

8 BITS DE PODER

Como programar “Frogger Eterno”



V36

Jose Javier García Aranda

INDICE

1	HISTORIA DE FROGGER	5
2	VERSIÓN DE FROGGER PROGRAMADA CON 8BP	7
3	BREVE INTRODUCCIÓN A 8BP	9
4	DISEÑO DE SPRITES.....	17
5	ESTRUCTURA DE UN VIDEOJUEGO	23
6	DISEÑO GENERAL	25
6.1	CAPA DE MARCADORES	26
6.2	CAPA DE LAYOUT (TILE MAP)	27
6.3	CAPA DE SPRITES	29
7	PRESENTACIÓN	33
8	LÓGICA DEL PROGRAMA PRINCIPAL.....	37
9	CONTROL DE LA RANA	39
10	RUTINA DE COLISIÓN.....	43
11	LÓGICA DEL CICLO DE JUEGO	45
12	RUTAS.....	47
12.1	TIPOLOGIA DE POSIBLES RUTAS.....	47
12.2	RUTAS DE VEHÍCULOS	47
12.3	RUTAS DE TORTUGAS.....	48
12.4	RUTAS DE TRONCOS.....	49
13	A JUGAR Y A PROGRAMAR!!	51

1 Historia de frogger

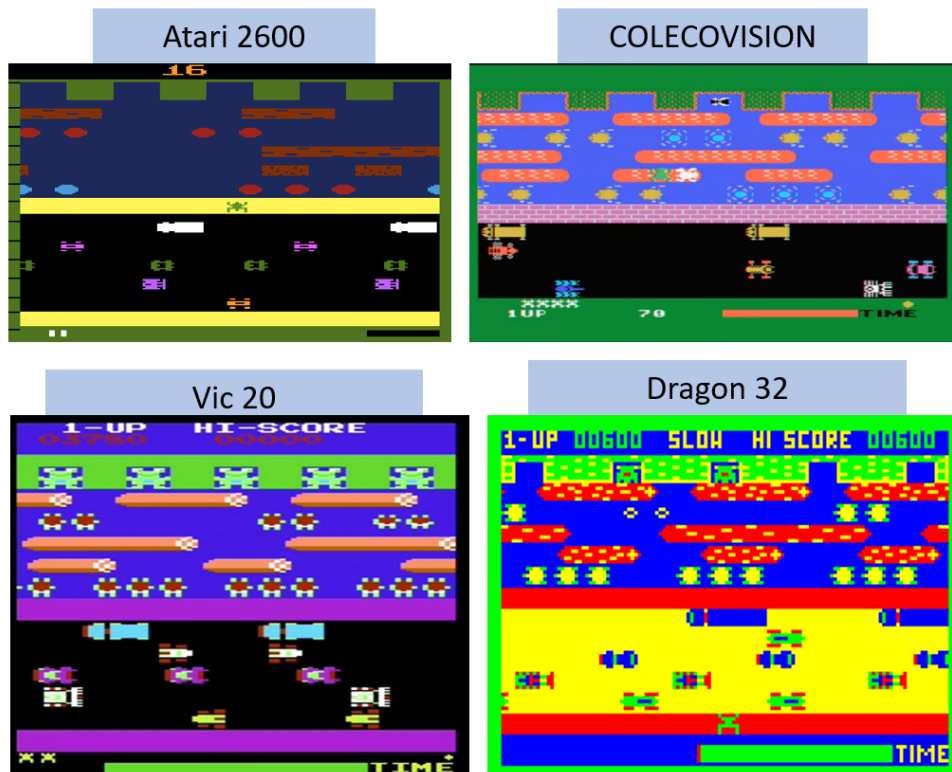
Frogger fue un juego desarrollado por **Konami** en 1981 para maquina arcade basada en Z80. Se convirtió en un clásico, un juego de referencia en la historia de los videojuegos.



Fig. 1 El clásico “Frogger”

El planteamiento del juego es sencillo. Debes manejar una rana y cruzar una carretera por la que pasan coches, y posteriormente un rio lleno de peligros. Al final del rio hay unas moscas y cada vez que te comes una vuelves a la zona inferior de la pantalla. Cuando te comes todas pasas de nivel, incrementando la dificultad.

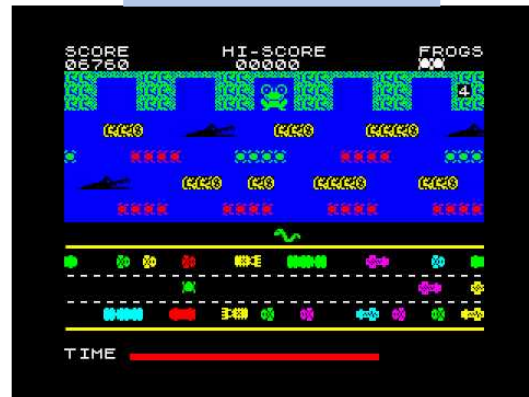
Hay versiones de frogger para todos los sistemas. Frogger es considerado como uno de los 10 mejores videojuegos de todos los tiempos según Killer List of Videogames (KLOV). El KLOV es un sitio web creado en 1991 que se dedica a catalogar juegos pasados y presentes



Atari 400/800



ZX spectrum



Amstrad CPC



MSX



Commodore 64



Apple II



Fig. 2 distintas adaptaciones de frogger

2 Versión de frogger programada con 8BP

La versión de frogger programada con 8BP ha sido titulada “frogger eterno”, en conmemoración del evento “amstrad eterno” celebrado el 30 de marzo de 2019.

Es un juego desarrollado en MODE 1 (se aprecia en los gráficos de alta resolución) y su listado BASIC ocupa 10KB.

Aunque algunos de los elementos del frogger original no han sido incluidos, tales como los cocodrilos o la capacidad de las tortugas para sumergirse, el juego refleja bastante bien el espíritu del juego original.

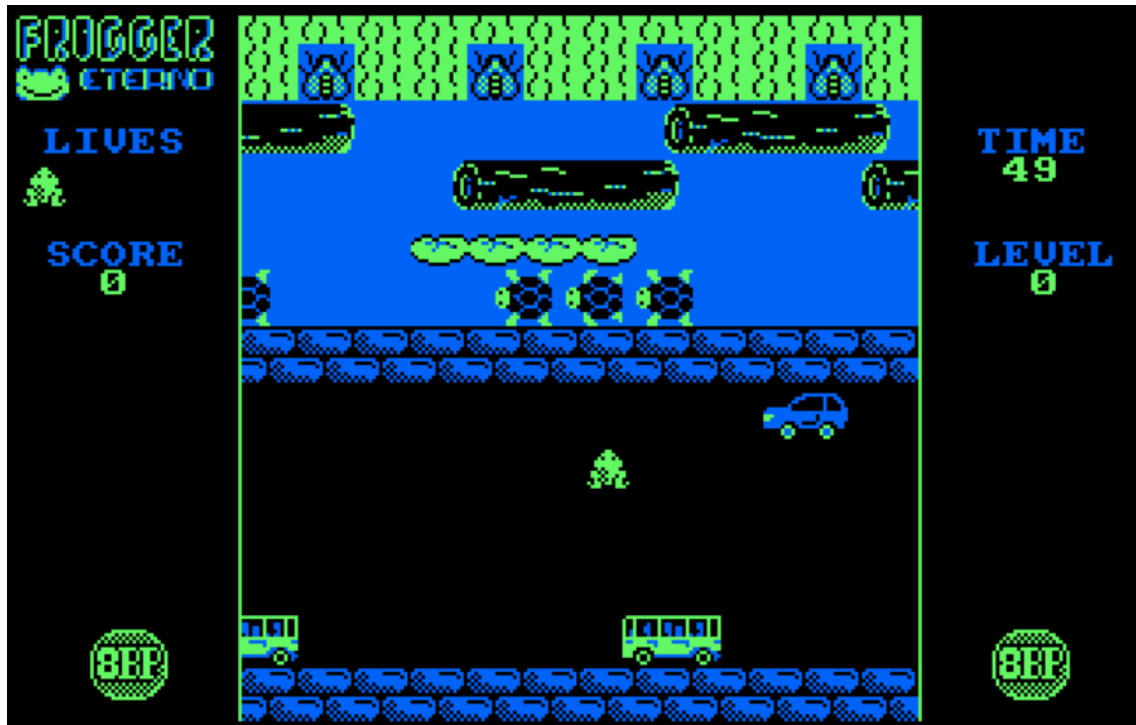


Fig. 3 Versión de frogger realizada con 8BP

Como puedes apreciar, el número de colores que aparecen en pantalla son 3 : negro, azul y verde. En mode 1 el Amstrad soporta 4 colores, pero para lograr que la rana sea capaz de soportar sobreescritura, es decir para que pueda pasar por encima de los ladrillos sin romperlos, 8BP utiliza una técnica que requiere de la reducción de colores simultáneos. A pesar de ello se pueden hacer buenos graficos con poco color.

Para programar con 8BP necesitas el WinApe (emulador/editor/ensamblador de amstrad para windows). Debes crear la estructura de carpetas de 8BP

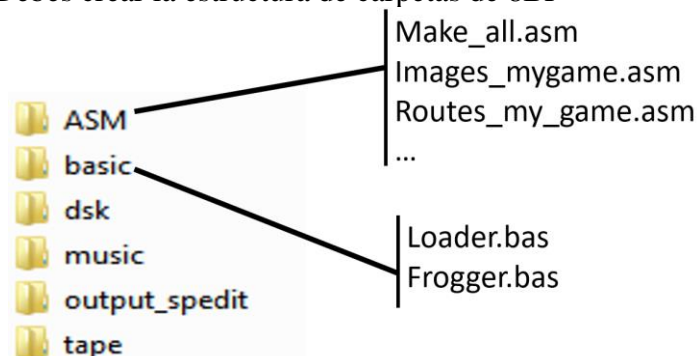


Fig. 4 Estructura de carpetas del juego

3 Breve introducción a 8BP

8BP es una librería de rutinas útiles para videojuegos y accesibles desde BASIC mediante comandos RSX (Resident System eXtensions)

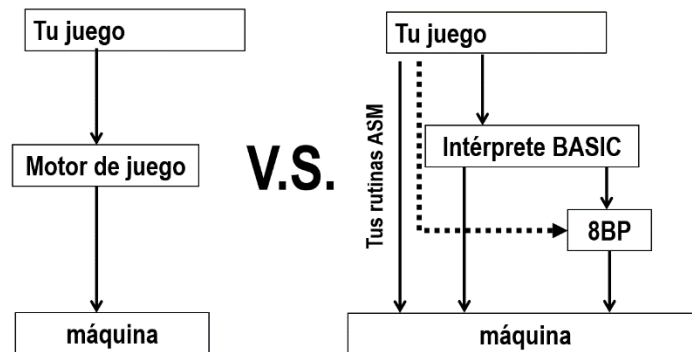


Fig. 5 motor vs librería 8BP

8BP Sólo ocupa 8 KB y te proporciona 27 comandos, 24 KB libre para BASIC, 1.4 KB música y 8.5 KB para sprites

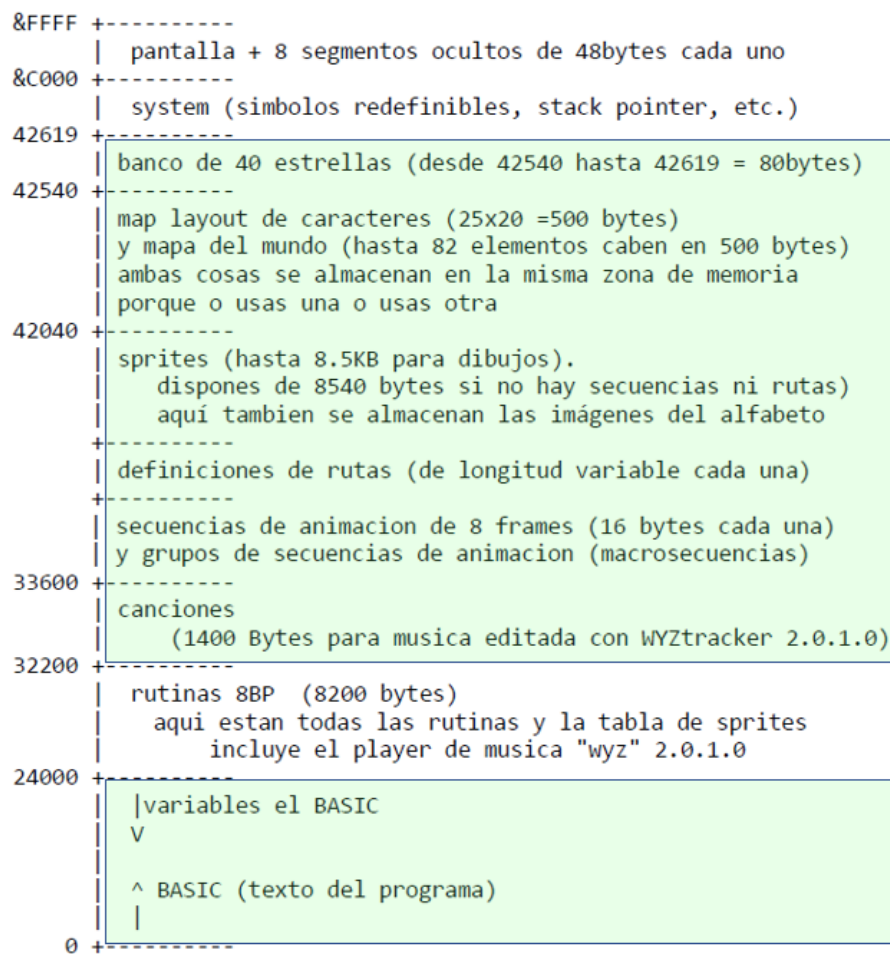


Fig. 6 mapa de memoria de 8BP

Las capacidades de 8BP son:

- 32 sprites con clipping (SETLIMITS), sobreescritura, ordenación y detección de colision (COLSPALL).
- Comandos para mover N sprites a la vez (MOVERALL, AUTOALL, ROUTEALL...)
- Secuencias de animacion y macrosecuencias (cualquier sprite puede cambiar su secuencia de animacion dependiendo de su Vx,Vy)
- Enrutado de sprites automatico con rutas definibles (loops, saltos,...)
- Scroll multidireccional (comandos MAP2SP y UMAP)
- Permite musica in-game basada en WYZtracker 2.0.1.0(comandos MUSIC and MUSICOFF)
- Capacidad de juegos con layout ("tile map"), con detección de colisión.
- Capacidad de animacion por tintas (RINK)
- Set de Minicaracteres definibles para usar en tus juegos (PRINTAT)
- Comando STAR para efectos de estrellas, tiera, Lluvia...
- Capacidad PSEUDO-3D
- Sólo ocupa 8 KB y reserva 8.5KB para sprites y 1.4KB para musica, dejando 24 KB para lógica BASIC.

En lo relativo a sprites:

- 8BP soporta 32 sprites de cualquier tamaño
- 16 bytes por sprite (9 parámetros)
- La tabla con la información de sprites comienza en 27000 (consulta manual)
- Podemos leer con PEEK o con |PEEK
- Podemos escribir sus parámetros con POKE o con |POKE o con |SETUPSP
- El primer byte es el de estado
- Los puedes colocar con |LOCATESP

	1byte	2 bytes	2 bytes	1byte	1byte	1byte	1byte	2 bytes	1byte
sprite	status	coordy	coordx	vy	vx	seq	frame	imagen	ruta
0	27000	27001	27003	27005	27006	27007	27008	27009	27015
1	27016	27017	27019	27021	27022	27023	27024	27025	27031
2	27032	27033	27035	27037	27038	27039	27040	27041	27047
3	27048	27049	27051	27053	27054	27055	27056	27057	27063
4	27064	27065	27067	27069	27070	27071	27072	27073	27079
5	27080	27081	27083	27085	27086	27087	27088	27089	27095
6	27096	27097	27099	27101	27102	27103	27104	27105	27111
7	27112	27113	27115	27117	27118	27119	27120	27121	27127
8	27128	27129	27131	27133	27134	27135	27136	27137	27143
9	27144	27145	27147	27149	27150	27151	27152	27153	27159
10	27160	27161	27163	27165	27166	27167	27168	27169	27175
11	27176	27177	27179	27181	27182	27183	27184	27185	27191
12	27192	27193	27195	27197	27198	27199	27200	27201	27207
13	27208	27209	27211	27213	27214	27215	27216	27217	27223
14	27224	27225	27227	27229	27230	27231	27232	27233	27239
15	27240	27241	27243	27245	27246	27247	27248	27249	27255
16	27256	27257	27259	27261	27262	27263	27264	27265	27271
17	27272	27273	27275	27277	27278	27279	27280	27281	27287
18	27288	27289	27291	27293	27294	27295	27296	27297	27303
19	27304	27305	27307	27309	27310	27311	27312	27313	27319
20	27320	27321	27323	27325	27326	27327	27328	27329	27335
21	27336	27337	27339	27341	27342	27343	27344	27345	27351
22	27352	27353	27355	27357	27358	27359	27360	27361	27367
23	27368	27369	27371	27373	27374	27375	27376	27377	27383
24	27384	27385	27387	27389	27390	27391	27392	27393	27399
25	27400	27401	27403	27405	27406	27407	27408	27409	27415
26	27416	27417	27419	27421	27422	27423	27424	27425	27431
27	27432	27433	27435	27437	27438	27439	27440	27441	27447
28	27448	27449	27451	27453	27454	27455	27456	27457	27463
29	27464	27465	27467	27469	27470	27471	27472	27473	27479
30	27480	27481	27483	27485	27486	27487	27488	27489	27495
31	27496	27497	27499	27501	27502	27503	27504	27505	27511

Fig. 7 Tabla de direcciones de sprites de 8BP

Uno de los bytes mas importantes de cada sprite es su byte de estado, que define su comportamiento al invocar los diferentes comandos de 8BP

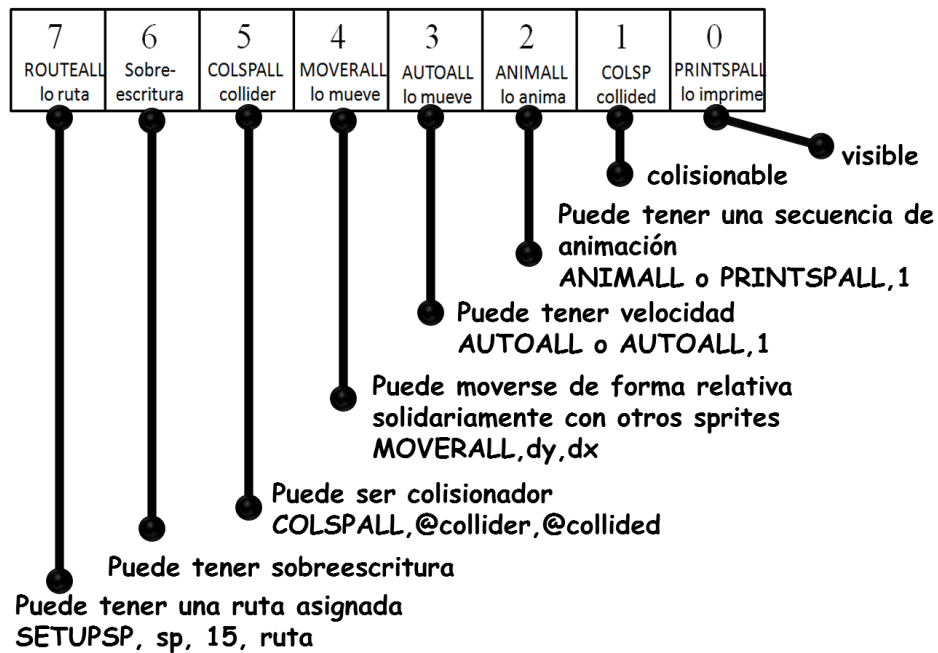
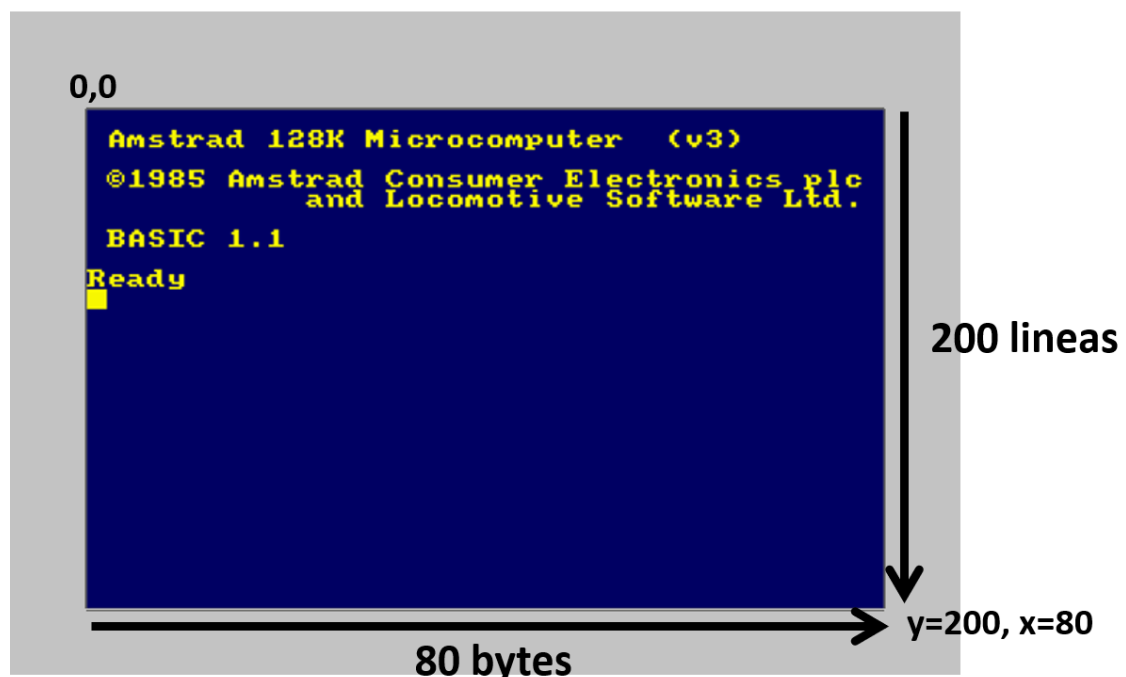


Fig. 8 Byte de estado de un sprite

A la hora de colocar en pantalla un sprite (por ejemplo, con el comando LOCATESP) debes tener en cuenta el sistema de coordenadas de 8BP:



1 BYTE = 2 pixels de MODE 0

1 BYTE = 4 pixels de MODE 1

Fig. 9 coordenadas de pantalla en 8BP

El siguiente ejemplo coloca una rana en el centro de la pantalla (asumiendo que la imagen 16 es la de una rana):

```

10 MEMORY 23999
20 CALL &6B78
30 DEFINT A-Z
35 frogimg = 16
40 |SETUPSP,31,0,1

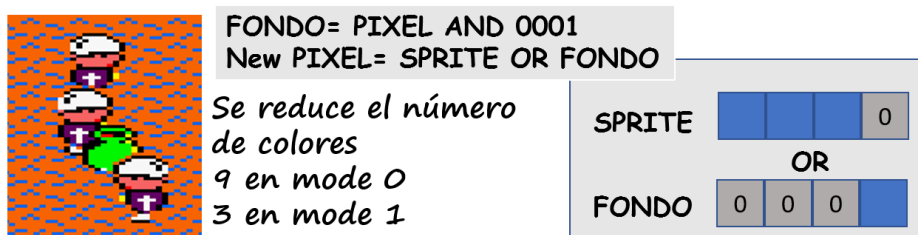
```

```

50 |SETUPSP,31,9,frogimg
55 x=40,y=100
60 |LOCATESP,31,y,x
70 |PRINTSPALL,0,0,0,0

```

La librería de comandos 8BP soporta sprites con sobreescritura. El sistema de sobreescritura de 8BP no usa doble-buffer así que no gasta memoria y es muy rápido. Jamás destruye el fondo. Es una técnica usada en juegos como “mision genocide” y “wonderboy”. Limita el color del fondo a 2 o 4 colores (seleccionable).



- Hay truco para usar mas de 9 colores en MODE 0 con sobreescritura
- 8BP también permite elegir 1 o 2 bit de fondo

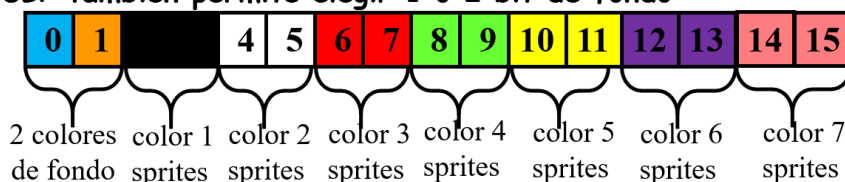


Fig. 10 sobreescritura en 8BP

Con la misma técnica podemos hacer que un sprite pase por delante de ciertos objetos del fondo y por detrás de otros.

8BP permite ordenar los sprites antes de imprimirlos, de modo que sea posible producir un efecto de profundidad, habitual en juegos tipo “Golden Axe”. Para ello usa internamente un algoritmo de “burbuja restringida a un solo cambio” que es lo mas rápido si los sprites están casi ordenados, lo cual suele suceder de un fotograma al siguiente.

Para ordenar los sprites basta indicarlo en el comando que los imprime todos PRINTSPALL

|PRINTSPALL, ordenini, ordenfin, anima, sync

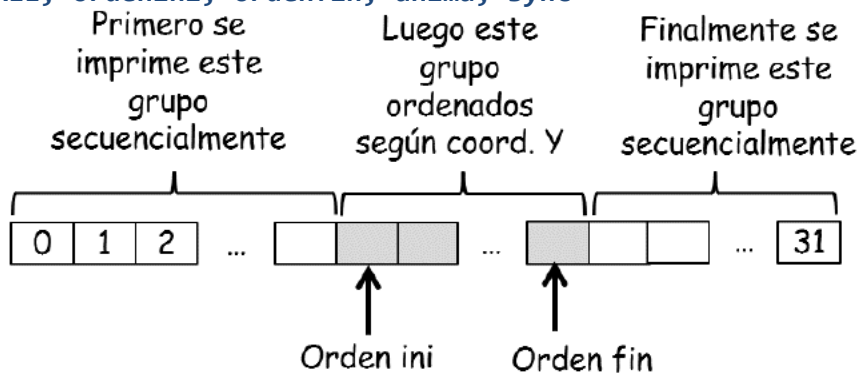
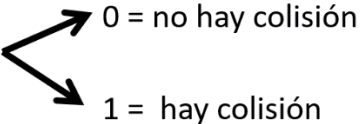


Fig. 11 ordenación de sprites


El comando hay que inicializarlo, pero solo una vez

PRINTSPALL,0	:	ordenacion	parcial	de	sprites	basado	en	Ymin
PRINTSPALL,1	:	ordenacion	total	de	sprites	basado	en	Ymin
PRINTSPALL,2	:	ordenacion	parcial	de	sprites	basado	en	Ymax
PRINTSPALL,3	:	ordenacion	total	de	sprites	basado	en	Ymax

La librería 8BP permite detectar colisiones entre sprites, así como colisiones entre sprites y el layout (o también llamado “mapa de tiles”) del juego mediante los comandos COLSPALL y COLAY respectivamente. En el juego “frogger eterno” se usa un layout, de modo que más tarde veremos como se construye

COLAY, sp, @col 

COLSPALL,@collider , @collided



32 = no hay colisión 32 = no hay colisión
<32 hay colision <32 hay colision

Fig. 12 comandos de colision con Layout y entre sprites

El comando COLAY nos dice si hemos colisionado (SI/NO) con el layout, mientras que COLSPALL nos dice los dos sprites que han colisionado. Como el comando solo nos entrega una colisión entre dos sprites, es importante tener en cuenta el orden de exploración que sigue el comando, ya que es más importante que encuentre primero una colisión de un disparo contra nuestra nave y nos maten, que una colisión de un disparo nuestro contra un enemigo

Funcionamiento de COLSPALL

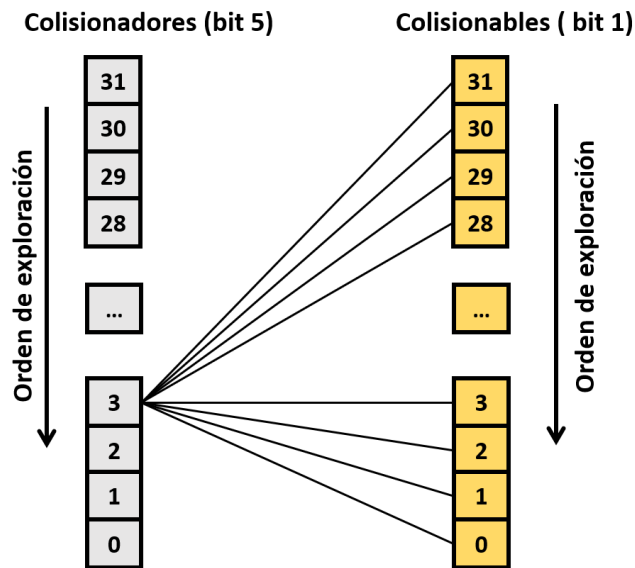


Fig. 13 orden de exploración del comando COLSPALL

En 8BP puedes asignar una ruta a un sprite, para definir su trayectoria y así reducir la lógica de tu ciclo de juego, bastará con que le pidas al sprite que se mueva para que cada vez de un paso en la ruta que tenga asignada.

El siguiente ejemplo se muestra la ruta de un pirata. Al terminar la ruta, el sprite se vuelve a colocar en la posición inicial gracias al último segmento. Al terminar de recorrer la ruta, el sprite vuelve a empezar a recorrerla a menos que uno de los segmentos de la ruta indique un cambio de estado que desactive su enrutado.

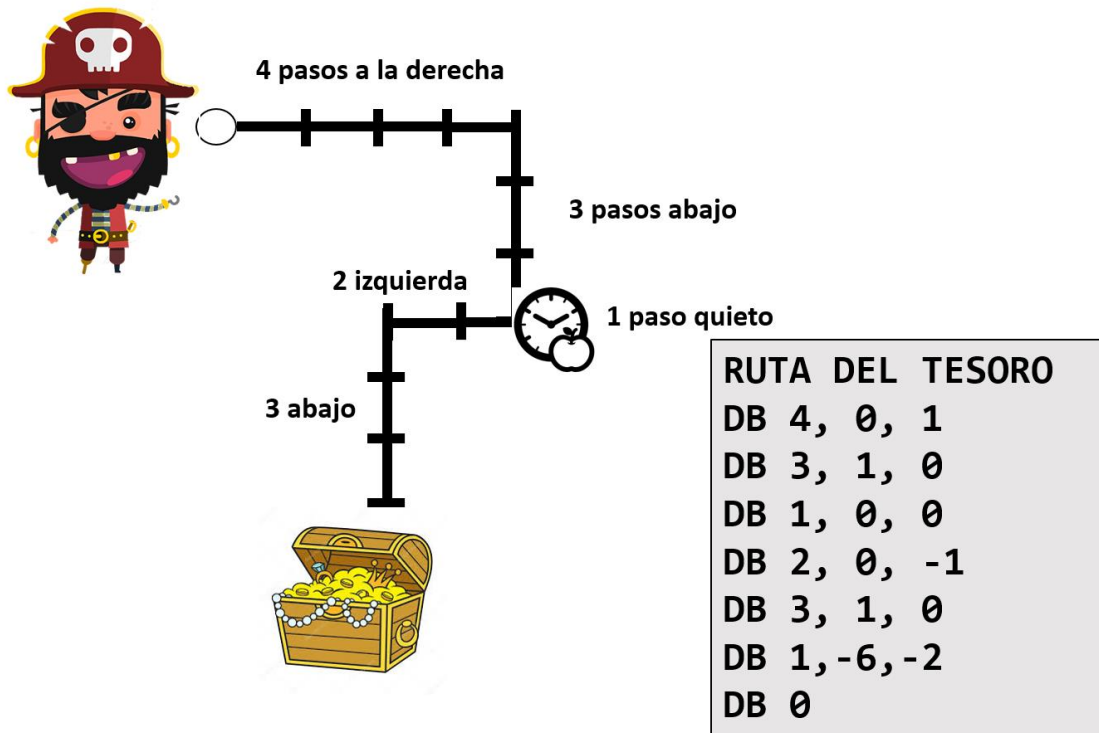


Fig. 14 enrutamiento de sprites

La librería 8BP también soporta scroll multidireccional mediante un mecanismo de ventana deslizante que “atrapa” los elementos del mundo y los transforma en sprites imprimibles, empezando desde el sprite cero en adelante

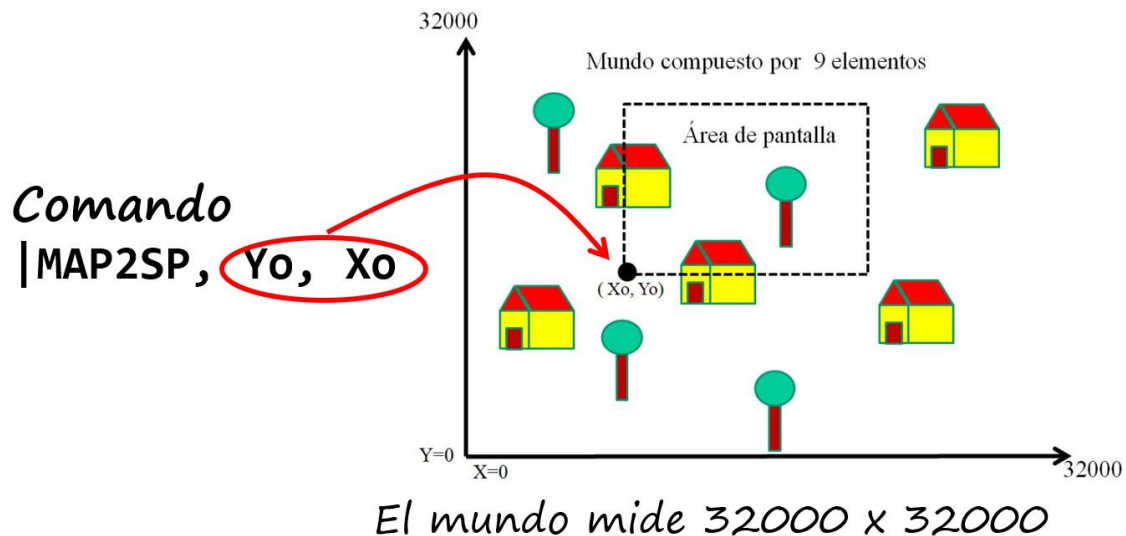


Fig. 15 scroll multidireccional de 8BP basado en una ventana deslizante

Este mecanismo gasta mas sprites cuantos mas elementos hay en pantalla, por lo que si usas scroll, debes crear tus personajes y enemigos desde el sprite 31 hacia abajo y de ese modo no habrá solape entre los sprites que usa el scroll y tus sprites

Si tu mapa del mundo es muy grande, 8BP te permite actualizando periódicamente con el comando UMAP (abreviatura de UPDATE MAP)

8BP también soporta pseudo3D, mediante el comando 3D, el cual te permite recorrer un mundo 2D proyectado en 3 dimensiones. En realidad es muy similar al uso de scroll, pero esta vez, proyectado

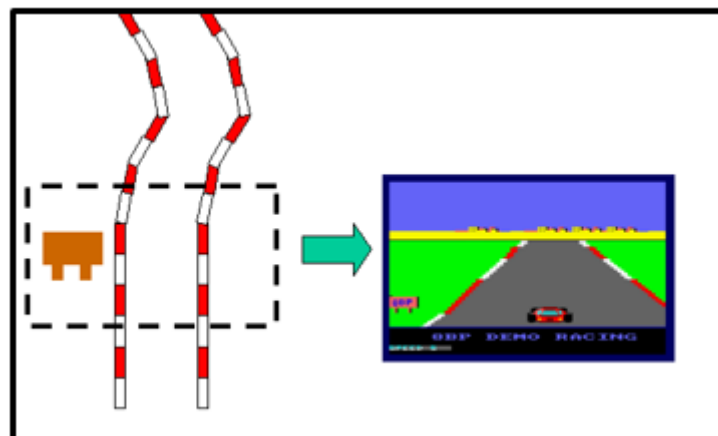


Fig. 16 Pseudo-3D en 8BP

El mecanismo pseudo 3D, también te permite definir imágenes de tipo ZOOM, de modo que al acercase se impriman versiones mayores de la misma imagen

Por ultimo, 8BP te permite poner música on-game, de modo que mientras juegas puede sonar una música compuesta por ti mismo usando el WYZ tracker creado por Augusto Ruiz. EL player de música esta integrado dentro de la librería y puedes hacer sonar una música simplemente invocando al comando MUSIC

4 Diseño de sprites

Todos los sprites de frogger han sido creados con la herramienta SPEDIT, que viene integrada en 8BP como una utilidad disponible en BASIC. Gracias a SPEDIT puedes crear tus propias imágenes y salvarlas en un fichero de texto.

La rana es el único sprite al que se le va a dotar de capacidad de transparencia durante el juego. Es por ello que la paleta de 4 colores se ve reducida a 3, ya que el tercer y cuarto color son iguales como se puede apreciar.



Fig. 17 Edición de sprites con SPEDIT

Una vez diseñada cualquier imagen, al pulsar “b”, se vuelca sobre el fichero de texto que tengas configurado en winape. Ello te genera un texto como el que se muestra a continuación, que es el correspondiente a la rana en reposo:

```
;----- BEGIN IMAGE -----  
db 4 ; ancho  
db 16 ; alto  
db 0 , 0 , 0 , 0  
db 0 , 0 , 0 , 0  
db 0 , 0 , 0 , 0  
db 0 , 1 , 8 , 0  
db 0 , 3 , 12 , 0  
db 0 , 5 , 10 , 0  
db 0 , 15 , 15 , 0  
db 0 , 15 , 15 , 0  
db 0 , 5 , 14 , 0  
db 1 , 10 , 15 , 8  
db 1 , 13 , 13 , 8  
db 0 , 10 , 11 , 0
```

```

db 2 , 13 , 11 , 4
db 1 , 12 , 3 , 8
db 0 , 0 , 0 , 0
db 0 , 0 , 0 , 0
;----- END IMAGE -----

```

Esta imagen debes pegarla en el fichero de imágenes del juego, que en mi caso se llama “images_mygame.asm”

Además de pegarla, debes ponerla una “etiqueta” (un nombre) para poder referenciarla después, por ejemplo “FROG_UP1”:

```

;----- BEGIN IMAGE -----
FROG_UP1
db 4 ; ancho
db 16 ; alto
db 0 , 0 , 0 , 0
db 0 , 0 , 0 , 0
db 0 , 0 , 0 , 0
db 0 , 1 , 8 , 0
db 0 , 3 , 12 , 0
db 0 , 5 , 10 , 0
db 0 , 15 , 15 , 0
db 0 , 15 , 15 , 0
db 0 , 5 , 14 , 0
db 1 , 10 , 15 , 8
db 1 , 13 , 13 , 8
db 0 , 10 , 11 , 0
db 2 , 13 , 11 , 4
db 1 , 12 , 3 , 8
db 0 , 0 , 0 , 0
db 0 , 0 , 0 , 0
;----- END IMAGE -----

```

Al principio del fichero de imágenes aparece una relación de las imágenes que quieres usar. A cada una de ellas 8BP le asigna un numero empezando por el numero 16. Todo esto es ensamblador, pero no son instrucciones, solo datos. No necesitas saber ensamblador. Lo único que necesitas saber es el significado de la palabra “ensamblar”, que es simplemente “cargar datos en memoria”. Y eso es lo que haremos

Este es el fichero “images_mygame.asm”

```

IMAGE_LIST
;-----
; pondremos aqui una lista de las imagenes que queremos usar
; se empiezan a numerar en 16
; podemos usar hasta 255 imagenes especificadas de este modo
;-----
DW FROG_UP1;16
DW FROG_UP2;17
DW STONE;18
DW STONE2;19
DW WATER;20
DW SAND;21
DW CAR1_L;22
DW MONEDA;23
DW FROG_L1_DEL;24
DW FROG_R1_DEL;25

```

```

DW FROG_L1;26
DW FROG_L2;27
DW CAR1_L;28
DW CAR1_R;29
DW TRONCO;30
DW TURTLESA;31
DW MURO;32
DW HOJAS2;33
DW HOJAS4;34
DW RIVER;35
DW MOSCA;36
dw WATERMOSCA;37
DW FROGGER;38

; ahora las imagenes
IMAGE0
;-----
; si no vas a usar el comando PRINTAT, sino simplemente los caracteres del
; amstrad, entonces puedes comentar la siguientes 3 lineas
_BEGIN_ALPHABET
read "alphabet_default.asm"
_END_ALPHABET
;=====
_SPRITE_HEIGHTS_TABLE      ds 32 ; 32 bytes para almacenar las alturas de los
sprites y poderlos ordenar. uso interno.
;=====
_BEGIN_FLIP_IMAGES
;=====
; aqui pon las imagenes que se definen como otras existentes pero flipeadas
;horizontalmente.
FROG_L1      dw FROG_R1
FROG_L2      dw FROG_R2
FROG_L1_DEL dw FROG_R1_DEL
;=====
_END_FLIP_IMAGES
;=====
_BEGIN_IMAGES
;----- BEGIN IMAGE -----
FROG_UP2
db 4 ; ancho
db 16 ; alto
db 2 , 8 , 1 , 4
db 3 , 0 , 0 , 12
db 1 , 1 , 8 , 8
db 1 , 11 , 13 , 8
db 0 , 13 , 11 , 0
db 0 , 7 , 14 , 0
db 0 , 13 , 15 , 0
db 0 , 10 , 15 , 0
db 0 , 5 , 14 , 0
db 0 , 10 , 15 , 0
db 0 , 9 , 9 , 0
db 3 , 12 , 3 , 12
db 1 , 8 , 1 , 8
db 2 , 8 , 1 , 4
db 0 , 0 , 0 , 0
db 0 , 0 , 0 , 0
;----- END IMAGE -----

```

A medida que vamos creando las imágenes, las incluimos en el fichero de imágenes. Y usarlas es tan sencillo como asignársela a un sprite e imprimirlo.
Por ejemplo, para imprimir el título del juego en los marcadores, simplemente hacemos:

```
600 |SETUPSP,0,0,1:|SETUPSP,0,9,frogger:|PRINTSP,0,0,0
```

Siendo la variable `frogger = 38`, que es el número asignado a la imagen que usamos como título. Dicho valor se asigna al principio, en la línea 280



Fig. 18 imagen título del juego

No todos los sprites tienen un número asignado pues no son referenciados desde BASIC, aunque aun así se pueden usar. Por ejemplo, se puede crear una secuencia de animación que contenga las etiquetas (los nombres) de varias imágenes que no son incluidas en el listado inicial de imágenes y al asignar esa secuencia a un sprite, sus imágenes se usan. Y lo mismo pasa con una ruta en la que se definan imágenes. Si la ruta indica que imágenes se deben asignar al sprite, no hace falta más que asignarle la ruta para que el sprite use esas imágenes.

Fíjate que he cometido el error de incluir “CARL1_L” dos veces en la lista inicial del fichero `images_mygame.asm`, por lo que se puede referenciar tanto con el número 22 como con el 28.

Imágenes de “Frogger Eterno”

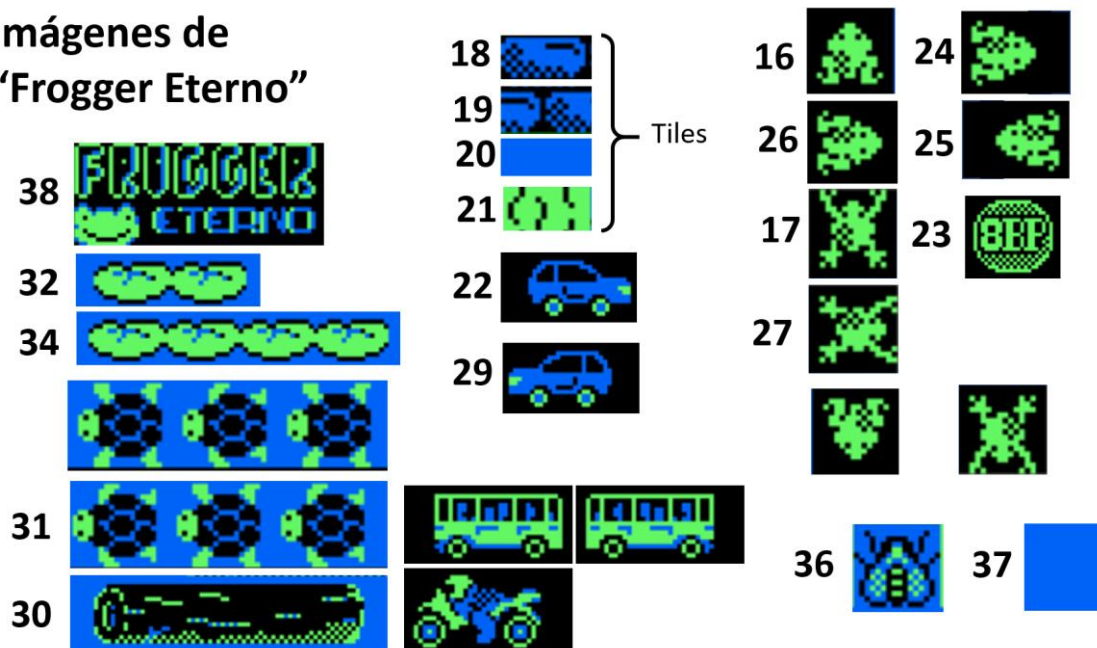


Fig. 19 imágenes de Frogger Eterno

Como puedes observar, los vehículos han sido dibujado sobre fondo negro, de modo que al desplazarse se borran a sí mismos (borran su “rastro”) sobre una carretera negra.

Por el contrario, las tortugas, los nenúfares y los troncos tienen fondo azul, para que al desplazarse se borren a sí mismos dejando el río del color que tiene, azul.

Algunos sprites no han sido creados, sino que simplemente se ha especificado que se use la versión volteada del mismo sprite. Esto se indica en la sección de imágenes volteadas del archivo de imágenes. En concreto se voltean 3 imágenes:

- la rana en horizontal en reposo (**FROG_L1**)
- la rana en horizontal saltando (**FROG_L2**)
- la rana en horizontal en reposo, pero con algo más de ancho (**FROG_L1_DEL**). Esta imagen es auxiliar, de momento no le des importancia.

```
;=====
_BEGIN_FLIP_IMAGES
;=====
; aquí pon las imágenes que se definen como otras existentes pero volteadas
;horizontalmente.
FROG_L1      dw FROG_R1
FROG_L2      dw FROG_R2
FROG_L1_DEL  dw FROG_R1_DEL
;=====
_END_FLIP_IMAGES
```

Los coches y autobuses podrían haberse volteado para ahorrar memoria, pero como este juego no consume muchos gráficos, decidí no usar el mecanismo de flipping en ellos y así aumentar la velocidad, ya que el flipping es costoso y si lo usas masivamente se puede llegar a notar que el juego va algo más despacio.

5 Estructura de un videojuego

Aunque puedan existir ciertas variaciones, casi todos los videojuegos se pueden estructurar en varias partes que se presentan a continuación

Inicio y Presentación
Lógica del programa principal GOSUB pantalla 1 GOSUB pantalla 2 GOTO FIN
Pantalla N: inicialización de sprites pintado de escenario ciclo de juego: leer teclado y mover personaje decisiones logicas mover sprites imprimir sprites detectar colisiones y lógica asociada

La parte mas importante del juego es lo que llamamos el “ciclo de juego”, que es donde se realizan todas las tareas del juego, mientras que el resto de la estructura del programa es todo aquello que sirve para inicializar el juego y organizarlo en fases o pantallas.

Cada una de las partes del ciclo de juego pueden ser rutinas. De este modo podemos simplificar la escritura del ciclo de juego invocando a estas rutinas. Podemos hacer una rutina de lectura de teclado y movimiento del personaje, otra para la lógica de las colisiones, otra para tomar decisiones logicas tales como crear nuevos enemigos, etc. Todas estas pequeñas rutinas las invocaremos desde el ciclo de juego.

6 Diseño general

Para programar el juego, podemos estructurar el juego que tenemos ante nosotros como una superposición de “capas”,

- Capa de marcadores
- Capa de mapa de tiles
- Capa de sprites

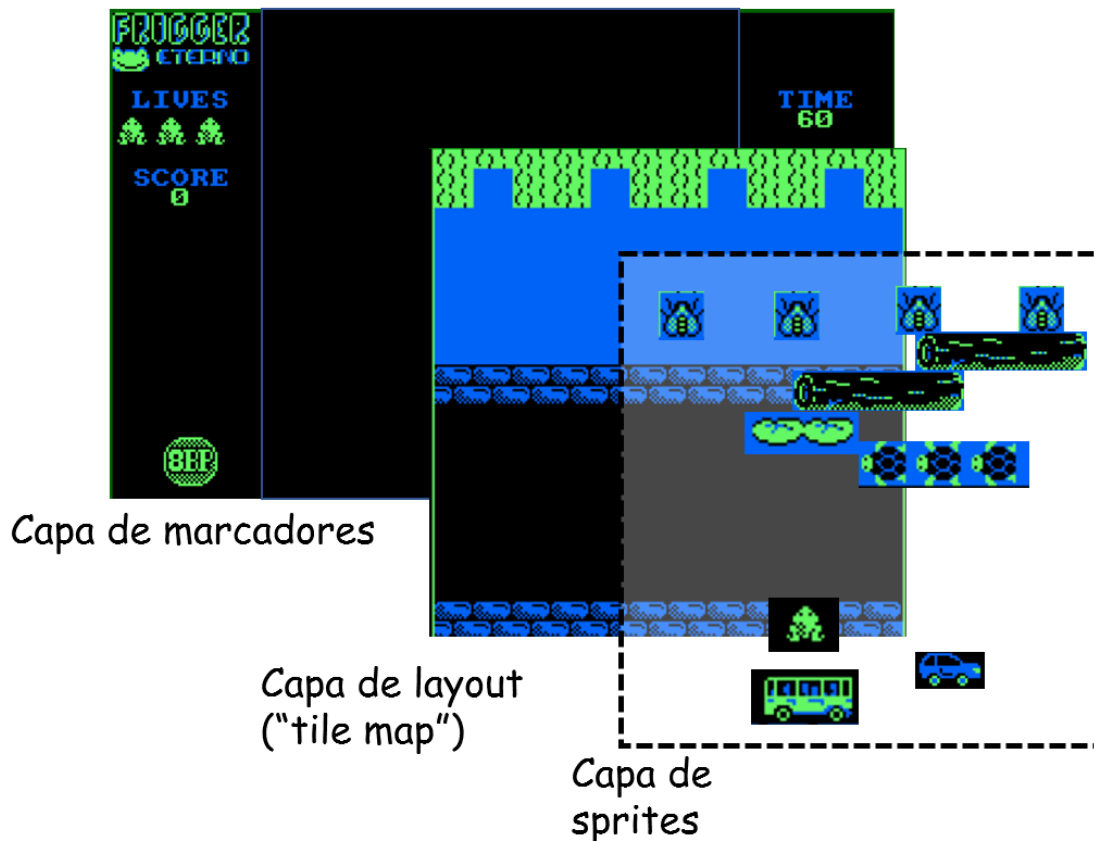


Fig. 20 Diferentes capas del juego

La capa de marcadores simplemente la imprimimos una vez al principio. Colocamos los textos y asignamos la imagen de la rana a un sprite que imprimimos 3 veces con **PRINTSP**. El título también es una imagen que asociamos a un sprite el cual imprimimos con **PRINTSP**. Durante el juego lo único que se va a actualizar periódicamente es el indicador de tiempo restante, lo cual consume poco tiempo.

El layout es un dibujo de fondo, que solo se imprime una vez mediante el comando **LAYOUT**. La rana utiliza impresión transparente y puede pasar por encima de los ladrillos que hacen de bordes de la carretera sin romperlos. Los colores de fondo son dos: azul y negro. La impresión transparente de 8BP en mode 1 permite respetar fondos contruidos con dos colores. Tan solo la rana usa impresión transparente pues es el único sprite que lo necesita.

Por último, está la capa de sprites. En cada ciclo de juego imprimiremos todos los sprites, pero no uno por uno sino todos juntos mediante el comando **PRINTSPALL**

6.1 capa de marcadores

Durante el juego se establecen los límites para el pintado de sprites en el área de juego, en la línea 940

940 |SETLIMITS,16,64,0,200

Sin embargo, los marcadores están fuera de esa área, en parte a la izquierda y en parte a la derecha. Es importante tener en cuenta que los límites establecidos en el comando SETLIMITS solo afectan al pintado de sprites, y no al texto que podamos imprimir con el comando PRINT de BASIC.

Puesto que dentro de los marcadores hay sprites (las vidas y el título del juego), la rutina que pinta los marcadores establece unos límites que abarcan toda la pantalla

```
580 '--- marcadores
590 |SETLIMITS,0,80,0,200
600 |SETUPSP,0,0,1:|SETUPSP,0,9,frogger:|PRINTSP,0,0,0
610|SETUPSP,0,0,1:|SETUPSP,0,9,23:|PRINTSP,0,172,4:|PRINTSP,0,172,68:
|SETUPSP,0,0,0
620 PEN 1:LOCATE 2,5: PRINT "LIVES": LOCATE 35,5:PRINT "TIME": PEN 2:
LOCATE 35,6: PRINT t
630 LOCATE 2,9: PEN 1:PRINT "SCORE": LOCATE 3,10:PEN 2:PRINT score
640 LOCATE 35,9: PEN 1:PRINT "LEVEL": PEN 2:LOCATE 36,10:PRINT (30-
dificultad)/5
650 FOR i=0 TO LIVES-1:|PRINTSP,31,40,i*4:NEXT
```

El resultado es el que se muestra a continuación:

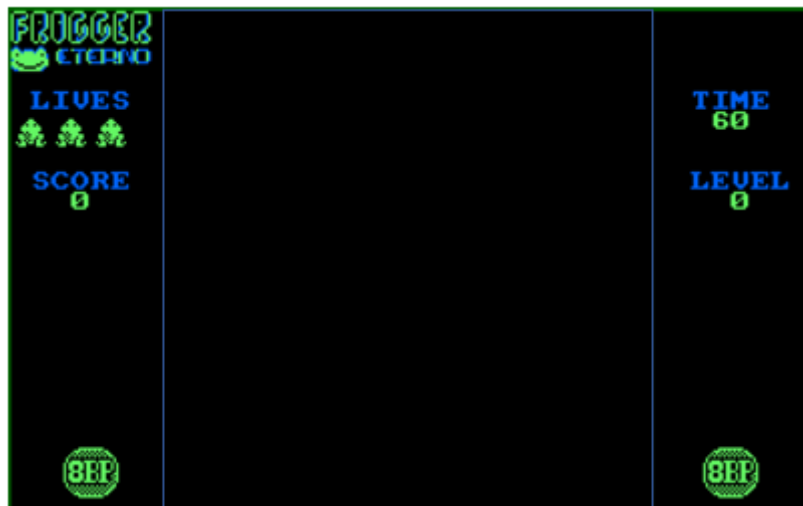


Fig. 21 marcadores de frogger

Para poder disminuir el tiempo poco a poco, dentro de la lógica del juego hay un mecanismo que disminuye e imprime el tiempo cada 32 ciclos de juego (cada 32 fotogramas)

```
1280 IF ciclo AND 31 THEN 1300
1290 t=t-1:LOCATE 35,6:PRINT t: IF t=0 THEN t=tmax: GOTO 1520
1300 ...
```

La expresión “ciclo AND 31” solo vale 0 cuando ciclo es multiplo de 31 y por lo tanto solo uno de cada 32 ciclos se ejecuta la línea 1290, que es donde se reduce el tiempo y se imprime.

6.2 Capa de layout (tile map)

La capa del layout se construye con cadenas de caracteres. Simplemente se asignan letras al los “tiles” del layout. Las letras representan sprites. Hay una correspondencia univoca entre las letras y los sprites. Por ejemplo, la letra “D” es el sprite 9. De modo que primero se asignan imágenes a los sprites y después se construye e imprime el layout.

```

660 '---layout
670 col%=0:|COLAY,67,31,@col%
680 |SETUPSP,6,9,stone: |SETUPSP,7,9,water: |SETUPSP,8,9,stone2:
|SETUPSP,9,9,sand
690 c$(0) = " DDDDDDDDDDDDDDD "
700 c$(1) = " DDBDDDBDDDBDD "
710 c$(2) = " DDBDDDBDDDBDD "
720 c$(3) = " DBBBBBBBBBBBBD "
730 c$(4) = " DBBBBBBBBBBBBD "
740 c$(5) = " DBBBBBBBBBBBBD "
750 c$(6) = " DBBBBBBBBBBBBD "
760 c$(7) = " DBBBBBBBBBBBBD "
770 c$(8) = " DBBBBBBBBBBBBD "
780 c$(9) = " DBBBBBBBBBBBBD "
790 c$(10) = " DBBBBBBBBBBBBD "
800 c$(11) = " DAAAAAAAAAAAAAD "
810 c$(12) = " DCCCCCCCCCCCCCD "
820 c$(13) = " D D "
830 c$(14) = " D D "
840 c$(15) = " D D "
850 c$(16) = " D D "
860 c$(17) = " D D "
870 c$(18) = " D D "
880 c$(19) = " D D "
890 c$(20) = " D D "
900 c$(21) = " D D "
910 c$(22) = " D D "
920 c$(23) = " DAAAAAAAAAAAAAD "
930 c$(24) = " DCCCCCCCCCCCCCD "
940 |SETLIMITS,16,64,0,200
950 FOR i=0 TO 24:|LAYOUT,i,0,@c$(i):NEXT:'print layout
960 PLOT 31*4+2,0,2:DRAW 31*4+2,400
970 PLOT 640-32*4,0,2:DRAW 640-32*4,400

```

Para aclararlo aun mas lo mejor es ver la tabla de correspondencias. El hecho de que una letra se corresponda un cierto sprite, implica que la imagen que tenga el sprite en el momento de imprimirse el layout es la imagen que se usará cada vez que aparezca esa letra. Una vez impreso el layout, el sprite utilizado para una letra cualquiera puede ser reconfigurado para que tenga otra imagen, y da igual, pues el layout ya está impreso.

La primera vez que se invoca al comando COLAY, que sirve para detectar colisiones con el layout, se hace con un parámetro extra, el 67. Me refiero a la línea 670. Significa que las letras con ASCII menores o iguales a 67 no se considerarán colisión (como es el caso de las letras A y C con las que se hacen los ladrillos).

670 col%=0: |COLAY,67,31,@col%

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 ← hemos usado este como umbral de colisión
“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 1 correspondencia entre caracteres y Sprites para el comando /LAYOUT



Fig. 22 Layout de flogger

6.3 Capa de sprites

Los sprites en 8BP tiene un byte de estado que permite definir el comportamiento del sprite ante determinados comandos.

La rana usa el sprite 31 y su estado se configura en la línea 550. Se activan los flags de colisionador, transparencia y printable

```
540 '--- setup frog
550 |SETUPSP,31,0,1+64+32: 'colider,transp,printable
560 |SETUPSP,31,9,frogup1
```

7	6	5	4	3	2	1	0
ROUTEALL	Sobre-	COLSPALL	MOVERALL	AUTOALL	ANIMALL	COLSP	PRINTSPALL
lo ruta	escritura	collider	lo mueve	lo mueve	lo anima	collided	lo imprime

Fig. 23 Flags en el byte de estado

Los sprites como los coches, autobuses, motos se crean con el flag colisionable (el bit 1) activo, de modo que la rana puede colisionar con ellos.

Hay unos sprites especiales en el juego que no se crean con el flag de imprimible, de modo que no se imprimen, pero tienen el flag de colisionable activo. De ese modo pueden matarnos, aunque no se impriman. Son “invisibles”: Estos sprites invisibles son dos muros a los lados del rio y 4 “ríos”

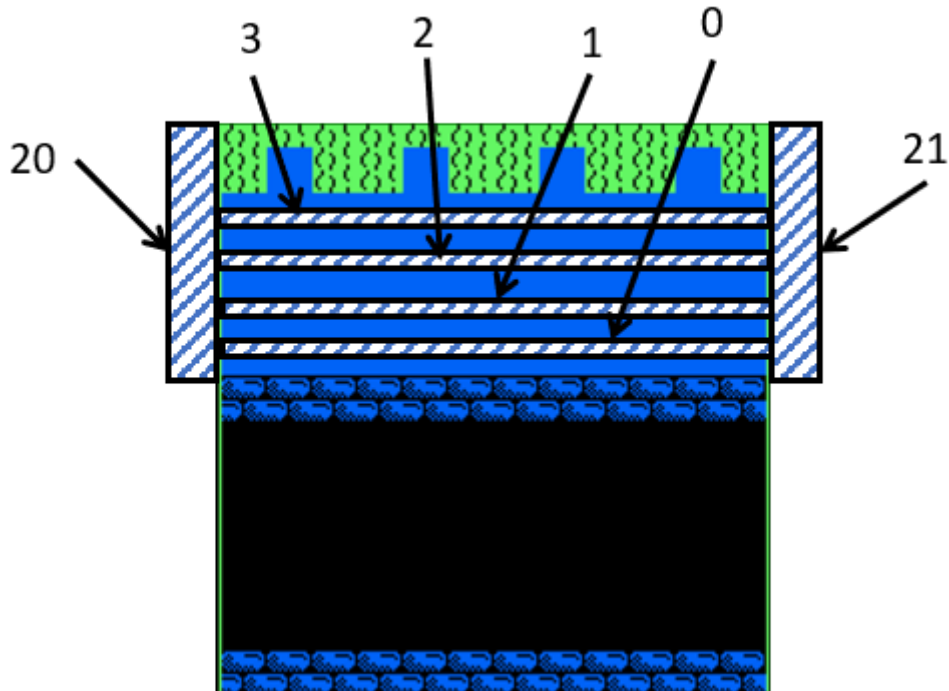


Fig. 24 sprites invisibles pero colisionables

Como no hace falta imprimirlos, estos sprites no le cuestan nada al comando **PRINTSPALL** aunque si que le hacen trabajar al comando **COLSPALL**.

Estos sprites se crean en las líneas:

1070		SETUPSP,0,0,2:		SETUPSP,0,9,river:		LOCATESP,0,72+4,16::	'rio 4
1080		SETUPSP,1,0,2:		SETUPSP,1,9,river:		LOCATESP,1,56+4,16::	'rio 3
1090		SETUPSP,2,0,2:		SETUPSP,2,9,river:		LOCATESP,2,40+4,16::	'rio 2
1100		SETUPSP,3,0,2:		SETUPSP,3,9,river:		LOCATESP,3,24+4,16::	'rio 1
1110		SETUPSP,20,0,2:		SETUPSP,20,9,muro:		LOCATESP,20,24,8:	'MURO L
1120		SETUPSP,21,0,2:		SETUPSP,21,9,muro:		LOCATESP,21,24,64:	'MURO R

Cuando se invoque el comando **COLSPALL**, tanto los 4 ríos como los dos muros serán considerados en la colisión, aunque no se impriman.

Las moscas también son sprites invisibles, pero se imprimen una vez al principio. Una vez impresas, puesto que no se mueven, ya no hace falta imprimirlas mas y por ello se crean con el flag de printable a cero.

Cuando te matan, tanto los ríos como los muros se vuelven a crear. En cuanto a las moscas, solo se crean aquellas que aun no te hayas comido. Veamos como se crea una mosca. Fíjate que solo se crea si `mosca(0)` vale algo diferente de cero.

1130	IF	mosca(0)	THEN
	SETUPSP,22,0,2:		SETUPSP,22,9,mosca:
	LOCATESP,22,8,20:		PRINTSP,22:
ELSE		PRINTSP,31,8,20:	'MOSCA 1

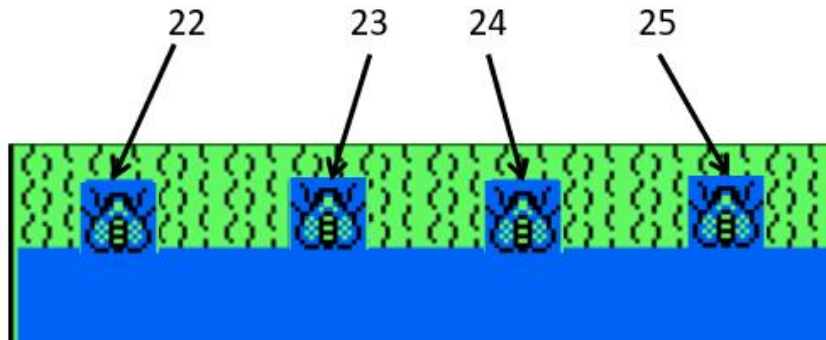


Fig. 25 moscas

Los vehículos (coches, motos, autobuses) se crean dinámicamente. Durante el ciclo de juego de forma periodica se invoca a la rutina de la línea 1540, la cual crea sprites desde el 12 hasta el 19 (ambos inclusive), pudiendo existir hasta 8 vehiculos a la vez. Fijate en como usa la lógica binaria (operación AND) para ir incrementando en 1 el identificador de sprite en cada invocación.

```

1540 '--- new car
1550 car=1+car AND 7:'max simultaneos cars = 8
1560 carsp=car+12:'12 en adelante. 12+7=19, el 20 ya esta libre
1570 p=(p+1 + RND*5) MOD 5:'pista
1580 |SETUPSP,carsp,9,16:|SETUPSP,carsp,0,128+8+2+1:
|SETUPSP,carsp,15,r(p):|LOCATESP,carsp,yc(p),xc(p)
1590 RETURN

```

Los vehículos se crean con el flag de ruta activo, el de movimiento automatico, el de colisionable y el de impresión.

La ruta asignada es aleatoria y la propia ruta contiene la imagen del vehiculo que se ha de mostrar asi como la descripción del movimiento (dirección y velocidad). Hay un total de 5 rutas definidas para los coches. Una por cada “pista” de la carretera.

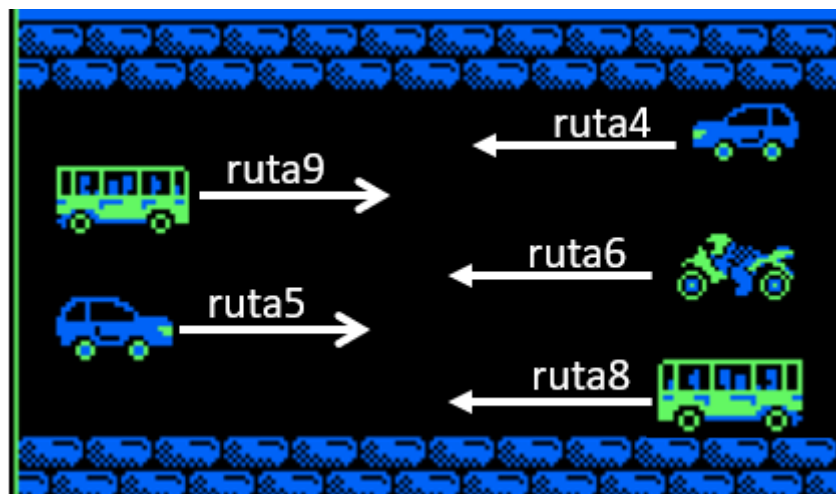


Fig. 26 Rutas de vehículos

Las rutas se guardan en el array r(p) y sus coordenadas iniciales en los arrays yc(p), xc(p), los cuales se crean al principio del juego:

```

330 xc(0)=64:xc(1)=12:xc(2)=64:xc(3)=12:xc(4)=64
340 yc(0)=104:yc(1)=120:yc(2)=136:yc(3)=152:yc(4)=168
350 r(0)=4:r(1)=9:r(2)=6:r(3)=5:r(4)=8

```

Veamos ahora los sprites asociados a los elementos móviles del río.

Se crean a partir de la línea 980. Por ejemplo, en la línea 990 se crea el primer tronco.

```
980 '--- setup river ---
```

```
990 |SETUPSP,4,0,128+8+2+1:|SETUPSP,4,9,tronco:|SETUPSP,4,15,10:
```

```
|LOCATESP,4,24,0:'tronco 1A
```

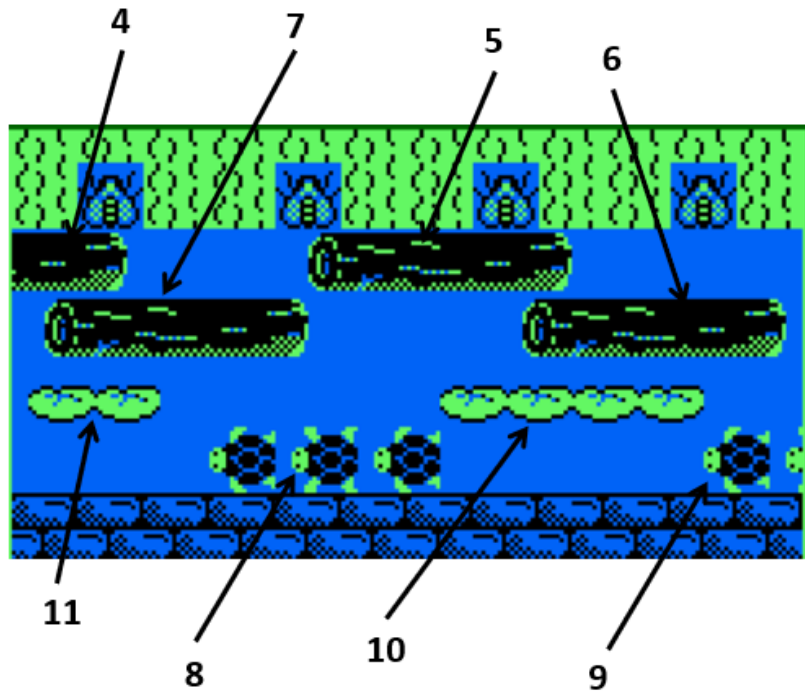


Fig. 27 sprites acuáticos

7 Presentación

Antes de empezar el juego, se inicializa la librería 8BP, mediante el comando CALL &6B78. Justo después se inicializa la paleta de colores a los valores por defecto por si tuviesen algún valor anterior con CALL &BC02.

A continuación, se establecen los límites en los que se va a poder pintar sprites mediante el comando |SETLIMITS de 8BP y se desactiva el modo 3D. realmente no estaba activado y no era necesario desactivarlo. Fíjate que los límites establecidos en SETLIMITS son toda la pantalla (80 bytes de ancho y 200 líneas de alto). Es decir, inicialmente podemos imprimir sprites en toda la pantalla

```
10 MEMORY 23999
20 CALL &6B78:CALL &BC02
30 MODE 1:BORDER 9:|SETLIMITS,0,80,0,200:|3D,0
40 DEFINT A-Z
50 '---sounds
60 ENT 1,10,-100,3:'mosca
70 ENV 1,1,15,1,15,-1,1:'boing
80 ENT 2,20,-125,1:'boing
90 ENT 3,100,5,3:'muerte
```

La línea “DEFINT A-Z” es muy importante para que las variables del BASIC sean números enteros (no decimales) y así todo funcione mas deprisa.

También se crean las envolventes de tono y sonido que se van a usar para los efectos de sonido especiales (comer una mosca, saltar y morir). Dichos sonidos serán creados mediante el comando SOUND.

A continuación, se imprime la siguiente pantalla de presentación:



Fig. 28 Presentación de frogger

La cual creamos con el siguiente listado:

```

100 '--- PRESENTACION
110 INK 0,0:PAPER 0:PEN 2:LOCATE 14,2: PRINT "FROGGER ETERNO"
120 PEN 1:LOCATE 1,5:PRINT "A fully 100% BASIC game created with 8BP"
130 LOCATE 4,4:PRINT "by Jose Javier Garcia Aranda 2019"
140 LOCATE 2,7:PRINT "8BP: The ultimate RSX library for games";
150 INK 2,22:PEN 2:PRINT "":PRINT "      www.github.com/jjarandal3/8BP"
160 PRINT "      www.8bitsdepoder.blogspot.com"
170 FOR i=0 TO 31:|SETUPSP,i,0,0:NEXT:'reset
180 b$=INKEY$:IF b$<>"" THEN 180
190 ' --- hall of fame
200 DIM pts(11): DIM name$(11):'scores
210 GOSUB 2040:'read score table
220 INK 3,7:PEN 3:LOCATE 16,12:PRINT "Hall of fame":LOCATE 15,13:PRINT "-----"
230 p=1:FOR i=0 TO 9: LOCATE 1,i+14:PEN p:PRINT ,name$(i),pts(i):p=1+(p MOD 3):NEXT
231 |MUSIC,0,6
240 b$=INKEY$:IF b$="" THEN 240 ELSE 250

```

Lo mas importante aquí es la forma de mostrar la tabla de puntuaciones. Se trata de un par de simples arrays de BASIC:

- El array **name\$()** contiene los nombres de los jugadores
- El array **pts()** contiene sus puntuaciones.

La forma de cargar la tabla de puntuaciones en ambos arrays se encuentra en la rutina ubicada en la línea 2040, a la que se invoca mediante **GOSUB 2040**

Tras imprimir la tabla de scores se activa la música con el comando **|MUSIC** y nos quedamos a la espera de que el usuario pulse una tecla (línea 240). Para evitar leer una pulsación anterior del teclado, el buffer de lectura de teclas se limpia en la línea 180.

```
180 b$=INKEY$: if B$<>"" THEN 180
```

Tras pulsar una tecla se para la música (con el comando **|MUSICOFF**) y se crean las variables del juego

```

250 '--- global vars
251 |MUSICOFF
260 frogup1=16:frogup2=17:stone=18:stone2=19:water=20:sand=21
270 moneda=23:tortugas=31:muro=32:hojas2=33:hojas4=34: river=35
280 tronco=30: mosca=36:watermosca=37:frogger=38
290 DIM c$(25):'layout
300 INK 0,0: INK 1,11: INK 2,22: INK 3,22
310 collider%=32:collided%=32:|COLSPALL,@collider%,@collided%
320 maxdemora=10:tmax=60
330 xc(0)=64:xc(1)=12:xc(2)=64:xc(3)=12:xc(4)=64
340 yc(0)=104:yc(1)=120:yc(2)=136:yc(3)=152:yc(4)=168
350 r(0)=4:r(1)=9:r(2)=6:r(3)=5:r(4)=8
360 '---init game

```

Las variables como “frogup1” o “moneda”, contienen el numero de imagen asignada al ensamblar las imágenes que hemos creado nosotros mismos.

Después se crea el array **c\$()** que contiene el layout del juego (también llamado “mapa de tiles”) con el que se construye el escenario con el rio, la vegetación, los ladrillos a los bordes de la carretera

Las variables **collider** y **collided** son muy importantes. Las usa el comando **COLSPALL** para dejar el resultado del análisis de colision de sprites. Tras crear ambas

variables, se inicializa el comando **COLSPALL** con ellas para indicarle donde debe dejar el resultado del análisis de colision.

Tenemos a continuación el tmax que son los segundos que tienes antes de que se te acabe el tiempo y pierdas una vida.

La variable maxdemora es el tiempo en fotogramas que has de esperar desde que tu rana comienza a saltar hasta que recuperas el control de la rana

Las variables xc e yc contienen posiciones donde aparecerán los coches de las cinco pistas que contiene la carretera y el array r() contiene las rutas que van a asignarse a cada una de las 5 pistas.

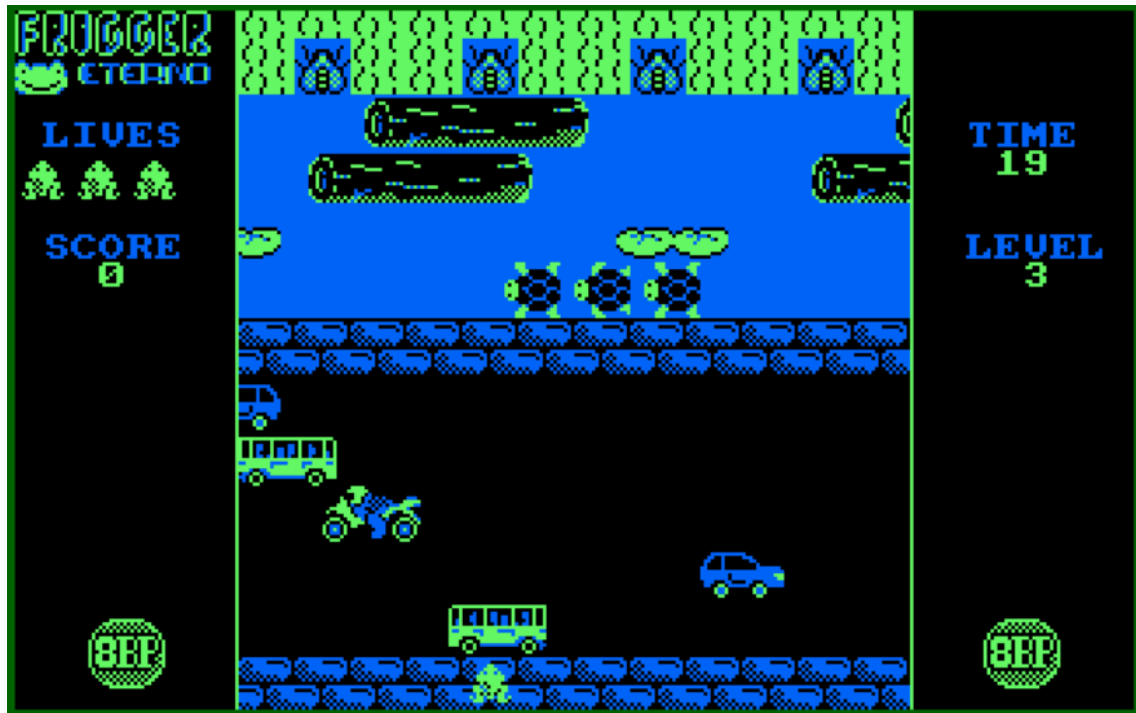


Fig. 29 tras la presentación, comienza el juego

8 Lógica del programa principal

La lógica principal del juego se compone de un bucle en el que se ejecuta la subrutina ubicada en la línea 540, la cual contiene la lógica para pintar el escenario y mover los personajes.

En la línea 390 se inicializan las 4 moscas, y el numero de moscas devoradas se establece en cero. Esta línea debe ejecutarse al principio de cada nivel, pero no debe ejecutarse si simplemente pierdes una vida y aun te quedan un par de moscas por comer.

En la línea 400 se comprueba el numero de moscas que se ha comido la rana, que inicialmente es cero, y las vidas. Si la rana conserva alguna vida y las moscas devoradas no han llegado a 4, se ejecuta la rutina de la línea 540, es decir, se pinta el escenario y se juega. En caso de perder una vida, la rutina retornará y se ejecutará la línea 430.

En la línea 430 hay un WEND por lo que itera el bucle WHILE y se vuelven a comprobar las vidas y las moscas.

En caso de haberse comido todas las moscas, se incrementa la dificultad (línea 540) y se pasa de nivel, empezando desde el principio (GOTO a la 390 desde la 520), con las 4 moscas sin comer.

```
360 '---init game
370 lives=3: level=0:score=0:dificultad=30
380 'game logic
390 mosca(0)=1:mosca(1)=1:mosca(2)=1:mosca(3)=1:moscas=0:t=tmax
400 WHILE (lives>0 AND moscas<4)
410 FOR i=0 TO 31:|SETUPSP,i,0,0:NEXT:'reset
420 CLS:GOSUB 540
430 WEND
440 IF moscas<4 THEN 520
450 IF dificultad>15 THEN dificultad=dificultad-5
460 'clean buffer teclado
470 b$=INKEY$:IF b$<>"" THEN 470
480 CLS:PEN 2: LOCATE 12,10:PRINT "CONGRATULATIONS!"
490 LOCATE 11,12:PRINT "READY FOR LEVEL :";(30-dificultad)/5
500 PEN 1: LOCATE 14,20:PRINT "PRESS ANY KEY"
510 b$=INKEY$:IF b$="" THEN 510
520 IF lives>0 THEN 390
530 GOTO 1800:'fin del juego
```

Ahora lo único que queda por entender es lo que hace la rutina ubicada en la línea 540. Esa es la rutina que contiene la construcción de los marcadores, el layout y la “logica del ciclo de juego”.

9 Control de la rana

El control de la rana se hace usando instrucciones de BASIC, pero siempre tratando de reducir el numero de instrucciones a ejecutar. El mecanismo consiste en comprobar si esta pulsada una tecla sin efectuar ninguna comparación. Las siguientes dos instrucciones son equivalentes

- **IF INKEY(67) THEN 1660**
- **IF INKEY(67)>0 THEN 1660**

Pero la segunda es mas lenta que la primera. Teniendo esto en cuenta y que cuando pulsas una tecla la función INKEY(tecla) retorna un cero, la forma mas rápida de leer el teclado es:

```
1620 IF INKEY(67) THEN 1660:  
    <Instrucciones si se pulsa Q y RETURN>  
1660 IF INKEY(69) THEN 1710
```

El listado de control de la rana es el que sigue:

```
1600 '--- keyboard control  
1610 y=PEEK(27497):x=PEEK(27499)  
1620 IF INKEY(67) THEN 1660:'el ajuste de x a multiplo de 4 es necesario  
    por si salto en horizontal dentro de un tronco  
1630 |MOVER,31,-16,0:|COLAY,31,@col%:|MOVER,31,16,0:IF col% THEN RETURN  
1640 POKE 27499,(x+1) AND 252:SOUND 1,638,0,0,1  
1650 |SETUPSP,31,0,233:|SETUPSP,31,15,0:demora=maxdemora:arrastre=0:RETURN  
1660 IF INKEY(69) THEN 1710:'el ajuste de x a multiplo de 4 es ...  
1670 IF y>=180 THEN RETURN  
1680 |MOVER,31,16,0:|COLAY,31,@col%:|MOVER,31,-16,0:IF col% THEN RETURN  
1690 POKE 27499,(x+1) AND 252:SOUND 1,638,0,0,1  
1700 |SETUPSP,31,0,233:|SETUPSP,31,15,1:demora=maxdemora:arrastre=0:RETURN  
1710 IF INKEY(27) THEN 1750:'el ajuste de x a multiplo de 4 es ...  
1720 |MOVER,31,0,4:|COLAY,31,@col%:|MOVER,31,0,-4:IF col% THEN RETURN  
1730 POKE 27499,(x+1) AND 252:SOUND 1,638,0,0,1  
1740 |SETUPSP,31,0,233:|SETUPSP,31,15,2:demora=maxdemora:arrastre=0:RETURN  
1750 IF INKEY(34) THEN RETURN:'el ajuste de x a multiplo de 4 es ...  
1760 |MOVER,31,0,-4:|COLAY,31,@col%:|MOVER,31,0,4:IF col% THEN RETURN  
1770 POKE 27499,(x+1) AND 252:SOUND 1,638,0,0,1  
1780 |SETUPSP,31,0,233:|SETUPSP,31,15,3:demora=maxdemora:arrastre=0:RETURN
```

Ahora vamos a ver que instrucciones debemos ejecutar cuando hemos pulsado una tecla. Concretamente vamos a analizar la pulsación hacia arriba (tecla Q)

```
1610 y=PEEK(27497):x=PEEK(27499)  
1620 IF INKEY(67) THEN 1660:  
1630 |MOVER,31,-16,0:|COLAY,31,@col%:|MOVER,31,16,0:IF col% THEN RETURN  
1640 POKE 27499,(x+1) AND 252:SOUND 1,638,0,0,1  
1650 |SETUPSP,31,0,233:|SETUPSP,31,15,0:demora=maxdemora:arrastre=0:RETURN
```

Con la instruccion **MOVER,31,-16,0** movemos la rana hacia arriba 16 lineas, al lugar que tendrá tras haber terminado el salto hacia arriba. A continuación, comprobamos la colision con el layout mediante la instrucción **COLAY**. Fijate que estamos colisionando la rana en un lugar donde aun no se encuentra impresa. Justo después dejamos la rana donde estaba con **MOVER,31,16,0**

La instrucción **COLAY** la hemos preparado para que deje el resultado de la colision en la variable col. Eso lo hicimos al empezar el juego, en la línea 670. El numero 67 que le pasamos a **COLAY** es un umbral ASCII para que considere que colisionar con los ladrillos del borde de la carretera esta permitido, la rana puede pasar por encima.

670 col%=0: |COLAY,67,31,@col%

A continuación, en la misma línea 1630 examinamos la colisión con el layout, que se encuentra en la variable col y si es mayor que cero es porque ha colisionado y no se puede saltar a ese lugar. Es posible que en ese lugar haya césped, por ejemplo. La rana no puede caminar sobre el césped.

En caso de que la instrucción COLAY retorne un cero en la variable col ejecutaremos la línea 1640. La línea 1640 lo primero que hace es mover a posición multiplo de 4 a la rana en su coordenada X antes de saltar. Esto lo hace por si acaso esta siendo arrastrada por un tronco y se encuentra en una posición que no es multiplo de 4. Decidí que ocurriera esto para que fuese mas fácil cazar las moscas. Para ello ejecuta:

POKE 27499,(x+1) AND 252

Básicamente esto lo que hace es sumar un 1 a la coordenada X y eliminar los 2 bits menos significativos. Por ejemplo:

- Si X= 3 (binario &x11) entonces X = (&x11) AND &x11111100 = &x100 =4
- Si X= 4 entonces X seguirá valiendo 4
- Si X= 5 (binario &x101) entonces X = (&x110) AND &x11111100 = &x100 =4
- Si X= 6 (&x110) entonces X = (&x111) AND &x11111100 = &x100 = 4
- Si X= 7 (&x111) entonces X =(&x1000) and &x11111100 = &x1000 =8
- Si X=8 se seguirá valiendo 8

En resumen, si te acercas a 4 (3,4,5,6) tomas el valor 4. Si te acercas a 8 (7,8,9,10) tomas el valor 8, si te acercas a 12 tomas el valor 12 y asi sucesivamente.

Por ultimo asignamos la ruta de salto a la rana y retornamos:

1650 |SETUPSP,31,0,233: |SETUPSP,31,15,0: demora=maxdemora: arrastre=0: RETURN

Esta línea hace varias cosas. La primera es **|SETUPSP,31,0,233**

El numero 233 en binario es 11101001. Por consiguiente, estamos asignado el estado rutable, sobreescritura, colisionador, movimiento automatico, imprimible, recuerda como van los bits de estado:

7	6	5	4	3	2	1	0
ROUTEALL lo ruta	Sobre- escritura	COLSPALL collider	MOVERALL lo mueve	AUTOALL lo mueve	ANIMALL lo anima	COLSP collided	PRINTSPALL lo imprime

Fig. 30 Flags en el byte de estado

Después ejecuta **|SETUPSP,31,15,0**

Esta instrucción asocia la ruta 0, que es la de salto hacia arriba, de acuerdo con nuestro fichero de rutas **routes_mygame.asm**

A continuación tienes la definición de la ruta 0:

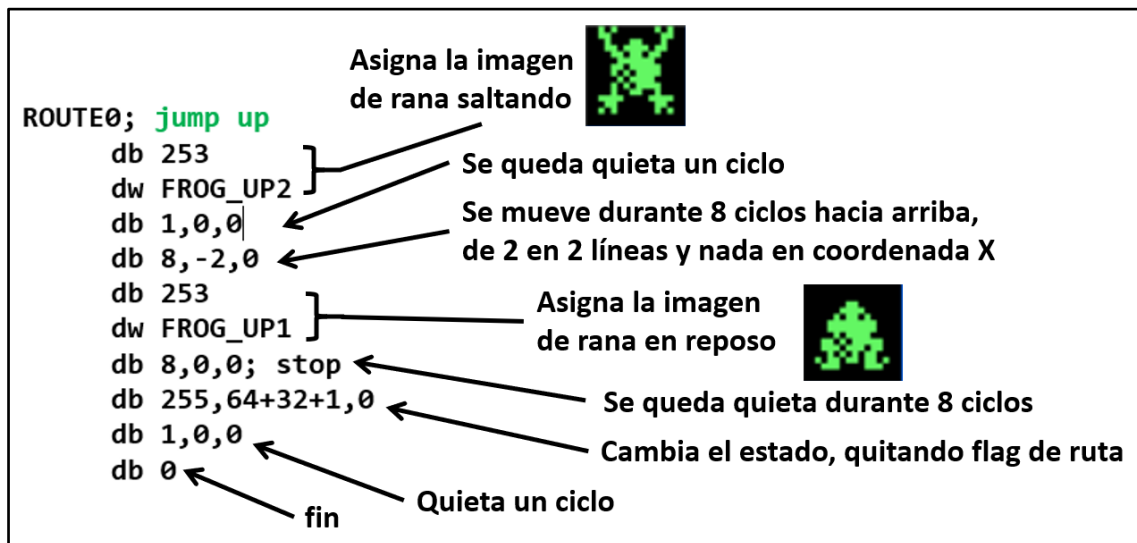


Fig. 31 ruta de salto hacia arriba de la rana

Por ultimo se establecen dos variables: **demora=maxdemora: arrastre=0**

La variable “demora” es el tiempo que ha de pasar hasta que nos sea permitido volver a saltar, ya que durante los próximos frames va a estar saltando y no se puede interrumpir un salto para cambiar de dirección. En cada ciclo de juego, demora se reduce en 1 mediante una instrucción en el bucle principal (línea 1120)

Con la asignacion **arrastre=0** lo que estamos haciendo es permitir que si al caer la rana ha caído en un tronco o similar (en la rutina de colisión, en línea 1340) , se le asociará una de las rutas de arrastre, la cual tendrá la dirección y velocidad del elemento (tronco, tortugas o nenúfares) en el que se haya subido. Una vez que se le asigne una ruta de arrastre en la rutina de colision, la variable arrastre se pondrá a 1.

Vamos a ver la rutina de colision para entender mejor el mecanismo de las rutas de arrastre

10 Rutina de colisión

La rutina de colisión se invoca cuando el `collider<32`. Recordemos que el comando `COLSPALL` retorna 32 en la variable `collider` cuando no hay ninguna colisión

Dentro del ciclo de juego la rutina de colisión se invoca en la línea 1320

1320 IF collider<32 THEN 1340

La rutina de colisión debe tener en cuenta todos los casos, que son:

- Colisión con enemigo (coche): debe morir
- Colisión con mosca: debe comérsela y volver abajo del todo
- Colisión con tronco, hojas o tortugas: debe asignarse una ruta de “arrastre” a la rana en función de la velocidad y dirección que lleve el elemento al que se ha subido
- Colisión con río: en el caso de que colisionemos con uno de los 4 ríos, debemos morir, siempre que no nos encontremos en mitad de un salto.
- Colisión con muros invisibles: debe morir

La rutina no es muy difícil de entender, pero lo más importante es tener en cuenta que los ríos tienen un identificador de sprite inferior a troncos, hojas y tortugas. Eso significa que aunque se solapen río y tronco, la rana colisiona con el sprite mayor. Siempre con el mayor. Gracias a esto, cuando colisionamos con tronco detectamos el tronco y no el río que está “debajo”

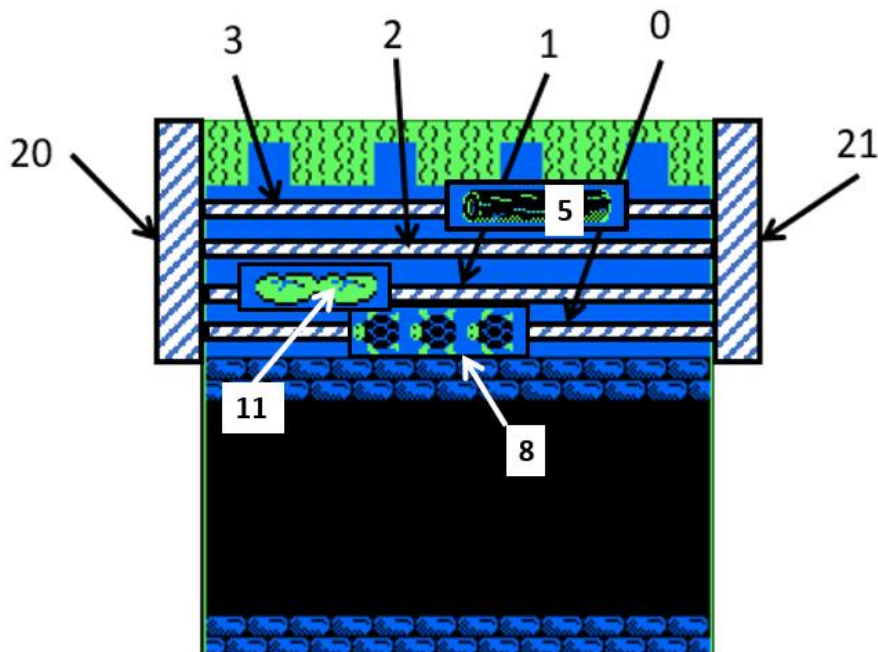


Fig. 32 Identificadores escogidos teniendo en cuenta el orden de colisión de COLSPALL

Veamos el listado. Cuando veas un goto 1210 es un salto para volver al ciclo de juego ya que como hemos saltado con `GOTO` debemos retornar con goto, no podemos volver con `return`. Podríamos haber entrado en esta rutina con `GOSUB` y quizás sería más fácil de leer, pero funciona igual. El único `RETURN` que tiene la rutina es para volver al

bucle principal cuando te matan, y es entonces cuando se reconstruye todo el layout y se comienza de nuevo

```
1340 '--- collision routine
1350 IF collided>11 THEN 1440
1370 IF collided<4 THEN if demora<2 then 1510 else 1210:'rio

1371 'hemos subido a un tronco.
1380 IF (demora+arrastre) THEN 1210
1390 IF collided<6 THEN |SETUPSP,31,0,233:|SETUPSP,31,15,14:
arrastre=1:GOTO 1210:'tronco R slow
1400 IF collided<8 THEN |SETUPSP,31,0,233:|SETUPSP,31,15,13:
arrastre=1:GOTO 1210:'tronco L fast
1410 IF collided<10 THEN |SETUPSP,31,0,233:|SETUPSP,31,15,16:
arrastre=1:GOTO 1210:'turtles L slow
1420 IF collided<12 THEN |SETUPSP,31,0,233:|SETUPSP,31,15,18:
arrastre=1:GOTO 1210:'hojas R fast
1430 GOTO 1210

1440 'colision con mosca o con enemigo
1450 IF collided<22 THEN 1520:'check mosca
1460 |SETUPSP,collided,9,watermosca:
|PRINTSP,collided:|SETUPSP,collided,0,0
1470 SOUND 1,638,30,15,0,1: mosca(collided-22)=0
1480 score=score+10:LOCATE 3,10:PEN 2:PRINT score
1490 WHILE demora :|AUTOALL,1:|PRINTSPALL: demora=demora-
1:WEND:'termina salto
1500 moscas=moscas+1:IF moscas=4 THEN RETURN ELSE GOTO 1180

1510 'colision enemigo
1520 SOUND 1,142,100,15,0,3:FOR i=1 TO 20:BORDER 7:CALL
&BD19:|PRINTSPALL,0,0,0,0,0:BORDER 0:CALL &BD19:NEXT
1530 lives=lives-1:RETURN
```

Una vez que se le asigne una ruta de arrastre a la rana en la rutina de colision, la variable arrastre se pondrá a 1. Con ello aunque en cada ciclo la rana vuelva a colisionar con el tronco en el que está subido, no se le reasigna una ruta de arrastre, porque ya la tiene.

Si nos fijamos en la colisión con el rio, vemos una comprobación interesante

```
IF collided<4 THEN if demora<2 then 1510 else 1210
```

En el caso de que colisionemos con uno de los 4 rios, debemos morir, siempre que no nos encontremos en mitad de un salto, por ello se exige que demora sea cero o 1, ya que si es 1 es que justo vamos a terminar un salto y antes de que demos otro, debemos dar muerte a la rana.

11 Lógica del ciclo de juego

Por increíble que parezca, lo mas sencillo de todo es la lógica del ciclo de juego

Preparación al ciclo de juego. Con estas dos líneas se configuran los comandos PRINTSPALL, COLSP y COLSPALL

```
1190 |PRINTSPALL,0,0,0,0:|COLSP,34,3,3:|COLSPALL,@collider%,@collided%
1200 |COLSP,32,0,25:'mas alla del 25 no hay colisionables
```

Y ahora si, entramos en el ciclo de juego. Todo el ciclo lo tienes aquí. No hay mas:

```
1201 '--- game cycle
1210 ciclo=ciclo+1: |AUTOALL,1:|PRINTSPALL:|COLSPALL
1220 IF demora THEN demora=demora-1
1230 IF demora GOTO 1260
1240 GOSUB 1600
1260 IF ciclo MOD dificultad THEN 1280
1270 GOSUB 1550
1280 IF ciclo AND 31 THEN 1300
1290 t=t-1:LOCATE 35,6:PRINT t: IF t=0 THEN t=tmax: GOTO 1520
1300 IF PEEK(27147)<90 THEN 1320
1310 |LOCATESP,8,72,60:|LOCATESP,9,72,90:'recoloca turtle
1320 IF collider<32 THEN 1340
1330 GOTO 1210
```

En la 1210 se incrementa en 1 la variable ciclo.

A continuación, se invoca el comando AUTOALL, 1, el cual enruta a los sprites actualizando su posición. Despues se invoca a PRINTSPALL que imprime todos los sprites de golpe y por ultimo se invoca a COLSPALL , que calcula las colisiones entre sprites y deja el resultado en las variables collider y collided, tal como fue configurado el comando en la línea 1190

A continuación, comprobamos si la variable demora es >0 y de ser asi, la decrementamos. Esa variable es >0 cuando la rana esta en mitad de un salto.

En la línea 1230 se comprueba si la demora es cero, si es cero, saltamos a la rutina de control de la rana por teclado, que se encuentra en la línea 1600

```
1220 IF demora THEN demora=demora-1
1230 IF demora GOTO 1260
1240 GOSUB 1600
```

Después se realiza la operacion

```
1260 IF ciclo MOD dificultad THEN 1280
1270 GOSUB 1550
```

Esto lo que hace es saltar a la 1280 casi todas las veces, salvo cuando ciclo MOD dificultad es cero, en cuyo caso ejecuta el GOSUB a la 1550, la cual crea un nuevo coche en pantalla.

Como dificultad empieza en 31, una de cada 32 ciclos crearemos un coche. En los siguientes niveles la dificultad llega a valer 15 por lo que se llegan a crear el doble de coches en el mismo tiempo.

A continuación, tenemos un par de líneas que se encargan de ir reduciendo el tiempo disponible para nuestra pobre rana

```
1280 IF ciclo AND 31 THEN 1300
1290 t=t-1:LOCATE 35,6:PRINT t: IF t=0 THEN t=tmax: GOTO 1520
```

Después tenemos la recolocación de las tortugas, que se hace cada cierto tiempo, cuando la coordenada x del sprite 9 es >90, que es debido a que la x se hace negativa y debido a la representación binaria de los números, estos son >=128 cuando son evaluados como si fuesen positivos.

La coordenada X ocupa 2 bytes, pero solo evaluamos el primero, pues para saber si ha pasado a negativo con comprobar que el primero (el menor) es negativo es suficiente. Recolocamos las dos tortugas a la vez

```
1300 IF PEEK(27147)<90 THEN 1320
1310 |LOCATE SP,8,72,60:|LOCATE SP,9,72,90: 'recoloca turtle
```

Lo que ocurre con la coordenada X del sprite 9 es que cuando llega a cero, se transforma en -1 y puesto que se trata de una coordenada de 16 bit, hablamos del número 65535, es decir -1 en binario es 65535. Eso son 2 bytes con valor 255 cada uno. Solo compruebo el menos significativo, el cual empieza valiendo 90 y llega hasta cero en el borde de la pantalla y justo después pasa a -1 (a 255 en realidad). Por lo tanto cuando dicho byte se hace 255 es mayor que 90 y acaba de cruzar el borde izquierdo de la pantalla. Es en ese momento cuando ejecutamos la 1310 para recolocar las tortugas (sprites 8 y 9).

Te preguntarás porque es necesario recolocar las tortugas y no hace falta recolocar troncos ni hojas ni coches. Es debido a como están contruidas las rutas. En los troncos y hojas es la propia ruta la que recoloca al sprite y en el caso de los coches, simplemente se crean periódicamente, reemplazando coches que ya se fueron lejos de la pantalla por otros nuevos que toman su mismo identificador de sprite. En el capítulo de rutas lo detallaré más

12 Rutas

12.1 Tipología de posibles Rutas

Hay varios tipos de rutas posibles:

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

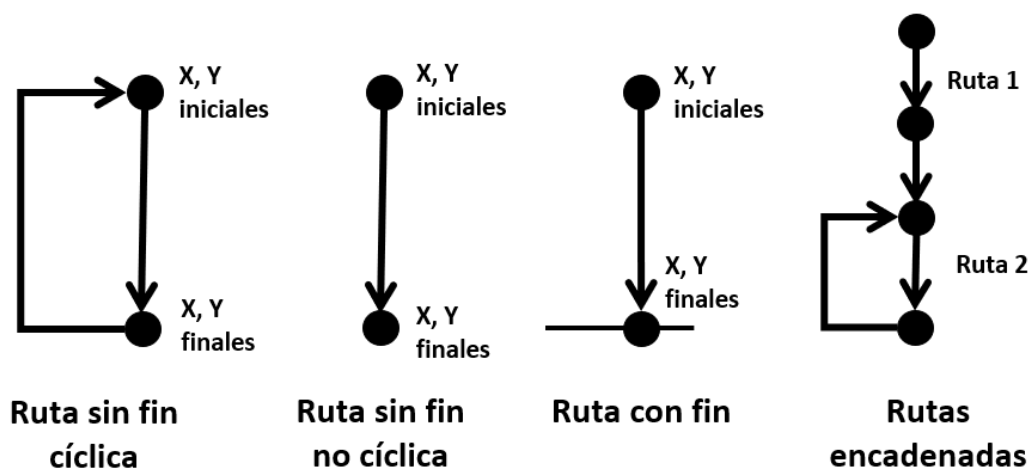


Fig. 33 Tipos de rutas

12.2 Rutas de vehículos

Las rutas de los vehículos son todas muy similares. Se diferencian en la dirección del movimiento, la imagen asociada y la velocidad. Por ejemplo, aquí tenemos la ruta del autobús a la derecha (lento) y la de la moto corriendo hacia a la izquierda (muy rápida). Como puedes comprobar la ruta del autobús es una ruta no cíclica sin fin y la ruta de la moto es una ruta con fin.

Los autobuses usan una ruta no cíclica pero periódicamente se crean nuevos vehículos que se reubican y reutilizan los sprites de vehículos que ya están muy lejos del área de juego y que por lo tanto, no se ven.

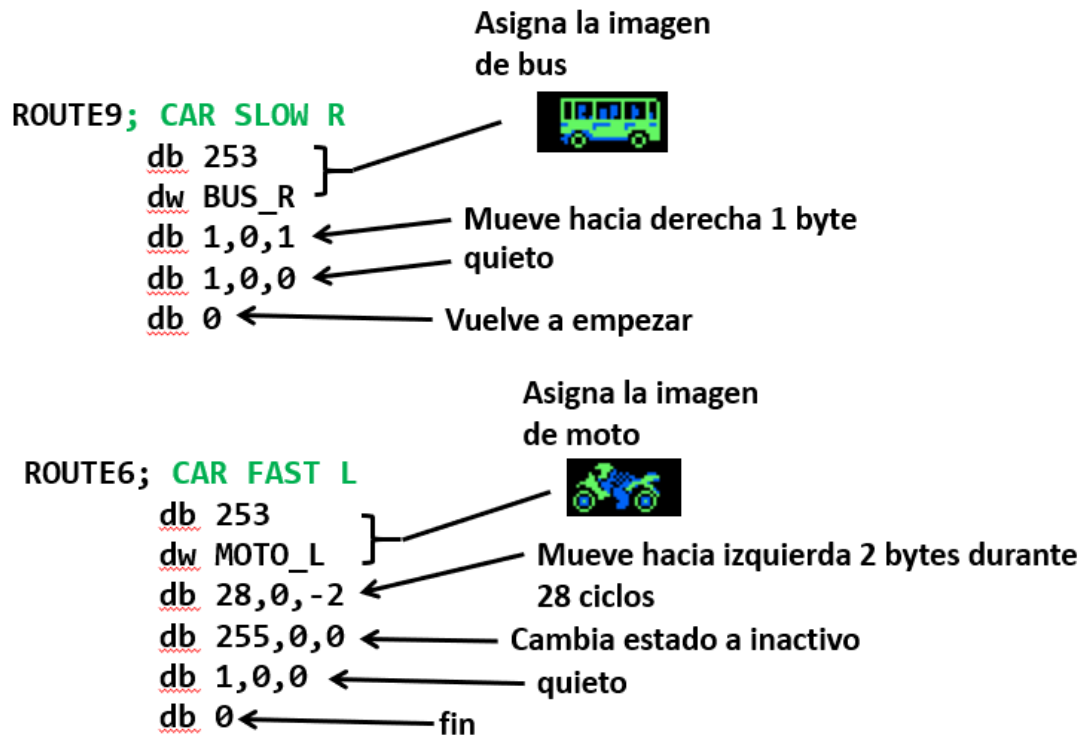


Fig. 34 rutas de vehiculos

Fíjate en la gran diferencia entre ambas rutas. La del autobús define un patron de movimiento con un paso en el que se desplaza y otro paso en el que no se desplaza y vuelta a empezar. Eso significa que no acaba nunca (ruta no cíclica sin fin). Sin embargo, como el autobús desaparece de la pantalla deja de ser peligroso (aunque sigue moviéndose fuera del área visible) y al cabo de algunos ciclos la lógica reasigna el sprite a un nuevo vehiculo.

En el caso de la moto, como todos los pasos son iguales, se puede poner un 28 en el numero de pasos que vamos a dar y tras los 28 pasos se desactiva el sprite, no gastando potencia de CPU en moverlo, aunque se encuentre fuera del área de pantalla. Es por lo tanto, una ruta con fin.

12.3 Rutas de tortugas

La ruta de las tortugas se parece a la del autobús, se trata de un patron de repetición compuesto por dos pasos en los que en uno se mueve y en otro están quietas, y dos cambios de imagen para que la tortuga parezca que mueva las aletas. Es una ruta no cíclica sin fin.

13 A jugar y a programar!!

Una vez entendido como funciona frogger, posiblemente te te ocurra como mejorarlo, como hacer un juego con mas opciones, nuevos niveles, etc. El primer paso para crear tus programas, tus ideas e innovaciones, es disfrutar jugando. De ahí nacerán las ideas.

Para crear tus propios juegos te daré un par de consejos:

1. No escatimes tiempo en la creación de graficos. Tus sprites si no bonitos le darán un aspecto muchísimo mejor al juego y lo harán mucho mas atractivo.
2. No seas muy ambicioso, piensa un proyecto grande...pero trata de hacer algo pequeño, que puedas materializar con relativamente poco esfuerzo. Es importante que veas resultados para no desanimarte y continuar tu proyecto hasta el final

Hasta pronto y felices juegos!

Jose Javier.