

# 8 BITS DE PODER

## En tu AMSTRAD CPC



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

V26

Jose Javier García Aranda



## INDICE

<b>1</b>	<b>¿POR QUÉ PROGRAMAR EN 2016 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	FUNCIONES DE 8BP .....	10
2.2	ARQUITECTURA DEL AMSTRAD CPC .....	11
2.3	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 5 FICHEROS .....	19
4.3	CREAR UN DISCO O UNA CINTA CON TU JUEGO .....	21
4.3.1	<i>Hacer un disco.....</i>	<i>21</i>
4.3.2	<i>Hacer una cinta .....</i>	<i>22</i>
<b>5</b>	<b>CICLO DE JUEGO.....</b>	<b>25</b>
5.1	COMO MEDIR LOS FPS DE TU CICLO DE JUEGO .....	25
<b>6</b>	<b>SPRITES .....</b>	<b>27</b>
6.1	EDITAR SPRITES CON SPEDIT Y ENSAMBLARLOS.....	27
6.2	SPRITES CON SOBREESCRITURA .....	32
6.3	TABLA DE ATRIBUTOS DE SPRITES .....	36
6.4	IMPRESIÓN DE TODOS LOS SPRITES Y ORDENADOS .....	40
6.5	COLISIONES ENTRE SPRITES .....	41
6.6	AJUSTE DE LA SENSIBILIDAD DE LA COLISIÓN DE SPRITES .....	42
6.7	QUIÉN COLISIONA Y CON QUIÉN: COLSPALL.....	43
6.7.1	<i>Cómo programar un disparo múltiple sin COLSPALL.....</i>	<i>44</i>
6.7.2	<i>Cómo programar un disparo múltiple con COLSPALL.....</i>	<i>46</i>
6.8	TABLA DE SECUENCIAS DE ANIMACIÓN .....	46
6.9	SECUENCIAS DE MUERTE .....	48
6.10	MACROSECUENCIAS DE ANIMACIÓN .....	49
<b>7</b>	<b>TU PRIMER JUEGO SENCILLO .....</b>	<b>51</b>
<b>8</b>	<b>JUEGOS DE PANTALLAS: LAYOUT.....</b>	<b>53</b>
8.1	DEFINICIÓN Y USO DEL LAYOUT .....	53
8.2	EJEMPLO DE JUEGO CON LAYOUT.....	55
8.3	CÓMO ABRIR UNA COMPUERTA EN EL LAYOUT .....	57
8.4	UN COMECOCOS: LAYOUT CON FONDO .....	58
8.5	CÓMO AHORRAR MEMORIA EN TUS LAYOUTS .....	61
<b>9</b>	<b>RECOMENDACIONES Y PROGRAMACIÓN AVANZADA .....</b>	<b>63</b>
9.1	RECOMENDACIONES DE VELOCIDAD .....	63
9.2	EJECUCIÓN ALTERNADA Y PERIÓDICA.....	71
9.2.1	<i>Aritmética modular.....</i>	<i>71</i>
9.2.2	<i>Operaciones binarias .....</i>	<i>71</i>
9.3	APROVECHA AL MÁXIMO LA MEMORIA .....	72
9.4	TÉCNICA DE “LÓGICAS MASIVAS” .....	74
9.4.1	<i>Ejemplo sencillo de lógica masiva .....</i>	<i>74</i>
9.4.2	<i>Mueve 32 sprites con lógicas masivas.....</i>	<i>75</i>

9.4.3	Técnica de lógicas masivas en juegos tipo “pacman” .....	77
9.4.4	Movimiento “en bloque” de escuadrones .....	78
9.4.5	Lógicas masivas: Movimiento “en fila india” .....	80
9.4.6	Lógicas masivas: Trayectorias complejas.....	80
9.4.7	Lógicas masivas: Controla el tiempo, no el espacio.....	83
9.5	ENRUTAR SPRITES CON ROUTEALL .....	87
<b>10</b>	<b>JUEGOS CON SCROLL.....</b>	<b>91</b>
10.1	SCROLL DE ESTRELLAS O TIERRA MOTEADA .....	91
10.2	CRÁTERES EN LA LUNA .....	93
10.3	MONTAÑAS Y LAGOS: TÉCNICA DE “MANCHADO” .....	97
10.4	TUNEL ROCOSO O CARRETERA .....	98
10.5	CAMINANDO POR EL PUEBLO .....	99
10.6	SCROLL BASADO EN UN MAPA DEL MUNDO.....	101
10.6.1	Mapa del mundo (Map Table).....	102
10.6.2	Uso de la funcion MAP2SP .....	104
<b>11</b>	<b>MÚSICA.....</b>	<b>109</b>
11.1	EDITAR MUSICA CON WYZ TRACKER.....	109
11.2	ENSAMBLAR LAS CANCIONES .....	110
<b>12</b>	<b>GUÍA DE REFERENCIA DE LA LIBRERÍA 8BP.....</b>	<b>113</b>
12.1	FUNCIONES DE LA LIBRERÍA .....	113
12.1.1	/ANIMA.....	113
12.1.2	/ANIMALL .....	114
12.1.3	/AUTO.....	114
12.1.4	/AUTOALL.....	115
12.1.5	/COLAY.....	115
12.1.6	/COLSP.....	117
12.1.7	/COLSPALL.....	118
12.1.8	/LAYOUT .....	119
12.1.9	/LOCATESP.....	121
12.1.10	/MAP2SP .....	121
12.1.11	/MOVER.....	122
12.1.12	/MOVERALL.....	123
12.1.13	/MUSIC.....	123
12.1.14	/MUSICOFF .....	124
12.1.15	/PEEK.....	124
12.1.16	/POKE.....	125
12.1.17	/PRINTSP.....	125
12.1.18	/PRINTSPALL.....	126
12.1.19	/ROUTEALL .....	127
12.1.20	/SETLIMITS.....	128
12.1.21	/SETUPSP.....	128
12.1.22	/SETUPSQ.....	130
12.1.23	/STARS.....	130
<b>13</b>	<b>ENSAMBLADO DE LA LIBRERÍA, GRÁFICOS Y MÚSICA .....</b>	<b>133</b>
<b>14</b>	<b>POSIBLES MEJORAS FUTURAS A LA LIBRERÍA .....</b>	<b>135</b>
14.1	MEMORIA PARA UBICAR NUEVAS FUNCIONES .....	135

14.2	AHORRAR MEMORIA DE GRÁFICOS.....	135
14.3	IMPRESIÓN A RESOLUCIÓN DE PÍXEL .....	135
14.4	LAYOUT DE MODE 1 .....	135
14.5	FUNCIONES DE SCROLL POR HARDWARE .....	136
14.6	MIGRAR LA LIBRERÍA 8BP A OTROS MICROORDENADORES.....	137
<b>15</b>	<b>ALGUNOS JUEGOS HECHOS CON 8BP .....</b>	<b>139</b>
15.1	MUTANTE MONTOYA .....	139
15.2	ANUNNAKI, NUESTRO PASADO ALIEN .....	141
<b>16</b>	<b>APENDICE III: ORGANIZACIÓN DE LA MEMORIA DE VIDEO .....</b>	<b>145</b>
16.1	EL OJO HUMANO Y LA RESOLUCIÓN DEL CPC .....	145
16.2	LA MEMORIA DE VIDEO.....	145
16.2.1	<i>Mode 2</i> .....	145
16.2.2	<i>Mode 1</i> .....	145
16.2.3	<i>Mode 0</i> .....	146
16.2.4	<i>Memoria de la pantalla</i> .....	146
16.3	BARRIDOS DE PANTALLA .....	148
16.4	CÓMO HACER UNA PANTALLA DE CARGA PARA TU JUEGO .....	149
<b>17</b>	<b>APENDICE IV: INKEY CODES.....</b>	<b>151</b>
<b>18</b>	<b>APENDICE V: PALETA.....</b>	<b>152</b>
<b>19</b>	<b>APENDICE VI: TABLA ASCII DEL AMSTRAD CPC.....</b>	<b>153</b>
<b>20</b>	<b>APENDICE VII: RUTINAS INTERESANTES DEL FIRMWARE.....</b>	<b>154</b>
<b>21</b>	<b>APENDICE VIII: TABLA DE ATRIBUTOS DE SPRITES .....</b>	<b>155</b>
<b>22</b>	<b>APENDICE IX: MAPA DE MEMORIA DE 8BP.....</b>	<b>156</b>
<b>23</b>	<b>APENDICE X: COMANDOS DISPONIBLES 8BP .....</b>	<b>157</b>



# 1 ¿Por qué programar en 2016 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	2.600.000.000	238310 @ 3Ghz (casi 500.000 veces mas rápido que un Z80 !)	PC

***Tabla 1 Comparativa 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:

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

También existe un blog con mucha información en:

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

y un canal de youtube

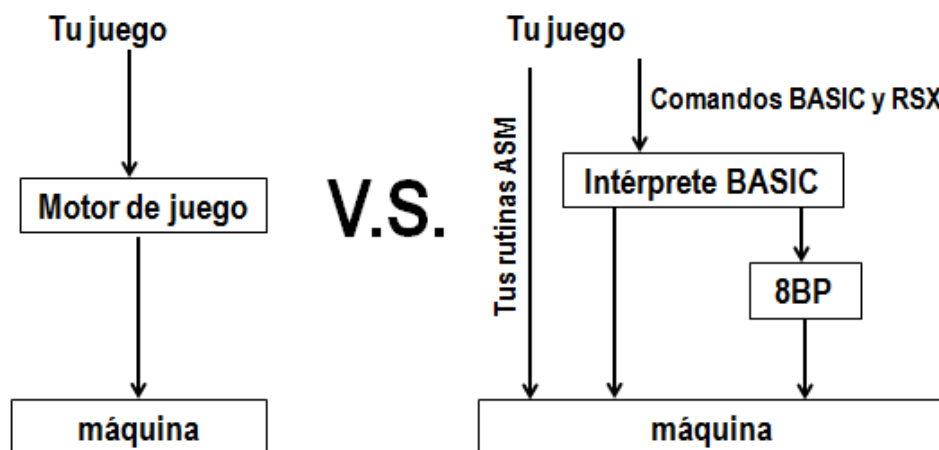
[https://www.youtube.com/channel/UCThvesOT-jLU\\_s8a5ZOjBFA](https://www.youtube.com/channel/UCThvesOT-jLU_s8a5ZOjBFA)



## 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 6 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 validos. 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.

## 2.1 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 (fíjate que aparece una barra vertical al principio de cada uno por ser “extensiones” del BASIC):

ANIMA, #	cambia el fotograma de un sprite segun su secuencia
ANIMALL	cambia el fotograma de los sprites con flag animacion activado
AUTO, #	movimiento automatico de un sprite de acuerdo a su velocidad
AUTOALL, <flag enrutado>	movimiento de todos los sprites con flag de mov automatico activo
COLAY, umbral_ascii, #, @colision	detecta la colision con el layout y retorna 1 si hay colision
COLSP, #, @id	retorna primer sprite con el que colisiona #
COLSPALL, @quien%, @conquien%	Retorna quien ha colisionado y con quién ha colisionado
LAYOUT, y,x, @string	imprime un layout de imagenes de 8x8 y rellena map layout
LOCATESP, #, y,x	cambia las coordenadas de un sprite (sin imprimirlo)
MAP2SP, y,x	crea sprites para pintar el mundo en juegos con scroll
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 a la velocidad deseada
MUSICOFF	deja de sonar la melodía
PEEK, dir, @variable%	lee un dato 16bit (puede ser negativo) de una dirección
POKE, dir, valor	introduce un dato 16bit (que puede ser negativo) en una dirección de memoria
PRINTSP, #, y,x	imprime un solo sprite (# es su numero) sin tener en cuenta byte de status
PRINTSPALL, orden, anima, sync	imprime todos los sprites con flag de impresion activo
ROUTEALL	Modifica la velocidad de los sprites con flag de ruta
SETLIMITS, xmin,xmax,ymin,ymax	define la ventana de juego, donde se hace clipping
SETUPSP, #, param_number, valor	modifica un parametro de un sprite
SETUPSQ, #, adr0,adr1,...,adr7	crea una secuencia de animacion
STARS, initstar,num,color,dy,dx	scroll de un conjunto de estrellas

*Tabla 2 Comandos disponibles en la librería 8BP*

Adicionalmente dispones de un comando experimental:

|RETROTIME, fecha

Este comando permite transformar tu CPC en una maquina 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 puedas 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.

## **2.2 Arquitectura del AMSTRAD CPC**

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

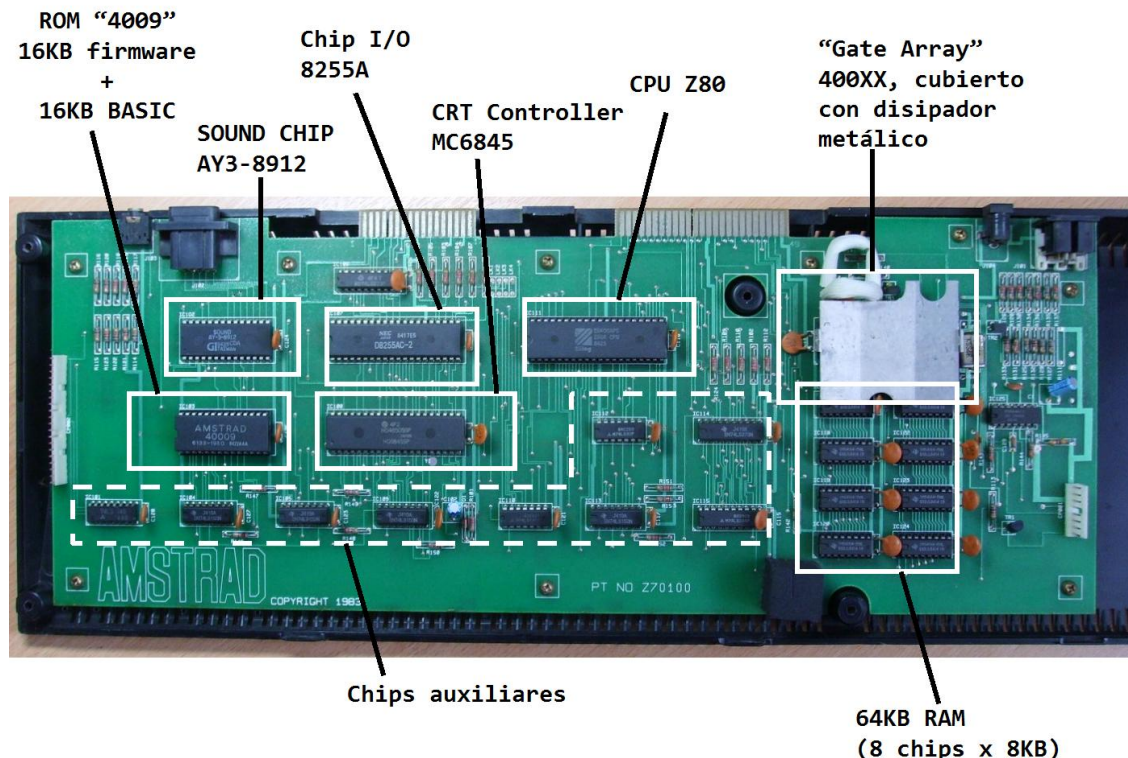
El amstrad es una computadora basada en el microprocesador Z80, funcionando a 4MHz. Como se aprecia en su diagrama de arquitectura, tanto la CPU como la matriz lógica de video (llamada "gate array") acceden a la memoria RAM, por lo que para poder "turnarse", los accesos a la memoria desde la CPU son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



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.

*Fig. 3 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. 4 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

ROM donde se almacena el intérprete BASIC, que esta 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 (intérprete BASIC y firmware) suman 32KB



*Fig. 5 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 video. Los programas en BASIC pueden ocupar desde la posición &40 (dirección 64) hasta la 42619, pues mas 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”).



*Fig. 6 Variable de sistema HIMEM*

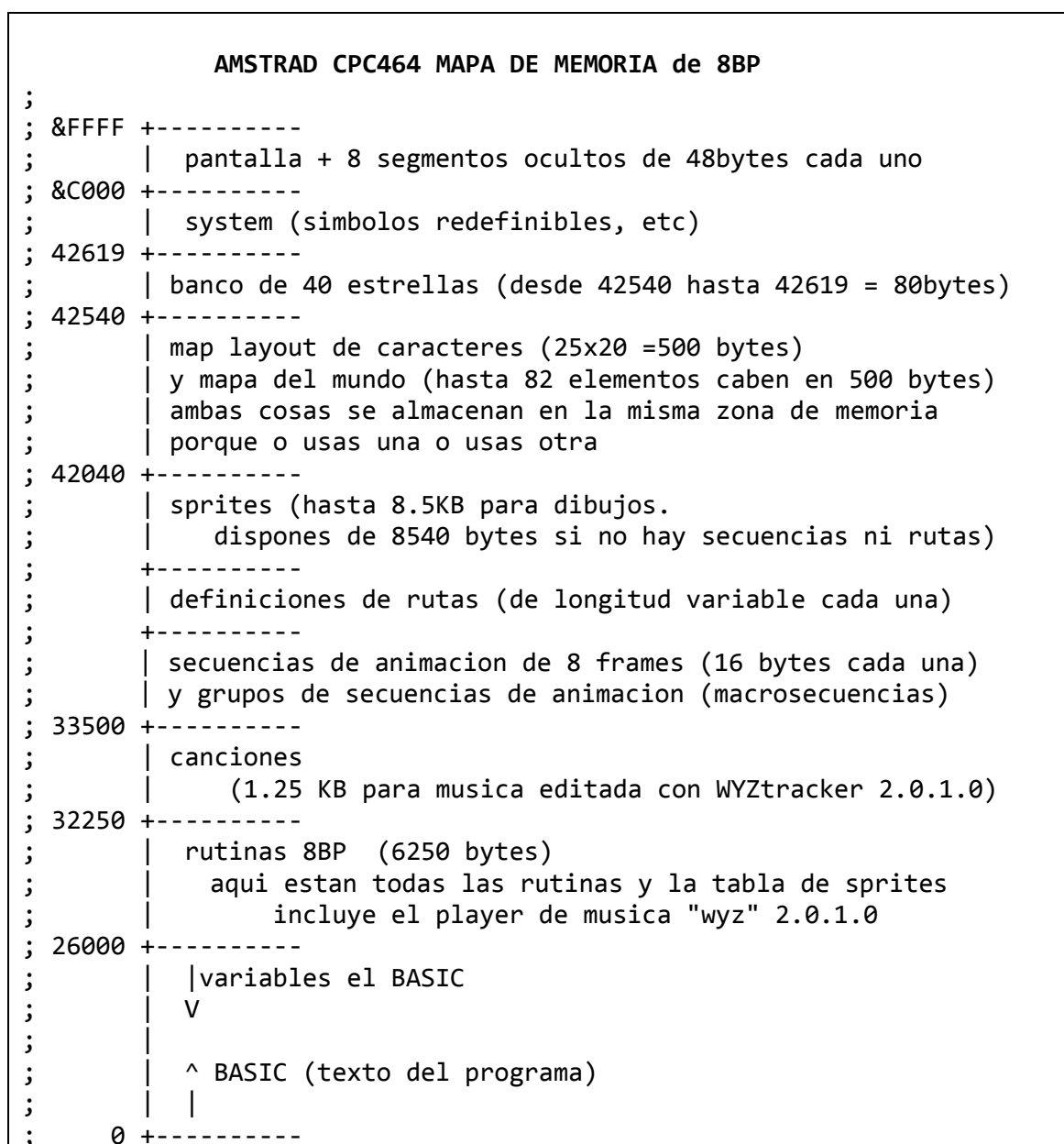
El funcionamiento del BASIC tiene en cuenta el almacenamiento del programa en direcciones crecientes desde la posición &40, mientras que una vez en ejecución, las variables que se declaran deben ocupar espacio para almacenar los valores que toman y puesto que no pueden ocupar la misma zona donde se almacena 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.

## 2.3 Uso de la memoria de 8BP

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

El texto de un programa escrito en BASIC se almacena a partir de la dirección &40 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 preocupes.

La librería se carga a partir de la dirección 26000, destinando la memoria a las funciones, el player de musica, las canciones, el mapa del mundo y los dibujos, tal y como se muestra en el siguiente diagrama



**Fig. 7 Memoria usando 8BP**

Antes incluso de cargar la librería, deberás ejecutar el comando

## MEMORY 25999

Para limitar el espacio ocupado por el BASIC. De esta manera, las variables de BASIC empezaran a ocupar espacio en ejecución desde la dirección 25999 hacia abajo. Tus programas pueden ser de casi 26KB 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 26KB, te lo aseguro.

Nota para programadores de versiones anteriores de 8BP:

*En versiones anteriores, la librería 8BP ocupaba 5250 bytes en lugar de 6250 bytes por lo que el comando MEMORY debía de invocarse con 26999, disponiendo de 27KB para tu programa en lugar de 26KB. Desde la versión V24 que incorpora un mecanismo de scroll multidireccional y mapa del mundo de juego, se consume 1KB más, por ello el comando MEMORY ahora debe apuntar a 25999. Puede que encuentres demos o ejemplos que usan versiones anteriores, aun así son compatibles con la V24, ya que un MEMORY 26999 simplemente “machacará” las funcionalidades nuevas de la v24, pero todo lo demás seguirá funcionando perfectamente. En este manual se incluye la descripción de la librería hasta la versión V26.*





### 3 Herramientas necesarias

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

**Spedit:** (“Simple Sprite Editor”) herramienta BASIC para editar tus graficos. El resultado de spedit es codigo 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.

**Wyztracker:** para componer musica, 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

Opcionalmente:

**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 mas 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.

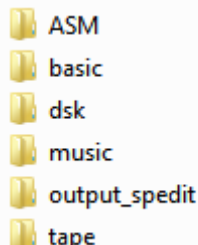


## 4 Pasos que debes dar para hacer un juego

### 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. 8 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, podras crear tus canciones y almacenarlas en formato .wyz en este directorio. Una vez que las “exportes”, se generarán un archivo asm que tendras 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 referencias 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 5 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 5 ficheros: 3 ficheros binarios y 2 ficheros BASIC

Los ficheros binarios son:

- Librería 8BP (es un fichero binario), incluyendo la tabla de atributos de sprites
- Fichero binario de musica con las melodías e instrumentos de tu juego
- Fichero binario de imágenes de sprites, incluyendo la tabla de secuencias de animacion

Y dos ficheros BASIC

- Cargador (carga la librería, musica 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, 33500, <tamaño>

### **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, 32250, 1250

### **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 que direcciones de memoria se han ensamblado (hay mas dependencias pero esa es una de ellas). Una vez re-ensamblada, tendrás que salvarla con el comando

SAVE "8BP.LIB", b, 26000, 6250

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 tu mismo has compuesto. La melodía numero 3 puede ser completamente diferente en otro juego.

### **PASO 4**

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

```
10 MEMORY 25999
```

```
20 LOAD "!8bp.lib"
```

```
30 LOAD "!music.bin"
```

```
40 LOAD "!sprites.bin"
```

```
50 RUN "!tujuego.bas"
```

## PASO 5

Programar tu juego, el cual debe primeramente ejecutar la llamada para instalar los comandos RSX, es decir CALL &6b78.

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 mas 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 Crear un disco o una cinta con tu juego

#### 4.3.1 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 25999
20 LOAD "!8bp.lib"
30 LOAD "!music.bin"
40 LOAD "!sprites.bin"
50 RUN "!tujuego.bas"
```

Para salvar cada uno de los ficheros debes usar el comando SAVE con los parámetros necesarios, por ejemplo:

```
SAVE "LOADER.BAS"  
SAVE "8BP.LIB", b, 26000, 6250  
SAVE "MUSIC.BIN", b, 32250, 1250  
SAVE "SPRITES.BIN", b, 33500, 8500  
SAVE "tujuego.BAS"
```

Si quieres 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 "engañar" al PC).

#### 4.3.2 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 25999  
20 LOAD "!8bp.lib"  
30 LOAD "!music.bin"  
40 LOAD "!sprites.bin"  
50 RUN "!tujuego.bas"
```

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

Para crear un wav o un cdt desde winape

file->tape->press record

En ese momento te saldrá un menú de administración de archivos para que podamos decidir que nombre le damos al fichero ".wav" o ".cdt"

Si estas 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. Asi la carga durará menos. Si no ejecutas ese comando, la grabación se realizará a 1000 baudios, más segura pero mucho mas 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, 26000, 6250
SAVE "MUSIC.BIN", b, 32250, 1250
SAVE "SPRITES.BIN", b, 33500, 8500
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





## 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. 9 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 12fps 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 cuantos frames por segundo es capaz de generar tu videojuego, porque así puedes tomar decisiones en la lógica de tu programa y medir cuanto 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 numero 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 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
```

Con ello habras 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

$$FPS = \text{ciclo} * 300 / (\text{TIME} - A)$$

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 &b8b7,0: 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
```

## 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 esta 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. 10 Redireccion de la impresora del CPC a un fichero con Winape*

Cuando ejecutes SPEDIT te aparecerá el siguiente menu, donde puedes elegir si vas a usar la paleta por defecto o bien una tuya que quieras definir. Si decides definir tu propia paleta, deberas 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.

```

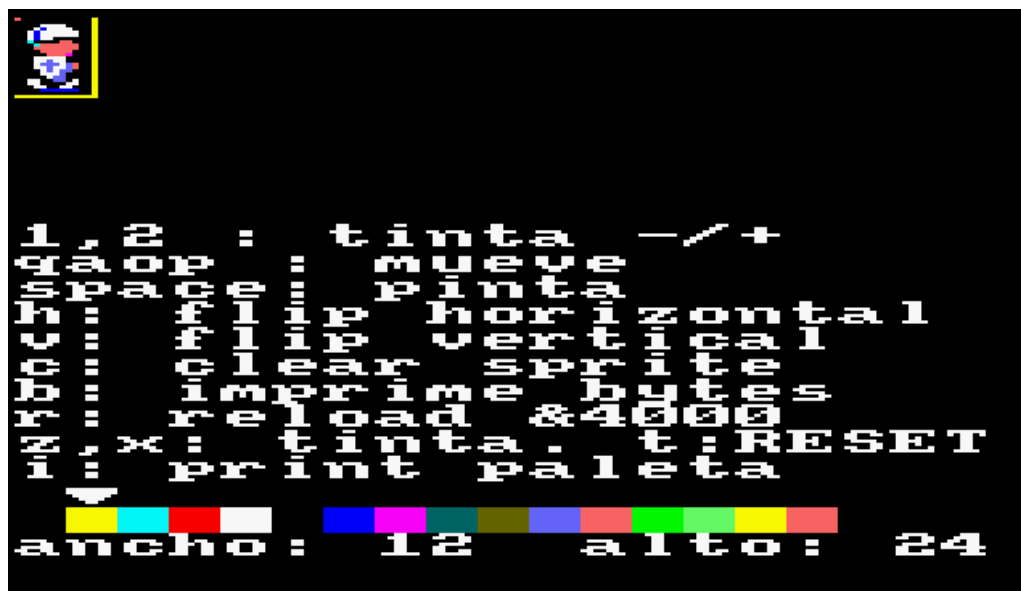
SPEDIT: Sprite Editor v2.0
---- CONTROLES EDICION----
1,2 : tinta -/+
qaop : mueve cursor pixel
space: tinta
h: flip horizontal
v: flip vertical
c: clear sprite
b: imprime bytes (asm) en printer
i: imprime paleta en printer
r: reload &4000
z, x: cambia color de tinta
t: RESET
-----
Antes de empezar puedes cargar un sprite
ensamblandolo en la direccion &4000.
No ensambles ancho y alto, solo los bytes
del dibujo.

La paleta la puedes cambiar pero debera
ser la misma en tu programa.
Selecciona paleta (1,2,3)
? 1:default 2:custom 3:overwrite

```

*Fig. 11 Pantalla ininical de SPEDIT*

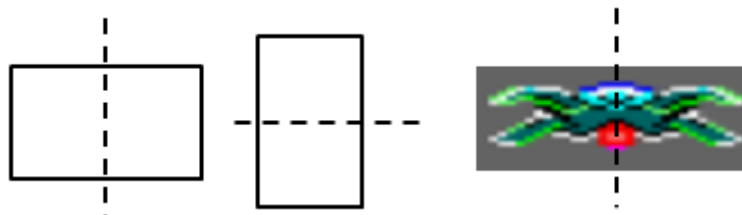
Suponiendo que eliges usar la paleta por defecto, la herramienta se pone en mode 0 y te permite editar dibujos, con la ayuda en la pantalla. Manejas un píxel que parpadea y en la parte inferior se muestra las coordenadas donde te encuentras y el valor del byte en el que te encuentras.



*Fig. 12 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. 13 sprites simétricos con SPEDIT*

Es fácil adaptar esta herramienta para que te permita editar en mode 1 si lo deseas. De hecho es tan sencillo como eliminar la línea de BASIC que establece el modo de pantalla. Como ves la herramienta no es del todo completa pero te permite hacer absolutamente todo lo que necesitas.

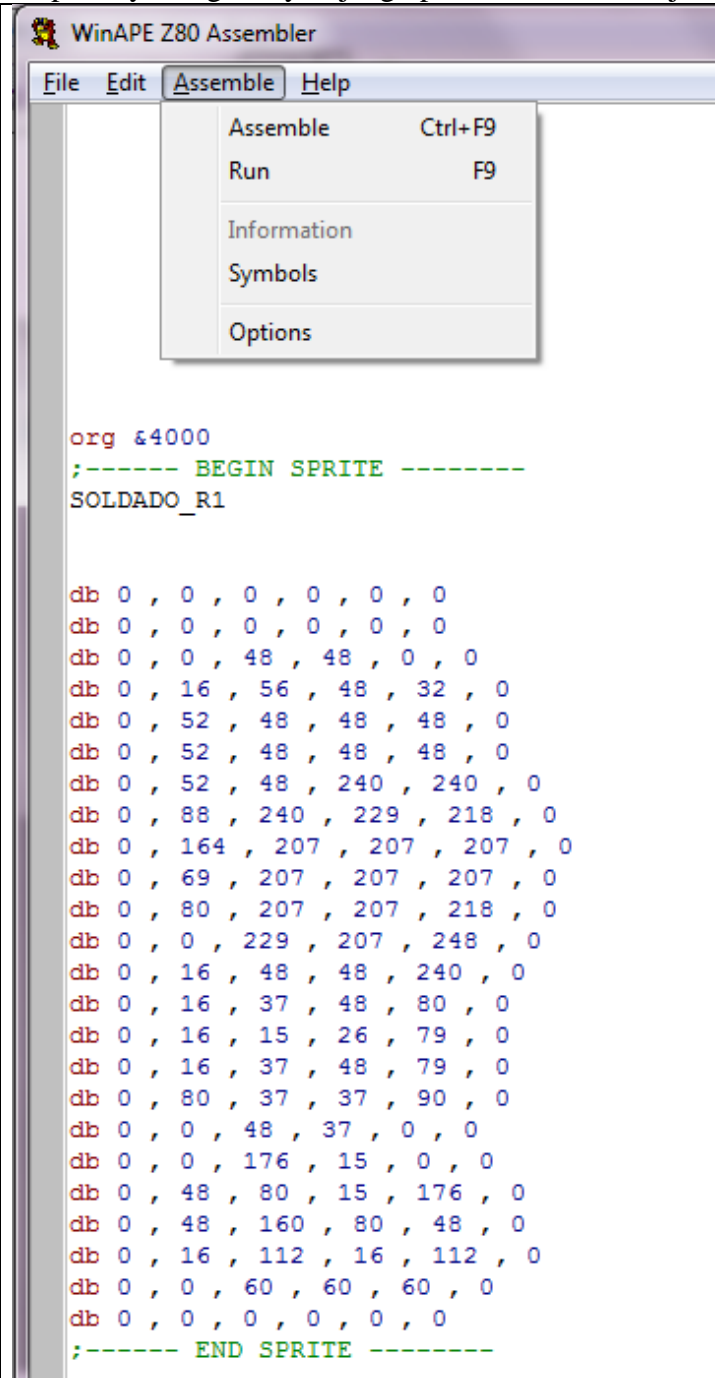
Una vez que has definido tu muñeco, para extraer el código ensamblador deberás pulsar la “b”. Esto mandará a la impresora (al fichero que hayamos definido como salida) un texto como el siguiente, al que puedes añadir un nombre, yo le he llamado “SOLDADO\_R1”

<pre> ;----- BEGIN SPRITE ----- 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 SPRITE ----- </pre>	 <p>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 si mismo”, ya que de lo contrario deraría un rastro, “manchando” la pantalla mientras avanza</p>
---	---

*Fig. 14 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 &4000, 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”. El sprite se cargará desde la dirección &4000, 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.

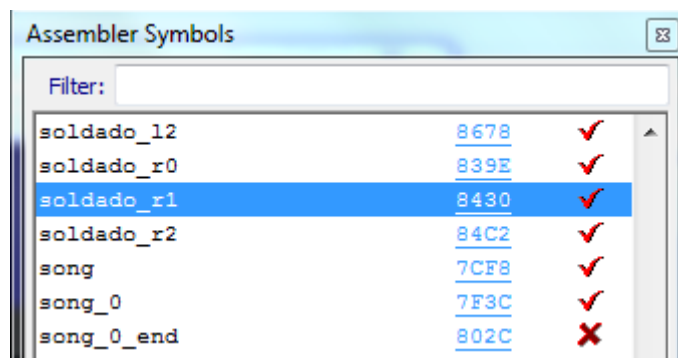
 <pre> WinAPE Z80 Assembler File Edit Assemble Help Assemble Ctrl+F9 Run F9 Information Symbols Options  org &amp;4000 ;----- BEGIN SPRITE ----- SOLDADO_R1  db 0 , 0 , 0 , 0 , 0 , 0 db 0 , 0 , 0 , 0 , 0 , 0 db 0 , 0 , 48 , 48 , 0 , 0 db 0 , 16 , 56 , 48 , 32 , 0 db 0 , 52 , 48 , 48 , 48 , 0 db 0 , 52 , 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 SPRITE ----- </pre>	<p>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 &amp;4000, que en decimal es 16384, es decir la posición 16KB</p> <p>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.</p> <p>Si algún día SPEDIT se hace mas grande y llega a tener mucha mas funcionalidad, habrá que llevarse este pequeño buffer mas lejos, pero de momento es perfectamente valido así, en la dirección &amp;4000.</p>
---	---

*Fig. 15 Ensamblado de gráficos*

Para saber en que dirección de memoria se ha ensamblado cada imagen, utiliza desde el menu de winape:

Assemble->symbols

Con ello veras una relacion de las etiquetas que has definido, como “SOLDADO\_R1” y la dirección de memoria (en hexadecimal) a partir de la cual se ha ensamblado.



*Fig. 16 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 que un 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. Una vez que estén todas hechas podrás ensamblar ese fichero en la dirección 33500 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 33500 ejecuta desde el BASIC del CPC en el emulador:

SAVE “sprites.bin”, b, 33500,2500

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 8500. No te olvides de la letra “b”, que sirve para especificar que se trata de un fichero binario. Si no estas seguro de cuanto ocupan tus imágenes, pon lo máximo, que es 8540 bytes

Si quieres salvar las imágenes y las secuencias de animación juntas (las secuencias se encuentran en la 33500) simplemente ejecuta

SAVE “sprites.bin”, b, 33500,8500

Para cargar tus sprites en memoria RAM, simplemente ejecuta:

LOAD “sprites.bin”

## 6.2 Sprites con sobreescritura

Desde la versión v22 de 8BP es posible editar sprites transparentes, es decir, que pueden sobrevolar un fondo y lo reestablecen al pasar. Para ello los sprites que disfruten 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 como 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 conmutar 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 conmutar, alternativamente, en cada fotograma. Estas técnicas, aunque funcionan muy bien, tienen un par de desventajas para nuestros propósitos: llevan mas tiempo de CPU y consumen mucha mas 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.

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. 17 Sprites con sobreescritura en 8BP*

En el AMSTRAD un pixel de mode 0 es representado con 4 bits, por lo que son posibles hasta 16 colores diferentes de una paleta de 27. 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 solo 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 arboles 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 una paleta así definida que puedes usar seleccionando la opción “3” 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. 18 Paleta ejemplo de sobreescritura*

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

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:

```

' ----- 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 solo 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 podras tener resultados muy coloristas.

El color 0001 tiene un uso “especial”. Si editas un sprite que no use el flag de sobreescritura, el color 1 será simplemente el color 1. 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.

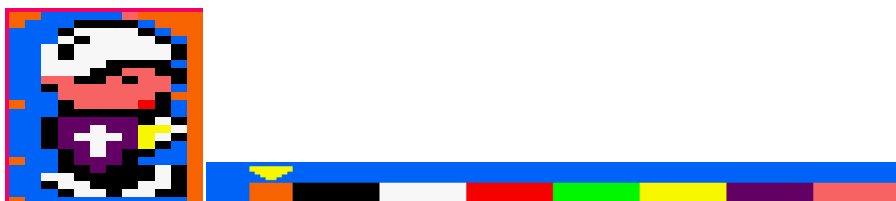
code	significado
0000	Color 1 de fondo. Si un sprite lo usa y le activas el flag de sobreescritura, significa "transparencia", es decir, imprimir reestableciendo el fondo
0001	Color 2 de fondo. Si un sprite lo usa y le activas el flag de sobreescritura, deja de significar un color para significar “no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

code	significado
0110	color 3 de sprite
0111	
1000	color 4 de sprite
1001	
1010	color 5 de sprite
1011	
1100	color 6 de sprite
1101	
1110	color 7 de sprite

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 color 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 pixeles a 0000 ya es suficiente para borrar el rastro del sprite mientras se desplaza.



*Fig. 19 sprite diseñado 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 si mismo, todo su contorno esta pintado con el color 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 mas inferior. Esto se hace con un simple parámetro al imprimir los sprites con el comando |PRINTSPALL, que veremos más adelante.

Fig. 20 Colisión multiple, efecto del color 0001

Las operaciones de impresión con este mecanismo son muy rápidas, sin necesidad de definir lo que se conoce como “mascaras 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				Metodo convencional: Se imprime Fondo AND mask OR sprite
0	2	2	0	1	0	0	1	
2	3	3	2	0	0	0	0	
0	2	2	0	1	0	0	1	
0	2	2	0	1	0	0	1	Metodo 8BP: Imprimimos Fondo OR sprite
0	0	0	0	1	1	1	1	

Fig. 21 En 8BP no son necesarias máscaras

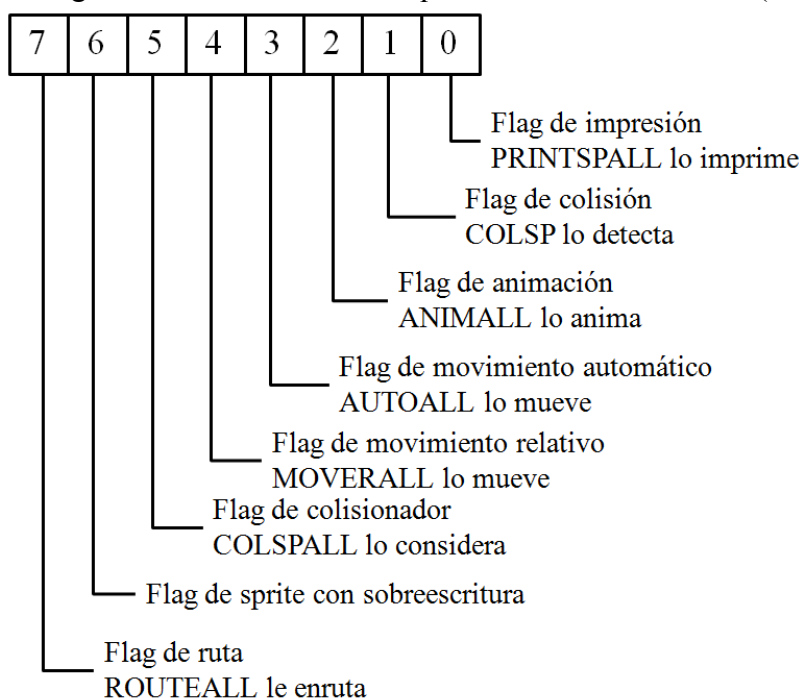
La impresión de sprites con el flag de sobreescritura activo es más costosa que la impresión sin sobreescritura. A pesar de no requerir máscara y ser muy rápida, esta impresión cosume aproximadamente 1.5 veces el tiempo que consume la impresión de un sprite sin sobreescritura. Por ese motivo, úsala cuando sea necesario, y no la uses si tu juego no va a tener un dibujo de fondo que los sprites deban respetar.

### 6.3 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 tambien 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”)



**Fig. 22 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 colision: una fruta o moneda por ejemplo pueden no tener flag de impresion pero tener el de colision
- bit 2: flag de animacion 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 “|MOVERALL, 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 funcion |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, reestableciéndolo 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 que 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 deteccion de colision con otros sprites y animacion debe tener:

$$\text{status} = 1(\text{bit } 0) + 2(\text{bit } 1) + 4(\text{bit } 2) = 7 = \&\text{x}0111$$

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

$$\text{status} = 1(\text{bit } 0) + 0(\text{bit } 1) + 0(\text{bit } 2) + 0(\text{bit } 3) + 16(\text{bit } 4) = 17 = \&\text{x}10001$$

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

$$\text{status} = 0(\text{bit } 0) + 2(\text{bit } 1) = 2 = \&\text{x}10$$

Una nave que va a seguir una trayectoria predefinida. Va a requerir del flag de ruta, el de movimiento automatico, el de animación, el de colision 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 ultimo he puesto 4 flags activos correspondientes respectivamente a los de movimiento automatico, animación, colision 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	Logitud (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 esta 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

Por ejemplo, la dirección de las coordenadas de cada sprite se puede calcular así

$$\text{Dirección coordenada Y} = 27000 + 16 * N + 1$$

$$\text{Dirección coordenada X} = 27000 + 16 * N + 3$$

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.

Las direcciones de los atributos de los 32 sprites para manejar con POKE y PEEK son:

	1byte	2 bytes	2 bytes	1byte	1byte	1byte	1byte	2 bytes	1byte
<b>sprite</b>	<b>status</b>	<b>coordy</b>	<b>coordx</b>	<b>vy</b>	<b>vx</b>	<b>seq</b>	<b>frame</b>	<b>imagen</b>	<b>ruta</b>
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.

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.



## 6.4 Impresión de todos los sprites y ordenados

En la librería 8BP dispones 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 3 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, <flag de orden>, <animar antes de imprimir>, <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.
- 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.
- Por último tenemos el parámetro de ordenamiento. Debemos indicar el número de sprites ordenados por coordenada Y que vamos a imprimir. Por ejemplo si ponemos un 0, entonces los sprites se imprimirán secuencialmente desde el sprite 0 hasta el sprite 31. Si ponemos un 10 se imprimirán del 0 al 10 ordenados (11 sprites) y del 11 al 31 de modo secuencial. Si ponemos un 31 se imprimirán todos los sprites 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. 23 Efecto del ordenamiento de sprites*

|PRINTSPALL, 0, 1, 0 : imprime de forma secuencial todos los sprites

|PRINTSPALL, 31, 1, 0 : imprime de forma ordenada todos los sprites



|PRINTSPALL, 7, 1, 0 : imprime de forma ordenada 8 sprites y el resto secuencial

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 2.5ms pero si ordenas solo 5 te puedes ahorrar 2ms. Quizas 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 estan casi ordenados y de un fotograma a otro solo se desordena uno o dos sprites, no mas, 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 tremendamente 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.

## **6.5 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 quieres 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 rellenará 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 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 cero. 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!

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. Es por ello que 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:

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

```
|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 1.1 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 : rem esta invocación es equivalente a |COLSP, 23, @col%
```

## **6.6 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)

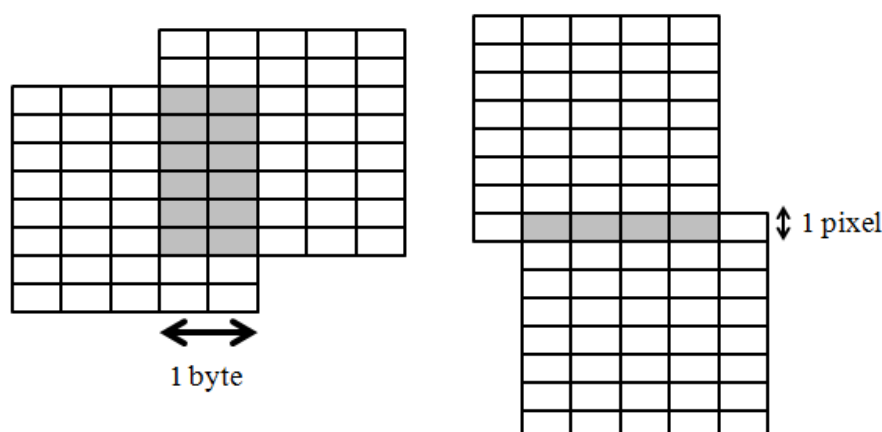
```
|COLSP, 34, dy, dx
```

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. 24 Colisión estricta con COLSP, 34, 0, 0*

Sin embargo, si estamos haciendo un juego en MODE 0, donde los pixels son mas anchos que altos, es quizás mas 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 colision 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 si 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

## **6.7 Quién colisiona y con quién: COLSPALL**

Con la funcion 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 4 invocaciones resulta costoso. Para ello disponemos de un comando adicional: COLSPALL

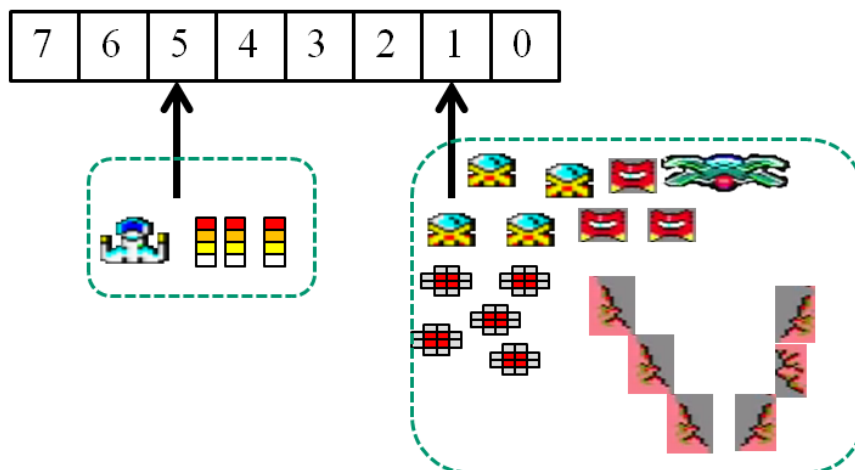
Esta función funciona en dos pasos, primero debemos especificar que variables van a almacenar el sprite colisionador y el colisionado

|COLSPALL, @colisionador%, @colisionado%

Y posteriormente, en cada ciclo de juego simplemente invocamos la función sin parámetros:

## |COLSPALL

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 colision (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



*Fig. 25 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. 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 solo se podrá detectar una colision, 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.

### 6.7.1 Cómo programar un disparo múltiple sin COLSPALL

Vamos a analizar como se detecta la colisión de una nave y sus disparos sin hacer uso de COLSPALL. Como veremos, debido a las veces que debemos invocar a COLSP, es más eficiente usar COLSPALL. Este ejemplo permitirá comprenderlo y al mismo tiempo nos evidencia que si solo tenemos un sprite colisionador (nuestra nave) o dos (nuestra nave y un disparo como mucho), se puede prescindir de COLSPALL.

Vamos a suponer que tenemos hasta 3 disparos con los sprites 7,8 y 9 y que nuestra nave es el sprite 31. La detección de colision de nuestra nave la haremos con una invocación a COLSP en cada ciclo de juego

Lo primero de todo, para evitar el paso de parámetros en cada invocación a COLSP haremos

col%=0

|COLSP,33,@col%: ' el sprite 33 no existe, se usa para indicar la variable de trabajo

De este modo sucesivas invocaciones a COLSP,<sprite> dejarán el resultado (número de sprite que colisiona) en la variable col%

|COLSP,31 : ' suponemos que el sprite 31 es nuestro personaje.

Para detectar las colisiones de nuestros disparos, la solución mas efectiva se basa en uno de los casos más sencillos de aplicación de la técnica de "lógicas masivas", restringiendo a contemplar solo una de las siguientes cosas a la vez:

- colisiona uno de los 3 disparos (solo uno), y mata a un enemigo
- un disparo sale del área de pantalla (solo uno)

La rutina seria algo así (dentro de cada ciclo de juego haríamos un gosub 750). En el peor de los casos hay 4 invocaciones a COLSP, una para nuestra nave y 3 para cada uno de los disparos si han sido disparados.

```
744 ' RUTINA DE COLISION DE DISPAROS
745 ' -----

746 ' CASO 1 comprobamos si hay alguna colision
747 ' -----
750 if peek(27112)>0 then |colsp,7: if col%<32 then dir=27113:goto 820
760 if peek(27128)>0 then |colsp,8: if col%<32 then dir=27129:goto 820
770 if peek(27144)>0 then |colsp,9: if col<32 then dir=27145:goto 820

780 ' CASO 2 comprobamos si el disparo se ha salido de pantalla
790 ' -----
800 dc=dc mod 3 +1:dir=ddisp(dc):|peek,dir,@yd%: if yd%<-10 then poke dir-1,0:
|POKE,dir,200:nd=nd-1
810 return:

820 ' continuacion del caso 1 en caso de colision
825 ' -----
830 ' desactivo el disparo
831 ' el sprite 6 sirve para borrar el disparo, simplemente
840 poke dir-1,0: nd=nd-1:|PRINTSP,6,peek(dir),peek(dir+2): poke dir,255
850 ' ahora proceso al enemigo segun sea duro o blando
860 if col>=duros then return
870 ' alcance de enemigo tipo blando. lo matamos, asignandole una secuencia de muerte y
poniendo su estado a no colisionar mas
871 ' la secuencia de animación 4 es una secuencia de animación de "Muerte", una explosion
880 if col>=blandos then |SETUPSP,col,7,4:|SETUPSP,col,0,&x101:return
890 return
```

Como puedes ver, la rutina comienza chequeando si cada disparo está activo mirando en la dirección de memoria de su byte de status, para justo a continuación comprobar si colisiona. Todo esto tiene un elevado coste. Funciona, pero podemos hacerlo mas deprisa si usamos COLSPALL, como ahora veremos

### 6.7.2 Cómo programar un disparo múltiple con COLSPALL

Ahora vamos a ver la ventaja de utilizar COLSPALL, mucho más rápido ya que no vamos a tener que invocar multiples veces a COLSP. Las únicas recomendaciones importantes son:

- Que nuestro sprite 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

```
col%=32:sp%=32:COLSPALL,@sp%,@col%
```

En el ciclo de juego pondremos:

```
|COLSPALL: if sp%<32 then if sp%=31 then gosub 300:goto 2000: else gosub 770
```

Con esta línea ya sabemos si hay collision, pues entonces la variable sp será <32

Además si es 31 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

La rutina de procesamiento de la colisión del disparo será algo como:

```
769' rutina colision disparo-----  
770 dir=ddisp(sp%-6): 'primero paro el disparo y luego actuo segun el tipo de enemigo  
771' el sprite 6 es de borrado, para eliminar el disparo  
775 poke dir-1,0: nd=nd-1:|PRINTSP,6,peek(dir),peek(dir+2): poke dir,255  
777 if col%>=duros then return  
778 ' la secuencia 4 es una secuencia de animación de "Muerte", una explosion  
780 if col%>=blandos then |SETUPSP,col%,7,4:|SETUPSP,col%,0,&x101:return  
785 return
```

En resumen, todo es más sencillo. No necesitamos comprobar el estado de los disparos ni necesitamos invocaciones extra a COLSP. Con una sola invocación a COLSPALL ya sabemos quién ha colisionado, y con quién ha colisionado.

## 6.8 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 mas, aunque siempre puedes hacer secuencias mas 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. Veamos un ejemplo en lenguaje ensamblador aunque también la puedes crear usando el comando |SETUPSQ

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

el equivalente en BASIC usando la librería 8BP es

```
SETUPSQ, 1, &926c,&92FE,&9390 ,&02fe ,0,0,0,0
```

Nótese que en BASIC requieres conocer las diferentes direcciones de memoria en las que se ha ensamblado cada imagen. Ello lo puedes ver desde el menu de winape Assemble->symbols

En ensamblador usas directamente las etiquetas de cada imagen, por lo que es más“entendible”. Pocas veces el ensamblador es mas fáil de entender que el BASIC!!!

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. Asi no tendras que consultar las direcciones de memoria cada vez que ensambles. Esto aplica tanto a la instrucción SETUPSP, #, 9, <dirección> como a SETUPSQ.

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, de modo que al final son 4 fotogramas:



*Fig. 26 secuencia de animación*

Si quisieses hacer una secuencia de mas de 8 fotogramas podrías simplemente encadenar dos secuencias seguidas y cuando el personaje llegase al ultimo 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. 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.

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;
;=====
; 31 secuencias de animacion de 8 frames
;=====
```

```

; 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 /secuencia
; al encontrar un cero se comienza de nuevo.
; si no hay cero, tras el frame 8 se comienza de nuevo
; en total se puede definir 31 secuencias diferentes
; la secuencia cero es que no hay secuencia.
; empezamos desde la secuencia 1

;-----secuencias de animacion del personaje -----
_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

```

La librería 8BP te proporciona un comando llamado |SETUPSQ con el que puedes crear secuencias de animación desde BASIC. Dicho comando lo que hace realmente es meter datos en las direcciones de memoria a partir de la 33500. Si las creas y las ensamblas y las salvas en el fichero de imágenes te ahorrarás tener que crearlas desde BASIC y por lo tanto ahorrarás líneas de BASIC.

## 6.9 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 estan 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

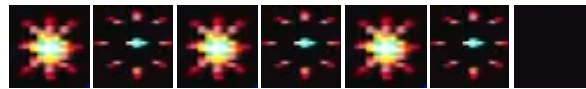
Si usas una secuencia de muerte, no te olvides de que el último fotograma antes de encontrar el “1” sea uno completamente vacío, de modo que no quede ningún resto de la explosión.

Ejemplo de secuencia de muerte:

```
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



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

## 6.10 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 ( $V_x < 0$ ,  $V_y = 0$ ), la siguiente para la derecha ( $V_x > 0$ ,  $V_y = 0$ ), etc, siguiendo el siguiente orden (ten cuidado porque es fácil equivocarse):



*Fig. 27 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 no se pueden crear con el comando SETUPSQ, hay que especificarlas en el fichero sequences\_tujuego.asm, del que a continuación tienes un ejemplo:

```
org 33500;
;=====
; secuencias de animacion
;=====
; cada secuencia contiene las direcciones de frames de animacion
; cíclica.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
; cada secuencia
; 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 NAVE,0,0,0,0,0,0,0;1
dw JOE1,JOE2,0,0,0,0,0,0;2 UP JOE
dw JOE7,JOE8,0,0,0,0,0,0;3 DW JOE
dw JOE3,JOE4,0,0,0,0,0,0;4 R JOE
dw JOE5,JOE6,0,0,0,0,0,0;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

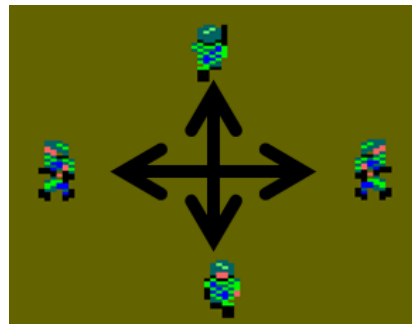
```

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 si, 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 25999
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
250 |AUTOALL:|PRINTSPALL
270 GOTO 199
280 |MUSICOFF:MODE 1: INK 0,0:PEN 1

```



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 que 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.

```
130 |SETUPSP, 31, 7, 2:|SETUPSP, 31, 7, 32
```

## 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 quedado con el identificador 9 y 10 para las direcciones de movimiento derecha e izquierda respectivamente.

Las dos secuencias de animación las hemos creado bien desde el fichero de secuencias.asm o bien en basic con el comando |SETUPSPQ (el cual no he incluido en este listado)

```
10 MEMORY 25999
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 imprimirlo 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 corretear horizontalmente. Fíjate que si al caminar hacia la izquierda sobrepasas el valor minimo del limite establecido con SETLIMITS, se producirá el “clipping” del personaje, mostrándose tan solo la parte que queda dentro del area de juego permitida



*Fig. 28 Un sencillo juego*



## 8 Juegos de pantallas: layout

### 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 estás 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.

Caracter	Sprite id	Código ASCII
“ “	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 imprimirlo 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 tu quieras justo después de imprimir el layout

Para detectar colisiones con el layout dispones de la función:

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

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. Y no sólo debe estar previamente definida, sino que debes poner el “%” para indicar que es una variable entera, aunque estes usando DEFINT A-Z.

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:

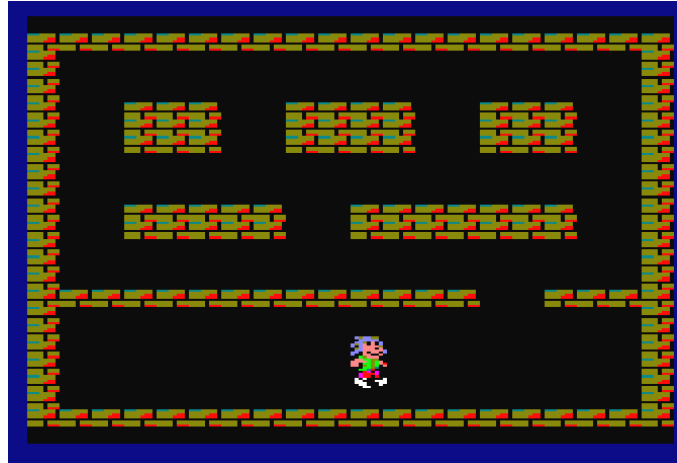
```
col%=0  
|COLAY,0,@col%
```

Si no hay colision, la variable tomará el valor cero. Si hay colisión, tomará el valor 1. Hay colisión si el sprite choca con algun 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.

## ***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.



En la rutina de control del personaje hemos incluido colisión con el layout. En funcion de la dirección en la que avanzamos, modificamos las coordenadas “nuevas” (yn , xn) e invocamos a la funcion de colision 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 imprimirle de nuevo

56



```

310 xa=40:xn=xa:ya=150:yn=ya:' coordenadas del personaje
311|SETUPSP,0,0,&x111:' deteccion de colision con sprites y layout
312|SETUPSP,0,7,1: ' secuencia = 1
320 |LOCATESP,0,ya,xa: 'colocamos al personaje (sin imprimirlo)
325 c1%=0:'declaramos la variable de colision, 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,0,@c1%:IF c1%=0 THEN 1536
1534 yn=ya:|POKE, 27001,yn:|COLAY,0,@c1%:IF c1%=0 THEN 1536
1535 xn=xa: yn=ynn:|POKE, 27001,yn:|POKE, 27003,xn:|COLAY,0,@c1%: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. 30 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 esta 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 colision con layout dará como resultado 0

```

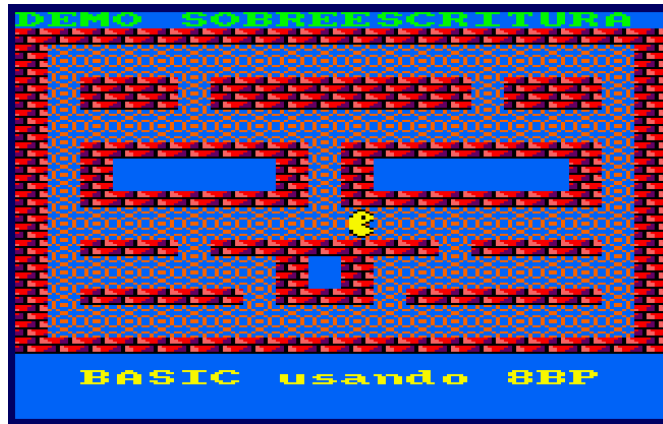
----- esta parte esta dentro del bucle de logica ----
6410 |PRINTSPALL,1,0
6411 |COLSP,0,@cs%:IF cs%<32 THEN IF cs%>=15 then gosub 6500
(... mas instrucciones . . .)
-----

----- rutina de apertura de compuerta-----
6499' se 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. 31 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 25999
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$(1)= "ZZZZZZZZZZZZZZZZZZZZZZ"
110 c$(2)= "ZZZZZZZZZZZZZZZZZZZZZZ"
120 c$(3)= "ZZZZZZZZZZZZZZZZZZZZZZ"
125 c$(4)= "ZYZZZYZZZZZZZZZYZZZY"
130 c$(5)= "ZYZZZYZZZZZZZZZYZZZY"
140 c$(6)= "ZZZZZZZZZZZZZZZZZZZZZZ"
150 c$(7)= "ZZZZZZZZZZZZZZZZZZZZZZ"
160 c$(8)= "ZYZZZZZZZYZZZZZZZZZY"
170 c$(9)= "ZYZ      ZYZ      ZYZ"
190 c$(10)="ZYZ      ZYZ      ZYZ"
195 c$(11)="ZYZZZZZZZYZZZZZZZZZY"
200 c$(12)="ZZZZZZZZZZZZZZZZZZZZ"
210 c$(13)="ZZZZZZZZZZZZZZZZZZZZ"
220 c$(14)="ZYZZZYZZZZZZZZZYZZZY"
230 c$(15)="ZZZZZZZZZYZZZZZZZZZY"
240 c$(16)="ZZZZZZZZZYZZZZZZZZZY"
250 c$(17)="ZYZZZZZYZZZYZZZZZZZY"
260 c$(18)="ZZZZZZZZZZZZZZZZZZZZ"
270 c$(19)="ZZZZZZZZZZZZZZZZZZZZ"
271 c$(20)="ZZZZZZZZZZZZZZZZZZZZ"
272 c$(21)=" "

```

```

273 c$(22)=""
274 c$(23)=""
300 'imprimimos el layout
310 FOR i=0 TO 20:|LAYOUT,i,0,@c$(i):NEXT
311 locate 1,1:pen 9:print "DEMO SOBREESCRITURA"
312 locate 3,23:pen 11:print "BASIC usando 8BP"
320 |SETUPSP,0,0,&x01000111:' deteccion colision con sprites y layout
330 |SETUPSP,0,7,1:dir=1: ' secuencia = 1 (coco derecha)
340 xa=20*2:xn=xa:ya=12*8:yn=ya:' coordenadas del personaje
350 |LOCATESP,0,yn,xa: 'colocamos al personaje (sin imprimirlo)
360 |PRINTSPALL,0,1,0:' imprime sprites
361 cl%=0:' variable colision
362 |COLAY,89,0,cl%:'umbral chr$("Y") es 89
400 ' COMIENZA EL JUEGO
401 |MUSIC,0,5
402' lectura teclado y colisiones.
403' 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+1:poke 27003,xn:|COLAY,0,@cl%: 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,0,@cl%: 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,0,@cl%: 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,0,@cl%: 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 : 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

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

Imagínate que solo hay un tipo de “ladrillo” en una pantalla (un ladrillo o ausencia del mismo). Esto se puede representar con un solo bit, de modo que en un byte caben 8 ladrillos. Puesto que no podemos escribir todos los códigos ASCII porque muchos de ellos son de control, al menos puedes usar la mitad. Ello “compactaría” la pantalla en una proporción de 1:7 (en el espacio de 10 pantallas podrás meter 70 pantallas) y simplemente antes de invocar a |LAYOUT deberás hacer la conversión entre tus bits (que estarán en forma de caracteres) y los caracteres que espera recibir el comando. Esa conversión la harías en BASIC y aunque sea lenta hablamos de imprimir la pantalla, lo cual no requiere excesiva velocidad.

La misma filosofía se puede aplicar si solo hay 4 tipos de ladrillos (dos tipos y ausencia). Puedes usar solo 2 bits por ladrillo y luego meter esos bits en caracteres (de 1 a 128), por lo que te caben 3.5 ladrillos por carácter. En ese caso la proporción de ahorro es 2:7 es decir, de 3.5 veces menos. En el espacio de 10 pantallas podrás meter 35 pantallas.

Otra solución sencilla y eficaz consiste en definir cada layout con menos información de la necesaria, “reduciendo” posibilidades. Sería algo así como definir el layout con menos resolución de la que tiene. Por ejemplo, podemos pensar en definir un layout (que mide 20x25) con una matriz de 5x6, de este modo:

```

NNxNN
xxxxx
xxxMM
Mxxxx
xxMMx
xxxxx
MMxxM

```

Cada carácter de este “microlayout” puede a continuación “expandirse” a algo que hayas predefinido, por ejemplo una “M” podría representar 3 ladrillos con césped por encima y las “x” podrían representar 3x2 espacios en blanco cada una. Tras expandir la definición de la pantalla podríamos tener un laberinto que ocupe todo el layout, aunque lógicamente hemos reducido las posibilidades del layout. Con esta sencilla estrategia

podemos almacenar 16 layouts “sencillos” en el espacio donde antes solo cabía un solo layout “complejo”. En resumen, hemos multiplicado 16 veces la capacidad de almacenamiento de layouts.

Puedes usar una estrategia menos “agresiva” y definir layouts simplemente en la mitad de espacio, por ejemplo en “microlayouts” de 10x12, que reducen menos las posibilidades, respecto al layout original de 20x25. Y estamos duplicando la capacidad de almacenamiento. Eso si, tendrás que programarte una rutina en BASIC que haga la conversión entre tus microlayouts y los “auténticos” layouts.

## 9 Recomendaciones y programación avanzada

### 9.1 Recomendaciones de velocidad

El intérprete 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 q;se encuentren en rangos validos (por ejemplo PEN 40 es ilegal) y muchas mas 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 numero de instrucciones posibles, aunque ello a veces implique escribir más instrucciones. El uso de GOTO es muy recomendable, dada su elevada velocidad de ejecución, como veremos mas adelante en una tabla comparativa.

Un factor decisivo a la hora de invocar un comando es el paso de parámetros. Cuantos mas parámetros tiene, mas 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 irremediabilmente.

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.

```
10 MEMORY 25999
20 DEFINT a-z
30 a!= TIME
40 FOR i=1 TO 1000
50 <aquí pones un comando, por ejemplo PRINT "A">
60 NEXT
70 b!=TIME
80 PRINT (b!-a!)
900 c!=1000/((b!-a!)*1/300)
100 PRINT c, "fps"
110 d!=c!/60
120 PRINT "puedes ejecutar ",d!, "comandos por barrido (1/50 seg)"
125 rem si dejas la linea 50 vacia , tardara 0.47 milisegundos
130 PRINT "el comando tarda ";((b!-a!)/300-0.47);"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. Hay que decir que es más rápido ejecutar una llamada directa a la dirección de memoria (un CALL &XXXX) que invocar el comando RSX. 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.

Comando	ms	Comentario
PRINT "A"	3.63	Lentísimo. Ni se te ocurra usarlo, salvo puntualmente para cambiar el numero de vidas pero no imprimas puntuación en un juego por cada enemigo que mates
LOCATE 1,1 PRINT puntos	24.87	Colocar el cursor de texto con LOCATE e imprimir el valor de una variable "puntos" es costosísimo. Si actualizas puntos hazlo sólo de vez en cuando y no en cada ciclo de juego
REM hola	0.20	Los comentarios consumen
' hola	0.25	Ahorras 2 bytes de memoria pero es mas lento!!
GOTO 60	0.196	Muy rápido!!! Más rápido incluso que REM. Usa este comando sin piedad, úsalo!!!
NOP	0.94 hasta 2.77	No hace nada, solo es un RET de ensamblador. Nos da una idea de lo que tarda el analizador sintáctico del BASIC. Este es el tiempo mínimo que va a tardar en ejecutarse cualquier comando RSX. El tiempo que tarda en procesarse un comando depende también de su posición en la lista de comandos RSX instalados. Si NOP fuese el primero de la lista, tardaría 0.94 pero si es el último tardará 2.77 La librería 8BP instala en primer lugar los comandos que más frecuentemente usarás en el ciclo de juego, como PRINTSPALL.
NOP,1,2,3	1ms mas que  NOP	El paso de parámetros es algo costoso. En pasar 3parametros se invierte 1ms
LOCATESP,i,y,x	2.8	Es muy rápida teniendo en cuenta la máxima velocidad alcanzable (2.15 es lo que tarda NOP con 3 parámetros) pero si no usas coordenadas negativas es mejor usar el comando BASIC POKE. Si quieres ubicar un escuadrón de enemigos es mejor usar AUTOALL y MOVERALL, ya que si por cada enemigo usas un LOCATESP invertirás demasiado tiempo.
POKE &XXXX, valor	0.71	Muy rápido! Úsalo para actualizar las coordenadas de los sprites
POKE d,valor	0.85	Muy rápido teniendo en cuenta que debe traducir la variable "d"
POKE,&xxxx,valor	2.5	Permite números negativos y si solo actualizas una coordenada (X o Y) es mejor que LOCATESP
X=PEEK(&xxxx)	0.93	Muy rápido!
X=INKEY(27)	1.12	Muy rápido. Apto para videojuegos aunque debes usarlo inteligentemente como se recomienda en este libro.
IF x>50 THEN x=0	1.42	Cada IF pesa, hay que tratar de ahorrarlos porque una lógica de juego va a tener muchos



If inkey(27)=1 then x=1	1.75	Buen uso combinado. Es más rápido que hacer b=INKEY(27) y después el IF...THEN
A=A+1: IF A>4 then A=0  Versus  A=A MOD 3 +1	2.6  Vs  1.84	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: $A=(A+1) \text{ MOD } 3$ nos cuesta 2 ms y sin embargo conseguimos lo mismo (mas o menos). Los paréntesis cuestan
:	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
PRINTSP,0,10,10	5.1	Un solo sprite de 14 x 24 . 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,0,0  (32 sprites 12x16)	59.3	Esto son unos 16,5fps a plena carga de sprites. Lo que tarda es  $T = 3.25 + N \times 1.77$  Es decir, unos 2ms 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 tomarian los valores de la ultima invocación), se ahorran 0.6ms en la parte fija, es decir:  $T = 2.6 + N \times 1.77$
PRINTSPALL,N,0,0 (ningún sprite activo)  N=0 N=10 N=31	2.6 4.3 5.9	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
COLAY,0,@x%	3.6	Aceptable. Usar solo con el personaje, no con los enemigos o el juego ira 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 mas tarda
CALL &XXXX,0,@x%	2.75	Es como invocar a  COLAY Es más rápido pero menos legible
GOSUB / RETURN	0.56	Aceptablemente rápido. La prueba es con una rutina que solo hace return.
SETUPSP, id, param, valor	3.5	Aceptable, aunque POKE es mucho mejor. Para ciertas cosas SETUPSP es mas legible y normalmente se va a usar poco en la lógica de modo que no es un problema

		usarlo (por ejemplo al cambiar de dirección un sprite). En el comando SETUPSP vemos el numero de sprite y parámetro que se toca. En POKE no se ve nada, es menos legible aunque mas rápido.
FOR / NEXT	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.
COLSP,0,@c%	5.6	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. Una estrategia como ejecutar  COLSP tan sólo la mitad de los frames no funcionaría porque requiere incrementar un contador y comprobarlo con un IF, por lo que aunque ahorras 2.5ms los gastarías en comprobaciones. 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
ANIMALL	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
AUTOALL	2.76	No es costosa y puede mover a la vez los 32 sprites
MOVERALL,1,1	3.4	No es muy costosa y puede mover a la vez los 32 sprites
SOUND	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. En cualquier caso si decides usarlo debe ser con sumo cuidado ya que consume mucho tiempo su ejecución (10 ms es muchísimo)
IF a>1 AND a>2 THEN a=2  Versus  IF a>1 THEN IF a>2 THEN a=2	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
A=RND*10	4.2	La funcion 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 asi. Otra solución sencilla es almacenar 10 numeros aleatorios en un array y utilizarlos en lugar de invocar a RND

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

Cuando hagas tu programa, trata de ver el coste de los comandos que usas y minimiza al máximo el uso de la CPU. Por ejemplo si puedes evitar pasar por un IF insertando un GOTO, siempre será preferible. O haz uso de POKE en lugar de LOCATESP a menos que uses coordenadas negativas. Cuando te falte velocidad y necesites un poquito más de rapidez utiliza CALL y abandona el uso de RSX.

Ten en cuenta que toda la lógica de tu programa debe acumular como mucho 20ms si quieres sincronizar con los barridos de pantalla, aunque la jugabilidad se mantendrá aceptable muy por encima, quizás hasta los 50ms, depende del tipo de juego. Es muy difícil que logres 20ms a menos que no haya apenas sprites. Pero un objetivo de 50ms es razonablemente rápido. Si tu juego tarda 50ms entonces generará 20fps (frames por segundo) y será muy aceptable. Y si consigues 40ms (25fps) incluso lograrás una suavidad profesional en los movimientos.

Más 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)
- Eliminar espacios en blanco
- 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
```

- Compactar en una línea todo aquello que sea lógica de juego y pueda ser compactado. Por ejemplo, las siguientes dos líneas:

```
10 if e1d=0 then e1x=e1x+1:if e1x>=70 then e1d=1:|SETUPSP,1,7,10
20 if e1d=1 then e1x=e1x-1:if e1x<=4 then e1d=0:|SETUPSP,1,7,9
```

Puedes cambiarlas por (fijate en el doble else para que aplique al primer if):

```
10 if e1d=0 then e1x=e1x+1:if e1x>=70 then e1d=1:|SETUPSP,1,7,10 else
else if e1d=1 then e1x=e1x-1:if e1x<=4 then e1d=0:|SETUPSP,1,7,9
```

- Evitar ejecución de líneas de lógica innecesarias. Esta recomendación es la misma que la anterior pero con un estilo más elegante de leer. Veamos un ejemplo. Con el ELSE 30 nos hemos evitado pasar por la línea 20 en muchos casos

En este ejemplo "e1d" es la dirección del sprite 1, "e1x" es su coordenada x

```

10 if e1d=0 then e1x=e1x+1:if e1x>=70 then e1d=1:|SETUPSP,1,7,10 else 30
20 if e1d=1 then e1x=e1x-1:if e1x<=4 then e1d=0:|SETUPSP,1,7,9
30 |LOCATESP,1,e1y,e1x

```

Y ahora una versión aun mejor. Hemos eliminado un IF innecesario en la linea 20

```

10 if e1d=0 then e1x=e1x+1:if e1x>=70 then e1d=1:|SETUPSP,1,7,10 else 30
20 e1x=e1x-1:if e1x<=4 then e1d=0:|SETUPSP,1,7,9
30 |LOCATESP,1,e1y,e1x

```

- En BASIC nunca sincronizar con el barrido de pantalla ya que ralentiza el juego. Es decir, usar siempre |PRINTSPALL,1,0 en lugar de |PRINTSPALL,1,1. Si compilas el juego con algún compilador como “fabacom” entonces merece la pena sincronizar, tanto para fijar la velocidad de juego a 50 frames por segundo como para conseguir mayor suavidad en los movimientos.
- Una vez que hayas invocado con parámetros al comando STARS o al comando PRINTSPALL, 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.
- Gestiona el teclado (y en general esto es aplicable a cualquier cosa que hagas) ejecutando el menor numero 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 mas rápida la segunda

Mal hecho (caso peor = 8 ejecuciones “IF INKEY”)

```

1671 IF INKEY(27)=0 and INKEY(67)=0 THEN IF dir<>2 THEN
|SETUPSP,0,7,2:dir=2:goto 1746 ELSE |ANIMA,0:xn=xa+1:yn=ya-2:goto 1746

1672 IF INKEY(27)=0 and INKEY(69)=0 THEN IF dir<>8 THEN
|SETUPSP,0,7,8:dir=8:goto 1746 ELSE |ANIMA,0:xn=xa+1:yn=ya+2:goto 1746

1673 IF INKEY(34)=0 and INKEY(67)=0 THEN IF dir<>4 THEN
|SETUPSP,0,7,4:dir=4:goto 1746 ELSE |ANIMA,0:xn=xa-1:yn=ya-2:goto 1746

1674 IF INKEY(34)=0 and INKEY(69)=0 THEN IF dir<>6 THEN
|SETUPSP,0,7,6:dir=6:goto 1746 ELSE |ANIMA,0:xn=xa-1:yn=ya+2:goto 1746

1675 IF INKEY(27)=0 THEN IF dir<>1 THEN |SETUPSP,0,7,1:dir=1:goto 1746
ELSE |ANIMA,0:xn=xa+1:goto 1746

1676 IF INKEY(34)=0 THEN IF dir<>5 THEN |SETUPSP,0,7,5:dir=5:goto 1746
ELSE |ANIMA,0:xn=xa-1:goto 1746

1677 IF INKEY(67)=0 THEN IF dir<>3 THEN |SETUPSP,0,7,3:dir=3:goto 1746
ELSE |ANIMA,0:yn=ya-4:goto 1746

```

```
1678 IF INKEY(69)=0 THEN IF dir<>7 THEN |SETUPSP,0,7,7:dir=7:goto 1746
ELSE |ANIMA,0:yn=ya+4:goto 1746
```

Bien hecho (caso peor = 4 ejecuciones "IF INKEY"):

```
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
```

- En juegos donde la lógica de los enemigos requiere del uso del calculo 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 en BASIC
- En juegos de naves es importante que no uses coordenadas negativas y si quieres que las naves aparezcan por los bordes y se perciba el clipping, reduce la pantalla un poco con SETLIMITS.
- En juegos de naves evita las comprobaciones con el layout. Esto supone casi 3.5ms de ahorro en cada ciclo de la lógica. Normalmente no requerirás de un layout en un juego de naves. Los escenarios los puedes simular con sprites que tengan desactivado el flag de colisión y que representen cráteres, bases espaciales, etc y junto con el suelo moteado con |STARS dará la sensación de tierra que se desplaza. 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.

- Usa el menor número de parámetros en las invocaciones a los comandos de 8BP. cada parámetro consume tiempo y cada milisegundo es importante en un juego, sobre todo de arcade.
- 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
- Simplifica: 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. Ello hará que parezca que es listo (solo es una idea)
- No uses coordenadas negativas si necesitas actualizar la posición de tu nave o personaje de forma muy rápida. El comando POKE (el de BASIC) es muy veloz pero solo soporta números positivos, al igual que PEEK. en caso de usarlas, usa |POKE y |PEEK (comandos de 8BP). Reserva el uso de |LOCATESP para cuando vayas a modificar ambas coordenadas y puedan ser positivas y negativas.
- 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 ...
- Al programar un disparo múltiple (una nave que puede disparar 3 proyectiles simultáneamente) usa la técnica de lógicas masivas y reduce la lógica. Si en cada fotograma solo puede morir un enemigo, reducirás mucho el número de instrucciones a ejecutar. Usa el comando |COLSPALL, el cual te permitirá programar más eficientemente.
- 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 ganarás colorido. Usala 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 Ejecución alternada y periódica

No es necesario que ejecutes todas las tareas en cada ciclo de juego. Por ejemplo si quieres 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” que te contaré en un apartado posterior.

Hay dos técnicas básicas para esto. El uso de aritmética modular y las operaciones lógicas

### 9.2.1 Aritmética modular

tecnica	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.

### 9.2.2 Operaciones binarias

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 numero 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 15 veces)
If ciclo AND 1=0 then gosub rutina	1.6ms (Se ejecuta una de cada 2 veces)

Si tienes varias cosas periodicas a ejecutar puedes hacerlo asi:

c=ciclo and 15

if c=0 then GOSUB <rutina1> ( se ejecuta “rutina1” una vez cada 15 veces)

if c=7 then GOSUB <rutina2>... ( rutina2 se ejecuta una vez cada 15 veces, pero alejado en el tiempo de la ejecución de la rutina1)

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

### 9.3 Aprovecha al máximo la memoria

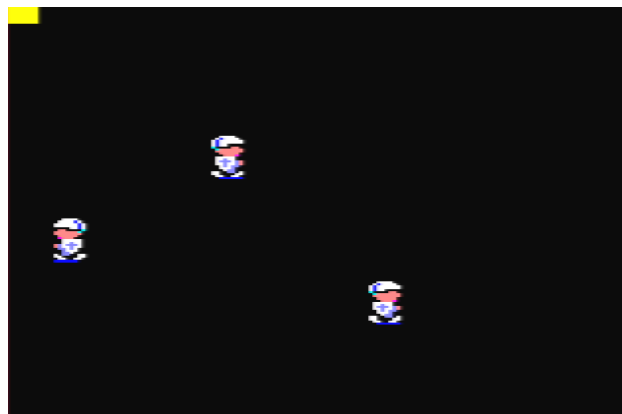
Cada pantalla ocupa una memoria considerable, si se trata de un juego de pantallas definidas a través del layout. En un juego sencillo, cada pantalla ocupará un texto de BASIC de alrededor de 800 bytes y un código de unos 2000 bytes, por lo que es difícil hacer un juego de más de 10 pantallas (recuerda que dispones de 26KB para tu juego)

Una de las cosas fundamentales que debes hacer es reutilizar código de lógica de enemigos en lugar de describirlo en cada pantalla.

La lógica de los enemigos de cada pantalla te puede ocupar más que el layout de la pantalla. En el juego “el mutante montoya” (realizado en BASIC usando 8BP), aproximadamente un tercio de cada pantalla es layout y los otros dos tercios son lógica. Si necesitas espacio para más pantallas lo más adecuado es “reutilizar” lógicas de enemigos entre pantallas, de un modo parametrizado. Por ejemplo un enemigo que se mueve de izquierda a derecha solo requiere 3 parámetros: donde comenzar (X, Y), donde termina su trayectoria por la derecha (X máxima) y donde termina por la izquierda (Xmin). Solo tendrías que inicializar estos 3 valores y con un GOSUB/RETURN ahorrarías líneas de BASIC en cada pantalla donde aparezca ese enemigo.

La clave es reutilizar código BASIC de unas pantallas en otras, igual que lo haces con la rutina de movimiento del personaje. Y dentro de cada pantalla si hay varios enemigos del mismo tipo, reutilizar el código de logica del enemigo, escribiéndolo sólo una vez.

Ejemplo: tres soldados paseándose por la pantalla



*Fig. 32 tres soldados y una única rutina de lógica*



```

10 MEMORY 25999
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)
50 x(1)=10:xmin(1)=10:xmax(1)=60:
y(1)=60:direccion(1)=0:|SETUPSP,1,7,9 :|SETUPSP,1,0,&x111
60 x(2)=20:xmin(1)=15:xmax(2)=40:
y(2)=100:direccion(2)=1:|SETUPSP,2,7,10 :|SETUPSP,2,0,&x111
70 x(3)=30:xmin(1)=5:xmax(3)=50:
y(3)=130:direccion(3)=0:|SETUPSP,3,7,9 :|SETUPSP,3,0,&x111
80 for i=1 to 3:|LOCATESP,i,y(i),x(i):next: 'colocamos los sprites

89 '----- BUCLE PRINCIPAL DEL JUEGO (CICLO DE JUEGO)-----
90 for i=1 to 3:gosub 100:next:'llamada a los 3 soldados
91 |PRINTSPALL,1,0: ' anima e imprime los 3 soldados
95 goto 90
96 '----- FIN DEL CICLO DE JUEGO -----
99 '----- rutina de soldado -----
100 IF direccion(i)=0 THEN x(i)=x(i)+1:IF x(i)>=xmax(i) THEN
direccion(i)=1:|SETUPSP,i,7,10 ELSE 120
110 x(i)=x(i)-1:IF x(i)<=xmin(i) THEN direccion(i)=0:|SETUPSP,i,7,9
120 poke 27003+i*16, x(i):' le colocamos en la nueva coordenada
130 return

```

Y como recomendaciones generales:

- Puedes superar las limitaciones de memoria mediante algoritmos que generen laberintos, o pantallas sin necesidad de almacenarlos. De este modo podrás hacer muchas mas pantallas. Esto requiere creatividad, desde luego, pero es posible.
- Puedes reutilizar la lógica de enemigos de una pantalla en otra, ahorrando muchas líneas de código. Aprovecha el mecanismo GOSUB/RETURN para ello, definiendo lógicas comunes configurables por parámetros. Por ejemplo un guardián que se mueve de izquierda a derecha puedes situarlo en muchas pantallas a diferentes alturas y con diferentes limites en su trayectoria sin programarlo de nuevo.
- 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)
- Intenta combinar layouts de pantallas consecutivas. Puede que las variaciones de una pantalla a la siguiente sean solo 10 líneas del layout (por ejemplo), ahorrando el 50% de la memoria para construirla (unos 400bytes de ahorro).
- Si quieres hacer muchas pantallas, compáctalas mediante la técnica explicada en el apartado 8.4

## 9.4 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 |MOVERALL 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 dos opciones:

- Usar una sola lógica que afecta a muchos sprites a la vez (usando los flag de movimiento automatico y/o relativo)
- Usar diferentes lógicas pero ejecutar solo una o unas pocas en cada ciclo del juego.

Ambas opciones 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 logica o logicas ejecutar en cada ciclo. En el caso mas sencillo, si tenemos N sprites simplemente ejecutaremos una de las N logicas. Pero en casos más complejos deberemos ser astutos para determinar que lógicas conviene ejecutar.

### 9.4.1 Ejemplo sencillo de lógica masiva

Volvamos al ejemplo de los 3 soldados. Esta vez 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 25999
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

```

Pruébalo y comprobarás que se ha multiplicando la velocidad casi por 3. 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 mas 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.

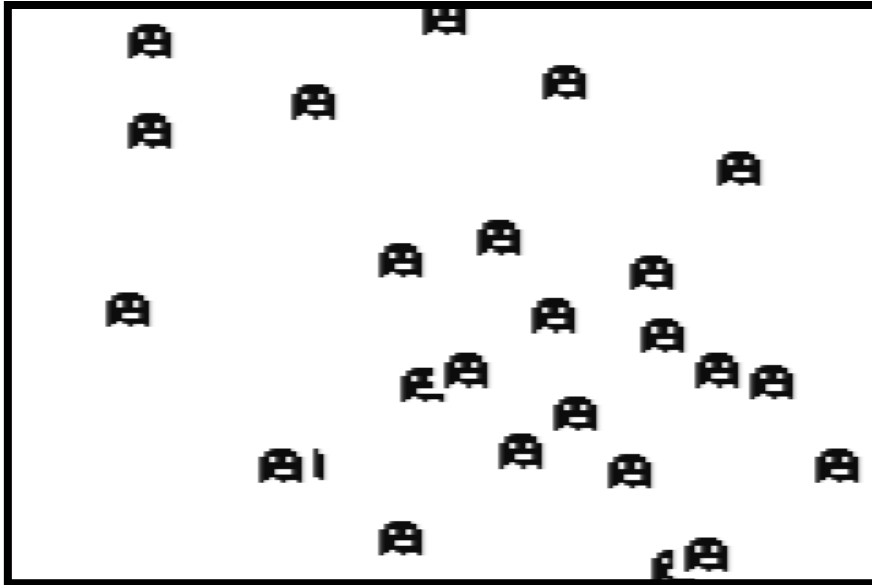
Imagínate un sprite que está en la posición  $x=20$  y queremos que se mueva hasta la posición  $x=30$  y dar la vuelta. Consideremos que el sprite tiene un movimiento automático con  $V_x=1$ . En ese caso comprobaremos su posición cuando  $x=20$ ,  $x=28$ ,  $x=36$ . Al llegar a 36 nos daremos cuenta de que nos hemos pasado!!! y cambiaremos la velocidad del sprite a  $V_x=-1$

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  $X_{final} = X_{inicial} + n*8$ .

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.

### 9.4.2 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. 33 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  $N \times 3$  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 25999: 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

```

### 9.4.3 Técnica de lógicas masivas en juegos tipo “pacman”

Si tienes muchos enemigos y deben tomar decisiones en cada bifurcación de un laberinto, quizás pienses que la técnica de lógica masiva no es precisa pues cada enemigo no comprueba su posición en cada ciclo de juego, pero esto se puede solucionar con un sencillo “truco”. Es simplemente colocar a los enemigos en posiciones bien escogidas al empezar el juego.

Supongamos que tienes 8 enemigos y que las bifurcaciones del laberinto ocurren en múltiplos de 8.

Si el primer enemigo está en una posición múltiplo de 8 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 8 y así poder decidir que camino tomar en la bifurcación, simplemente empezamos el juego con este segundo enemigo colocado en un múltiplo de 8 menos uno. Considerando coordenadas que comienzan en cero, los múltiplos de 8 son:

Primer enemigo: posición 0 o 8 o 16 o 24 o 32 o XX (en eje x o y, da igual)

Segundo enemigo: posición 7 o 15 o 23 ...

Tercer enemigo : posición 6 o 14 o 22 ...

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

Posición = múltiplo de 8 – n, siendo n el numero de sprite

Y cada vez que le toque a un enemigo ejecutar su lógica, podrá encontrarse en una bifurcación. Eso no significa que no deba comprobar que no se encuentra en una posición en mitad de un pasillo sin bifurcaciones. Debe comprobarlo y para ello puede usar PEEK contra un carácter del layout, ya que su coordenada dividida entre 8 es precisamente una posición del layout. El PEEK es muy rápido (unos 0.9 ms) frente a la detección de colisión que consume mas de 3ms. Con PEEK compruebas si la posición superior esta ocupada por un ladrillo (por ejemplo) y si hay via libre entonces el enemigo podría tomar esa nueva dirección.

¿ y si solo tienes 5 enemigos? Pues muy fácil. Hay ciclos que puedes simplemente no ejecutar ninguna lógica y así los cinco siempre toman decisiones al encontrarse en posiciones de bifurcación, múltiplos de 8.

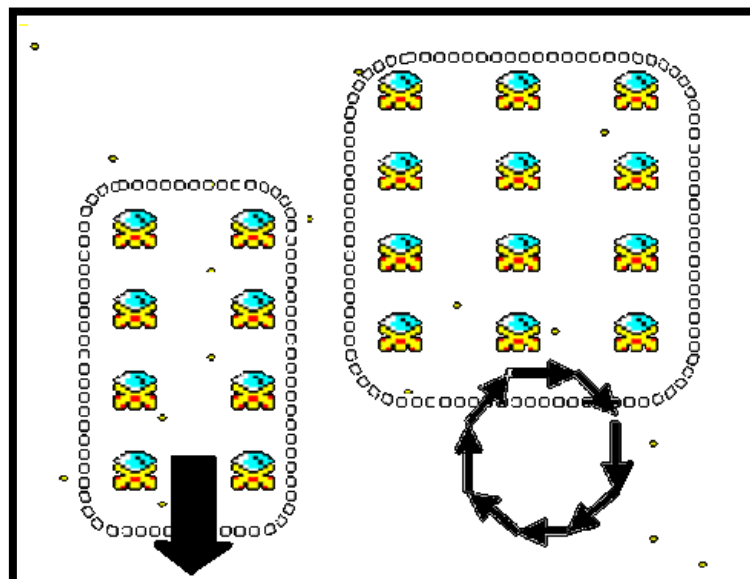
#### 9.4.4 Movimiento “en bloque” de escuadrones

Si lo que quieres 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 tien un parámetro opcional para invocar internamente a |ROUTEALL antes de mover a los sprites
- Si usas |MOVERALL tienes que activar el bit 5 del byte de estado a los sprites que vayas a mover. Este comando requiere que como parámetros introduzcas cuanto movimiento relativo en Y y en X desees

El uso combinado de ambas estrategias te puede permitir mover dos escuadrones con trayectorias complejas y diferentes entre si, veamos un ejemplo

En este videojuego hemos definido el escuadrón de la derecha con un movimiento relativo (usando MOVERALL), siguiendo una trayectoria circular almacenada en un array. El grupo de la izquierda tiene el flag de movimiento relativo desactivado pero tiene activado el flag de movimiento automático, y usando AUTOALL se desplazan hacia abajo pues todos ellos tienen Vy=2



*Fig. 34 Movimiento de escuadrones*

Aquí tienes el listado del ejemplo

```
1 MODE 0
10 MEMORY 25999
20 DEFINT a-z
25 REM la ruta de movimiento relativo la almacenamos en los arrays ry,
rx
30 DIM rx(24):DIM ry(24)
40 rx(0)=1:ry(0)=2
50 rx(1)=1:ry(1)=2
60 rx(2)=1:ry(2)=2
```

```

70 rx(3)=0:ry(3)=2
80 rx(4)=0:ry(4)=2
90 rx(5)=0:ry(5)=2
100 rx(6)=-1:ry(6)=2
110 rx(7)=-1:ry(7)=2
120 rx(8)=-1:ry(8)=2
130 rx(9)=-1:ry(9)=0
140 rx(10)=-1:ry(10)=0
150 rx(11)=-1:ry(11)=0
160 rx(12)=-1:ry(12)=-2
170 rx(13)=-1:ry(13)=-2
180 rx(14)=-1:ry(14)=-2
190 rx(15)=0:ry(15)=-2
200 rx(16)=0:ry(16)=-2
201 rx(17)=0:ry(17)=-2
202 rx(18)=1:ry(18)=-2
203 rx(19)=1:ry(19)=-2
204 rx(20)=1:ry(20)=-2
205 rx(21)=1:ry(21)=0
206 rx(22)=1:ry(22)=0
207 rx(23)=1:ry(23)=0
210 rem -----inicializamos los escuadrones -----
220 FOR j=0 TO 31:|SETUPSP,j,0,&X0:NEXT
230 FOR j=0 TO 16 STEP 4
240 FOR i=j TO j+3
250 |SETUPSP,i,0,&X10111:|SETUPSP,i,7,14 : |SETUPSP,i,6,1
:|SETUPSP,i,5,1
260 |LOCATESP,i,10+(i-j)*28,3*j+10
270 NEXT
280 NEXT j
281 rem inicializamos el segundo escuadron con movimiento automatico
282 for i=0 to 7 : |SETUPSP,i,0,&X01111: |SETUPSP,i,5,2:
|SETUPSP,i,6,0:next

310 rem -----bucle de logica del programa -----
420 |PRINTSPALL,0,0
421 |AUTOALL
430 |MOVERALL,ry(t),rx(t)
431 |STARS,0,20,1,3,0
432 rem aqui cambiamos el indice de la ruta
440 t=t+1 : IF t=24 THEN t=0
470 GOTO 420

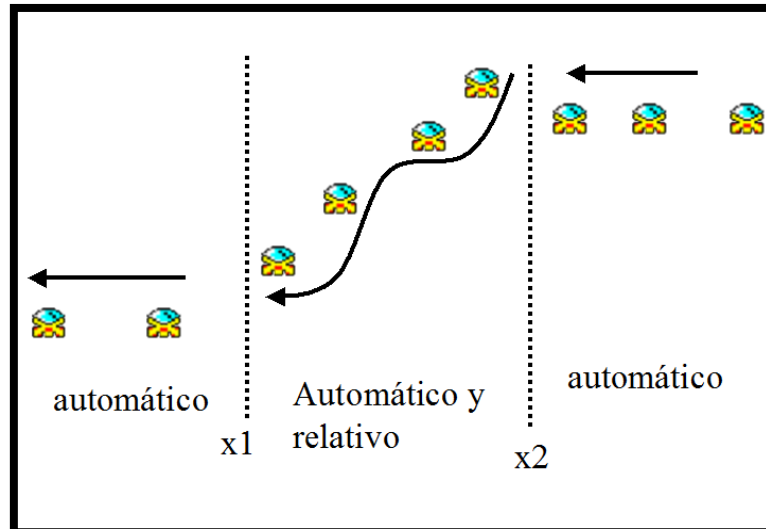
```

En el ejemplo anterior todas las naves se mueven “en bloque”. En cualquier momento una de ellas puede desactivar su flag de movimiento relativo y/o automatico y tomar vida propia, con una lógica específica individual que le haga volar hacia nuestra nave y atacarnos.

Pero si lo que deseas es algo mas avanzado, como hacer que un grupo de naves no se muevan “en bloque” sino que sigan una trayectoria en fila “india”, de modo que la nave que está en cabeza hace unos movimientos que van imitando las demás, puedes usar estrategias como la que te voy a describir a continuación.

#### 9.4.5 Lógicas masivas: Movimiento “en fila india”

Un grupo de naves pueden tener un movimiento automático horizontal pero tienen una lógica de modo que cuando su coordenada x exceda de un cierto valor, pasan a tener movimiento automático y relativo a la vez. Y cuando pasan de otra coordenada x vuelven a desactivar el movimiento relativo



*Fig. 35 Trayectorias en fila india*

Esta sencilla estrategia te permite hacer el efecto de “fila india” de naves de un modo muy eficiente, sin actuar sobre las coordenadas de ninguna de forma individual.

La forma de hacerlo no es controlando la coordenada x de cada nave pues eso implicaría un bucle de comprobaciones muy lento. La forma rápida es mediante un contador de tiempo. En cada ciclo de juego se incrementa. Algo como lo que te muestro a continuación

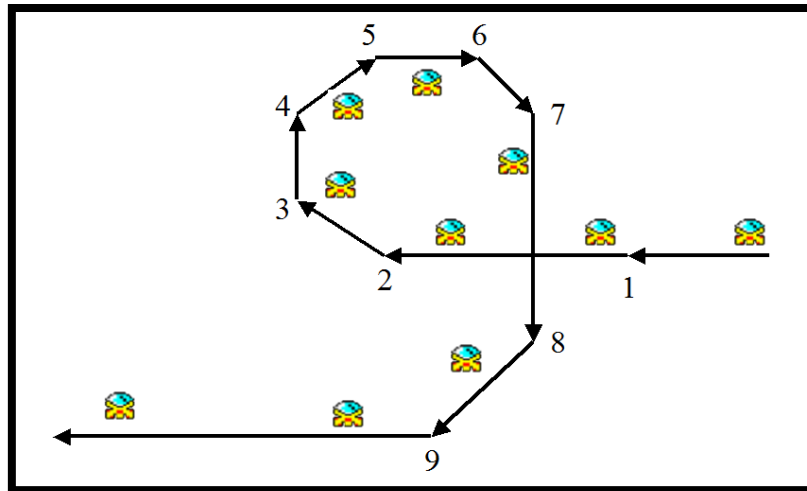
```
t=t+1: if t>= 20 and t<= 25 then |SETUPSP,t-20,0,&x10111|
```

Con esa línea dentro de la lógica del juego, cada vez que se ejecuta se incrementa el contador y cuando llega a 25, una a una, las naves correspondientes a los sprites 0,1,2,3,4 van cambiando de modo de funcionamiento. Cuando el contador llega a 30, habrán pasado las 5 naves al nuevo modo de comportamiento. El contador hace las veces de control de la coordenada x de todas las naves pero de un modo infinitamente más eficaz. Esta estrategia es en esencia no ejecutar la lógica de cada nave en cada ciclo, sino tan solo la lógica de la nave que interesa ejecutar en cada momento. Imagina lo costoso que sería hacer la comprobación de la coordenada x en 8 naves cada ciclo. De acuerdo a la tabla de rendimiento, un IF se toma 1.42ms y por lo tanto 8 naves como mínimo se tomarían unos 12 ms para hacer lo mismo que hemos hecho con un solo IF aplicado al contador de tiempo.

#### 9.4.6 Lógicas masivas: Trayectorias complejas

Si lo que deseas es algo aun mas complejo, donde las naves describan un círculo en fila india, deberás basarte sólo en movimiento automático. Piensa en la siguiente figura





*Fig. 36 Puntos de control de las trayectorias*

Con un solo contador y nueve puntos de control puedes ir cambiando una a una las  $V_y, V_x$  de cada nave, de modo que en cada ciclo de tu lógica solo se actúa como mucho sobre una nave

Por ejemplo el punto de control “2” sería algo como

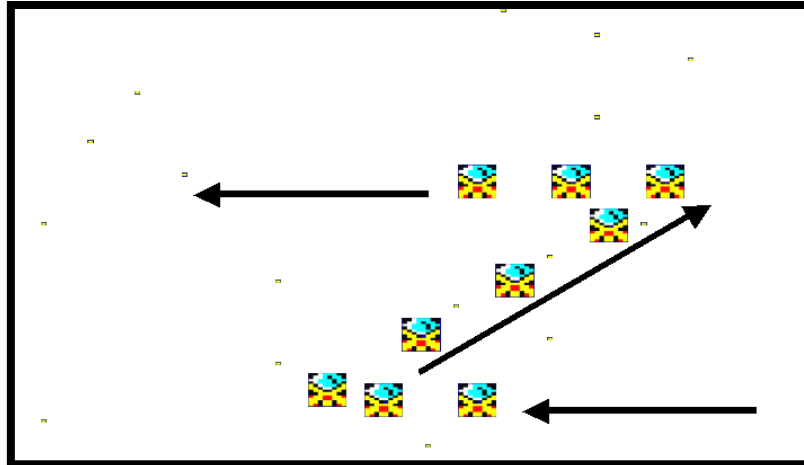
$t=t+1$ :if  $t \geq 15$  and  $t \leq 20$  then |SETUPSP, $t-15,5,-1$ :|SETUPSP, $t-15,5,-1$

Establecer 9 puntos de control para el contador supone establecer 9 sentencias IF por las que pasa el programa principal pero puedes utilizar estrategias para saltar con GOTO, tales como

if  $t < 50$  goto primer grupo de IF  
 if  $t < 100$  goto segundo grupo  
 if  $t < 100$  goto tercer grupo  
 etc

De esta manera con solo 3 IF en cada ciclo del juego podrás controlar los 9 puntos de control.

Vamos a ver un ejemplo simplificado. El siguiente ejemplo funciona muy suave con un escuadrón de 8 naves. Tiene sólo 2 puntos de control pero es esencialmente la misma estrategia. He utilizado dos contadores para evitar que las naves cambien de modo de funcionamiento demasiado seguido. El contador que las hace cambiar es “t” y solo avanza cada vez que el contador “tt” cuenta hasta 10, momento en el que “tt” vuelve a comenzar. El número 10 es precisamente la separación entre sprites considerando la coordenada x.



*Fig. 37 Trayectorias complejas con lógicas masivas*

```

1 MODE 0
10 MEMORY 25999
20 DEFINT a-z
100 rem inicializacion-----
220 FOR j=0 TO 31:|SETUPSP,j,0,&X0:NEXT
221 FOR j=0 TO 7:
222 |SETUPSP,j,0,&X1111:|setupsp,j,7,14:
223 |setupsp,j,5,0:|setupsp,j,6,-1:
224 |locatesp,j,150,40+j*10:
225 NEXT

400 t=0:tt=0:rem bucle de logica juego-----
420 |PRINTSPALL,0,0
421 |autoall
431 |STARS,0,20,1,0,-2
432 tt=tt+1: if tt=10 then tt=0:gosub 1000
441 rem primer tramo (sin logica)
442 goto 420

499 rem segundo tramo-----
500 |setupsp,t-1,5,-2:|setupsp,t-1,6,1:
510 return
599 rem tramo 3-----
600 |setupsp,t-5,5,0:|setupsp,t-5,6,-1:
610 return
999 rutina de contador c-----
1000 t=t+1:if t<=8 then gosub 500
1010 if t >4 and t<=13 then gosub 600
1020 return

```

Vamos a ver otro ejemplo para clarificar aun más el concepto de utilizar un contador de tiempo como mecanismo para alterar los comportamientos de sprites, sin que haya que ejecutar lógica independiente para cada uno.

En este ejemplo los 8 enemigos empiezan dispuestos en dos diagonales de 4 cada una. Cada grupo de 4 avanza en dirección contraria y a medida que llegan al final cambian de dirección, produciendo un efecto de entrelazamiento de trayectorias muy atractivo

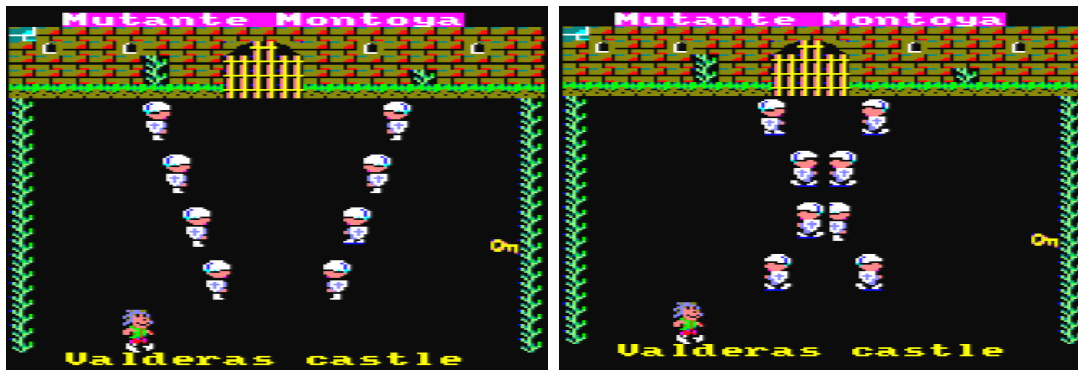


Fig. 38 uso de Logicas masivas en el “Mutante Montoya”

En esta lógica, cada 4 ciclos del bucle principal (controlados por el contador cc) se ejecuta el incremento del contador c. Y cuando c se encuentra entre 10 y 15 (ciclos 40 a 60) se van alterando el comportamiento de los soldados que se encuentran colocados en diagonales opuestas. Solo se altera el comportamiento de 2 soldados a la vez. Transcurrido un tiempo correspondiente a  $c=20$  (80 unidades de tiempo) se empiezan a dar la vuelta otra vez.

El bucle principal controla también la colisión entre sprites y la salida de la pantalla cuando la coordenada Y del personaje (controlada por “yn”) es menor que 26

```

4401 c1=10:c2=20:c=0:cc=0
4409 '----- bucle de logica principal -----
4410 GOSUB 1500: ' control del personaje
4421 cc=cc+1: if cc=4 then cc=0: gosub 4700
4609 |AUTOALL : ' movimiento automatico de 8 sprites
4610 |PRINTSPALL,1,0 ' impresion masiva de sprites (9= 8+personaje)
4611 |COLSP,0,@cs%:IF cs%<32 THEN GOSUB 600:GOTO 4020
4612 IF yn<26 THEN RETURN
4620 GOTO 4410
4699 '----- rutinas de control del escuadron ----
4700 c=c+1:if c>c1 and c<=c1+4 then gosub 4800
4701 if c>c2 and c<=c2+4 then gosub 4900
4702 if c=30 then c=10
4710 return
4799 '----- a darse la vuelta de 2 en 2 -----
4800 |setupsp,c1+5-c,7,10:|setupsp,c1+5-c,6,-1
4801 |setupsp,c1+5-c+4,7,9:|setupsp,c1+5-c+4,6,1
4810 return
4899 '----- a darse la vuelta otra vez -----
4900 |setupsp,c2+5-c,7,9:|setupsp,c2+5-c,6,1
4901 |setupsp,c2+5-c+4,7,10:|setupsp,c2+5-c+4,6,-1
4910 return

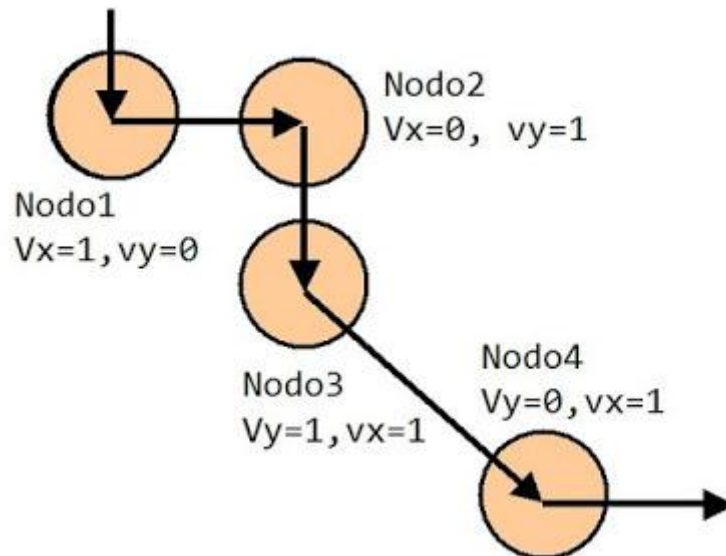
```

#### 9.4.7 Lógicas masivas: Controla el tiempo, no el espacio

Ahora vamos a ir un paso más lejos, llevando al límite esta técnica. En el ejemplo anterior hemos diferenciado los valores del contador de tiempo con varias sentencias “IF”, pero podemos hacerlo todo con una sola sentencia, mucho más eficiente en trayectorias.

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 caso.

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)



*Fig. 39 Trayectoria definida con "nodos de control"*

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. Y es que no es una estrategia eficiente computacionalmente. Puesto que estamos ante un escenario "**determinista**", podríamos 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")

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  $t(1)+70$

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 linea podemos ir cambiando las velocidades de cada nave a medida que van pasando cada una de ellas por el nodo 1. Durante los 8 instantes de tiempo posteriores a  $t(1)$  se van actualizando cada una de las naves, justo cuando pasan por el nodo de control

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 ultima 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.

```

j=j+1: IF j=10 THEN j=0: i=i+1: n=nmax
IF n<nmin THEN 50:' no hay que actualizar mas naves
IF i>=t(n) AND i<t(n)+8 THEN [actualiza i-t(n)]:IF i-t(n)=0 THEN
nmax=nmax+1: nmin=nmax-3
n=n-1

50 ' mas instrucciones del juego

```

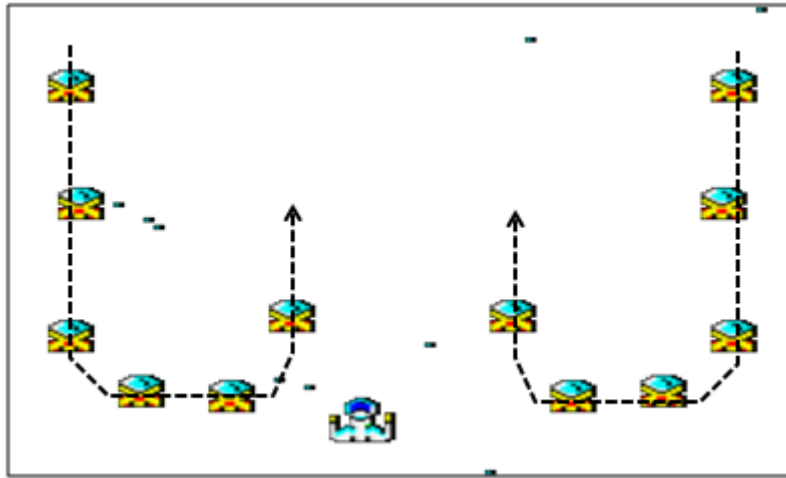
Como veis cuando se incrementa en 1 la decena de tiempo, se comprueban los nodos desde "nmax" hasta "nmin" Y en cada nodo actualizamos la nave  $i-t(n)$ . Pensad que si  $t(3)$  es, por ejemplo, 23, entonces cuando llegamos al nodo 3 en el instante  $i=23$ , actualizamos la nave  $i-23 = 0$ , con los valores de velocidad del nodo 3.

En el siguiente ciclo de juego comprobamos si aun sigue "vigente" el intervalo de tiempo del nodo 2, y actualizamos la nave 1 con los valores de velocidad del nodo 2,

y en el siguiente ciclo comprobamos el nodo 1 y si sigue vigente actualizamos la nave 2, y asi sucesivamente, siempre teniendo en cuenta que la primera nave puede haber llegado a nmax, mientras que la siguiente nave puede estar en  $nmax-1$  y la siguiente en  $nmax-2$ , etc.

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!

A continuación se incluye un fragmento del videojuego “annunaki” que utiliza esta técnica para manejar las trayectorias de dos hileras simétricas de 6 naves cada una



*Fig. 40 Dos hileras con lógicas masivas*

Inicialización de los puntos de control de la trayectoria. Con diferentes parámetros se consiguen diferentes trayectorias

```
2970'12 naves
2971 for i=0 to 12: k(i)=1000:next:kx(0)=-1:ky(0)=2:
2972 k(2)=1: kx(2)=0:ky(2)=5
2973 k(3)=4: kx(3)=-1:ky(3)=1
2974 k(4)=5: kx(4)=-1:ky(4)=-1
2975 k(5)=6: kx(5)=0:ky(5)=-5
2976 k(12)=20:finm=3:inim=2: m=inim: goto 3000
```

Lógica masiva que gestiona las dos hileras. En rojo he destacado la línea que gobierna el movimiento de las 12 naves

```
2999' fase 4 ----trayectorias en dos hileras simetricas configurables
por k(i), kx(i),ky(i). se separan de 10 en 10 en x -----
3000 totaln=12: blandos=31-totaln:duros=100: ini=31-
totaln:|COLSP,32,ini,30
3010 for i=ini to 24
3020
|SETUPSP,i,9,navemala1:|SETUPSP,i,7,0:|SETUPSP,i,5,ky(0):|SETUPSP,i,0,
&x01011:|SETUPSP,i,6,kx(0)
3021
|SETUPSP,i+6,9,navemala1:|SETUPSP,i+6,7,0:|SETUPSP,i+6,5,ky(0):|SETUPS
P,i+6,0,&x01011:|SETUPSP,i+6,6,-kx(0)
3030 |LOCATESP,i,-30-(i-ini)*20,80+(i-ini)*10:|LOCATESP,i+6,-30-(i-
ini)*20,0-(i-ini)*10-6:'6 es el ancho en bytes de la navemala y 26 el
alto
3040 |STARS:gosub 500:gosub
750:|AUTO,7:|AUTO,8:|AUTO,9:|PRINTSPALL,1,0
3050 next
3060 ciclo=0: t=0: col%=32:yd%=200
3100 gosub 500: gosub 750
```

```

3109 |AUTOALL:|PRINTSPALL:|STARS
3120 |COLSPALL: if sp<32 then if sp=31 then gosub 300:|MUSIC,0,5:goto
3000 else gosub 770
3130 ciclo=ciclo+1: if ciclo =10 then ciclo=0:t=t+1:m=inim
3131 if m=finm+1 then 3100
3132 IF t>=k(m) AND t<k(m)+6 THEN i= t-
k(m)+ini:|SETUPSP,i,5,ky(m),kx(m):|SETUPSP,i+6,5,ky(m),-kx(m):IF
m=finm THEN finm=m+1:inim=finm-3 ELSE ELSE IF t>=k(12) THEN RETURN
3133 m=m+1:goto 3100

```

## 9.5 Enrutar sprites con 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 y cada segmento tiene tres parámetros:

- Cuantos pasos vamos a dar en ese segmento
- Qué velocidad Vy se va a mantener durante el segmento
- Qué velocidad Vx se va a mantener durante el segmento

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 aquí 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
    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

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 automatico y el de animación y el de impresión.

```

10 MEMORY 25999
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
81 REM |LOCATESP,i,200*RND,80*RND
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.

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.



*Fig. 41 una ruta en forma de 8*



## 10 Juegos con scroll

La librería 8BP puede hacer scroll a partir de la versión V24, mediante el uso combinado de un “mapa del mundo” y una función llamada |MAP2SP, que explicaré al final de este capítulo.

Hay varias técnicas que puedes utilizar de forma independiente o combinada para hacer scroll. Vamos a ver varios ejemplos

### 10.1 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 profundidad.

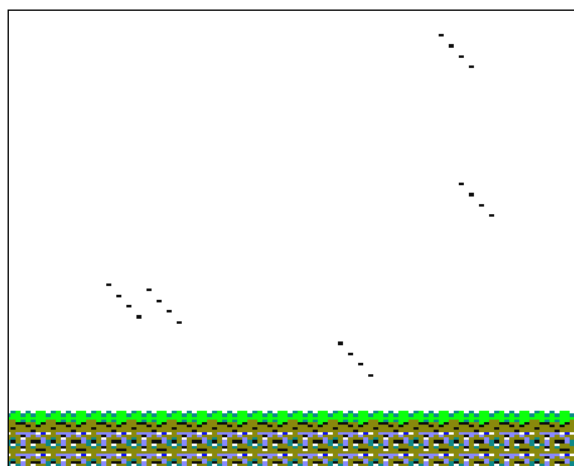
El banco consiste en 40 pares de bytes representando coordenadas (y,x). Ocupando desde la dirección 42540 hasta 42619 (son 80 bytes en total)

Una forma de generar 40 estrellas aleatorias sería

```
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 encontraras distintos ejemplos para simular estrellas, tierra, estrellas con dos planos de profundidad, lluvia o incluso nieve. Probablemente con imaginación sea posible simular mas 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, podras conseguir un desplazamiento de movimiento de “segmentos”, algo que podría ser ideal para simular lluvia.



*Fig. 42 Efecto de lluvia con STARS*

```
10 MEMORY 25999
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 'ESCENARIO DE LLUVIA
141 '-----
150 |SETLIMITS,0,80,50,200: ' limites de la pantalla de juego
151 cespel=&84d0:|SETUPSP,30,9,cespel:'letra Y es el sprite 31
152 rocas=&84f2:|SETUPSP,21,9,rocas: 'letra P es el sprite 21
160 cadena$="YYYYYYYYYYYYYYYYYYYY"
170 |LAYOUT,22,0,@cadena$:'esto pinta el cespel
180 cadena$="PPPPPPPPPPPPPPPPPPPPPP"
190 |LAYOUT,23,0,@cadena$:'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 capitulo 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. 43 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 25999
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: ' establecemos los 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

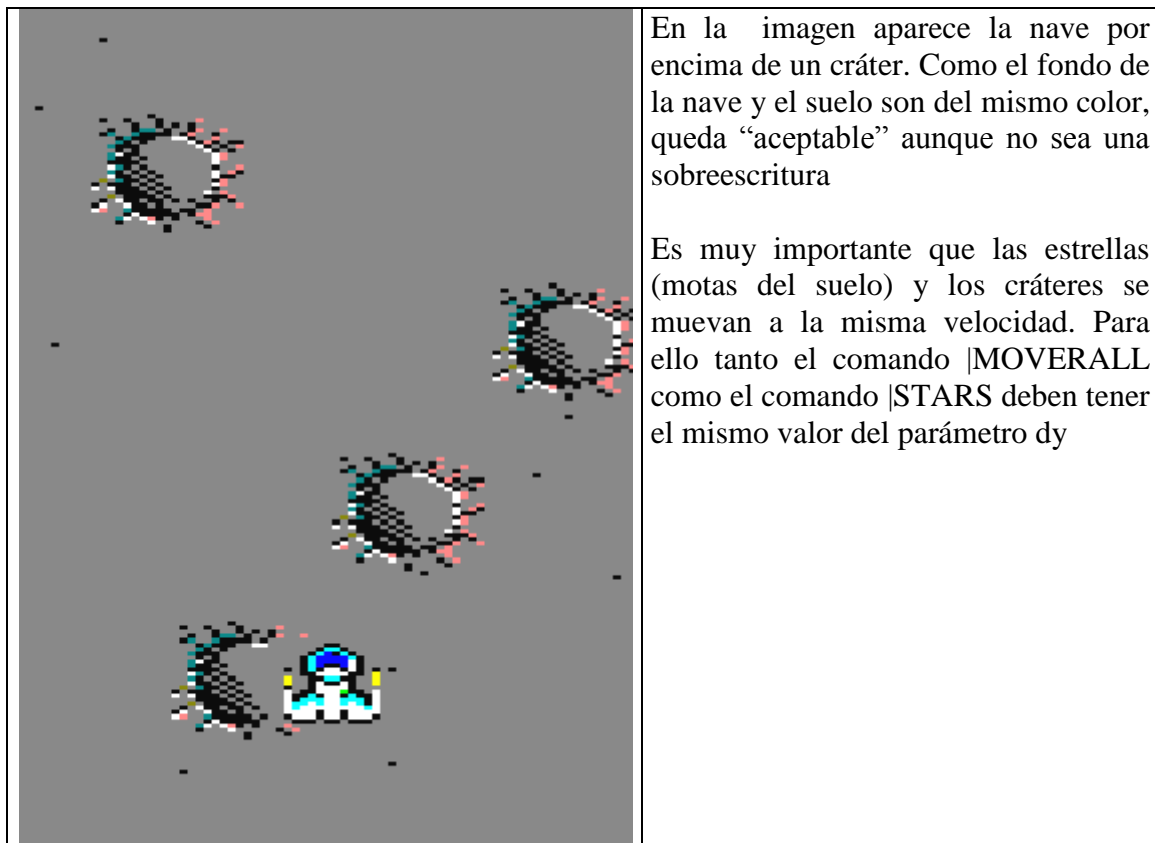
49'----- ciclo de juego-----
50 |STARS,0,20,5,1,0:' estrellas negras sobre suelo gris
55 gosub 100:' movimiento de la nave
60 |PRINTSPALL,0,0
70 goto 50

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
```

## 10.2 Cráteres en la luna

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 la luna, es decir, como simular un scroll de pantalla con estos elementos

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 crater es un sprite inferior a 31 se imprimirá antes que la nave y la nave quedará “por encima”, dando la sensación de que lo esta sobrevolando.



*Fig. 44 sobrevolando la luna*

Este es el código BASIC

```

10 MEMORY 25999
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: ' establecemos los 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
210 if c=6 then c=0
220 |PEEK,27001+c*16,@cy%
230 if cy%>200 then |POKE,27001+c*16,-20
240 return

```

En el siguiente ejemplo se ha puesto un escuadrón de naves espaciales, los crateres y tu nave espacial (9 sprites en total y las estrellas). Gracias a los comandos MOVERALL y AUTOALL, es posible mover todo esto a un ritmo adecuado para un juego de arcade



*Fig. 45 juegos de arcade con scroll usando 8BP*

```

10 MEMORY 25999
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: ' establecemos los 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,&X1001:|SETUPSP,i,5,3:|SETUPSP,i,6,0: ' impresion y
movimiento auto
50 x(i)=RND*40+20:y(i)=i*40
60 |LOCATESP,i,y(i),x(i)
70 NEXT
71 t=0
75 ' naves
76 FOR i=4 TO 7 :
|SETUPSP,i,7,14:|SETUPSP,i,0,&X10101:|LOCATESP,i,RND*20,i*10-20
77 NEXT :
78 inc=1
80 '----- ciclo de juego-----
81 |STARS,0,20,5,3,0:' movimiento estrellas negras
82 GOSUB 100:' movimiento de la nave
84 t=t+1: IF t> 10 THEN inc=-inc: t=0:GOSUB 200:' control de crateres
85 |MOVERALL,2,inc
86 |AUTOALL
90 |PRINTSPALL,0,0
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
210 IF c=4 THEN c=0
220 |PEEK,27001+c*16,@cy%
230 IF cy%>200 THEN |POKE,27001+c*16,-20
240 RETURN

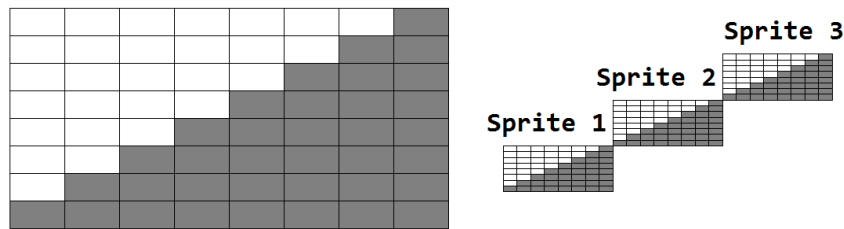
```



### 10.3 Montañas y lagos: técnica de “manchado”

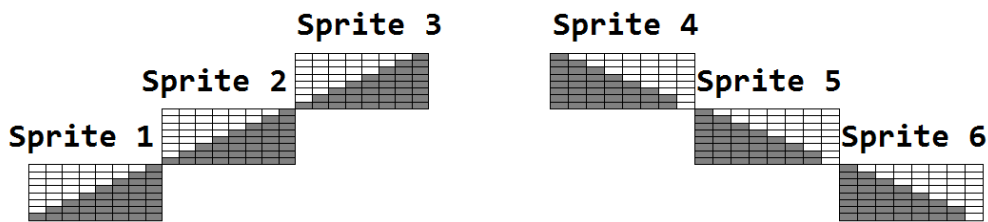
La técnica para pintar montañas en juegos con scroll horizontal y lagos en juegos con scroll vertical es la misma

Lo que haremos es pintar solo el comienzo de la montaña, mediante un sprite que nos sirve para pintar su lado derecho. Pondremos tantos como queramos. En este caso yo he puesto tres



*Fig. 46 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 izquierdo de la montaña. Cuidado de que la imagen espejada al menos tenga la ultima columna de pixels a cero. Esto le permitirá borrarse a si misma al avanzar a la izquierda.



*Fig. 47 Disposición de Sprites para construir una montaña*

Al mover todos los sprites hacia la izquierda mediante movimiento automatico 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.



*Fig. 48 un scroll horizontal de una montaña*

aquí tienes el ejemplo que lo ilustra

```
1 MODE 0
10 MEMORY 25999: CALL &6B78
20 DEFINIT a-z
30 FOR j=0 TO 31:|SETUPSP,j,0,&X0:NEXT
35' sprites de lateral derecho de la montana
40 FOR i=1 TO 3:|SETUPSP,i,9,&9102:|SETUPSP,i,6,-1:
|SETUPSP,i,0,&X1111:NEXT:'r
45 ' sprites para el lateral izquierdo de la montana
50 FOR i=7 TO 9:|SETUPSP,i,9,&9124:|SETUPSP,i,6,-1:
|SETUPSP,i,0,&X1111:NEXT:'l
55 x=80: inc=1
60 |LOCATESP,7,176,x:|LOCATESP,8,184,(x-4):|LOCATESP,9,192,(x-8)
70 |LOCATESP,1,176,x+20:|LOCATESP,2,184,(x+24):|LOCATESP,3,192,(x+28)
71' coloco 5 naves
75 FOR i=20 TO 24
:|SETUPSP,i,7,14:|SETUPSP,i,0,&X11101:|LOCATESP,i,RND*100,(i-
20)*10:NEXT :' relativo

79 ' logica principal
80 |PRINTSPALL,0,0
90 |AUTOALL
91 t=t+1: GOSUB 200
92 |STARS,0,20,1,0,-2
93 |MOVERALL,0,inc
94 tt=tt+1: GOSUB 300
100 GOTO 80

199' rutina que recoloca la montaña a la izquierda de la pantalla
200 IF t <140 THEN RETURN
210 x=100:|LOCATESP,7,176,x:|LOCATESP,8,184,(x-4):|LOCATESP,9,192,(x-
8)
220 |LOCATESP,1,176,x+20:|LOCATESP,2,184,(x+24):|LOCATESP,3,192,(x+28)
230 t=0:RETURN
300 IF tt=40 THEN inc=-inc: tt=0
310 RETURN
```

En el caso de un juego de scroll vertical, si queremos pintar un lago sobre un terreno marron, haremos lo mismo, unos sprites que van “manchando” el terreno y otros mas 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!

### **10.4 Tunel rocoso o carretera**

Una forma de hacer un túnel rocoso es simplemente concatenar los sprites que hacen de muro. En el siguiente ejemplo del videojuego “Anunnaki”, se utilizan 10 sprites en cada

lateral, de 24 pixels de alto cada uno y 8 de ancho. Aunque la velocidad del scroll no es trepidante, queda bastante decente.



*Fig. 49 Efecto túnel rocoso*

Periódicamente (mediante la siguiente instrucción) se invoca a la rutina que pinta el túnel rocoso. Se invoca cada 8 ciclos de juego porque los muros se desplazan con movimiento automatico con  $V_y=3$

```
6555 if ciclo mod 8=0 then gosub 6700

<...>

6699'logica de desfiladero
6700 des=des+1: if des=20 then des=10
6701 if x1 > 30 THEN img=rocall: |SETUPSP,des,9,img: |LOCATESP,des,-
24,x1:x1=x1-4:goto 6730
6702 if x1 < 0 THEN img=rocalr:x1=x1+4:
|SETUPSP,des,9,img:|LOCATESP,des,-24,x1:goto 6730
6703 azar=int(rnd*2)
6704 if azar=1 THEN img=rocall: |SETUPSP,des,9,img: |LOCATESP,des,-
24,x1:x1=x1-4:goto 6730
6705 img=rocalr:|SETUPSP,des,9,img:x1=x1+4:|LOCATESP,des,-24,x1
6710 ' ahora el lado derecho
6730 if xr<x1+30 then
img=rocarr:|SETUPSP,des+10,9,img:|LOCATESP,des+10,-24,xr:xr=xr+4:
return
6740 xr=xr-4:img=rocarl:|SETUPSP,des+10,9,img:|LOCATESP,des+10,-24,xr:
return
```

## **10.5 Caminando por el pueblo**

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 el. 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. 50 El pueblo entero se mueve*

```

10 MEMORY 25999
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 40

```

## 10.6 Scroll basado en un mapa del mundo

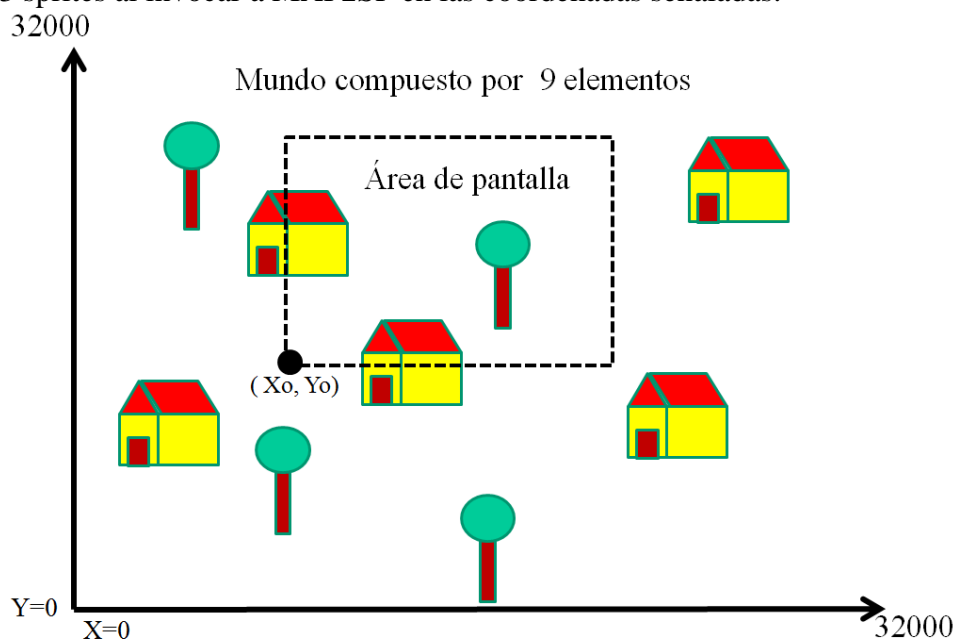
Todas las técnicas anteriores (comando STARS para hacer un suelo moteado, técnica de manchado, movimiento relativo de elementos) son perfectamente válidas para hacer scroll, e incluso compatibles con lo que vamos a ver ahora, que es la técnica fundamental que te va a permitir diseñar un “mapa del mundo” y hacer que tu personaje o tu nave se desplace por el, con tan solo una línea de código.

La idea es sencilla: crearemos una lista de elementos que conforman el mapa del mundo (hasta 64 elementos a los que llamaremos “elementos de mapa” o “map ítems”). Cada elemento esta descrito por las coordenadas 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 funcion analiza la lista de los 64 elementos y determine cuales 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 funcion, los map ítems que ya no estan presentes en la escena no consumiran sprites en la tabla, y otros map ítems tomarán el relevo. Esto significa que la funcion 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. 51 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.

Debes invocar MAP2SP en cada ciclo de juego o al menos cada vez que modifiques las coordenadas del punto de vista desde donde quieres visualizar el mundo.

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

### 10.6.1 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 las 64 entradas que definen las imagenes 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. Y 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 64 ítems pero el número de ítems se puede limitar con uno de los parámetros globales.

La lista ocupa los 5 bytes iniciales + 64 ítems x 6 bytes = 5+384=399 bytes

La tabla comienza con 3 parámetros:

- El alto máximo de cualquier map item
- Ancho máximo de cualquier map ítem (debe expresarse como un número negativo)
- Número de ítems (como máximo será 64)

Los dos primeros parámetros son importantes para chequear cuando un sprite puede estar parcialmente apareciendo en pantalla, ya que la funcion 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.

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

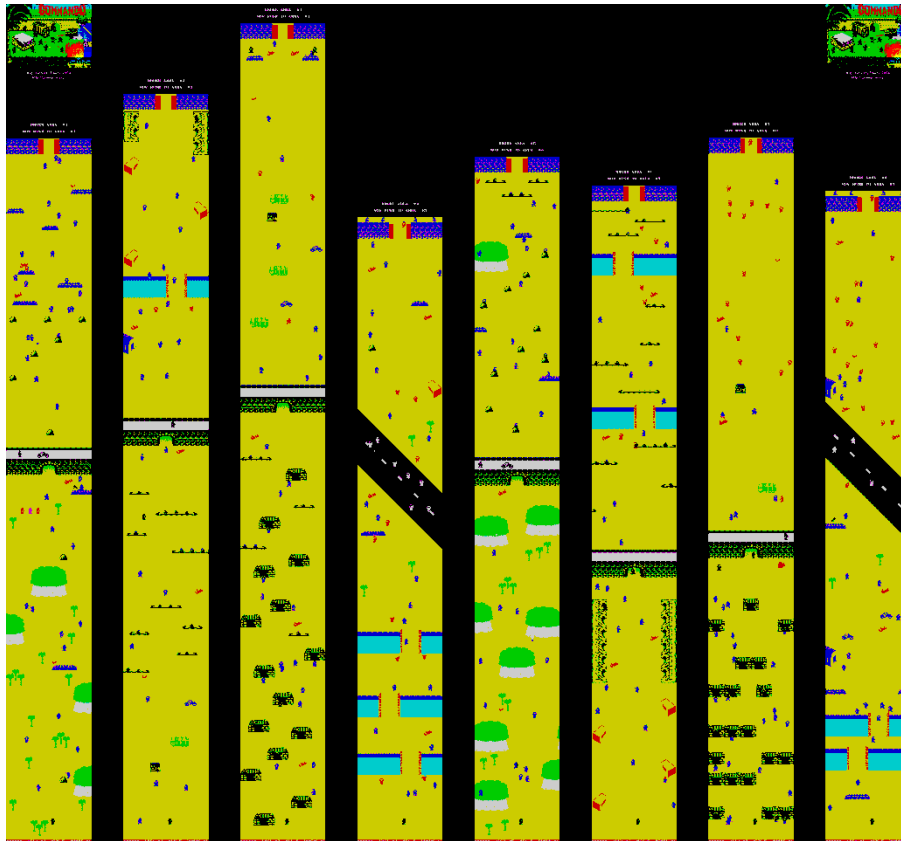
```
;MAP TABLE
;-----
; primero 3 parametros antes de la lista de "map items"
dw 50; maximo alto de un sprite por si se cuelga por arriba y ya hay
que pintar parte de el
```

```
dw -40; máximo ancho de un sprite por si se cuelga por la izquierda
(numero negativo)
db 64; numero de elementos del mapa.como mucho debe ser 64

; 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 el los elementos que quieres que tenga tu mundo. Cada cuadradito del cuaderno puede representar una cantidad fija como 8 pixeles 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. Aquí tienes un ejemplo del mapa del videojuego Commando. Como ves cada mapa es una fase. En 8BP puedes cambiar el mapa cuando quieras usando funciones POKE, o bien tener mapas grabados en archivos de 400 bytes que cargas en la dirección 42040, etc. Hay muchas soluciones para hacer diferentes fases sin necesidad de recurrir a ficheros en disco, ya que cada mapa ocupa solo 400 bytes como mucho y puedes tener varias fases en memoria RAM



*Fig. 52 mapa del clásico juego “commander” (8 fases)*

### 10.6.2 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 minijuego 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 colision para que solo se detecten colisiones con enemigos y disparos y no con elementos de fondo, usando COLSP, 32, <sprite inicial>, <sprite\_final>.





*Fig. 53 Minijuego con scroll inspirado en "commando"*

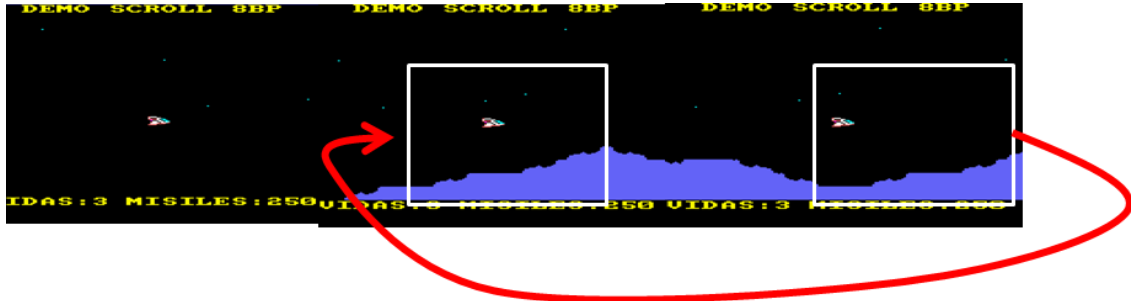
Ahora veamos el listado, como ves, es muy pequeño, pero lo tiene todo: scroll multidireccional, lectura de teclado, cambio de secuencias de animación del personaje, detección de colisión, música...

```

10 MEMORY 25999
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 el mapa del mundo solo tiene 13 elementos



*Fig. 54 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 cuelga 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;26;13; numero de elementos del mapa a considerar. como mucho
debe ser 64

; 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,MONTUP;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,MONTUP;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 25999
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 Música

Las herramientas de las que voy a hablar en este apartado no las he hecho yo, pero están integradas en 8BP y son realmente buenas.

### 11.1 Editar musica con WyZ tracker

Esta herramienta es un secuenciador de musica para el chip de sonido AY3-8912. Las musicas 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.

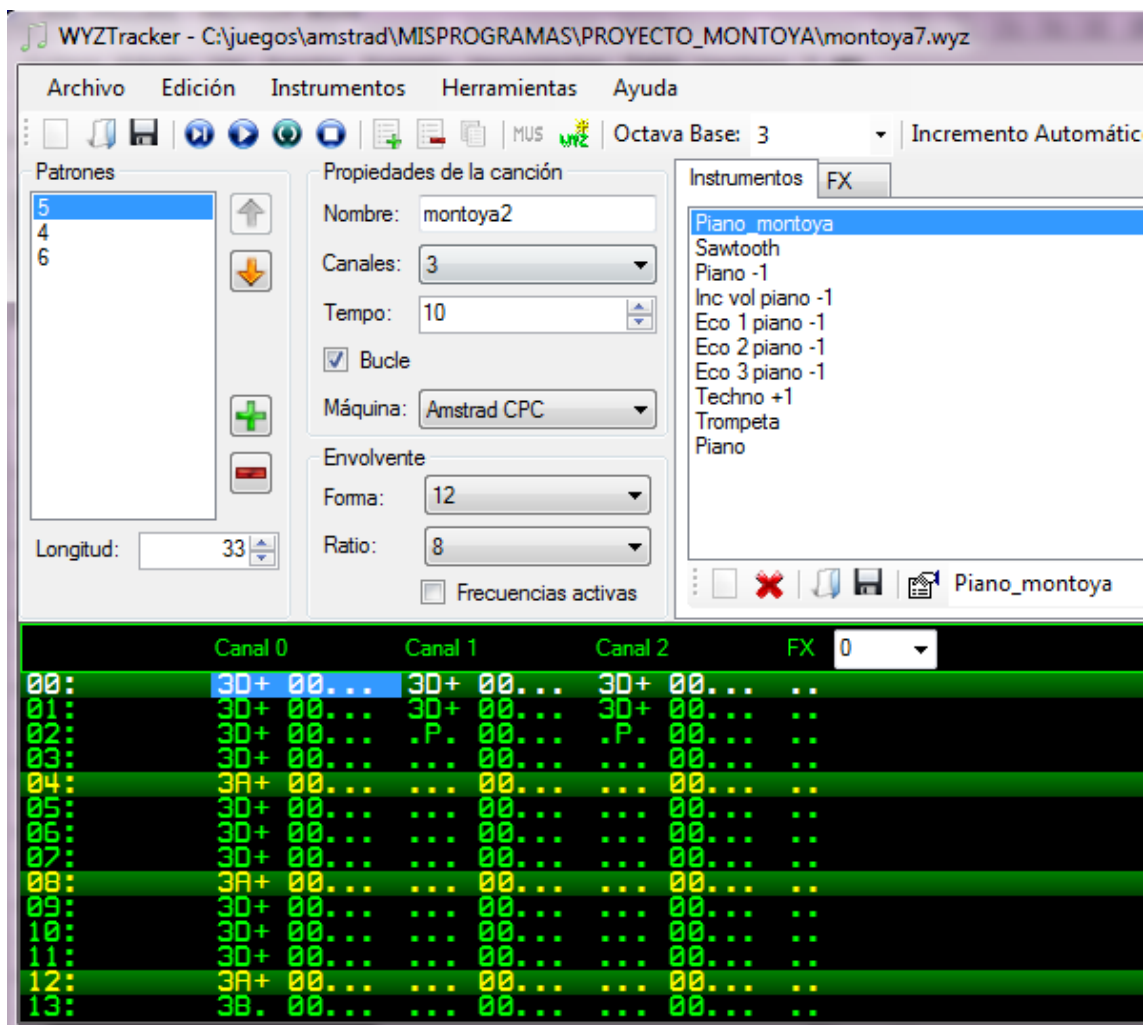


Fig. 55 WYZTracker

Este secuenciador de musica 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

## 11.2 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 musica asi:

```
; tras ensamblarlo, salvalo con save "musica.bin",b,32250,1250

org 32250
;-----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 musica (que esta 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, 26000, 6250

y las músicas:

SAVE "music.bin", b, 32250, 1250





## 12 Guía de referencia de la librería 8BP

### 12.1 Funciones de la librería

#### 12.1.1 |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 válida 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 alcanzarlos 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
--

### 12.1.2 |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

Es recomendable su uso si vas a animar muchos sprites ya que es mucho más 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

### 12.1.3 |AUTO

Este comando mueve un sprite (cambia sus coordenadas) de acuerdo a sus atributos de velocidad Vy,Vx

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 automatico activo en el campo status

#### **12.1.4      |AUTOALL**

Este comando mueve todos los sprites que tengan el flag de movimiento automatico 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.

#### **12.1.5      |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.

Uso:

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

O bien:

|COLAY, <num\_sprite>,<@colision%>

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)

Ejemplo:

|COLAY, 0,@colision%

La variable que uses para colisión puede llamarse como quieras.

Esta rutina modifica la variable colision ( la cual debe ser entera y por eso el “%”) poniéndola a 1 si hay colision del sprite indicado con el layout. Si no hay colision el resultado es 0.

```

xanterior=x
x=x+1
|LOCATESP,0,y,x: ' posicionamos el sprite en nueva posicion
|COLAY,0,@colision%: ' chequeo de la colision

```

Ahora comprobamos la colision y si hay colision lo dejamos en su ubicación anterior

```

if colision%=1 then x=xanterior: LOCATESP,0,y,x

```

Si nuestro personaje puede moverse en diagonal, a menudo queremos que al pulsar derecha+arriba nuestro sprite avance en una dirección aunque en la otra haya un muro y quede bloqueado. Esto da una sensación de mayor fluidez al movimiento aunque complica la lógica. Aquí se muestra como hacerlo, a partir de la línea 1721. Primero se coloca al sprite con LOCATESP y luego en función de las colisiones que se detectan con COLAY se recoloca nuevamente el sprite con LOCATESP

En este ejemplo las variables tienen el siguiente significado

Xa: coordenada x anterior  
 ya: coordenada y anterior  
 Xn: coordenada x nueva  
 yn: coordenada y nueva

```

1500' rutina de 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,0,@c1%:IF c1%=0 then 1450

1534 yn=ya:|LOCATESP,0,yn,xn:|COLAY,0,@c1%:IF c1%=0 then 1450

1535 xn=xa: yn=ynn:|LOCATESP,0,yn,xn:|COLAY,0,@c1%:IF c1%=1 THEN
yn=ya:|LOCATESP,0,yn,xn

1536 ya=yn:xa=xn

1537 return

```

### 12.1.6 |COLSP

Este comando permite detectar la colision 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 funcion 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)

Al igual que la impresión de sprites con PRINTSPALL, la funcion 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 colision. 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, 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 de colisión activo).

Esta estrategia permite reducir bastante tiempo, por ejemplo si solo debemos chequear 6 enemigos, preconfigurando 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 minimo (de un pixel en vertical y/o un byte en horizontal) debes hacer:

```
|COLSP, 34, 0, 0
```

### **12.1.7 |COLSPALL**

Uso

Para configurar:

```
|COLSPALL, @colisionador%, @colisionado%
```

Para comprobar las colisiones

```
|COLSPALL
```

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 colision 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.

### 12.1.8 |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.

Caracter	Sprite id	Codigo ASCII
“ “	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 ingles).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 tendras 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 limites establecidos con SETLIMITS deben permitir que se imprima donde desees. Si posteriormente quieres hacer clipping en una zona mas reducida puedes invocar de nuevo a SETLIMITS cuando todo el layout este impreso

### **Ejemplo**

2070   SETLIMITS,0,80,0,200	En este ejemplo se usan varios ladrillos que
-----------------------------	--



2090 c\$(1)= "PPPPPPPPPPPPPPPP P"	previamente se han creado con la
2100 c\$(2)= "PU P"	herramienta "spedit"
2110 c\$(3)= "P P"	; sprite 20 --> O arbusto
2120 c\$(4)= "P P"	; sprite 21 --> P roca
2130 c\$(5)= "P T P P P U T P P P P P P P"	; sprite 22 --> Q nube
2140 c\$(6)= "P T P"	; sprite 23 --> R agua
2150 c\$(7)= "P P"	; sprite 24 --> S ventana
2160 c\$(8)= "P P"	; sprite 25 --> T arco de puente derecho
2170 c\$(9)= "P Y Y Y Y Y Y Y Y P P"	; sprite 26 --> U arco de puente izq
2190 c\$(10)= "P T P P P P P P P U P"	; sprite 27 --> V bandera
2195 c\$(11)= "P P"	; sprite 28 --> W planta
2200 c\$(12)= "P P"	; sprite 29 --> X pico de torre
2210 c\$(13)= "P P"	; sprite 30 --> Y cesped
2220 c\$(14)= "Y Y Y Y Y Y Y Y P P Y Y Y Y Y Y"	; sprite 31 --> Z ladrillo
2230 c\$(15)= "R R R R R R R R R R R R R R R R"	
2240 c\$(16)= "P P P P P P P P P P P P P P P P"	
2250 c\$(17)= "P U T P P U T P"	
2260 c\$(18)= "P T U P"	
2270 c\$(19)= "P P"	
2271 c\$(20)= "P P"	
2272 c\$(21)= "P W P"	
2273 c\$(22)= "P P W P P"	
2274 c\$(23)= "P P P P P P P P P P P P P P P P P P"	
2280 for i=0 to 23	
2281   LAYOUT,i,0,@c\$(i)	
2282 next	



### 12.1.9 |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 funcion |POKE, ya que con el POKE de BASIC sería ilegal

El comando |LOCATE no imprime el sprite, sólo lo posiciona para cuando sea impreso.

### 12.1.10 |MAP2SP

Esta funcion 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>

Ejemplo

|MAP2SP, 1500,2500

Los parámetros 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 número de ítems del mundo (máximo 64)

```
;MAP TABLE
;-----
; primero 3 parametros antes de la lista de "map items"
dw 50; maximo alto de un sprite por si se cuelga por arriba y ya hay
que pintar parte de el
dw -40; máximo ancho de un sprite por si se cuelga por la izquierda
(numero negativo)
db 64; numero de elementos del mapa a considerar. como mucho debe ser
64

; 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

...
```

### 12.1.11 |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

### **12.1.12 |MOVERALL**

Este comando mueve de forma relativa todos los sprites que tengan el flag de movimiento relativo activado

Uso

|MOVERLALL,<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

### **12.1.13 |MUSIC**

Este comando permite que una melodía comience a sonar

Uso:

|MUSIC,<numero\_melodía>,<velocidad>

el numero de la melodía estará comprendido entre 0 y 7

la velocidad “normal” es 5. si usamos un numero superior se reproducirá más lentamente y si el numero 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 estan 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
```

#### 12.1.14 |MUSICOFF

Este comando paraliza la reproducción de cualquier melodía. No tiene parámetros

Uso:

|MUSICOFF

Internamente lo que hace es desinstalar la interrupción

#### 12.1.15 |PEEK

Este comando lee el valor de un dato de 16 bit de una direccion de memoria dada. Esta 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 mas rápido.

### 12.1.16 |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

### 12.1.17 |PRINTSP

Uso:

|PRINTSP, <sprite id >, <y >, <x>

Ejemplo

imprime el sprite 23 en las coordenadas y=100, x=40

|PRINTSP, 23,100,40

Esta rutina imprime un sprite en la pantalla, pero no por ello actualiza las coordenadas del sprite en la tabla de sprites.

Las coordenadas consideradas son

Numero de líneas en vertical [-32768..32768]. las correspondientes al interior de la pantalla son [0..199]

Numero de bytes en horizontal [-32768..32768]. las correspondientes al interior de la pantalla son [0..79]

Normalmente en la lógica de un viejuego harás uso de |PRINTSPALL, ya que es mas 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

```

7089 telon=&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

```



*Fig. 56 Un ejemplo de uso de PRINTSP*

### 12.1.18 |PRINTSPALL

Uso:

|PRINTSPALL, <orden>, <flag anima>, <flag sync>

Ejemplo:

imprime todos los sprites animándolos primero y sin sincronizar con barrido, y sin ordenar

|PRINTSPALL, 0, 1, 0

Esta rutina imprime de una sola vez todos los sprites que tengan el bit0 de estado activo. Si se pone a 1 el flag de animación, 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.

Cuanto 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

Por ultimo tenemos el parámetro de ordenamiento. Debemos indicar el número de sprites ordenados por coordenada Y que vamos a imprimir. Por ejemplo si ponemos un 0, entonces los sprites se imprimirán secuencialmente desde el sprite 0 hasta el sprite 31. Si ponemos un 10 se imprimirán del 0 al 10 ordenados (11 sprites) y del 11 al 31 de modo secuencial. Si ponemos un 31 se imprimirán todos los sprites ordenados.

El ordenamiento es muy útil para hacer juegos tipo renegade o golden AXE, donde es necesario dar un efecto de profundidad.

|PRINTSPALL, 0, 1, 0 : imprime de forma secuencial todos los sprites  
|PRINTSPALL, 31, 1, 0 : imprime de forma ordenada todos los sprites  
|PRINTSPALL, 7, 1, 0 : imprime de forma ordenada 8 sprites y el resto secuencial

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 2.5ms pero si ordenas solo 5 te puedes ahorrar 2ms. Quizas tengas muchos sprites y no merezca la pena ordenar algunos, como los disparos o sprites que sabes que no se van a solapar.

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.

### 12.1.19 |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 esta 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 ultimo segmento es cero, indicando que la ruta se ha terminado y que el sprite debe comenzar desde el principio.

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
```

### 12.1.20 |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 area 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 area permitida, las funciones |PRINTSP y |PRINTSPALL imprimirán solo la parte que se encuentra dentro del área permitida.

### 12.1.21 |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 parametro a la vez. El parametro que vamos a cambiar se especifica con param\_number. El param\_number es en realidad la posicion relativa del parametro en la SPRITES\_TABLE

- param\_number= 0 → cambia el status (ocupa 1 byte)



- param\_number= 5 → cambia Vy (ocupa 1byte, valor en lineas verticales). También se puede modificar Vx a la vez si lo añadimos al final como parámetro
- param\_number= 6 → cambia Vx (ocupa 1byte, valor en bytes horizontales)
- param\_number= 7 → cambia secuencia (ocupa 1byte, toma valores 0..31)
- param\_number= 8 → cambia frame\_id (ocupa 1byte, toma valores 0..7)
- param\_number= 9 → cambia dir imagen (ocupa 2bytes)
- param\_number= 15 → cambia la ruta (ocupa 1bytes)

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\_tujuego.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 aquí una lista de las imágenes que queremos usar sin especificar la dirección 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 gráficos o se reensamblan en
; direcciones diferentes, el número que asignemos no cambiará
; NO tienen que tener todos un número, solo aquellas que vamos a usar con |setupsp, id, 9,<num>
; se empiezan a numerar en 16
; podemos usar hasta 255 imágenes especificadas de este modo
; no hace falta que la lista tenga 255 elementos. es de longitud variable, incluso puede estar vacía
;-----
DW NAVE ; 16
DW OTRA_NAVE ; 17

;----- BEGIN SPRITE -----
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 SPRITE -----

```

En el caso de param\_number=5, podemos incluir Vx como parámetro al final:  
|SETUPSP, 31, 5, Vx, 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 estas 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.

### 12.1.22 |SETUPSQ

Este comando crea una secuencia de animación

Uso:

|SETUPSQ, <numero de secuencia>, <dirección1>, <direccionN>, 0,...,0

Se deben rellenar 8 direcciones o completar hasta 8 direcciones con ceros

El número de secuencia debe estar entre 1 y 31

Ejemplo que crea la secuencia 1 con las 4 direcciones donde hay ensambladas las imágenes (fotogramas) de un personaje animado

SETUPSQ, 1, &926c, &92FE, &9390, &02fe , 0, 0, 0, 0

Al igual que con el comando SETUPSP, puedes hacer uso de los identificadores de la IMAGE\_LIST en el comando SETUPSQ, de modo que te será mas sencillo crear secuencias en BASIC.

Importante: con el comando SETUPSQ no se pueden crear macrosecuencias de animación, debes crearlas definiéndolas en el fichero sequences\_tujuego.asm. Las macrosecuencias usan identificadores a partir de 32

### 12.1.23 |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 esta 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

```
10 CALL &6b78: rem instala los comandos 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 ahora 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 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

## 13 Ensamblado de la librería, gráficos y música

Tanto si quieres hacer cambios en la librería como si añades música y gráficos deberás reensamblarla. Esto es debido a que por ejemplo, el player de música esta integrado en la librería y necesita conocer donde comienza (dirección de memoria) cada canción, por lo que es necesario reensamblar 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 tu 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:

- Librería 8BP (es un fichero binario), incluyendo la tabla de atributos de sprites
- Fichero binario de musica, 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, musica 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.

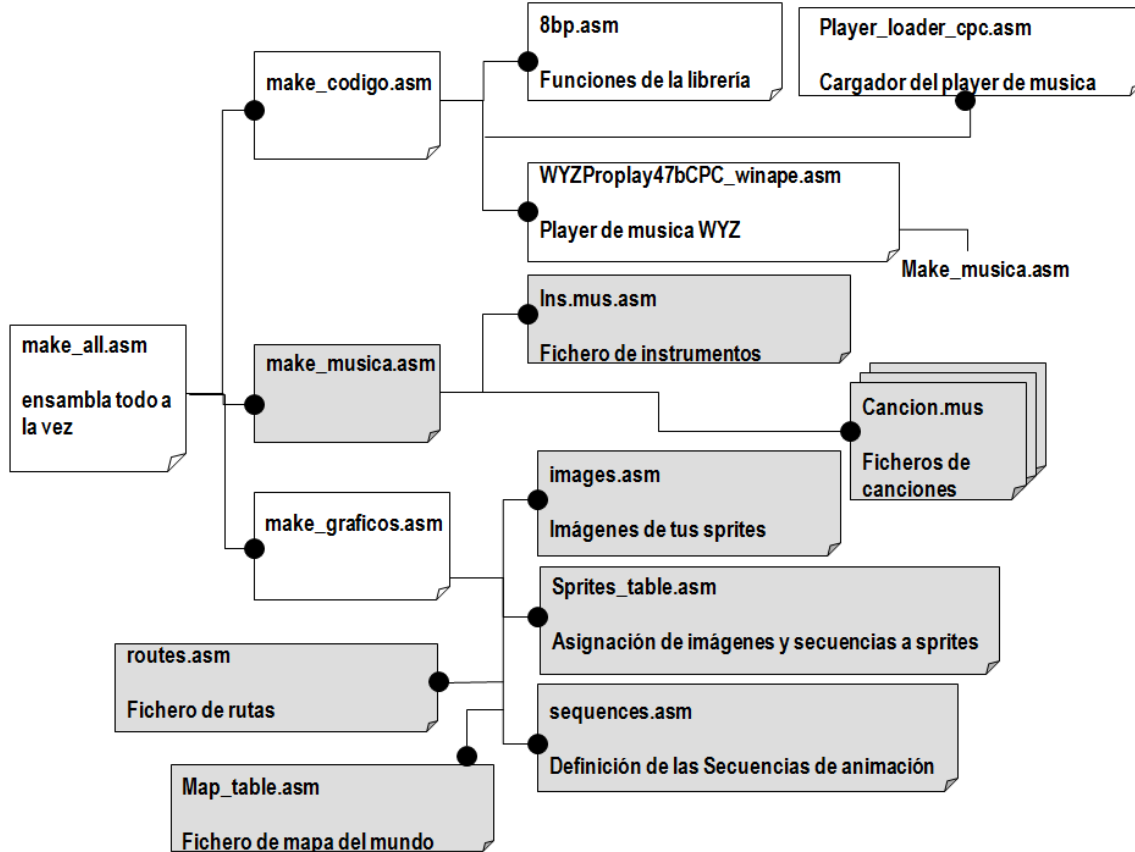
```
SAVE "8BP.LIB", b, 26000, 6250
SAVE "MUSIC.BIN", b, 32250, 1250
SAVE "SPRITES.BIN", b, 33500, 8500
```

Ahora queda comprender como ensamblar todo en memoria (librería, musica y graficos). 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 asi 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, (aunque no es estrictamente necesario pues tienes el comando |SETUPSQ )
- 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



*Fig. 57 Ficheros para ensamblar*

Puedes ensamblarlo todo con make\_all y después usar el comando SAVE para salvar las imágenes, musica y librería 8BP en diferentes ficheros binarios, tal como hemos visto. Para ello simplemente abres el fichero make\_all dentro del editor de winape y pulsas “assemble”.

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 codigo 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 y volver a salvar tus ficheros “8BP.LIB” y “MUSIC.BIN”

## **14 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

### **14.1 Memoria para ubicar nuevas funciones**

Actualmente la librería ocupa 6250 Bytes, de los cuales 75 bytes están libres para futuras ampliaciones.

El espacio destinado para la música es actualmente de 1250bytes y no conviene reducirlo, por lo que si en un futuro fuera necesaria más memoria aparte de esos 300 bytes, hay dos opciones:

- Reducir el el espacio disponible para el programador (26KB).
- Reducir los 8KB destinados a gráficos

### **14.2 Ahorrar memoria de gráficos**

Una posible operación:

|FLIP, secuencia, dirección

Podría dar la vuelta a todos los frames de una secuencia de animación, ahorrando mucha memoria de gráficos. Sería una instrucción fácil de programar y muy rentable en el ahorro. Al programador le obligaría a invocarla cuando un sprite cambia de dirección o bien, para evitar cualquier ralentización por ese motivo, se podría hacer FLIP de todos los tipos de enemigos que fuesen a aparecer en una pantalla justo antes de entrar en ella, reutilizando la memoria en otras pantallas con otros enemigos diferentes.

### **14.3 Impresión a resolución de píxel**

Actualmente 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 conseguiras un movimiento pixel a pixel.

### **14.4 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 esta 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) y que manipulásemos desde BASIC con POKE

### **14.5 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 firebird:

- Misión genocida (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. 58 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 si, 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 “como 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 mas eficiente pues mueve menos memoria y a la vez permite movimiento multidireccional.

#### **14.6 Migrar la librería 8BP a otros microordenadores**

Esta librería seía 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 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. 59 Sinclair ZX y Commodore 64, dos clásicos*



## 15 Algunos juegos hechos con 8BP

En este capitulo voy a describir como estan hechos algunos juegos que puedes encontrar en la web <https://github.com/jjaranda13/8BP> realizados con 8BP

- **Anunnaki:** un juego de naves, género arcade
- **Mutante Montoya:** un juego de pasar pantallas, podría encuadrarse en género plataformas



### 15.1 Mutante Montoya

Un primer tributo al Amstrad CPC, con un título inspirado en el clasico "mutant monty"

Disponible como parte de la distribución freeware de la libreria 8BP en dsk y wav en <https://github.com/jjaranda13/8BP>

Es un juego sencillo de 5 niveles. Se basa en el uso del layout de 8BP para construir cada pantalla. A continuación algunas "pistas" de como está hecho, aunque el listado basic es accesible

	<p>La presentación simula lluvia mediante el comando  STARS. Trata de apreciar como pasan por debajo de sprites y elementos del layout sin dañarlos.</p> <p>El castillo está construido como un layout. En el listado BASIC lo podrás distinguir construido con letras asociadas a sprites</p>
	<p>El primer nivel es muy sencillo. la condicion de salida es que la coordenada Y del personaje sea menor que cero (cuando consigues salir por arriba).</p> <p>todos los sprites se imprimen simultáneamente con el comando de 8BP " PRINTSPALL"</p>

	<p>El segundo nivel no usa ninguna tecnica especial. No usa logicas masivas y aun asi puedes ver la potencia y velocidad de la libreria 8BP. hay 7 sprites</p> <p>También puedes apreciar el "clipping" de la rutina de impresión si mueves el personaje fuera de la pantalla, hacia la izquierda. En ese caso se mostrará sólo parcialmente</p>
	<p>El tercer nivel usa la técnica de "lógicas masivas" para mover los 8 soldados. A pesar de haber 9 sprites en pantalla a la vez, y estar ejecutando en BASIC, todo funciona a un ritmo adecuado. Es interesante analizar como se abre la compuerta. Se actúa sobre el layout cuando montoya coge la llave.</p>
	<p>El cuarto nivel usa "lógicas masivas". Podría haber metido 8 fantasmas y habría seguido funcionando sin apenas diferencia de velocidad, pero habría sido muy difícil pasar la pantalla!</p>
	<p>En esta pantalla no se usan lógicas masivas aunque haya 5 sprites, pues es suficientemente rápido. Se emplea la misma técnica que en el tercer nivel para abrir las compuertas</p>



Al final todos los personajes salen a saludar. El efecto del telon que baja se hace con un solo sprite, imprimiéndolo en un bucle e invocando al comando |PRINTSP. es muy sencillo pero interesante efecto. El sprite va "manchando" la pantalla mientras baja, dando la sensación de un telón en movimiento

## 15.2 Anunnaki, nuestro pasado alien

Este es un juego muy interesante para analizar y adentrarse en la técnica de programación de "lógicas masivas". El annunaki es un videojuego de arcade que muestra un uso intensivo de la técnica de lógicas masivas y simulación de scroll vertical

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. Se encuentra disponible como parte de la distribución freeware de la librería 8BP .

Disponible en dsk y wav en <https://github.com/jjaranda13/8BP>

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 a los que debes destruir.




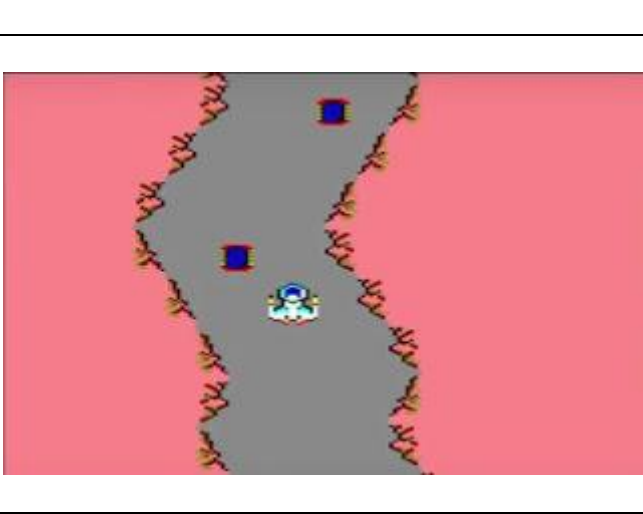
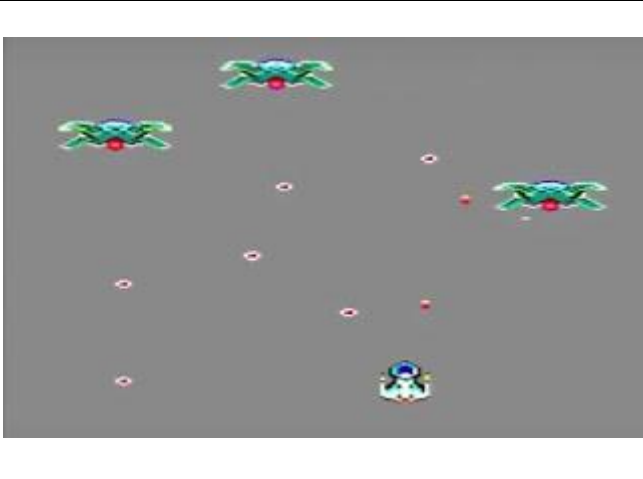

la pantalla de presentación muestra un scroll de estrellas de 4 capas a diferente velocidad para dar sensación de profundidad. Para ello se usa el comando STARS

los controles son QAOP y la barra espaciadora para disparar. Durante el juego, al pulsar Q, la nave sube y muestra fuego en sus cohetes propulsores. Esto simplemente se realiza cambiando la secuencia de animación asociada a la nave con un POKE a la dirección correspondiente a

	<p>la secuencia de animación de la nave en la tabla de sprites</p>
	<p>los meteoritos son indestructibles. Al usar la instrucción de colisión COLSPALL, se determina que el colisionado es un sprite cuyo identificador es mayor que un cierto número y con ello se concluye que es un enemigo "duro" que no se puede destruir.</p> <p>se utiliza lógicas masivas, de modo que en cada fotograma solo un meteorito puede decidir cambiar su dirección de movimiento</p> <p>la colisión del disparo múltiple se realiza mediante COLSPALL, ya que los 3 disparos tienen flag de "colisionador" activo en el byte de status, y para ahorrar ciclos solo puede empezar a morir un enemigo en un mismo fotograma, lo cual no representa ninguna limitación en el juego, ya que vamos matando uno a uno a los enemigos.</p>
	<p>todos los pajarracos tienen el flag de movimiento relativo en su byte de status, y se mueven con el comando MOVERALL siguiendo una trayectoria almacenada en un array. De este modo todos se mueven a la vez con una sola instrucción</p> <p>hay una horda de naves que bajan en "zig-zag", que también usan la misma técnica, los desplazamientos calculados con la función coseno son almacenados en un array para después usarlo en MOVERALL</p> <p><code> MOVERALL,0,rx(i): i=(i+1) MOD 15</code></p> <p>como puedes ver, a la vez hay estrellas de fondo que pasan "por debajo" de todos los enemigos, mediante el comando STARS.</p> <p>las explosiones son "secuencias de muerte", de modo que tras terminar la animación de la explosión, el enemigo</p>

	<p>en cuestión queda desactivado y no se imprime mas.</p>
	<p>las distintas hordas de naves enemigas en formación siguen trayectorias diferentes, siempre en dos hileras de 6 naves cada una. Para moverlas se utiliza la técnica de logicas masivas controlando en cada instante de tiempo que par de naves deben cambiar su dirección de movimiento. El movimiento se realiza asignando a cada nave un par <math>V_x, V_y</math> e invocando a AUTOALL, ya que todas tienen el flag de movimiento automático</p>
	<p>el enemigo final (llamado "arkaron") es semi-duro, significa que tiene un contador de disparos que le alcanzan y muere al recibir diez impactos.</p> <p>la música durante estos momentos cambia, siendo una melodía mas "angustiosa". las 3 melodías que tiene el juego ocupan poco, no mas de 200 bytes cada una</p>
	<p>En la luna ("hollow moon" significa "luna hueca" pues se trata de un satélite artificial alienígena) se simula scroll con "motas" y cráteres que se mueven a la misma velocidad. Las motas se hacen con STARS y los cráteres son sprites que se mueven hacia abajo.</p> <p>Además, se utilizan dos sprites a los lados que "manchan" la pantalla dando el efecto de muros laterales.</p>  <p>todas las hordas usan "lógicas masivas" para moverse</p>
	<p>Más hordas, esta vez con una trayectoria que cambia de dirección pasados unos instantes.</p> <p>Fíjate como la nave pasa "por encima"</p>



	<p>del cráter a pesar de que no se está usando sobreescritura. Es gracias a que la nave tiene un identificador de sprite superior al cráter y por ello se imprime después. Obviamente "borra" un trozo de cráter pero el efecto es aceptable</p>
	<p>el tunel se construye con dos hileras de sprites que simulan los bordes del tunel, dando un efecto de scroll vertical</p> <p>es realmente pesado porque son sprites muy grandes pero el efecto es aceptable</p> <p>superar el túnel es difícil, se genera aleatoriamente de modo que a veces es más difícil y a veces más fácil.</p>
	<p>al final hay que destruir 3 jefes cuya dificultad estriba en que ninguno muere de forma independiente sino que cada uno de ellos debe recibir al menos 10 disparos para que los 3 mueran a la vez. si disparas mil veces al mismo no conseguiras superar este reto. El número de sprites en pantalla es muy elevado si sumamos todos los disparos de la nave y de los enemigos pero los movimientos de los disparos tienen poca lógica y todo se mueve con velocidad suficiente</p>
	<p>Al destruir a los 3 jefes, llegamos a la pantalla del final del juego, con una melodía diferente y un mensaje de enhorabuena.</p> <p>Enki es bueno?...o es malo? los textos sumerios lo describen como el dios bueno y su hermano Enlil era el malo...pero y si todo hubiese sido un engaño?</p>



## 16 APENDICE III: Organización de la memoria de video

### 16.1 El ojo humano y la resolución del CPC

La memoria de video del amstrad CPC tiene 3 modos de funcionamiento. El modo mas utilizado para los juegos es el mode 0 (160x200) por disponer de mas color pero tambien se ha usado bastante el mode 1 (320x200) para programar juegos.

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 mas largo, hay menos resolución que en vertical (160 en horizontal y 200 en vertical). Te preguntará por qué. Además no es el unico 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.

### 16.2 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.

#### 16.2.1 Mode 2

En mode 2, cada píxel esta 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 (pi) 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 mas situado a la izquierda. El píxel 0, es precisamente tambien el pixel situado mas a la izquierda, es decir, aquí no hay nada “al revés”. Está todo correcto

#### 16.2.2 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 mas compleja. El pixel 0 por ejemplo se codifica con los bits 7 y 4

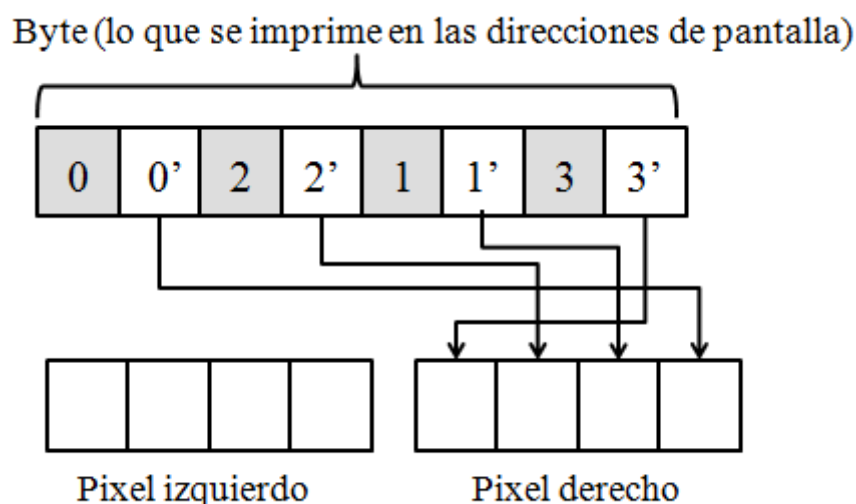
bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
p0(1)	p1(1)	p2(1)	p3(1)	p0(0)	p1(0)	p2(0)	p3(0)

### 16.2.3 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

<b>bit7</b>	<b>bit6</b>	<b>bit5</b>	<b>bit4</b>	<b>bit3</b>	<b>bit2</b>	<b>bit1</b>	<b>bit0</b>
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:



*Fig. 60 pixels y bits en mode 0*

No te puedo decir cual 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.

### 16.2.4 Memoria de la pantalla

Los pixels de la pantalla pertenecientes a una misma linea se encuentran codificados en los bytes que tambien son contiguos. Sin embargo, de una linea a otra hay saltos.

Si avanzamos en direcciones de memoria, al llegar al final de una linea saltamos a una linea que se encuentra 8 lineas mas abajo. Y si queremos continuar en la linea 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)

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	FB00
14	C410	CC10	D410	DC10	E410	EC10	F410	FC10
15	C460	CC60	D460	DC60	E460	EC60	F460	FC60
16	C4B0	CCB0	D4B0	DCB0	E4B0	ECB0	F4B0	FCB0
17	C500	CD00	D500	DD00	E500	ED00	F500	FD00
18	C550	CD50	D550	DD50	E550	ED50	F550	FD50
19	C5A0	CDA0	D5A0	DDA0	E5A0	EDA0	F5A0	FDA0
20	C5F0	CDF0	D5F0	DDF0	E5F0	ED50	F550	FD50
21	C640	CE40	D640	DE40	E640	EE40	F640	FE40
22	C690	CE90	D690	DE90	E690	EE90	F690	FE90
23	C6E0	CEE0	D6E0	DEE0	E6E0	EEE0	F6E0	FEE0
24	C730	CF30	D730	DF30	E730	EF30	F730	FF30
25	C780	CF80	D780	DF80	E780	EF80	F780	FF80
spare start	C7D0	CFD0	D7D0	DFD0	E7D0	EFD0	F7D0	FFD0
spare end	C7FF	CFFF	D7FF	DFFF	E7FF	FFFF	F7FF	FFFF

*Fig. 61 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  $25 \times 80 \text{ bytes} = 2000$  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 &EFD0 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 Líberia 8BP estos segmentos se usan para almacenar variables locales de algunas funciones, cuyo valor puede ser borrado sin riesgos.

### **16.3 Barridos de pantalla**

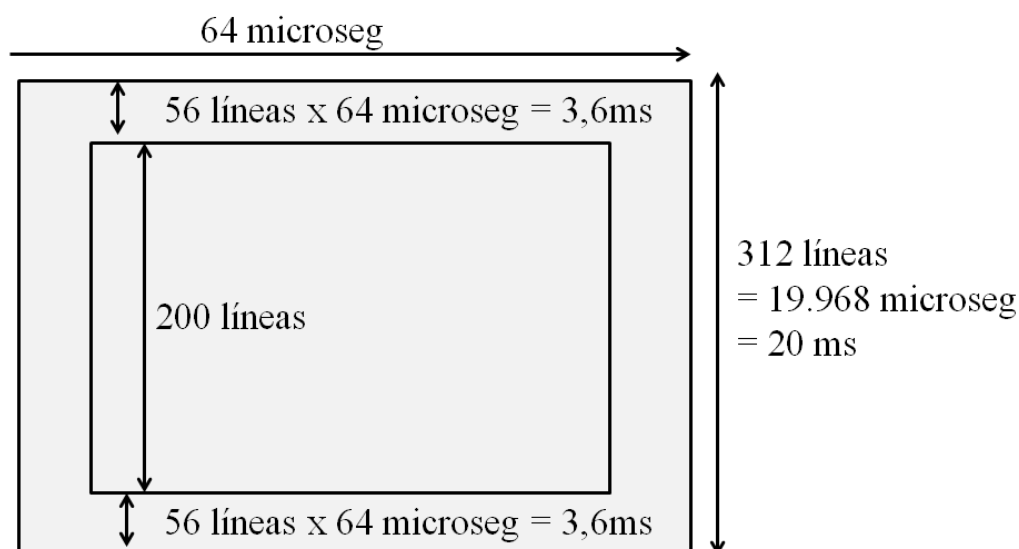
El amstrad genera 50 imágenes por segundo. Eso significa que cada 20ms aproximadamente se debe de producir un nuevo barrido de pantalla.

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. Fijate 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 podras imprimir como mucho 2 sprites pequeños.



*Fig. 62 tiempos en un barrido de pantalla*

## **16.4 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 menus 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

Y verías poco a poco como mientras carga se va dibujando en pantalla, 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

Por último puedes usar una herramienta como ConvImgCPC (tambien hay otras), un conversor 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

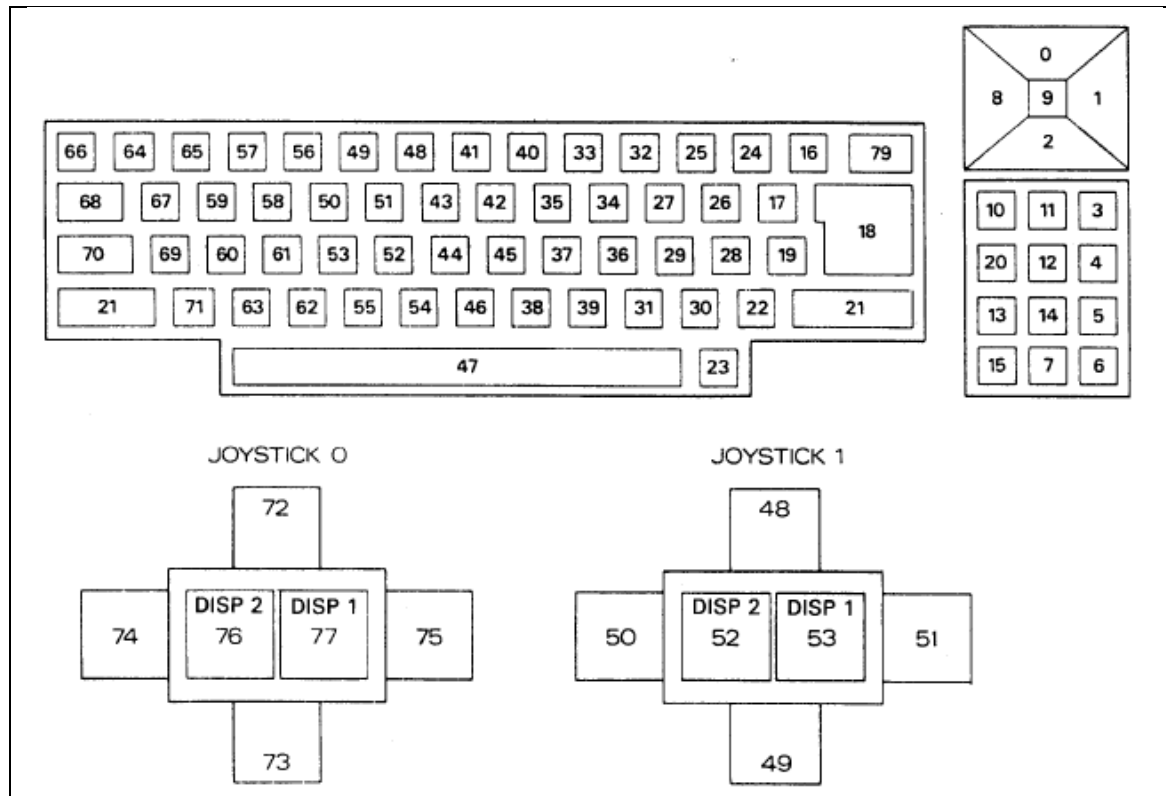
Para meter este fichero en un disco (en un fichero .dsk) debes usar el CPCDiskXP que es otra herramienta que te permite meter ficheros dentro de ficheros .dsk

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 mas posible a los colores originales. Si invocas `CALL &C7D0` conseguiras ver la imagen con los colores seleccionados por convImgCPC

Para dejar la paleta en sus valores por defecto, usa `CALL &BC02`, que es una rutina del firmware.

## 17 APENDICE IV: INKEY codes



## 18 APENDICE V: Paleta

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 el color 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 configurado el color cero para que sea rojo



19 APENDICE VI: Tabla ASCII del AMSTRAD CPC

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	□	□		0	Q	P	`	p		.	^	α	/	—	⊗	↑
1	Γ	0	!	1	A	Q	a	q		!	'	β	\		⊗	↓
2	└	0	"	2	B	R	b	r		-	"	γ	/	—	⊕	←
3	└	0	#	3	C	S	c	s		└	£	δ	\		⊕	→
4	⚡	0	\$	4	D	T	d	t		!	©	€	^	▴	♥	▲
5	⊗	×	%	5	E	U	e	u		!	π	θ	>	▴	⊕	▼
6	✓	π	&	6	F	V	f	v		!	§	λ	√	▴	○	▶
7	Ω	—	'	7	G	W	g	w		!	'	ρ	<	▴	●	◀
8	←	Σ	<	8	H	X	h	x		-	¼	π	/	⊗	□	⊗
9	→	⊗	>	9	I	Y	i	y		!	½	σ	/	⊗	■	⊗
A	↓	?	*	:	J	Z	j	z		!	¾	ρ	○	⊗	♂	⊗
B	↑	⊗	+	;	K	I	k	i		!	±	ρ	×	⊗	♀	⊗
C	⊗	⊗	,	<	L	\	l	l		!	÷	×	/	⊗	↓	⊗
D	⊗	⊗	—	=	M	]	m	]		!	+	—	ω	\	⊗	⊗
E	⊗	⊗	.	>	N	↑	n	~		!	+	δ	Σ	⊗	⊗	⊗
F	⊗	⊗	/	?	O	_	o	⊗		!	+	i	Ω	⊗	⊗	⊗

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240
1	1	17	33	49	65	81	97	113	129	145	161	177	193	209	225	241
2	2	18	34	50	66	82	98	114	130	146	162	178	194	210	226	242
3	3	19	35	51	67	83	99	115	131	147	163	179	195	211	227	243
4	4	20	36	52	68	84	100	116	132	148	164	180	196	212	228	244
5	5	21	37	53	69	85	101	117	133	149	165	181	197	213	229	245
6	6	22	38	54	70	86	102	118	134	150	166	182	198	214	230	246
7	7	23	39	55	71	87	103	119	135	151	167	183	199	215	231	247
8	8	24	40	56	72	88	104	120	136	152	168	184	200	216	232	248
9	9	25	41	57	73	89	105	121	137	153	169	185	201	217	233	249
A	10	26	42	58	74	90	106	122	138	154	170	186	202	218	234	250
B	11	27	43	59	75	91	107	123	139	155	171	187	203	219	235	251
C	12	28	44	60	76	92	108	124	140	156	172	188	204	220	236	252
D	13	39	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

## 20 APENDICE VII: 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 relentizar 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

## 21 APENDICE VIII: Tabla de atributos de Sprites

La siguiente tabla contiene las direcciones de memoria en la que se encuentran almacenados los atributos de cada srpite, y la longitud en bytes de cada uno.

	1byte	2 bytes	2 bytes	1byte	1byte	1byte	1byte	2 bytes	1byte
<b>sprite</b>	<b>status</b>	<b>coordy</b>	<b>coordx</b>	<b>vy</b>	<b>vx</b>	<b>seq</b>	<b>frame</b>	<b>imagen</b>	<b>ruta</b>
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*

## 22 APENDICE IX: 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, 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)
;      +-----
;      | 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)
; 33500 +-----
;      | canciones
;      |      (1.25 KB para musica editada con WYZtracker 2.0.1.0)
; 32250 +-----
;      | rutinas 8BP (6250 bytes)
;      |      aqui estan todas las rutinas y la tabla de sprites
;      |      incluye el player de musica "wyz" 2.0.1.0
; 26000 +-----
;      | variables el BASIC
;      | V
;      |
;      | ^ BASIC (texto del programa)
;      |
;      0 +-----
```

## 23 APENDICE X: comandos disponibles 8BP

ANIMA, #	cambia el fotograma de un sprite segun su secuencia
ANIMALL	cambia el fotograma de los sprites con flag animacion activado
AUTO, #	movimiento automatico de un sprite de acuerdo a su velocidad
AUTOALL , <flag enrutado>	movimiento de todos los sprites con flag de mov automatico activo
COLAY,umbral_ascii, #, @colision	detecta la colision con el layout y retorna 1 si hay colision
COLSP, #, @id	retorna primer sprite con el que colisiona #
COLSPALL, @quien%, @conquien%	Retorna quien ha colisionado y con quién ha colisionado
LAYOUT, y,x, @string	imprime un layout de imagenes de 8x8 y rellena map layout
LOCATESP, #,y,x	cambia las coordenadas de un sprite (sin imprimirlo)
MAP2SP,y,x	crea sprites para pintar el mundo en juegos con scroll
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 a la velocidad deseada
MUSICOFF	deja de sonar la melodía
PEEK,dir, @variable%	lee un dato 16bit (puede ser negativo) de una dirección
POKE,dir,valor	introduce un dato 16bit (que puede ser negativo) en una dirección de memoria
PRINTSP, #,y,x	imprime un solo sprite (# es su numero) sin tener en cuenta byte de status
PRINTSPALL,orden, anima, sync	imprime todos los sprites con flag de impresion activo
ROUTEALL	Modifica la velocidad de los sprites con flag de ruta
SETLIMITS, xmin,xmax,ymin,ymax	define la ventana de juego, donde se hace clipping
SETUPSP, #, param_number, valor	modifica un parametro de un sprite
SETUPSQ, #, adr0,adr1,...,adr7	crea una secuencia de animacion
STARS, initstar,num,color,dy,dx	scroll de un conjunto de estrellas

***Tabla 8 Comandos disponibles en la librería 8BP***

