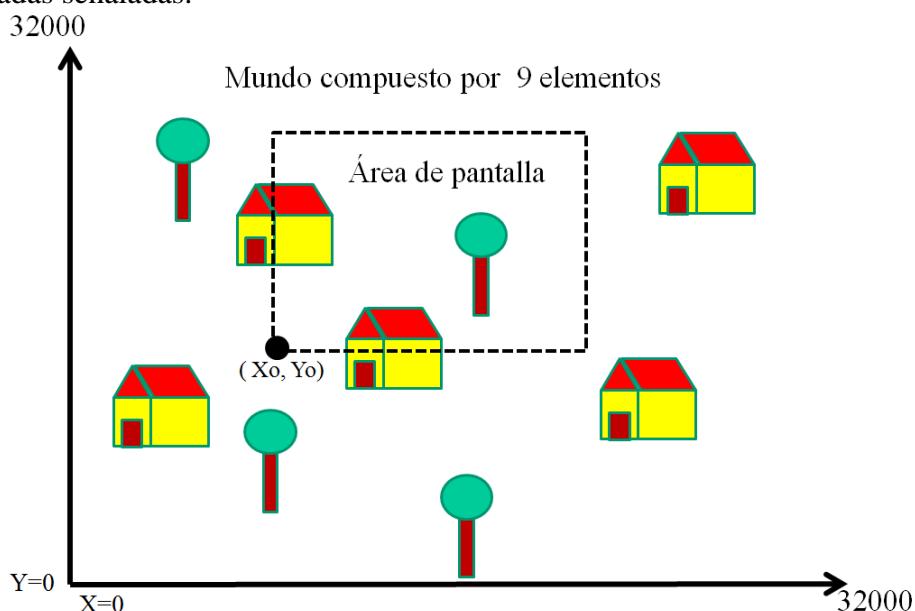
Todas las técnicas anteriores son perfectamente válidas para hacer scroll, e incluso compatibles con lo que vamos a ver ahora, que es la técnica fundamental que te va a permitir diseñar un “mapa del mundo” y hacer que tu personaje o tu nave se desplace por él, con tan sólo una línea de código.

La idea es sencilla: crearemos una lista de elementos que conforman el mapa del mundo (hasta 82 elementos a los que llamaremos “elementos de mapa” o “map ítems”). Cada elemento está descrito por las coordenadas Y,X donde se ubica y la dirección de memoria donde se encuentra la imagen del elemento en cuestión (una casa, un árbol, etc.). La imagen asociada a un elemento de mapa podrá tener el tamaño que quieras. Las coordenadas de cada elemento serán un número entero positivo, desde 0 hasta 32000.

Una vez creado el mapa, invocaremos la función:

**|MAP2SP, Yo, Xo**

Esta función analiza la lista de elementos del mundo y determina cuáles de ellos están siendo visualizados si el mundo se observa colocando la esquina inferior de la pantalla en las coordenadas (Yo, Xo). La función transforma en sprites los “map ítems”, ocupando las posiciones de la tabla de sprites de la cero en adelante. Esto puede consumir muchos o pocos sprites, dependiendo de la densidad de map ítems que tengas. En otra invocación posterior a la misma función, los map ítems que ya no están presentes en la escena no consumirán sprites en la tabla, y otros map ítems tomarán el relevo. Esto significa que la función MAP2SP consume un número de sprites variable e indeterminado, que depende del número de map ítems visibles en pantalla en cada momento. En el ejemplo siguiente usaría 3 sprites al invocar a MAP2SP en las coordenadas señaladas.



*Fig. 71 Mapa del mundo y MAP2SP*

Si usas este mecanismo, tu personaje y los enemigos deben usar los sprites desde 31 hacia abajo, de ese modo evitarás posibles choques entre los sprites que usa el

mecanismo de scroll y tus personajes. Si por un casual MAP2SP se encuentra con mas de 32 items para traducir a sprites, ignorará los que excedan de 32.

Debes invocar MAP2SP en cada ciclo de juego o al menos cada vez que modifiques las coordenadas del punto de vista desde donde quieras visualizar el mundo. La ventana "deslizante" con la que cazas las imágenes que se transforman en sprites siempre mide lo que mide la pantalla (80 bytes x 200 lineas) con independencia de como configures el comando SETLIMITS. Aunque SETLIMITS establezca un área de pintado muy pequeño, se transforman en sprites todo lo que haya cazado la ventana.

Los sprites creados por MAP2SP se crean por defecto con estado 3, es decir, con el flag de impresión activo (PRINTSPALL lo imprime) y con el flag de colisionable activo (COLSP colisionará con el). Si necesitas que los sprites sean creados con otro estado, simplemente debes invocar una vez al comando MAP2SP con un solo parámetro indicando el estado con el que se deben crear los sprites. Con una única invocación de este tipo, queda configurado el comando para las siguientes invocaciones con coordenadas.

#### |MAP2SP, <status>

Ejemplo:

|MAP2SP, 1 : REM esto configura al comando MAP2SP para que se impriman pero no sean colisionables

Una vez aclarado el concepto vamos a revisar en detalle cómo se especifica el mapa del mundo y un ejemplo de uso de la función MAP2SP.

#### 11.4.1 Mapa del mundo (Map Table)

La tabla donde daremos de alta todos los elementos del mapa se llama MAP\_TABLE y se especifica en un fichero .asm llamado map\_table\_tujuego.asm

Esta tabla contiene los ítems que definen las imágenes del mapa del mundo para tus juegos con scroll. La tabla se ensambla en la misma dirección de memoria que el LAYOUT, es decir, en la dirección 42040. Esto significa que no se pueden usar simultáneamente el layout y el mapa del mundo, pero no es un problema ya que un juego con scroll no va a usar el layout y viceversa. Además, la restricción es únicamente que no se pueden usar a la vez, pero un juego podría tener una fase en la que usa el layout y otra en la que hace scroll basado en mapa del mundo.

La tabla de entradas del mapa del mundo comienza con 3 parámetros globales (que ocupan 5 bytes en total) y una lista de "map items", los cuales están descritos por 3 parámetros cada uno (x, y, dirección de imagen)

La lista puede contener hasta 82 items, pero el número de ítems se puede limitar con uno de los parámetros globales. La lista, como máximo, ocupa los 5 bytes iniciales + 82 items x 6 bytes = 5+492=497 bytes. Si metiésemos un ítem mas, superaríamos los 500Bytes reservados para el mapa ( que son los mismos reservados para el Layout).

La tabla comienza con 3 parámetros:

- El alto máximo de cualquier map ítem

- Ancho máximo de cualquier map ítem (debe expresarse como un número negativo)
- Número de ítems (como máximo será 82)

Los dos primeros parámetros son importantes para chequear cuando un sprite puede estar parcialmente apareciendo en pantalla, ya que la función MAP2SP no conoce ni averigua el ancho ni el alto de cada imagen. Tan solo conoce donde está situado el map ítem y suponiendo el alto y ancho máximos, averigua si ese ítem puede estar entrando en la pantalla. En caso de que así sea, se crea un sprite a partir del map ítem. Si esos dos parámetros se ponen a cero, será necesario que la esquina superior izquierda del map ítem esté dentro de la pantalla para que dicho ítem sea transformado en un sprite.

Cada ítem es una tupla de 3 parámetros precedidos por el nemónico “DW”:  
**DW Y, X, <imagen>**

Veamos un ejemplo del fichero llamado map\_table\_tujuego.asm

```
;MAP TABLE
;-----
; primero 3 parametros antes de la lista de "map items"
dw 50; maximo alto de un sprite por si se cuela por arriba y ya hay
que pintar parte de el
dw -40; maximo ancho de un sprite por si se cuela por la izquierda
(numero negativo)
db 82; numero de elementos del mapa.como mucho debe ser 82

; a partir de aqui comienzan los items
dw 100,10,CASA; 1
dw 50,-10,CACTUS;2
dw 210,0,CASA;3
dw 200,20,CACTUS;4
dw 100,40,CASA;5
dw 160,60,CASA;6
dw 70,70,CASA;7
dw 175,40,CACTUS;8
dw 10,50,CASA;9
dw 250,50,CASA;10
dw 260,70,CASA;11
dw 290,60,CACTUS;12
dw 180,90,CASA;13
dw 60,100,CASA;14

...
```

Para diseñar tu mundo te recomiendo que cojas un cuaderno a cuadros y vayas dibujando sobre él los elementos que quieras que tenga tu mundo. Cada cuadradito del cuaderno puede representar una cantidad fija como 8 píxeles o 25 píxeles. El caso es que debes tomarte tu tiempo en dibujar el mundo que deseas y el modo en que se va a recorrer. Por ejemplo, hay juegos tipo “Gauntlet” multidireccionales y otros de scroll vertical como el comando. Tú eliges, pero en cualquier caso hazlo con tiempo y paciencia y el resultado valdrá la pena.

Cada fase de tu juego puede ser un mapa. En 8BP puedes cambiar el mapa cuando quieras usando funciones POKE. Yo normalmente utilizo 1KB fuera del espacio de memoria usado por 8BP, por ejemplo, desde la 23000 hasta la 24000, para almacenar todas las fases (mapas) del juego y cada vez que entro en una fase cargo en la dirección 42040 el mapa correspondiente haciendo PEEK y POKE. Es decir, que mi fichero de mapa lo construyo en la dirección 23000 y ocupa 1Kbyte, dejando 23 KB para mi programa BASIC. Para que esta información de mapas no sea machacada por el basic, tendré que hacer un MEMORY 22999 al principio del juego.

#### 11.4.2 Uso de la función MAP2SP

Ahora vamos a ver un ejemplo de uso de esta función. Básicamente hay que invocarla una vez en cada ciclo de juego con las nuevas coordenadas del origen desde donde se observa el mundo.

La función creará un numero de sprites variable desde el sprite 0 en adelante y al crearlos lo va a hacer con sus coordenadas de pantalla adaptadas. Es decir, aunque un map ítem tenga una coordenada  $x=100$ , si el origen móvil lo ubicamos en la posición  $x=90$  entonces ese sprite será creado con la coordenada de pantalla  $x'=x-90=10$ . La coordenada en el eje Y tendrá en cuenta que el eje Y en el Amstrad crece hacia abajo, mientras que el mapa del mundo crece hacia arriba. Por ello la coordenada Y es adaptada usando la ecuación  $Y'=200-(Y-Y_{orig})$ . Pero no te preocupes, esta adaptación ya la hace la función MAP2SP. Tu solo tienes que ir cambiando el origen móvil desde donde se debe visualizar el mapa del mundo.

En este mini juego se ha realizado un mundo compuesto de casas y cactus y nuestro personaje camina entre los elementos. En este ejemplo, en caso de colisión (detectado con COLSPALL), el personaje no podrá continuar. En un juego de aviones en el que los map ítems sean “sobrevolables”, podríamos parametrizar la colisión para que solo se detecten colisiones con enemigos y disparos y no con elementos de fondo, usando COLSP, 32, <sprite inicial>, <sprite\_final>.



*Fig. 72 Mini juego con scroll inspirado en “commando”*

Ahora veamos el listado, como ves, es muy pequeño, pero lo tiene todo: scroll multidireccional, lectura de teclado, cambio de secuencias de animación del personaje, detección de colisión, música...

**10 MEMORY 23999**

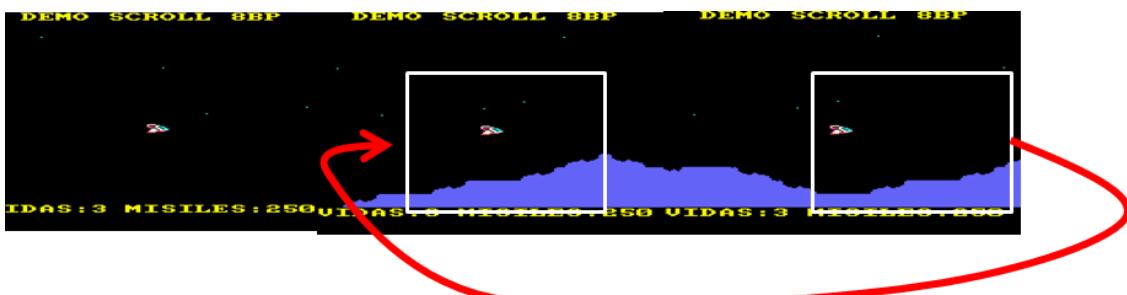
**20 MODE 0**

```

30 ON BREAK GOSUB 280
40 CALL &6B78
50 DEFINT a-z
60 INK 0,12
70 FOR y=0 TO 400 STEP 2
80 PLOT 0,y,10:DRAW 78,y
90 PLOT 640-80,y,10:DRAW 640,y
100 NEXT
110 x=0:y=0
120 |SETUPSP,31,0,&X100001
130 |SETUPSP,31,7,1:dir=1: ' direccion inicial hacia arriba
140 |locatesp,31,100,36
150 |MUSIC,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: REM colision en cuanto hay un mnimo solape
190 'comienza ciclo de juego
200 IF INKEY(27)=0 THEN x=x+1:IF dir<>3 THEN dir=3:|SETUPSP,31,7,3:
GOTO 220
210 IF INKEY(34)=0 THEN x=x-1:IF x<0 THEN x=0:ELSE IF dir<>4 THEN
dir=4:|SETUPSP,31,7,4
220 IF INKEY(67)=0 THEN y=y+2:IF x=xa AND dir <> 1 THEN
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 IF xa=x AND ya=y THEN 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 |MUSICOFF:MODE 1: INK 0,0: PEN 1

```

Vamos a ver ahora otro ejemplo de scroll horizontal, donde se ha conseguido un efecto interesante de mapa del mundo “infinito”, haciendo que el final del mapa sea igual al principio y provocando un salto brusco cuando Xo llega a un determinado valor. De hecho, este mapa del mundo solo tiene 13 elementos



*Fig. 73 Mapa del mundo “infinito”*

Este es el mapa que se ha utilizado

```

_MAP_TABLE
; primero 3 parametros antes de la lista de "map items"

```

```

dw 50; maximo alto de un sprite por si se cuela por arriba y ya hay
que pintar parte de el
dw -18; ancho maximo de cualquier map item. debe expresarse como
numero negativo
db 13; numero de elementos del mapa. como mucho debe ser 82

; a partir de aqui comienzan los items
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
;;aqui repito elementos para encajar con la posicion 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
;-----

```

Y este el programa BASIC, donde he destacado la línea en la que el mundo vuelve atrás sin que el jugador note nada.

```

10 MEMORY 23999
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 CALL &6B78
50 DEFINT a-z
51 INK 0,0
52 |MUSIC,0,5
110 xo=0:yo=0
111 x=36:y=100
120 |SETUPSP,31,0,&X100001
130 |SETUPSP,31,7,1:dir=1:' direccion inicial hacia arriba
140 |LOCATESP,31,y,x
160 |SETLIMITS,0,80,0,176: |PRINTSPALL,0,1,0
161 LOCATE 1,23 :PEN 1: PRINT "VIDAS:3 MISILES:250"
162 LOCATE 1,1:PRINT " DEMO SCROLL 8BP"
170 col%=32:sp%=32:|COLSPALL,@sp%,@col%
180 |COLSP, 34, 0, 0: REM colision en cuanto hay un mnimo solape
190 'comienza ciclo de juego
200 IF INKEY(27)=0 THEN x=x+1: GOTO 220
210 IF INKEY(34)=0 THEN x=x-1:IF x<0 THEN x=0
220 IF INKEY(69)=0 THEN y=y+2: GOTO 240
230 IF INKEY(67)=0 THEN y=y-2:IF y<0 THEN y=0
240 IF xa=x AND ya=y THEN dir=0 ELSE |ANIMA,31
250 |MAP2SP,yo,xo:|COLSPALL:IF col<32 THEN END
260 |PRINTSPALL

```

```

261 ciclo=ciclo +1: IF ciclo=2 THEN |STARS,0,5,2,0,-1:ciclo=0
262 xo=xo+1:IF xo=210 THEN xo=100
263 |LOCATESP,31,y,x
270 GOTO 200
280 |MUSICOFF:MODE 1: INK 0,0: PEN 1

```

### 11.4.3 Ejemplo de fichero de fases

Si quieres tener varias fases en un juego con scroll, como he comentado antes puedes tenerlas precargadas en un área de memoria. Por ejemplo, puedes ensamblar las fases en la dirección 23000 y así dispones de 1000 bytes para almacenar varios mapas del mundo, ya que 8BP comienza en la dirección 24000. En ese caso tu juego tendrá que empezar con un MEMORY 22999

El videojuego “Nibiru” lo hace así, aunque fue creado cuando 8BP empezaba en la dirección 26000 (fue creado con la 8BP v26), por lo que almacena el mapa a partir de la 25000. Para cargar una fase simplemente lee la zona donde se ha ensamblado cada fase y la escribe sobre la dirección donde debe encontrarse la tabla del mundo antes de empezar a jugar en esa fase. En estas tres líneas de BASIC se muestra como se copia la fase 1 del juego (&61a8 = 25000)

```

310 ' pokes del mapa del mundo
320 dirmap!=42040:FOR i!=&61A8 TO &620D
330 dato=PEEK(i!):POKE dirmap!,dato:dirmap!=dirmap!+1
340 NEXT

```

Existe una forma mas rápida de cargar las fases, mediante el comando |UMAP que más adelante explicaré, pero este bucle for con POKES es perfectamente válido.

Por último, a continuación te muestro el fichero de fases del juego “Nibiru”, que ensamblamos en la dirección 25000 (recordemos que la versión de 8bp con la que fue creado nibiru empezaba en la 26000, por lo que desde la 25000 hasta la 26000 había 1000 bytes libres. Con la versión actual de 8BP ensamblaríamos las fases sin llegar a la dirección 24000)

```

org 25000
;FASE1
=====
_START_FASE1
; primero 3 parametros antes de la lista de "map items"
dw 50; maximo alto de un sprite por si se cuela por arriba y ya hay que
pintar parte de el
dw -18; ancho maximo de cualquier map item. debe expresarse como numero
negativo
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
;aqui repito elementos para encajar con la posicion 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
_END_FASE1
=====
;FASE2
=====
_START_FASE2
dw 50; maximo alto de un sprite por si se cuela por arriba y ya hay que
pintar parte de el
dw -6; ancho maximo de cualquier map item. debe expresarse como numero
negativo
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,PLACA_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,CUPULA2_OV
_END_FASE2
=====
;FASE3
=====
_START_FASE3
dw 50; maximo alto de un sprite por si se cuela por arriba y ya hay que
pintar parte de el
dw -80; ancho maximo de cualquier map item. debe expresarse como numero
negativo
db 4
dw 40,0,MAR; 1
dw 40,80,MAR; 2
dw 189,0,NUBES; 2
dw 189,80,NUBES; 2
_END_FASE3

```

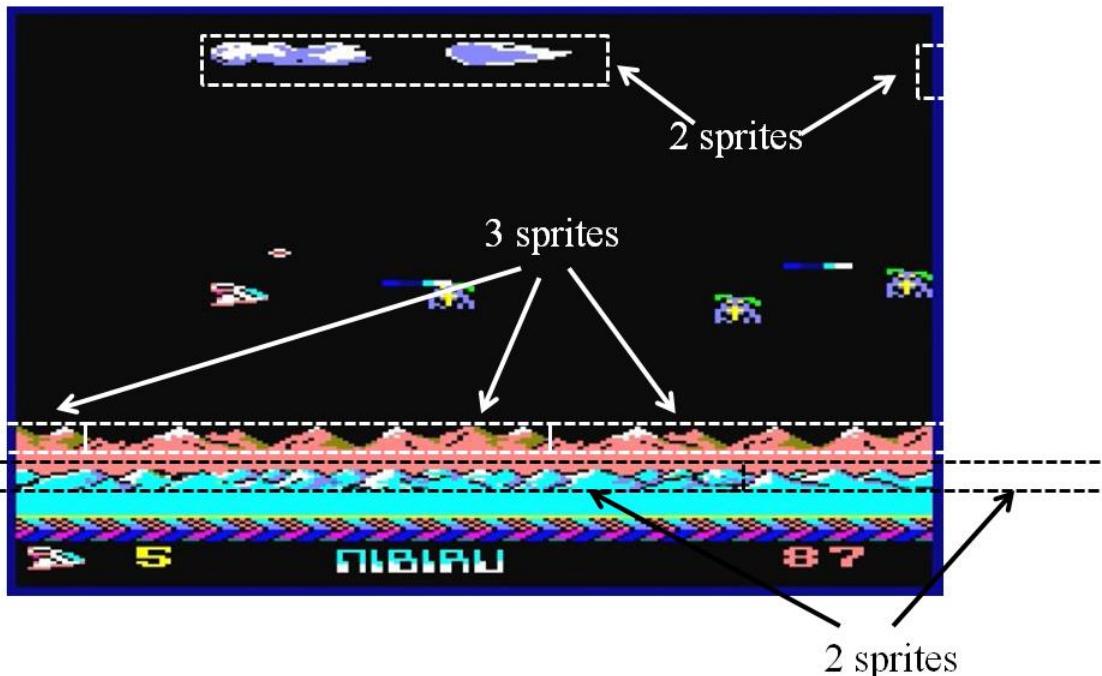
## 11.5 Scroll Parallax

Vamos a ver como se puede hacer un scroll parallax, es decir, con varios “planos” a diferente velocidad. Quizás se te ocurra a ti otra forma de hacerlo, esto es solo una posible manera, empleada en el juego “Nibiru”.

Lo primero que debes saber es que es mucho más rápido imprimir un sprite horizontal muy largo que muchos sprites pequeños. Esto es debido a las operaciones que hay que realizar tras terminar de pintar cada scanline de un sprite. Por el mismo motivo es

mucho más rápido imprimir un sprite horizontal muy grande que un sprite vertical del mismo tamaño.

Colocaremos en el mapa del mundo dos sprites gigantes para hacer el agua. Yo hice uno de 160 x8 de modo que puse dos para que cuando se desplazase empezase a aparecer el siguiente. Al recorrer toda la pantalla, la función MAP2SP se vuelve a colocar en x=0 y todo se repite indefinidamente. En el mapa del mundo también puse dos sprites para las nubes.



*Fig. 74 scroll parallax*

Para las montañas usé 3 sprites normales, fuera del mapa del mundo. Les di movimiento automático en su flag de status y les hice moverse hacia la izquierda. Pero en los ciclos impares les desactive tanto el flag de impresión como el flag de movimiento automático, de modo que solo se mueven e imprimen uno de cada dos ciclos de juego, logrando una velocidad mitad que la que lleva el agua. Los sprites de las montañas son el 16,17 y 18 y con estos pokes se actúa sobre su byte de status.

```
mc=ciclo AND 1: IF mc THEN POKE 27256,0: POKE 27272,0: POKE
27288,0 ELSE POKE 27256,11: POKE 27272,11: POKE 27288,11
```

En el ejemplo “mc” es la variable que determina si el ciclo es par o impar.

## 11.6 Actualización dinámica del mapa: |UMAP

Puede que en nuestro juego necesitemos un mapa de mas de 82 elementos. O bien simplemente queremos que el comando |MAP2SP se ejecute mas rápido usando un mapa mas pequeño que actualicemos periódicamente. ¡O bien queremos ambas cosas!

Para ello desde la versión 32 de 8BP, existe el comando |UMAP (abreviatura de “UPDATE MAP”). Este comando actualiza el mapa con información ubicada en otra zona de memoria donde tengamos un mapa mayor. El comando hace que se reconstruya

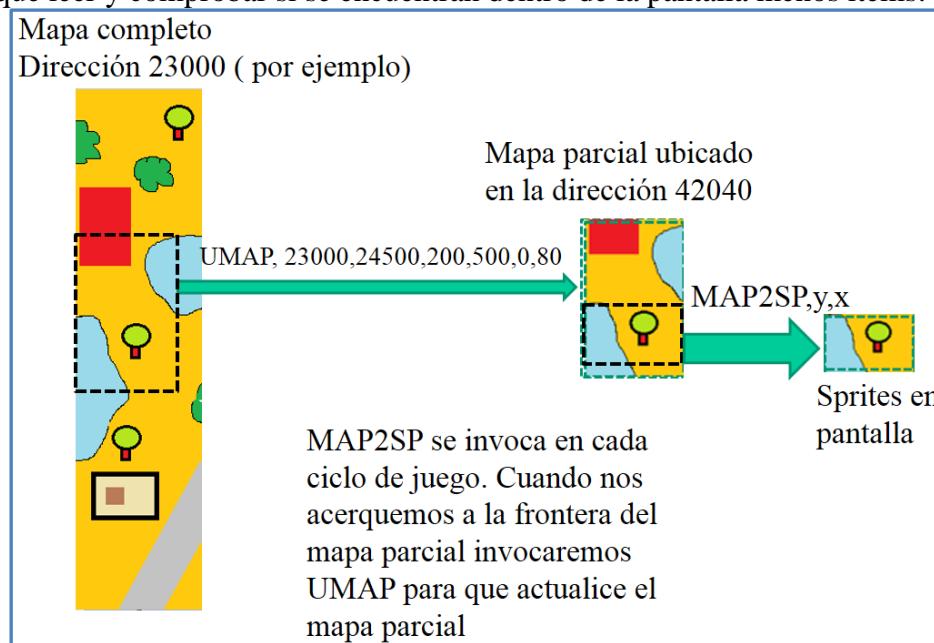
el mapa por completo, incluyendo solo aquellos ítems que cumplan unos determinados rangos de coordenadas X, Y

**|UMAP, <map\_ini>, <map\_fin>, <y\_ini>, <y\_fin>, <x\_ini>, <x\_fin>**

UMAP no es un simple copiado de elementos. Es un copiado “selectivo”. Por ejemplo, si tenemos un mapa ubicado en la dirección 22500 que ocupa 1500bytes y queremos que se actualice el mapa con las coordenadas de nuestro personaje, con margen suficiente para avanzar en la coordenada Y hasta 100 lineas y en coordenada x hasta 20 bytes en todas direcciones:

**|UMAP, 22500,23999, y-100, y+100, x-20, x+20**

Este comando chequeara las coordenadas de los ítems localizados en el mapa de la dirección 22500 y si se encuentran dentro de los márgenes de X, Y que hemos puesto, se copiaran en la zona de memoria que 8BP usa para el comando |MAP2SP, es decir, los copiará a partir de la dirección 42040. Eso si, tan solo copiará los que cumplen la condición. Al ser menos items, el comando |MAP2SP se ejecutará mas rápido pues tendrá que leer y comprobar si se encuentran dentro de la pantalla menos items.



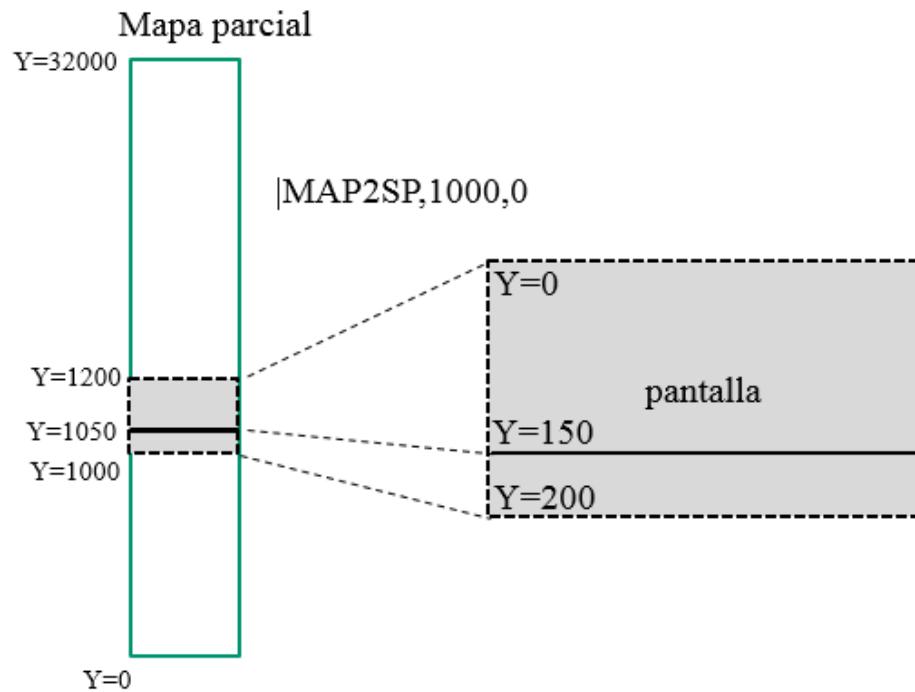
**Fig. 75 UMAP**

Periódicamente (no en cada ciclo de juego) podemos actualizar el mapa con UMAP y así podemos crear mundos muy grandes con muchos elementos y a la vez conseguiremos más velocidad en MAP2SP. El comando UMAP es muy rápido, pero invocarlo en cada ciclo de juego no tiene sentido ya que MAP2SP puede trabajar con un mapa mucho más grande que lo que cabe en pantalla y podemos invocar a |UMAP únicamente cuando necesitemos zonas del mapa original que no están presentes en el mapa parcial. Yo lo he usado en el juego de carreras de coches “3D Racing one”, pues el mapa del circuito era muy grande (excedía de 82 elementos) y lo que hice fue invocar |UMAP periódicamente a medida que el coche avanza.

En la dirección del mapa completo (en el ejemplo la 22500), únicamente tendremos una lista de ítems. **No tendremos los 3 datos iniciales** que contiene el mapa de la 42040

(me refiero al alto máximo de cualquier map ítem, el ancho máximo de cualquier map ítem y el número de ítems). El número de ítems lo actualiza |UMAP (según cuantos ítems cumplan los márgenes impuestos). Los otros dos parámetros los fijarás tu a tu gusto en el fichero “map\_table\_tujuego.asm”.

El comando |UMAP mete los items en un mapa parcial en un orden pensado para que el mapa parcial quede ordenado según la coordenada Y de pantalla. Lo normal es que tu edites tu mapa global en un fichero llamado “misupermapa.asm” o algo así. Y lo ensambles en la 22500 (por ejemplo). En dicho fichero escribirás uno por uno los ítems de tu mapa, en orden ascendente de coordenada Y. Pues bien, para conseguir que los sprites salgan ya ordenados por coordenada Y (en orden ascendente de coordenada de pantalla), el comando |UMAP los lee desde el final hasta el principio. De ese modo los sprites que se vayan generando posteriormente con |MAP2SP estarán ordenados por coordenada Y de pantalla. Recuerda que la pantalla usa un sistema de coordenadas inverso respecto el mapa, es decir, la coordenada 150 de la pantalla es la coordenada 50 del mapa (en el caso MAP2SP,0,0). Si no entiendes esto muy bien, no te preocupes. Es algo del funcionamiento interno de UMAP y MAP2SP simplemente para hacerlo más eficiente. En la siguiente figura he representado el mapa y la pantalla del CPC. Como ves el mapa puede ser muy grande pero además sus coordenadas crecen hacia arriba mientras que las coordenadas de pantalla crecen hacia abajo.



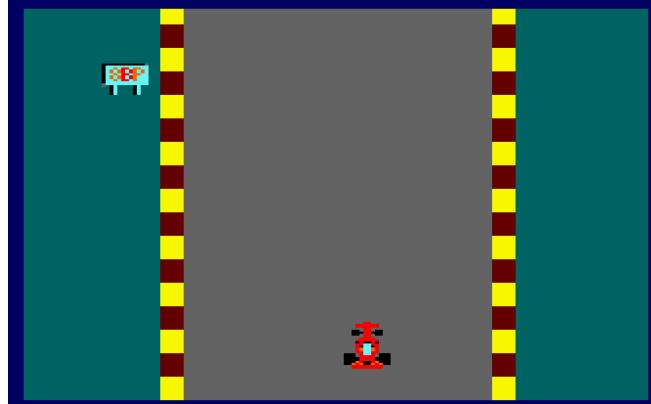
*Fig. 76 MAPA y Pantalla*

## 11.7 Animación y scroll por tintas: comando RINK

Existen juegos que requieren mover grandes bloques de memoria de pantalla para dar sensación de movimiento, como es el caso de las franjas laterales de los juegos de carreras o bloques grandes de ladrillos o tierra. El comando MAP2SP permite hacer

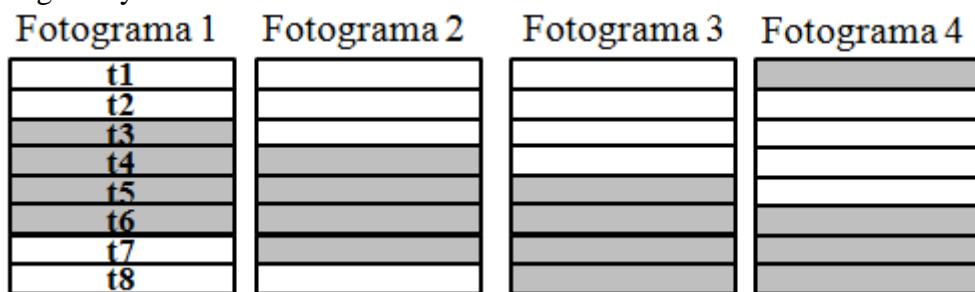
todo eso, pero la velocidad no es trepidante porque gasta mucha CPU moviendo sprites. La animación por tintas es el complemento perfecto en estos casos.

En ordenadores como el AMSTRAD con su potente paleta de 16 colores simultáneos, muchos juegos hacen uso de la animación por tintas. Un claro ejemplo son algunos juegos de coches cuyas franjas laterales de carretera se prestan a este tipo de animación.



*Fig. 77 Franjas animadas por tintas*

La animación por tintas consiste en definir un conjunto de tintas sobre las que se va a hacer rotar un conjunto de colores. Vamos a ver un ejemplo con unas franjas blancas/grises y 8 tintas:



*Fig. 78 Animación por tintas*

Básicamente para dar sensación de movimiento lo que hay que hacer primeramente es asignar los colores a las tintas que van a rotar. En este caso los colores blanco (26) y gris (13) se asignan a las tintas t1..t8. Vamos a suponer que la tinta t1 es la 8, de modo que la tinta t8 será la 15. El resto de tintas (0 a 7) las usaremos para los sprites. En cada instante de tiempo habrá que reasignar los valores de las 8 tintas para dar la sensación de rotación. Esto es lo que hace el comando RINK (abreviatura de “Rotate INK”).

RINK te permite definir un patrón de colores a rotar sobre un conjunto de tintas. Una tinta no es un color. Una tinta es un identificador en el rango [0..15] que identifica a un color del rango [0..26]. Para definir el patrón de colores a rotar, usaremos el comando RINK del siguiente modo:

**RINK, <tinta\_inicial>, <color1>,<color2>, <color3>, ... ,<colorN>**

Esto indica que van a rotar N tintas comenzando por la tinta\_inicial usando el patrón de colores que se indica. Ten en cuenta que, si usas muchas tintas para hacer una animación, te quedarán menos tintas para tus sprites.

Una vez establecido el patrón, podemos rotar las tintas más o menos deprisa con

**RINK, <step>**

El valor del step es el número de desplazamientos que sufrirán las tintas del patrón y por consiguiente un valor superior proporciona un efecto de desplazamiento a mayor velocidad

Recomendación: debido al uso de interrupciones RINK produce “parones” en algunos casos cuando se usa a la vez que el comando |MUSIC a velocidad 6. En caso de querer usar ambos a la vez sin que haya interferencias, usa otra velocidad para la música (puedes usar velocidad 5 o 7, ambas te funcionarán bien).

### 11.7.1 Carreras de coches 2D

El ejemplo del juego de coches lo tienes en el siguiente listado. El sonido del motor intenta causar la sensación de la aceleración. Para los carteles que aparecen a los lados se ha usado un sprite con movimiento relativo, el cual se mueve a la misma velocidad que el step de las franjas. El patrón de tintas se define en la línea 100

|RINK,1,3,3,3,3,24,24,24,24: rem franjas amarillas y rojas

```
1 MEMORY 23999
2 CALL &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 CALL &BC02:'default paleta
50 GOSUB 430
60 INK 0,13
70 INK 14,10
80 linestinta=3
90 rangotintas=8
91 ' establecimiento de patron de tintas
100 |RINK,1,3,3,3,3,24,24,24,24: rem franjas amarillas y rojas
101 |RINK,0
110 y=400
120 ' PAINT ROAD -----
121 tini=1
130 FOR franjas=1 TO 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 NEXT
240 NEXT franjas
250 saltob=-16:xc=65: cosa=0
270 REM ciclo de juego -----
```

```

293 IF saltob=-16 THEN 296
294 IF saltob>0 THEN cosa=-salto ELSE cosa=cosa-1
295 IF cosa<0 THEN |RINK,cosa:vy=3*cosa:posv=posv-3*cosa:|MOVEALL,-vy,0:IF
saltob <=0 THEN cosa=2-saltob
296 |PRINTSPALL
351 ciclo=ciclo+1:IF posv>240 THEN posv=-30:|LOCATESP,30,posv,xc:IF xc=10
THEN xc=65 ELSE xc=10
361 IF INKEY(27)=0 THEN IF x<52 THEN x=x+1:POKE 27499,x:GOTO 370
362 IF INKEY(34)=0 THEN IF x>21 THEN x=x-1:POKE 27499,x
370 IF INKEY(67)=0 THEN IF saltob<16 THEN saltob=saltob+1:salto=saltob/4
380 IF INKEY(69)=0 THEN IF saltob>-16 THEN saltob=saltob-1:salto=saltob/4
390 SOUND 1,6000/(salto+17),1,15
400 GOTO 270
421 REM PALETA
430 INK 0 , 12
440 INK 1 , 5
450 INK 2 , 20
460 INK 3 , 6
470 INK 4 , 26
480 INK 5 , 0
490 INK 6 , 2
500 INK 7 , 8
510 INK 8 , 10
520 INK 9 , 12
530 INK 10 , 6
540 INK 11 , 15
550 INK 12 , 0
560 INK 13 , 23
570 INK 14 , 0
580 INK 15 , 11
590 RETURN

```

### 11.7.2 Scroll de Ladrillos

En este ejemplo vamos a combinar el uso de la animación por tintas con el scroll basado en MAP2SP. Mediante la animación por tintas moveremos un patrón de ladrillos que habremos dibujado con repetición de un sprite mientras que el castillo y el árbol serán movidos por MAP2SP.

El movimiento de los ladrillos implicaría un trabajo inmenso si se realizase por CPU, de modo que esta técnica permite “lo imposible”. Es una técnica muy potente si la empleas con ingenio.



Fig. 79 Scroll usando animación por tintas y MAP2SP a la vez.

El ladrillo empleado es en realidad un sprite de 8 tintas, con el diseño que se muestra a continuación:

								0
8	9	10	11	12	13	14	15	

Y el patrón de tintas es 0,6,3,3,3,3,3,3 . para crearlo simplemente se ejecuta el comando:  
|RINK,8,0,6,3,3,3,3,3,3

El personaje (sprite 31) permanece en el centro de la pantalla, pudiendo saltar y moverse en ambas direcciones. Para sincronizar el movimiento de tintas y el MAP2SP, se establece un step=2 para el comando |RINK, ya que un byte son dos pixels y MAP2SP mueve a nivel de byte.

Es interesante observar como el pájaro (sprite 30) debe ser afectado también por el movimiento ya que a su velocidad hay que sumar o restar la del personaje.

En cuanto al color, puesto que se usa un patrón de 8 colores y además se usa sobreescritura, nos quedan 2 colores para el fondo (castillo, ramas) y 3 colores para los sprites (personaje y pájaro). Es sencillo modificar el programa para usar un patrón de rotación de 4 colores y de ese modo disponer de 5 colores para sprites, más razonable.

A continuación, tienes el listado completo.

```

10 MEMORY 23999
11 ON BREAK GOSUB 5000
20 CALL &6B78
30 FOR i=0 TO 31:|SETUPSP,i,0,0:NEXT:|AUTOALL,1:|PRINTSPALL,0,1,0
40 |SETUPSP,31,7,1:|SETUPSP,31,0,65:|LOCATESP,31,130,36: 'personaje
50 |SETUPSP,30,7,7:|SETUPSP,30,0,157:|LOCATESP,30,50,80:
|SETUPSP,30,15,2: 'bird
60 MODE 0
70 DEFINT a-z
80 CALL &BC02:'default paleta
90 BORDER 10

```

```

100 INK 6,15:INK 7,15
110 INK 4,26:INK 5,26
120 INK 2,0:INK 3,0
130 INK 1,14
140 ' establecimiento de patron de tintas
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 y |MAP2SP"
220 ' PAINT wall -----
230 |SETUPSP,29,9,23:'ladrillo
231 y=152
232 FOR fila=1 TO 6
240 FOR ladrillo=xini to 42 step 2:|PRINTSP,29,y,ladrillo*2:next
241 xini=(xini-1) mod 2: y=y+8
242 next
390 dir=1:x=0:xp=80: ciclo=40: stepy=2
400 |MUSIC,0,7
409 ' ciclo de juego -----
410 |AUTOALL:|PRINTSPALL
450 IF INKEY(27)=0 THEN |RINK, stepy:x=x+1:|MAP2SP,0,x:|MOVER,30,0,-
1:if jump=0 then IF dir=2 THEN dir=1:|SETUPSP,31,7,dir ELSE |ANIMA,31
460 IF INKEY(34)=0 THEN |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=ciclo:|SETUPSP,31,0,205:|SETUPSP,31,15,dir-1:|SETUPSP,31,7,31+dir
472 IF ciclo-jump=20 THEN jump=0:|SETUPSP,31,7,dir:|MOVER,31,5,0:
490 |PEEK,27483,@xp:IF xp<-20 THEN |LOCATESP,30,50,80:'pajaro vuelta a
empezar
501 ciclo=ciclo+1
502 IF xant=x THEN |MAP2SP,0,32000
503 xant=x:' IF quieto THEN no imprimo el castillo asi no parpadea
510 GOTO 410
5000 |musicoff:CALL &BC02:pen 1:MODE 2:END

```

## 12 Juegos de plataformas

La dificultad de programar un juego de plataformas reside fundamentalmente en manejar correctamente la física de los saltos y las colisiones de plataformas. Algo que podrás encontrar en el videojuego: "fresh fruits & vegetables" disponible en 8BP



Fig. 80 Fresh fruits & vegetables

### Saltos:

Básicamente para la física de los saltos en lugar de usar la ecuación de Newton he definido una ruta en la que la Vy comienza en -5 y va disminuyendo fotograma a fotograma. Al llegar a la posición cenital, se cambia la imagen para que se borre a si mismo en su parte superior y la velocidad Vy se torna positiva, y poco a poco va aumentando. En el fondo es como aplicar la ecuación de Newton, pero sin cálculos.

### Comprobación del suelo

Mientras el personaje camina, compruebo en cada fotograma si hay suelo. Como las plataformas pertenecen al mapa del mundo, usan identificadores de sprite bajos (<10) de modo que si con COLSPALL detecto un numero <10 es que hay suelo. Si el resultado de la detección es un 32 (los sprites van de 0 a 31) es que no hay nada y el personaje debe empezar a caer. Los enemigos no detectan nada. simplemente caminan a través de rutas predefinidas, en las que se indica cuantos pasos deben dar en cada dirección y vuelta a empezar. prácticamente su lógica es casi nada pues le asignas una ruta a un sprite y le activas el flag de movimiento automático y de ruta. A partir de ahí el sprite recorre solito la ruta paso a paso al invocar a |AUTOALL (la cual internamente ya invoca a |ROUTEALL).

### Colisiones de plataformas:

cuando el personaje está en ruta de caída, lo muevo (sin imprimirla) hacia abajo 5 unidades y detecto colisión de sprites. A continuación, lo muevo (sin imprimirla) 5 unidades hacia arriba. si la colisión es 32 es que no hay colisión y mantengo al personaje cayendo. Si la colisión es menor que 10 es que ha colisionado con una plataforma. En ese caso muevo el personaje para que encaje perfectamente con el comienzo de la plataforma, así: posición de la cabeza del muñeco es "posy" posición de los pies es posy+26 pues el personaje mide 21 y le sumo 5 pues esta cayendo (hay 5 de autoborrado), de modo que en realidad mide 26. Como las plataformas las hemos ubicado en múltiplos de 8, simplemente me quedo con el resto de la división entera, que lo puedo obtener con un AND 7 en resumen, dos instrucciones muy bien pensadas, pero solo dos:

```
dy =(posy%+26) AND 7  
|MOVER,31,5-dy,0
```

Luego le asigno la secuencia de animación de andar, que no es la de caer y sus fotogramas miden 21 de alto.



## 13 Minicaracteres redefinibles: PRINTAT

El juego de caracteres de Amstrad es bonito y bien construido. Sin embargo, en mode 0 tan solo disponemos de 20 caracteres de ancho por línea y aparecen demasiado “ensanchados”, por lo que en ocasiones no son adecuados para mostrar ciertos textos o marcadores de un juego. Además, el comando PRINT es muy lento, por lo que es poco recomendable actualizar los marcadores frecuentemente, ya que el juego se “detiene” mientras se está imprimiendo, son pocos milisegundos, pero es perceptible.

Por ese motivo, a partir de la versión v31 de 8BP se ha incorporado el comando PRINTAT, el cual puede imprimir una cadena de caracteres usando un nuevo juego de caracteres mas pequeño (los llamo “minicaracteres”). Este nuevo comando permite usar el mecanismo de transparencia de los sprites, de modo que podrás imprimir caracteres respetando el fondo. Funciona del siguiente modo:

```
|PRINTAT, <flag transparencia>, y, x, @string
```

Ejemplo:

```
cad$= "Hola"
```

```
|PRINTAT, 0, 100, 10, @cad$
```



Fig. 81 PRINTAT

El comando |PRINTAT imprime cadenas de caracteres y no variables numéricas, por lo que si quieres imprimir un número (por ejemplo, los puntos en el marcador de tu videojuego) debes hacerlo así:

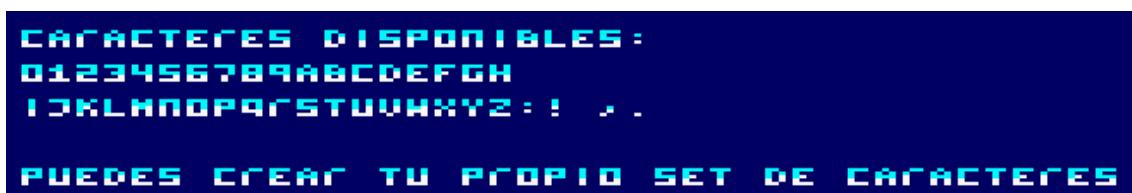
```
puntos=puntos+1  
cad$= str$(puntos)  
|PRINTAT,0,100,10, @cad$
```

El comando |PRINTAT no se ve afectado por los límites para el clipping establecidos con |SETLIMITS. Esto es lo más lógico puesto que normalmente usarás PRINTAT para imprimir puntuaciones en tus marcadores, que se encontrarán fuera del área delimitada por |SETLIMITS.

A diferencia del comando PRINT del BASIC, el comando |PRINTAT es bastante rápido y puede ser utilizado para actualizar los marcadores de tu videojuego con frecuencia.

PRINTAT usa un alfabeto redefinible, que puede contener una versión reducida o diferente de los caracteres “oficiales” del amstrad. Por defecto 8BP proporciona un pequeño alfabeto compuesto por números, letras mayúsculas, el espacio en blanco y algunos símbolos. No podrás usar un carácter que no se encuentre entre este conjunto, como las letras minúsculas. Los caracteres de dicho alfabeto miden todos lo mismo: 4pixels de ancho x 5 pixels de alto, es decir 2 bytes x 5 líneas.

Esta cadena contiene todos los caracteres que puedes usar con el alfabeto de serie de 8BP. Fíjate que no hay minúsculas y faltan muchos símbolos, aunque puedes crear tu propio alfabeto que los contenga. El último símbolo es el “.”  
 "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ:!,."



*Fig. 82 juego de caracteres disponible por defecto en 8BP para usar con PRINTAT*

**IMPORTANTE:** Si tratas de imprimir con PRINTAT un carácter no existente en el alfabeto creado, se imprimirá el ultimo definido en la lista de caracteres, que en el alfabeto por defecto es el punto “.”

He creado los caracteres usando las tintas 2 y 4, para que se pueda usar la sobreescritura, ya que los colores de fondo son 0 y 1 y usando sobreescritura la tinta 2 debe ser igual a la 3 y la 4 debe ser igual a la 5 (consulta el capítulo donde explico la sobreescritura). Para usar la sobreescritura simplemente pon a “1” el flag de transparencia en el comando PRINTAT.

### 13.1 Crea tu propio alfabeto de minicaracteres

Los caracteres que vayas a usar con el comando PRINTAT se definen en un fichero en el directorio ASM y se importan desde el fichero “images.asm”

Veamos un fragmento de “images.asm”

Encontraremos estas tres líneas:

```

; si no vas a usar el comando PRINTAT, sino simplemente los caracteres del amstrad
; entonces puedes comentar las siguientes 3 lineas
_BEGIN_ALPHABET
read "alphabet_default.asm"
_END_ALPHABET
  
```

El alfabeto son unos pocos datos y las imágenes de cada carácter. Todo ello se ensambla dentro de la zona de memoria destinada a imágenes de 8BP. El alfabeto por defecto ocupa poco mas de 400 bytes.

El fichero alphabet\_default.asm contiene el alfabeto que 8BP incluye de serie. Tu mismo puedes crearte tu propio fichero de alfabeto. Este fichero contiene 3 variables que indican el tamaño de los caracteres, los cuales puedes dibujarlos del tamaño que quieras. Por defecto yo he creado un alfabeto de 2 bytes de ancho por 5 lineas de alto, pero tu puedes decidir usar otro tamaño, incluso crear caracteres gigantes. Si cambias el ancho o el alto, también deberás cambiar consecuentemente la variable ALPHA\_TAMANO en tu fichero alphabet.asm.

```

ALPHA_ancho db 2; ancho alfabeto.Todas las letras miden lo mismo
ALPHA_alto db 5; alto alfabeto.todas las letras miden lo mismo
ALPHA_tamano db 2*5
  
```

A continuación, encontramos la cadena que indica los caracteres válidos para usar con el comando PRINTAT. Estos caracteres son válidos porque tienen un dibujo asociado. Tras esta cadena, se encuentran uno por uno los dibujos de todos esos caracteres, en el mismo orden en el que figuran en la cadena de texto ALPHA\_LIST

```
ALPHA_LIST
"0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ:!.," ; caracteres creados
db 0 ;byte cero indica final de cadena de caracteres de la Alpha_list
```

Los dibujos de cada carácter de la cadena ALPHA\_LIST vienen a continuación. Yo los he creado con la herramienta SPEDIT. El primero de ellos debe ser el primero de la cadena ALPHA\_LIST, es decir, el “0”.

Aquí se muestran los bytes correspondientes a este carácter:

<pre>; caracter 0 db 12 , 8 db 8 , 8 db 8 , 8 db 32 , 32 db 48 , 32</pre>	
---	--

Seguidamente encontraremos una por una el resto de las letras, números y símbolos definidos. Con algo de paciencia puedes crearte tu propio juego de minicaracteres, lo cual le dará más personalidad a tu videojuego. Sólo necesitas crear los que vayas a usar y cuantos menos sean, menos memoria gastarás de la zona de imágenes.

Recuerda que, si tratas de imprimir un carácter que no has creado, se imprimirá el último de los caracteres definidos en la ALPHA\_LIST



## 14 Pseudo 3D

En los años 80 fueron populares los juegos de coches estilo “Pole position”. Daban sensación 3D, pero no realizaban los cálculos 3D, tan solo para el plano de la carretera, y en ocasiones ni eso, pues eran aproximaciones que daban una buena sensación de velocidad.

En las máquinas recreativas se usaban chips específicos para realizar “sprite scaling” haciendo que los sprites aumentasen de tamaño suavemente, y el calculo de la carretera mediante un chip específico (como el “sega Road chip”) que se dedicaba exclusivamente a pintar la carretera con sus franjas. El plano de la carretera después se sumaba al plano de sprites y se componía la imagen final. Estos chips se usaban en máquinas arcade tales como “Pole Position” y “Space Harrier”.



Fig. 83 Pole position y Out Run (máquinas arcade)

La característica común a las técnicas software (usadas en ordenadores de 8bit) y hardware (usadas en maquinas recreativas) es que las curvas son una ilusión: se deforma la carretera al tiempo que se mueven unas montañas en el horizonte para dar la sensación de giro, pero no hay ningún giro. Los resultados eran en no pocas ocasiones muy convincentes pero las curvas eran realmente una ilusión.

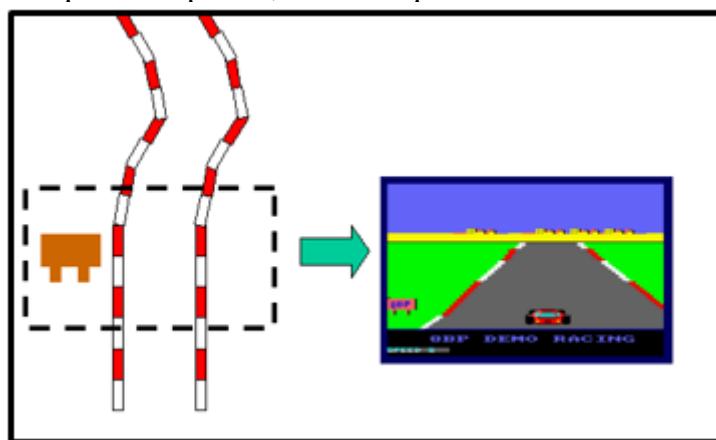
Las técnicas utilizadas en ordenadores de 8bits eran muy variopintas. Cualquier cosa era aceptable con tal de hacerte creer al jugador que se encontraba en un circuito de carreras. En muchas ocasiones se usaba rotación de tintas para dar sensación de velocidad. Muchos juegos sufrían de un bajo frame rate, por debajo de 5 fps. Entre los mejores juegos para amstrad se encuentran el “3D grand Prix” (que usa sprite scaling por software combinado con animación por tintas), el “Buggy boy” basado en una técnica de programación muy avanzada de efecto raster y algunos otros como el crazy cars o el Chase HQ.

Desde la versión V32 de 8BP tienes a tu disposición capacidad Pseudo-3D, utilizada en el juego de demostración “3D Racing one”. Es muy sencillo usarla y con ella podrás construir tus juegos de carreras, de tanques, de naves, etc

Las capacidades que aporta 8BP para poder disponer de pseudo-3D son:

- 1) **Proyección 3D:** consiste en proyectar un mapa del mundo 2D en 3D, de modo que, aunque lo recorramos en 2D, nos da la sensación de verlo en 3D. Esto lo haremos invocando al comando |3D para configurar los comandos PRINTSPALL y PRINTSP de modo que proyecten en 3D antes de pintar en pantalla. No habrá posibilidad de girar en el plano, siempre “miraremos” hacia

delante, pero el efecto combinado de una curva unido a unas casas o montañas en el horizonte que se desplacen, simulará que estamos tomando una curva.



*Fig. 84 Proyección 3D*

- 2) **Zoom:** consiste en poder usar distintas versiones de una misma imagen de un objeto para dar efecto de zoom a medida que nos acercamos a él. Esto simplemente lo haremos en el fichero de imágenes, definiendo 3 versiones de una misma imagen y agrupándolas en una estructura. En la imagen se muestra el típico ejemplo del cartel que se acerca. Los comandos PRINTSPALL y PRINTSP escogerán la versión más adecuada de la imagen según a la distancia a la que se encuentre, sin necesidad de hacer nada más que definir la imagen como una imagen de tipo “Zoom”.



*Fig. 85 Zoom images*

- 3) **Segmentos:** consiste en poder disponer de sprites de tipo “segmento”, definidos por una sola scanline horizontal (por lo que ocupan muy poco), a los que podamos asociar una longitud y una inclinación. Con ellos podremos construir caminos, ríos, etc y bordes de carreteras. Esto simplemente lo haremos en el fichero de imágenes definiendo un tipo de imagen que además de ancho y alto posee inclinación. Estos segmentos los utilizaremos en la construcción del mapa del mundo.



*Fig. 86 segmentos con distinta inclinacion*

Los segmentos utilizados en el juego “3D Racing one” son rojos o blancos y aunque sean largos, solo estan definidos por una scanline. Por ejemplo, el segmento blanco izquierdo consiste en unos pocos bytes verdes para el cesped, un par de blancos y otros pocos grises para la carretera. En el momento de imprimirse, el segmento asi definido se replica hasta medir la longitud deseada, y se imprime en perspectiva, por lo que, aunque sea un segmento recto, se mostrará torcido.

Por último, mencionar que en muchas ocasiones será necesaria la actualización dinámica del mapa (comando |UMAP). Gracias a este comando, el mapa puede ser muy grande (lo cual es necesario si creamos un circuito con muchísimos segmentos). Este comando se describe en el capítulo de scroll.

A continuación, vamos a ver en profundidad estas 3 características y como usarlas en tus programas.

## 14.1 Proyección 3D

Para proyectar disponemos del comando |3D

Uso

```
|3D, 1, <Sprite_fin>, <offsety> : REM esto activa la proyección 3D
|3D, 0 : REM desactiva la proyección 3D
```

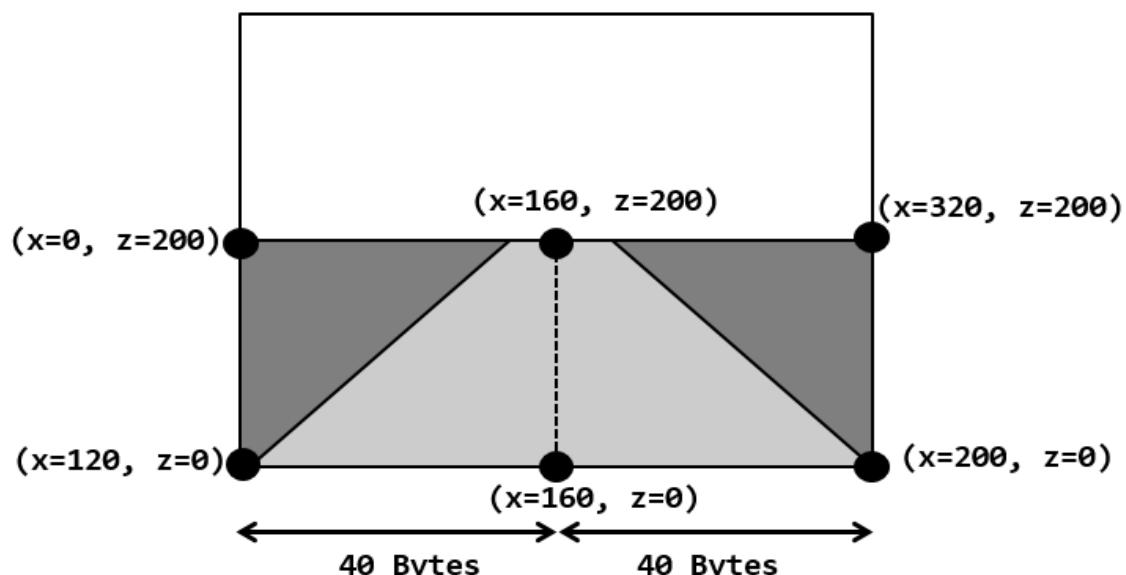
Este comando activa la proyección 3D en el commando PRINTSP y en PRINTSPALL. Esto significa que antes de imprimirse en pantalla se calcularán las coordenadas “proyectadas” y a continuación se imprimirán en pantalla. Las coordenadas de los sprites no se ven afectadas, es decir, las coordenadas seguirán siendo las mismas, pero en el mundo 2D. En pantalla ahora tenemos una vista 3D y las coordenadas en las que se imprimirán serán el resultado de ejecutar la función de proyección sobre sus coordenadas 2D.

Los sprites afectados son todos desde el Sprite 0 hasta el <Sprite\_fin>. El resto de sprites no se proyectarán, simplemente se imprimirán en pantalla en sus coordenadas 2D.

Este comando **no afecta a los mecanismos de colisión**, es decir, si usamos COLSPALL y detectamos una colisión entre sprites proyectados, la colisión se estará produciendo en el plano 2D. Puede que en algún caso se solapen parcialmente dos sprites proyectados en pantalla, pero eso no significa que hayan colisionado, puede que uno esté ligeramente mas adelante que el otro y al proyectarse se solapen, pero eso no es una colisión real. Las colisiones se detectan en el plano 2D.

En cuanto al último parámetro `<offsety>` es para proyectar mas arriba o mas abajo, de modo que podamos ubicar los marcadores del juego donde queramos. Al proyectar la pantalla, que mide 200 de alto, se transforma en 100 pixels de alto, de modo que podemos escoger a que altura ubicamos la proyección. Si un Sprite no es afectado por la proyección por ser superior a `<Sprite_fin>`, entonces tampoco le afecta el `<offsety>`. Este es el caso, por ejemplo, de las nubes y las casas en el horizonte en el juego “3D Racing one”

Para entender las coordenadas de pantalla en la que finalmente aparecen tus sprites proyectados, te recomiendo que leas el siguiente apartado de matemáticas, muy sencillo, con el que lo entenderás por completo. La siguiente figura representa cuales son las coordenadas del mapa del mundo que se proyectan en ciertos puntos representativos de la pantalla cuando MAP2SP es invocado con (`yo=0, xo=0`). La coordenada Z representa la distancia a la que se encuentran los objetos, también llamada “profundidad”.



*Fig. 87 coordenadas del mundo proyectadas*

Si en lugar de (`xo=0, yo=0`) usamos otra coordenada para MAP2SP, las coordenadas del mundo 2D correspondientes a los puntos referenciados en la imagen estarán desplazadas en ( $x, z$ ) lo que se indique con ( $xo, yo$ ).

#### 14.1.1 Matemáticas de la proyección pseudo 3D

La proyección pseudo 3D consiste en proyectar sobre la pantalla el plano del suelo, que es donde está nuestro mapa 2D del mundo.

## Cálculo de la coordenada Y

Nuestro suelo puede ser infinito, pero proyectaremos solo 200 pixels de largo. A dicho largo se le llama “profundidad” o coordenada “Z”. Dichos 200 pixels de profundidad, al proyectarse se comprimen en 100 líneas de alto (coordenada Y proyectada). Los píxeles más lejanos proyectados constituyen la línea de horizonte. Ojo, no vamos a ver objetos lejanísimos en el horizonte, tan solo los que se encuentren como mucho a 200 de profundidad.

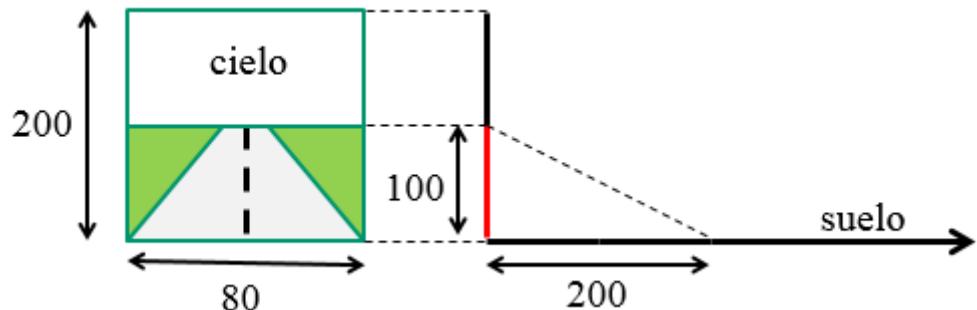


Fig. 88 proyección del suelo en la pantalla

A medida que los objetos se alejan se ven mas y mas pequeños. No es una relación lineal la existente entre la pantalla y el suelo, es decir, para saber a que altura se debe imprimir un pixel que esta a cierta distancia, es incorrecto pensar que como la profundidad abarcada es 200 y se proyecta sobre 100 lineas de alto, basta dividir entre dos. En una función lineal (tal como  $y = f(z) = A \cdot z + B$ ) la derivada ( $A$ ) es constante, pero en la proyección, lo que es constante es la “segunda derivada” de la función  $f(z)$ . Ahora veremos esto en detalle.

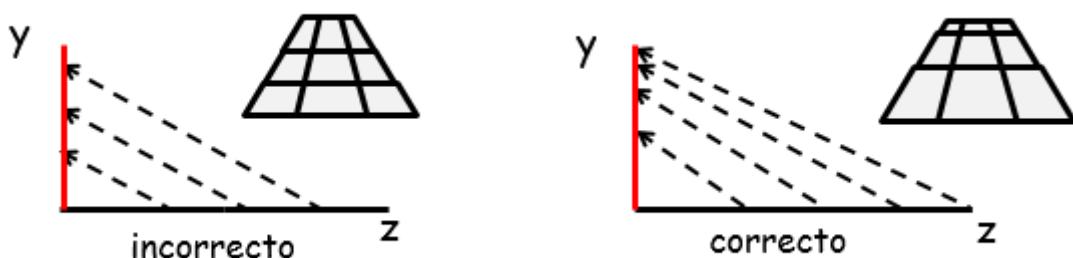


Fig. 89 La proyección correcta no es lineal

Supongamos que empezamos con “ $Z=0$ ” y queremos saber cuánto vale “ $Y$ ”. Es muy fácil, en la línea de suelo ( $z=0$ ) la coordenada “ $y$ ” también es cero.

Si incrementamos  $z$  con una pequeña variación “ $dz$ ”, la “ $y$ ” valdrá la “ $y$ ” anterior (cero) más un incremento  $dy$ , que inicialmente será igual al incremento  $dz$  pues el incremento ha sido pequeño y estamos cerca del horizonte.

$$dy (\text{ inicial}) = dz$$

Ahora bien, si de nuevo nos alejamos  $dz$ , el incremento  $dy$  debe de disminuir, y si volvemos a alejarnos  $dz$ , el  $dy$  que debemos sumar cada vez será más pequeño. Es decir:

Cada vez que nos alejamos  $dz$ , sumamos un **incremento** a “y” que cada vez es menor  
 $z=z+dz$   
 $dy = dy + ddy$  , siendo  $ddy$  negativo  
 $y= y + dy$

El incremento que sufre  $dy$  es constante. Ese incremento lo hemos llamado  $ddy$ . Pues bien,  $dy$  es la “derivada” de  $y$ , mientras que “ $ddy$ ” es lo que se llama “segunda derivada”. Aquí la derivada no es constante, pero la segunda derivada sí lo es.

El valor constante que asignemos a “ $ddy$ ” producirá proyecciones mas o menos exageradas. En 8BP el valor  $ddy$  es negativo, de aproximadamente -0.005, haciendo que en la línea de suelo el  $dy$  sea prácticamente 1, mientras que, en la línea de horizonte, la acumulación de 200  $ddy$  hace que el valor de  $dy$  acabe en cero.

A pesar de la simplicidad de estos cálculos, hacerlos en tiempo real es costoso para nuestro amado Amstrad CPC, por lo que en realidad 8BP realiza una aproximación para evitar cálculos y traducir “ $z$ ” a “ $y$ ” de un modo mucho más simple y con un resultado muy similar.

Si  $0 \leq z < 50$  , entonces  $dy = dz$ , por lo tanto  $y = z$

Si  $50 \leq z < 110$  , entonces  $dy = dz/2$  , por lo tanto  $y = 50 + (z-50)/2$

Si  $110 \leq z \leq 200$  , entonces  $dy = dz/4$ , por lo tanto  $y = 50 + (110-50)/2 + (z-110)/2$

Es decir, hemos dividido la pantalla en tres franjas y cada franja se trata como una zona “lineal” (pues “ $dy$ ” es constante) pero con diferente valor de “ $dy$ ” respecto las otras franjas. Esta aproximación da resultados visualmente muy similares a los matemáticamente correctos.

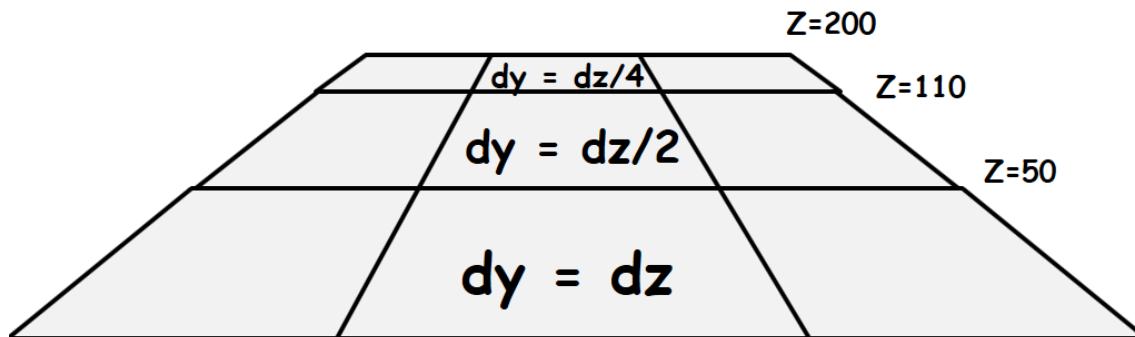


Fig. 90 Aproximación de 8BP

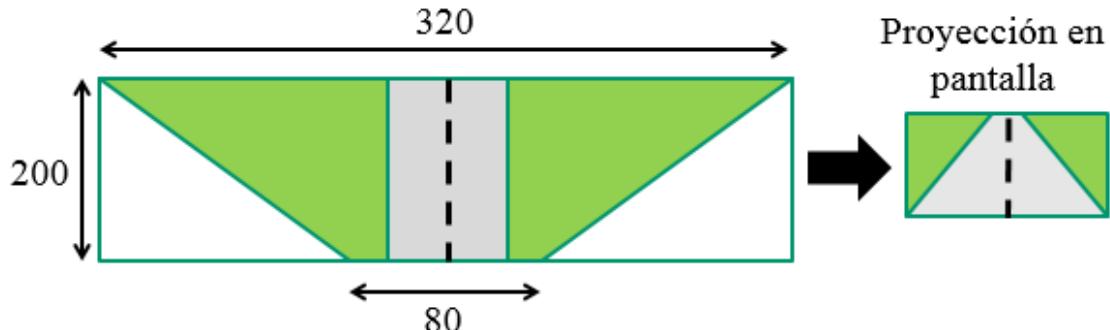
Las ecuaciones usadas en 8BP tienen además en cuenta que las coordenadas de pantalla en realidad están “invertidas”, es decir la coordenada cero es la superior y la 200 es la inferior, pero eso es solo un ajuste final muy sencillo.

#### Ahora pasemos al cálculo de la coordenada “X”:

Hay algo fundamental que diferencia la línea de horizonte de la línea de suelo. La línea de suelo mide 80 Bytes, pero la de horizonte representa más anchura, pues la carretera es recta y lo que mide 80 en el suelo, mide una fracción en el horizonte, concretamente en 8BP mide 4 veces menos. La carretera se estrecha en el horizonte, porque el horizonte representa mucho más. En la proyección que aplica 8BP representa 320 bytes. Y si el horizonte mide 320 y el suelo 80, es porque el área real total que somos capaces de abarcar en pantalla una vez efectuada la proyección es un área trapezoidal. NO es

que el suelo sea un trapecio, sino que el área de suelo que somos capaces de ver en pantalla tiene forma de trapecio

### Área trapezoidal situada en el suelo



*Fig. 91 Área trapezoidal visualizada*

Intuitivamente y sin mostrar ninguna ecuación, ya está claro lo que queremos hacer y cómo funciona la proyección. Ahora veamos las ecuaciones.

En esencia las matemáticas de la coordenada “X” son las mismas que las de la coordenada “Y”. Sin embargo, no se hacen del mismo modo, porque una vez que tenemos la coordenada “Y” calculada, podemos hacer una ecuación lineal  $x=f(y)$  ya que debido a la no linealidad de “Y” respecto “Z”, es como crear una relación no lineal entre “X” y “Z”.

Anteriormente hemos dicho que el horizonte mide 320 Bytes y el suelo mide 80 Bytes. Esto significa que el suelo mide 4 veces menos que el horizonte. En la lejanía debemos dividir entre 4 (dividir es costoso así que preferiremos multiplicar por un factor 0.25) mientras que en la cercanía deberemos multiplicar por un factor = 1.

Lo que tenemos que hacer es centrar la coordenada X del objeto a proyectar respecto el centro de la pantalla. A continuación, multiplicaremos por un factor que dependerá de la coordenada “Y” proyectada. Esto establecerá una relación no lineal entre “X” y “Z”

Debemos construir una ecuación que retorne un resultado 4 veces menor en el horizonte, por ejemplo:

```

Factor = ((100-y)+ 32 ) / 2
x= (x - centro) * Factor + centro
si z=200 entonces y= 100, y entonces factor =16
si z=0 entonces y= 0, y entonces entonces factor =64 ( 4 veces mas)

```

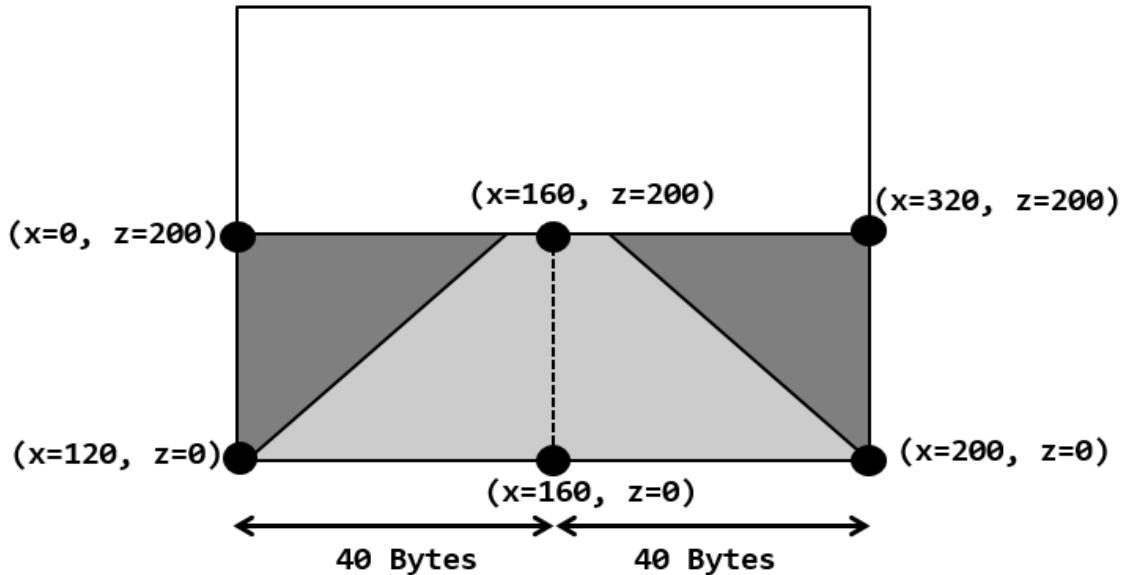
La ecuación escogida es interesante porque la división entre 2 es algo que el amstrad puede hacer rotando en binario un bit. Es decir, puede hacerla en pocos ciclos de reloj.

Como se ha mostrado en la ecuación, para proyectar la coordenada “X”, basta con centrarla y a continuación multiplicarla por el factor obtenido. El numero obtenido así es muy grande, pues en el caso de la línea de horizonte estaremos multiplicado por 16 y en la línea de suelo se ha multiplicado por 64, es decir hemos multiplicado la coordenada x por un factor cuyo valor se encuentra entre 16 y 64. Entre el horizonte y suelo tendremos todos los 48 valores posibles para el factor, por lo que, aunque el factor sea un numero entero, evolucionará suavemente.

A continuación, dividimos entre 64, que no es mas que rotar en binario 6 veces, y listo. Esto último será equivalente a haber multiplicado por 0.25 el horizonte, por 1 el suelo y

por el valor decimal que corresponda a cualquier altura intermedia entre el suelo y el horizonte, pero lo hemos hecho con números enteros, que el Amstrad puede manejar rápido. A esta técnica se la llama “aritmética de coma fija”.

Teniendo todo esto en cuenta, y suponiendo que le pedimos a MAP2SP que nos genere los sprites del mapa del mundo a partir de la coordenada ( $yo=0$ ,  $xo=0$ ) lo que estaremos viendo en la pantalla tendrá las siguientes coordenadas del mundo. Seguro que ahora te resulta sencillo entender la figura.



*Fig. 92 coordenadas del mundo proyectadas*

Como podrás deducir el punto ( $x=0, y=0$ ) del mapa del mundo queda proyectado fuera de la pantalla, no se visualiza. Si en lugar de ( $xo=0, yo=0$ ) usamos otra coordenada para MAP2SP, las coordenadas del mundo 2D correspondientes a los puntos referenciados en la imagen, estarán desplazadas en ( $x, z$ ) lo que se indique con ( $xo, yo$ ).

### 14.1.2 Curvas

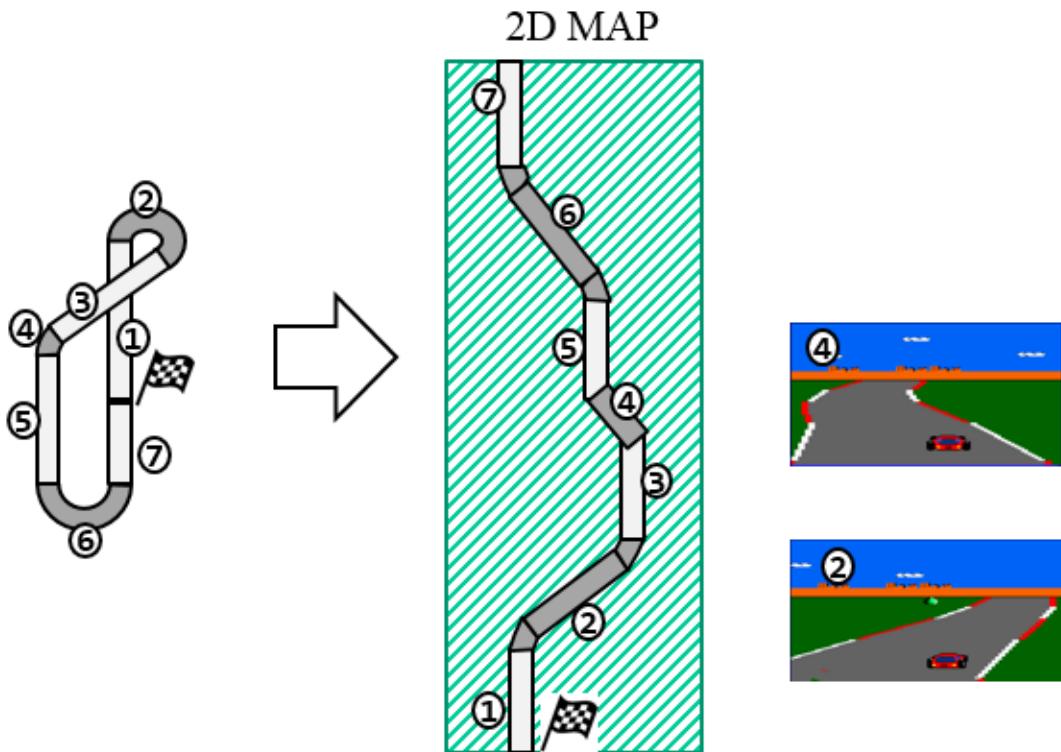
Como he mencionado al principio de este capítulo, la proyección 3D que usa 8BP no permite rotaciones del plano, por lo que para simular una curva tendremos que hacer un pequeño truco. Se trata de torcer la carretera a derecha o izquierda, al tiempo que movemos sprites tales como casas o montañas en el horizonte. Estos sprites no serán proyectados y para evitarlo usaremos identificadores por encima de `<Sprite_fin>`. Recuerda que para activar la proyección 3D haremos:

```
|3D, 1, <Sprite_fin>, <offsety>
```

Por consiguiente, un aparente circuito con curvas quedará representado en nuestro mapa 2D como un camino con inclinaciones a derecha e izquierda, simplemente.

Con esta estrategia, podemos crear la sensación de que un piloto de carreras recorre un circuito con verdaderas curvas, y dar la sensación de que su coche gira en las curvas mediante unas casas o montañas en el horizonte que se mueven en dirección contraria al avance en la coordenada X. Si eres un buen artista y tuerces en mayor o menor grado paulatinamente diferentes segmentos, darás la sensación de curvas muy realistas

Esta estrategia es computacionalmente muy eficiente y suficientemente realista. Si quisiésemos rotar el plano del suelo de verdad, tendríamos que aplicar cálculo matricial para rotar, con muchas operaciones muy costosas y, además, las texturas de los sprites ubicados en el suelo deberían también rotar, de modo que el costo computacional sería enorme.



*Fig. 93 circuito imaginario y mapa del mundo 2D*

Hoy en día los ordenadores son tan potentes que las estrategias aquí presentadas carecen de sentido, pero son estrategias superiores en elegancia e ingenio a la “fuerza bruta” actual. Recuerda que “las limitaciones no son un problema, sino una fuente de inspiración”.

Te deberás servir del comando |UMAP para recorrer mapas de circuitos grandes, pero recuerda invocar a |UMAP no en cada ciclo sino tan solo de vez en cuando.

## 14.2 Zoom images

Para definir una “zoom image”, simplemente crearemos las distintas versiones de la imagen (3 versiones). Después, en el fichero de imágenes “images\_mygame.asm” buscaremos una etiqueta llamada “\_3D\_ZOOM\_IMAGES”.

Encontraremos esto:

```
_BEGIN_3D_ZOOM_IMAGES
;=====
; limites aplicables a todas las imágenes con zoom
; para estos límites se considera el horizonte como el 0 y hacia abajo
; va creciendo hasta 200
_ZOOM_LIMIT_A
db 120; entre 200 (suelo) y limitA se pone imagen 3
_ZOOM_LIMIT_B
```

```

db 50
; entre este limite y el limite A, se pone imagen 2
; mas cerca del horizonte que limit B se pone imagen 1
;=====
CARTEL_ZOOM
db 1; ancho simbolico
db 1; alto simbolico
dw CARTEL1, CARTEL2, CARTEL3

_END_3D_ZOOM_IMAGES

```

Todas las imágenes de tipo ZOOM se deben definir después de la etiqueta “\_BEGIN\_3D\_ZOOM\_IMAGES”. Son imágenes que tienen un ancho y un alto simbólico, pues en realidad un Sprite al que se asigne una de estas imágenes, usará el ancho y alto de la versión de la imagen que se escoja automáticamente en función de su coordenada Y. Es decir, “CARTEL1” es una imagen normal, definida previamente, con su ancho, su alto y sus bytes que contienen el dibujo. Y lo mismo se puede decir de “CARTEL2” y “CARTEL3”. Si asociamos la imagen “CARTEL\_ZOOM” a un Sprite (esto lo podemos hacer asociando un identificador a dicha imagen al principio del fichero de imágenes) lo que ocurrirá es que en función de la posición del Sprite en pantalla se imprimirá una u otra versión de la imagen.

Para la elección automática de la imagen se establecen unos umbrales de coordenada “y”. Dichos umbrales los puedes cambiar, aunque los que hay por defecto funcionan bien y son:

- entre el horizonte =0 y 50, se escoge la primera imagen (en el ejemplo, “CARTEL1”)
- entre el 50 y 120, se escoge la segunda imagen (en el ejemplo, “CARTEL2”)
- entre el 120 y 200, se escoge la tercera imagen (en el ejemplo, “CARTEL3”)

La elección de estos límites para delimitar las franjas de la pantalla es configurable y puedes poner los que quieras. Ten en cuenta que la elección se realiza en base a la coordenada Y del Sprite sin proyectar. Una vez proyectado, su coordenada Y varía mucho, pero no es la “Y” proyectada la que se utiliza para delimitar las 3 franjas, sino la “Y” del Sprite en 2D. Cuando el Sprite pase de una franja a otra, su imagen cambiará automáticamente. Si te gusta más que aparezca antes alguna imagen puedes modificar los umbrales.

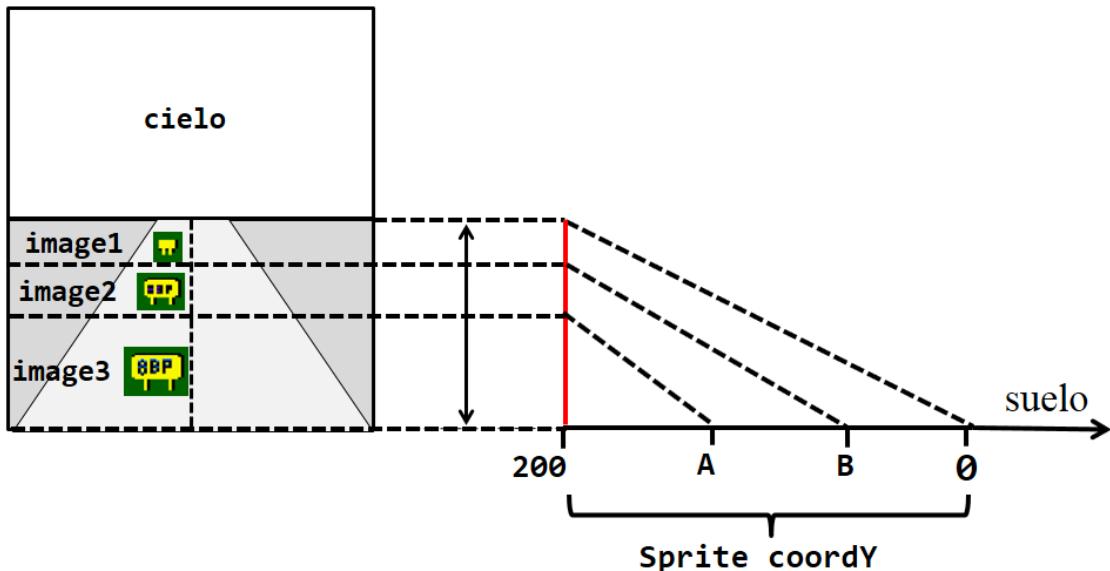


Fig. 94 franjas de uso de las 3 versiones de la ZOOM image

Por ejemplo:

```

REM supongamos que CARTEL_ZOOM tiene id=16 en el fichero de imágenes
|SETUPSP,20,9,16 : REM asocia CARTEL_ZOOM al Sprite 20
|LOCATESP, 20,100, 160: REM Sprite en el centro del "trapecio"
(coordy=100)
|3D,1,31,0: REM se proyectaran todos los sprites
|PRINTSP,20: REM se imprime la versión 2 del CARTEL

```

### 14.3 Uso de segmentos

Ahora que sabes cómo construir un mapa 2D que “simule” ser un circuito con curvas, te presento una forma potente de construir los segmentos con diferente grado de inclinación que necesitas para construir circuitos de carreras o caminos

Un segmento es una imagen de una sola línea de alto, que por repetición puede alcanzar la longitud que queramos. Además de la longitud, tiene otro parámetro, que es la inclinación total del segmento, en bytes. Dicha inclinación puede ser negativa (inclinación hacia la derecha) o positiva (inclinación hacia la izquierda).

Para definir este tipo de imágenes debes crearlas en el fichero de imágenes de tu juego “images\_mygame.asm”, después de la etiqueta “\_BEGIN\_3D\_SEGMENTS”

```

;=====
;_BEGIN_3D_SEGMENTS
;=====
;podria haber una definicion diferente de segmentos
; el ancho es el de la scanline
; el alto es el alto 2D del segmento
; luego va el dx, que puede ser positivo ( inclinado a izquierda) o
negativo (inclinado a derecha)
db 0; esto es para que la primera imagen de tipo segmento sea >
_3D_SEGMENTS

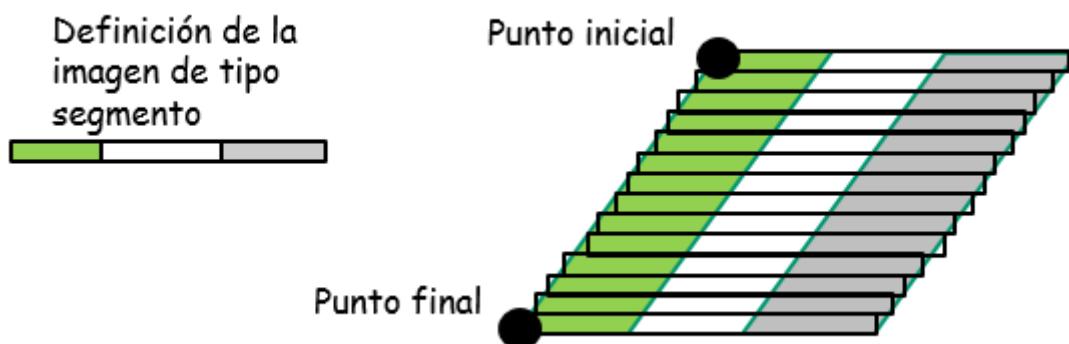
```

```

;----- SEGMENT_EDGE_LWI10 -----
SEGMENT_EDGE_LWI10
db 22; ancho
db 50; alto
db 10; dx
db 192,192,192,192,192,192,192,192,192,192, 240,
,0,0,0,0,0,0,0,0,0,0
;-----
```

En el ejemplo hemos definido un segmento que tiene césped a la izquierda (bytes verdes), luego dos bytes blancos y después bytes grises (carretera) a la derecha. Lo de verdes o grises depende de los colores que hayamos asociado a las tintas. Es un segmento con una inclinación a la izquierda de 10 bytes. Si en la inclinación (dx) hubiésemos puesto un cero, sería un segmento recto, no torcido. Mide 50 líneas de alto, pero cuando se proyecta mide menos, salvo que se encuentre muy cerca.

Los puntos del segmento que se proyectan son sólo dos. Una vez proyectados, se pintan las scanlines una por una con un cierto desplazamiento horizontal para que la última línea acabe empezando en el punto final. Observa que, aunque el segmento se pinte en perspectiva, su ancho es constante. No se pinta más estrecha la parte superior (más lejana) y más ancha la inferior (más cercana), sino que se va a pintar cada scanline exactamente tal como ha sido definida en el fichero de imágenes.



*Fig. 95 en un segmento se proyectan 2 puntos*

Como ves he usado muchos bytes de césped y de carretera. Es para que el segmento se “borre a sí mismo” al avanzar, aunque si el coche va muy deprisa podrán quedar rastros. Una vez que pongas tu juego a funcionar comprobarás si la velocidad es excesiva y quedan rastros.



*Fig. 96 Relación del grado de inclinación de un segmento y la velocidad máxima*

Como se aprecia en la figura, la velocidad de avance máxima para un segmento que se borra a sí mismo puede ser mayor si el grado de inclinación es moderado. Si está muy torcido la velocidad no puede ser tan grande, o quedarán rastros. Una vez que el segmento se proyecta, quedará más corto por efecto de la perspectiva y en función de

donde se encuentre puede quedar aún más torcido o menos. Conviene hacerlos anchos para que se borren a sí mismos con más seguridad.

En el juego “3D Racing one” hay una fase llamada “superfast” en la cual, usando segmentos menos torcidos, aumenté la velocidad del coche sin que por ello los segmentos dejasen rastros. Un sencillo “truco” muy efectivo. Si el juego es lento (por ejemplo, un juego de tanques, que se supone que van despacio) entonces los segmentos pueden estar muy torcidos pues no dejarán rastro por efecto de la velocidad.

En resumen, para aumentar la velocidad tienes dos opciones:

- Usar segmentos menos torcidos
- Usar segmentos más anchos (con más margen de borrado de sí mismos)



## 15 Música

La herramienta de las que voy a hablar en este apartado no la he programado yo, pero están integradas en 8BP y son realmente buenas.

### 15.1 Editar música con WyZ tracker

Esta herramienta es un secuenciador de música para el chip de sonido AY3-8912. Las músicas que genera se pueden exportar y dan como resultado dos archivos

- Un archivo de instrumentos “.mus.asm”
- Un archivo de notas musicales “.mus”

Puedes componer canciones con esta herramienta y la única limitación que tendrás es que todas las canciones que integres en tu juego deberán compartir el mismo fichero de instrumentos.

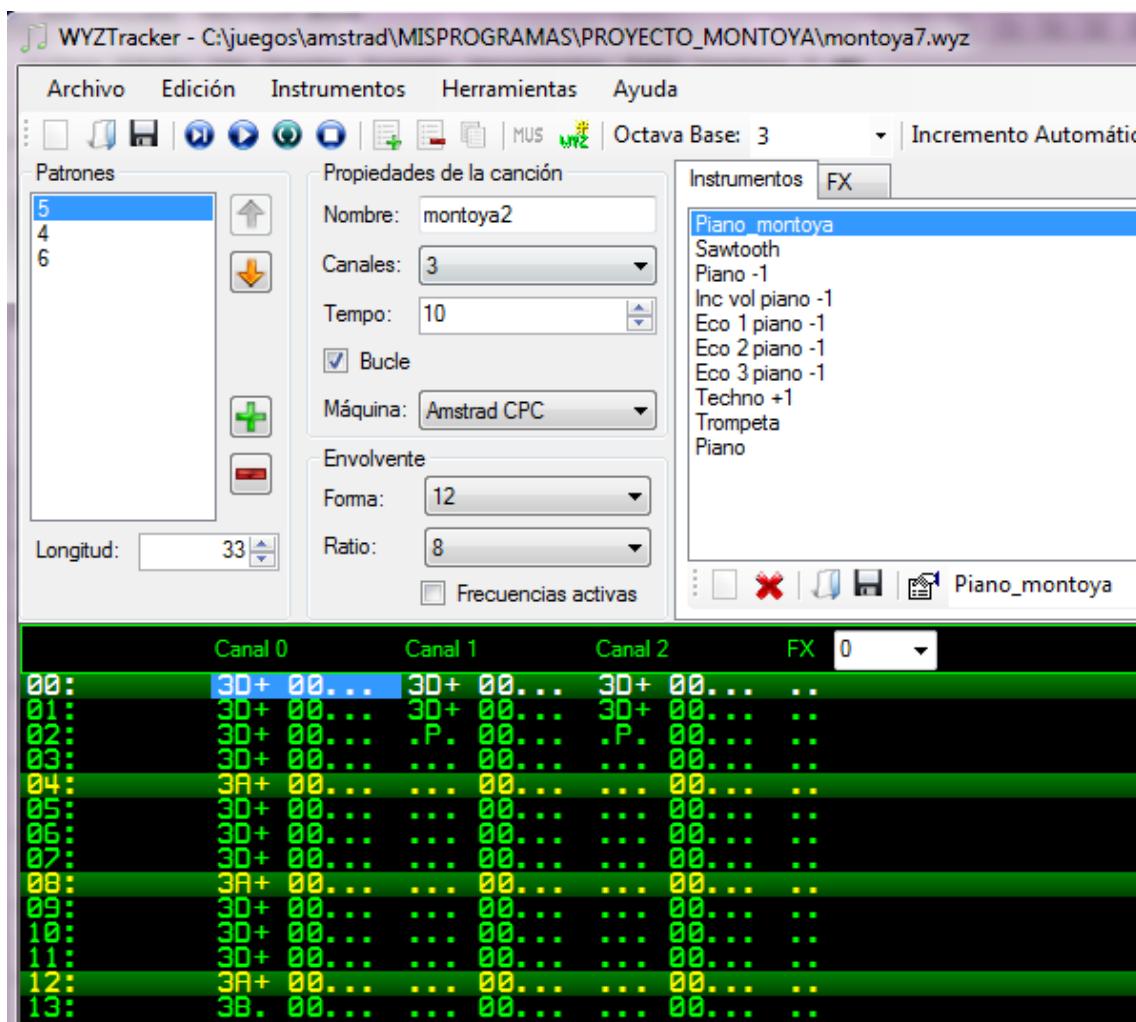


Fig. 97 WYZTracker

Este secuenciador de música se complementa con el WYZplayer que está integrado en la librería 8BP.

Desde la versión V26 de 8BP, las músicas pueden ser compuestas con la versión 2.0.1.0 de WYZtracker y funciona realmente bien. Hasta la versión V25 de 8BP la compatibilidad era con WYZtracker 0.5.07 y había algunos pequeños problemas, pero todo eso ha desaparecido con WYZtracker 2.0.1.0

Una recomendación importante a la hora de crear tus canciones con el WyZtracker es eliminar los instrumentos que no vayas a usar. De este modo el fichero de instrumentos (que acaba en “.mus.asm”) ocupará menos y puesto que 8BP sólo reserva 1.400 bytes para la música cada byte es importante

## 15.2 *Ensamblar las canciones*

Una vez que has compuesto tu canción y ya tienes los dos archivos, simplemente editas el fichero make\_music.asm e incluyes tus ficheros de música así:

```
; tras ensamblarlo, salvalo con save "musica.bin",b,32200,1400

org 32200
-----MUSICA-----
; tiene la limitacion de tan solo poder incluir un solo fichero de
; instrumentos para todas las canciones
; la limitacion se solventa simplemente metiendo todos los
; instrumentos en un solo fichero.

;archivo de instrumentos. OJO TIENE QUE SER SOLO UNO
read "instrumentos.mus.asm"

; archivos de musica

SONG_0:
INCBIN "micancion.mus" ;
SONG_0_END:

SONG_1:
INCBIN "otra_cancion.mus" ;
SONG_1_END:

SONG_2:
INCBIN "tercera_cancion.mus" ;
SONG_2_END:
SONG_3:
SONG_4:
SONG_5:
SONG_6:
SONG_7:
```

Por último re-ensamblas la librería 8BP para que el player de música (que está integrado en la librería) conozca los parámetros de instrumentos y el lugar donde han quedado ensambladas las canciones.

Para ello simplemente ensamblas el fichero make\_all.asm, que tiene este aspecto

```

; 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
;-----CODIGO-----
;incluye la libreria 8bp y el playerWYZ de musica
read "make_codigo.asm"

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

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

```

Y con esto ya tienes todo ensamblado. Ahora debes generar tu librería 8BP así:

**SAVE “8BP.LIB”, b, 24000, 8200**

Y las músicas:

**SAVE “music.bin”, b, 32200, 1400**

### 15.3 Qué hacer si no te cabe la música en 1400 bytes

Es posible que 1.400 bytes no sean suficientes para tus canciones. En caso de que una canción no te quepa (y lo sabrás chequeando donde se ensambla la etiqueta “\_END\_MUSIC”) puedes especificar una dirección diferente de ensamblado para esa canción y las siguientes. En el caso del videojuego “Nibiru”, se hace esto con la tercera canción, ensamblándola en una dirección inferior a la de la librería 8BP (por ejemplo, la 23000 podría servir). En caso de hacer esto, tu juego BASIC deberá comenzar con un comando MEMORY que especifique una dirección inferior a este nuevo límite, por ejemplo, MEMORY 22999 serviría.

A partir de la versión 34 de 8BP, la librería se ensambla a partir de la dirección 24000 por lo que, si quieres usar espacio extra para la música, deberá ocupar direcciones inferiores a la 24000. Por ejemplo, desde la 23500 hasta la 23999.

Este es el

```

; tras ensamblarlo, salvalo con save "musica.bin",b,32200,1400
org 32200
;-----MUSICA-----
; tiene la limitacion de tan solo poder incluir un solo fichero de ;
instrumentos para
; todas las canciones. la limitacion se solventa simplemente metiendo
todos los
; instrumentos en un solo fichero.

;archivo de instrumentos. OJO TIENE QUE SER SOLO UNO

```

```

read  "../MUSIC/nibiru5.mus.asm" ;

; archivos de musica
SONG_0:
INCBIN      "../MUSIC/attack5.mus" ;
SONG_0_END:

SONG_1:
INCBIN      "../MUSIC/nibiru5.mus" ;
SONG_1_END:

org 23500 ; esta linea la uso porque no me cabe la tercera cancion !!

SONG_2:
INCBIN      "../MUSIC/gorgo3.mus" ;

SONG_3:
SONG_4:
SONG_5:
SONG_6:
SONG_7:
-END_MUSIC

```

Este mismo tipo de solución es aplicable para el caso en el que no te quepan todos tus gráficos en la zona reservada por 8BP, aunque como dispones de 8540 bytes para gráficos es menos probable que tengas ese problema.

## 16 Guía de referencia de la librería 8BP

### 16.1 Funciones de la librería

#### 16.1.1 |3D

Este comando activa la proyección 3D en los comandos PRINTSP y PRINTSPALL  
Para proyectar disponemos del comando |3D

Uso

Para activar la proyección 3D:

|3D, 1, <Sprite\_fin>, <offsety>

Para desactivarla:

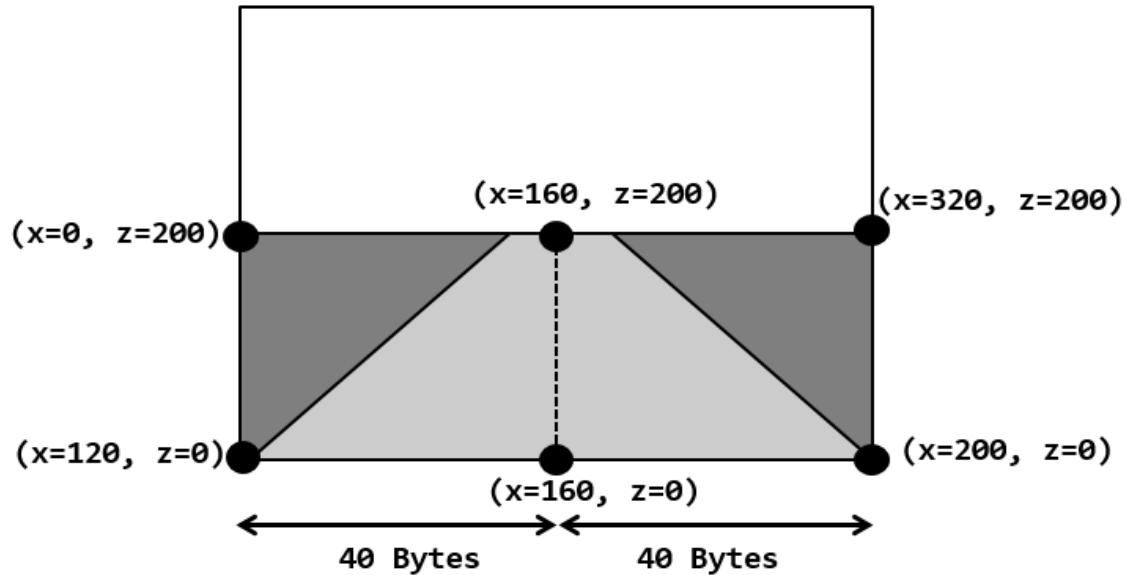
|3D, 0

Los sprites afectados son desde el Sprite 0 hasta el <Sprite\_fin>. Este comando activa la proyección 3D en el comando PRINTSP y en PRINTSPALL. Esto significa que antes de imprimirse en pantalla se calcularán las coordenadas “proyectadas” y a continuación se imprimirán en pantalla. Las coordenadas de los sprites no se ven afectadas, es decir, las coordenadas 2D en la tabla de sprites seguirán siendo las mismas.

Este comando **no afecta a los mecanismos de colisión**, es decir, si usamos COLSPALL y detectamos una colisión entre sprites proyectados, la colisión se está produciendo en el plano 2D.

En cuanto al último parámetro <offsety> es para proyectar más arriba o más abajo, de modo que podemos ubicar los marcadores del juego donde queramos. Al proyectar la pantalla, que mide 200 de alto, se transforma en 100 pixels de alto, de modo que podemos escoger a qué altura ubicamos la proyección. Si un Sprite no es afectado por la proyección por ser superior a <Sprite\_fin>, entonces tampoco le afecta el <offsety>.

La siguiente figura representa cuáles son las coordenadas del mapa del mundo que se proyectan en ciertos puntos representativos de la pantalla cuando MAP2SP es invocado con (yo=0, xo=0).



*Fig. 98 coordenadas del mundo proyectadas*

Si en lugar de  $(x_0=0, y_0=0)$  usamos otra coordenada para MAP2SP, las coordenadas del mundo 2D correspondientes a los puntos referenciados en la imagen, estarán desplazadas en  $(x, z)$  lo que se indique con  $x_0$  e  $y_0$ .

### 16.1.2 |ANIMA

Este comando cambia el fotograma de animación de un sprite, teniendo en cuenta su secuencia de animación asignada

Uso:

**|ANIMA, <sprite number>**

Ejemplo:

**|ANIMA, 3**

El comando lo que hace es consultar la secuencia de animación del sprite, y si es distinta de cero entonces se va a la tabla de secuencias de animación (la primera secuencia valida es la 1 y la última es la 31). Escoge la imagen cuya posición es la siguiente al fotograma actual y actualiza el campo frame de la tabla de atributos de sprites.

Si en la secuencia el siguiente fotograma es cero entonces se cicla, es decir, se escoge el primer fotograma de la secuencia.

Además de cambiar el campo frame, se cambia el campo image y se le asigna la dirección de memoria del lugar donde se almacena el nuevo fotograma.

|ANIMA no imprime el sprite pero lo deja preparado para cuando se imprima, de modo que se imprima el siguiente fotograma de su secuencia

|ANIMA no verifica que el flag de animación esté activo en el byte de estado del sprite. De hecho, nuestro personaje normalmente solo lo vamos a querer animar cuando se mueva y no siempre que se imprima.

Si la secuencia de animación es una “secuencia de muerte” (incluye un “1” en su último fotograma), entonces al llegar al frame cuya dirección de memoria de imagen sea 1, el sprite pasará a inactivo.

La librería 8BP te permite hacer “secuencias de muerte”, que son secuencias que, al terminar de recorrerlas, el sprite pasa a estado inactivo. Esto se indica con un simple “1” como valor de la dirección de memoria del fotograma final. Estas secuencias son muy útiles para definir explosiones de enemigos que están animados con |ANIMA o |ANIMALL. Tras alcanzarles con tu disparo, les puedes asociar una secuencia de animación de muerte y en los siguientes ciclos del juego pasarán por las distintas fases de animación de la explosión, y al llegar a la última pasarán a estado inactivo, no imprimiéndose más. Este paso a inactivo se hace automáticamente, de modo que lo que debes hacer es simplemente chequear la colisión de tu disparo con los enemigos y si colisiona con alguno le cambias el estado con |SETUPSP para que no pueda colisionar más y le asignas la secuencia de animación de muerte, también con |SETUPSP.

Si usas una secuencia de muerte, no te olvides de que el último fotograma antes de encontrar el “1” sea uno completamente vacío, de modo que no quede ningún resto de la explosión.

Ejemplo de secuencia de muerte

```
dw EXPLOSION_1,EXPLOSION_2,EXPLOSION_3,1,0,0,0,0
```

### 16.1.3 |ANIMALL

Este comando anima todos los sprites que tengan el flag de animación activado en el byte de estado. Este comando no tiene parámetros

Uso  
|ANIMALL

**IMPORTANTE:** desde la versión v37 de la librería, este comando solo es accesible via CALL (ver tabla de correspondencias en el apéndice), y no mediante comando RSX. Sacarlo de la lista permitió ahorrar algunos bytes de memoria y se puede seguir usando tanto desde un parámetro en PRINTSPALL como desde una llamada CALL.

Es recomendable su uso si vas a animar muchos sprites ya que es mucho mas rápido que invocar varias veces al comando |ANIMA

Como normalmente se va a desear invocar a ANIMALL en cada ciclo de juego, antes de imprimir los sprites, hay una forma de invocar más eficiente y consiste en poner a “1” el parámetro correspondiente de la función PRINTSPALL, es decir

|PRINTSPALL,1,0

Esta función invoca internamente a ANIMALL antes de imprimir los sprites, ahorrando 1.17ms respecto de lo que se tardaría en invocar separadamente |ANIMALL y |PRINTSPALL

#### **16.1.4 |AUTO**

Este comando mueve un sprite (cambia sus coordenadas) de acuerdo a sus atributos de velocidad Vy,Vx. Estos atributos son los que tenga el sprite en la tabla de sprites.

Uso:

**|AUTO, <sprite number>**

Ejemplo:

**|AUTO, 5**

Lo que hace este comando es actualizar las coordenadas en la tabla de sprites, sumando la velocidad a la coordenada actual

Las coordenadas nuevas son

X nueva = coordenada X actual + Vx

Y nueva = coordenada Y actual + Vy

No es necesario que el sprite tenga el flag de movimiento automático activo en el campo status

#### **16.1.5 |AUTOALL**

Este comando mueve todos los sprites que tengan el flag de movimiento automático activo, de acuerdo a sus atributos de velocidad Vy, Vx.

Uso:

**|AUTOALL, <flag de enrutado>**

Ejemplo

**|AUTOALL,1** invoca a |ROUTEALL antes de mover los sprites

**|AUTOALL,0** no invoca a |ROUTEALL

**|AUTOALL** se utiliza como parámetro el último valor usado (tiene memoria)

El flag de enrutado es opcional. Puesto que el comando ROUTEALL no modifica las coordenadas de los sprites, deben ser movidos con AUTOALL e impresos (y animados) con PRINTSPALL. Es por ello que dispones de un parámetro opcional en |AUTOALL, de modo que AUTOALL,1 invoca internamente a ROUTEALL antes de mover el sprite, ahorrándote una invocación desde BASIC que siempre va a suponer un precioso milisegundo.

#### **16.1.6 |COLAY**

Detecta la colisión de un sprite con el mapa de pantalla (el layout). Tiene en cuenta el tamaño de dicho sprite para saber si colisiona, y considera que los elementos del layout miden todos 8x8 pixeles de mode 0 (es decir, 4 bytes x 8 líneas). Se puede invocar con 3,2,1 o ningún parámetro. Si se invoca sin parámetros se usarán los valores de la última llamada con parámetros y es mucho más rápido.

Uso:

```
|COLAY, <umbral ASCII>, @colision% ,<sprite number>
|COLAY, @colision% ,<sprite number>
|COLAY, <num_sprite>
|COLAY
```

El parámetro opcional <umbral ASCII> basta con usarlo en una primera invocación para establecer el umbral de colisión en el comando COLAY. Este umbral representa el mayor código ASCII del elemento de layout que es considerado como “no colision”. Por defecto es 32 (el del espacio en blanco). Para configurar el umbral que deseas, consulta la tabla ASCII del comando LAYOUT.

Ejemplo:

```
|COLAY, 65, @col%,31 : rem sprite es 31, umbral es 65
```

La variable que uses para colisión puede llamarse como quieras. Yo he puesto “col”

Esta rutina modifica la variable colision ( la cual debe ser entera y por eso el “%”) poniéndola a 1 si hay colisión del sprite indicado con el layout. Si no hay colisión el resultado es 0.

```
10 xanterior=x
20 x=x+1
30 |LOCATESP,0,y,x: ' posicionamos el sprite en nueva posicion
40 |COLAY,@colision%,0: 'chequeo de la colision
```

Ahora comprobamos la colisión y si hay colisión lo dejamos en su ubicación anterior

```
50 if colision%=1 then x=xanterior: LOCATESP,0,y,x
```

También puedes utilizar el comando |MOVER para posicionar el sprite y hacer el chequeo

```
10 |COLAY,65,@col,31: 'configuracion. Solo lo hacemos una vez
20 |MOVER,31,1,1: ' lo movemos a la derecha y abajo
30 |COLAY: ' lo invitamos sin parámetros (mas rápido)
30 if col THEN MOVER,31,-1,-1 : rem ha colisionado y por eso lo
dejo donde estaba
```

### 16.1.7 |COLSP

Este comando permite detectar la colisión de un sprite con el resto de sprites que tengan el flag de colisión activo

Uso :

Para configurar:

```
|COLSP, 32, <sprite inicial>, <sprite final>
|COLSP, 33, @colision%
|COLSP, 34, dy, dx
```

Para detector colisiones:

**|COLSP,<sprite number>, @colsp%**

Ejemplo:

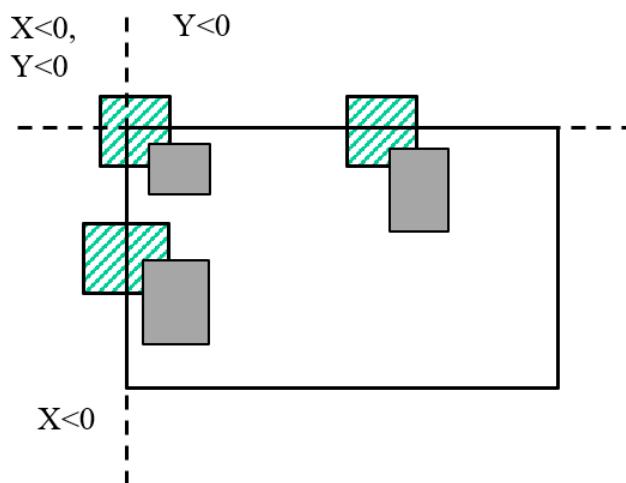
**col% = 0**

**|COLSP,0,@col%**

La función retorna en la variable que le pasemos como parámetro, el número del sprite con el que colisiona, o si no hay colisión retorna un 32 pues el sprite 32 no existe (solo existen del 0 al 31).

**IMPORTANTE:** la variable de colisión del comando COLSP no es la que se usa en el comando COLSPALL. Son variables diferentes (a menos que le pases a ambos comandos la misma variable para que actúen sobre ella)

**IMPORTANTE:** no se detectan colisiones con sprites que tengan alguna de sus dos coordenadas negativas con sprites que las tengan positivas



**Fig. 99 colisiones no detectables**

Al igual que la impresión de sprites con PRINTSPALL, la función COLSP chequea los sprites comenzando en el 31 y acabando en el cero. Si tienen flag de colisión de sprites activo (bit 2 del byte de status) entonces se comprueba la colisión. Si dos sprites colisionan a la vez con nuestro sprite, se retorna el número de sprite mayor pues es el que se comprueba antes.

#### **Invocaciones para configurar el comando:**

Existe una forma de configurar COLSP para que realice menos trabajo chequeando la colisión de menos sprites y así ahorrar tiempo de ejecución. La configuración se la indicaremos con el uso del sprite 32 (el cual no existe).

**|COLSP, 32, <sprite inicial a chequear>, <sprite final a chequear>**

Si por ejemplo los enemigos de nuestro personaje son los sprites 25 al 30, y los configuramos como colisionables (no colisionadores) podemos invocar (una sola vez) al comando así:

### **|COLSP, 32, 25, 30**

Con eso estaremos indicando que cualquier invocación posterior al comando |COLSP tan solo debe chequear la colisión de los sprites del 25 al 30 (siempre que tengan el flag “collided” activo).

Esta estrategia permite reducir bastante tiempo, por ejemplo, si solo debemos chequear 6 enemigos, pre-configurando el comando para que solo chequee desde el 25 en adelante, podemos ahorrar hasta 2.5ms en cada ejecución. Esto se hace especialmente importante en juegos donde el personaje puede disparar, ya que en cada ciclo de juego al menos habrá que chequear la colisión del personaje y de los disparos.

Otra interesante optimización, capaz de ahorrar 1.1 milisegundos en cada invocación, es decirle al comando que siempre use la misma variable BASIC para dejar el resultado de la colisión. Para ello se lo indicaremos usando como sprite el 33, que tampoco existe

```
col% = 0
|COLSP, 33, @col%
```

Una vez ejecutadas estas dos líneas, las siguientes invocaciones a COLSP, dejarán el resultado en la variable col, sin necesidad de indicarlo, por ejemplo:

```
|COLSP, 23
```

Por último, es posible ajustar la sensibilidad del comando COLSP, decidiendo si el solape entre sprites debe ser de varios pixels o de uno solo, para considerar que ha habido colisión.

Para ello se puede configurar el número de pixels de solape necesario tanto en la dirección Y como en la dirección X, usando el comando COLSP y especificando el sprite 34 (que no existe)

```
|COLSP, 34, dy, dx
```

Los valores por defecto para dy y dx son 2 y 1 respectivamente. Ten en cuenta que en la dirección Y se consideran pixels, pero en la dirección X se consideran bytes (un byte son dos pixels en mode 0).

Para una detección con un solape mínimo (de un pixel en vertical y/o un byte en horizontal) debes hacer:

```
|COLSP, 34, 0, 0
```

### **16.1.8 |COLSPALL**

Uso:

Para configurar:

```
|COLSPALL, @colisionador%, @colisionado%
```

Para comprobar las colisiones

```
|COLSPALL
|COLSPALL, <colisionador inicial>
```

Esta función comprueba quien ha colisionado (entre el grupo de sprites que tengan a “1” el flag de colisionador del byte de status) y con quien ha colisionado (entre el grupo de sprites que tengan a “1” el flag de colisión del byte de status).

Es una función muy recomendable cuando tienes que manejar colisiones de tu personaje y de varios disparos, ya que ahorra invocaciones a COLSP y por consiguiente, acelera tu videojuego.

**Importante:** los colisionadores (bit 5 de estado) se comprueban desde el 31 hasta el 0. Para cada colisionador, los colisionables (bit 1 de estado) también se comprueban desde el 31 al 0. Ello nos permite determinar quien va a colisionar en caso de solape multiple

**IMPORTANTE:** al igual que ocurre con COLSP, no se detectan colisiones con sprites que tengan alguna de sus dos coordenadas negativas con sprites que las tengan positivas

En caso de invocar a COLSPALL con un único parámetro,

**|COLSPALL, <colisionador inicial>**

Se explorarán los colisionadores desde el colisionador indicado hasta el sprite cero, en orden descendente. De este modo si necesitas detectar mas de una colision por ciclo de juego, podras hacerlo invocando sucesivamente a COLSPALL hasta que la variable colisionador tome el valor 32

### 16.1.9 |LAYOUT

Uso:

**|LAYOUT, <y>,<x>, <@cadena\$>**

Ejemplo:

**cadena\$ = "XYZZZZ ZZ"**  
**|LAYOUT, 0,1, @cadena\$**

ojo, usar |LAYOUT, 0,1, "XYZZZZ ZZ" sería incorrecto en un CPC464 aunque funciona en un CPC6128. Además, en CPC6128 puedes obviar el uso de la "@" pero en CPC464 es obligatorio.

Esta rutina imprime una fila de sprites para construir el layout o "laberinto" de cada pantalla. Además de dibujar el laberinto, o cualquier gráfico en pantalla construido con pequeños sprites de 8x8, también podrás detectar las colisiones de un sprite con el layout, usando el comando |COLAY

Los sprites a imprimir se definen con un string, cuyos caracteres (32 posibles) representan a uno de los sprites siguiendo esta sencilla regla, donde la única excepción es el espacio en blanco que representa la ausencia de sprite.

<b>Caracter</b>	<b>Sprite id</b>	<b>Codigo ASCII</b>
“ “	NINGUNO	32
“;”	0	59
“<”	1	60
“=”	2	61
“>”	3	62
“?”	4	63
“@”	5	64
“A”	6	65
“B”	7	66
“C”	8	67
“D”	9	68
“E”	10	69
“F”	11	70
“G”	12	71
“H”	13	72
“I”	14	73
“J”	15	74
“K”	16	75
“L”	17	76
“M”	18	77
“N”	19	78
“O”	20	79
“P”	21	80
“Q”	22	81
“R”	23	82
“S”	24	83
“T”	25	84
“U”	26	85
“V”	27	86
“W”	28	87
“X”	29	88
“Y”	30	89
“Z”	31	90

*Tabla 6 correspondencia entre caracteres y Sprites para el comando /LAYOUT*

**IMPORTANTE:** Tras imprimir el layout puedes cambiar los sprites para que sean personajes, por lo que seguirás disponiendo de los 32 sprites

Las coordenadas y,x se pasan en formato caracteres. La librería mantiene internamente un mapa de 20x25 caracteres, por lo que las coordenadas toman los siguientes valores:

y toma valores [0,24]

x toma valores [0,19]

Los sprites a imprimir deben ser de 8x8 píxeles. son "ladrillos" ("bricks" en inglés). A este tipo de concepto también se le suele llamar "tiles" (azulejos)

Si usas otros tamaños de sprite, esta función no funcionará bien. Realmente imprimirá los sprites, pero si un sprite es grande tendrás que colocar espacios en blanco para dejarle espacio.

La librería mantiene un mapa interno del layout y esta función actualiza los datos del mapa interno del layout de modo que será posible detectar colisiones. Dicho mapa es un array de 20x25 caracteres, donde cada carácter se corresponde con un sprite

El @string es una variable de tipo cadena. no puedes pasar directamente la cadena, aunque en el CPC6128 el paso de parámetros lo permite, pero sería incompatible con CPC464

#### Precauciones:

La función no valida la cadena que le pasas. Si contiene minúsculas u otro carácter diferente de los permitidos puede provocar efectos indeseados, tales como el reinicio o cuelgue del ordenador. ¡Tampoco puede ser una cadena vacía!

Los límites establecidos con SETLIMITS deben permitir que se imprima donde deseas. Si posteriormente quieras hacer clipping en una zona más reducida puedes invocar de nuevo a SETLIMITS cuando todo el layout este impreso

#### Ejemplo:

```

2070 |SETLIMITS,0,80,0,200
2090 c$(1)= "PPPPPPPPPPPPPPPPPPP P"
2100 c$(2)= "PU P"
2110 c$(3)= "P P"
2120 c$(4)= "P P"
2130 c$(5)= "P TPPPPU TPPPPPPP" ; sprite 20 --> O arbusto
2140 c$(6)= "P TP" ; sprite 21 --> P roca
2150 c$(7)= "P P" ; sprite 22 --> Q nube
2160 c$(8)= "P P" ; sprite 23 --> R agua
2170 c$(9)= "P YYYYYYYYYY P" ; sprite 24 --> S ventana
2190 c$(10)="P TPPPPPPPPU P" ; sprite 25 --> T arco de puente derecho
2195 c$(11)="P P" ; sprite 26 --> U arco de puente izq
2200 c$(12)="P P" ; sprite 27 --> V bandera
2210 c$(13)="P P" ; sprite 28 --> W planta
2220 c$(14)="YYYYYYYYYP PYYYYYY" ; sprite 29 --> X pico de torre
2230 c$(15)="RRRRRRRRRR RRRRRRR" ; sprite 30 --> Y césped
2240 c$(16)="PPPPPPPPPP PPPP" ; sprite 31 --> Z ladrillo
2250 c$(17)="PU TP PU TP" ; sprite 32 --> A flor
2260 c$(18)="P T U P" ; sprite 33 --> B flor
2270 c$(19)="P P" ; sprite 34 --> C flor
2271 c$(20)="P P" ; sprite 35 --> D flor
2272 c$(21)="P W P" ; sprite 36 --> E flor
2273 c$(22)="PP W PP" ; sprite 37 --> F flor
2274 c$(23)="PPPPPPPPPPPPPPPPPPPP" ; sprite 38 --> G flor

2280 for i=0 to 23
2281|LAYOUT,i,0,@c$(i)
2282 next

```

En este ejemplo se usan varios ladrillos que previamente se han creado con la herramienta “spedit”

; sprite 20 --> O arbusto

; sprite 21 --> P roca

; sprite 22 --> Q nube

; sprite 23 --> R agua

; sprite 24 --> S ventana

; sprite 25 --> T arco de puente derecho

; sprite 26 --> U arco de puente izq

; sprite 27 --> V bandera

; sprite 28 --> W planta

; sprite 29 --> X pico de torre

; sprite 30 --> Y césped

; sprite 31 --> Z ladrillo



#### 16.1.10 |LOCATESP

Este comando cambia las coordenadas de un sprite en la tabla de atributos de sprites

Uso

|LOCATESP, <sprite number>, <y>,<x>

Ejemplo

|LOCATESP,0,10,20

Una alternativa a este comando, si solo deseamos cambiar una coordenada es usar el comando POKE de BASIC, insertando en la dirección de memoria ocupada por la coordenada X o Y, el valor que queramos. Si deseamos introducir una coordenada negativa es necesaria la función |POKE, ya que con el POKE de BASIC sería ilegal

El comando |LOCATE no imprime el sprite, sólo lo posiciona para cuando sea impreso.

### 16.1.11 |MAP2SP

Esta función recorre el mapa del mundo que se describe en el fichero map\_table.asm y transforma en sprites los map ítems que puedan estar entrando en pantalla parcial o totalmente.

Uso

|MAP2SP, <y>, <x>  
|MAP2SP, <status>

Ejemplo

|MAP2SP, 1500, 2500

Los sprites creados por MAP2SP se crean por defecto con estado 3, es decir, con el flag de impresión activo (PRINTSPALL lo imprime) y con el flag de colisionable activo (COLSP colisionará con el). Si necesitas que los sprites sean creados con otro estado, simplemente debes invocar una vez al comando MAP2SP con un solo parámetro indicando el estado con el que se deben crear los sprites

Si por casual MAP2SP se encuentra con mas de 32 items para traducir a sprites, ignorará los que excedan de 32.

|MAP2SP, <status>

|MAP2SP , 1 : REM esto configura al comando MAP2SP para que se impriman pero no sean colisionables

Los parámetros <y>,<x> de la función son el origen móvil desde el cual se muestra el mundo en la pantalla. Hay otros tres parámetros que se encuentran en la MAP\_TABLE, la tabla con la que se define el mundo. Estos parámetros son el alto máximo, el ancho máximo (en negativo) y el numero de ítems del mundo (máximo 82)

Cada ítem es una tupla de 3 parametros precedidos por el nemónico “DW”:

DW Y, X, <imagen>

```
;MAP TABLE
;-----
; primero 3 parametros antes de la lista de "map items"
dw 50 ; maximo alto de un sprite por si se cuela por arriba y ya hay
que pintar parte de el
```

```

dw -40 ; máximo ancho de un sprite por si se cuela por la izquierda
(numero negativo)
db 64 ; numero de items del mapa a considerar. como mucho debe ser 82
; a partir de aqui comienzan los items
dw 100,10,CASA; 1
dw 50,-10,CACTUS;2
dw 210,0,CASA;3
dw 200,20,CACTUS;4
dw 100,40,CASA;5
dw 160,60,CASA;6
dw 70,70,CASA;7
dw 175,40,CACTUS;8
dw 10,50,CASA;9
dw 250,50,CASA;10
dw 260,70,CASA;11
dw 290,60,CACTUS;12
dw 180,90,CASA;13
dw 60,100,CASA;14
...

```

### 16.1.12 |MOVER

Este comando mueve un sprite de forma relativa, es decir, sumando a sus coordenadas unas cantidades relativas

Uso:

**|MOVER,<sprite number>, <dy>, <dx>**

Ejemplo:

**|MOVER,0,1,-1**

El ejemplo mueve el sprite 0 hacia abajo y hacia la derecha a la vez. No es necesario que el sprite tenga el flag de movimiento relativo activado

Hay una forma de usar |MOVER sin especificar ni “dy” ni “dx”. Para ello indicaremos el sprite 32, que no existe, y pondremos como parámetros las direcciones de memoria de las variables que queremos usar para almacenar tanto “dy” como a “dx”.

La dirección de memoria de una variable se obtiene simplemente anteponiendo el símbolo “@”

Ejemplo:

**dy%= 5**

**dx%= 2**

**|MOVER,32, @dy, @dx**

A partir de este momento podremos usar:

**|MOVER, <id>**

Y con ello el sprite “id” se moverá según indiquen las variables dy, dx. Este mecanismo también funciona con |MOVERALL

### **16.1.13 |MOVERALL**

Este comando mueve de forma relativa todos los sprites que tengan el flag de movimiento relativo activado

Uso

**|MOVERALL, <dy>, <dx>**

Ejemplo

**|MOVERALL, 2,1**

El ejemplo mueve todos los sprites con flag de movimiento relativo hacia abajo (2 líneas) y 1 byte hacia la derecha.

Si no se especifican parámetros, se usarán las variables especificadas en la invocación de MOVER con sprite 32, es decir

**|MOVER,32, @dy, @dx  
|MOVERALL**

Es equivalente a |MOVERALL, dy, dx

Este uso “avanzado” del comando evita el paso de parámetros en cada invocación y por lo tanto es más rápido, lo cual es fundamental en nuestros programas BASIC

### **16.1.14 |MUSIC**

Este comando permite que una melodía comience a sonar

Uso:

**|MUSIC,<numero\_melodía>,<velocidad>**

El número de la melodía estará comprendido entre 0 y 7.

La velocidad “normal” es 5. Si usamos un número superior se reproducirá más lentamente y si el número es inferior se reproducirá más deprisa.

Ejemplo:

**|MUSIC,0,5**

Internamente el comando lo que hace es instalar una interrupción que se dispara 300 veces por segundo. Si ponemos velocidad 5, una de cada 5 veces que se dispara, se ejecuta la función de reproducción musical

Al basarse en una interrupción, es necesario que haya un programa en ejecución para que pueda sonar la música, pues mientras el intérprete BASIC se encuentra esperando a recibir comandos, dichas interrupciones no están habilitadas. Si ejecutas simplemente el comando |MUSIC, no oirás nada, pero si lo ejecutas dentro de un programa como el que se muestra a continuación, la música sonará

```
10 |MUSIC,0,5
20 goto 20: ' bucle infinito. Al estar en ejecución, la música suena
```

### **16.1.15 |MUSICOFF**

Este comando paraliza la reproducción de cualquier melodía. No tiene parámetros

Uso:

**|MUSICOFF**

Internamente lo que hace es desinstalar la interrupción

### **16.1.16 |PEEK**

Este comando lee el valor de un dato de 16 bit de una dirección de memoria dada. Está pensado para consultar las coordenadas de sprites que se mueven con movimiento automático o relativo

Uso

**|PEEK, <dirección>, @dato%**

Ejemplo

**dato%=0**

**|PEEK, 27001, @dato%**

Si las coordenadas son solo positivas y menores de 255 puedes usar el comando PEEK de BASIC, ya que es algo más rápido.

### **16.1.17 |POKE**

Este comando introduce un dato de 16 bit (positivo o negativo) en una dirección de memoria. Esta pensada para modificar coordenadas de sprites, ya que el comando POKE no puede manejar coordenadas negativas o mayores de 255 ya que POKE funciona con bytes mientras que |POKE es un comando que funciona con 16bit

Uso:

**|POKE, <dirección>, <valor>**

Ejemplo:

**|POKE, 27003, 23**

Este ejemplo pone el valor 23 en la coordenada x del sprite 0.

Es una función muy rápida, aunque si vas a manejar solo coordenadas positivas es mejor usar POKE pues es más rápida aun

### **16.1.18 |PRINTAT**

PRINTAT puede imprimir una cadena de caracteres usando un nuevo juego de caracteres mas pequeño (los llamo “minicaracteres”). Este nuevo comando permite usar el mecanismo de transparencia de los sprites, de modo que podras imprimir caracteres respetando el fondo. Funciona del siguiente modo:

Uso:

**|PRINTAT,<flag transparencia>, y,x,@string**

Ejemplo:

```
cad$= "Hola"  
|PRINTAT,0,100,10, @cad$
```

El comando |PRINTAT imprime cadenas de caracteres y no variables numéricas, por lo que si quieres imprimir un número (por ejemplo, los puntos en el marcador de tu videojuego) debes hacerlo así:

```
puntos=puntos+1  
cad$= str$(puntos)  
|PRINTAT,0,100,10, @cad$
```

El comando |PRINTAT no se ve afectado por los límites para el clipping establecidos con |SETLIMITS. Esto es lo más lógico puesto que normalmente usarás PRINTAT para imprimir puntuaciones en tus marcadores, que se encontrarán fuera del área delimitada por |SETLIMITS

A diferencia del comando PRINT del BASIC, el comando |PRINTAT es bastante rápido y puede ser utilizado para actualizar los marcadores de tu videojuego con frecuencia. PRINTAT usa un alfabeto redefinible, que puede contener una versión reducida o diferente de los caracteres “oficiales” del amstrad. Por defecto 8BP proporciona un pequeño alfabeto compuesto por números, letras mayúsculas y algunos símbolos. Es el siguiente:

```
"0123456789ABCDEFGHIJKLMNPQRSTUVWXYZ:!.,"
```

No podrás usar un carácter que no se encuentre entre este conjunto, como las letras minúsculas. Si tratas de hacerlo, se imprimirá el último carácter definido en la cadena (en este caso el espacio).

Los caracteres de dicho alfabeto miden todos lo mismo: 4pixels de ancho x 5 pixels de alto, es decir 2 bytes x 5 líneas.

El alfabeto por defecto no contiene minúsculas y faltan muchos símbolos, aunque puedes crear tu propio alfabeto que los contenga.

### 16.1.19 |PRINTSP

Uso:

```
|PRINTSP, <sprite id >, <y >,<x>  
|PRINTSP, <sprite id >  
|PRINTSP, 32, <numero de pixels de fondo>
```

Ejemplo:

Imprime el sprite 23 en las coordenadas y=100, x=40 (actualizando sus coordenadas)  
**|PRINTSP, 23,100,40**

Ejemplo:

Imprime el sprite 23 en las coordenadas que ya tenga asignadas en la tabla de sprites:  
**|PRINTSP, 23**

Si se especifica el sprite 32 (el cual no existe), entonces el siguiente parámetro se utiliza para especificar el numero de bits de fondo en la impresión transparente. Si se usa 1 bit, entonces se podrán usar 2 colores de fondo. Si se especifica un 2, entonces se podrán usar 4 colores de fondo.

Si se especifica un sprite\_id <32 entonces el comando imprime un sprite en la pantalla, y si se especifican coordenadas, además las actualiza.

Las coordenadas consideradas son:

- Número de líneas en vertical [-32768..32768]. Las correspondientes al interior de la pantalla son [0..199]
- Número de bytes en horizontal [-32768..32768]. Las correspondientes al interior de la pantalla son [0..79]

Normalmente en la lógica de un videojuego harás uso de |PRINTSPALL, ya que es más rápido imprimirlos todos de golpe. Sin embargo, en otros momentos del juego puede interesarte imprimir sprites por separado. En este ejemplo se muestra la bajada de un “telón”, usando un solo sprite que se repite horizontalmente y al ir bajando va “tiñendo” de rojo la pantalla, dando la sensación de un telón que baja

<pre> 7089 telon=&amp;8ec0 7090  setupsp,1,9,telon 7100 for y=8 to 168 step 4 7110 for x=12 to 64 step 4 7111  PRINTSP,1,y,x 7112 next 7113 next </pre>	
---	--

*Fig. 100 Un ejemplo de uso de PRINTSP*

### 16.1.20 |PRINTSPALL

Esta rutina imprime de una sola vez todos los sprites que tengan el bit0 de estado activo.

Uso:

|PRINTSPALL, <ordenini>, <ordenfin>, <flag anima>, <flag sync>  
|PRINTSPALL, < tipo de orden>,

Ejemplo:

Con los siguientes valores, el comando imprime todos los sprites animándolos primero y sin sincronizar con barrido, y sin ordenar:

|PRINTSPALL, 0, 0, 1, 0  
|PRINTSPALL, 0, 1, 0: rem si se omite el ordenini, toma el ultimo valor asignado, o cero si nunca se asignó

**flag de animación**, puede valer 1 o 0. Si se asigna un 1, entonces antes de imprimir los sprites se cambia el fotograma en su secuencia de animación, siempre que los sprites tengan el bit 3 de estado activo

El <flag sync> es un flag de sincronización con el barrido de pantalla. Puede ser 1 o 0 . La sincronización solo tiene sentido si compilas el programa con un compilador como “Fabacom”. La lógica en BASIC se ejecuta lentamente y sincronizar con el barrido produce pequeñas esperas adicionales en cada ciclo del juego de modo que no es conveniente.

Como regla general, solo es conveniente si tu juego es capaz de generar 50fps por segundo, o lo que es lo mismo, un ciclo completo de juego cada 20 milisegundos. Si compilas el juego con un compilador como “fabacom”, entonces es recomendable que sincronices con el barrido de pantalla porque casi seguro que vas a alcanzar esos 50fps y si los superas, tu juego producirá más fotogramas de los que puede mostrar la pantalla y entonces algunos no se podrán mostrar y el movimiento no será suave.

Cuantos más sprites tengas en pantalla imprimiéndose, más tardará el comando, aunque es muy rápido. Hay muchos sprites que pueden aparecer en pantalla, pero no es necesario imprimir (pueden tener el bit 0 de estado desactivado) como pueden ser frutas, monedas, elementos de bonus en general y/o personajes que no se mueven y no tienen animación. Aunque no se impriman pueden tener el bit de colisión activo y así afectar en la rutina |COLSP y |COLSPALL

**Los parámetros de orden (“ordenini”, “ordenfin”)** indican los sprites inicial y final que definen el grupo de sprites ordenados por coordenada “Y” que vamos a imprimir. Por ejemplo, si asignamos los valores 0,0 entonces se imprimirán secuencialmente desde el sprite 0 hasta el sprite 31. Si asignamos 0,8 se imprimirán del 0 al 8 ordenados (9 sprites) y del 10 al 31 de modo secuencial. Si ponemos un 0,31 se imprimirán todos los sprites ordenados. Si ponemos un 10,20 se imprimirán secuencialmente los sprites del 0 al 9, luego se imprimirán ordenados del 10 al 20 y finalmente se imprimirán secuencialmente del 21 al 31

El ordenamiento es muy útil para hacer juegos tipo “Renegade” o “Golden AXE”, donde es necesario dar un efecto de profundidad.

Si el parámetro “ordenini” se omite, se considera el ultimo valor asignado, o bien cero si nunca se ha asignado un valor. Ademas, si vas a modificar alguno de los dos parámetros de ordenamiento, conviene primero ejecutar PRINTSPALL,0,0,0,0 para que primero se reordenen los sprites secuencialmente antes de ordenarlos con una nueva configuración.

Imprimir de forma ordenada es más costoso computacionalmente que imprimir de forma secuencial. Si solo tienes 5 sprites que deben ser ordenados, pasa un 4 como parámetro de ordenamiento, no pases un 31. Ordenar todos los sprites lleva unos 2.5 ms pero si ordenas solo 5 te puedes ahorrar 2ms. Quizás tengas muchos sprites y no merezca la pena ordenar algunos, como los disparos o sprites que sabes que no se van a solapar.

El ordenamiento de |PRINTSPALL es parcial, es decir, solo se ordena una pareja de sprites desordenados en cada invocación. Puede que alguna vez desees que la ordenación sea completa en cada fotograma. Es decir, que no se ordene un par de sprites en cada invocación a PRINTSPALL, sino tener la seguridad de que todos estan

ordenados. La librería 8BP te lo permite mediante sus cuatro modos de ordenamiento, que puedes establecer mediante la invocación del comando PRINTSPALL con un solo parámetro (basta con ejecutarlo una vez para fijar el modo de ordenación):

<b>PRINTSPALL,0 : ordenamiento parcial usando Ymin</b>
<b>PRINTSPALL,1 : ordenamiento completo usando Ymin</b>
<b>PRINTSPALL,2 : ordenamiento parcial usando Ymax</b>
<b>PRINTSPALL,3 : ordenamiento completo usando Ymax</b>

Los ordenamientos que usan Ymax se basan en la coordenada Y mayor de los sprites, es decir, donde se encuentran sus pies en lugar de su cabeza. Si los sprites son del mismo tamaño, un ordenamiento basado en Ymin te puede servir, pero si los sprites tienen diferente altura puede que desees ordenar según donde se encuentren los pies de cada personaje y para ello tendrás que usar el modo de ordenar 2 o el 3.

Los modos de ordenar con Ymax son mas lentos, aproximadamente 0.128 ms por sprite, de modo que úsalos cuando los necesites realmente.

La ordenación completa consume muy poco mas que la parcial (aproximadamente 0.3ms). Esto es debido a que los sprites apenas se desordenan de un fotograma al siguiente, pero incluso esos 0.3ms merece la pena ahorrarlos si es posible.

Hay un comportamiento de esta función muy interesante para ahorrar 1ms en su ejecución. Consiste en invocarla con parámetros una vez y las siguientes veces invocarla sin parámetros. En ese caso se asumirán que, aunque no se pasen parámetros, sus valores son iguales a los últimos que se pasaron. De esta manera el analizador sintáctico trabaja menos y reduce el tiempo de ejecución.

### 16.1.21 |RINK

Este comando permite realizar una animación por tintas.

Uso:

|RINK, <tinta\_inicial>, <color1>, <color2>, . . . , <colorN>  
|RINK, <step>

RINK rota un conjunto de tintas comenzando en la tinta inicial N tintas (cualquier numero de tintas), según el tamaño del patrón de color que se defina.

La velocidad de rotación se puede controlar con el parámetro step, que indica el número de saltos de color que realiza cada tinta en una invocación.

Recomendación: debido al uso de interrupciones |RINK produce “parones” en algunos casos cuando se usa a la vez que el comando |MUSIC a velocidad 6. En caso de querer usar ambos a la vez sin que haya interferencias, usa otra velocidad para la música (puedes usar velocidad 5 o 7, ambas te funcionarán bien).

Ejemplos:

Este comando define un patrón de 4 rojos (color =3) y 4 amarillos (color =24) a rotar desde la tinta 8 hasta la tinta 15

|RINK, 8, 3, 3, 3, 3, 24, 24, 24, 24

Este comando define un patrón de 2 blancos (color =26) y 2 grises (color=13) a rotar desde la tinta 3 hasta la tinta 6

|RINK, 3, 26, 26, 13, 13

Rota un color del patrón todas las tintas

|RINK, 1

En un patrón de 8 colores, el siguiente comando deja todo como estaba

|RINK, 8

Este comando no haría nada, salvo forzar que las tintas adopten el color del patrón

|RINK, 0

### 16.1.22 |ROUTEALL

Este comando te permite enrutar a todos los sprites que tengan activo el flag de ruta en su byte de estado a través de la ruta que tengan asignada (parámetro 15 de SETUPSP)

Uso:

|ROUTEALL

No tiene parámetros por lo que es muy sencillo de invocar. Este comando lo que hace internamente es llevar una cuenta de pasos por el segmento que está cursando cada sprite, de modo que, si se acaba el segmento, altera la velocidad del sprite.

El comando no modifica las coordenadas de los sprites, de modo que deben ser movidos con AUTOALL e impresos (y animados) con PRINTSPALL. Es por ello que dispones de un parámetro opcional en |AUTOALL, de modo que |AUTOALL,1 invoca internamente a ROUTEALL antes de mover el sprite, ahorrándote una invocación desde BASIC que siempre va a suponer un precioso milisegundo.

Las rutas se definen en el fichero de rutas routes\_tujuego.asm

```
; DEFINICION DE CADA RUTA
;=====
ROUTE0; un circulo
;-----
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
```

El último segmento es cero, indicando que la ruta se ha terminado y que el sprite debe comenzar desde el principio. Asegurate de que el número de pasos de cada segmento no excede de 250 y que tanto Vy como Vx estan entre -127 y 127.

Para asignar una ruta a un sprite debes usar el comando SETUPSP especificando el parámetro 15. El siguiente ejemplo asocia la ruta 3 al sprite 31

|SETUPSP, 31, 15, 3

Existen 4 funcionalidades que puedes usar en mitad (**ojo en mitad, no al final**) de cualquier ruta:

Código de escape (campo “número de pasos”)	Descripción	Ejemplo
255	Cambio de estado del sprite.	DB 255,3,0 Estado pasa a valor 3. El cero del final es de relleno
254	Cambio de secuencia de animación del sprite	DB 254,10,0 Se asocia la secuencia 10. El cero es de relleno. Si la secuencia asignada es la que ya tiene el sprite, entonces es inocuo (no se reinicia la secuencia de animación)
253	Cambio de imagen	DB 253 DW new_img Se asocia la imagen “new_img” que debe ser una dirección de memoria
252	Cambio de ruta	DB 252,2,0 Se asocia la ruta 2
251	Pasa al siguiente frame de la animación.	DB 251,0,0 Se anima el Sprite. Los dos ceros son de relleno

**IMPORTANTE:** ten mucho cuidado de escribir DB y DW donde deben usarse, es decir, por ejemplo, si cambias de imagen debes preceder la imagen con DW y no con DB. Si cometes un error de este tipo, tu ruta no funcionará.

**IMPORTANTE:** Una ruta puede medir a lo sumo 255 bytes y un segmento ocupa 3 bytes, por lo tanto, una ruta puede tener como mucho 84 segmentos. Es posible que necesites construir una ruta aun mas larga y en ese caso podrás hacerlo concatenando el fin de una ruta con un cambio de ruta hacia otra ruta (código 252), y puedes concatenar tantas rutas como deseas.

**IMPORTANTE:** los codigos de escape puedes emplearlos en mitad de una ruta, pero el ultimo segmento no puede ser un código de escape, debe ser un movimiento, aunque sea quedarse quieto, algo como “DB 1,0,0”

En estos casos, |ROUTEALL interpretará que se debe forzar un cambio de estado, secuencia, imagen, ruta del sprite o animar y además, ejecutar el paso siguiente. Los cambios se pueden forzar en cualquier segmento de la ruta, no hace falta que sea al final, y se pueden forzar tantos cambios como se desee.

```
ROUTE0; un disparo a la izquierda
;-----
    db 100,0,-1; cien pasos a la izquierda a velocidad Vx=-1
    db 255,0,0; deactivacion del sprite con status=0
    db 1,0,0 ; no mover nada en este paso
```

```

db 0
ROUTE1; un salto a la derecha
db 253
dw SOLDADO_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 SOLDADO_R1_DOWN
db 1,-5,1; subo para que UP y down encajen
db 2,1,1
db 2,2,1
db 2,3,1
db 2,4,1
db 1,5,1
db 253
dw SOLDADO_R1
db 1,5,1; baja una mas
db 255,13,0; nuevo estado, sin flag ruta y con flag animacion
db 254,32,0; macrosecuencia 32
db 1,0,0; quietooo.!!!!
db 0

```

### 16.1.23 |ROUTESP

Este comando te permite enrutar un solo Sprite que tenga activo el flag de ruta en su byte de estado a través de la ruta que tengan asignada (parámetro 15 de SETUPSP)

Uso:

|ROUTESP, <spriteid>, <pasos>  
|ROUTESP, <spriteid> : rem en este caso “pasos” se considera=1

Este comando mueve un sprite el numero de pasos que deseas a lo largo de la ruta que tenga asignada. El comando hace recorrer al sprite todos los pasos que se indiquen, dejándolo finalmente ubicado en la misma posición que tendría si hubiésemos ejecutado |AUTOALL,1 un número de veces igual al numero de pasos.

### 16.1.24 |SETLIMITS

Este comando establece los límites del área donde se van a poder imprimir sprites o estrellas.

Uso:

|SETLIMITS, <xmin>, <xmax>, <ymin>, <ymax>

Ejemplo que establece toda la pantalla como área permitida

|SETLIMITS,0,80,0,200

Fuera de estos límites se realiza clipping de los sprites, de modo que si un sprite se encuentra parcialmente fuera del área permitida, las funciones |PRINTSP y |PRINTSPALL imprimirán solo la parte que se encuentra dentro del área permitida.

### 16.1.25 |SETUPSP

Este comando carga datos de un sprite en la SPRITES\_TABLE

Uso:

**|SETUPSP, <id\_sprite>, <param\_number>, <valor>**

Ejemplo:

**|SETUPSP, 3, 7, 2**

Permite por ejemplo asignar una nueva secuencia de animación cuando el sprite cambia de dirección, o simplemente cambiar su registro de flags de status

Con esta función podemos cambiar cualquier parámetro de un sprite, menos X, Y (que se hace con LOCATE\_SPRITE)

Solo podremos cambiar un parámetro a la vez. El parámetro que vamos a cambiar se especifica con param\_number. El param\_number es en realidad la posición relativa del parámetro en la SPRITES\_TABLE

Param number	Acción	Possible uso de POKE o  POKE como alternativa
0	cambia el status (ocupa 1 byte)	SI
5	cambia Vy (ocupa 1byte, valor en lineas verticales). También se puede modificar Vx a la vez si lo añadimos al final como parámetro	SI
6	cambia Vx (ocupa 1byte, valor en bytes horizontales)	SI
7	cambia secuencia (ocupa 1byte, toma valores 0..31)	NO
8	cambia frame_id (ocupa 1byte, toma valores 0..7)	SI
9	cambia dir imagen (ocupa 2bytes)	No es igual si se usa una imagen <255. Si se usa una dirección de memoria podría usarse  POKE como alternativa
15	cambia la ruta (ocupa 1bytes)	NO

Ejemplo:

En este ejemplo le hemos dado al sprite 31 la imagen de una nave que está ensamblada en la dirección &a2f8

```
nave = &a2f8
|SETUPSP, 31, 9, nave
```

Hay una forma más sencilla de especificar la imagen para el sprite haciendo uso de la lista IMAGE\_LIST que aparece en el fichero images\_tujuuego.asm. Si tenemos la NAVE en la IMAGE\_LIST, podremos asociar un identificador entre 16 y 255

|SETUPSP,31, 9, 16 : rem el 16 es el identificador de la NAVE en la IMAGE\_LIST

#### IMAGE\_LIST

```
;-----  
; pondremos aqui una lista de las imagenes que queremos usar sin especificar la direccion de memoria desde basic  
; de este modo el comando |SETUPSP,<id>,9,<address> se transforma en |SETUPSP,<id>,9,<numero>  
; la ventaja de no usar direcciones de memoria en BASIC es que si ampliamos los graficos o se reensamblan en  
; direcciones diferentes, el numero que asignemos no cambiara  
; NO tienen que tener todas un numero, solo aquellas que vamos a usar con |setupsp, id, 9,<num>  
; se empiezan a numerar en 16  
; podemos usar hasta 255 imagenes especificadas de este modo  
; no hace falta que la lista tenga 255 elementos. es de longitud variable, incluso puede estar vacia  
;  
DW NAVE ; 16  
DW OTRA_NAVE ; 17  
  
;----- BEGIN IMAGE -----  
NAVE  
db 7 ; ancho  
db 12 ; alto  
db 0 , 0 , 0 , 0 , 0 , 0 , 0  
db 0 , 0 , 0 , 0 , 0 , 0 , 0  
db 0 , 154 , 48 , 0 , 0 , 0 , 0  
db 0 , 112 , 240 , 48 , 0 , 0 , 0  
db 0 , 207 , 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 , 207 , 207 , 80 , 0 , 0 , 0  
db 0 , 0 , 0 , 0 , 0 , 0 , 0  
db 0 , 0 , 0 , 0 , 0 , 0 , 0  
;----- END IMAGE -----
```

En el caso de param\_number=5, podemos incluir Vx como parámetro al final:  
|SETUPSP, 31, 5, Vy, Vx

De este modo actualizaremos las dos velocidades con un solo comando, que cuesta 3.73ms frente a los 6.8 que costaría invocar a dos comandos separadamente.

En el caso de usar param\_number=7, además de cambiar la secuencia de animación, automáticamente el comando actualiza el fotograma, colocándolo en el inicial (el cero) y se actualiza la dirección de la imagen, de modo que ya no necesitas invocar con param\_number=9 para que cambie la imagen del sprite a la primera imagen de la nueva secuencia asignada. Si estás usando |ANIMALL antes de imprimir o |PRINTSPALL con flag de animación, aunque SETUPSP te coloque la animación en el frame cero, saltarás al frame 1 antes de imprimir. Esto normalmente no va a suponer ningún problema, pero en caso de tratarse de una secuencia de muerte en la que por ejemplo el primer frame es para borrar al sprite, puede que no te interese pasar directamente al frame 1. En ese caso un sencillo truco puede ser repetir el frame cero en la definición

de la secuencia de muerte. Así te aseguras de que dicho frame se vea. Otra opción es quitarle el flag de animación y animarle con ANIMASP después de imprimir.

En el caso de param\_number=15, además de asignar la ruta al sprite en la tabla de atributos, el comando realiza un reseteo de datos internos para que el sprite comience a recorrer dicha ruta a partir del primer segmento de la ruta en cuestión.

### 16.1.26 |STARS

Mueve un banco de hasta 40 estrellas en la pantalla (dentro de los límites establecidos por |SETLIMITS), sin pintar sobre otros sprites que ya existiesen impresos.

**|STARS,<estrella inicial>,<num estrellas>,<color>,<dy>,<dx>**

Ejemplo:

**|STARS, 0, 15, 3, 1, 0**

El ejemplo desplaza 15 estrellas de color 3 (rojo) un píxel verticalmente (ya que dy=1 y dx=0). Invocado repetidas veces da sensación de fondo de estrellas que se desplaza. Cuando una estrella se sale del límite de la pantalla o el establecido por |SETLIMITS, reaparece por el lateral opuesto, de modo que hay sensación de continuidad en el fluir de las estrellas.

El banco de estrellas está situado en la dirección 42540 (=A62C) y tiene capacidad para 40 estrellas, llegando hasta la dirección 42619. Cada estrella consume 2 bytes, uno para la coordenada Y, y el otro para la coordenada X.

Se pueden mover grupos de estrellas por separado, comenzando en la estrella que quieras. Las coordenadas iniciales de las estrellas deben ser inicializadas por el programador.

Ejemplo de inicialización y uso en un scroll de cuatro planos de estrellas para dar sensación de profundidad. Cada plano va a moverse a una velocidad diferente

```
1 MEMORY 23999
10 CALL &6b78: rem instala los commandos RSX
20 banco=42540
30 FOR star=0 TO 39:' bucle para crear 40 estrellas
40 POKE banco+star*2,RND*200
50 POKE banco+star*2+1,RND*80
60 NEXT
70 MODE 0
80 REM vamos a pintar y mover 4 planos de estrellas de 10 estrellas
cada uno
90 |STARS,0,10,3,0,-1: ' el 3 es rojo. Las mas lejanas se mueven mas
despacio
91 |STARS,10,10,2,0,-2: ' el 2 es azul
92 |STARS,20,10,1,0,-3: ' el 1 es amarillo
93 |STARS,30,10,4,0,-4: ' el 4 es blanco. Las mas cercanas van mas
deprisa
95 goto 90
```

Los usos de este comando pueden ser muy diversos.

- Usando varios bancos de estrellas a la vez con diferente velocidad y color puedes dar sensación de profundidad
- Si la dirección de las estrellas es diagonal puedes hacer un “efecto de lluvia”
- Si el color es negro y el fondo es marrón o naranja puedes dar sensación de avance sobre un territorio arenoso
- Si el movimiento es de balanceo y el color de las estrellas es blanco puedes dar sensación de nieve. El movimiento de balanceo lo puedes lograr con un zigzag en X manteniendo la velocidad en Y, o incluso usando funciones trigonométricas como el coseno. Obviamente si usas el coseno en la lógica de un juego va a ser muy lento, pero puedes almacenar el valor del coseno precalculado en un array.

Ejemplo de efecto nieve:

```
1 MEMORY 23999: MODE 0
30 ' inicializacion banco de 40 estrellas
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: IF i=359 THEN i=0
70 GOTO 50
```

Existe un modo de conseguir una ejecución más rápida, y es evitando pasar parámetros. A lo largo de este libro hemos visto como el paso de parámetros es costoso incluso aunque el comando invocado no haga nada. Pues bien, estamos ante un comando que requiere 5 parámetros por lo que es especialmente costoso. Si queremos reducir el tiempo que requiere el BASIC para interpretar los parámetros, simplemente podemos invocar una vez el comando con parámetros y las siguientes veces no pasar parámetros.

|STARS,0,10,1,5,0

|STARS :’ esta invocación sin parámetros asume los mismos valores de la última invocación

Esta posibilidad es especialmente útil en juegos donde queremos invocar el comando en cada ciclo de juego para mover estrellas, pues ahorraremos unos 1.7 ms

IMPORTANTE: al comando STARS le afectan los límites de SETLIMITS pero solo si Vx o Vy son distintos de cero. Si ambos son cero entonces SETLIMITS no le afecta y se pueden pintar estrellas en toda la pantalla.

### 16.1.27 |UMAP

Este comando actualiza el mapa con información ubicada en otra zona de memoria donde tengamos un mapa mayor. EL comando hace que se reconstruya el mapa por completo, incluyendo solo aquellos ítems que cumplen unos determinados rangos de coordenadas X, Y

Uso:

|UMAP, <map\_ini>, <map\_fin>, <y\_ini>, <y\_fin>, <x\_ini>, <x\_fin>

Por ejemplo, si tenemos un mapa ubicado en la dirección 22.000 que ocupa 1500bytes y queremos que se actualice el mapa con las coordenadas de nuestro personaje, con margen suficiente para avanzar en la coordenada Y hasta 100 lineas y en coordenada x hasta 20 bytes en todas direcciones:

**|UMAP, 22000, 23500, y-100, y+100, x-20, x+20**

Este comando chequeara las coordenadas de los ítems localizados en el mapa de la dirección 23000 y si se encuentran dentro de los márgenes de X, Y que hemos puesto, se copiaran en la zona de memoria que 8BP usa para el comando |MAP2SP, es decir, los copiará a partir de la dirección 42040. Eso si, tan solo copiará los que cumplan la condición. Al ser menos items, el comando |MAP2SP se ejecutará mas rápido pues tendrá que leer y comprobar si se encuentran dentro de la pantalla menos items.

**IMPORTANTE:** el mapa a copiar NO debe incluir los 3 parametros de todo mapa:

<Max\_alto> ( que es un DW , es decir 2 bytes)

<max\_ancho> ( que es un DW , es decir 2 bytes)

<Num\_items> ( que es un DB , es decir 1 byte)

Es decir, que son 5 bytes que no se deben incluir en el mapa a copiar.

## 17 Ensamblado de la librería, música y gráficos

Tanto si quieras hacer cambios en la librería como si añades música y gráficos deberás re-ensamblarla. Esto es debido a que, por ejemplo, el player de música está integrado en la librería y necesita conocer donde comienza (dirección de memoria) cada canción, por lo que es necesario re-ensamblar y guardar la versión de la librería específica para tu juego, así como el fichero de gráficos ensamblado y el fichero de música ensamblado.

Como expliqué en el apartado de los “pasos” que debes dar, ésta será una versión de la librería específica para tu juego. Por ejemplo, el comando `|MUSIC,3,5` hará sonar la melodía número 3 que tú mismo has compuesto. La melodía numero 3 puede ser completamente diferente en otro juego. Lo mismo ocurre con los datos del fichero de instrumentos. Hay ciertas dependencias entre el código del player de música y las direcciones donde se ensamblan los datos de instrumentos y las melodías.

Es muy sencillo, pero hay que comprender la estructura de la librería para hacerlo

Lo primero que debes tener claro es que tu juego se compone de 3 ficheros binarios ( o uno solo si decides compactarlo todo en un único fichero):

- Librería 8BP (es un fichero binario), incluyendo la tabla de atributos de sprites
- Fichero binario de música, con las melodías de tu juego
- Fichero binario de imágenes de sprites, incluyendo la tabla de secuencias de animación

Y dos ficheros BASIC:

- Cargador (carga la librería, música y sprites y por último tu juego)
- Programa BASIC (tu juego)

En este apartado vamos a centrarnos en como generar los 3 ficheros binarios que necesitas. Para generar los 3 ficheros primero debes ensamblarlo todo (ahora te diré como) y después que tengas todo ensamblado en memoria ejecutas estos comandos para generar los ficheros. Como ves estos comandos simplemente toman fragmentos de la memoria y la salvan en ficheros independientes.

```
SAVE "8BP.LIB", b, 24000, 8200  
SAVE "MUSIC.BIN", b, 32200, 1400  
SAVE "SPRITES.BIN", b, 33500, 8440  
SAVE "MAP.BIN", b, 42040, 500 :'
```

estos solo si tu juego usa mapa

Si decides compactar todo en un solo fichero (mas facil) puedes hacer una de estas dos opciones:

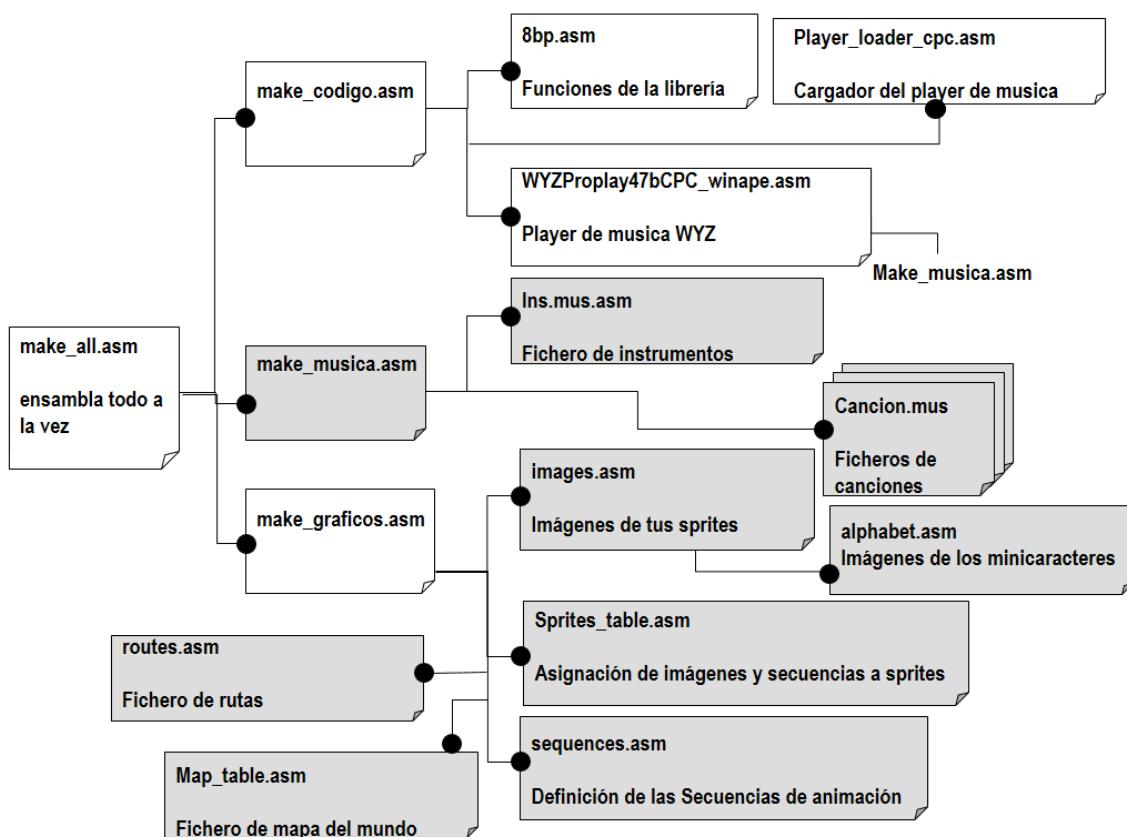
```
SAVE "MYGAME.BIN", b, 24000, 18540 (no incluye el banco de estrellas)  
SAVE "MYGAME.BIN", b, 24000, 18620 (incluye el banco de estrellas)
```

Ahora queda comprender como ensamblar todo en memoria (librería, música y gráficos). Para ello debes comprender la estructura de los ficheros .asm que debes manejar y sus dependencias. Un solo fichero .BIN en realidad va a requerir de más de un fichero .asm para generarlo.

El siguiente diagrama presenta todos los ficheros .asm de un juego que use 8BP así como las dependencias entre ellos.

En gris aparecen aquellos ficheros que tienes que editar, como son:

- las canciones y fichero de instrumentos, que generas con el WYZtracker
- el fichero make\_musica donde indicas que ficheros “.mus” hay que ensamblar
- el fichero de imágenes que creas con el SPEDIT
- la tabla de sprites donde asignas imágenes a los sprites (aunque no es estrictamente necesario pues tienes el comando |SETUPSP )
- la tabla de secuencias, donde defines que imágenes conforman una secuencia,
- el mapa del mundo: donde defines hasta 64 elementos que conforman el mundo
- el fichero de rutas: donde defines las trayectorias de los sprites que quieras
- el fichero alphabet.asm: si quieres crear un alfabeto diferente al de serie de 8BP



*Fig. 101 Ficheros para ensamblar*

Puedes ensamblarlo todo abriendo el fichero “make\_all.asm” y pulsando “assemble” en el menú de Winape. Después puedes usar el comando SAVE para salvar las imágenes, música y librería 8BP en diferentes ficheros binarios, o en uno solo, tal como hemos visto.

Si sólo cambias los gráficos puedes ensamblarlos por separado, seleccionando el fichero “make\_graficos.asm” y pulsando assemble.

Si cambias las músicas debes re-ensamblar el código de la librería pues hay una dependencia entre el código y las canciones, debido a que el código necesita conocer donde comienza cada canción. Por ello si cambias o añades canciones debes ensamblar con make\_all.asm

Puede que necesites ensamblar algo más, como un mapa de circuito de carreras que usa tus imágenes. En ese caso, añadelo al fichero “Make\_graficos.asm” para que se

ensamble después de images.asm. El orden de ensamblado es importante. Primero se deben ensamblar las imágenes, se le asocian etiquetas y después ya se pueden ensamblar los mapas o circuitos que usan dichas etiquetas.

## 17.1 Makeall.asm

Este es el fichero que permite ensamblar todo. Internamente invoca a tres ficheros que ensamblan en código de la librería y del player de música, las canciones y los gráficos.

```
; Makefile para los videojuegos que usan 8bits de poder
; si alteras una parte solo tienes que
; ensamblar el make correspondiente
; por ejemplo puedes ensamblar el make_graficos si cambias dibujos
;-----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 imágenes y secuencias de animación
; y la tabla de sprites inicializada con dichas imágenes y secuencias
read "make_graficos_mygame.asm"
```

Cada uno de esos tres ficheros se encarga de ensamblar cosas diferentes y por ejemplo el de los gráficos invoca a otros ficheros como son el de imágenes, el de secuencias, el de rutas y el mapa del mundo

## 17.2 Estructura del fichero de imágenes

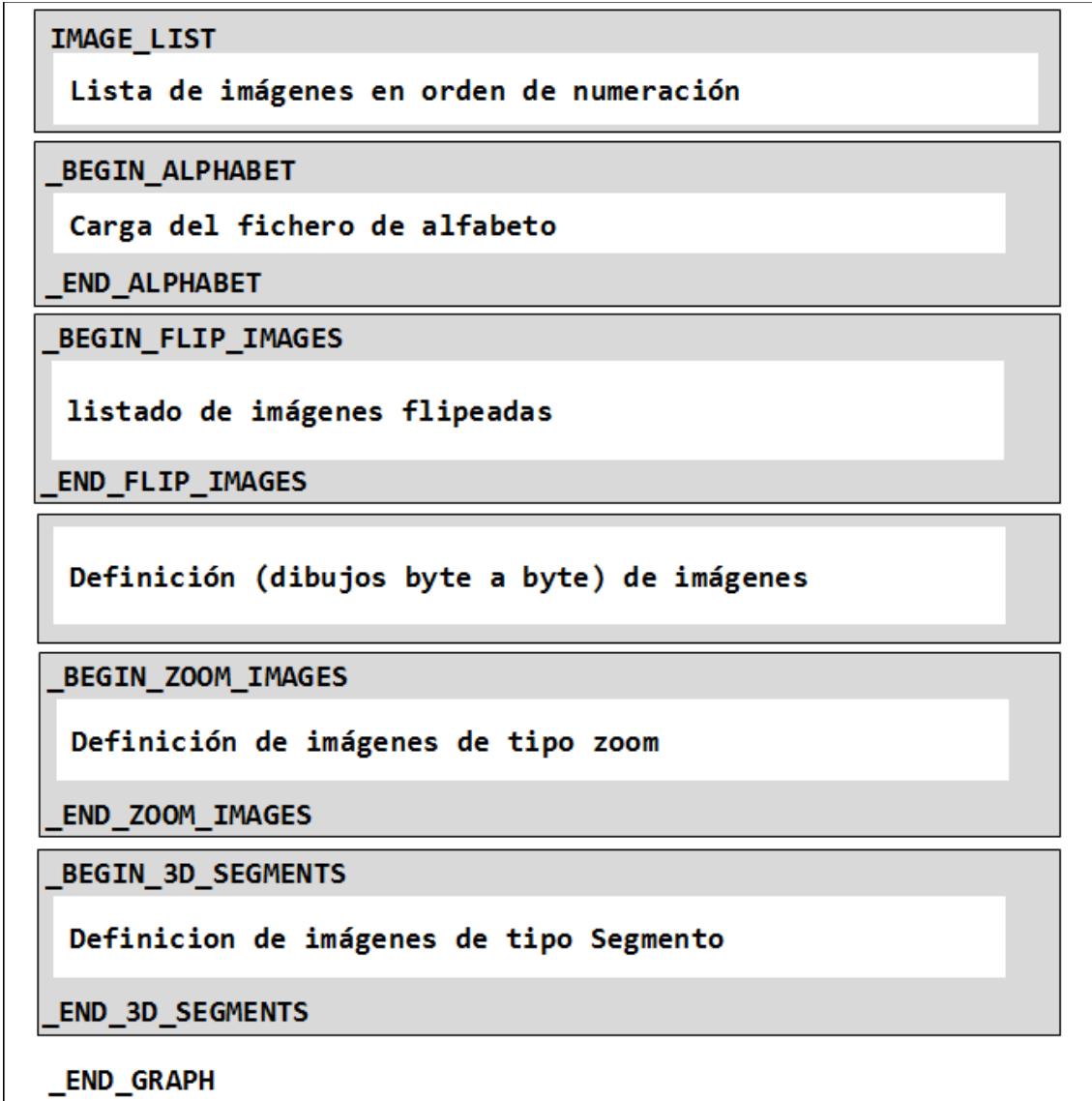


Fig. 102 estructura del fichero de imagenes

## 17.3 Estructura del fichero de secuencias de animación

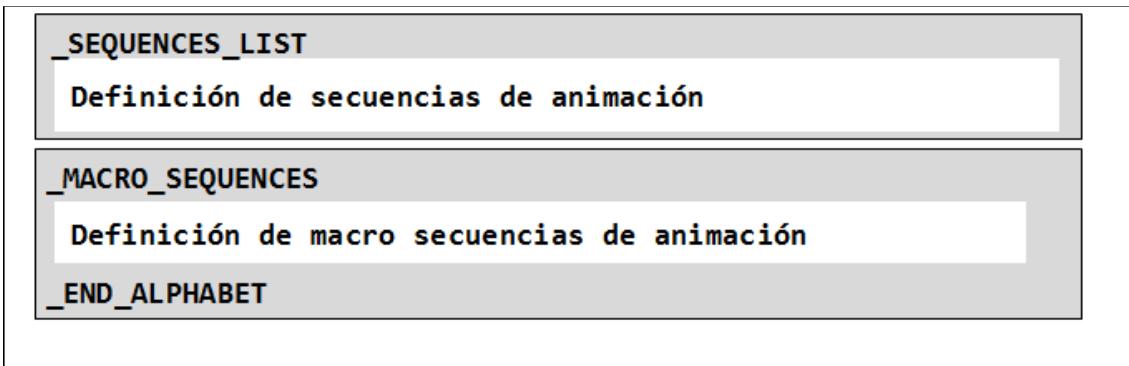


Fig. 103 estructura del fichero de secuencias de animacion

## 17.4 Estructura del fichero de rutas

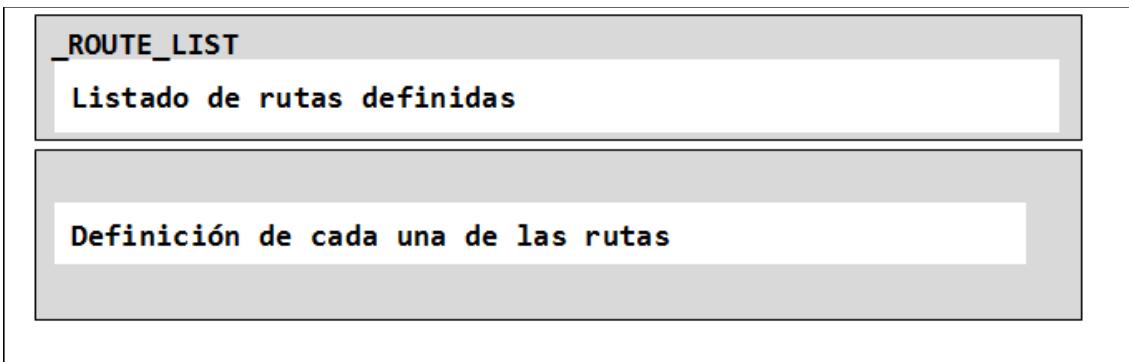


Fig. 104 estructura del fichero de rutas

## 17.5 Estructura del fichero de mapa del mundo

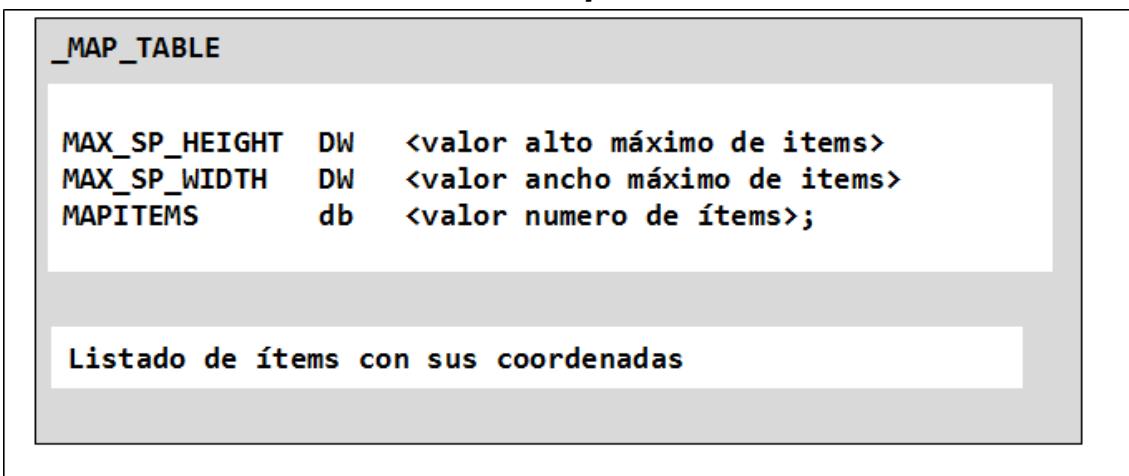


Fig. 105 estructura del fichero de mapa del mundo



## 18 Como hacer una tabla de puntuaciones

En muchos juegos es interesante disponer de un mecanismo de tabla de puntuación (también llamada “Hall of fame”) que almacene las mejores puntuaciones de forma ordenada de distintas partidas. Se puede programar en BASIC mediante un array que almacene la puntuación de cada partida, pero cada vez que paramos el juego y hacemos RUN, el interprete BASIC ejecuta internamente un CLEAR y se borran todos los valores que tuviese dicha tabla. Una forma de evitar esto es impedir que el usuario interrumpa el programa pulsando la tecla ESC dos veces mediante la rutina de firmware CALL &bb48. Otra opción es almacenar los puntos en una tabla en memoria y leerla y modificarla desde BASIC. Se puede programar de muchas formas, y aquí te propongo un ejemplo. Los pasos que hay que dar son:

Incluir en make\_all.asm una lectura del fichero “score\_table.asm”

```
;-----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
read "make_graficos_mygame.asm"
read "score_table.asm"
```

A continuación, debemos crear dicho fichero score\_table.asm con datos de ejemplo. Yo he creado uno con nombres de diosas sumerias y puntuaciones del 10 al 1. Cada nombre ocupa 8 caracteres. Algo muy importante es el org \_end\_graph. Con este comando estamos indicando que la tabla se va a ensamblar después de los graficos, en las direcciones de memoria que van a continuación.

```
org _end_graph
_SCORE_TABLE
db "ISHTAR    "
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
_END_SCORE_TABLE
```

Ahora viene la parte BASIC: Ensamblaremos con winape y después miraremos (con el winape, opción assemble->symbols) en que dirección de memoria se ha ensamblado la etiqueta end\_graph. En mi caso ha sido la &9685. En nuestro programa BASIC lo tendremos en cuenta (lo almacené en la variable "dir")

```

190 ' --- hall of fame
200 DIM pts(11): DIM name$(11):'scores
210 GOSUB 2040:'read score table
220 INK 3,7:PEN 3:LOCATE 15,12:PRINT " Hall of fame ":LOCATE
15,13:PRINT "-----"
230 p=1:FOR i=0 TO 9:LOCATE 1,i+14:PEN p :PRINT , name$(i), pts(i) :
p=1+(p MOD 3):NEXT
240 b$=INKEY$:IF b$="" THEN 240 ELSE 250

```

Veamos la rutina que lee la tabla de score en la línea 2040

```

2040 '--- READ SCORE TABLE
2050 dir=&9685: FOR i=0 TO 9: name$(i)=""
2070 FOR j=dir TO dir +7:'lee character a character las 8 letras
2080 letra=PEEK (j): name$(i)=name$(i)+CHR$(letra)
2090 NEXT j: dir=dir+8:'tras las 8 letras estan los puntos (un entero)
2100 pts(i)=0:|PEEK,dir,@pts(i):dir=dir+2:'un entero son 2 bytes
2110 NEXT i
2120 RETURN

```

Por último, cada vez que se acaba una partida, chequeamos si la puntuación (variable "score" en el ejemplo) es superior a algun registro de la tabla de scores (array "pts") y de ser así, modificamos dicha tabla. Al modificar la tabla en las direcciones de memoria, no perderemos los valores, aunque hagamos RUN.

```

1800 '--- FIN JUEGO & CHECK HIGH SCORE ---
1810 INK 0,0:BORDER 5: INK 2,15:INK 1,20:|MUSICOFF
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 'desplaza una posicion todas las puntuaciones inferiores
1840 FOR i=8 TO j STEP -1: pts(i+1)=pts(i): name$(i+1)=name$(i): NEXT
1850 b$=INKEY$:IF b$<>"" THEN 1850:'limpia buffer teclado
1860 MODE 1: BORDER 5: INK 3,8: LOCATE 6,8: PEN 3: PRINT
"CONGRATULATIONS! NEW HIGH SCORE"
1880 LOCATE 14,10:PEN 2:PRINT "ENTER YOUR NAME"
1900 LOCATE 15,12:PEN 1:INPUT name$(j): name$(j)=MID$(name$(j),1,8)
1910 pts(j)=score
1920 '--- WRITE SCORE TABLE ON MEMORY ---
1930 dir=&9685: FOR i=0 TO 9: k=1
1950 FOR j=dir TO dir +7:'escribimos carácter a carácter, las 8 letras
1960 dato$=MID$(name$(i),k,1): IF dato$="" THEN dato$=" "
1970 dato=ASC(dato$)
1980 POKE j,dato:k=k+1:NEXT j
1990 dir=dir+8: 'escribimos la puntuacion tras el nombre (8 letras)
2000 |POKE,dir,pts(i)
2010 dir=dir+2:'la puntuación es un numero entero = 2 bytes
2020 NEXT i
2030 RUN

```

## **19 Posibles mejoras futuras a la librería**

La librería 8BP es mejorable, añadiendo nuevas funciones que podrían abrir nuevas posibilidades para el programador. Aquí se muestran algunas sugerencias para hacerlo

### **19.1 Memoria para ubicar nuevas funciones**

Actualmente la librería ocupa 8200 Bytes y deja 24 KB para el BASIC. No voy a ampliar la memoria consumida por la librería porque dejaría menos espacio para el BASIC y eso reduciría tus posibilidades creativas. Los 24KB para BASIC no deben reducirse. Aún hay algo de espacio libre dentro de los 8200 Bytes, pero muy poco, apenas unos bytes para ligeras modificaciones. El player de música (“WYZ player”) forma parte de estos 8200 bytes y ocupa 1711 bytes. En caso de hacer un juego sin música se podrían ahorrar estos bytes y optar por otras nuevas capacidades, pero en ese caso tendríamos que elegir entre un pack de nuevas funciones y poder tener música.

### **19.2 Impresión a resolución de píxel**

Actualmente 8BP usa resolución de bytes y coordenadas de byte, que son 2 pixels de mode 0. En realidad, una forma de solventar esta limitación es mediante la definición de 2 imágenes para el mismo sprite que se encuentren desplazadas un solo pixel. Al mover dicho sprite por la pantalla puedes alternar entre simplemente cambiar la imagen por la desplazada y mover el sprite un byte. De ese modo conseguirás un movimiento pixel a pixel.

### **19.3 Layout de mode 1**

El layout actual funciona como un buffer de caracteres de  $20 \times 25 = 500$ Bytes

Se puede usar en juegos en mode 1 sin problemas, pero habrá cosas que no podamos hacer, como definir una pieza que ocupe 3 caracteres de ancho de mode 1, ya que los caracteres de mode 0 ocupan el doble de ancho de los de mode 1. No es un problema, pero sí es una limitación.

Un layout de mode 1 ocuparía 1KB pues  $40 \times 25 = 1000$ . Puesto que el layout de mode 0 y el de mode 1 no se usarían simultáneamente, podrían solaparse en memoria y teniendo en cuenta que el de mode 0 está en 42000 hasta 42500, simplemente el de mode 1 lo situaríamos entre 41500 y 42500, “robando” 500bytes a la memoria de sprites de 8KB, situada entre 34000 y 42000

Los cambios para soportar esta mejora son mínimos, afectando solo a dos funciones |LAYOUT y |COLAY que deberían ser conscientes del modo de pantalla, mediante una variable que actuase a modo de flag (layout0/layout1). Esta modificación podría estar bien pero aun sin tenerla, podemos usar layouts en juegos de mode 1 sin problemas.

### **19.4 Capacidad filmation**

Podría ser interesante crear un modo “filmation” para hacer juegos de tipo “knight lore”. Haciendo uso de funciones ya existentes en la librería, con un poquito de código extra se podría implementar esta interesante capacidad.

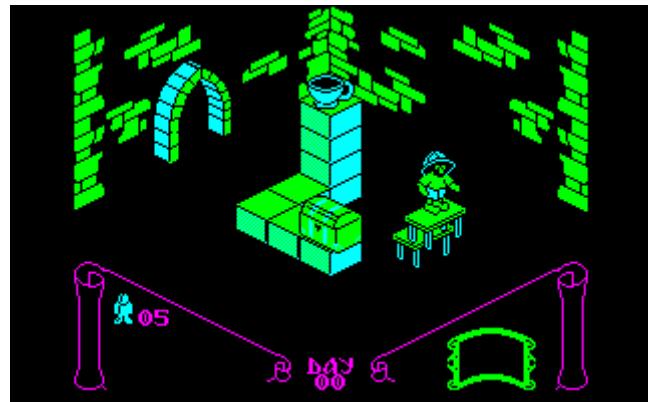


Fig. 106 el mítico “Knight Lore”

## 19.5 Funciones de scroll por hardware

Existen pocos juegos de Amstrad CPC con un scroll suave de calidad, programado usando las capacidades del chip controlador de video M6845. Actualmente la librería dispone de un mecanismo de scroll basado en CPU (no es por hardware pero es eficiente y versátil para juegos con scroll en cualquier dirección de movimiento).

El hecho de que no existan muchos juegos así responde al hecho de que en los años 80 los programadores de videojuegos no tenían mucha información y además en muchos casos eran aficionados

Entre los pocos juegos que tienen scroll suave destacan 2 de la casa Firebird:

- “Misión genocide” (de Firebird, 1987, por Paul Shirley, un excelente programador que además inventó una técnica de sobrescritura ultrarrápida sin uso de máscaras)
- “Warhawk” ( de Firebird, 1987)



Fig. 107 Juegos de Firebird con scroll rápido y suave

La técnica de scroll de estos dos juegos es la misma, conocida como “ruptura vertical”. La técnica de ruptura consiste en controlar exactamente el instante en el que se produce el barrido de pantalla. En ese momento engañamos al CTRC 6845 diciéndole que la pantalla termina antes de lo normal. Eso sí, antes de terminar esa sección de la pantalla, le decimos que incorpore menos scanlines de las que corresponden a una sección de ese tamaño. A continuación, y en un instante muy preciso que debemos controlar al microsegundo, le decimos al chip que comience una nueva pantalla, sin haberse

producido la señal de sincronismo vertical. Eso nos permite dibujar una segunda zona de pantalla (los marcadores, por ejemplo) y compensamos el número de scanlines de la primera sección. Si hacemos bien el mecanismo de compensación de número de scanlines, podemos hacer que una de las dos secciones de pantalla se mueva con extraordinaria suavidad. El problema de llevar esta técnica a un comando para ser usado desde BASIC es que el control de las interrupciones es impreciso debido a la ejecución del intérprete, y aquí necesitamos un control muy muy preciso.

El problema del scroll por hardware (que afecta también a un scroll por software que mueva toda la pantalla) es que “arrastra” los sprites presentes con él, de modo que al recolocarlos notaremos una vibración indeseada en los enemigos y/o en nuestro personaje. Para solventarlo se puede usar doble buffer y conmutar entre dos bloques de 16KB cada vez que un fotograma esté listo. Eso evitará que se pueda ver “cómo se hace” cada fotograma. En 8BP el doblebuffer lo descarté para poder dejar un buen espacio de RAM al programador y por eso estas técnicas no han sido implementadas.

Por todos los motivos expuestos, el scroll en 8BP se basa en un mapa del mundo que al moverse no arrastra a los sprites y por lo tanto es más eficiente pues mueve menos memoria y a la vez permite movimiento multidireccional.

## **19.6 Migrar la librería 8BP a otros microordenadores**

Esta librería sería fácilmente portable a otros microordenadores basados en el Z80, como el Sinclair ZX Spectrum. En el caso del ZX spectrum habría que rescribir las rutinas que pintan en pantalla pues la memoria de video se maneja de modo diferente. La migración a ZX ya es un proyecto en firme, tras haber recibido numerosas peticiones de usuarios de ZX.

La migración de la librería a un Commodore 64 también sería factible, aunque no se podría reutilizar el código ensamblador, ya que está basado en otro microprocesador. Además, en el caso del commodore 64, la migración de la librería 8BP debería aprovechar las características propias de la máquina como sus 8 sprites hardware, de modo que lo que debería incorporar internamente la librería 8BP sería un sprite multiplexer, ofreciendo 32 sprites, pero internamente usando los 8 sprites hardware.



*Fig. 108 Sinclair ZX y Commodore 64, dos clásicos*



## 20 Algunos juegos hechos con 8BP

En este capítulo voy a describir como están hechos algunos juegos que puedes encontrar en la web <https://github.com/jjaranda13/8BP> realizados con 8BP

- **Happy Monty:** velocísimo juego del estilo mutant monty
- **Eridu:** un juego del estilo del clásico “scramble” con scroll horizontal
- **Space phantom:** inspirado en space harrier
- **Frogger eterno:** un remake del clásico frogger, presentado en la famosa feria “amstrad eterno”
- **3D Racing one:** primer juego de carreras que usa la capacidad pseudo 3D
- **Fresh Fruits & vegetables:** un juego de plataformas que usa scroll horizontal y gestión avanzada de rutas de sprites
- **Nibiru:** un juego de naves de scroll horizontal, que usa características avanzadas de 8BP
- **Anunnaki:** un juego de naves, género arcade
- **Mutante Montoya:** un juego de pasar pantallas. Podría encuadrarse en género plataformas. Fue el primer juego que hice con 8BP.
- **Minijuegos:** son juegos didácticos, sencillos y cortos, para iniciarse en la programación con 8BP. Hay disponible una versión del clásico “pong”, llamado “Mini-pong” y una versión del clásico “Space Invaders”, llamado “mini-invaders”

### 20.1 Mutante Montoya

Un primer tributo al Amstrad CPC, con un título inspirado en el clásico "mutant monty"

Es un juego sencillo de 5 niveles. Se basa en el uso del layout de 8BP para construir cada pantalla.



## **20.2 Anunnaki, nuestro pasado alien**

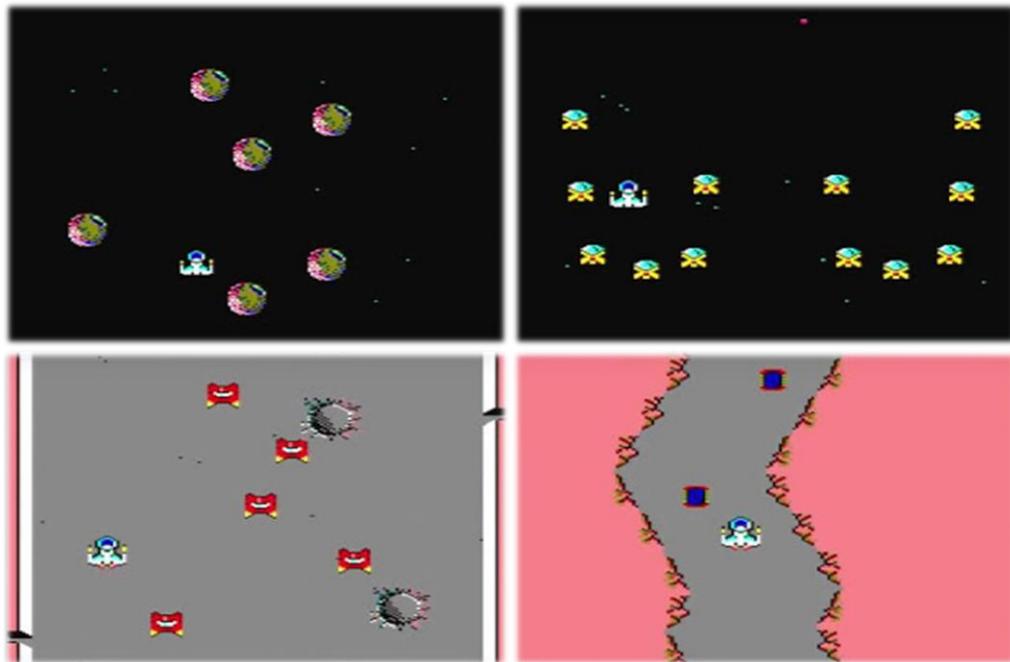
Este es un videojuego de arcade muy interesante para analizar y adentrarse en la técnica de programación de "lógicas masivas". Cuando fue programado, la librería 8BP aun no disponía de comando de scroll ni de enrutamientos de sprites, de ahí que la programación de este juego sea tan interesante, pues mediante lógicas masivas lo logra todo.

A diferencia del "Mutante Montoya", el videojuego "Anunnaki" no hace uso del layout, ya que se trata de un juego donde se trata de avanzar y destruir naves enemigas, no es un juego de laberintos ni de pasar pantallas. Este juego, además hace uso de técnicas de scroll "simulado", muy interesantes.

Eres Enki, un comandante anunnaki que se enfrenta a razas alienígenas para conquistar el planeta tierra y así someter a los humanos a su voluntad.

El juego consta de 2 niveles, aunque si pierdes una vida, continuas en el punto del nivel en que te encuentres, no vuelves al principio del nivel.

El primer nivel es una fase en el espacio interestelar, donde debes esquivar meteoritos y matar hordas de naves y pajarracos espaciales. Al final del nivel debes destruir un "jefe". El segundo nivel se desarrolla en la Luna, donde debes destruir hordas de naves, tras lo cual debes atravesar un túnel plagado de minas hasta encontrarte con tres "jefes" a los que debes destruir.



## **20.3 Nibiru**

Este es un juego que pone a prueba muchas de las características de 8BP y de la técnica de programación de "lógicas masivas", y tiene detalles como un gráfico de carga lujoso y tres melodías durante el juego, así como una tabla de scores que no se pierde, aunque reinicies el juego y otros aspectos técnicos avanzados como scroll parallax, rutas, macrosecuencias, etc. El listado BASIC ocupa poco mas de 16KB.

Eres el piloto de una nave destructora y debes vencer al planeta Nibiru y a su líder, "Gorgo", un reptil milenario casi invencible. Debes destruir a los pájaros galácticos que viven en sus lunas y una vez que llegues al planeta te debes enfrentar a sus peligros antes de poder luchar con Gorgo.

El Juego consiste en tres fases y utiliza el mecanismo de scroll de 8BP basado en el comando MAP2SP y también usa enrutamiento de sprites y macrosecuencias de animación. ¡Todo desde BASIC! gracias a 8BP y a la técnica de "lógicas masivas"



El juego mantiene una tabla de scores que no se borra, aunque pares el juego. Esto se consigue almacenándola en RAM con pokes, en lugar de almacenarla en variables BASIC.

## 20.4 Fresh Fruits & vegetables

Este es un juego de plataformas, donde tu misión consiste en recoger todas las frutas para dejar a la población sin nada que comer, de modo que tengan que sacrificar a un pobre cerdo para alimentarse.

Su mayor novedad es el uso avanzado de rutas, concatenando unas a otras para pasar de "caer" a "andar" y el uso de RINK combinado con MAP2SP como técnica de scroll



Los ladrillos azules de la presentación del juego usan impresión de sprites sincronizada (PRINTSPALL,0,0,1) mientras que durante el juego la impresión de sprites no es sincronizada (PRINTSPALL,0,0,0). El efecto de la sincronización en la presentación es que todo aparece sincronizado, incluida la animación por tintas RINK (usada en los ladrillos azules). Esto proporciona un efecto de scroll suave. En mi opinión, no se debe sincronizar un juego porque, aunque consigues mayor suavidad, también la velocidad del juego disminuye. Dependiendo del tipo de juego te puede interesar o no.

## 20.5 “3D Racing one”

Este es el primer juego realizado usando la capacidad pseudo3D de 8BP. Posee un lujoso gráfico de carga y 4 circuitos: uno de entrenamiento con charcos en la carretera, otro en el que competimos con otros 4 coches, otro donde la velocidad es el doble, usando segmentos de menor inclinación, y una ultima fase nocturna. El juego usa también el comando PRINTAT y un juego de caracteres propio. Además, posee tabla de scores, una música principal, dos secundarias y efectos de sonido.

Para la presentación se usa un mapa 2D lleno de bolas y se configura el comando MAP2SP con sobreescritura. Las bolas son “zoom images”.

El primer circuito se ha construido con un fichero en el que hemos ubicado uno por uno todos los elemtos (segmentos, carteles, arboles, charcos...) pero el resto de los circuitos se crean dinámicamente desde BASIC, pokeando la dirección del mapa del mundo.

Para detectar las colisiones y la salida de carretera, se usa el comando PEEK en lugar de COLSPALL, de este modo en cuanto detecta un byte sobre el coche de otro color que no sea la carretera, se considera que estás colisionando y tu motor se daña.

Otra novedad interesante es el uso de rutas dinamicas. Tanto el circuito de competición como las rutas de los coches para recorrerlo se crean desde BASIC, siguiendo el procedimiento explicado en este manual.



## 20.6 Space Phantom

Este es un juego realizado con la versión v35 de la librería. Usa las capacidades pseudo-3D para la presentación de títulos al estilo “Star Wars”. Usa también impresión transparente con sprites (monedas) que pasan detrás del fondo (la tabla de scores).

El juego está inspirado en el clásico “Space Harrier” y manejas a un héroe equipado con un jet-pack que vuela por el espacio, matando meteoritos, ovnis, pájaros espaciales y un dragón como enemigo final. Se compone de tres fases y un final épico.

El juego de caracteres es propio, aunque sólo se han definido los números, al estilo “reloj casio” para los marcadores.

En la primera fase se usan rutas para las estrellas, que son sprites al igual que los meteoritos y los pájaros.

En la segunda se usa sobreescritura de sprites con fondo de 2 bit (4 colores) y animación por tintas usando RINK. Las naves en hilera se construyen ubicándolas mediante el uso del comando mejorado ROUTESP, disponible en la V35.

Aunque el juego simula 3 dimensiones, no se usa la proyección pseudo-3D, sino rutas de sprites en los que se va cambiando la versión del enemigo por una de mayor tamaño para dar sensación de que se acerca. Para que una colisión con un enemigo lejano no nos afecte, se utiliza un flag no usado registro de estado para marcar los enemigos lejanos como inofensivos.

En la tercera fase se ha usado movimiento relativo horizontal de las piedras del suelo, en combinación con una ruta de movimiento vertical acelerado.



## 20.7 Frogger Eterno

El juego “Frogger Eterno” ha sido realizado con la versión V36 de 8BP, y su título evoca tanto el clásico “frogger” de konami lanzado en 1981 como la feria “Amstrad Eterno” celebrada en 2019, evento en el que este juego hizo su aparición.

Es un juego programado en mode 1, que usa LAYOUT, impresión transparente en la rana y rutas de diversa naturaleza para los sprites. Algunos de los sprites son invisibles, pero colisionables, tales como 4 ríos “invisibles” que se encuentran bajo los troncos, nenúfares y tortugas sobre los que la rana debe saltar, como “muros invisibles” a los laterales del río, para que la rana no pueda escapar.



## 20.8 Eridu: The space port

Este juego es un clone del clásico "Scramble" de Konami creado en 1981. Realmente tiene muchas diferencias con el original, pero en esencia está inspirado en el juego clásico, ya que incorpora la necesidad de recargar combustible, forzando al jugador a arrinconarse para destruir los tanques de combustible y así no perder una vida.

Es bastante rápido a pesar de funcionar con un scroll potente, mostrando 32 sprites en pantalla en muchos momentos. Llega a alcanzar los 18 fps

El juego tiene 5 fases, y diferentes músicas on-game, además de un gráfico de presentación muy lujoso.

Eridu es un videojuego que conecta con una antigua historia "prohibida" de la humanidad. Eridú fue la primera ciudad del mundo, creada por los "Anunnaki" hace 400.000 años, una raza extraterrestre según las tablillas sumerias encontradas en el desierto de Irak. Allí establecieron un puerto espacial llamado "Tierra 1".



Los mapas de las diferentes fases se cargan en diferentes bancos de memoria ocupando cada uno 500 bytes de datos del mundo y 200 bytes para la descripción de la ubicación de los enemigos. El scroll lógicamente está hecho con el comando MAP2SP

## 20.9 Happy Monty

Es un velocísimo juego de pasar pantallas, que imita casi a la perfección el clásico mutant Monty. Incluso "clona" su pantalla inicial. Este juego está hecho con la versión 37 de la librería y alcanza los 25 FPS.

Hace un uso intensivo del layout y por supuesto, de lógicas masivas. Consigue almacenar 25 niveles mediante una sencilla técnica para compactar los layouts (cada layout gasta solo 160 bytes) explicada en este libro.

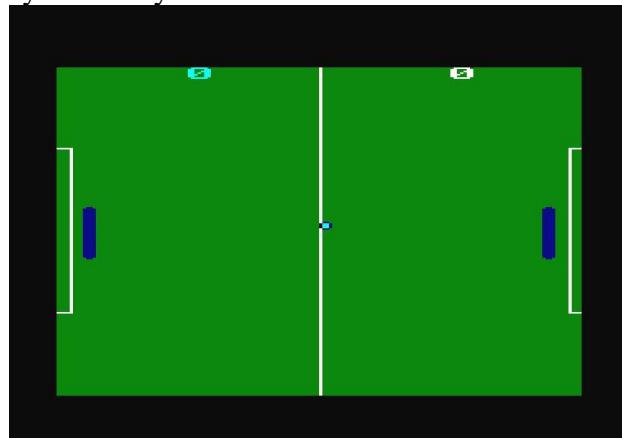


## 20.10 Mini Juegos

Son juegos realizados con fines didácticos. Sencillos de entender y cortos, para ayudar en la elaboración de juegos propios al programador novel.

### 20.10.1 Mini-pong

Es un videojuego muy sencillo y didáctico. Basado en el clásico "Pong" de Atari (1972)



*Fig. 109 Videojuego "mini-pong"*

La barra del oponente (el ordenador) comienza a tomar decisiones cuando la pelota sobrepasa la mitad del campo, por lo que es posible ganarle. Si le ponemos a tomar decisiones antes llega un momento que es imposible ganar.

Algunas pinceladas sobre el juego:

- Uso de |COLSPALL: para detectar colisiones entre el "colisionador" (la pelota) y los "colisionados" (las barras). En el byte de estado de los sprites se activa el flag de colisionador para la pelota (sprite 29) y se activa el flag de colisionado en el 31 (nuestra barra) y el 30 (barra del oponente)
- Usa sobreescritura en la pelota para respetar la raya blanca del campo y no borrarla al pasar. Para ello se usa una paleta con sobreescritura y se activa el flag de sobreescritura en el byte de estado del sprite de la pelota (el 29)
- sólo hay dos imágenes (la pelota=17 y la barra=16) que se asignan a los 2 sprites (a los sprites 30 y 31 se les asigna la imagen 16 y al sprite 29 se le asigna la 17)
- Los sprites usan movimiento automático. Para ello tienen activado el flag de movimiento automático y la instrucción AUTOALL los mueve (les cambia sus coordenadas) de acuerdo con su velocidad.

- Todos los sprites se imprimen con PRINTSPALL en cada ciclo de juego

### 20.10.2 Mini-Invaders

Al igual que el “Mini-pong”, este es un juego realizado con fines didácticos, inspirado en el clásico “Space Invaders” de Taito (1978)



*Fig. 110 Videojuego “mini-invaders”*

Algunas pinceladas sobre cómo está realizado:

- El juego usa 32 sprites
- La nave es el sprite 31
- Los disparos que puedes lanzar con la nave son el 29 y el 30
- Los invaders disparan usando el sprite 28
- Los invaders usan los sprites del 0 al 27 (28 invaders en total)
- Los sprites 31,30 y 29 tienen flag de colisionador activo
- El resto de sprites son "colisionados" y tienen flag de colisionado activo
- Los invasores tienen flag de movimiento automático activo y se les asocia la ruta "0" que los mueve de derecha a izquierda y hacia abajo, lo típico de los invasores
- Los disparos de la nave y de los invaders usan una característica de la V27, recorren la pantalla y al salir se desactivan automáticamente con un cambio de estado definido al final de su ruta, simplificando de este modo la lógica de BASIC y por consiguiente acelerando el juego.



## 21 APENDICE I: Organización de la memoria de video

### 21.1 El ojo humano y la resolución del CPC

La memoria de video del Amstrad CPC tiene 3 modos de funcionamiento. El modo más utilizado para los juegos es el mode 0 (160x200) por disponer de más color, pero también se ha usado bastante el mode 1 (320x200) para programar juegos. El mode 2 (640 x 200) apenas o nunca se usó para juegos debido a sus escasos 2 colores.

Puesto que la cantidad de memoria de video es la misma, se sacrifica resolución para ganar en cantidad de colores, pero curiosamente en horizontal, que es el lado de la pantalla más largo, hay menos resolución que en vertical (160 en horizontal y 200 en vertical). Te preguntarás por qué. Además, no es el único microordenador que hacía eso, muchos otros ordenadores también usaban la misma estrategia con el lado horizontal

El motivo tiene que ver con el funcionamiento del sistema visual humano. El ojo percibe mas detalles en vertical que en horizontal, de modo que “dañar” la resolución en el eje horizontal no es tan grave como dañarla en vertical. Subjetivamente es más aceptable el resultado. El sistema visual humano es como el mode 0, pixels muy anchos. Por eso el campo de visión horizontal es mayor que el vertical

### 21.2 La memoria de video

La información más completa y clara se encuentra en el manual del firmware del Amstrad. Esta información te será útil si quieres construir un editor de sprites mejorado o quieres adentrarte en el ensamblador y programar rutinas de impresión con sobreescritura o cualquier otra cosa.

#### 21.2.1 Mode 2

En mode 2, cada píxel está representado por un bit. De modo que un byte representa a 8 pixels. Si tomamos cualquier byte de la memoria de video, su correspondencia con los pixeles es de 1 bit por cada píxel, en esta tabla se representan los bits y a que pixeles (p) corresponden

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
p0	p1	p2	p3	p4	p5	p6	p7

En un byte se numera el bit 7 como el más situado a la izquierda. El píxel 0, es precisamente también el pixel situado más a la izquierda, es decir, aquí no hay nada “al revés”. Está todo correcto

#### 21.2.2 Mode 1

En mode 1 tenemos 4 colores (representados por 2 bits). Un byte representa por lo tanto a 4 pixeles. La correspondencia entre pixeles y bits es algo más compleja. El pixel 0 por ejemplo se codifica con los bits 7 y 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)

### 21.2.3 Mode 0

Aquí tenemos un pequeño follón. Cada byte representa solo dos pixels, de los cuales la correspondencia con los bits del byte es la siguiente: El pixel 0 se codifica con los bits 7,5,3 y 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)

Con la siguiente imagen seguro que te queda más claro:

Byte (lo que se imprime en las direcciones de pantalla)

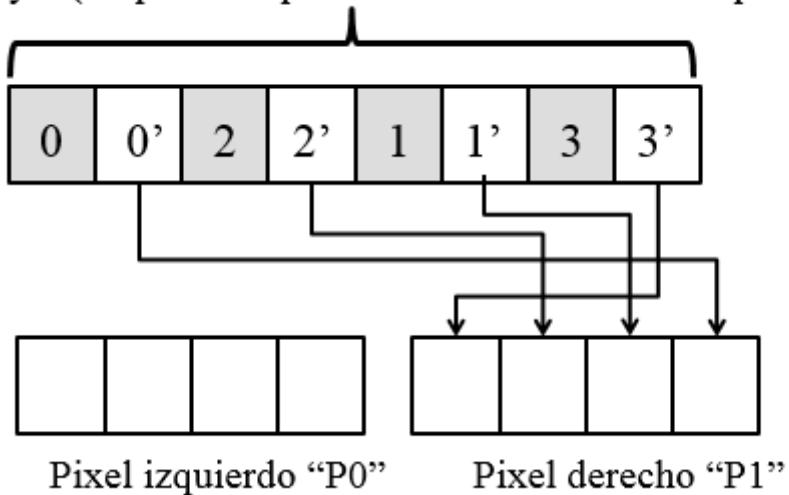


Fig. 111 pixels y bits en mode 0

No te puedo decir cuál es el oscuro motivo para haber organizado así la memoria, pero me imagino que la causa se encuentra en el GATE ARRAY, el chip que traduce estos bits a señal de video. Imagino que el diseñador redujo circuitería con este retorcido diseño.

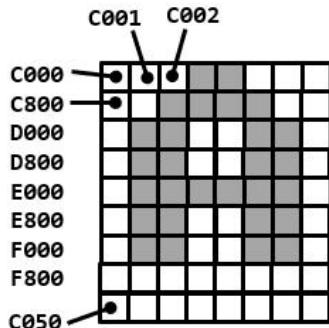
### 21.2.4 Memoria de la pantalla

Los pixels de la pantalla pertenecientes a una misma línea se encuentran codificados en los bytes que también son contiguos. Sin embargo, de una línea a otra hay saltos.

Si avanzamos en direcciones de memoria, al llegar al final de una línea saltamos a una línea que se encuentra 8 líneas más abajo. Y si queremos continuar en la línea siguiente lo que tenemos que saltar en direcciones de memoria son 2048 posiciones.

En la siguiente tabla está representada la memoria de video. En la izquierda aparece la línea de caracteres (de 1 a 25) y para cada línea, la dirección de comienzo de cada una de las 8 scanlines que la componen (denominadas como 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	CFD0	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. 112 Mapa de memoria de pantalla**

La pantalla del Amstrad tiene 200 líneas x 80 bytes de ancho cada una, por lo que la memoria que se muestra en pantalla es  $200 \times 80 = 16.000$  bytes. Sin embargo, la memoria de video son 16384 bytes. Hay 384 bytes “escondidos” en 8 segmentos de 48 bytes cada uno, los cuales no se muestran en pantalla, aunque formen parte de la memoria de video. Estos 8 segmentos son lo que en la tabla anterior se llaman “spare”. Cada segmento mide 48 bytes porque como he dicho antes para saltar de una línea a la línea siguiente hay que sumar 2048 pero en realidad las 25 líneas de memoria contigua que las separan tan sólo ocupan  $25 \times 80$  bytes =2000 bytes.

Desde la &C7D0 hasta la C7FF ambos inclusive
Desde la &CFD0 hasta la CFFF ambos inclusive
Desde la &D7D0 hasta la D7FF ambos inclusive
Desde la &DFD0 hasta la DFFF ambos inclusive
Desde la &E7D0 hasta la E7FF ambos inclusive
Desde la &EF00 hasta la EFFF ambos inclusive
Desde la &F7D0 hasta la F7FF ambos inclusive
Desde la &FFD0 hasta la FFFF ambos inclusive

Puedes comprobarlo haciendo POKE sobre esas direcciones de memoria, y verás que no alterarás el contenido de la pantalla.

Resulta tentador pensar en usar estas zonas “escondidas” de la memoria para almacenar pequeñas rutinas en ensamblador o variables. Sin embargo, es peligroso porque un

comando MODE ejecutado desde BASIC borra completamente esos segmentos de memoria, por lo que si lo usas debes ser consciente de ello. En la librería 8BP estos segmentos se usan para almacenar variables locales de algunas funciones, cuyo valor puede ser borrado sin riesgos.

### **21.3 Cálculo de una dirección de pantalla**

Si quieres saber la dirección de memoria a la que se corresponden unas coordenadas concretas, deberás realizar la siguiente operación

$$\text{Dir} = \&C000 + \text{INT}(y/8)*80 + (y \bmod 8)*2048 + x$$

Esto es muy útil si por ejemplo quieras usar PEEK para averiguar si hay un determinado elemento o color en un byte concreto de la pantalla y utilizarlo como mecanismo de detección de colisión. Esta técnica se usa en el videojuego “3D-Racing One”.

### **21.4 Barridos de pantalla**

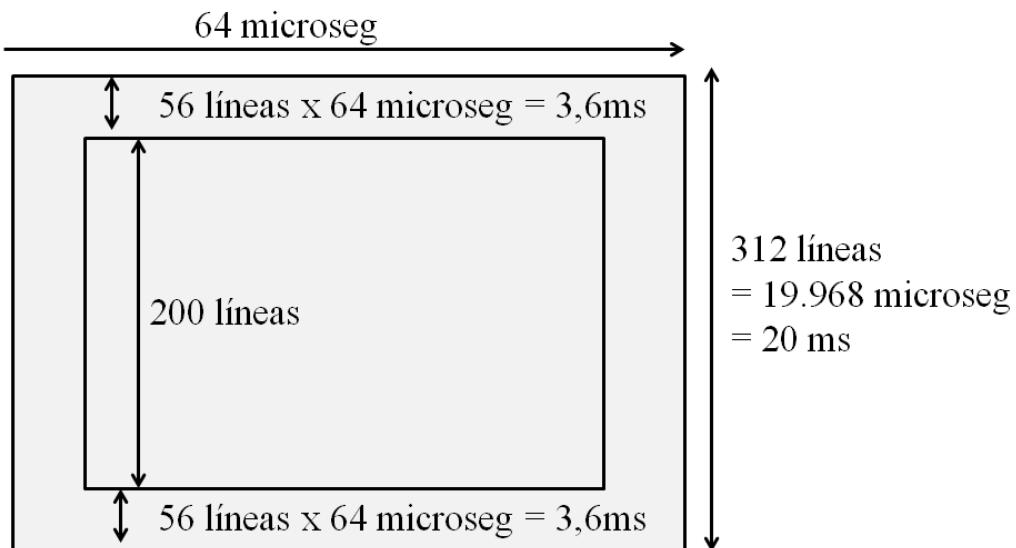
El Amstrad genera 50 imágenes por segundo. Eso significa que cada 20ms aproximadamente se debe de producir un nuevo barrido de pantalla.

Podríamos pensar que quizás el barrido, lo que es pintar la pantalla, consume una fracción de esos 20 ms pero no es así. Pintar la pantalla le lleva al Amstrad los 20ms completos, de modo que, aunque sincronices tu impresión de sprites con el barrido de pantalla es muy probable que te pille, dando lugar a dos conocidos efectos:

- **Flickering:** (parpadeo), ocurre cuando borras el sprite antes de imprimirlo en su nueva posición. Para evitarlo hay una solución muy simple: no lo borres. Simplemente haz que el sprite vaya borrando su propio rastro, dejando un borde en el sprite para que cumplan esa función. El sprite es más grande pero no parpadeará, aunque le pille el barrido a la mitad, pues no desaparece
- **Tearing:** (quiebro): ocurre cuando nos pilla el barrido a la mitad del sprite. La mitad se imprime con la nueva posición (la cabeza y el tronco) y la otra mitad no da tiempo (las piernas). Entonces el sprite queda impreso “mal”, aunque es corregido en el siguiente fotograma, pero por un instante es como si se deformase o quebrase. El tearing es un efecto malo, pero mucho más aceptable que el flickering. La solución perfecta implica controlar en cada milisegundo donde se encuentra el barrido para conseguir imprimir cada sprite sin que nos alcance.

Una típica recomendación es imprimir los sprites de abajo a arriba, para minimizar estos efectos. De ese modo solo es posible que el barrido te pille una vez en uno de los sprites, mientras que imprimiendo de arriba abajo, te puede pillar en varios sprites el barrido por que ambos (CPU y rayo catódico) trabajan en la misma dirección. Lamentablemente lo más interesante es ordenar de arriba abajo para dar efecto de profundidad en los sprites (útil en determinados juegos tipo “Golden axe”, “Double dragon”, “Renegade”, etc.)

Los tiempos que consume la pantalla son los siguientes. Fíjate que desde que se produce la interrupción de barrido, dispones de 3,5ms para pintar sin que sea posible que te pille. Pero en ese tiempo podrás imprimir como mucho 2 sprites pequeños.



*Fig. 113 tiempos en un barrido de pantalla*

## 21.5 Cómo hacer una pantalla de carga para tu juego

Hay muchas formas de hacerla. Una muy sencilla es construir un gráfico con el programa spedit modificado para que te permita pintar en toda la pantalla sin mostrarte menús y al final pulsar alguna tecla que te lance un comando SAVE como este

```
SAVE "mipantalla.bin", b, &C000, 16384
```

Como ves el comando salva 16KB desde la dirección de comienzo de la pantalla, que es la &C000

La forma de cargarlo sería

```
LOAD "mipantalla.bin", &C000
```

Si no usas la paleta por defecto, entonces previamente debes de ejecutar los comandos INK correspondientes a la paleta que hayas usado, antes de cargar la pantalla. Al cargar la pantalla verás poco a poco como se va dibujando en pantalla mientras carga, puesto que es ahí precisamente donde lo estas cargando, en la memoria de video.

Otra forma es generar un layout bien trabajado y salvarlo mediante el comando SAVE anterior. El caso es hacer un dibujo y como ves hay muchas formas.

Por último, puedes usar una herramienta como ConvImgCPC (también hay otras), un conversor/editor de imágenes que funciona bajo windows. Esta herramienta te permite transformar una imagen cualquiera (que puede ser un escaneo de un dibujo tuyo) en un fichero binario (con extensión .scr) apto para el CPC. Esta herramienta además te permite editar pixel a pixel y retocarla hasta que te quede perfecta. Incluso puedes editarla desde cero, sin escanear nada. En mi opinión esta es la mejor opción.

Para meter este fichero en un disco (en un fichero .dsk) debes usar el CPCDiskXP que es otra herramienta que te permite meter ficheros dentro de ficheros .dsk

Una vez dentro del .dsk puedes cargarlo con LOAD “mipantalla.bin”,&C000 Sin embargo, los colores no se verán bien porque ConvImgCPC ajusta la paleta para aproximarse lo más posible a los colores originales. Para verlos bien debes ejecutar la rutina donde ConvImg coloca la rutina de cambio de paleta, que es CALL &C7D0. Esta rutina esta “escondida” en el primero de los 8 segmentos ocultos de la memoria de video, por lo que la imagen .scr no ocupa más por contener dicha rutina. Lo malo es que hasta que no cargues la pantalla no puedes ejecutarla y por lo tanto verás como carga la imagen con colores erróneos y al final podrás invocar ese call, cambiando los colores. Lo que puedes hacer es preparar un archivo especial de paleta. Haciendo esto:

```
Load "imagen.scr", &c000
Save "paleta.bin", b, &c7d0, 48, &c7d0
```

Ahora tienes un archivo de 48 bytes que contiene la paleta. En el cargador de tu juego harías esto:

```
Load "!paleta.bin"
Call &c7d0
Load "!imagen.scr", &c000
```

Yo te recomiendo que simplemente coloques unos cuantos commandos INK antes del comando Load “imagen.scr”, &c000

Para dejar la paleta en sus valores por defecto, usa CALL &BC02, que es una rutina del firmware.

#### **Y para salvar la pantalla en una cinta?**

Hemos visto como hacerlo en un disco pero CPCDiskXP no nos va a dejar el fichero .scr dentro de la cinta, de modo que debemos hacer algo como:

```
|DISK
MEMORY 15999
LOAD "imagen.scr", 16000
|TAPE
SPEED WRITE 1
SAVE "imagen.scr",b,16000,16384
```

Y al cargarla, lo haremos igual que en disco:

```
Load "!imagen.scr", &c000
```

## 22 APENDICE II: La Paleta

En la siguiente tabla aparece la paleta del AMSTRAD. Dentro de cada color y entre paréntesis figura el numero de tinta que tiene asignado ese color en la paleta por defecto.

Los 27 colores son:

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)

Los valores de la paleta por defecto en cada modo son:

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

Los valores de la paleta en cada modo se gestionan con el comando INK, consulta el manual de referencia BASIC del Amstrad para más información. Por ejemplo, para configurar la tinta cero como rojo, consultamos la paleta de los 27, vemos que el rojo es el sexto color y escribimos

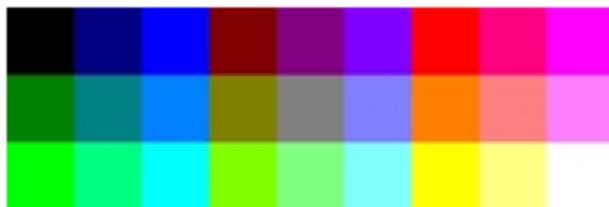
INK 0,6

Y ya tenemos configurada la tinta cero para que sea rojo. Como ves, una tinta no es un color concreto, sino que puede ser configurada para que sea el color que quieras.

El motivo por el que la paleta del Amstrad es tan buena, es porque los 27 colores ofrecen muchas posibilidades, aunque solo se puedan usar 16 a la vez. Los colores están ordenados por brillo.

Si representamos en escala RGB de 24bit (8 bit por componente) los colores del Amstrad, encontramos que dispone de 3 valores de rojo, 3 de verde y 3 de azul, (dichos

valores son 0,127 y 255) siendo el numero de combinaciones  $3 \times 3 \times 3 = 27$ . EL color gris está justo en el centro de la paleta, donde los valores de R,G,B son iguales ( $R=127, G=127, B=127$ ). Tambien son las 3 componentes iguales en el blanco ( $R=255, G=255, B=255$ ) y el negro ( $R=0, G=0, B=0$ ).



*Fig. 114 Paleta de Amstrad*

El hecho de tener una paleta de 27 permite que, aunque sólo podamos escoger 16, haya siempre colores para elegir, que nos permitan crear difuminados y mezclas. Sin embargo, otros ordenadores de la época como el C64 (gran máquina) tenia 16 colores de una paleta de 16. EL C64 poseía 3 tonos de gris dentro de tan reducida paleta, lo cual, aunque ha sido criticado, opino que no es malo pues al ser colores poco saturados combinan bien con el gris, y también combinan bien entre ellos, es decir, que son 16 colores “cercaños” entre si.

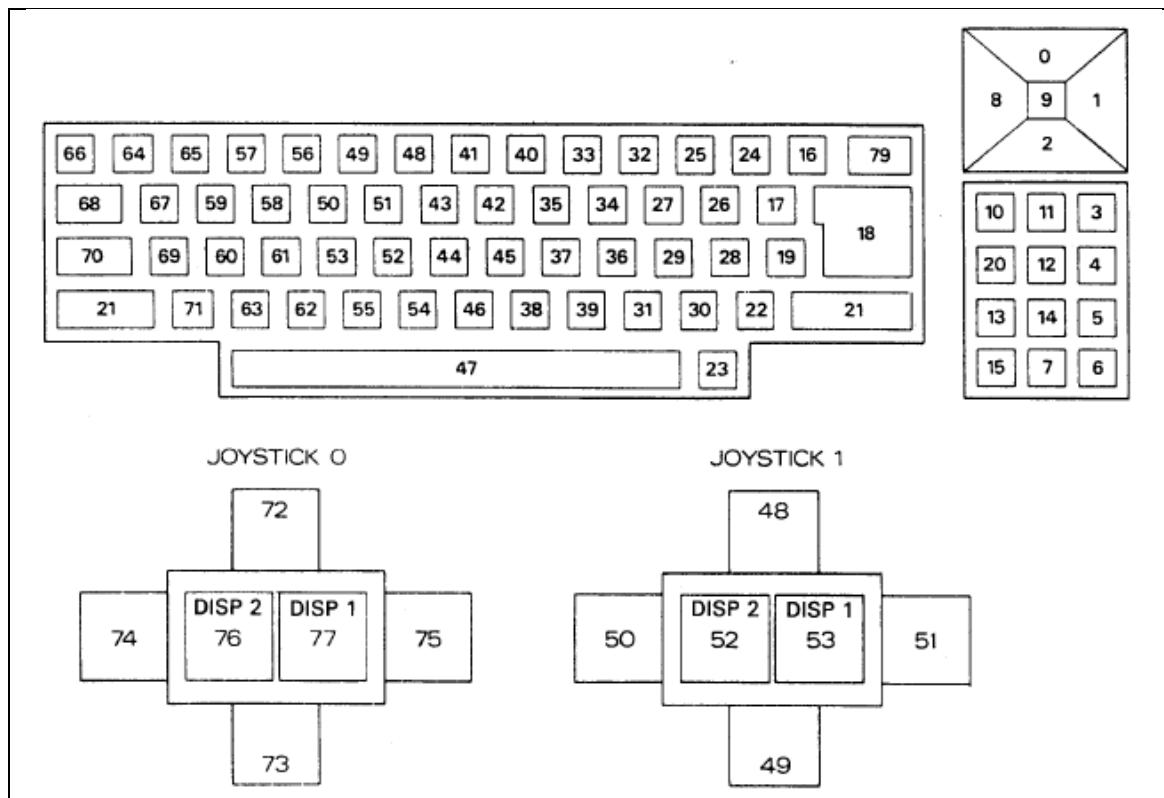


*Fig. 115 Paleta de C64*

En resumen, en Amstrad puedes elegir entre mas colores y asi siempre encuentras el color adecuado para difuminar, sombrear o simplemente encontrar el tono de color que necesitas. El gran acierto de Amstrad fue crear una paleta de 27 colores, aunque solo se puedan usar 16 a la vez. También puedes elegir 16 colores que no combinen bien entre sí y conseguir graficos muy horteras (lo cual es imposible en C64, al no poder elegir).

La paleta de 27 permitió que en muchos gráficos de carga de Amstrad haya auténticas obras de arte.

## 23 APENDICE III: INKEY codes



Siempre que quieras leer el teclado procura primero vaciar el buffer de lectura de las ultimas teclas pulsadas. Es muy habitual que en una pantalla en el que le pides el nombre al usuario por haber obtenido una alta puntuación, se “cuelen” las últimas pulsaciones del juego (movimientos y disparos), mostrando cosas como “OPPPOQQAAA” en el comando INPUT. Para evitarlo, ejecuta algo como:

```
10 B$=INKEY$: if B$<>"" THEN 10
20 INPUT "nombre:";$name
```

Además de esta recomendación, recuerda que la forma mas rápida de procesar el teclado es la siguiente:

```
10 IF INKEY(keycode) THEN 30: REM salta a 30 si no has pulsado
20 <instrucciones en caso de pulsar keycode>
30
```



## 25 APENDICE IV: Tabla ASCII del AMSTRAD CPC

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□
1	Г	Ф	!	1	А	Q	а	q	■	І	В	В	І	Г	+	↑
2	Л	Р	"	2	Б	R	b	r	■	-	”	В	/	-	♣	←
3	Л	Р	#	3	С	S	c	s	■	Л	Е	5	х	І	♦	→
4	Л	Р	*	4	Д	T	d	t	■	і	®	ε	^	■	♥	▲
5	Л	Р	×	5	Е	U	e	u	■	І	π	θ	)	■	♣	▼
6	✓	Π	&	6	F	V	f	v	■	р	г	λ	у	■	○	▶
7	Л	Р	'	7	G	W	g	w	■	Р	т	‘	μ	◀	●	◀
8	←	Х	(	8	H	X	h	x	■	-	і	4	π	‘	□	▼
9	→	Х	)	9	I	Y	i	y	■	І	σ	ے	॥	■	▲	▲
A	↓	?	*	:	J	Z	j	z	■	-	ۣ	ۤ	ۥ	ۦ	ۧ	ۧ
B	↑	Θ	+	:	K	[	k	]	■	±	ۢ	ۢ	ۢ	ۢ	ۢ	ۢ
C	Ψ	¶	,	<	L	\	l	l	■	۷	۷	۷	۷	۷	۷	۷
D	←	¶	-	=	M	]	m	3	■	۷	۷	۷	۷	۷	۷	۷
E	Φ	¶	.	>	N	t	n	~	■	۷	۷	۷	۷	۷	۷	۷
F	Ω	¶	/	?	O	_	o	❀	■	+	۷	۷	۷	۷	۷	۷

	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



## **26 APENDICE V: algunos efectos de sonido**

En primer lugar, debes saber que el Amstrad tiene 3 canales y sus identificadores son 1, 2 y 4. Para hacer que suenen dos canales o los 3 a la vez simplemente tienes que sumarlos

### **Uso del comando SOUND:**

SOUND canal,tono,duración,volumen, envolvente tono, envolvente volumen, ruido

El volumen va de 0 a 7

El ruido va de 0 a 31

### **Ejemplos:**

SOUND 1,2000,10,7 : suena el canal 1

SOUND 1+2,2000,10,7 : suenan los canales 1 y 2

Ahora te propongo algunos ejemplos para usar directamente en tus programas o para inspirarte y crear otros sonidos

Recoger un diamante o una moneda

ENT 1,10,-100,3:

sound 1,638,30,15,0,1

boing

ENV 1,1,15,1,15,-1,1:

SOUND 1,638,0,0,1

Boing 2

ENT 2,20,-125,1:

SOUND 1,1500,10,12,,2

muerte

ENT 3,100,5,3:

SOUND 1,142,100,15,0,3

Explosion

SOUND 7,1000,20,15,,,15

Explosion 2

ENV 1,11,-1,25:ENT 1,9,49,5,9,-10,15

SOUND 3,145,300,12,1,1,12

disparo

ENT -5,7,10,1,7,-10,1:

SOUND 1,25,20,12,,5



## 27 APENDICE VI: Rutinas interesantes del firmware

En este apartado voy a incluir algunas rutinas del firmware que se pueden invocar desde BASIC y que pueden ser interesantes en tus programas.

**CALL 0** : produce un reset del ordenador

**CALL &bc02** : inicializa la paleta a su valor por defecto. Es una buena práctica invocarla al principio de tu programa por si se encontrase alterada

**CALL &bd19** : sincroniza con el barrido de pantalla. Si manejas muy pocos sprites puedes conseguir un movimiento más suave pero ten en cuenta que esta instrucción va a ralentizar mucho tu programa.

**CALL &bb48** : desactiva el mecanismo de BREAK, impidiendo parar el programa si se encuentra en ejecución.

**CALL &bd21 , &bd22, &bd23, &bd24, &bd25** : produce un efecto de flash en la pantalla

Para resetar el TIMER del AMSTRAD:

**En un 6128**

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

**En un 464**

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

**CALL &bca7** : deja de sonar cualquier sonido que estuviese sonando

El comando GRAPHICS PAPER existe en CPC6128 pero no en CPC464. Sin embargo, hay una forma de tenerlo, mediante el **CALL &BBE4** y tantos parámetros con valor 1 como color de tinta queramos, por ejemplo:

**CALL &BBE4,1,1:** igual que "GRAPHICS PAPER 2" pero vale en cpc464

**CALL &BB18** : espera a que pulses una tecla



## 28 APENDICE VII: Tabla de atributos de Sprites

La siguiente tabla contiene las direcciones de memoria en la que se encuentran almacenados los atributos de cada sprite, y la longitud en bytes de cada uno.

	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

Tabla 7 direcciones de atributos de 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
--------------------------	--------------------------	---------------------------	---------------------------	--------------------------	--------------------------	------------------------	-------------------------------

Tabla 8 flags en el byte de estado



## 29 APENDICE VIII: Mapa de memoria de 8BP

```
AMSTRAD CPC464 MAPA DE MEMORIA de 8BP
;
; &FFFF +-----+
; | pantalla + 8 segmentos ocultos de 48bytes cada uno
; &C000 +-----+
; | system (simbolos redefinibles, stack pointer, etc.)
; 42619 +-----+
; | banco de 40 estrellas (desde 42540 hasta 42619 = 80bytes)
; 42540 +-----+
; | map layout de caracteres (25x20 =500 bytes)
; | y mapa del mundo (hasta 82 elementos caben en 500 bytes)
; | ambas cosas se almacenan en la misma zona de memoria
; | porque o usas una o usas otra
; 42040 +-----+
; | sprites (hasta 8.5KB para dibujos).
; | dispones de 8540 bytes si no hay secuencias ni rutas)
; | aquí tambien se almacenan las imágenes del alfabeto
; +-----+
; | definiciones de rutas (de longitud variable cada una)
; +-----+
; | secuencias de animacion de 8 frames (16 bytes cada una)
; | y grupos de secuencias de animacion (macrosecuencias)
; 33600 +-----+
; | canciones
; | (1400 Bytes para musica editada con WYZtracker 2.0.1.0)
; 32200 +-----+
; | rutinas 8BP (8200 bytes)
; | aqui estan todas las rutinas y la tabla de sprites
; | incluye el player de musica "wyz" 2.0.1.0
; 24000 +-----+
; |variables el BASIC
; | V
; |
; | ^ BASIC (texto del programa)
; |
; 0 +-----+
```



## 30 APENDICE IX: comandos disponibles 8BP

Lista de comandos disponibles en orden alfabético:

3D, <flag>, #, offsety	Activa el modo de proyección pseudo 3D
ANIMA, #	cambia el fotograma de un sprite según su secuencia
ANIMALL	cambia el fotograma de los sprites con flag animación activado (no hace falta invocarla, basta con un flag en la instrucción PRINTSPALL para que sea invocada)
AUTO, #	movimiento automático de un sprite de acuerdo a su velocidad en la tabla de sprites
AUTOALL, <flag enrutado>	movimiento de todos los sprites con flag de movimiento automático activo
COLAY, umbral_ascii, @colision, #  COLAY, @colision, #  COLAY, #  COLAY	detecta la colisión con el layout y retorna 1 si hay colisión. Acepta un número variable de parámetros (siempre en el mismo orden) desde 4 hasta ninguno (tiene memoria)
COLSP, #, @collided%  COLSP, 32, ini, fin  COLSP, 33, @collided%  COLSP, 34, dy, dx	retorna primer sprite con el que colisiona #. Se puede configurar el comando con los códigos 32,33 y 34
COLSPALL, @quien%, @conquien%  COLSPALL, colisionador  COLSPALL	Retorna quien ha colisionado (collider) y con quién ha colisionado (collided)
LAYOUT, y, x, @string\$	imprime un layout de imágenes de 8x8 y rellena map layout
LOCATESP, #, y, x	cambia las coordenadas de un sprite (sin imprimirla)
MAP2SP, y, x  MAP2SP, status	crea sprites para pintar el mundo en juegos con scroll. Los sprites se crean con estado = status
MOVER, #, dy, dx	movimiento relativo de un solo sprite
MOVERALL, dy,dx	movimiento relativo de todos los sprites con flag de movimiento relativo activo
MUSIC, cancion, speed	comienza a sonar una melodía
MUSICOFF	deja de sonar la melodía
PEEK, dir, @variable%	lee un dato 16bit (puede ser negativo)
POKE, dir, valor	introduce un dato 16bit (que puede ser negativo)
PRINTAT, flag, y, x, @string	Imprime una cadena de "minicaracteres" redefinibles
PRINTSP, #, y, x  PRINTSP, #  PRINTSP,32, bits	imprime un solo sprite (# es su número) sin tener en cuenta byte de status. Si se especifica 32 entonces establecemos bits fondo
PRINTSPALL, ini, fin, anima, sync  PRINTSPALL, ordermode	imprime todos los sprites con flag de impresión activo. Si se invoca con un solo parámetro, se establece el modo de ordenamiento
RINK,tini,color1,color2,...,colorN  RINK, salto	Rota un conjunto de tintas de acuerdo a un patrón definible compuesto por cualquier número de tintas
ROUTESP, #, pasos	Hace recorrer de golpe N pasos en la ruta asignada a un sprite,
ROUTEALL	Modifica la velocidad de los sprites con flag de ruta (no hace falta invocarla, basta con un flag en la instrucción AUTOALL para que sea invocada)
SETLIMITS, xmin, xmax, ymin, ymax	define la ventana de juego, donde se hace clipping
SETUPSP, #, param_number, valor	modifica un parámetro de un sprite
STARS, initstar, num, color, dy, dx	scroll de un conjunto de estrellas
UMAP,adr_ini, adr_end, yini, yfin, xini, xfin	Actualiza los ítems del mapa del mundo con un subconjunto de elementos de un mapa mayor



## 31 APENDICE X: correspondencias RSX/CALL

Lista de comandos disponibles en orden alfabético y su dirección asociada para usar la invocación CALL &XXXX cuando sea necesario para aumentar la velocidad.

3D	&646E
ANIMA	&6F9C
ANIMALL	&76D3
AUTO	&714E
AUTOALL	&719C
COLAY	&71CB
COLSP	&736 <sup>a</sup>
COLSPALL	&756F
LAYOUT	&702C
LOCATESP	&6C7C
MAP2SP	&64AA
MOVER	&74FC
MOVERALL	&76AD
MUSIC	&6F55
MUSICOFF	&5F3E
PEEK	&6931
POKE	&6944
PRINTAT	&6081
PRINTSP	&6C9F
PRINTSPALL	&62BE
RINK	&75ED
ROUTESP	&6608
ROUTEALL	&65E7
SETLIMITS	&6870
SETUPSP	&70CB
STARS	&73FA
UMAP	&5F4C