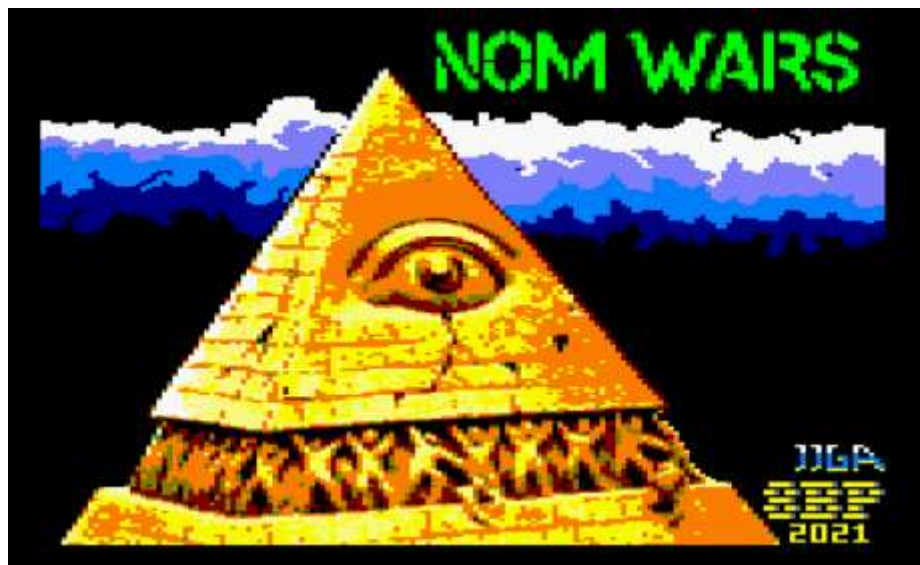


8 BITS DE PODER

Como programar “NOMWARS”



V41

Jose Javier García Aranda

Contenido

1	HISTORIA DE COMMANDO	2
2	VERSIÓN PROGRAMADA CON 8BP	5
3	DISEÑO GENERAL	8
4	LÓGICA DEL PROGRAMA PRINCIPAL.....	10
5	LÓGICA DEL CICLO DE JUEGO	14
5.1	Ciclo de juego	14
5.2	Control del personaje.....	15
5.3	Cómputo de la secuencia de animación	16
5.4	Disparo del personaje	19
5.5	Mapa y eventos.....	20
5.6	Mecanismo del puente.....	23
6	RUTAS.....	24
7	¡A JUGAR Y A PROGRAMAR!	26

1 Historia de commando

Juego creado en 1985 creado por Capcom. Un clásico “shoot ‘em up” de scroll vertical. Fue un éxito comercial y de crítica y no podía faltar en una sala de juegos recreativos.

Nuestro personaje es un soldado llamado “Joe”, al que un helicóptero deja en una jungla y tiene que luchar contra un ejército casi infinito de enemigos, usando su rifle de asalto y granadas. Aunque nadie diga que está ambientado en la guerra del vietnam, el entorno inicial de la jungla podría sugerir eso. Hubo una película en 1985 titulada como este juego, de Arnold Schwarzenegger, pero **no tiene nada que ver con el juego**, es una casualidad. Hay quien dice que cambiaron el título original japonés “lobo del campo de batalla” por “comando” para aprovechar el tirón de la película, pero la película es de diciembre de 1985 y el juego de mayo de 1985. No parece que tenga relación, aunque “las casualidades no existen”.



Una de sus señas de identidad es el famoso puente que debe atravesar Joe, y que debe aparecer en cualquier clon que se precie.

Un ejemplo de que las casualidades no existen lo podemos ver en la caratula del juego RASTAN



Rastan...¿es conan? ¿Está inspirado en conan? Puede ser, porque se parecen un pelín. Puede que sean como Clark kent y superman, que se parecen pero no son el mismo.

ELITE hizo versiones para todos los ordenadores de 8 y 16 bits y también consolas. La conversión de Commodore Amiga (16 bits) destaca porque es prácticamente idéntica al original, pero en el resto de conversiones hay notables diferencias. Los juegos aparecieron en noviembre de 1985 (antes de la película commando)

Versión ZX



Version CPC



Versión C64



Versión MSX



Versión BBC micro



Versión Atari 7800



Versión NES



Versión commodore Amiga



Podemos destacar la versión de ZX por su rapidez y buenos gráficos (a pesar de ser una máquina sin sprites es una máquina rápida “gracias” a su escasa memoria de video) y la de Amstrad por que tiene mérito lograr un juego con scroll vertical rápido, música y multitud de sprites en esta máquina. La versión de c64 aprovecha su hardware (scroll y sprites). La peor versión para mi gusto es la de MSX, que como ocurría con muchos otros juegos, tenía un scroll

a saltos muy incómodo para jugar. Otra versión malilla es la de BBC micro, pero ya sabéis que esto va en gustos y a veces lo cutre puede incluso gustar más.

Hay quien considera otro clásico “Ikari Warriors” como un clon de Commando pero posee muchas diferencias. Manejas un soldado, el scroll es vertical...pero eso no basta para ser un clon. Ikari Warriors tiene su propia “personalidad”.

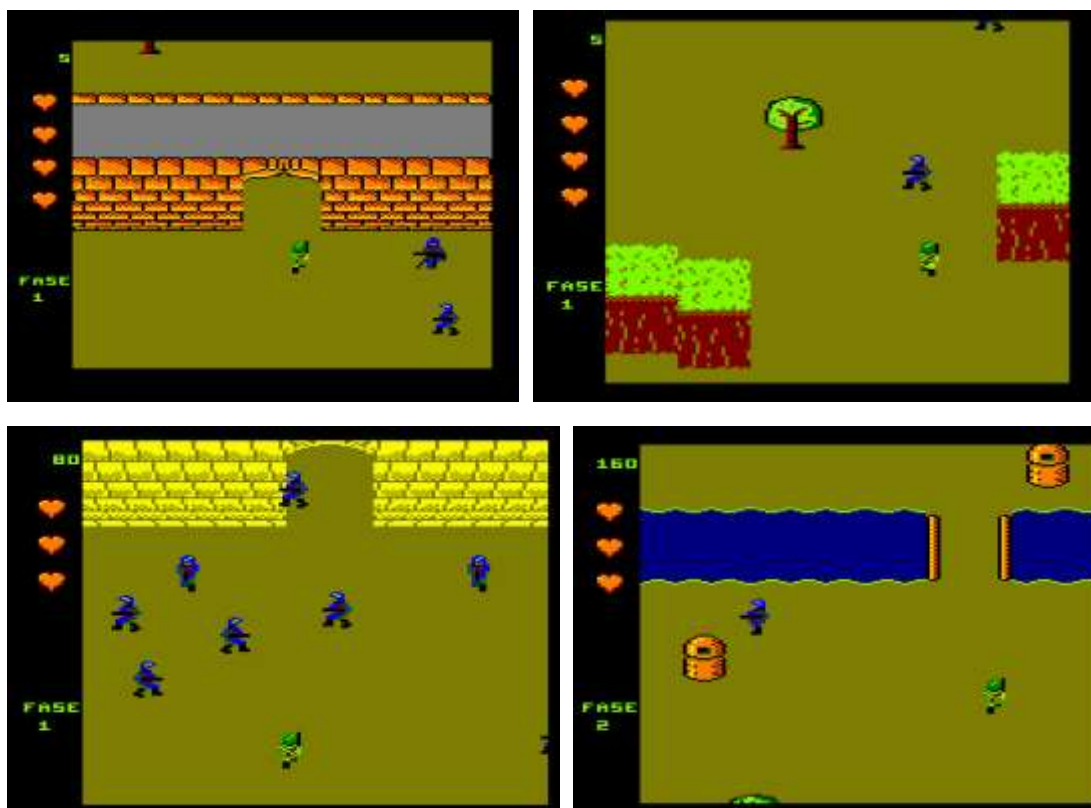
2 Versión programada con 8BP

La versión programada con 8BP ha sido titulada “NOMWARS”, por estar de moda las historias sobre el nuevo orden mundial como amenaza. El título es lo de menos porque igual se podría haber titulado “el retorno de Joe” o “supercommando”, no deja de ser un clon de Commando.

Ojo, no es un juego mejor que el original, es simplemente un clon. Tiene interés por ser divertido, rápido y haber sido programado en BASIC con 8BP, demostrando una vez más que con 8BP puedes hacer cualquier tipo de juego.

En esta versión se explota la capacidad de scroll de 8BP (comando MAP2SP) y se incluye el famoso puente por el que Joe pasa por debajo usando una técnica basada en el comando SETLIMITS de 8BP.

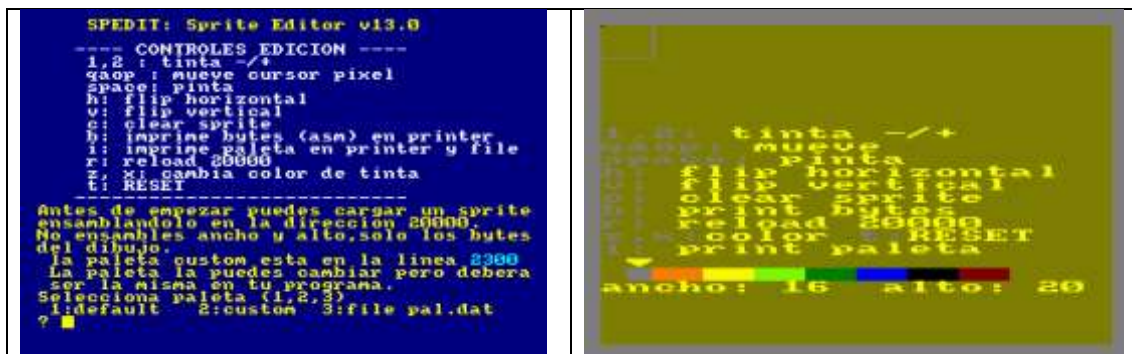
El juego se ofrece en dos versiones: la versión de BASIC pura y la versión de ciclo compilado (ciclo traducido a lenguaje C usando el wrapper de 8BP y el minibasic de 8BP). Ambas versiones son idénticas, pues la traducción a C es una réplica total, casi un espejo de la versión BASIC. De hecho, el juego ha sido programado en BASIC y el último día ha sido pasado a C con la facilidad que tiene para ello 8BP.



Algunos elementos del juego original no han sido incluidos, tales como los jeeps o los camiones, pero el juego refleja bastante bien el espíritu original. Este “clon” de commando se compone de cuatro fases.

Los sprites han sido diseñados con el spedit13 (herramienta que viene con 8BP) . En este caso al tratarse de la versión 13, te permite elegir la paleta y reutilizarla de una ejecución a otra pues se salva como pal.dat dentro del disco Amstrad y así la puedes volver a usar sin redefinirla.

Muchas imágenes se han creado a través de la capacidad de 8BP para “espejar” imágenes, de modo que la secuencia de animación de Joe caminando hacia la izquierda se compone de las mismas imágenes que la secuencia de caminar hacia la derecha espejando. Así no se gasta memoria.



Las imágenes espejadas las tienes en el fichero images_mygame.asm Aquí puedes ver dos de ellas, aunque son muchas las imágenes definidas como “espejo” de otras

```

;=====
_BEGIN_FLIP_IMAGES
;=====
; aqui pon las imagenes que se definen como otras existentes pero flipeadas
horizontalmente.
; los frames del soldado a la izquierda los defino como flipeados. es mas
lento pero gasta menos ram
JOE_UPL1 DW JOE_UPR1
JOE_UPL2 DW JOE_UPR2

```

La música se ha creado con el WYZtracker, creado por augusto ruiz, que también viene con 8BP (y el player de música está integrado en la librería a través del comando MUSIC)

El juego acepta joystick y teclado, sin necesidad de selección. Si pulsas <fire> en tu joystick, se ejecuta en modo joystick y si pulsas la barra espaciadora, el juego se ejecuta en modo teclado, siendo las teclas las famosas QAOP + barra

Si mantienes el fuego pulsado unos instantes podrás lanzar una bomba. Tienes bombas ilimitadas

La versión física editada en DES incluye una intro muy inquietante sobre la historia y los planes del NOM, y ofrece un menú para elegir la versión a la que deseas jugar (puro BASIC o ciclo compilado).

Historia de NOMWARS:

Siglo XXI, el Nuevo Orden Mundial ha creado una pandemia y se ha hecho con el control de los medios, ha arruinado la economía y está utilizando técnicas de ingeniería social para que la población acepte un régimen totalitario basado en la tecnología. Mediante un plan de vacunación masiva están esterilizando silenciosamente a la población. Cualquier fiesta está prohibida.

Quedan pocos pensadores libres como tú, y tu misión es unirse a una fiesta clandestina, para lo cual deberás destruir al ejército del nuevo orden mundial. Una vez destruido, la humanidad te agradecerá haber roto sus cadenas

Me consta que hay gente que le ha “alertado” la historia de NOMWARS, porque contradice lo que se dice en los medios de comunicación sobre el NOM, pero una vez más quiero dejar claro que NOMWARS es un videojuego, no es un libro de actualidad geopolítica ni de historia. Es un simple juego y si hubiese hecho un juego de un hombre que debe sobrevivir en una jungla matando caníbales tendría que aclarar que el juego no es racista, que es simplemente un juego. En los años 80 la sociedad diferenciaba mejor lo que era ficción de lo que era realidad y no había que aclarar nada, pero en los últimos años hemos visto como todo lo que se salga del pensamiento oficial genera desasosiego en muchas personas. Si eres de esa clase de personas, puedes estar tranquilo. NOMWARS es un juego.

Controles del juego:

Para comenzar a jugar, pulsa espacio (o el disparo del joystick)

Q (arriba), A (abajo), O (izquierda), P (derecha)

disparo: espacio.

Bombas: mantener el disparo pulsado unos instantes.

El juego también acepta joystick, basta con pulsar el botón de fuego del joystick para comenzar, en lugar de la barra espaciadora.

3 Diseño general

En este apartado vamos a ver los sprites que usamos para cada elemento del juego.

Distribución de Sprites:

La distribución de sprites es importante para identificar en la rutina de colisión si te has chocado contra un muro del decorado, o un disparo enemigo. También es necesario para elegir un Sprite ID cuando se va a crear un disparo nuevo, por ejemplo. Definir este reparto es fundamental. Una cosa que se deduce de esta tabla es que Joe puede tener hasta 6 disparos “vivos” simultáneos en pantalla.

Sprite ID	Uso
31	Joe
30,29	Bombas (2)
28..23	Disparo joe (6)
22..15	Enemigos (8)
14..11	Disparos enemigos (4)
0..10	Elementos del mapa para scroll (11)

Distribución memoria versión BASIC:

El programa BASIC empieza con una instrucción MEMORY 18499 para que las variables se almacenen desde la 18499 hacia abajo. Recuerda que en BASIC el listado se almacena abajo y las variables arriba, creciendo hacia abajo. En este caso el programa BASIC ocupa exactamente 11kB, por lo que quedan libres 7500 bytes para variables BASIC (ya que 18499-11000 =7500)

Rango memoria	Uso	comentarios
0-18499	BASIC y variables BASIC	El listado basic ocupa 11 KB
18500-21130	Mapas de niveles y eventos	4 fases
21130-24000	Gráficos extra	No cabían todos en la zona habitual
24000-32200	Rutinas 8BP	8,2KB
32200-33600	Canciones	3 canciones(1.4KB)
33600-42040	Rutas y gráficos	8.4 KB
42040-42540	Mapa del mundo	Inicialmente vacia. Se rellena con cada fase

Distribución memoria versión ciclo compilado:

El programa BASIC ocupa menos memoria, concretamente ocupa 4700 bytes, por lo que con un MEMORY 10999 disponemos de 6300 bytes para variables. Ahora bien, tenemos un nuevo elemento: el binario resultante de compilar el ciclo, de modo que ambas versiones (BASIC puro o con ciclo compilado) ocupan parecido.

El ciclo compilado ocupa bastante: 7384 bytes. El motivo es que el wrapper de 8BP y el minibasic ocupan unos 3.2KB (por lo tanto, el ciclo consume unos 4.2kB sumando el propio ciclo de juego y las variables)

Rango memoria	Uso	Comentarios
0-10999	BASIC y variables BASIC	El listado basic ocupa 4700 bytes
11000-18384	Ciclo compilado	Ocupa 7384 bytes
18500-21130	Mapas de niveles y eventos	4 fases
21130-24000	Gráficos extra	No cabían todos en la zona habitual

24000-32200	Rutinas 8BP	8,2KB
32200-33600	Canciones	3 canciones (1.4KB)
33600-42040	Rutas y gráficos	8.4 KB
42040-42540	Mapa del mundo	Inicialmente vacía. Se rellena con cada fase

Todos los sprites han sido creados con speditV13, el cual permite definir una paleta y guardarla (en un fichero llamado pal.dat) para reusarla cada vez que vuelves a usar el spedit

Para mejorar la velocidad se usan varias recomendaciones que aparecen en el manual de 8BP, como son:

- Líneas de comentarios eliminadas (aparecen en el listado con una comilla simple pero eso es una abreviatura de REM, muchas de esas líneas están en el listado PC pero no se cargan en el fichero BASIC del Amstrad)
- Todas las variables son enteras (usando DEFINT A-Z)
- Uso de lógicas masivas: instrucciones que afectan a muchos sprites, ejecución alternada y periódica de ciertas tareas (rutina de colisión cada dos ciclos, rutina de disparo enemigo cada 8 ciclos, rutina de creación de enemigos en fin de fase cada 16 ciclos, lectura eficiente de teclado para evitar ejecutar instrucciones, arrays con cosas precalculadas, como las direcciones de memoria de los 32 sprites, etc.)
- Muchas líneas con comandos 8BP aparecen en el listado original dos veces. La primera con el comando 8BP y la segunda con el comando reemplazado por un **CALL**, que es lo mismo pero un poco más rápido. Eso permite programar de un modo cómodo y con un listado fácil de entender y simplemente al final añadí líneas en las que cambiaba cada comando por un CALL y así aceleré un poco el programa.

Ficheros:

Los ficheros de que consta el juego tienen una correspondencia directa con las tablas de rangos de memoria que acabas de ver.

Rango memoria	Uso	ficheros	
		"LOADER.BAS", en la versión de BASIC puro es "LOABAS.BAS"	Carga todos los ficheros del juego. El ultimo que carga es "NOMWARS.BAS", el cual entra en ejecución.
0-10999	BASIC y variables BASIC	"NOMWARS.BAS", en la versión de BASIC puro es "NOMBAS.BAS"	Es el juego
11000-18384	Ciclo compilado	CICLO.BIN	Este fichero no se carga en la versión BASIC pura
18500-21130	Mapas de niveles y eventos	GAME.BIN	Es un binario auxiliar, podría haberlo unido a 8BP.bin.
21130-24000	Gráficos extra		
24000-32200	Rutinas 8BP	8BP.BIN	
32200-33600	Canciones		
33600-42040	Rutas y gráficos		
42040-42540	Mapa del mundo		

4 Lógica del programa principal

El programa principal es muy sencillo. Además no hay tabla de puntuaciones, tan solo almacenamos el valor de la mayor puntuación (similar al juego “happy Monty”).

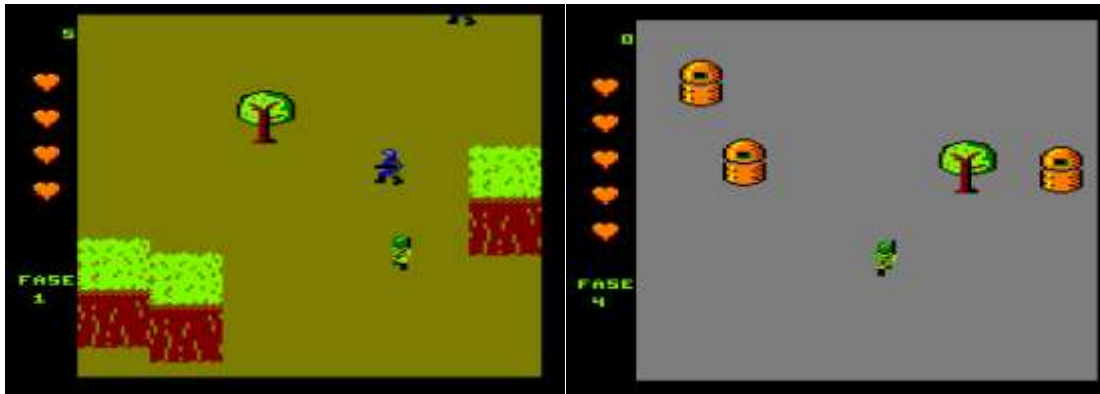
La lógica del juego (**no confundir con el ciclo de juego**) comienza en la línea 350. La lógica del programa principal es simplemente la que nos asigna las vidas iniciales y lanza el ciclo de juego, del cual retornamos cuando nos matan. En la línea 430 se lanza el ciclo de juego con el GOSUB 480.

```
350 '-- GAME LOGIC
360 |MUSIC:vidas=5:level=0:PAPER 0:CLS:GOSUB 2130
361 mid=0:|POKE,42548,mid: 'mitad de nivel.
370 puntos=0:newhs=0
371 poke 42547,0:'inmunidad
380 |SETUPSP,31,0,64+32+1:y=16*8:|LOCATESP,31,y,x:dir=0:|SETUPSP,31,7,1
381 |poke,42540,puntos:poke 42545,vidas:poke 42544,level
390 |MUSIC,0,0,1,6
391 BORDER 0:CLG 12:c$=STR$(vidas):c$="LIVES"+c$:|PRINTAT,0,80,34,@c$:
400 c$=STR$(level+1):c$="LEVEL"+c$:|PRINTAT,0,90,34,@c$:
401 c$=STR$(PUNTOS):c$="SCORE"+c$:|PRINTAT,0,100,34,@c$:
402 if newhs then c$="CONGRATULATIONS. NEW
HISCORE!":|PRINTAT,0,120,14,@c$:newhs=0
403 |SETUPSP,0,9,18:|PRINTSP,0,55,35:|SETUPSP,0,9,19:|PRINTSP,0,55,40
410 b$=INKEY$: IF b$<>" " THEN 410
411 if level=3 then ink 0,13
420 IF INKEY(fr) THEN 410 ELSE MODE 0:BORDER
0:|SETUPSP,31,0,64+32+1:y=16*8:x=40:|LOCATESP,31,y,x: FOR i=0 TO
30:|SETUPSP,i,0,0:NEXT
430 IF vidas THEN CLG 12:WINDOW 3,18,1,25:CLS:WINDOW
1,20,1,25:|SETLIMITS,8,72,0,200:GOSUB 480:'game cycle
440 FOR i=1 TO 500:NEXT
441 |PEEK,42540,@puntos:if puntos<=HISCORE then 443
442 |POKE,42542,PUNTOS:hiscore=puntos:newhs=1
443 level=peek(42544):vidas=peek(42545)
450 IF level=4 then gosub 5000:level=0
451 if vidas THEN 380
460 RUN
```

Hay algunas líneas interesantes si quieres “crackear” el programa:

1. En la línea 360 se asignan las vidas iniciales. Puedes ponerte 50 vidas, por ejemplo
2. En la línea 360 se establece el nivel inicial. Puedes cambiarlo y así empezar en otro nivel
3. En la línea 371 se establece la inmunidad, y si pones un 1, serás inmune

La línea 411 establece el color del suelo (tinta cero) a 13 (gris) en caso de que el nivel sea 3 (los niveles son cuatro y sus valores van de 0 a 3). El ultimo nivel presenta un aspecto algo diferente a los demás (imagen derecha).



Una vez que entramos en el ciclo de juego (comienza en la línea 480) realmente no comienza el ciclo de juego, sino que nos encontramos una serie de instrucciones para configurar el nivel al que entramos, la instrucción más importante es la que carga el mapa del nivel en la memoria de mapa del mundo con la instrucción "UMAP", abreviatura de "update map"

El UMAP lo tienes en la línea 540. Esta instrucción es muy importante y además esta invocada con unos parámetros cuyo significado es:

|UMAP, lbm, lem, 0, 9999, 0, 99

- **Lbm**: variable BASIC que indica la dirección de memoria del comienzo del mapa "level begin map"
- **Lem**: variable BASIC que indica la dirección de memoria del fin del mapa "level end map"
- Los parámetros **0,9999,0,99** indican que cogemos todos los elementos del mapa que nos encontremos en ese rango de coordenadas, y como son números grandes pues cogemos todos, es decir, se carga el mapa completo del nivel en la memoria de mapa del mundo, donde caben hasta 82 elementos.

```

480 '--LEVEL CONFIG
490 |SETLIMITS,0,80,0,200:|SETUPSP,0,9,16
510 FOR i=1 TO vidas:|PRINTSP,0,12+i*20,2:NEXT
521 c$="FASE":|PRINTAT,0,144,0,@c$
522 c$=str$(level+1):|PRINTAT,0,154,0,@c$
530 |SETLIMITS,8,72,0,200
531 evdir=0:|PEEK,42550+level*6,@evdir
532 lbm=0:|PEEK,42552+level*6,@lbm
533 lem=0:|PEEK,42554+level*6,@lem
534 Nbanda= level*8+16
540 |UMAP,lbm,lem,0,9999,0,99
541 m=0:if mid then |PEEK,(42574+level*4+2),@m:|PEEK,42574+level*4,@evdir
542 nextm=m
550 |MAP2SP,3:|MAP2SP,m,0:|PRINTSPALL,0,0,1,0:|COLSP,32,0,22:|AUTOALL,1
560 puntos$=STR$(puntos):|PRINTAT,0,8,4-tabu,@puntos$
579 evtipo=0
580 nextm=0:|PEEK,evdir,@nextm
580 |PEEK,evdir,@nextm
590 scroll =0:modo=0:c=3:' c=3 para que no coincidan los disp enemigos con
los eventos basados en m
600 banda=0:muertos=0:j=0:enesp=0:k=0
610 |MUSIC,0,0,2,6
620 GOSUB 2190:a=TIME

```

Fíjate como hemos asignado las variables lbm y lem. Por ejemplo la variable lbm se ha leído con un |PEEK sobre la dirección de memoria 42552+level*6

```
532 lbm=0:|PEEK,42552+level*6,@lbm
```

En el fichero images_extra_mygame.asm puedes ver al final del mismo una serie de etiquetas en ensamblador:

```
org 42550
dw _level_0_begin_events
dw _level_0_begin_map
dw _level_0_end_map

dw _level_1_begin_events
dw _level_1_begin_map
dw _level_1_end_map

dw _level_2_begin_events
dw _level_2_begin_map
dw _level_2_end_map

dw _level_3_begin_events
dw _level_3_begin_map
dw _level_3_end_map
```

Esas etiquetas se usan en otro punto del fichero, por ejemplo la etiqueta _level0_begin_map aparece al principio del fichero con los elementos del mapa del nivel cero (árboles, muros etc)

```
_LEVEL_0_BEGIN_MAP
;screen1
dw 170,20, ARBOL
dw 80,60, ARBOL

;screen2
dw 1*200+207,8,MUROSIMPLE
dw 1*200+207,24,MUROSIMPLE
dw 1*200+207,40,MUROSIMPLE
dw 1*200+207,56,MUROSIMPLE
```

Esto permite que al ir programando el nivel 0, el 1, el 2 etc. las direcciones de memoria de la palabra **_LEVEL_0_BEGIN_MAP** se actualiza sola, al ensamblar y siempre en 42552+level*6 encontramos justo el valor de esa dirección de memoria.

Programar así los mapas me permite ir construyendo el mapa, o tocar un mapa anterior y despreocuparme de si han cambiado las direcciones de memoria donde comienza un determinado mapa por haber añadido un elemento al mapa del nivel anterior. Cuando se ensambla queda en la dirección 42550 una lista con las direcciones de mapas de los niveles y listas de eventos de cada nivel (luego veremos qué es esto de los eventos).

Hay un mecanismo que permite empezar a mitad de nivel cuando has llegado suficientemente lejos, para que no de tanta “rabia” volver a empezar desde el principio.

```
541 m=0:|PEEK,42548,@mid;if mid then
|PEEK,(42574+level*4+2),@m:|PEEK,42574+level*4,@evdir
```

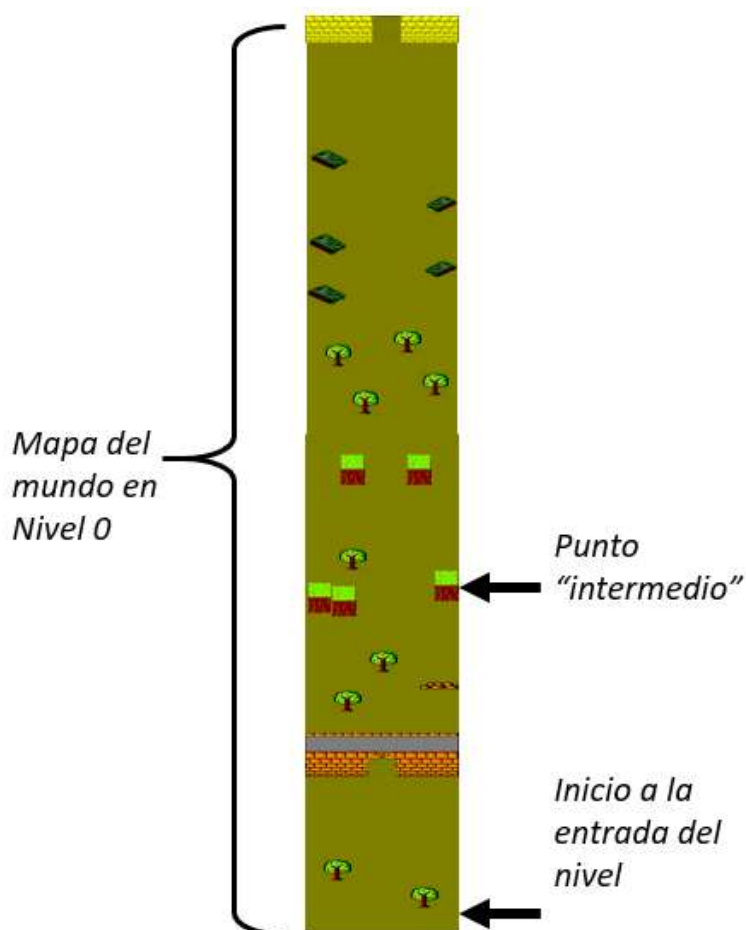
El primer PEEK lee la variable mid, que la tenemos almacenada en la dirección 42548. Podríamos haber usado una simple variable basic (sin almacenarla en una dirección) pero así el código del programa en C es compatible con estas líneas, ya que desde el programa C compilado no se pueden leer las variables basic pero sí que se pueden leer datos en direcciones de memoria.

Si “mid” vale 1, entonces es que debemos empezar a mitad del nivel y en ese caso se lee la posición donde debemos empezar. Esa posición la almacenamos en la variable m. Cada nivel tiene su propia “posición intermedia”, no es justo a la mitad del nivel.

Después, en la línea 550 hacemos un MAP2SP para comenzar el scroll en la posición “m” del mapa del mundo, que habrá tomado un valor intermedio en lugar de en la posición m=0

550	MAP2SP, 3:	MAP2SP, m, 0:	PRINTSPALL, 0, 0, 1, 0:	COLSP, 32, 0, 22:	AUTOALL, 1
-----	------------	---------------	-------------------------	-------------------	------------

Una aclaración IMPORTANTE: como ves el mapa del mundo es el mismo tanto si comienzas la fase a la mitad como si comienzas desde el principio. El mapa del mundo se carga con UMAP en la línea 540 y es en la línea 550 donde ubicamos la “ventana” que vemos en pantalla en la posición cero o en una posición “intermedia”.



5 Lógica del ciclo de juego

Para abreviar, al personaje le llamaremos Joe, que es el nombre del personaje del juego comando.

A modo de resumen, el ciclo de juego consta de las siguientes tareas:

1. creación periódica de disparo enemigo
2. creación periódica de soldado enemigo si estamos a final de fase
3. ejecución de eventos en función de la posición de Joe en el mapa
4. control de teclado y scroll
5. cómputo de la secuencia de animación de Joe
6. animación del personaje y posicionamiento de Joe
7. control del disparo de Joe
8. creación del disparo de Joe
9. impresión de sprites
10. rutina de colisión de sprites

Como ves son muchas tareas. Algunas de ellas no se ejecutan en todos los ciclos, como la creación de disparo enemigo o la creación de enemigos nuevos a final de fase. Incluso la detección de colisión no se hace en cada ciclo, porque con hacerla cada dos ciclos proporciona un buen resultado.

5.1 Ciclo de juego

La lógica del ciclo de juego comienza en la línea 630 y lo primero que hace es incrementar el 1 el número de ciclos.

Justo después, en la línea 660 nos encontramos con una instrucción propia de lógicas masivas: ejecución periódica basada en una operación binaria (matemáticas modulares) que nos permite ejecutar un bloque de instrucciones una de cada 8 veces (cuando C and 7 es cero). Ese bloque de instrucciones sirve para:

- crear el disparo enemigo. (la creación de disparo se invoca con GOSUB 1990)
- También sirve cuando llegamos al final de fase para invocar la creación de un nuevo soldado cada 16 ciclos (la línea 690)

```
630 '-- GAME CYCLE
640 c=c+1
660 IF (c AND 7) THEN 720
680 GOSUB 1990:'creacion de disp enemigo
690 IF banda THEN IF c AND 15 THEN 740 ELSE GOSUB 1880:GOTO 740
'700 '-- control de eventos basados en m (m es la posicion del mapa)
3*8=24. la resolucion es salto m = 24 (cada 8 ciclos)
720 IF m=nextm THEN GOSUB 1500 :'ejecucion del evento
```

A continuación, tenemos la **creación de “eventos”** que son instrucciones a ejecutar cuando la variable m, que es la que dice en que coordenada vertical del mapa estamos, se encuentra en una posición concreta (nextm). Un evento puede ser crear un enemigo estático o un enemigo que se mueve. También hay mas tipos de eventos que luego veremos. Una vez procesado el evento en la rutina 1500, se actualiza la posición del siguiente evento (la variable nextm)

Después tenemos los controles del personaje a partir de la línea 740, que además incluyen la invocación a la función de scroll MAP2SP. Si hay avance, además se invoca a MOVERALL, para que los enemigos sumen su velocidad al movimiento hacia debajo de todo el escenario debido al avance

A continuación, tenemos el **cómputo de la secuencia de animación**, a partir de la línea 950. Esto requiere cierto calculo porque el personaje no tiene una secuencia de animación directamente relacionada con la tecla o teclas que pulsas. Si vas hacia abajo y pulsas la tecla para ir hacia arriba, pasaras por diferentes secuencias de animación en cada fotograma hasta llegar a tener la secuencia de subir.

Tras la elección de la secuencia, se anima y coloca el personaje en las nuevas coordenadas que se han decidido tras el control de teclado. Son las líneas 104 y 1050

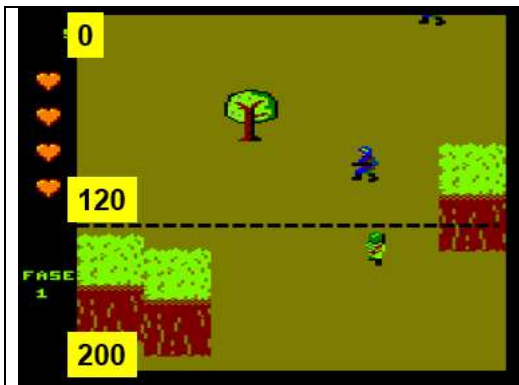
1040	ANIMA,31
1050	LOCATESP,31,y,x

5.2 Control del personaje

El movimiento vertical hacia arriba es el más complejo y vamos a verlo en detalle. Una cosa que puedes apreciar es que hay una variable "t" que va a almacenar un valor en función de las teclas que pulsemos. En el caso UP, esta variable toma el valor 1. Usaremos esta variable después para determinar la dirección de movimiento del personaje y su secuencia de animación.

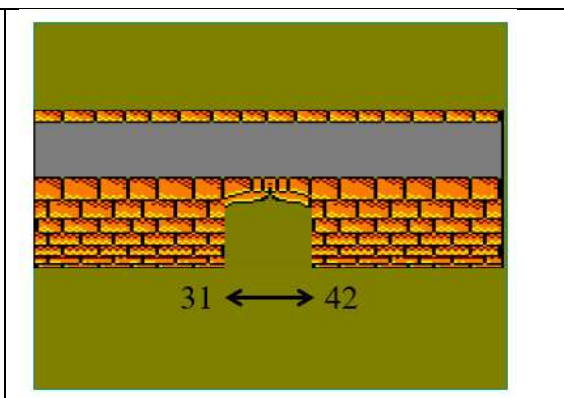
740	IF INKEY(up) THEN 840 ELSE IF y THEN 750 ELSE 880
750	t=1:IF y>120 THEN 820
'760	'con scroll si y =120
770	IF scroll THEN 820
'780	'con scroll
790	IF modo THEN IF x>31 THEN IF x<42 THEN
	m=m+3: MAP2SP,m,0: MOVERALL,3,0:GOTO 880
800	MOVER,31,-3,0: COLSP,31:IF col=32 THEN
	m=m+3: Map2sp,m,0: moverall,3,0:GOTO 880 ELSE mover,31,3,0:GOTO 880
'810	'sin scroll
820	IF modo THEN IF x>31 THEN IF x<42 THEN y=y-2:GOTO 880
830	MOVER,31,-2,0: COLSP,31:IF col>22 THEN y=y-2:GOTO 880 ELSE
	MOVER,31,2,0:GOTO 880

La **variable scroll** significa lo contrario de lo que parece. Si scroll=1 entonces es que hemos llegado a fin de fase y no hay que hacer scroll aunque nos movamos hacia arriba

	<p>El soldado debe forzar scroll solo si su posición es $y > 120$, de ese modo si el personaje retrocede no hay scroll, hasta que vuelva a $y = 120$</p> <p>En la imagen se muestra la altura a la que se encuentra la línea imaginaria $y = 120$ que hace que el scroll vertical avance cuando intentamos superarla, manteniendo a Joe en la misma posición vertical.</p>
---	--

En las líneas 790 y 800 se ejecuta el mecanismo de scroll avanzando 3 líneas cada vez ($m = m + 3$)

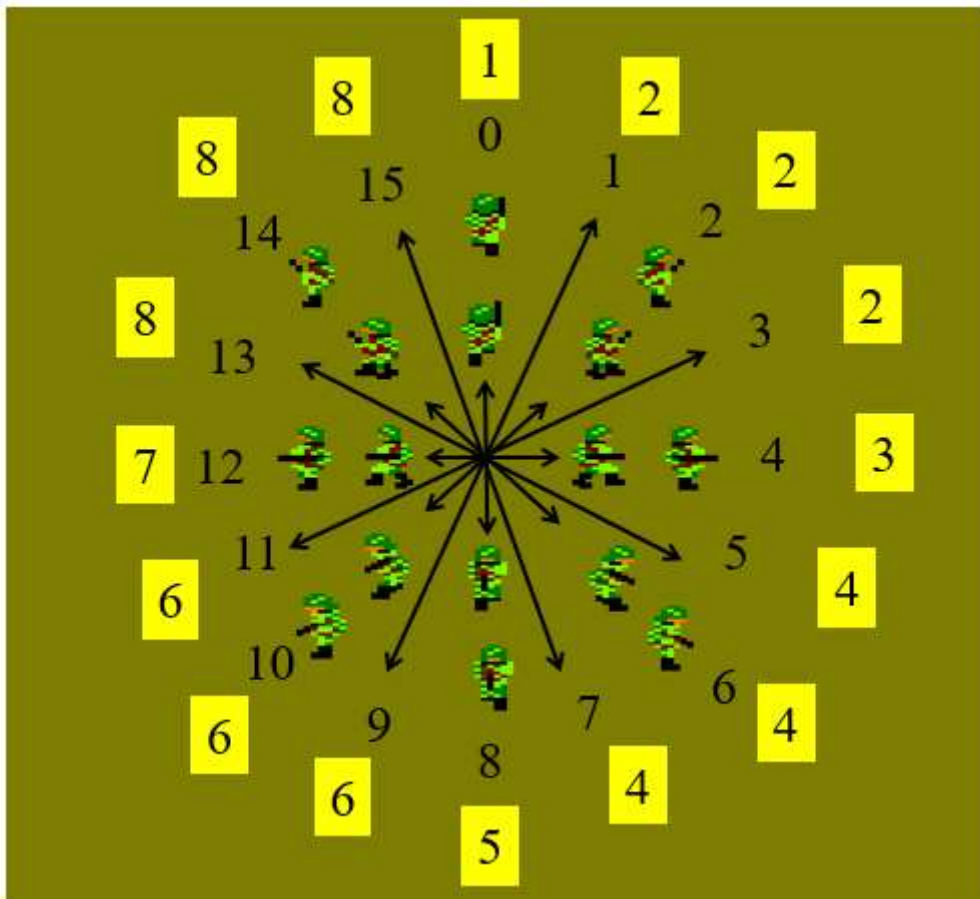
El control de coordenadas en caso de encontrar $\text{modo} = 1$ es para el puente. Si estamos en mitad del puente solo podemos avanzar si nos encontramos en $31 < x < 42$.

<p>En esta imagen aparece el puente con las coordenadas del agujero y la altura a la que se encuentra la línea imaginaria $y = 120$ que hace que el scroll vertical avance cuando intentamos superarla.</p>	
--	---

5.3 Cómputo de la secuencia de animación

La secuencia de animación a asignar depende de la secuencia que actualmente tiene Joe y de la tecla que hayas pulsado. Si por ejemplo estas caminando hacia la izquierda y pulsas derecha, la secuencia final a asignar será la secuencia derecha, pero pasarás por 3 secuencias de animación intermedias

Hay 16 direcciones de movimiento contempladas, pero tan solo 8 secuencias de animación (para ahorrar memoria). En este diagrama puedes ver las 16 direcciones y **en recuadros amarillos las distintas secuencias de animación** creadas (desde 1 hasta 8). Joe se puede mover en las 16 direcciones, pero solo le he pintado en las 8 diferentes que hay, para que se vea más claro. Por ejemplo, las direcciones 1,2,3 comparten la secuencia número 2.



La correspondencia entre direcciones de movimiento y secuencias de animación se encuentra en el array seq

```
250 DIM seq(16):seq(0)=1:seq(1)=2:seq(2)=2:seq(3)=2:seq(4)=3:
seq(5)=4:seq(6)=4:seq(7)=4:seq(8)=5
260 seq(9)=6:seq(10)=6:seq(11)=6:seq(12)=7:seq(13)=8:seq(14)=8:
seq(15)=8
```

Dentro del ciclo de juego se calcula la dirección de movimiento y después (en rojo) se asigna la secuencia de animación correspondiente a la dirección que se ha determinado asignar

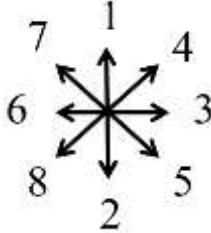
```
'940 '-- DIR computation
950 IF t THEN 960 ELSE 1070
960 dif=dir-po(t):t=0
970 IF dif THEN 980 ELSE 1040
980 IF dif<0 THEN IF dif<-8 THEN dir=dir-1:GOTO 1010 ELSE
dir=dir+1:GOTO 1020
990 IF dif>8 THEN dir=1+dir AND 15:GOTO 1020
1000 dir=dir-1
1010 dir=dir AND 15
1020 |SETUPSP,31,7,seq(dir)
```

Para calcular la dirección de movimiento primeramente se inspecciona la variable t. Si has pulsado alguna tecla, t será diferente de cero e irás a la 960, de lo contrario la secuencia de

animación no cambia, y te saltas ese bloque de instrucciones yendo directamente a la línea 1070. La variable t contiene un código numérico en función de la combinación de teclas que has pulsado

La dirección actual se almacena en "dir" y la dirección objetivo a la que te diriges en función de las teclas que has pulsado es $po(t)$. Si hay diferencia entre ambas significa que debes cambiar de dirección, poco a poco, pasando por cada una de las direcciones intermedias en cada fotograma.

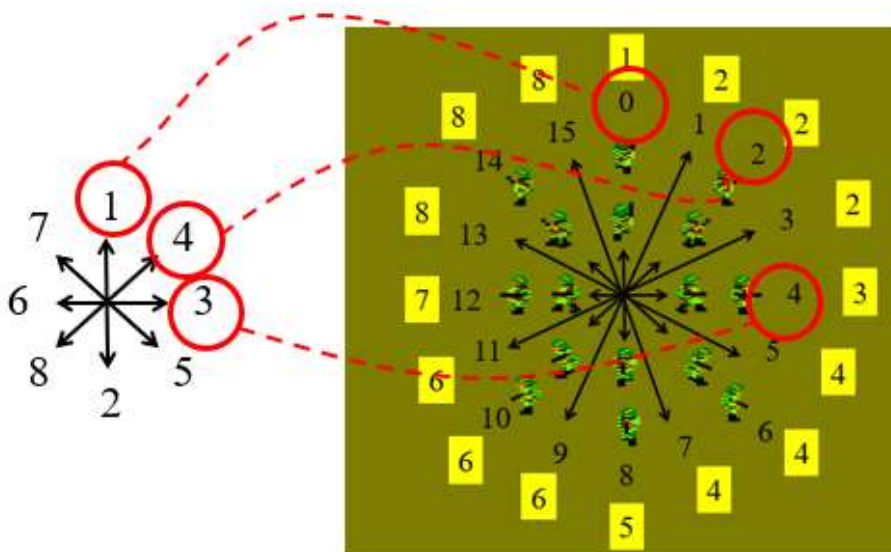
Nos queda explicar que es $po(t)$. Para ello debemos aclarar cómo se asigna la variable t en función de las teclas que pulsas

<ul style="list-style-type: none"> • No pulsas: $t=0$ • UP: $t=1$ • Down : $t=2$ • Right: $t=t+3$ • Left: $t=t+6$ 	<p>Con esas asignaciones a la variable "t", nos quedan estos valores:</p> 
---	--

Como vemos, tenemos un valor de variable t diferente para cada dirección, que es justo lo que necesitamos para poder tener un array de direcciones en función de la combinación de teclas pulsadas. Ese array es $po(t)$

240 $po(1)=0:po(4)=2:po(3)=4:po(5)=6:po(2)=8:po(8)=10:po(6)=12:po(7)=14$

Si ahora cotejas este array con las 16 direcciones de Joe, verás que cada una de las 8 posibilidades se corresponde con 8 posibles direcciones de movimiento de Joe (Joe tiene además otras 8 direcciones de movimiento intermedias). Y como puedes ver para $t=1$ $po(1)$ vale cero, justo la dirección objetivo que se persigue al pulsar esa tecla. Y si pulsas Q+P entonces $t=4$ y la dirección es 2 pues $po(4)=2$. Y así con todas.



Volviendo al mecanismo de cómputo de la secuencia, lo que hacemos es pasar por las direcciones intermedias entre la que tiene Joe y la dirección final que se obtiene al pulsar esas teclas. De esta manera se usan las 16 direcciones, pero hay 8 que son las mas importantes porque son las direcciones “finales” a las que llegamos al pulsar una combinación de teclas.

```
'940 '-- DIR computation
950 IF t THEN 960 ELSE 1070
960 dif=dir-po(t):t=0
970 IF dif THEN 980 ELSE 1040
980 IF dif<0 THEN IF dif<-8 THEN dir=dir-1:GOTO 1010 ELSE
dir=dir+1:GOTO 1020
990 IF dif>8 THEN dir=1+dir AND 15:GOTO 1020
1000 dir=dir-1
1010 dir=dir AND 15
1020 |SETUPSP,31,7,seq(dir)
```

Como ves se calcula la diferencia entre ambas direcciones (variable dif) y en función de dif se cambia la dirección actual para acercarnos a la dirección que indican las teclas, pero no de golpe, sino que dir cambia de uno en uno, pasando por las direcciones intermedias. Si el camino mas corto es aumentar dir, entonces dir se incrementa una unidad en este fotograma, otra unidad en el siguiente, así hasta llegar. De ese modo vamos pasando por las direcciones intermedias siguiendo la dirección de la aguja de un reloj. En caso de que el camino mas corto sea el inverso, la dirección se irá decrementando. No te preocupes si no entiendes perfectamente esos tres IF que permiten calcular la siguiente dirección. Si has entendido la idea, en realidad lo has entendido todo.

5.4 Disparo del personaje

El disparo del personaje comienza en la línea 1070

```
'1060 '-- FIRE CONTROL
1070 IF INKEY(fr) THEN IF pres THEN pres=0:bomb=0:GOTO 1160 ELSE 1160
1080 IF pres THEN bomb=bomb+1:IF bomb<8 THEN 1160 ELSE 1430
1090 dis= 23 + dis MOD 6
'1100 '-- FIRE CREACION
1110 |SETUPSP,dis,15,dir
1120 |LOCATESP,dis,y+4,x+2
1130 |SETUPSP,dis,0,233
1140 pres=1
```

La primera línea IF ejecuta el THEN solo si la tecla de disparo NO ha sido pulsada. Entonces asigna un cero a la variable “pres” que permite saber si la tecla de disparo estaba ya pulsada anteriormente. Ello te obliga a tener que despulsar y volver a pulsar si quieres disparar. Si la dejas pulsada durante 8 ciclos entonces saltarás a la línea 1430, que es donde se encuentra la rutina de lanzamiento de bomba

El disparo se crea con un Sprite desde el 23 hasta el 28 mediante la línea

```
1090 dis= 23 + dis MOD 6
```

Y tras crear el Sprite se le asigna como ruta, la dirección que tenga Joe. **Esto es un truco interesante de lógica masiva.** He creado rutas para los disparos con los identificadores correspondientes a las direcciones de movimiento de Joe, de modo que no me hace falta

consultar una tabla de correspondencia entre la dirección de joe y la ruta que debe de usar el disparo, sino que la traducción es directa **y por lo tanto más rápida!**.

1110 |SETUPSP,dis,15,dir

Dicho de otro modo, las primeras 16 rutas son las correspondientes al disparo de Joe y se corresponden exactamente con cada una de las 16 direcciones de movimiento que tiene Joe

Vamos a analizar la rutina de bomba, a la que saltas cuando la variable bomb llega a 8. Esto ocurre si durante 8 fotogramas dejas pulsado el disparo. En ese caso en lugar de disparar, la variable bomb va aumentando y al llegar a 8 se lanza una bomba. Puede haber hasta dos bombas a la vez en pantalla, correspondientes a los sprites 29 y 30

```
'1420 '-- RUTINA BOMBA
1430 j=29 + j MOD 2: IF PEEK(dirsp(j)) THEN 1160
1440 |SETUPSP,j,15,16
1450 |LOCATESP,j,y,x+1
1460 |SETUPSP,j,0,217:'no colisionador
1470 bomb=0
1480 GOTO 1160
```

La elección del Sprite se hace con la línea marcada en rojo. Si intentas lanzar 3 bombas demasiado rápido, la tercera bomba no se lanzará porque la sentencia **IF PEEK(dirsp(j))** detectará que el Sprite escogido aun esta activo.

La bomba se crea con la ruta 16. Y nada mas lanzarla se asigna un cero a “bomb”, de modo que al menos deben pasar 8 ciclos entre dos lanzamientos de bombas

5.5 Mapa y eventos

Ahora que sabemos como se controla el personaje vamos a pasar a describir lo que hemos llamado “eventos” del juego, que son rutinas que se ejecutan condicionalmente, ante ciertos valores de “m”, que es la posición que tiene Joe en el mapa.

Como Joe se mueve de 3 en 3 (es decir que cuando caminas hacia arriba se le suma un 3 a m) entonces los eventos están ubicados en múltiplos de 3. Y como además, solo se evalua la condición IF m=nextm cada 8 ciclos, tenemos que m debe ser multiplo de $3 \times 8 = 24$. Como vamos a ver todos los eventos están ubicados en posiciones múltiplos de 24. Estos eventos son de 7 tipos:

Tipo	descripcion
1	Principio de puente
2	Fin de puente
3	Cambio de tintas. Se usa para cambiar el color de la tinta 1 que se usa en el muro de final de fase.
4	Fin de scroll. Se usa para delimitar el fin del mapa
5	Evento banda: creación de enemigo cuando llegas al final de fase. Es un evento que se ejecuta cada 16 ciclos al llegar a un final de fase.
6	Enemigo fijo, sin movimiento, pero con capacidad de disparo
7	Enemigo móvil con capacidad de disparo (soldados enemigos)

La línea del ciclo de juego que “dispara” los eventos es la 720

```
720 IF m=nextm THEN GOSUB 1500 :'ejecucion del evento
```

Como vemos, no hacen falta 40 sentencias IF para ubicar 40 eventos, basta con que al final de la ejecución de cada evento cambiemos el valor de “nextm”, que es la posición del mapa donde se encuentra el siguiente evento.

Los eventos y sus tipos deben ser almacenados en una tabla, junto con algunos parámetros de cada evento, como son las coordenadas del nuevo enemigo a crear etc.

La lista de eventos de cada fase se encuentra en el fichero images_extra_mygame.asm. Vamos a ver los eventos de la primera fase. Fíjate que todos los eventos tienen como primer parámetro el valor de “m” en el que deben aparecer y dicho valor de m es siempre múltiplo de 24 (tal y como hemos explicado antes). Tras ejecutar un evento se almacena en “nextm”, el primer valor del siguiente evento. Como son words (no son bytes), los leemos desde la rutina de eventos con la función |PPEEK de 8BP

```
org 18500

;EVENTOS
;-----
; valores de eventos m1,m2,m3,m4 etc todos multiplos de 24
; luego esta el tipo de evento y luego los parametros del evento
; tipo 1 es puente
; tipo 2 es fin puente
; tipo 7 es enemigo m,5,seq,y,x,ruta
; NIVEL-1

_LEVEL_0_BEGIN_EVENTS
dw 24,7,32,100,72,17,0 ; soldado
dw 48,7,32,100,4,21,0 ; soldado
dw 72,7, 32,-10,16,19,0; soldado
dw 96,7,32,-10,50,22,0;soldado
dw 120,6,NAZIMETRALLETA,3,0,60; soldado fijo
dw 240,7, 32,46,72,17,64; soldado en puente1
dw 264,1,271; puente
dw 288,7, 32,90,2,18,64; soldado en puente2
dw 312,6,NAZIMETRALLETA,3,-10,64;
dw 336,7,32,-80,20,22,0; soldado
dw 360,7,32,-10,60,23,0;soldado
dw 384,2; fin puente. siempre es el evento puente +120
dw 408,7,32,0,72,17,0;soldado
dw 432,7,32,8,-6,24,0;soldado
dw 576,7,32,-20,50,23,0; soldado
_LEVEL_0_MID_EVENTS; m=200*3=600
dw 600,7,32,-40,72,20,0; soldado
dw 720,7,32, -100,40,23,0; soldado
dw 744,7,32, -100,50,23,0; soldado
dw 792,7,32,-10, 0,21,0; soldado
dw 816,7,32,-100, 72,20,0; soldado
dw 888,7,32,-40,72,25,0; soldado
dw 912,7,32,-100,0,24,0; soldado
```

```

dw 1080,7,32 ,-20,0,21,0; soldado
dw 1104,7,32 ,-20,72,20,0; soldado
dw 1128,7,32 ,-100,0,21,0; soldado
dw 1152,7,32 ,-60,72,25,0; soldado
dw 1248,7,32 ,-30,72,20,0; soldado
dw 1536,3,24; tinta
dw 1608,4 ; finscroll
dw 1608,5; banda
dw 5000,9;fin
_LEVEL_0_END_EVENTS
_LEVEL_0_BEGIN_MAP
;screen1
dw 170,20, ARBOL
dw 80,60, ARBOL

;screen2
dw 1*200+207,8,MUROSIMPLE
dw 1*200+207,24,MUROSIMPLE
dw 1*200+207,40,MUROSIMPLE
dw 1*200+207,56,MUROSIMPLE
...

```

Lo que hemos hecho es crear los eventos de la primera fase a partir de la dirección 18500. Después de la lista de eventos tenemos el mapa de la primera fase, que empieza con dos árboles. He pintado de color azul las líneas del mapa para diferenciarlas de las de los eventos.

A continuación, lo mismo, pero de los niveles 2, 3 y 4

Si nos vamos a la rutina de procesamiento de eventos, vemos como se usa la sentencia “ON GOTO” para saltar a distintas rutinas en función del tipo de evento. Vamos a ver el evento que mas veces aparece, el de creación de enemigo móvil

```

1490 '-- EVENT ROUTINE
1500 |PEEK,evdir+2,@evtipo: ON evtipo GOTO
1530,1570,1600,1620,1880,1790,1650

...

'1640 '-- tipo 7 evento enemigo movil nazi
1650 enesq=0:|PEEK,evdir+4,@enesq:'macroseq
1660 eney=0:enex=0:|PEEK,evdir+6,@eney:|PEEK,evdir+8,@enex:
&6931,evdir+8,@enex:
1670 ener=0:|PEEK,evdir+10,@ener:'ruta
1680 tr=0:|PEEK,evdir+12,@tr:'transp
1690 enesp=15+ enesp MOD 8 :'8 ene del 15 al 22
1700 |SETUPSP,enesp,5,1:|SETUPSP,enesp,7,enesq:
1710 |LOCATESP,enesp,eney,enex
1720 |SETUPSP,enesp,0,159+tr
1730 |SETUPSP,enesp,15,ener
1740 evdir=evdir+14
'1750 '-- fin evento y prepara siguiente
1760 |PEEK,evdir,@nextm
1770 RETURN

```

Lo que hace esta rutina es leer los parámetros del evento con la instrucción |PEEK de 8BP pues son datos de 16 bits. Los datos del evento vienen en este orden:

m, 7, secuencia, y, x, ruta, transparencia

por ejemplo:

dw 24,7,32,100,72,17,0

Este es un soldado que aparece en m=24, de tipo 7 (es decir, es un soldado), con secuencia de animación 32, en coordenadas 100,72, con ruta 17 y con transparencia 0 (no transparente).

Lo mas importante es el mecanismo usado para crear los eventos: simplemente leemos de una tabla y vamos creando un Sprite enemigo con esos datos. Y el ultimo dato que leemos es el valor de m en el que aparecerá el siguiente evento (valor que cargamos en "nextm")

```
'1750 '-- fin evento y prepara siguiente
1760 |PEEK,evdir,@nextm
1770 RETURN
```

5.6 Mecanismo del puente

El mecanismo del puente se puede considerar parte del control del personaje. Consiste en dos eventos, uno de comienzo de puente y otro de fin de puente. Es un evento que afecta al personaje en la forma en cómo se imprime. En concreto son los eventos 1 y 2

```
'1520 '-- tipo 1 evento puente
1530 modo=1:|SETUPSP,31,0,64:
1540 pini=0:|PEEK,evdir+4,@pini:pfin1=pini+44:pfin2=pfin1+80:p2=-pfin1+112:p1=-
pini+120:
1550 evdir=evdir+6:GOTO 1760
'1560 '-- tipo 2 evento fin de puente
1570 IF y=120 THEN |SETLIMITS,8,72,0,200:|SETUPSP,31,0,64+32+1:modo=0: evdir=evdir
+4:GOTO 1760
1580 RETURN
```

Básicamente cuando nos encontramos el primer evento (el 1), asignamos un 1 a la variable "modo" (ojo, no tiene nada que ver con el modo de video, simplemente le he dado ese nombre) y le quitamos el flag de impresión a Joe, de modo que ya no se va a imprimir con PRINTSPALL, pero sigue siendo un Sprite colisionador. Durante el tiempo que tenga vigencia el modo 1, la impresión de Joe se hace después del resto de sprites, usando PRINTSP y con un SETLIMITS para que la parte superior de Joe intersecte con el puente y solo imprimamos una parte gracias al clipping. Fijate que cuando modo =1 usamos dos SETLIMITS distintos: uno para cuando estas entrando en el puente y otro para cuando estas saliendo.

```
'1150 '-- MODOS 0=normal, 1=puente
1160 |AUTOALL:IF modo THEN 1200
1160 call &71d2:IF modo THEN 1200
'1170 '-- modo 0 normal
1180 |PRINTSPALL:GOTO 1230
'1190 '-- modo 1 puente
```

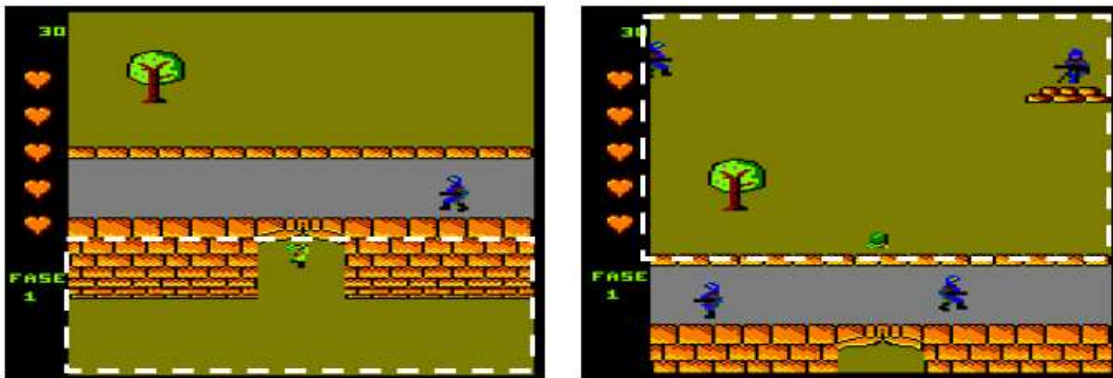


```

1200 IF m>pfin1 THEN |SETLIMITS,8,72,0,m+p2 ELSE
|SETLIMITS,8,72,m+p1,200
1210 |PRINTSP,31:|SETLIMITS,8,72,0,200:|PRINTSPALL

```

No te preocupe no entender exactamente como se calculan los limites de SETLIMITS. Lo que importa es el concepto, el mecanismo mediante el que se ha implementado este “truco”. En estas dos imágenes se ilustra: tras imprimir todos los sprites, se hace un SETLIMITS para restringir el área de impresión de la pantalla y se imprime el personaje. El mecanismo de “clipping” de la impresión de sprites hace que Joe desaparezca justo al entrar en el puente. Justo después de imprimir a Joe se deja el SETLIMITS como estaba para poder usar todo el área de juego al imprimir sprites en el siguiente ciclo. Y vuelta a empezar, hasta que salimos de ese modo, es decir, hasta que lleguemos al evento 2 y pongamos modo=1



6 Rutas

Vamos a revisar las rutas de sprites del videojuego con una tabla, donde encontraras todas las rutas de enemigos y disparos. Como ya hemos visto, las rutas de los disparos de joe ocupan las primeras 16 posiciones

En total el juego tiene 48 rutas definidas. Se encuentran en el fichero routes_mygame.asm

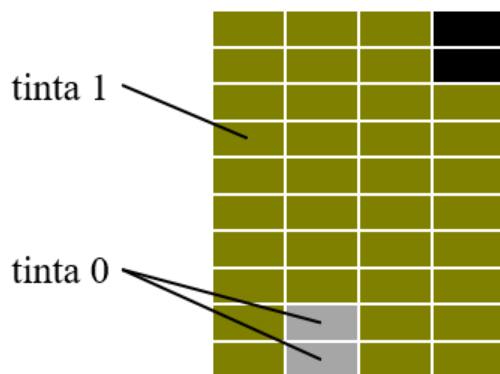
rutas	descripción
0-15	Direcciones de disparo de joe
16	Bomba, solo requiere una ruta
17-31	Un total de 15 Rutas de diferentes trayectorias de soldados enemigos
32-47	En total son 16 rutas de disparo enemigo, mas lento y corto que el disparo de joe
48	Logo de 8BP botando en la pantalla de inicio del juego

Vamos a ver unas rutas interesantes, la del disparo de Joe. Son interesantes por como están creadas las imágenes. Son imágenes que permiten borrar la posición anterior mientras avanzan bastante

Vamos a ver la ruta 1, de disparo hacia arriba-derecha con unos 60 grados de ángulo

```
ROUTE1; ruta de disparo 1 , 60 grados
db 253
dw DISPARO_1
db 9,-8,1
db 254,9,1
db 251,0,0
db 255,64+4+1,0; quito auto y asi no se mueve
db 1,0,0;return
db 0
```

Lo más interesante es ver cómo es la imagen DISPARO_1, para comprender el mecanismo que permite que sea un disparo tan rápido.

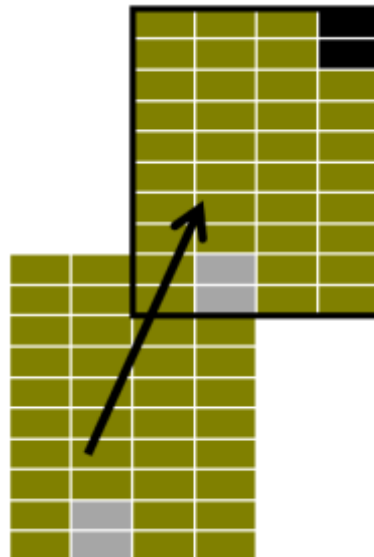


Gracias a esta imagen, es posible hacer un disparo muy rápido y con una dirección diagonal con el ángulo deseado. Cada ruta tiene un Sprite de disparo diferente. Esta imagen se llama DISPARO_1 porque es el de la ruta 1, pero hay 16 imágenes de disparo. Una por cada ruta

Como puedes observar es un Sprite transparente que usa la tinta 1 para respetar lo que hubiese ya impreso y la tinta cero para reestablecer el fondo. En esos dos pixels con tinta cero estaba el disparo (negro, situado en la esquina) del anterior fotograma, ya que la ruta mueve 8 líneas hacia arriba y 1 byte (dos píxeles) hacia la derecha en cada paso.

En total la ruta se compone de 9 pasos idénticos hasta asignar la secuencia de animación de disparo explotando

Al moverse un paso:



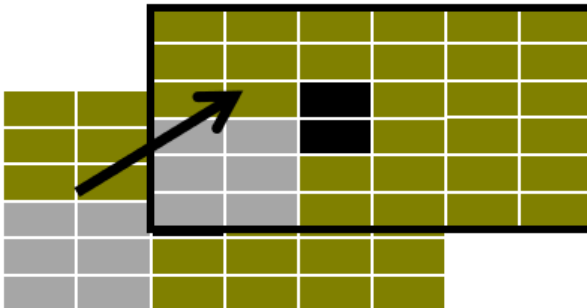
Las rutas de disparos enemigos son más lentas requieren sprites más pequeños. Son las rutas de la 32 a la 47. Vamos a ver una (la 33)

```
ROUTE33; ruta de disparo 1 , 60 grados
db 253
dw DISPARONAZI_UPR
db 20,-2,1
db 254,9,1
db 251,0,0
db 255,64+4+1,0; quito auto y asi no se mueve
db 1,0,0;return
db 0
```

Esta ruta da 20 pasos pequeños subiendo 2 líneas hacia arriba y un byte(2 pixels) a la derecha. Al igual que con el disparo de Joe, la imagen asociada lógicamente está diseñada para borrarse a sí misma al desplazarse:



Al moverse un paso en la ruta que estamos analizando:



Esta imagen nos sirve para varias direcciones de movimiento, en concreto la usaremos en 3 diagonales distintas (60,45 y 30 grados), por lo que de ese modo ahorramos memoria

7 ¡A Jugar y a programar!

Hemos visto como esta estructurado el videojuego NOMWARS y cuales son sus mecanismos de funcionamiento. El control del personaje y el mecanismo de eventos son las cosas mas interesantes y sencillas de programar que caracterizan este videojuego.

Es el momento de comenzar tu propio juego. Un consejo: No olvides dedicarles tiempo a los gráficos, pues un juego con dibujos atractivos es mucho mas agradable y hasta divertido.

¡A jugar y a programar!