Fourspriter 1.0

Biblioteca sencilla para el manejo de Sprites desde Sinclair BASIC

Introducción

Todo desarrollador de juegos en BASIC (interpretado o compilado) se ve, por lo general, bastante limitado artísticamente hablando debido a la dificultad que entraña restaurar el fondo al mover un sprite. Al final nos vemos obligados a usar el famoso fondo negro o emplear un patrón sencillo, ya que todo manejo más allá de esto implica o bien demasiado código, o bien una ejecución lenta.

Por esa razón, en MojonTwins nos decidimos a realizar una colección de rutinas en lenguaje de ensamblador muy sencillas que fueran capaces de ocuparse, de forma automática y totalmente transparente para el programador, de almacenar el fondo que ocupa un sprite y restaurarlo al cambiarlo de posición. Estas rutinas son, además, lo suficientemente rápidas para funcionar justo por delante de la actualización de la pantalla, con lo que no se percibe ningún tipo de parpadeos. Lo único que tiene que hacer el programador, antes de llamar a la rutina para que actualice la pantalla, es POKEar en determinadas posiciones de memoria la nueva posición de los sprites, y olvidarse de mayores complicaciones.

Construyendo la biblioteca

Fourspriter 1.0 se entrega en un archivo de código fuente de lenguaje de ensamblador listo para ser ensamblado con Pasmo. Además de ofrecer así el código fuente profusamente comentado para su estudio (y posible mejora) por otros programadores, permitimos que la biblioteca pueda colocarse donde mejor convenga dentro de la memoria. De todos modos, para los menos iniciados, hemos incluido un binario con la biblioteca generada en un lugar bastante conveniente: al final de la RAM, dejando espacio por encima para los UDG y un juego extra de caracteres, algo que resulta muy conveniente para programar en BASIC con vistas a compilar el resultado, especialmente con los compiladores MCoder 3 o Hisoft BASIC.

Lo primero que debemos hacer, si nos decidimos a compilar la biblioteca nosotros mismos, es hacernos un mapa de memoria. Para ilustrar esta sección, tomaremos el mapa de memoria que emplea la versión pre-ensamblada que se entrega con esta distribución. Dicha configuración de memoria es la que se muestra en la figura siguiente, teniendo en cuenta que la biblioteca ensamblada ocupa exactamente 797 bytes y que vamos a colocarlo todo al final de la RAM, dejando espacio para los UDG (168 bytes) y un set de caracteres alternativo (768 bytes) que podemos emplear para dibujar los fondos de nuestro juego.

Posición	Tamaño	Descripción
65368	168	UDG
64600	768	Charset
63803	797	fourspriter

Teniendo claro que vamos a colocar nuestra biblioteca en 63803, editamos el código fuente de la misma (`fourspriter.asm`) en nuestro editor de textos favorito, y examinamos la primera parte, donde definiremos mediante la directiva ORG dónde ensamblará Pasmo el binario para que quede listo para usar. El principio del archivo debe quedar tal y como se muestra en la siguiente figura:

```
;; Biblioteca simple de sprites de 16x16 que se mueven de 8 en 8 para BASIC
;; la idea es usarlo desde BASIC compilado.

;; Copyleft 2009 The Mojon Twins.
;; Pergreñado por na_th_an

;; Úsalo y modifícalo como te de la gana, pero "porfa", menciónanos :-)

;; Usa UDG y los lee de donde digan las variables del sistema.

;; Supuestamente esto sirve para mover hasta cuatro sprites por la pantalla
;; conservando los fondos.

;; A ver qué tal sale.

            org     63803           ; Esto lo ajustaré adónde más me convenga luego.
```

Una vez hayamos hecho esta modificación, salvamos el archivo al disco, nos abrimos una linea de comandos, y ejecutamos Pasmo para ensamblar el código (se presupone que tenemos instalado Pasmo y su ejecutable está accesible desde la ruta donde tengamos almacenado **Fourspriter 1.0**). Es importante que generemos un archivo de símbolos, ya que esto nos indicará los puntos de acceso a las rutinas. Esto se hace de la siguiente forma, donde "$" representa el símbolo del sistema:

```
$ pasmo fourspriter.asm fourspriter.bin fourspriter.txt
```

Con esto, pasmo generará un archivo fourspriter.txt donde relacionará cada etiqueta del código con su posición en memoria (en hexadecimal). Si lo editamos veremos la siguiente información, que tendremos que tener muy a mano cuando nos pongamos a programar:

```
attrs1          EQU 0F963H
attrs2          EQU 0F993H
attrs3          EQU 0F9C3H
attrs4          EQU 0F9F3H
borra_char      EQU 0FAEEH
borra_sprites   EQU 0FA99H
bufchars        EQU 0FB14H
buffatrs1       EQU 0F967H
buffatrs2       EQU 0F997H
buffatrs3       EQU 0F9C7H
buffatrs4       EQU 0F9F7H
buffer1         EQU 0F943H
buffer2         EQU 0F973H
buffer3         EQU 0F9A3H
buffer4         EQU 0F9D3H
charsabuff      EQU 0FB94H
copia_char      EQU 0FA65H
copyattrs       EQU 0FBA8H
cx_pos1         EQU 0F941H
cx_pos2         EQU 0F971H
cx_pos3         EQU 0F9A1H
cx_pos4         EQU 0F9D1H
cxpos           EQU 0F9FDH
cy_pos1         EQU 0F942H
cy_pos2         EQU 0F972H
cy_pos3         EQU 0F9A2H
cy_pos4         EQU 0F9D2H
cypos           EQU 0F9FEH
datap           EQU 0F93BH
docopy          EQU 0FBFAH
draw_sprites    EQU 0FB18H
get_attr_address EQU 0FC3AH
get_scr_address EQU 0FC20H
getde           EQU 0FBE1H
i4chars         EQU 0FA04H
i4chars2        EQU 0FA9EH
i4chars3        EQU 0FB1DH
i4chars4        EQU 0FBC5H
init_sprites    EQU 0F9FFH
```

```
nxt1            EQU 0FA61H
nxt2            EQU 0FAEAH
nxt3            EQU 0FB90H
spare           EQU 0FC50H
udgs1           EQU 0F93BH
udgs2           EQU 0F96BH
udgs3           EQU 0F99BH
udgs4           EQU 0F9CBH
update_coordinates EQU 0FBBDH
update_sprites  EQU 0FA8BH
x_pos1          EQU 0F93FH
x_pos2          EQU 0F96FH
x_pos3          EQU 0F99FH
x_pos4          EQU 0F9CFH
xpos            EQU 0F9FBH
y_pos1          EQU 0F940H
y_pos2          EQU 0F970H
y_pos3          EQU 0F9A0H
y_pos4          EQU 0F9D0H
ypos            EQU 0F9FCH
```

En esta tabla vienen las direcciones de los puntos de entrada a las rutinas (que llamaremos con RANDOMIZE USR) y de las variables que necesitaremos alterar desde BASIC para especificar los parámetros de los sprites (posición, gráficos y colores) (que escribiremos con POKE).

Una vez hecho esto, nos vamos a crear nuestra sesión de programación en Spectaculator. Los usuarios de otros emuladores, o de BasIn u otros entornos tendrán su propia forma de obrar al respecto, pero los pasos básicos son lo mismo: situar RAMTOP por debajo de nuestras rutinas, y cargar el binario en memoria.

Lo que solemos hacer el Mojonia es abrir Spectaculator y seleccionar un modo de 48K (o uno de 128K si estás más cómodo, que los gustos son como los culos: cada cual tiene el suyo propio). Una vez listos, tecleamos CLEAR 63802 (la dirección de ensamblado menos 1), o sea, X 6 3 8 0 2 ENTER, y luego seleccionamos del menú “File” la opción “Open” (o símplemente arrastramos fourspirter.bin sobre la ventana del emulador). Tras esto, el programa nos preguntará la dirección a partir de la cual queremos ubicar nuestro binario, a lo que responderemos 63803, o sea, la dirección a partir de la que hemos ensamblado la biblioteca. Una vez hecho esto, estaremos listos para empezar, por lo que lo mejor es grabar un snapshot (antes de hacerlo, podría interesarte emplear el mismo método para importar los binarios con el UDG y el Charset, generados por ejemplo con SevenuP).

Variables de la biblioteca

Lo primero que tendremos que hacer para usar la biblioteca será definir nuestros Sprites: qué UDG usarán para dibujarse, qué colores tendrán dichos UDG, y las posiciones iniciales. Para saber dónde tenemos que escribir toda esa información tendremos que conocer qué estructuras en memoria emplea la biblioteca.

La biblioteca maneja cuatro sprites. La información para cada uno de ellos, que comprende la posición actual y la posición anterior, los cuatro UDG que forman el sprite, los cuatro atributos que lo colorean, 32 bytes para almacenar el bitmap de fondo que el sprite sobrescribe, y 4 bytes más para almacenar los atributos correspondientes, se ubica de forma contigua al principio del binario que hemos generado y ocupa 48 bytes, que se distribuyen de acuerdo a la siguiente tabla. De estos valores, algunos (como los buffers) sólo se emplean de forma interna y no necesitaremos tocarlos para nada, ya que de esa tarea se encarga de forma automática nuestra biblioteca.

Nombre	Offset	Tamaño	Descripción
UDG	0	4	UDG que forman el sprite
X	4	1	Posición X actual del sprite en pantalla (0-30)
Y	5	1	Posición Y actual del sprite en pantalla (0-22)
CX	6	1	Posición X anterior del sprite en pantalla
CY	7	1	Posición Y anterior del sprite en pantalla
buffer	8	32	Buffer con el contenido del fondo sobre el que se pinta el sprite
attrs	40	4	Atributos que forman el sprite
buffer_attrs	44	4	Buffer con los atributos del fondo sobre el que se pinta el sprite

Numeraremos nuestros cuatro sprites 0, 1, 2 y 3. Si, por ejemplo, queremos escribir en la posición X del tercer sprite (número 2), tendremos que calcular su dirección de la forma 63803 + 2 * 48 + 4 = 63903, ya que nuestro binario se ubica a partir de 63803, tenemos que saltarnos la información de los dos primeros sprites (2 * 48) y sumarle el offset a partir del que se ubica la coordenada X. Del mismo modo, para escribir los atributos del segundo sprite (el número 1), tendremos que escribir en la posición 63803 + 1 * 48 + 40 = 63891 y las tres siguientes.

Esto puede parecer tedioso, por lo que recomiendo hacer una tablita con todas estas direcciones. Para la ubicación por defecto (63803) la tabla es la siguiente. Sólo especifico los valores que necesitaremos tocar en nuestro programa, dejando fuera los buffers internos:

Sprite 1	
Nombre	Dirección(es)
UDG	63803, 63804, 63805 y 63806
Atributos	63843, 63844, 63845 y 63846
X	63807
Y	63808
CX	63809
CY	63810

Sprite 2	
Nombre	Dirección(es)
UDG	63851, 63852, 63853 y 63854
Atributos	63891, 63892, 63893 y 64894
X	63855
Y	63856
CX	63857
CY	63858

Sprite 3	
Nombre	Dirección(es)
UDG	63899, 63900, 63901 y 63902
Atributos	63939, 63940, 63941 y 63942
X	63903
Y	63904
CX	63905
CY	63906

Sprite 4	
Nombre	Dirección(es)
UDG	63947, 63948, 63949 y 63950
Atributos	63987, 63988, 63989 y 63990
X	63951
Y	63952
CX	63953
CY	63954

Con estos valores ya calculados, podremos empezar a programar nuestro juego usando la biblioteca.

Definiendo nuestros sprites

El primer paso para definir nuestros sprites consiste en especificar qué cuatro UDG y qué cuatro atributos empleará cada uno. Esta información puede ser alterada en cualquier momento, por ejemplo para hacer animaciones. En el juego de ejemplo que acompaña a la biblioteca, por ejemplo, cada vez que el jugador pulsa “izquierda” o “derecha” el programa pokea nuevos valores para el sprite número 0 consiguiendo que el gráfico mire en la dirección del movimiento.

Para definir esto, habrá que escribir los cuatro valores “UDG” y “Atributos” para cada sprite POKEando los valores correctos en las direcciones que hemos calculado anteriormente (ver tablas en el epígrafe anterior si se está empleando la ubicación por defecto).

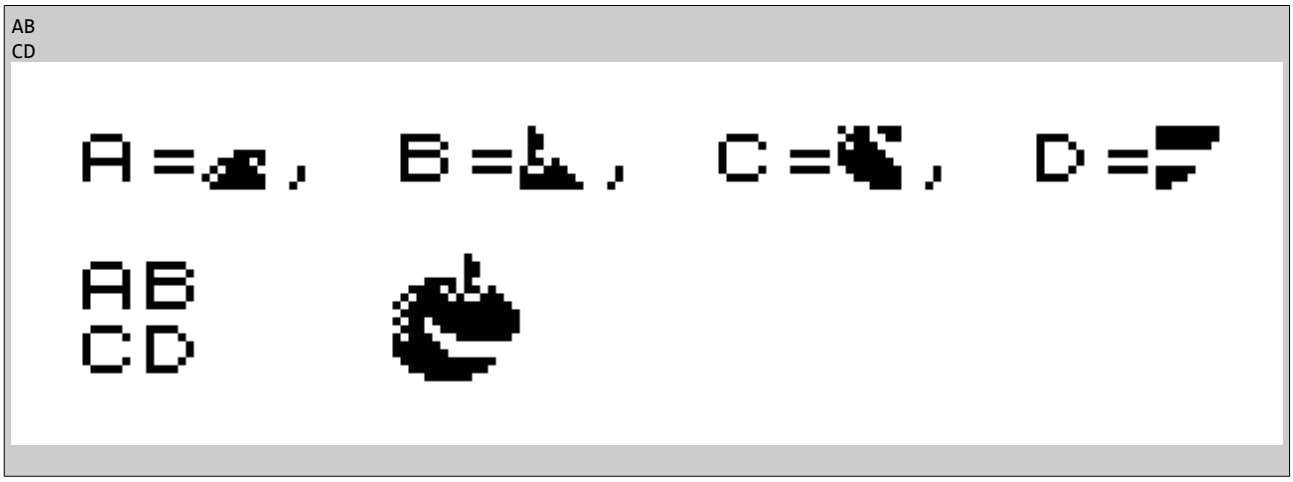
De entrada, los valores de UDG están a 99 y los de Atributos a 7. El 99 es una convención de la biblioteca: cualquier sprite cuyo primer UDG sea 99 no será dibujado ni actualizado. Esto nos permite tener menos de cuatro sprites en pantalla cuando esto sea necesario. Por ejemplo, en el juego incluido con la biblioteca sólo se emplean dos sprites, el 0 y el 1. Como el 2 y el 3 no son necesarios, sus UDG valen 99 y por ello son ignorados por **Fourspriter 1.0**.

Como hemos dicho, cada sprite emplea cuatro UDG. Los sprites se numeran de 0 a 20 (21 en total), donde el 0 representa a la A en modo gráfico y así sucesivamente. Podemos, por ejemplo, emplear los cuatro primeros UDG, A, B, C y D en modo gráfico, para dibujar un muñequito de 16x16. Para que la biblioteca emplee estos UDG para el primer sprite (el sprite 0), deberemos pokear los números correspondientes, esto es, 0, 1, 2 y 3 en las posiciones de los UDG para el sprite 0, o sea:

```
10 POKE 63803,0: POKE 63804,1: POKE 63805,2: POKE 63806,3
```

Con esto, de un plumazo, habremos activado el sprite 0 para la biblioteca, y le habremos asignado los UDG A, B, C y D, que formarán el sprite organizándose de forma natural:

```
AB
CD
```



El paso siguiente es definir sus atributos. Por ejemplo, emplearemos la esquina superior izquierda en blanco y los otros tres caracteres en amarillo. Los valores se calculan de la forma usual, o sea, INK + 8 * PAPER + 64 * BRIGHT + 128 * FLASH. Nuestros valores, por tanto, serán 7, 6, 6 y 6. Los pokearemos en las direcciones de atributos:

```
20 POKE 63843,7: POKE 63844,6: POKE 63845,6: POKE 63846,6
```

Con esto tendremos definido el aspecto del primer sprite (sprite 0). Para los demás sprites actuaremos de forma parecida, empleando las direcciones correctas de las zonas de UDG y Atributos. Por ejemplo, emplearemos los UDG I, J, K y L, que, si contamos un poco, resultan ser los correspondientes a los números 8, 9, 10 y 11, y los colorearemos de verde con la esquina superior en amarillo, o sea, 6, 4, 4 y 4:

```
30 POKE 63851,8: POKE 63852,9: POKE 63853,10: POKE 63854,11
40 POKE 63891,6: POKE 63892,4: POKE 63893,4: POKE 63894,4
```

En principio no emplearemos los sprites 2 y 3. Aunque por defecto estén desactivados (con sus UDG valiendo 99) no está de más hacerlo nosotros de forma explícita, por lo que pueda pasar. Desactivamos de forma explícita los sprites 2 y 3 pokeando un 99 como valor de su primer UDG:

```
50 POKE 63899,99: POKE 63947,99
```

Con esto habremos terminado. Tenemos definidos los sprites 0 y 1 con sus UDG correspondientes y sus atributos, y hemos desactivado los sprites 2 y 3. El siguiente paso consistirá en ubicarlos en su posición inicial y llamar a la rutina de inicialización de **Fourspriter 1.0**.

Definiendo la posición inicial de nuestros sprites

Una vez definido el aspecto de nuestros sprites y cuáles de ellos estarán activos para la biblioteca, lo siguiente es especificar su posición inicial y llamar a la rutina de inicialización, que capturará el

fondo por primera vez. Por ello, antes de hacer esto, lo suyo será dibujar nuestro fondo. Para probar, llenaremos la pantalla de caracteres al azar:

```
100 CLS: FOR i=0 TO 21: FOR j=0 TO 31: PRINT INK INT(RND*8); PAPER INT(RND*8); CHR$(65+INT(RND*26));: NEXT
j: NEXT i
```

Esto (aunque de forma un poco lenta) llenará la pantalla de mierda colorida. Ya tenemos nuestro fondo de prueba.

Para establecer la posición inicial de nuestro sprite, y esto **deberá hacerse cada vez que cambiemos el fondo de la pantalla** (por ejemplo, al pasar de una pantalla a otra, o al cambiar de fase, o lo que sea), tendremos que escribir en las variables X, Y y también los mismos valores en CX, CY.

Vamos a definir en nuestro programa BASIC cuatro variables, PX y PY almacenarán la posición del jugador, y EX y EY harán lo propio con las del segundo sprite. Estos manejes pueden hacerse directamente sobre las variables de la biblioteca, pero es mucho más cómodo operar así, como se verá. Colocaremos nuestros sprites en lugares enfrentados de la pantalla:

```
110 LET PX=5: LET PY=10: LET EX=25: LET EY=10
```

Además definiremos dos variables más para que el sprite 1 se mueva automáticamente rebotando por la pantalla:

```
120 LET EMX=1: LET EMY=1
```

Ahora viene la chicha: tenemos que pokear esos valores en sus posiciones correspondientes para el sprite 0 y el sprite 1, de modo que la biblioteca sepa dónde tiene que pintar los sprites y de dónde tiene que tomar el fondo. Consultamos una vez más nuestra tabla de direcciones y POKEamos los valores correctos:

```
130 POKE 63807,PX: POKE 63808,PY: POKE 63809,PX: POKE 63810,PY
140 POKE 63855,EX: POKE 63856,EY: POKE 63857,EX: POKE 63858,EY
```

Si os fijáis, hemos pokeado en las direcciones de X, Y, CX y CY del sprite 0 (o sea, 63807, 63808, 63809 y 63810) los valores PX, PY, PX y PY, respectivamente. Luego hemos hecho lo mismo para el sprite 1.

Una vez hecho, la biblioteca ya sabe dónde ubicar los sprites, así que llamaremos a la rutina de inicialización. La rutina de inicialización se llama init_sprites. Para saber dónde está, miraremos al archivo fourspriter.txt que generamos antes, al ensamblar. Para la ubicación estándar de la biblioteca, el punto de entrada a init_sprites es 0F9FFh o, lo que es lo mismo, 63999 en decimal:

```
150 RANDOMIZE USR 63999
```

Con esto ejecutamos la inicialización de la biblioteca, que mirará qué hay en la pantalla en la posición X, Y de cada sprite activo (el 0 y el 1 en nuestro ejemplo) y lo almacenará en los búfferes correspondientes.

Moviendo nuestros sprites

Una vez que todo está definido, mover nuestros sprites es tan sencillo como actualizar sus coordenadas X e Y, y llamar a la rutina update_sprites, cuya dirección miraremos en la tabla de símbolos fourspriter.txt que generamos antes. Para la ubicación por defecto, la entrada a la rutina update_sprites es 0FA8Bh, o 64139 en decimal.

El movimiento en sí de los sprites lo programaremos en BASIC. Para probar, haremos que el sprite 0 se controle con O, P, Q y A y el sprite 1 rebote por la pantalla. Primero programamos el comportamiento del sprite 0. Además, actualizaremos la definición del gráfico de este sprite en tiempo real: al pulsar O haremos que mire haciaa la izquierda, usando los UDG E, F, G y H (números 4, 5, 6 y 7) y al pulsar P haremos que mire hacia la derecha, usando los UDG A, B, C y D (números 0, 1, 2 y 3):

```
190 POKE 23658,8: REM TECLADO EN MAYUSCULAS
200 IF INKEY$=”O” THEN IF PX>0 THEN LET PX=PX-1: POKE 63803,4: POKE 63804,5: POKE 63805,6: POKE 63806,7
210 IF INKEY$=”P” THEN IF PX<30 THEN LET PX=PX+1: POKE 63803,0: POKE 63804,1: POKE 63805,2: POKE 63806,3
220 IF INKEY$=”Q” THEN IF PY>0 THEN LET PY=PY-1
230 IF INKEY$=”A” THEN IF PY<22 THEN LET PY=PY+1
```

Seguidamente, programaremos el comportamiento del sprite 1 que rebotará por la pantalla, sin alterar su gráfico (que definimos antes para emplear los UDG I, J, K y L):

```
250 LET EX=EX+EMX: IF EX=0 OR EX=30 THEN LET EMX=-EMX
260 LET EY=EY+EMY: IF EY=0 OR EY=22 THEN LET EMY=-EMY
```

Una vez programado el movimiento de ambos sprites, haremos lo que explicábamos al principio de este epígrafe: actualizar las coordenadas X e Y del sprite0 y el sprite1 en las variables de la biblioteca, y llamar a la rutina update_sprites. Volvemos a las tablas de direcciones para recordar las de X e Y para sprite0 y sprite1 y pokeamos los valores necesarios, para posteriormente llamar a la rutina y volver al principio del bucle del juego, en la linea 200:

```
300 POKE 63807,PX: POKE 63808,PY
310 POKE 63855,EX: POKE 63856,EY
320 RANDOMIZE USR 64139
330 GOTO 200
```

Como véis, las variables CX y CY sólo es necesario establecerlas la primera vez, antes de llamar a la rutina de inicialización de **Fourspriter 1.0**. En el bucle del juego, sólo necesitamos actualizar X e Y, ya que CX y CY son empleadas de forma interna por la rutina update_sprites para recordar la última posición y restaurar el fondo correctamente.

Si ejecutamos nuestro programa veremos la rutina en funcionamiento. ¡No tendremos que hacer nada más!

Este sencillo programa de demostración está también incluido en el paquete, dentro del archivo simple_demo.z80, pero recomendamos programarlo a mano y paso a paso para entender bien qué hace cada POKE y cada RANDOMIZE USR. Este snapshot tiene incluido un binario con UDGs para que quede más bonito.

Consideraciones de interés

Esta primera versión de **Fourspriter 1.0** emplea UDG para dibujar sus sprites. Para ello, las rutinas de dibujado miran en la variable del sistema "UDG", situada en las direcciones 23675 y 23676 para saber dónde están definidos. Esto significa que podremos usar varios sets de UDG de la misma forma que lo solemos hacer desde BASIC, esto es, pokeando la variable del sistema "UDG" con la dirección de los sets alternativos.

Otra cosa que puede ser interesante documentar (porque he tenido que emplear esta información para la elaboración del juego de demostración incluido en el paquete) son los otros putos de entrada a las rutinas.

La rutina update_sprite llama a las siguientes subrutinas, en orden:

- borra_sprites: Esta subrutina se encarga de pintar los fondos almacenados en los bufferes en las anteriores posiciones (CX, CY) de cada sprite.
- init_sprites: Esta subrutina se encarga de capturar los fondos en pantalla en las posiciones actuales (X, Y) de cada sprite.
- draw_sprites: Esta subrutina se encarga de dibujar los UDGs y los atributos en las posiciones actuales (X,Y) de cada sprite.
- update_coordinates: Esta subrutina se encarga de actualizar CX y CY a los valores de la posición actual, esto es, hace CX=X: CY=Y.

Si cambiamos algo en la pantalla, por ejemplo hacemos desaparecer un objeto porque el jugador lo ha cogido, es conveniente que llamemos a estas funciones de forma ordenada siguiendo este orden:

1. RANDOMIZE USR borra_sprites
2. Alterar el fondo
3. RANDOMIZE USR init_sprites
4. RANDOMIZE USR draw_sprites
5. RANDOMIZE USR update_coordinates

De esta forma la biblioteca se comportará de forma consistente con el cambio en el fondo y no obtendremos artefactos ni cosas raras. Los puntos de entrada a todas estas subrutinas los tenemos, como todo, en nuestra tabla de símbolos fourspriter.txt

Y con esto y un bizcocho...

Nos quedamos aquí esperando a que algún valiente se atreva a plasmar su creatividad en un juego BASIC empleando nuestra biblioteca. Huelga decir que el código BASIC que emplee **Fourspriter 2.0** es totalmente compatible con los compiladores HiSoft BASIC y Mcoder 3. Además, la biblioteca es muy fácilmente adaptable a otros lenguajes y/o compiladores como ZX BASIC de Boriel o z88dk, entre otros. Todos los conceptos tratados en este manual son perfectamente aplicables.

Happy coding!