“HAPPY MONTY”

making of

1 pantalla de carga del juego

Índice

1 Introducción y retos

El videojuego “happy monty” ha sido programado en Locomotive BASIC usando la librería 8BP (8 bits de poder). 8BP mantiene una relación de cooperación con la AUA de modo que ambos logos son mostrados con el objeto de difundir la cultura y valores de el “arte retro”.

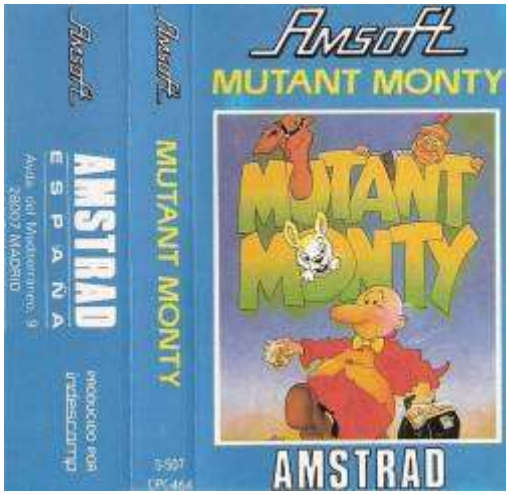
La librería 8BP son un conjunto de rutinas escritas en ensamblador que se pueden usar desde BASIC gracias a las extensiones RSX. Es posible igualmente usar estas rutinas desde cualquier otro lenguaje e incluso desde un programa en ensamblador. El videojuego Happy Monty esta creado en Locomotive BASIC y se ejecuta interpretado, no compilado.

La música ha sido creada con WYZtracker 2.0.1.0 ya que dentro de la librería 8BP esta integrado el player de WYZ.

La lógica BASIC del juego ha ocupado 17 KB aproximadamente, aunque el código fuente incluye muchos comentarios de modo que se podría reducir algo.

Los retos de este desarrollo han sido:

- **Lograr una velocidad aceptable de juego**. Teniendo en cuenta que la lógica esta en BASIC y que el intérprete esta consumiendo muchísimo tiempo, la estrategia de programación de lógicas masivas ha resultado indispensable. La velocidad alcanzada depende de cada pantalla. Hay pantallas a 30 FPS y otras que solo alcanzan 18 FPS. También se aprecia como al mover el personaje bajan algo los FPS, asi como al activar la música (tecla “m”), que es opcional.
- **Incluir un numero elevado de niveles**: se han incorporado 25 niveles mediante una descripción compacta de los mismos que tan solo gasta 160 bytes por nivel. Los niveles se numeran del 0 al 24. El nivel 0 esta inspirado en el primer nivel del videojuego “mutant Monty” creado por John Price en 1984 para la compañía Artic Computing. En España fue distribuido por Amsoft El juego happy Monty destina en total 25*160bytes= 4kB para todo el mapeado de pantallas.



2. carátula del clásico “Mutant Monty”

- **Rutas versátiles:** Para evitar definir decenas de rutas de enemigos de distinta longitud, se ha optado por definir unas pocas rutas básicas (vertical y horizontal de 3 velocidades) y unos **sprites invisibles que actúan a modo de “inversores**”, de modo

que cuando un enemigo colisiona con un inversor, cambia la dirección de movimiento. Este mecanismo de los “inversores” ha también posibilitado la creación de rutas con giros de 90 grados, pues se han definido inversores para cambiar a la dirección opuesta y también inversores para cambiar de ruta y así implementar los giros que dan algunos enemigos como los que hay en el nivel 1.

- **Control del personaje “avanzado”:** si pulsas una tecla, el personaje se mueve despacio, pero si la dejas pulsada unos instantes, Monty acelera y va más deprisa.

2 Construcción de los 25 niveles y sprites “inversores”

El videojuego posee 25 niveles, pero una sola lógica para todos ellos.

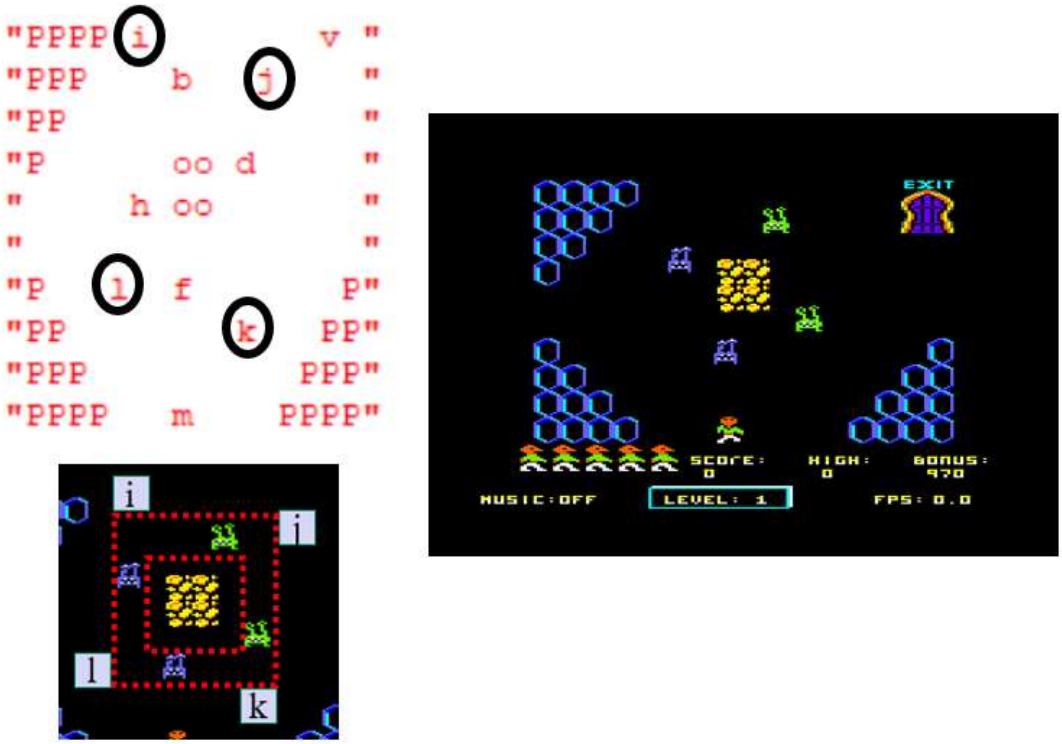
Cada nivel se ha definido con un conjunto de letras tanto los muros y elementos ornamentales como los sprites enemigos (“a”, “b,”, ”c”, ”d,”, “e”, ”f”, ”g”, ”h”) el oro(”o”), los sprites “Inversores” (“z”, ”x”, ”q”, ”w”) y la posición de Monty (“m”). En el siguiente ejemplo podemos ver la definición de un nivel, que como se puede comprobar, ocupa 160 caracteres y es muy fácil de editar.

Era posible definir el nivel de un modo aun mas compacto, gastando unos pocos bits por bloque en lugar de un carácter. Sin embargo, al utilizar caracteres, la creación de nuevos niveles ha sido una tarea mucho más sencilla.

```
"IK m    o     JG"
" IGGGGGH IGGGGHq'
" z  c     xb    '
" C CCGK        d'
" F oC IDD DDDDD  '
" F  C  F      F  '
" Fv C JHz b xoF  '
" F  C IGGGGGGGH  '
" IGGH z    a   xw'
"      EEEEEoEEE  "
```

3. Nivel 0 del juego y su definición con 160 bytes

En el siguiente nivel se ven los sprites inversores capaces de hacer girar a los enemigos 90 grados al colisionar con ellos, con lo que se producen rutas de enemigos más “peligrosas”. Los sprites “inversores” son los denominados con las letras “i”,”j”,”k”,”l”. Estos sprites están ahí pero no se ven porque no tienen el flag de impresión activo en el byte de estado del sprite.



4. Sprites inversores de tipo "codo"

2.1 Construcción de niveles y creación de enemigos

Los niveles son cadenas de 160 caracteres que se leen desde la posición 20000 hasta la 24000 (hay 25 niveles y por tanto ocupan 160x25=4000)

Para leer cada nivel se usa la rutina "Level builder" que comienza en la línea 910. Según las letras que va leyendo con PEEK, va transformando el código ASCII de cada letra en un sprite:

- Si es una letra mayúscula se trata de un muro a imprimir con LAYOUT. Hay muchos tipos de muros, tantos como letras mayúsculas son utilizadas. Cada letra se corresponde con un sprite ID (tal y como se explica en el manual de programación 8BP , donde se explica el comando layout)
- Si es una letra minúscula, se trata de un enemigo, un sprite inversor, una pieza de oro, una puerta de salida o el mismo Monty (así lo ubicamos al empezar cada nivel donde queramos)

Al entrar en cualquier nivel, se invoca a la rutina ubicada en la línea 1250, que a partir de las letras minúsculas leídas en ese nivel, crea los sprites de cada tipo (enemigos, oros, puerta, etc). Para decidir la secuencia de animación asociada a cada enemigo se usa una "paleta de enemigos" que se va cambiando de nivel en nivel. De ese modo la letra "a" significa "enemigo lento que camina hacia la derecha" pero es la paleta de enemigos la que decide en función del nivel asociarle un extraterrestre, una rana, una ardilla o una araña

La rutina de la línea 1405 se encarga de asignar las secuencias de animación a los enemigos que se mueven en horizontal byte a byte (H1) , de dos en dos bytes (H2) y en vertical (V2). Las secuencias de animación están definidas en sequences_mygame.asm. Por ejemplo, la secuencia 6 es la del extraterrestre con los dos ojos saltones estilo caracol y la 9 es una rana. La 11 es una especie de ardilla (o un perro, no se muy bien a que se parece más)

```
1405 '--- paleta aliens (secuencias de animacion)
1406 alienH1(0)=6:alienH2(0)=7:alienV2(0)=8
1407 alienH1(1)=6:alienH2(1)=7:alienV2(1)=8
1408 alienH1(2)=9:alienH2(2)=7:alienV2(2)=8
1409 alienH1(3)=9:alienH2(3)=7:alienV2(3)=8
1410 alienH1(4)=9:alienH2(4)=7:alienV2(4)=10
1411 alienH1(5)=6:alienH2(5)=7:alienV2(5)=8
1412 alienH1(6)=6:alienH2(6)=7:alienV2(6)=8
1413 alienH1(7)=11:alienH2(7)=7:alienV2(7)=8
1414 alienH1(8)=9:alienH2(8)=7:alienV2(8)=12
1415 alienH1(9)=11:alienH2(9)=7:alienV2(9)=10
1419 return
```

3 Ciclo de juego y Lógicas masivas

Las "lógicas masivas" están presentes en todo el desarrollo: desde las interioridades algorítmicas de los comandos propios de la librería 8BP (tales como la colisión de sprites) hasta la forma en que se lee el teclado (**en ciclos pares se leen las teclas Q,A y en ciclos impares las O,P**.

Además, **cada 6 ciclos** se comprueban las colisiones con el oro, y la puerta de salida, evitando hacerlo en cada ciclo y de ese modo acelerando el juego. La pulsación de la tecla "m" para activar y desactivar la música se comprueba también **cada 6 ciclos**. Otras de las cosas que tienen que ver con esta técnica es la **detección de colisión con el layout, que solo se realiza cuando las coordenadas (x,y) de Monty son múltiplos de 4 y 16 respectivamente**, de modo que solo si se encuentra en una bifurcación se permite el cambio de dirección del personaje. **Esto ahorra tiempo de cómputo** al tiempo que impide que el personaje pueda quedar parado entre dos muros. Cuando Monty se para, lo hace siempre en una posición múltiplo de 4 (en X) y múltiplo de 16 (en Y).

Para acelerar aún más, las rutas de enemigos "lentos" (especificados con las letras "a" y "e" en los mapas) , **fuerzan cada 2 frames la desactivación del flag de impresión** que utiliza el comando PRINTSPALL de la librería 8BP, de modo que **hay ciclos en los que no se imprimen si no se han movido, ahorrando algo de tiempo**. Esto se ha hecho gracias a las capacidades que tiene 8BP para definir rutas de sprites.

Para acelerar un poco mas el juego, se ha perfeccionado la librería 8BP (v37), acelerando algunos comandos como COLAY y MOVER. En el caso de COLAY (rutina que detecta colisiones con el layout), se le ha dotado de "memoria", de modo que no es necesario invocarlo con parámetros si son los mismos que la ultima vez. Esta es una característica que comparten muchos otros comandos de 8BP, tales como PRINTSPALL o STARS, pero faltaba en COLAY. EL paso de parámetros es una de las cosas mas costosas en BASIC pues el interprete analiza cada parámetro, su formato, etc. Y conviene evitarlo si es posible.

Por último, muchos de los comandos RSX de la librería 8BP en el ciclo de juego se han reemplazado por su correspondiente CALL, lo cual permite acelerar algo su ejecución, aunque a costa de que el código fuente sea algo menos legible.

El ciclo de juego tiene 3 partes:

- Todos los ciclos: Impresión de sprites, detección de colisiones y lógica de enemigos
- Lectura de teclado en ciclos impares y acciones de monty
- Lectura de teclado en ciclos pares y acciones de Monty

3.1 Todos los ciclos

Este código empieza en la línea 400. La variable "c" es el ciclo de juego que se incrementa 1 cada vez.

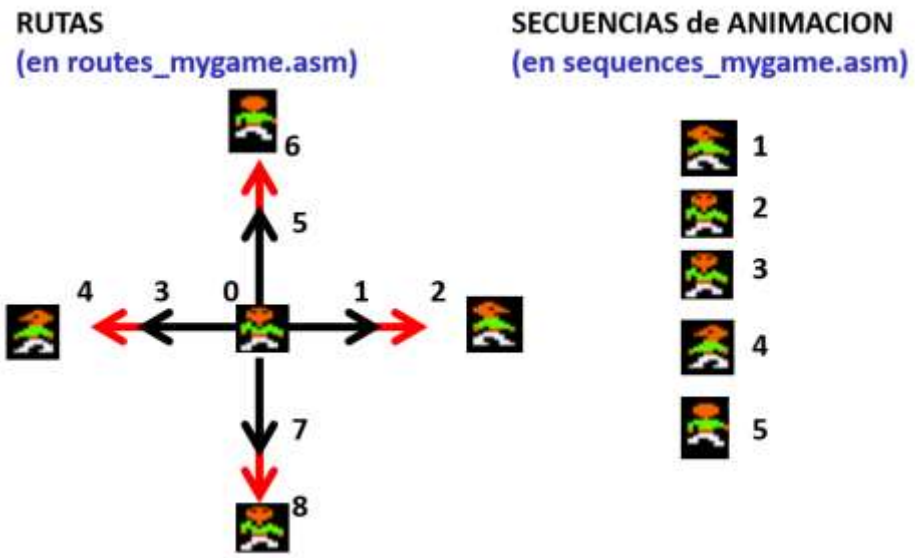
```
400 '--- ciclo de juego
410 c=c+1:|AUTOALL,1:|PRINTSPALL:|COLSPALL:IF cor=32 THEN 500
```

El código que se ejecuta mueve los sprites (AUTOALL,1), anima e imprime los sprites (PRINTSPALL) y detecta colisiones (COLSPALL). A continuación, si se ha colisionado un sprite enemigo con un sprite inversor, se cambia la ruta del enemigo. Por ejemplo, si un fantasma que viaja hacia la derecha se choca con un sprite inversor, el fantasma cambia su ruta y desactiva el sprite inversor con el que ha colisionado. También activa el sprite inversor contrario. **Esto permite que solo uno de los dos inversores este activo a la vez y con ello el comando COLSPALL puede hacer su trabajo mas deprisa**. **Es un claro ejemplo de "lógicas masivas".**

A priori un fantasma no sabe quienes son los sprites inversores que va a tener a su derecha e izquierda (si su ruta es horizontal) o arriba y abajo (si su ruta es vertical). Por ello, cada sprite al colisionar rellena un array donde guarda quienes son los inversores con los que choca, para poder desactivarlos/activarlos estos arrays son **corx(<sprite_id>)** y **corz(<sprite_id>)** y representan los inversores de tipo "x" (izquierda) y "z" (derecha) de un enemigo horizontal o vertical

3.2 Ciclos impares

Leemos las teclas "O" y "P" y cambiamos la secuencia de animación de Monty en consecuencia. Aquí es importante mencionar que Monty tiene dos velocidades. La primera vez que pulsas la tecla "P", Monty adquiere una velocidad lenta (se mueve cada dos ciclos) pero si la dejas pulsada, se mueve en todos los ciclos. Las **rutas asociadas** a cada dirección y velocidad asi como las **secuencias de animación** son las siguientes :



5. rutas de movimiento y secuencias de animacion de Monty

Cada vez que pulsas una tecla se actualiza la variable "task" . Tras actualizarla, se comprueban las coordenadas de Monty. Si la coordenada X no es múltiplo de 4 y la coordenada Y no es múltiplo de 16, la tarea no se lleva a cabo, aunque queda apuntada para cuando las coordenadas cumplan esa condición que es indicativo de que estas en una posible bifurcación (líneas 720 a 732).

```
720 IF vx THEN 721 else 730
721 x=PEEK(27499)+mx(r)
722 IF x AND 3 THEN 410 ELSE 760
730 if vy then 731 else 760
731 y=PEEK(27497)+my(r)
732 IF y AND 15 THEN 410
```

Hay un caso en el que la respuesta es inmediata con independencia de las coordenadas y es que la tarea sea ir en dirección contraria a la actual. En ese caso, se cambia la dirección de Monty inmediatamente. Antes de cambiar de dirección hay que corregir la coordenada de Monty, ya que cuando va hacia la derecha, el dibujo de Monty comienza 1 byte más allá de su coordenada, debido a que tiene una columna de bytes negros para borrarse a si mismo mientras camina. Para ello se usa unas variables mx() y my() que contienen las correcciones correspondientes para cada dirección.

3.3 Ciclos pares

En lo que respecta a la lógica de Monty son iguales que los impares salvo por que se comprueban en este caso las teclas Q y A (para ir en dirección vertical)

Además de eso, en los ciclos pares se comprueba si estamos ante un ciclo múltiplo de 6. En caso afirmativo se procede a comprobar la colisión de Monty con las piezas de oro. Esto no se hace en todos los ciclos debido a que comprobar las colisiones con todos los enemigos y con los oros a la vez restaría algo de velocidad y he tratado de acelerar el juego al máximo (filosofía de **lógicas masivas**). Pues bien, si estamos ante un ciclo múltiplo de 6 se comprueba la

colisión con el oro reconfigurando el comando COLSP para colisionar a Monty con todas las piezas posibles de oro. Hay que tener en cuenta que:

- Los sprites de los enemigos van del 0 al 5 (máximo 6 enemigos)
- Los oros van del 6 al 13
- La puerta de salida es el 14
- Los inversores van del 18 al 29 (máximo 12 inversores)
- Monty es el 31

Si observamos las primeras líneas del ciclo par, se reconfigura la colisión y se lanza el comando COLSP,31 que comprueba la colisión del sprite 31 (Monty) únicamente

```
619' --- CICLO PAR
620 IF c mod 6 THEN 639
621 |COLSP,32,6,14:|COLSP,31:' configura comando y comprueba
colision de monty
622 IF cod=32 then |COLSP,32,0,5:goto 630:' no hay colision con oros
ni puerta
```

Tras realizar esto se lee la tecla “m” para activar o desactivar la música. Tambien se comprueba esto cada 6 ciclos. Por ultimo se comprueban los FPS leyendo cada 100 ciclos el tiempo del sistema. Según el tiempo transcurrido se calculan e imprimen los fps en pantalla, para dar información al jugador “friki” del rendimiento del juego. Esto se hace en la línea 635

```
635 if c>100 then fps=10*c*300/(TIME-A):f1=fps/10:f2=fps mod
10:C$=STR$(F1)+"."+right$(str$(f2),1):|PRINTAT,0,24*8,67,@c$: gosub
1700:c=0
```

Después de comprobar todo esto , se leen las teclas Q y A y se opera en consecuencia, mediante el mismo esquema que se ha usado para las teclas OP (apuntando la tarea, comprobando si estamos en bifurcación , etc.)

4 Los 25 niveles de happy Monty

En el último nivel aparece la princesa, que no está presente en el resto de niveles. Tras recoger el oro y dirigirnos a ella nos espera un épico final

Los 25 niveles de Happy Monty son los siguientes:


EXIT
SCORE: 0
HIGH: 0
BONUS: 990
MUSIC:OFF
LEVEL: 2
FPS:....


EXIT
SCORE: 0
HIGH: 0
BONUS: 990
MUSIC:OFF
LEVEL: 3
FPS:....


EXIT
SCORE: 0
HIGH: 0
BONUS: 990
MUSIC:OFF
LEVEL: 4
FPS:....


EXIT
SCORE: 0
HIGH: 0
BONUS: 990
MUSIC:OFF
LEVEL: 5
FPS:....


EXIT
SCORE: 0
HIGH: 0
BONUS: 990
MUSIC:OFF
LEVEL: 6
FPS:....


EXIT
SCORE: 0
HIGH: 0
BONUS: 990
MUSIC:OFF
LEVEL: 7
FPS:....


SCORE: 0
HIGH: 0
BONUS: 980
MUSIC:OFF
LEVEL: 8
FPS:....


EXIT
SCORE: 0
HIGH: 0
BONUS: 990
MUSIC:OFF
LEVEL: 9
FPS:....


EXIT
SCORE: 0
HIGH: 0
BONUS: 990
MUSIC:OFF
LEVEL: 10
FPS:....


SCORE: 0
HIGH: 0
BONUS: 990
MUSIC:OFF
LEVEL: 11
FPS:....


EXIT
SCORE: 0
HIGH: 0
BONUS: 990
MUSIC:OFF
LEVEL: 12
FPS:....


EXIT
SCORE: 0
HIGH: 0
BONUS: 990
MUSIC:OFF
LEVEL: 13
FPS:....


EXIT
SCORE: 0
HIGH: 0
BONUS: 990
MUSIC:OFF
LEVEL: 14
FPS:....


EXIT
SCORE: 0
HIGH: 0
BONUS: 970
MUSIC:OFF
LEVEL: 15
FPS: 0.0


EXIT
SCORE: 0
HIGH: 0
BONUS: 990
MUSIC:OFF
LEVEL: 16
FPS:....


EXIT
SCORE: 0
HIGH: 0
BONUS: 990
MUSIC:OFF
LEVEL: 17
FPS:....


EXIT
SCORE: 0
HIGH: 0
BONUS: 990
MUSIC:OFF
LEVEL: 18
FPS:....


EXIT
SCORE: 0
HIGH: 0
BONUS: 980
MUSIC:OFF
LEVEL: 19
FPS:....

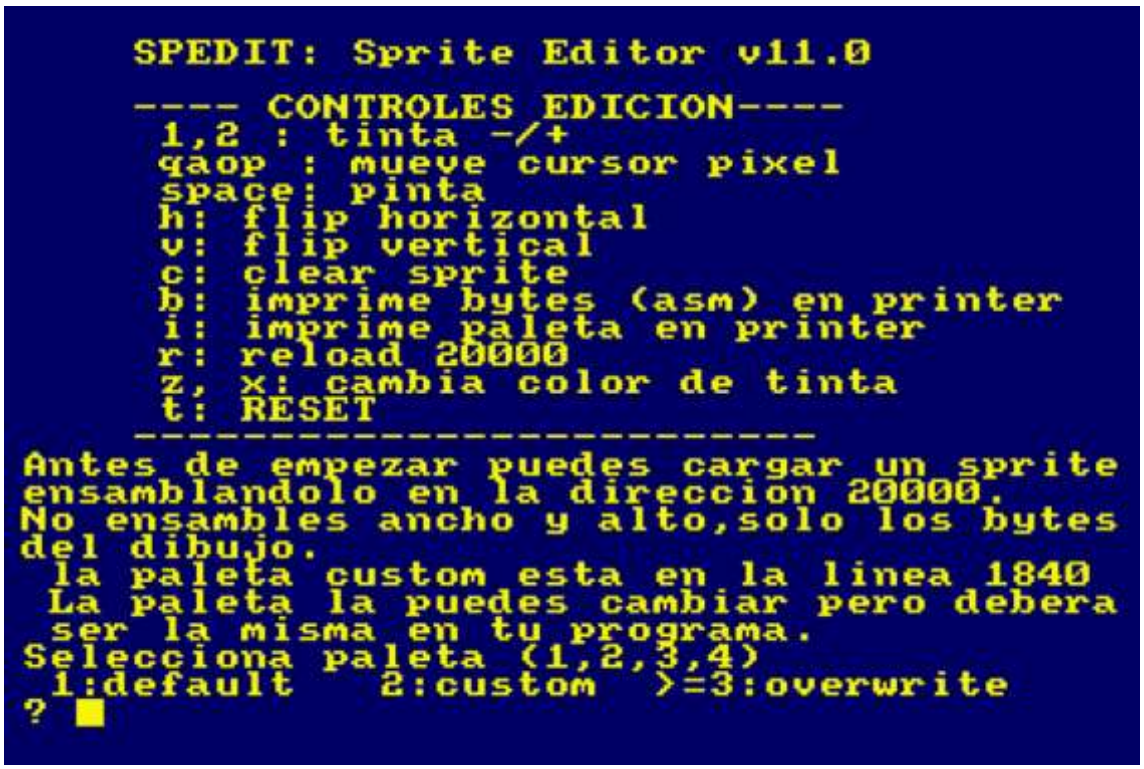

EXIT
SCORE: 0
HIGH: 0
BONUS: 990
MUSIC:OFF
LEVEL: 20
FPS:....


EXIT
SCORE: 0
HIGH: 0
BONUS: 990
MUSIC:OFF
LEVEL: 21
FPS:....



5 Diseño de sprites y pantalla de carga

Todos los diseños de sprites y decorados han sido construidos con la herramienta SPEDIT, que viene incluida en 8BP. SPEDIT es una herramienta sencilla escrita en BASIC que permite editar en mode 0 y mode 1, permite espejar sprites, editar y generar código fuente para hacer paletas, etc. Se ejecuta en el mismo AMSTRAD (dentro del emulador winape)



Por último, la pantalla de carga ha sido realizada con la herramienta convImgcpc v0.14

6 Mapa de memoria del juego

El mapa de memoria del juego, es el siguiente:

- 0000 hasta 24000 : libre para BASIC. el videojuego en BASiC ocupa 17 KB
- **20000 hasta 240000: los 25 niveles, a 160 bytes cada uno**
- 24000 hasta 32200: librería 8BP
- 32200 hasta 33600 : músicas (hay 2 músicas, la segunda suena al final del juego)
- 33600 hasta 42040: gráficos
- 42040 hasta 42540: aquí se almacena el layout que se esté ejecutando. Aunque una pantalla ocupa 160 bytes, se usan 500 bytes como buffer de layout en 8bp
- 42540 hasta 42619 : banco de estrellas (se usan en el final del juego)