8 BITS DE PODER

In your AMSTRAD CPC

"A Guide Developer 8 bit in the XXI Century"

V25

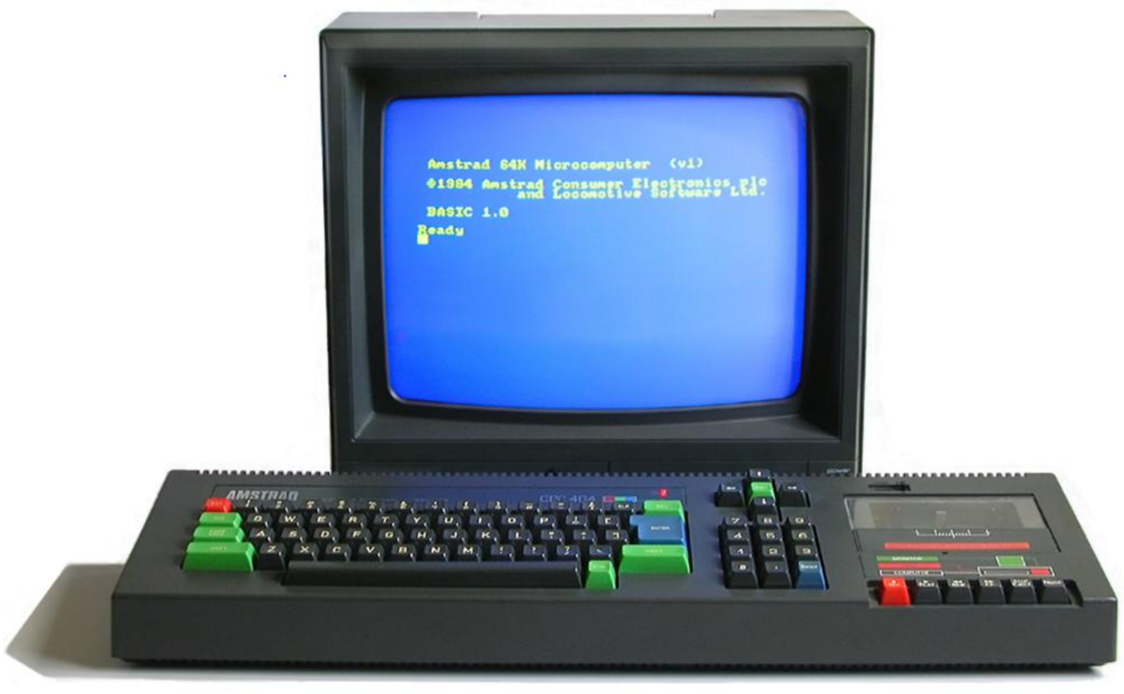
Jose Javier García Aranda

INDEX

1 Why programming in 2016 a machine of 1984?

Because constraints are not a problem but a source of inspiration.

Limitations, whether of a machine or a human being, or in general any available resource stimulate our imagination to overcome them. The AMSTRAD, a machine based on the 1984 Z80 microprocessor, has a small memory (64KB) and reduced processing capacity, but only when compared with today's computers. This machine is actually a million times faster than Alan Turing built to decipher the enigma machine messages in 1944
Like all computers of the 80s, the AMSTRAD CPC ripped in less than a second, with the BASIC interpreter ready to receive user commands, with the BASIC language with which programmers learned and made their early developments. The BASIC AMSTRAD was particularly fast compared to its competitors. And it was aesthetically very attractive computer!



```
Amstrad 64K Microcomputer  (v1)
©1984 Amstrad Consumer Electronics plc
         and Locomotive Software Ltd.

BASIC 1.0

Ready
```

Fig. 1. The mythical AMSTRAD model CPC464

As for the Z80 is not even able to multiply (in BASIC can multiply but that is based on an internal program that implements the multiplication by addition or displacement records), only can do addition, subtraction and logical operations. Despite this was the best 8-bit CPU and only consisting of 8500 transistors, unlike other processors such as the M68000 whose name comes from having precisely it 68000 transistors.

CPU	Number of transistors	MIPS (millions of instructions per second)	Computers and consoles that incorporate
6502	3,500	0.43 @ 1Mhz	COMMODORE 64, NES, ATARI 800 ...
Z80	8,500	0.58 @ 4MHz	AMSTRAD, ColecoVision, SPECTRUM, MSX ...
68000	68,000	2.188 @ 12.5 Mhz	AMIGA, SINCLAIR QL, ATARI ST ...
Intel 386DX	275,000	2.1 @ 16Mhz	PC
Intel 486DX	1,180,000	11 @ 33 Mhz	PC
Pentium	3,100,000	188 @ 100Mhz	PC
ARM1176		4744 @ 1Ghz (1186 per core)	Raspberry Pi 2, Nintendo 3DS, samsung galaxy, ...
Intel i7	2,600,000,000	238310 @ 3 GHz (nearly 500,000 times faster than a Z80!)	PC

Table 1Comparison of MIPS

This fact makes programming it extremely interesting and challenging in order to achieve satisfactory results. All our programming should be aimed at reducing computational complexity (operations) space (memory) and temporary, forcing us to invent tricks, ruses, algorithms, etc., and making programming an exciting adventure. Therefore, the programming of low processing capacity machines is not subject to fashions or conditioned by the evolution of technology, it is a timeless concept.

All code in this book, including the library, is available at GitHub "8BP" project at this URL:

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

There is also a blog (in Spanish) with lot of information at:

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

And a YouTube channel at:

https://www.youtube.com/channel/UCThvesOT-jLU_s8a5ZOjBFA

2 8BP functions and memory usage

The 8BP library is not a "game engine". It is something between a simple extension of BASIC commands and a game engine.
Game engines as the maker game, the AGD (Arcade Game Designer), the Unity, and many others, limit to some extent the imagination of the programmer, forcing him to use a certain structures, programming in limited scripting language logic an enemy, to define and link game screens, etc

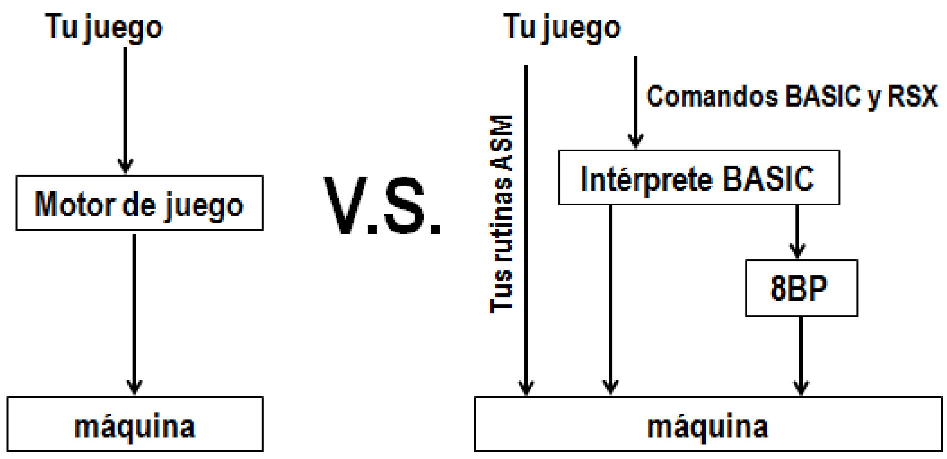


Fig. 2 Game engines versus 8BP

The 8BP library is different. It is a library of run quickly what the BASIC can not do able. Things like print sprites at full speed, or move banks screen stars, are things that BASIC can not do and 8BP get it. And it takes only 6 KB

BASIC is an interpreted language. That means that every time the computer runs a program line must first verify that is a valid command, comparing the string of command with all valid commands chains. You must then validate the expression syntactically, command parameters and even the allowable ranges for the values of these parameters. In addition reads parameters in text (ASCII) and must convert them to digital data. All this work is finished, proceed with the execution. Well, all that work done in each instruction sets is a compilation of a program interpreted as written in BASIC program.

Endowing the BASIC commands provided by 8BP, it is possible to play professional quality, since the logic of the game programes can run on BASIC while intensive operations in the CPU usage as printing on screen or detect collisions between sprites, etc. are carried out in machine code for Liberia.

However, not everything is easy and no problems. Although 8BP library will provide great features in video games, you should use it with caution because each command that you invoke pierce the layer of parsing BASIC, before reaching the underworld machine code where the function is located, so the performance will never be optimal. You should be smart and save instructions, measuring execution times of instructions and pieces of your program and thinking strategies to save runtime. an adventure of wit and fun. Here you will learn how to do it and even'll introduce a technique I have called "massive logic" that lets you accelerate your games to limits that perhaps thought were impossible.

In addition to the library, you have at your disposal a simple but complete sprite editor and graphics and a number of great tools that allow you to enjoy in the XXI century adventure program a microcomputer.

2.1 Functions 8BP

After loading the library with the command: LOAD "8BP.BIN" and invoke from the _INSTALL_RSX BASIC function (defined in machine code) using the BASIC command:

CALL & 6b78

Can use the following commands, you will learn to use this book (note that a vertical bar appears at the top of each as "extensions" of BASIC):

| ANIMA, #	changes the frame of a sprite according to their sequence
| animall	changes the frame of animation sprites flag enabled
| AUTO, #	automatic movement of a sprite according to its speed
| AUTOALL	movement of all active flag sprites with automatic mov
| Colay, umbral_ascii, #, @ collision	detects the collision with the layout and returns 1 if there is collision
| COLSP, #, @ id	returns with first sprite collides #
| COLSPALL, who% @, @ whoyou%	Who returns and who has collided collided
| LAYOUT, y, x, @ string	prints a 8x8 layout images and fills in layout map
| LOCATESP, #, y, x	change the coordinates of a sprite (without printing)
| MAP2SP, y, x	creates sprites to paint the world in games with scroll
| MOVE, #, dy, dx	relative movement of a single sprite
| MOVERALL, dy, dx	relative movement of all sprites with relative motion flag active
| MUSIC, song, speed	He starts playing a tune to the desired speed
| MUSICOFF	stops ringing melody
| PEEK, dir, @variable%	reads a 16bit data (can be negative) direction
| POKE, dir, value	introduces a 16bit data (which may be negative) in a memory address
| PRINTSP, #, y, x	printing a single sprite (# is the number) regardless of status byte
| PRINTSPALL, order, anima, sync	prints all sprites with flag active print
| ROUTEALL	Modifies the speed of the sprites map with flag
| SetLimits, xmin, xmax, ymin, ymax	defines the game window, which is clippling
| SETUPSP, #, param_number, value	modifies a parameter of a sprite
| SETUPSQ, #, ADR0, ADR1, ..., adr7	creates a sequence of animation
| STARS, initstar, num, color, dy, dx	scroll a set of stars

Table 2 8BP commands available in the library

Additionally you have an experimental command:

| RETROTIME, date

This command allows you transform your CPC in a time machine, just by entering the date desired destination. The only limitation of the command is to enter a date on or after the birth Amstrad CPC, April 1984

| RETROTIME, "04/01/1984"

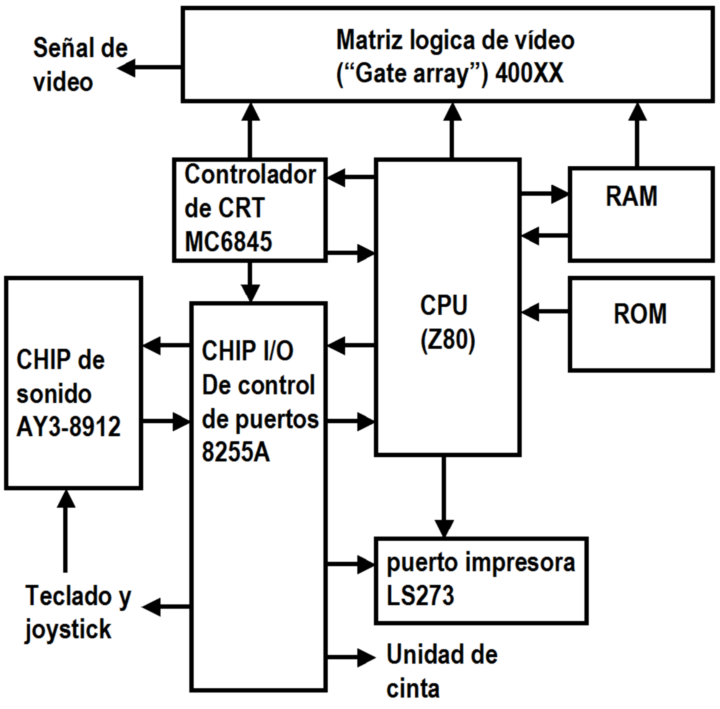
Please use this feature with caution. You could create a time paradox and destroy the world.
But for now you can have some skepticism about what you can do with the 8BP library, you will soon discover that using this library along with techniques advanced programming you'll learn in this book will allow you to make professional games in BASIC, something that perhaps thought impossible.

Important note for the programmer:

The 8BP library is optimized for very fast. That is why not check you have correctly placed the parameters of each command, or having an appropriate value. If a parameter is wrong because it is very possible that the computer crashes when you run the command. Check these things takes time and execution time is a resource that can not be wasted, even a millisecond.

2.2 Architecture AMSTRAD CPC

This section is useful to further understand how memory uses the library 8BP.
Amstrad is a computer based on the Z80 microprocessor, running at 4MHz. As shown in the diagram of architecture, both the CPU and the logical matrix video (called "gate array") access the RAM, so to "take turns", the memory accesses from the CPU are delayed resulting in an effective rate 3.3Mhz. This is still enough power.



The video memory, what we see on screen, is part of 64KB RAM, 16KB particular are located at the top of memory. The memory is numbered from 0 to 65535 bytes. Well, the 16KB of between 49152 and 65535 management is the video memory. In hexadecimal it is shown as C000 to FFFF.

Fig. 3Architecture AMSTRAD

The video RAM is accessed by the gate array 50 times per second to send an image to the screen. On older computers (such as zx81) this task was entrusted to the processor, taking away even more power.

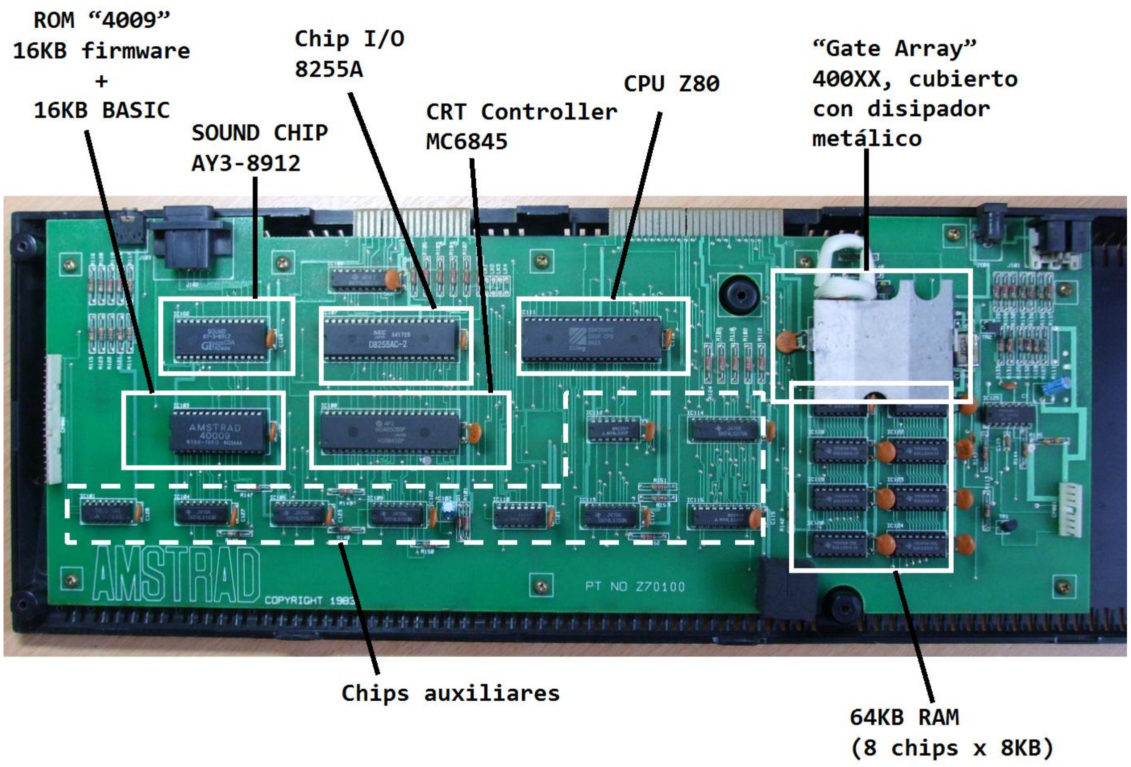


Fig. 4 Identification of components on the board

The Z80 has a 16bit address bus, so it is not able to address more than 64KB. However the Amstrad has 64kB 32kB RAM and ROM. To route them power, AMSTRAD is able to "switch" between some banks and others, so that, for example if it invokes a BASIC command switches to bank ROM where the BASIC interpreter, which is overlapped with the stored 16KB screen. This mechanism is simple and effective.

In addition to the ROM containing the BASIC 16KB interpreter located in the high memory, there are other 16KB ROM located in low memory, where the routines firmware (what could be considered the operating system of the machine). Total (BASIC interpreter and firmware) add 32KB

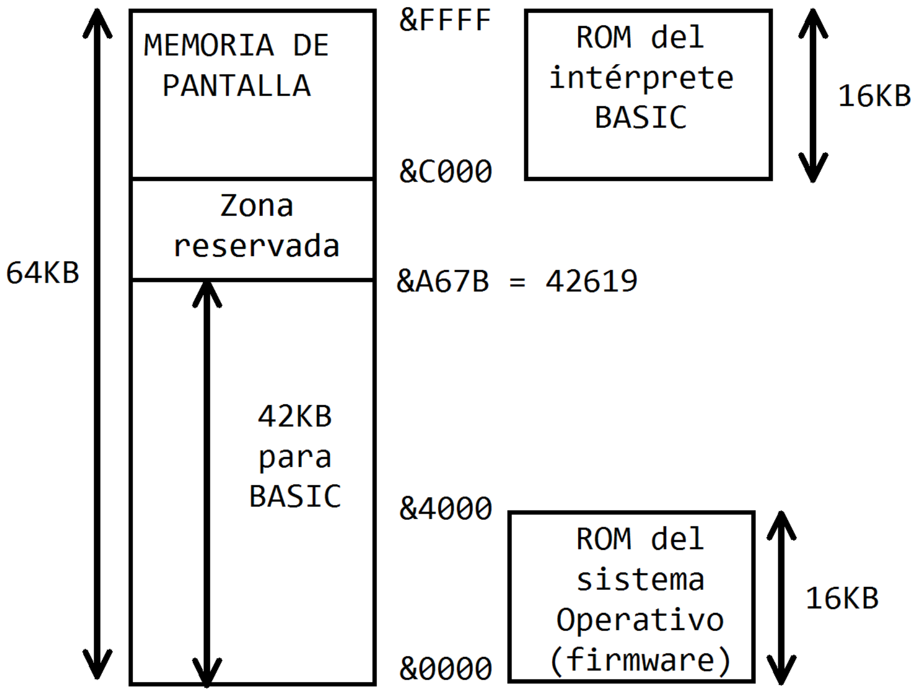
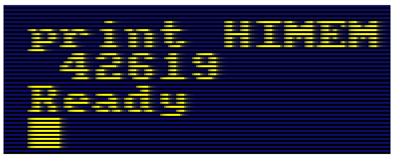


Fig. 5 AMSTRAD memory

As seen in the map memory, 64KB of RAM, 16KB (from & C000 to & FFFF) are the video memory. BASIC programs can take from the position & 40 (direction 64) to 42619, because there beyond system variables. That is, it has a 42KB for BASIC, as we can see when printing variable HIMEM system (short for "High Memory").



```
print HIMEM
 42619
Ready
```

Fig. 6 HIMEM variable system

The operation of the BASIC takes into account the storage of the program in increasing direction from the position & 40, while once running, the variables are declared must occupy space to store the values taken and since they can not occupy the same area where the program is stored, simply they begin to store in the highest possible direction, the 42619 and as more variables are used decreasing memory addresses are consumed.
AMSTRAD each number in the integer variable occupy 2 bytes of memory.

2.3 Using memory 8BP

The 8BP library is loaded in the high memory area available. It is important to understand how BASIC works in order to use the library.
The text of a program written in BASIC is stored from address & 40 but once it starts running, the values that are taking program variables are stored occupying decreasing positions from the 42619, so that if a program is large , could reach "hit" with the text of the program itself, destroying part of it. This will not normally happen, do not worry.

The library is loaded from address 27000, allocating memory functions, music player, songs and drawings, as shown in the following diagram

```
                         AMSTRAD CPC464 memory map of 8BP
; AMSTRAD CPC464 memory map of 8BP
;
;  &FFFF + ----------
;        | screen(16000 Bytes) + 8 hidden screen segments each 48bytes
;  &C000 + ----------
;        | system (definible symbols, etc.)
;  42619 + ----------
;        | bank of 40 stars (from 42540 to 42619 = 80bytes)
;  42540 + ----------
;        | map layout of characters (25x20 = 500 bytes)
;  42040 + ----------
;        | sprites (up to drawings 8.5KB)
;        |
;        + ----------
;        | Route definitions (variable length each)
;        + ---------
;        | animation sequences 8 frames (16 bytes each)
;        | and groups of animation sequences (macrosequences)
;  33500 + ----------
;        | songs (1.25kB to music)
;        |
;  32250 + ----------
;        | 8BP routines (5860 bytes)
;        | Here are all the routines and table sprites
;        | It includes music player "wyz"
;  26390 + ----------
;        | world map (389 bytes)
;  26000 + ----------
;        | | BASIC variables
;        | V
;        |
;        | ^ BASIC (program text)
;        | |
;      0 + ----------
```

Fig. 7 Memory using 8BP

Before you even load the library, you must run the command

MEMORY 25999

To limit the space occupied by the BASIC. Thus, BASIC variables begin to take up space from 25,999 down direction running. Your programs can be almost 26KB of memory, although as the variables take up space it is normal that the maximum size of your program is less. However we are talking about a very respectable amount of memory. It will cost a lot to make a game of 26KB, I assure you.

Note to programmers earlier versions of 8BP:
In previous versions, the occupied library 8BP 5250 bytes instead of 6250 bytes so the MEMORY command must be invoked with 26999, having 27KB for your program

instead of 26KB. Since the V24 version that incorporates a mechanism multidirectional scroll and world map game, 1KB consume more, so the MEMORY command should now point to 25999. You may find demos or examples using earlier versions, yet they are compatible with the V24, because a MEMORY 26999 simply "crush" the new features of the v24, but everything else still work perfectly.

3 Tools needed

WinAPE: Emulator for Windows OS editor to edit and test your BASIC program. And also to assemble graphics and music

Spedit("Simple Sprite Editor") BASIC tool to edit your graphics. The result of spedit is assembler code that is sent to the printer Amstrad CPC. Running the tool within WinAPE, the printer is redirected to a text file so that your graphics are stored in a txt file. This tool has been created to complement the 8BP library.

Wyztracker: To compose music under windows. The program can play melodies composed by Wyztracker is the Wyzplayer, which is integrated within 8BP. After assembling the music you can make it sound with a simple command | MUSIC

Libreria 8BP: Install new accessible from BASIC commands for your program. As you will see, this will be the "heart" to move the machinery that you build.

CPCDiskXPIt lets you record a 3.5 "diskette which you can then insert into your CPC6128 if you have a cable to connect a floppy drive. If you want to make an audiotape for CPC464 not need this tool

optionally:
RGAS:(Retro Game Asset Studio) Powerful sprite editor, evolved from a tool AMSprite, created by Lachlan Keown. This sprite editor supports 8BP and runs under Windows. When you outgrow Spedit is, this may be the best option.

fabacom:compiler executable within the Amstrad CPC emulator 6128 or from the BASIC WinAPE to compile your program and make it run faster. It supports calls to commands 8BP library. However it is not recommended for several reasons:

- your program will take much more fabacom need for additional 10KB for their libraries, and besides, once you compile your program continues to occupy the same, so a 10KB program becomes one of 20KB.
- There are documented some incompatibility issues this compiler with some BASIC instructions.
- Moreover, as you'll see throughout this book, you can achieve a very high speed without recompiling.

4 Steps you must take to make a game

4.1 Directory structure of your project

The best practice schedule when your game is you structure the different files on 7 folders, depending on the type of file in question.
It is perfectly possible to put everything in the same directory and work without folders, but is more "clean" do as I will present below

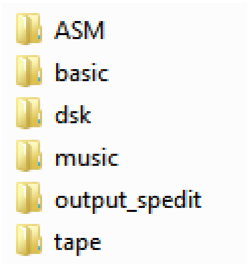


Fig. 8 directory structure

- **ASM:** You shall put text files written in esnamblador (.asm), as is the library 8BP own, sprites generated with sprite editor SPEDIT, and some auxiliary files.

- **BASIC:** You shall put your game and utilities like SPEDIT and charger (Loader).

- **DSK:**You bring in the file .dsk ready to run in a amstrad cpc. Inside you will locate five files which are discussed below

- **Music:**WYZtracker with the music sequencer, you can create your songs and store them in .wyz format in this directory. Once the "exportes" an asm file that will have to be saved in the ASM file and a binary file that also will store in the ASM folder (also can leave in this folder to be generated such that referencies properly in the make_musica.asm the ASM file directory)

- **Output_spedit:**in this folder you can store the text file generated spedit. SPEDIT it does is send to the printer assembly sprites WinAPE emulator format and can pick up the printer output to a file Amstrad. Here we will place

- **Tape:** You can store the .wav if you want to make a tape to load into the Amstrad CPC464

4.2 Your game 5 files

In this section we will look at the steps you need to give. It is not sequential, you can whet your graphics as programs and the same goes for music. Do not worry if you do not understand precisely now every step. Throughout the book you go understanding exactly what they mean precisely. And in the end you have an appendix with detailed information in this regard.

What moment must understand it is that your game should be composed of 5 files: 3 binary files and 2 files BASIC

Binary files are:

- 8BP library (it is a binary file), including sprite attribute table
- binary file of music with the melodies of your game
- binary image file sprites, including animation sequence table

And two BASIC files

- Charger (load library, music and sprites and finally your game). If you also want to make a splash screen while the game is loading is displayed, it will be the first thing you charge this charger
- BASIC program (your game)

To make these 5 files you must take these steps

STEP 1
Edit graphics with SPEDIT
Assemble graphics WinAPE
Save the graphics with SAVE "sprites.bin" b, 33500, <size> command

STEP 2
Edit music with WYZtracker
Assemble music with WinAPE. The melodies are assembled one after another, so that each begin at a different memory address that depend on the size they occupy.
Save music with SAVE "music.bin" b, 32250, 1250 command

STEP 3
8BP reassembling the library, so that part of the library you select the tunes (the player wyz) can know that memory addresses are assembled (there are more dependencies but this is one of them). Once re-assembled, you'll have to save it with the command
SAVE "8BP.LIB" b, 26000, 6250
This will be a specific version of the library for your game. For example, the command | MUSIC, 3.5 will sound the melody number 3 you yourself have made. The melody number 3 can be completely different in another game.

STEP 4
Load all with a loader, you should do in BASIC. For example:

```
10 MEMORY 25999
20 LOAD "! 8bp.lib"
30 LOAD "! Music.bin"
40 LOAD "! Sprites.bin"
50 RUN! "Tujuego.bas"
```

STEP 5
Schedule your game, which must first run the call to install the RSX commands, ie CALL & 6b78.

Your game you can program using the editor WinAPE, much more versatile than the AMSTRAD editor to edit and serves both assembler (.asm) to edit BASIC (.bas). WinAPE editor is sensitive to keywords and automatically changes color, facilitating the work of programming. After writing a BASIC program to copy / paste into the window WinAPE CPC. To make it faster can activate the "High Speed" WinAPE option during bonding, thereby the process will be immediate.
Optionally you can compile your game with a tool called fabacom and use the compiled version, but not necessary, 8BP your games and work very fast.

STEP 6
Create a tape or a disk with your game

4.3 Create a disk or tape with your game

4.3.1 Make a record

To create a new album from WinAPE do

File-> drive A-> new blank disk

This results in a file management window will appear for you to give name to new file .dsk

Once created you can now save files with the SAVE command. To delete a file using the command "| ERA" (short for ERASE), which only exists in CPC 6128 as part of the "AMSDOS" operating system (this in there because CPC464 worked with cassette tape)

| ERA, "game *."

and deleted

To load the game you need a charger that charges one by one the necessary files. Something like:

```
10 MEMORY 25999
20 LOAD "! 8bp.lib"
30 LOAD "! Music.bin"
40 LOAD "! Sprites.bin"
50 RUN! "Tujuego.bas"
```

To save each file you must use the SAVE command with the required parameters, for example:

```
SAVE "LOADER.BAS"
SAVE "8BP.LIB" b, 27000, 6250
SAVE "MUSIC.BIN" b, 32250, 1500
SAVE "SPRITES.BIN" b, 33500, 8500
SAVE "tujuego.BAS"
```

If you want to record the .dsk on a 3.5 "diskette and connect it to an external floppy drive your AMSTRAD CPC 6128, you need the CPCDiskXP, very easy to use program. From a .dsk you can record a floppy disk 3.5 "double density (do not forget to plug the hole on the diskette to" trick "the PC)

4.3.2 Make a tape

The most important thing when creating a tape is to keep it files in the order in which they will be loaded by the computer. A tape is not like a disk you can load any stored file, but files are one after another, so you should be especially careful at this point.

If your charger game is as follows:

```
10 MEMORY 25999
20 LOAD "! 8bp.lib"
30 LOAD "! Music.bin"
40 LOAD "! Sprites.bin"
50 RUN! "Tujuego.bas"
```

Then you must first save the charger (say called "loader.bas"), then "8BP.LIB", then "MUSIC.BIN" then "SPRITES.BIN" file and finally "tujuego.BAS"

To create a wav or cdt from WinAPE

file-> tape-> press record

then you will get a menu of file management so we can decide what name we give to wav file or cdt

If you are in CPC 6128 mode, then then you must run from basic

| TAPE

and then

WRITE SPEED 1

With this command we will have done is to tell the AMSTRAD record at 2000 baud. So the charge will last less. If you do not execute this command, the recording will be made at 1000 baud, safer but much slower

save "LOADER.BAS"

Leave a message indicating that pressures rec & play

and then you press "enter".

Then record each and every one of the files:

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

Finally, we must make one last operation to close the file WinAPE.

file-> remove tape

After making the remove tape, the file will acquire its size (if you do not you see that in the drive of your PC and the file does not grow is because it has not been flushed to disk)

To load the game, if you are in a CPC6128
| TAPE
RUN ""

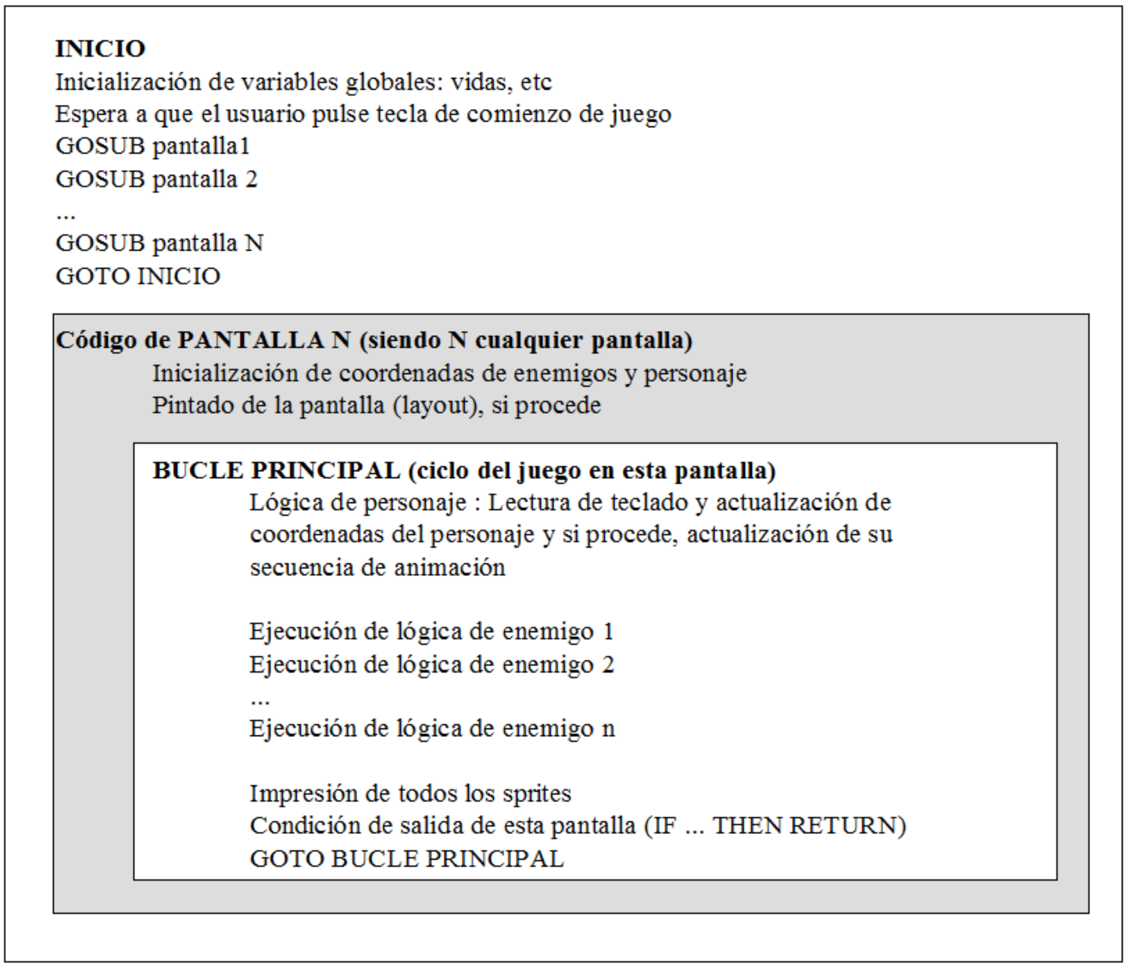
To reuse the disk
| DISC

5 Game cycle

A video game arcade, platforms, adventure, usually has a similar type of structure, in which a certain operations will be repeated cyclically in what we will call "game cycle".

In each update cycle game we will print positions of sprites and sprite screen, so that the number of cycles game running per second equivalent to the "frames per second" (fps) of the game.

The following pseudo code outlines the basic structure of a game



```
INICIO
Inicialización de variables globales: vidas, etc
Espera a que el usuario pulse tecla de comienzo de juego
GOSUB pantalla1
GOSUB pantalla 2
...
GOSUB pantalla N
GOTO INICIO

Código de PANTALLA N (siendo N cualquier pantalla)
        Inicialización de coordenadas de enemigos y personaje
        Pintado de la pantalla (layout), si procede

        BUCLE PRINCIPAL (ciclo del juego en esta pantalla)
                Lógica de personaje : Lectura de teclado y actualización de
                coordenadas del personaje y si procede, actualización de su
                secuencia de animación

                Ejecución de lógica de enemigo 1
                Ejecución de lógica de enemigo 2
                ...
                Ejecución de lógica de enemigo n

                Impresión de todos los sprites
                Condición de salida de esta pantalla (IF ... THEN RETURN)
                GOTO BUCLE PRINCIPAL
```

Fig. 9 Basic structure of a game

If the logic of the enemies is very heavy for having many enemies or be very complex, it will consume more time in each game cycle and therefore the number of cycles per second will be reduced. Try not to lose 10fps or 15 fps for the game to maintain an acceptable level of action.

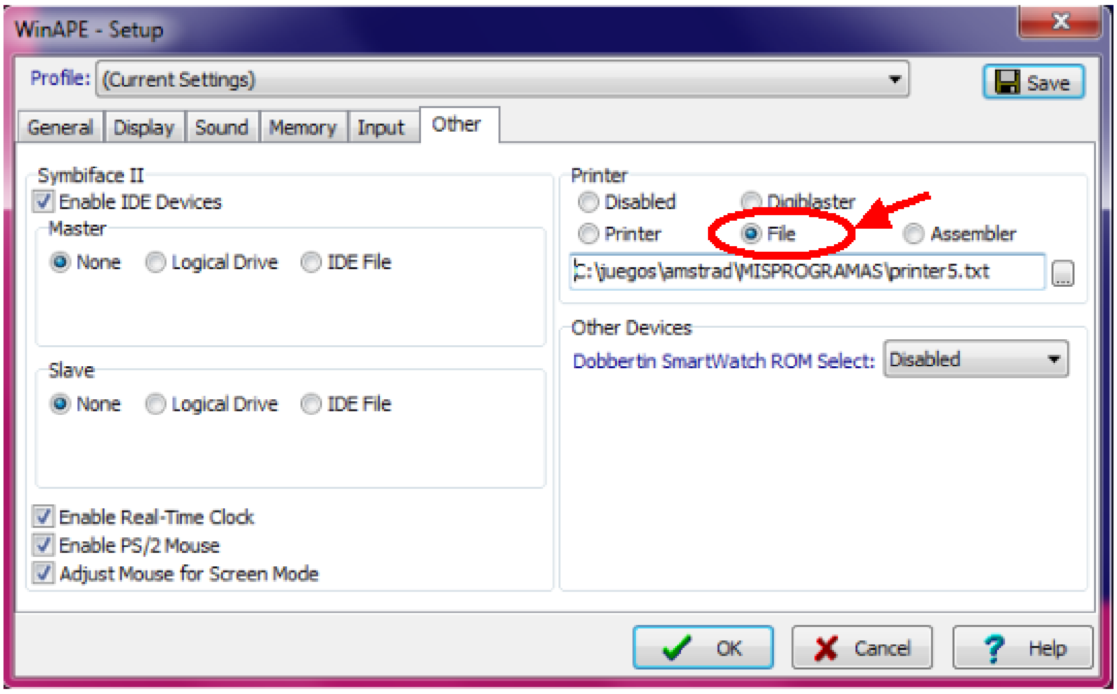
6 Sprites

6.1 Edit sprites with spedit and assemble

Spedit (Simple Sprite Editor) is a tool that will allow you to create your images of characters and enemies and use them in your programs BASIC

Spedit is made in BASIC, and it is very simple, so you can modify it to do things that are not covered and interest you. It runs on the Amstrad CPC, although this meant for you to use from the WinAPE emulator.

The first thing to do is set WinAPE output to the printer to remove a file. In this example I have set the printer output to file printer5.txt


WinAPE - Setup
Profile: (Current Settings)
Save
General | Display | Sound | Memory | Input | Other
OK
Cancel
Help

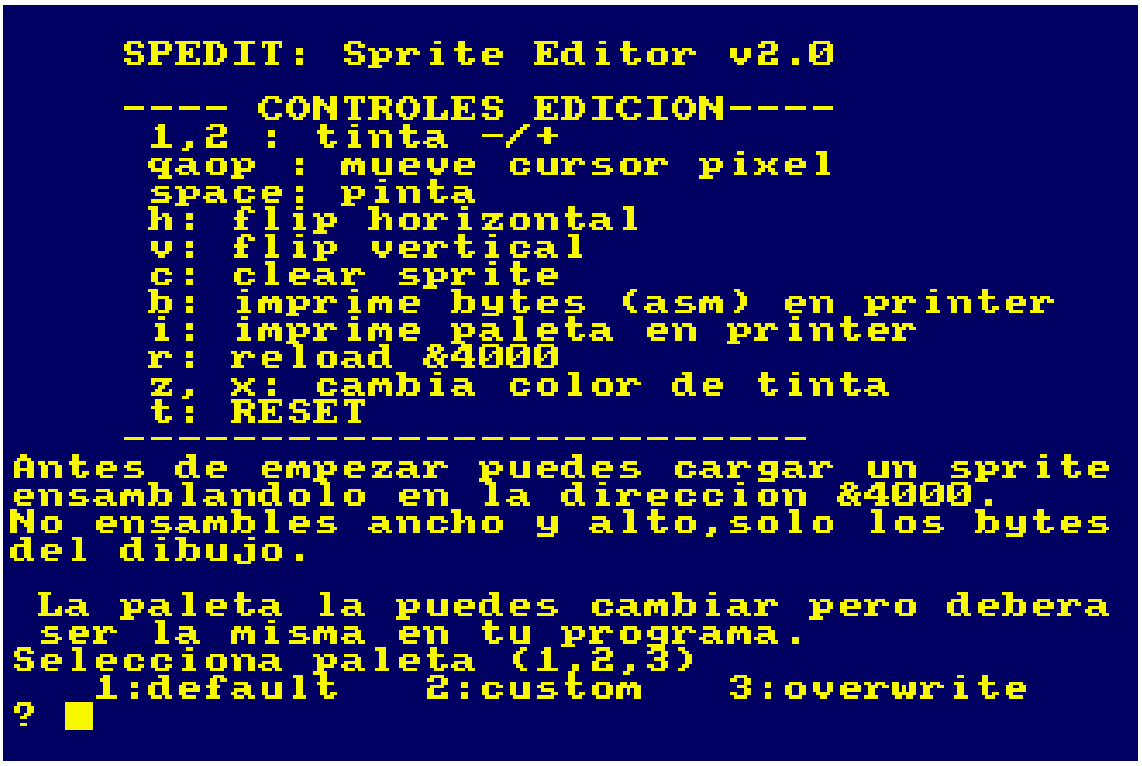
Symbiface II
[✓] Enable IDE Devices
Master: [● None] ○ Logical Drive ○ IDE File
Slave: [● None] ○ Logical Drive ○ IDE File
[✓] Enable Real-Time Clock
[✓] Enable PS/2 Mouse
[✓] Adjust Mouse for Screen Mode

Printer
○ Disabled ○ Digiblaster
○ Printer [✓ File] ○ Assembler
C:\juegos\amstrad\MISPROGRAMAS\printer5.txt

Other Devices
Dobbertin SmartWatch ROM Select: Disabled

Fig. 10 Printer redirection to a file with CPC WinAPE

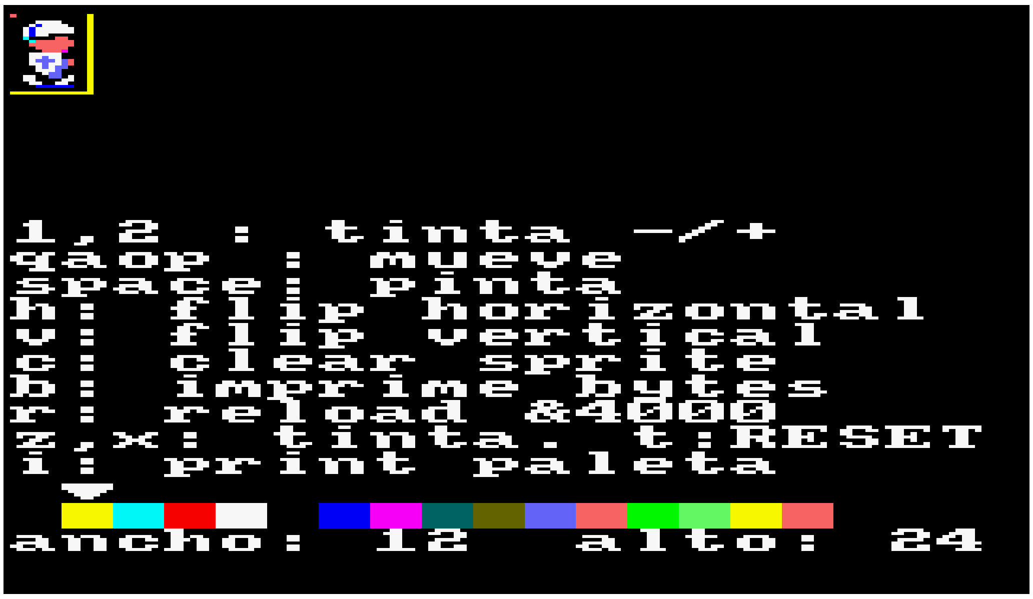
When you run SPEDIT you the following menu where you can choose whether you will use the default palette or a yours you want to define appear. If you choose to define your own palette, you will have to reprogram the lines of BASIC where the alternative palette, which is a subroutine that is invoked with GOSUB is defined when you press "N" in the answer to the question of whether you want to use the default palette .



```
     SPEDIT: Sprite Editor v2.0

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

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

Fig. 11 ininical screen SPEDIT

Assuming that choose to use the default palette, the tool is set to mode 0 and allows you to edit drawings, with the help on the screen. Driving a flashing pixel and at the bottom the coordinates where you are and the value of the byte in which you are shown.



```
1,2  :  tinta  -/+
qaop  :  mueve
space:  pinta
h:  flip  horizontal
v:  flip  vertical
c:  clear  sprite
b:  imprime  bytes
r:  reload  &4000
z,x:  tinta.  t:RESET
i:  print  paleta

ancho:  12   alto:  24
```

Fig. 12 Editing screen SPEDIT

SPEDIT allows you to "mirror" your image to the same doll walking leftward effortlessly, simply press H (horizontal flip) and the same can be done vertically. It also allows you to "mirror image" about an imaginary axis located in the center of the character, both vertically and horizontally. This is very useful for symmetrical or almost symmetrical, where a help to draw always good characters.

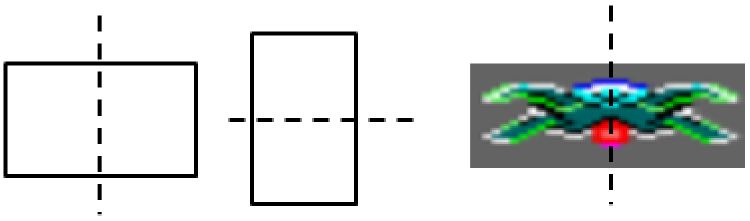
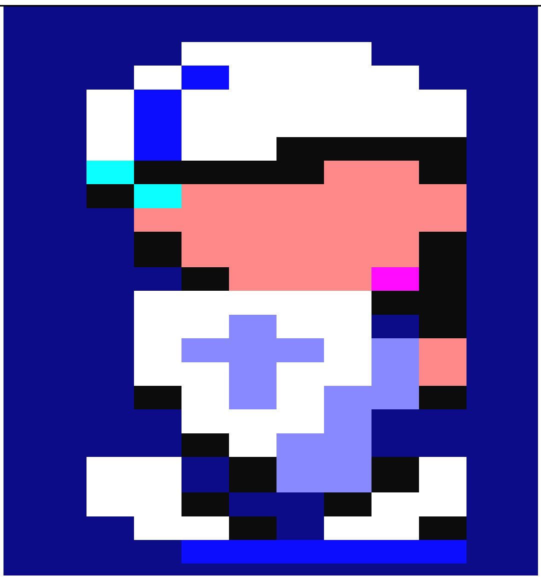


Fig. 13 sprites symmetrical with SPEDIT

It is easy to adapt this tool to allow you to edit in mode 1 if you wish. In fact it is as simple as removing the BASIC line that sets the screen mode. As you see the tool is not yet complete but allows you to do absolutely everything you need.

Once you have defined your doll, to remove the assembly code must press the "b". This will send to the printer (the file we have defined as output) a text like the following, which can add a name, I called him "SOLDADO_R1"

```
;------ BEGIN SPRITE --------
SOLDADO_R1
db 6 ; ancho
db 24 ; alto
db 0 , 0 , 0 , 0 , 0 , 0
db 0 , 0 , 0 , 0 , 0 , 0
db 0 , 0 , 48 , 48 , 0 , 0
db 0 , 16 , 56 , 48 , 32 , 0
db 0 , 52 , 48 , 48 , 48 , 0
db 0 , 52 , 48 , 48 , 48 , 0
db 0 , 52 , 48 , 240 , 240 , 0
db 0 , 88 , 240 , 229 , 218 , 0
db 0 , 164 , 207 , 207 , 207 , 0
db 0 , 69 , 207 , 207 , 207 , 0
db 0 , 80 , 207 , 207 , 218 , 0
db 0 , 0 , 229 , 207 , 248 , 0
db 0 , 16 , 48 , 48 , 240 , 0
db 0 , 16 , 37 , 48 , 80 , 0
db 0 , 16 , 15 , 26 , 79 , 0
db 0 , 16 , 37 , 48 , 79 , 0
db 0 , 80 , 37 , 37 , 90 , 0
db 0 , 0 , 48 , 37 , 0 , 0
db 0 , 0 , 176 , 15 , 0 , 0
db 0 , 48 , 80 , 15 , 176 , 0
db 0 , 48 , 160 , 80 , 48 , 0
db 0 , 16 , 112 , 16 , 112 , 0
db 0 , 0 , 60 , 60 , 60 , 0
db 0 , 0 , 0 , 0 , 0 , 0
;------ END SPRITE --------
```

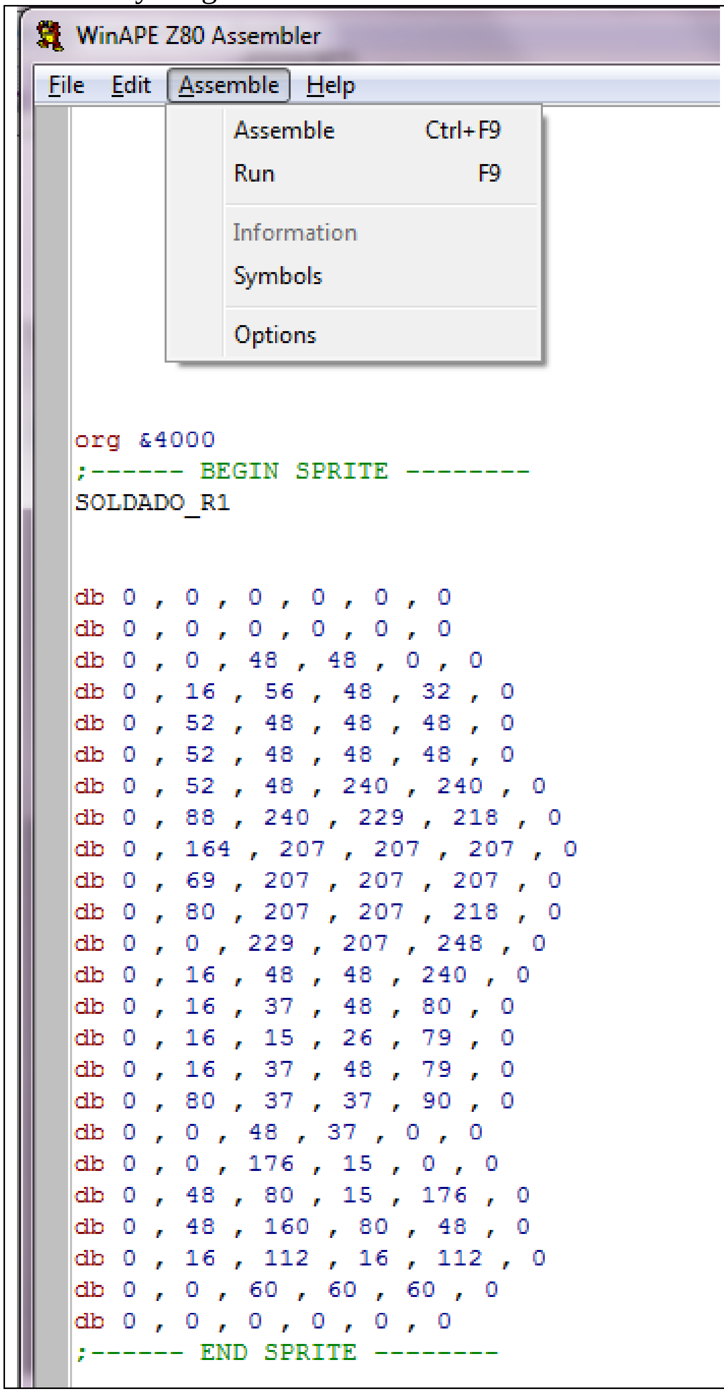


Notice how I have always left a byte to the left to zero. I have done so to move the soldier to the right, is "clear himself" because otherwise deraría a trail, "smearing" the screen while navigating

Fig. 14 Soldier in .asm format

Once you've done the first frame of your soldier you can quit your job and continue another day. From Soldier to you have drawn and continue to tweak or modify it to build another frame, you can join the soldier in the direction & 4000, removing the width and height. Once assembled from WinAPE, you tell SPEDIT you're going to edit a sprite of the same size and once you are on the editing screen you press "r". The sprite is loaded from the address & 4000, where it has "joined"

Much of the appeal of a game are your sprites. Do not skimp this time, do it slowly and taste and your game will look much better.



```
org &4000
;------ BEGIN SPRITE --------
SOLDADO_R1


db 0 , 0 , 0 , 0 , 0 , 0
db 0 , 0 , 0 , 0 , 0 , 0
db 0 , 0 , 48 , 48 , 0 , 0
db 0 , 16 , 56 , 48 , 32 , 0
db 0 , 52 , 48 , 48 , 48 , 0
db 0 , 52 , 48 , 48 , 48 , 0
db 0 , 52 , 48 , 240 , 240 , 0
db 0 , 88 , 240 , 229 , 218 , 0
db 0 , 164 , 207 , 207 , 207 , 0
db 0 , 69 , 207 , 207 , 207 , 0
db 0 , 80 , 207 , 207 , 218 , 0
db 0 , 0 , 229 , 207 , 248 , 0
db 0 , 16 , 48 , 48 , 240 , 0
db 0 , 16 , 37 , 48 , 80 , 0
db 0 , 16 , 15 , 26 , 79 , 0
db 0 , 16 , 37 , 48 , 79 , 0
db 0 , 80 , 37 , 37 , 90 , 0
db 0 , 0 , 48 , 37 , 0 , 0
db 0 , 0 , 176 , 15 , 0 , 0
db 0 , 48 , 80 , 15 , 176 , 0
db 0 , 48 , 160 , 80 , 48 , 0
db 0 , 16 , 112 , 16 , 112 , 0
db 0 , 0 , 60 , 60 , 60 , 0
db 0 , 0 , 0 , 0 , 0 , 0
;------ END SPRITE --------
```

With this you know what it means to "join" a sprite. Is simply put the data bytes constituent in consecutive memory addresses, in this case beginning with the & 4000, which is 16384 in decimal, that is the position 16KB

SPEDIT takes up very little memory and that direction is far from the program so that there is no problem that we are damaging to assemble "the SPEDIT program.

If someday SPEDIT becomes bigger and gets to have much more functionality, we must take this small buffer further, but for now it is perfectly valid way, in the direction & 4000.

Fig. 15 Assembled graphics

To know that memory address is assembled each image used from the menu WinAPE:

Assemble-> symbols

This relationship will see a label that you have defined as "SOLDADO_R1" and the memory address from which it has been assembled.

Once you've made the various stages of animation of your soldier, you can group them into a "sequence" of animation.
Animation sequences are lists of images and not defined SPEDIT. With SPEDIT simply edit the "frames". In a later section I will explain how to tell the library 8BP a set of images that make up an animation sequence.

The images go by for your game sees keeping them all in a single file, which is titled "images_mijuego.asm" for example. Once they are all made can assemble this file in the direction 34000 and be saved in a binary file from the amstrad with the command SAVE

For example if you have made 2000 bytes of images, after assembling in 34000 BASIC runs from the CPC in the emulator:

```
SAVE "sprites.bin", b, 34000.2000
```

With this you will have saved your disk image file. In this example I have set a length of 2000 but if you have drawn many sprites you may have to put up 8500. Do not forget the letter "b" which is used to specify that it is a binary file.

If you want to save images and animation sequences together (the sequences found in the 33500) simply run

```
SAVE "sprites.bin", b, 33500.8500
```

To load your sprites in RAM, simply run:

```
Load "sprites.bin"
```

6.2 Sprites Overwrite

Since version 22 8BP you can edit transparent sprites, ie, which can fly over a background and reestablish the pass. To do the sprites that disfuten this possibility must be configured with a "1" in the overwrite flag status byte (bit 6). The following section will detail the status byte properly. Let's see how a sprite is edited with this capability SPEDIT

Many games use a technique called "double buffering" to reset the background when a sprite moves across the screen. It is based on having a copy of the screen (or the playing area) into another memory area, so although our sprites destroy the background, we can always consult in that area beneath and thus reset. Actually that is the basic principle but is somewhat more complex. It is printed in double buffering (also called backbuffer) and when you're all printed, turns to the screen or done to switch the start address of the video memory from the new original direction display, the double buffer. The switching is instantaneous. To build the next frame the direction of original screen where it now is no longer pointing video memory is used. There the new frame is constructed and switched back alternately in each frame. These techniques, although work very well, have a couple of disadvantages to nuestos purposes: take more CPU time and consume much more memory (up additional 16KB), leaving very little memory for our BASIC program. If a game is entirely in assembler, this is not as serious because 10KB assembler go far, but 10KB BASIC is an understatement.

The solution adopted in 8BP is inspired by the programmer Paul Shirley (author of "mission Genocide", but is slightly different. I will explain directly our 8BP solution:

In amstrad a pixel of mode 0 is represented with 4 bits, so are possible up to 16 different colors from a palette of 27. Well, if we use a bit for the background color and 3 colors for sprites, we will have a total of 2 colors background colors + 7 + 1 color to indicate transparency = 9 colors in total. This will allow us to "hide" the background color in the color of the sprite, but we pay the price of reducing the number of colors in just 16 9. In addition, the fund may only be two colors. However the ornamental elements of the game screen may have more color, as the sprites will not pass over, so we can get a certain amount of color in our game. Here is an example of the results that can be obtained with this technique:

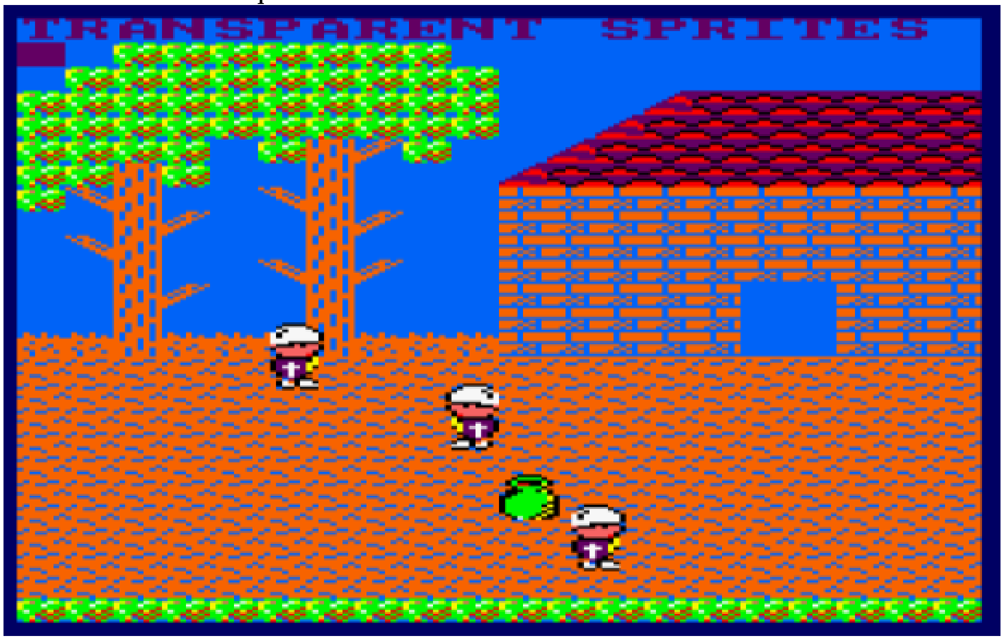
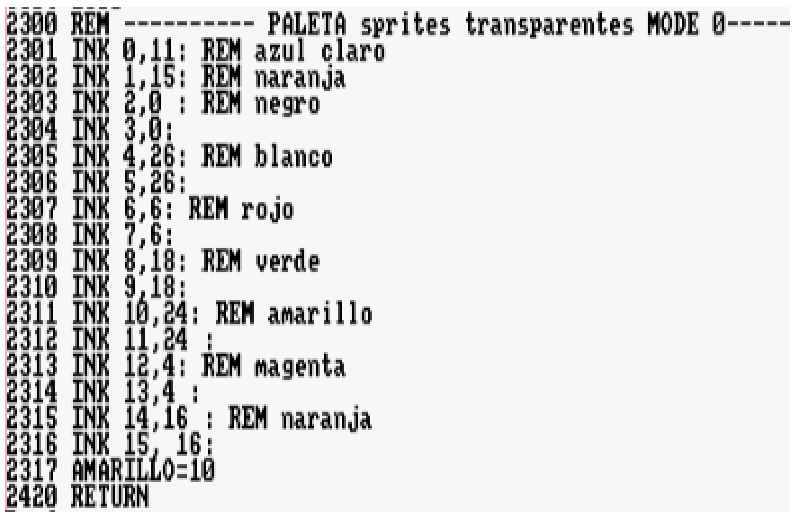


Fig. 16 Sprites Overwrite

To edit such sprites must use an appropriate palette of 9 colors, where each color sprites two binary codes (corresponding to 0 and 1 bit of background) are used. In the SPEDIT thus defined is a palette that you can use selecting "3" in choosing the palette. It has been built like this:



```
2300 REM ---------- PALETA sprites transparentes MODE 0-----
2301 INK 0,11: REM azul claro
2302 INK 1,15: REM naranja
2303 INK 2,0 : REM negro
2304 INK 3,0:
2305 INK 4,26: REM blanco
2306 INK 5,26:
2307 INK 6,6: REM rojo
2308 INK 7,6:
2309 INK 8,18: REM verde
2310 INK 9,18:
2311 INK 10,24: REM amarillo
2312 INK 11,24 :
2313 INK 12,4: REM magenta
2314 INK 13,4 :
2315 INK 14,16 : REM naranja
2316 INK 15, 16:
2317 AMARILLO=10
2420 RETURN
```

Fig. 17 Overwriting example palette

You see, after the color 0 and 1, all colors are repeated two veces.Tu can build your own palette in this way. You can help by referring to the appendix devoted to the color palette.

With the SPEDIT editor you can modify the palette to your liking.without editing manually INK commands, and allows export to copy in our BASIC programs. The export is done by sending the printer INK commands that make up the palette. (The printer redirected to a file from WinAPE). We have the z / x keys to alter the palette and the "i" option to export the output file. This is an example of what we export:

```
'BEGIN ------ -------- INK PALETTE 0, 1 INK 1 INK 24 2, 20 INK 3, 6
INK 4, 26 INK 5, 0 INK 6, 2 INK 7, 8 INK 8, 10 INK 9, 12 INK 10, 14
INK 11, 16 INK 12, 18 INK 13, 22 INK 14, 0 INK 15, 11 'PADDLE --------
------ END
```

Sprites you use to build the drawings of background colors can only be 0 and 1 but the sprites you use to decorate, where not going to happen moving sprites can use the 9 colors.
You can also increase the colorful decorated with elements that are sprites instead of funds, such as green cauldron in the previous example. Thus you can have very colorful results.
The 0001 color has a "special" use. If you edit a sprite without overwriting flag is simply the color 1, but if you edit a sprite with overwrite flag when printed will be left unpainted those pixels, respecting what had underneath. This allows collisions between sprites are not "rectangular" but retain the shape of the sprite.

code	meaning
0000	Background Color 1. If a sprite and uses active overwrite the flag, it means "transparency", ie, print reestablishing the background
0001	2 color background. If a sprite and uses active overwrite the flag, ceases to mean a color to mean "not print". It is respected whatever is in this

code	meaning
0110	sprite color 3
0111	
1000	4 color sprite
1001	
1010	5 color of sprite
1011	

	pixel, for example, a colored previously printed by a sprite with which we are overlapping pixel.
0010	Color 1 sprite
0011	
0100	2 color sprite
0101	
1100	6 color sprite
1101	
1110	7 color sprite

9 colors in total:

- 2 background
- 7 for sprites (actually 8 but a means transparency -000)
- Ornamental elements can use 9.

Then I'll show you a sprite where I painted color 0001 what can not be painted, ie, where not even going to restore the background, as with other pixels to 0000 is enough to erase the trace sprite while traveling.

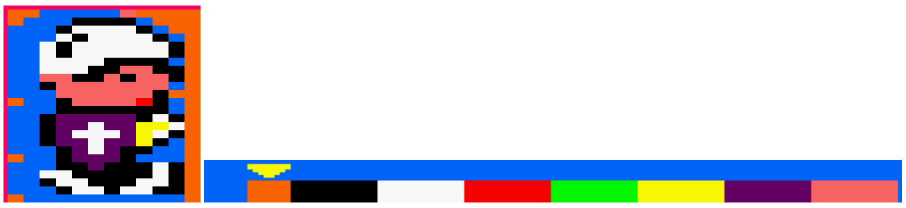
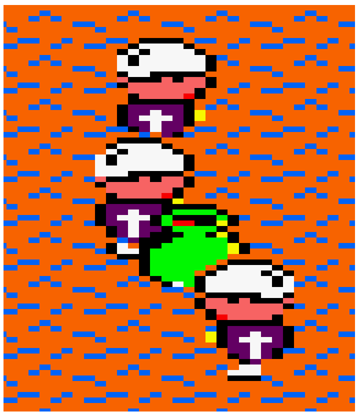


Fig. 18 sprite designed to overwrite

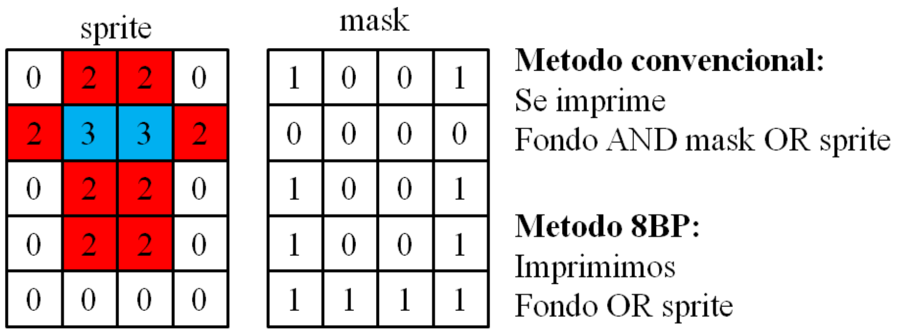
As you can imagine, in the case of the pot, being a sprite that does not move and therefore does not clear itself, all its outline is painted with color 0001. This allows perfect collisions without rectangular shapes that show that sprites are actually rectangles. The end result is shown below in a multiple collision



As you may have guessed, the collision besides being perfect, evidence that sprites have been sorted according to their Y coordinate, so that the last printed is located at the lowest position. This is done with a simple parameter to print sprites with the command | PRINTSPALL, to be discussed later.

Fig. 19 Multiple collision, color effect 0001

Printing operations with this mechanism are very fast, without defining what is known as "sprites masks." Masks are bitmaps the size of a sprite that serve to accelerate the printing operations. In this case they are not necessary. The figure below shows a typical mask associated with a sprite. First it is usually done the AND operation between the background and the mask and then made the OR with sprite. In 8BP it is faster, because the sprite does not touch the bit for the bottom, so that the OR operation between the background and sprite respects the background while painting the sprite. If you do not understand this very well, do not worry, it is not important to understand it is not necessary in 8BP.



0	2	2	0
2	3	3	2
0	2	2	0
0	2	2	0
0	0	0	0

1	0	0	1
0	0	0	0
1	0	0	1
1	0	0	1
1	1	1	1

Metodo convencional:
Se imprime
Fondo AND mask OR sprite

Metodo 8BP:
Imprimimos
Fondo OR sprite

Fig. 20 In 8BP masks are not necessary

Print sprites with active overwrite flag is more expensive than printing without overwriting. Despite not require mask and be very fast, this impression cosume approximately 1.5 times the time-consuming printing a sprite without overwriting. For this reason, use it when necessary, and do not use it if your game is not going to have a background drawing sprites must respect.

6.3 Sprite attribute table

The sprites are stored in a table containing a total of 32 sprites.
Each entry in the table contains all attributes of the sprite and occupies 16 bytes for performance reasons, because 16 is a multiple of 2 and it allows access to any sprite with a very inexpensive multiplication. The table is located in the 27000 memory address, so that you can access from PEEK and POKE basic, but also have RSX commands to manipulate the data in this table.

Sprites have a set of parameters, of which the first one is the status byte flags. In this byte, each bit represents a flag and each flag means one thing, specifically represent if the sprite is taken into consideration when executing certain functions
The following table summarizes what happens if they are active ("1")

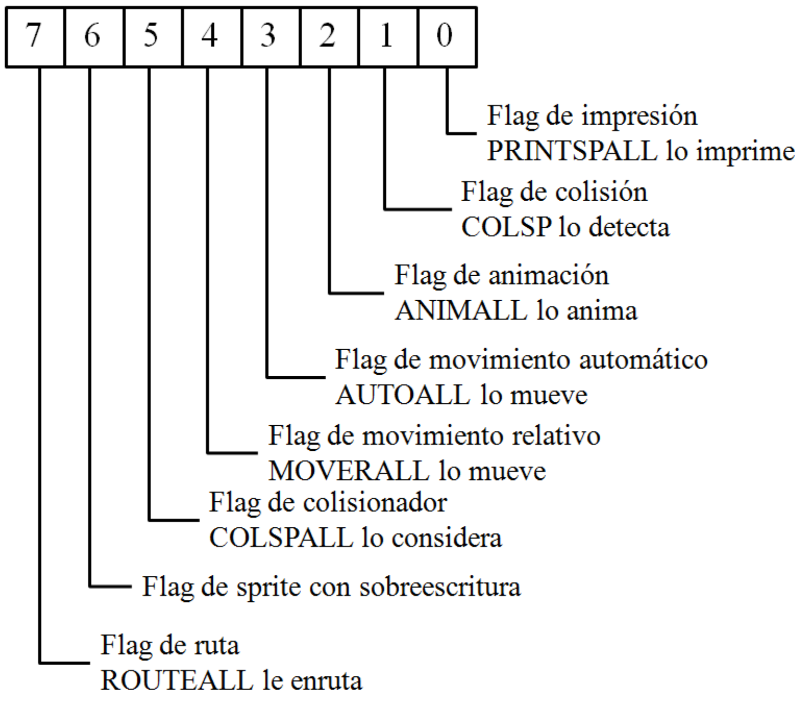


Fig. 21 flags in the status byte

To understand the power of these flags we will see some examples:

- Bit 0: print flag: our character or enemy ships will have it activated and call upon each cycle of the game to be printed PRINTSPALL and all at once

- bit 1: collision flag: a fruit or currency for example can not have print flag but having the collision

- bit 2: Automatic flag animation: is taken into account in ANIMA_ALL (). In the case of the character, I recommend it off, because if I stay still should not change the frame.

- bit 3: automatic movement flag. It moves only by invoking AUTO_ALL () taking into account its speed. useful in meteorites and guards come and go.

- bit 4: relative motion flag. All sprites that have this flag move while invoking MOVER_ALL (incy, incx) very useful in warehouses in training and arrivals planets. It also serves to simulate a scroll if you leave your character in the center and press the controls homes or move elements around. It seems that your character progresses for the territory.

- bit 5: flag collider. All sprites with this active flag are considered by the COLSPALL function, when detect a possible collision with other sprites.

- Bit 6: flag Overwrite: If this flag is active, the sprite can move above a background, reestableciéndolo passing. This is an advanced option and involves the use of a special color palette, 9 colors. Overwriting is this "price".

- Bit 7: Route flag: If this flag is active, the ROUTEALL command lets you move a sprite through a route that you define, defined by a series of segments. The ROUTEALL command knows that segment and each sprite position is and if it comes to a segment change, change the speed of the sprite according to the conditions of the next segment. ROUTEALL not move the sprite, just touch your speed. To move you have to use it in conjunction with AUTOALL.

Examples of allocation status byte value:

Tipico enemy: a sprite to be printed in each cycle, with collision detection with other sprites and animation must have:

status = 1 (bit 0) + 2 (bit1) + 4 (bit 2) = 7 = & x0111

A house that moves to move: a sprite that is printed on each cycle without collision detection with each other and relative movement

status = 1 (bit 0) + 0 (bit1) + 0 (bit 2) + 0 (bit 3) + 16 (bit 4) = 17 = & x10001

A fruit that gives bonus: a sprite that is not printed on each cycle but has collision detection

status = 0 (bit 0) + 2 (bit1) = 2 = & x10

The sprite attribute table consists of 32 entries of 16 bytes each, starting at address 27000
The reason for having 16 bytes is none other than the performance, and to calculate the direction of the sprite N involves multiplying by 16, which being a multiple of 2, can be done with an offset. This is useful in operations involving a single sprite. For operations that run through the table sprites (as | PRINTSPALL or | COLSP), internally the table is walking with an index to which is added 16 to move from one sprite to the next. The sum is the fastest in that case.

The attributes of each sprite are:

attribute	Byte	Logitud	meaning

		(bytes)	
status	0	1	Byte contains the status flags for operations PRINTSPALL, COLSP, animall, AUTOALL, MOVERALL, COLSPALL and ROUTEALL
Y	1	2	Y coordinate [-32768..32768] corresponding values within the screen are [0..199]
X	3	2	Easting in bytes [-32768..32768] corresponding values within the screen are [0..79]
Vy	5	1	Step to take in the automatic movement
Vx	6	1	Step to take in the automatic movement
Sequence	7	1	Identifier of the animation sequence [0..31]. If you have no sequence must be assigned a zero
photogram	8	1	Frame number in the sequence [0..7]
Image	9	2	Memory address where this image
previous Sprite	10	2	Internal use for sorting mechanism sprites
following Sprite	12	2	Internal use for sorting mechanism sprites
Route	fifteen	1	ID route should follow the sprite

The direction of the coordinates of each sprite can be calculated so

coordinate direction Y = 27000 +16 * N +1
coordinate direction X = 27000 * N +3 +16

In this way we can make a change in that POKE coordinate any sprite
And in general, the addresses of the 32 sprites to deal with POKE and PEEK are:

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

Table 3 Addresses attributes of the 32 sprites

The space occupied by each sprite in the table is 16 bytes. As you see the coordinates X and Y are numbers 2 bytes. Sprites accept negative coordinates so you can partially print a sprite on the screen, giving the feeling that is coming slowly. You can not set negative coordinates with POKE, but if you can do it with | LOCATESP and also | POKE, which is a version of BASIC POKE command but accepts negative numbers.

It is good practice to place the character or spacecraft at position 31 (there are 32 numbered sprites from 0 to 31). If your ship has position 31 last, above the rest of sprites overlap it should be printed.

6.4 Printing all sprites and ordered

In the library 8BP You have a command that prints at once all sprites that have the flag active print. This is the command | PRINTSPALL

This command has 3 parameters, but only need to fill the first time you invoke, for the following times, will remember the parameters, and only need to fill them again if you want to change any of them. This is useful because parameter passing time consuming (you can save more than 1ms avoiding passing parameters).

The parameters are:
| PRINTSPALL, <flag Order>, <cheer before printing>, <sync>

- The timing parameter may take the values 0 or 1 and indicates that waits for an interrupt scanning screen to print. If you want speed do not recommend it

- The animation parameter can take the values 0 or 1. Should be active, before printing each sprite animation its flag is checked in the status byte and if you have set, then it will be changed before printing frame. It is very useful with enemies but with your character not normally, because only you will want to move him and encourage him not in each frame

- Finally we have the order parameter. We must indicate the number of sprites sorted by Y coordinate that we will print. For example if we put a 0, then the sprites are printed sequentially from the sprite sprite 0 to 31. If we put a 10 art print is from 0 to 10 ordered (11 sprites) and from 11 to 31 sequentially. If we all ordered 31 sprites are printed.

The system is very useful for games such as “Renegade” or “Golden AXE”, where it is necessary to give a depth effect. The system is seen when there is overlap between sprites.

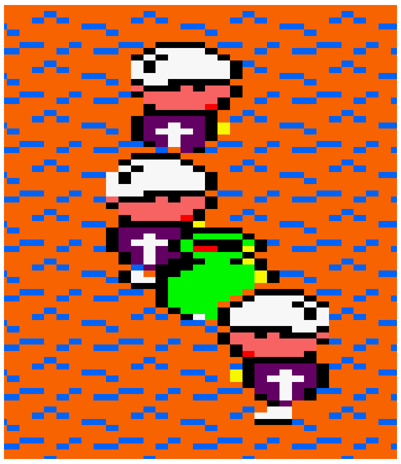


Fig. 22 Effect of order of sprites

| PRINTSPALL, 0, 1, 0: sequentially prints all sprites
| PRINTSPALL, 31, 1, 0: print neatly all sprites
| PRINTSPALL, 7, 1, 0: prints in an orderly and sequential 8 sprites rest

Print orderly computationally more costly to print sequentially. If you have just 5 sprites that must be ordered, pass a 4 as a parameter ordering, not spend a 31. Sort all sprites takes about 2.5ms but if you order just 5 you can save 2ms. Maybe you have many sprites and not worth ordering some, such as shots or sprites you know you are not going to overlap.

The algorithm used to sort the sprites is a variant of the algorithm known as "bubble". Although you will find in the literature that the algorithm called "bubble" is very inefficient, they say that those who speak considering a list of random numbers to order. We are faced with a case where the sprites are normally ordered and almost one frame to another only clutters one or two sprites, no more, as their coordinates evolve "smoothly". Therefore, the algorithm through the list of sprites and when it finds a pair of unordered sprites, turns them over and stop following ordering. It is blazingly fast, and if only able to order a couple of sprites ever, is ideal for this use case. Only if you have two untidy sprites and also they are overlapping, there is a frame in which we see one of them being printed in disarray, but it will be corrected in the next frame. It is imperceptible.

6.5 Collisions between sprites

To check if your character or your shot has collided with other sprites You have the command

```
| COLSP, <sprite_number>, @ collision%
```

Where is the sprite sprite number you want to check (your character or your shot) and the collision variable is an integer variable that previously had to be defined by assigning an initial value, for example:

```
% = 0 collision
| COLSP, 1, @ collision%
```

The collision variable is filled with the first identifier is detected sprite has collided with your sprite, although a multiple collision could occur, but the command just hands you a result.

Internally the library 8BP sprites runs from 31 to 0, and if they have the active collision flag (bit 1 byte status) then checks if your sprite collides with. If there is no collision, collision% variable value is zero. If any will return the number of sprite is colliding with your sprite. If, for example Collide 4 and 12, the function will return a 12 for checks before the 12 4.

Neither your character and your shot must have the active flag collision sprites, since otherwise always collide ... themselves!

The collision between sprites is a costly task. Internally, the library needs to calculate the intersection between boxes containing each sprite to determine if there is overlap between them. That is why to save calculations, it is best to locate enemies in consecutive positions of sprites. If for example the enemies that are the sprites can hit 15 to 25, we can set the collision to only check those sprites. To invoke this sprite collision on 32 that does not exist. That will tell the 8BP library that is configuration information for the command:

```
| COLSP, 32, <initial sprite>, <end sprite>
```

```
| COLSP, 32, 15, 25
```

This optimization although not very significant, it starts to be when invoked several times COLSPALL COLSP or internally invoked several times COLSP command is used.

Another interesting optimization, able to save 1.1 milliseconds on each invocation, is to tell the command that you always use the same variable BASIC to make the result of the collision. To do this we will tell you using as sprite 33, which does not exist

```
col = 0%
| COLSP, 33, @Col%
```

Once executed these two lines, the following invocations COLSP, will leave the result in the col variable, without indicate, for example:

```
| COLSP, 23: rem this invocation is equivalent to | COLSP, 23, @Col%
```

6.6 Adjusting the sensitivity of the collision of sprites

You can adjust the sensitivity of the COLSP command, deciding whether the overlap between sprites must be several pixels or one to consider that there has been a collision.

For this you can set the number of pixels (pixels in the Y direction, X direction bytes) Overlap necessary both in the Y direction and the X direction, using the COLSP command and specifying the sprite 34 (which does not exist)

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

The library does not use 8BP "pixels" in the X coordinate, but bytes, so you should be aware that a collision of 1 byte, actually are 2 pixels and that is the minimum possible collision when you set dx = 0.
In the Y coordinate, the library works with lines so that dy = 0 means a collision of a single pixel.

A strict, useful collision for shots would be one that does not tolerate any margin, considering collision as a minimum overlap between sprites (1 pixel in the Y direction or a byte address X)

```
| COLSP, 34, 0, 0: rem collision as a minimum overlap
```

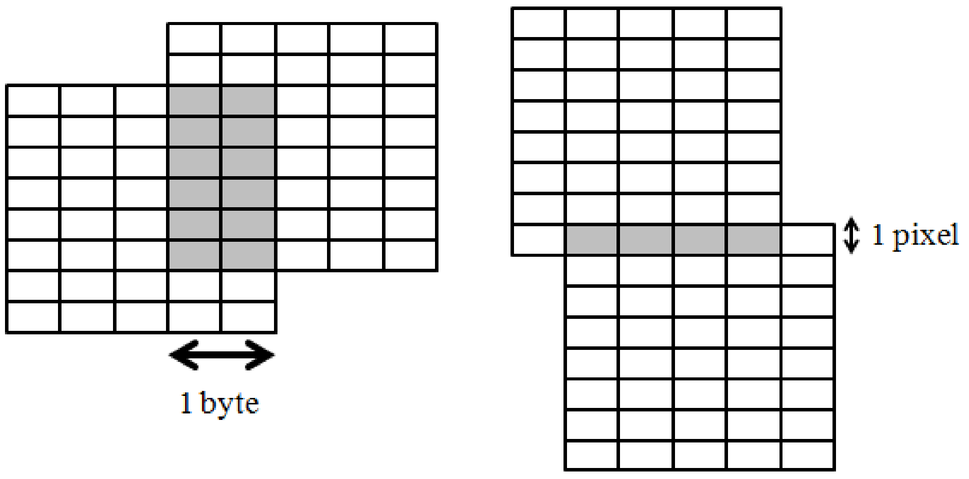


Fig. 23 Strict collision using COLSP, 34, 0, 0

However, if we are doing a game in MODE 0, where the pixels are wider than tall, it is perhaps more appropriate to give some leeway in Y and X. For example nothing

| COLSP, 34, 2, 0: rem collision with 3 pix 1 byte Y and X

My recommendation is that if there are narrow shots or small, fits collision with (dy = 1, dx = 0) while if there is only large characters can leave it with more margin (d = 2, dx = 1). You should also consider that if your sprites have a "margin" to move around erase erasing themselves, that margin should not be part of the consideration collision so it makes sense that both dx dy as non-zero. In any case it is something that will decide depending on the type of game you do

6.7 Who collides and with whom: COLSPALL

With the COLSP function we've seen so far, it is possible to detect collision of a sprite with everyone else. However if we have a multi-shot, where for example our ship can shoot up to 3 shots simultaneously, we would have to detect the collision of each and additionally our ship, resulting in 4 invocations COLSP.

We must remember that each invocation through the layer of syntactic analysis, so 4 invocations costly. For this we have an additional command: COLSPALL

This function works in two steps, we must first specify which variables will store the sprite collider and collided

| COLSPALL, @ collider% @ collided%

And then in every game cycle simply we call the function with no parameters:

| COLSPALL

The function will be considered as sprites "colliders" those with the flag of collider to "1" in the status byte (the bit 5), and "collided" those sprites that have a "1" the flag of collision (bit 1) of the status byte. Sprites colliders should be our ship and our shots

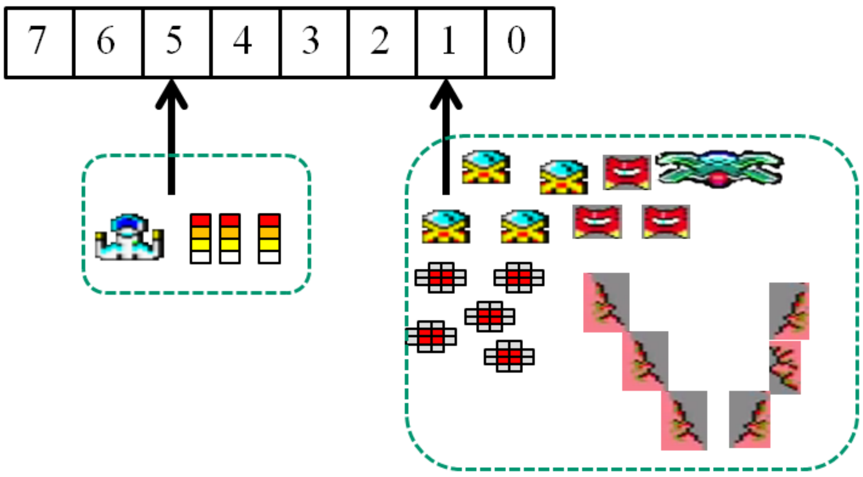


Fig. 24 colliders versus collided

The COLSPALL function begins by checking the sprite 31 (if collider) and goes down to the sprite 0, internally invoking collider COLSP for each sprite. When it detects a collision, it interrupts its execution and returns the value of the collider and collided. It is therefore important that our ship has more than our shots sprite. Thus, if we reach, it will detect if we have reached an enemy with a shot at the same instant.

In each game cycle it may only detect a collision, but it's enough. It is an important limitation in each frame can only begin to "explode" an enemy. If, for example, throw a Granada and there is a group of five soldiers involved, each soldier will start to die in a different frame, and 5 frames are all exploding. Using COLSPALL not explode all at once, but your game will be faster and in an arcade is very important.

6.7.1 How to program a multiple shot without COLSPALL

We will analyze how the collision of a ship and fire is detected without using COLSPALL. As we shall see, because of the times we invoke COLSP, it is more efficient to use COLSPALL. This example will understand and at the same time we have evidence that if only one sprite collider (our ship) or two (our ship and shot as much), can be dispensed with COLSPALL.

Let's assume we have up to 3 shots with 7.8 and 9 sprites and our ship is 31. sprite collision detection we will make our ship with an invocation to COLSP in each game cycle

First of all, to avoid passing parameters in each invocation will COLSP

```
col = 0%
| COLSP, 33, @ col%:'Sprite 33 does not exist, it is used to indicate the variable work
```

This successive invocations COLSP mode, <sprite> leave the result (number of sprite that collides) in col variable%

```
| COLSP, 31:'Assume the sprite 31 is our character.
```

To detect collisions of our shots, the most effective solution is based on one of the simplest cases of application of the technique of "massive logic", restricting to contemplate only one of the following things at once:

- collides one of 3 shots (one), and kills an enemy
- a shot out of the screen area (only one)

The routine would be something like (within each game cycle would make a GOSUB 750). In the worst case there are 4 COLSP invocations, one for our ship and 3 for each of the shots if they have been fired.

```
744 'ROUTINE OF COLLISION OF SHOTS
745 '---------------------------------------------- -------

746 'CASE 1 check if there is any collision
747 '-----------------------------------------
750 if peek (27112)> 0 then | colsp, 7: if col% <32 then a dir = 27113: goto 820
760 if peek (27128)> 0 then | colsp, 8: if col% <32 then a dir = 27129: goto 820
770 if peek (27144)> 0 then | colsp, 9: if col <32 then a dir = 27145: goto 820

780 'CASE 2 check if the shot has left screen
790 '---------------------------------------------- ----------
800 dc = dc mod 3 +1: dir = ddisp (dc): | peek, dir, @ d%: if d% <- 10 THEN poke dir-1.0: |
POKE, dir, 200: nd = nd- 1
810 return:

820 'continuation of the case 1 on collision
825 '---------------------------------------------- ------------
830 'disable the shot
831 'sprite 6 is used to clear the shot, just
Poke dir-840 1.0: na = nd-1: | PRINTSP, 6, peek (dir), peek (dir + 2): poke dir, 255
850 'now process the enemy as may be hard or soft
860 if col> = hard then return
870 'enemy reach soft type. we kill, assigning a sequence of death and putting its status to not
collide more
871 'animation sequence 4 is an animation of "Death", an explosion
880 if col> = soft then a | SETUPSP, cabbage, 7.4: | SETUPSP, col, 0, & x101: return
890 return
```

As you can see, the routine begins by checking if each shot is active looking in the direction of memory your status byte, for just then check if it collides. All this has a high cost. It works, but we can do it more quickly if we use COLSPALL, as we shall see

6.7.2 How to program a multiple shot COLSPALL

Now let's see the advantage of using COLSPALL, much faster because we will not have to invoke multiple times COLSP. The only major recommendations are:

- Our sprite is greater than our shots, so COLSPALL check it before the shooting
- We have set COLSP to just check the list of sprites that are enemies and are necessary to collide, using COLSP 32, start, end

Before starting the game cycle we define our variables
col% = 32: sp = 32%: | COLSPALL, sp @% @ col%

In the game cycle will:

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

With this line we know if there is collision, because then the variable sp is <32
In addition, if 31 is our ship (they have given us) and if not, then certainly one of our shots has reached an enemy ship and go to the routine located on line 770

The processing routine shot collision will be something like:

```
769 'routine shot collision ---------------------------------
770 dir = ddisp (sp% -6): 'first stop the shot and then act according to the type of enemy
771 'sprite 6 is erase, to delete the shot
Poke dir-775 1.0: na = nd-1: | PRINTSP, 6, peek (dir), peek (dir + 2): poke dir, 255
777 col if%> = hard then return
778 'sequence 4 is an animation of "Death", an explosion
If col 780%> = soft then a | SETUPSP, col%, 7.4: | SETUPSP, col%, 0, & x101: return
785 return
```

In short, everything is easier. No need to check the status of the shots or need extra COLSP invocations. With a single call to COLSPALL we know who has collided, and who has collided.

6.8 Animation sequence table

The animations are usually composed of an even number of frames, although this is not a strict rule. Think for example in the simple animation of a character with just two frames: open and closed legs. Are two frames. Now think of an improved animation, with an intermediate phase movement. This involves creating the sequence: intermediate-closed-open-intermediaries and back again. As you can see is an even number, are four

Animated sequences are lists 8BP 8 frames, can not have more, but you can always make shorter sequences.

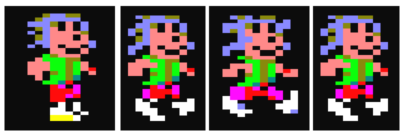
The frames of an animation sequence are the memory addresses which are assembled images which are composed, may be different in size, but usually are alike. If half of the sequence you enter a zero, the meaning is that the sequence is over. Here's an example in assembly language but also you can create using the command | SETUPSQ

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

BASIC equivalent using the library is 8BP

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

Note that in BASIC require knowing the different memory addresses in which each image has been assembled. This we can see from the menu WinAPE Assemble-> symbols

Assembly directly using tags each image, so it is more "understandable". Rarely has the assembler is more fáil understand that the BASIC !!!

This is an animation of 3 different frames but to be fluid before starting again have to go through the "intermediate" frame again, so that in the end are 4 frames:



and back again

Fig. 25 animation sequence

If you wanted to make a sequence of more than 8 frames you could simply string together two consecutive sequences and when the character reached the last frame of the first sequence using the command | SETUPSP to assign the second sequence

Animation sequences are assembled from address 33500, so that it has 500 bytes before reaching the address 34000 that is where graphics are assembled. Therefore You have space to store up to 31 animation sequences.

Each sequence corresponding addresses stores 8 8 frames memory, this is 16 bytes. 500 bytes there is room for 31 sequences and left over 4 bytes.

31 * 16 = 496

Your animation sequences file may look like this:

```
org 33500;
; ================================================ ====================
; 31 animation sequences (500bytes) 8 frames
; ================================================
; It must be a non-variable and fixed table
; each sequence contains the addresses of frames of animation cyclic
; each sequence are 8 directions image memory
; even number because the animations are usually an even number
; zero means end of sequence, but always
; spend eight words / sequence
; to find a zero starts again.
; if there is no zero frame 8 after it begins again
; in total 31 different sequences fit (have 500 bytes)
; SEQUENCES:
; the zero sequence is that no sequence.
; We start from sequence 1

; -------------- Character animation sequences --------
_SEQUENCES_LIST
dw MONTOYA_R0, MONTOYA_R1, MONTOYA_R2, MONTOYA_R1,0,0,0,0   ;1
dw MONTOYA_UR0, MONTOYA_UR1, MONTOYA_UR2, MONTOYA_UR1,0,0,0,0; 2
dw MONTOYA_U0, MONTOYA_U1, MONTOYA_U0, MONTOYA_U2,0,0,0,0; 3
dw MONTOYA_UL0, MONTOYA_UL1, MONTOYA_UL2, MONTOYA_UL1,0,0,0,0; 4
dw MONTOYA_L0, MONTOYA_L1, MONTOYA_L2, MONTOYA_L1,0,0,0,0; 5
dw MONTOYA_DL0, MONTOYA_DL1, MONTOYA_DL2, MONTOYA_DL1,0,0,0,0; 6
dw MONTOYA_D0, MONTOYA_D1, MONTOYA_D0, MONTOYA_D2,0,0,0,0; 7
dw MONTOYA_DR0, MONTOYA_DR1, MONTOYA_DR2, MONTOYA_DR1,0,0,0,0; 8

; -------------- Soldier animation sequences --------
dw SOLDADO_R0, SOLDADO_R2, SOLDADO_R1, SOLDADO_R2,0,0,0,0; 9
dw SOLDADO_L0, SOLDADO_L2, SOLDADO_L1, SOLDADO_L2,0,0,0,0, 10
```

The 8BP library gives you a command called | SETUPSQ with which you can create animation sequences from BASIC. This command is really what makes entering data in memory addresses intended to sequences (from 33500 to 34000). If you believe and you assemble and saved in the image file you save having to create them from BASIC and therefore save lines of BASIC.

6.9 Death sequences

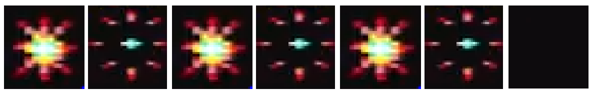
The 8BP library allows you to "death sequences" that are sequences that at the end of roam around, the sprite becomes inactive. This is indicated by a simple "1" as the value of the memory address of the final frame. These sequences are very useful to define explosions enemies are animated | ANIMA or | ANIMAALL. After catch up with your shot, you can associate a death animation sequence and subsequent cycles of the game will go through the various stages of animation of the explosion, and when you reach the last pass inactive, not in print anymore. This step-idle is done automatically, so you have to do is simply check the collision of your shot with enemies and if it collides with any of you change the state with SETUPSP so you can not collide more and assign the animation sequence death, also with SETUPSP

If you use a sequence of death, do not forget that the last frame before finding the "1" is a completely empty, so remove all traces of the explosion.

Example death sequence:

```
dw EXPLOSION_1, EXPLOSION_2, ExPLOSION_3,1,0,0,0,0
```

an interesting effect is to repeatedly go through several frames before finishing with a black frame that serves to erase

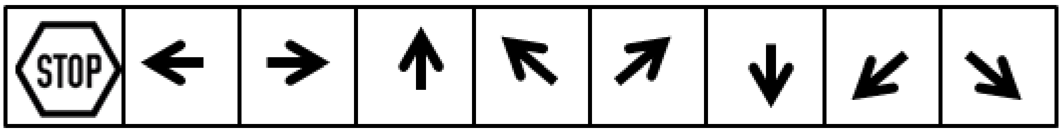


```
dw EXPLOSION_1, EXPLOSION_2, EXPLOSION_1, EXPLOSION_2, EXPLOSION_1,
EXPLOSION_2, ExPLOSION_3,1
```

6.10 Macrosequences animation

This is an "advanced" feature. From the V25 version is available the possibility of defining a "macrosequence" which is nothing more than a sequence consisting of sequences. Each of the sequences of animation is animation constituents to be carried out in a particular direction. Thus, when animating a sprite with AUTOALL automatically change your animation sequence without having to do anything.

The macrosequences are numbered starting at 32. It is very important to place the sequences within the macrosequence in the correct order, ie, the first sequence should be for when the character is still, the next to when he goes to the left (Vy=0, Vx<0), the following to the right (Vy=0, Vx>0), etc., in the following order (be careful because it is easy to make a mistake):

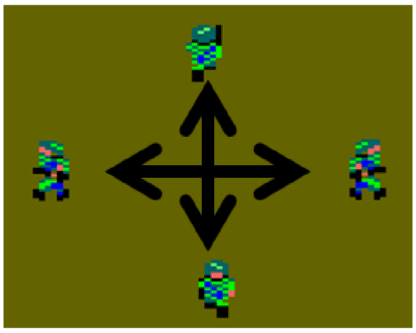


If the sequence assigned to the position is still zero, then simply it encouraged with the last assigned sequence.
The macrosequences can not be created from BASIC, you must specify them in the sequences_tujuego.asm file, which then is an example:

```
org 33500;
; ==================================================
; animation sequences
; ==================================================
; each sequence contains the addresses of frames of animation
; cyclical
; each sequence are 8 directions image memory
; even number because the animations are usually an even number
; zero means end of sequence, but always spent eight words; per
sequence
; to find a zero starts again.
; if there is no zero frame 8 after it begins again

; the zero sequence is that no sequence.
; We start from sequence 1

; ------------- -------- Animation sequences
_SEQUENCES_LIST
dw SHIP, 0,0,0,0,0,0,0; 1
dw joe1, JOE2,0,0,0,0,0,0; 2 UP JOE
dw joe7, JOE8,0,0,0,0,0,0, 3 DW JOE
dw JOE3, JOE4,0,0,0,0,0,0; 4 R JOE
dw JOE5, JOE6,0,0,0,0,0,0; 5L JOE

_MACRO_SEQUENCES
; -------- --------------------- MACRO SEQUENCES
; are groups of sequences, one for each direction
; the meaning is:
; still, left, right, up, up-left, up-right, down, down-left, down-
right
; They are numbered from 32 onwards
db 0,5,4,2,5,4,3,5,4, 32 sequences joe JOE soldier
```

With this definition of sequences we can make a simple game that allows joe move around the screen without control your animation sequence. We assign the sequence 32 and altering the speed, ANIMA command (invoked from within PRINTSPALL) is responsible for changing the animation sequence if its speed denotes a change of direction. That if we need to move the sprite invoke |AUTOALL, since pressing the controls do not change their coordinates but its speed and |AUTOALL update the coordinates of the sprite according to their speed.

```
10 MEMORY 25999
20 MODE 0: INK 0.12
30 ON BREAK GOSUB 280
40 CALL & 6B78
50 DEFINT az
111 x = 36 y = 100
120 | SETUPSP, 31.0, & X1111
130 | SETUPSP, 31,7,2: | SETUPSP, 31,7,32
140 | LOCATESP, 31, and x
160 | setLimits, 0,80,0,200
161 | PRINTSPALL, 0,1,0
190 'starts game cycle
199 vy = 0: vx = 0
200 IF INKEY (27) = 0 THEN vx = 1: GOTO 220
210 IF INKEY (34) = 0 THEN vx = -1
220 IF INKEY (69) = 0 THEN vy = 2: GOTO 240
230 IF INKEY (67) = 0 THEN vy = -2
240 | SETUPSP, 31.5, vy, vx
250 | AUTOALL: | PRINTSPALL
270 GOTO 199
280 | MUSICOFF: MODE 1: 0.0 INK: PEN 1
```



7 Your first single game

You already have the knowledge to try a first step in creating video games. To do this we will see a simple example of a soldier that you are controlling, making him walk right and left across the screen

Suppose we have edited a soldier, thanks to SPEDIT. And we have built their animation sequences, which have been with the identifier 9 and 10 for directions right and left respectively movement.
The two animation sequences have created the well from the file secuencias.asm or in basic with the command | SETUPSQ (which I have not included in this list)

```
10 MEMORY 25999
20 MODE 0: DEFINT AZ: CALL & 6B78: 'install RSX
25 Call & BC02: 'restores default palette case
26 ink 0.0: 'black background
30 FOR j = 0 TO 31: | SETUPSP, j, 0, & X0: NEXT: 'reset sprites
40 | setLimits, 12,80,0,186: 'establish the limits of the game screen
50 x = 40 y = 100: 'character coordinates
51 | SETUPSP, 0,0, & 1: 'status Character
52 | SETUPSP, 0,7,9 'animation sequence assigned to start
53 | LOCATESP, 0, y, x, 'put the sprite (without printing even)

60 'cycle game
70 GOSUB 100
80 | PRINTSPALL, 0.0
90 goto 60

99 'routine movement character -------------
100 IF INKEY (27) = 0 THEN IF dir <> 0 THEN | SETUPSP, 0,7,9: dir = 0:
return ELSE | ANIMA, 0: x = x + 1: GOTO 120
110 IF INKEY (34) = 0 THEN IF dir <> 1 THEN | SETUPSP, 0,7,10: dir =
1: return ELSE | ANIMA, 0: x = x-1
120 | LOCATESP, 0, y, x
130 RETURN
```

With this list you already have a minigame that lets you control a soldier and do scamper horizontally. Note that if walking towards the left surpass the minimum value of the limit established setLimits, it will produce the "clipping" of the character, showing only the part that lies within the area of play allowed

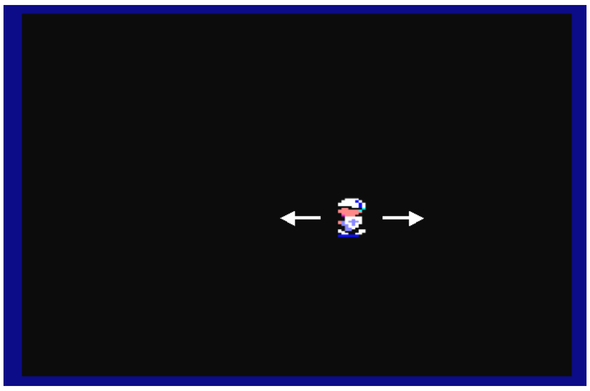


Fig. 26 A simple game

8 Games screens: layout

8.1 Using the definition and layout

Often you want your games consist of a set of screens where the character must collect treasures and avoid enemies in a maze. In these cases is the use of a matrix which define the building blocks of every "labyrinth" or also called "layout" screen essential

8BP in the library you have to do a very simple mechanism, which also provides a function of collision for you to check if your character has moved to an area occupied by a "brick" area.

In 8BP a layout is defined by a matrix of 20x25 "blocks" of 8x8 pixels, which can be busy or not. That is, there are many blocks as has the character display in mode 0.

To print a layout on screen have ordained command:

| LAYOUT, <and>, <x>, @ string

This routine prints a row of sprites to build the layout or "labyrinth" of each screen. The matrix or "map layout" is stored in a memory area that handles 8BP so that when printing blocks actually are not only printing on the screen, but also those filling the memory area occupied by the layout (20x25 bytes) where each byte represents a block.

Coordinates and x are passed in character format, ie

    and takes values [0.24]
    x takes values [0,19]

Printing blocks function | LAYOUT are built with strings and each character corresponds to a sprite that should exist. Thus the block "Z" corresponds to the image that has been assigned the sprite 31. The block "Y" corresponds to the image that has been assigned the sprite 30, and so on

Print sprites are defined with a string, whose characters (32 possible) represent one of the sprites by following this simple rule, where the only exception is the white space that represents the absence of sprite.

Character	Sprite id	ASCII code
""	ANY	32
";"	0	59
"<"	1	60
"="	2	61
">"	3	62
"?"	4	63
"@"	5	64
"TO"	6	65
"B"	7	66
"C"	8	67
"D"	9	68

"AND"	10	69
"F"	eleven	70
"G"	12	71
"H"	13	72
"I"	14	73
"J"	fifteen	74
"K"	16	75
"L"	17	76
"M"	18	77
"N"	19	78
"OR"	twenty	79
"P"	twenty-one	80
"Q"	22	81
"R"	2. 3	82
"S"	24	83
"T"	25	84
"OR"	26	85
"V"	27	86
"W"	28	87
"X"	29	88
"Y"	30	89
"Z"	31	90

Table 4 correspondence between characters and Sprites for the command | LAYOUT

The @string is a string variable. You can not go straight chain. That is to say, it would be illegal to something like:

| LAYOUT, 1.0, "ZZZ YYY"

The correct is:
string = "YYY ZZZ"
| LAYOUT, 1.0, @ chain

Be careful that the chain is not empty, otherwise the computer may crash !!. You must also precede the symbol "@" in the variable of type string so that the library can go to memory address where the string is stored and so to cross it, printing one by one the corresponding sprites.

You must keep in mind that the blanks mean absence of sprite, ie, at positions corresponding to the spaces nothing prints. If there was something in that position previously, it will not be deleted. If you want to delete you need to define yourself delete a sprite 8x8, where are all zeros.

Although you use sprites to print the layout, just after printing can redefine the sprites | SETUPSP and assign images of soldiers, monsters or whatever you want, ie, the layout is "supported" in the mechanism of sprites to print but not you limit the number of sprites, as you have 32 to be what you want right after printing the layout

To detect collisions with the layout offers on the function:

| Colay <threshold ASCII>, <sprite number>, @ collision%

Given a sprite and depending on your coordinates and size, this function will find out if is colliding with the layout and will alert through the variable collision%, which must be previously defined. And not only must be previously defined, but you must put the "%" to indicate that it is an integer variable, even though you're using DEFINT AZ

The <threshold ASCII> parameter is optional and helps the collision command does not consider those ASCII codes below that threshold. The default is 32 (which is for the blank). To understand this we must take into account the correspondence between ASCII and Sprites values shown in the table above. For example, if we as a threshold 69 (code "E" sprite 10), then the sprites 9, 8, 7, 6, 5, 4, 3, 2.1, and 0 will not be "colisionables" so if your character goes above simply will not be detected collision.

Just you need to invoke colay parameter threshold once, as successive invocations have already factored that threshold.

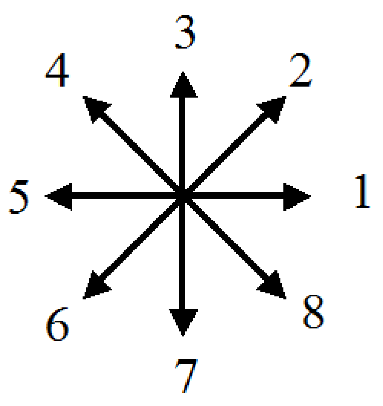
Example of use:
col = 0%
| Colay, 0, @ col%

If there is no collision, the variable will take the value zero. If there is a collision, it will take the value 1. There collision if the sprite collides with some element of the layout different from blank (""), whose ASCII code is 32. If using the threshold, there would be a collision if the element layout has a higher threshold defined ASCII.

Let's see an example of creating a layout and movement of a character within the layout, correcting its position if you have collided.

8.2 Example game with layout

We will evolve a little game presented in the previous chapter, with regard to the control of the character. This time we will use to Montoya as an example, which has 8 animation sequences, each to move in a different direction. A animation sequences we have assigned a number ranging from 1 to 8.



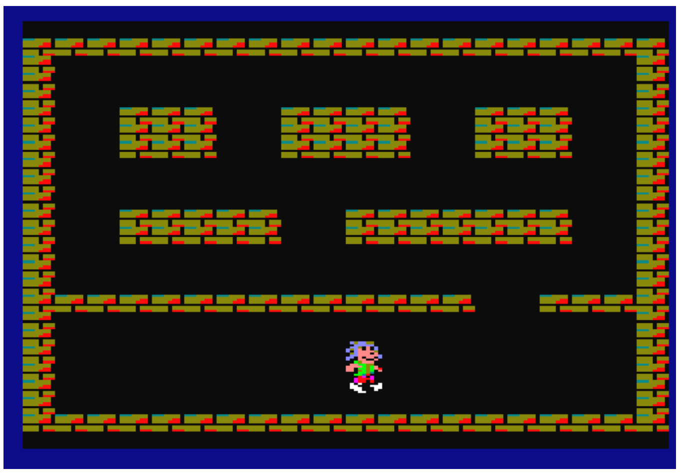


Fig. 27 Using the layout in a game

In the control routine of the character we have included collision with the layout. Depending on the direction in which we move, we modify the "new" (n, xn) coordinates and invoke the function collision with layout | colay, 0 to check if the sprite 0 (our character) has collided. If you have collided, we correct the coordinates (one or both) to let in a position without collision before printing again

```
10 MEMORY 25999
20 MODE 0: DEFINT AZ: CALL & 6B78: 'install RSX
25 Call & BC02: 'restores default palette case
26 ink 0.0: 'black background
30 FOR j = 0 TO 31: | SETUPSP, j, 0, & X0: NEXT: 'reset sprites
40 | setLimits, 0,80,0,200: 'establish the limits of the game screen
50 dim c $ (25) for i = 0 to 24: $ c (i) = "": next
C $ 100 (1) = "zzzzzzzzzzzzzzzzzzzz"
C $ 110 (2) = "Z Z"
C $ 120 (3) = "Z Z"
C $ 125 (4) = "Z Z"
C $ 130 (5) = "Z Z ZZZ ZZZ ZZZZ"
C $ 140 (6) = "Z Z ZZZ ZZZ ZZZZ"
C $ 150 (7) = "Z Z ZZZ ZZZ ZZZZ"
C $ 160 (8) = "Z Z"
C $ 170 (9) = "Z Z"
C $ 190 (10) = "Z Z"
C $ 195 (11) = "Z ZZZZZ ZZZZZZZ Z"
C $ 200 (12) = "Z ZZZZZ ZZZZZZZ Z"
C $ 210 (13) = "Z Z"
C $ 220 (14) = "Z Z"
C $ 230 (15) = "Z Z"
C $ 240 (16) = "zzzzzzzzzzzzzz ZZZZ"
C $ 250 (17) = "Z Z"
C $ 260 (18) = "Z Z"
C $ 270 (19) = "Z Z"
C $ 271 (20) = "Z Z"
C $ 272 (21) = "Z Z"
C $ 273 (22) = "Z Z"
C $ 274 (23) = "zzzzzzzzzzzzzzzzzzzz"
300 GOSUB 550: 'prints the layout
310 x = 40: xn = xa: ya = 150: n = ya: 'character coordinates
311 | SETUPSP, 0,0, & x111 'collision detection with sprites and
layout
312 | SETUPSP, 0,7,1 'sequence = 1
320 | LOCATESP, 0, ya, xa: 'put the character (without printing)
325 cl% = 0: 'declare the variable collision, explicitly whole (%)

330 'cycle game ----- ---------------------
340 GOSUB 1500: 'routine movement keyboard and character reading
350 | PRINTSPALL, 0.0
Goto 360 340

550 'routine print layout ----------------------------------------
560 FOR i = 0 TO 23: | LAYOUT, i, 0, @ c $ (i): NEXT
570 RETURN
```

```
1500 ------------- character movement routine
1510 IF INKEY (27) <0 GOTO 1520
1511 IF INKEY (67) = 0 THEN IF dir <> 2 THEN | SETUPSP, 0,7,2: dir =
2: GOTO 1533 ELSE | ANIMA, 0: xn = x + 1: n = ya-2: GOTO 1533
1512 IF INKEY (69) = 0 THEN IF dir <> 8 THEN | SETUPSP, 0,7,8: dir =
8: GOTO 1533 ELSE | ANIMA, 0: xn = x + 1: n = and + 2: GOTO 1533
IF 1513 dir <> 1 THEN | SETUPSP, 0,7,1: dir = 1: GOTO 1533 ELSE |
ANIMA, 0: xn = x + 1: GOTO 1533
1520 IF INKEY (34) <0 GOTO 1530
1521 IF INKEY (67) = 0 THEN IF dir <> 4 THEN | SETUPSP, 0,7,4: dir =
4: GOTO 1533 ELSE | ANIMA, 0: xn = x-1: n = ya-2: GOTO 1533
1522 IF INKEY (69) = 0 THEN IF dir <> 6 THEN | SETUPSP, 0,7,6: dir =
6: GOTO 1533 ELSE | ANIMA, 0: xn = x-1 and n = + 2: GOTO 1533
IF 1523 dir <> 5 THEN | SETUPSP, 0,7,5: dir = 5: GOTO 1533 ELSE |
ANIMA, 0: xn = xa-1: GOTO 1533
1530 IF INKEY (67) = 0 THEN IF dir <> 3 THEN | SETUPSP, 0,7,3: dir =
3: GOTO 1533 ELSE | ANIMA, 0: n = ya-4: GOTO 1533
1531 IF INKEY (69) = 0 THEN IF dir <> 7 THEN | SETUPSP, 0,7,7: dir =
7: GOTO 1533 ELSE | ANIMA, 0: n = and + 4: GOTO 1533
1532 RETURN
1533 | LOCATESP, 0, n, xn: ynn = n: | colay, 0, @ cl%: IF cl% = 0 THEN
1536
1534 n = ya: | POKE, 27001, yn: | colay, 0, @ cl%: IF cl% = 0 THEN
1536
1535 xn = x: n = ynn: | POKE, 27001, yn: | POKE, 27003, xn: | colay,
0, @ cl%: IF THEN cl% = 1 and n = ya: | POKE, 27001, yn
1536 and a = n: x = xn
1537 RETURN
```

8.3 How to open a gate in the layout

If you want your character can pick up a key and open a gate or generally remove a portion of the layout to allow access, you have to do is two steps:

1) Having defined a 8x8 sprite delete where all are zeroes. Using | LAYOUT print it in the positions you want

2) Then using again | LAYOUT, print it places where you've deleted. So the map layout will be with the character "" in those positions and function of collision with the layout will be zero

In the game "mutant montoya" this technique is used to open the castle gate, so as to open the gates leading to the princess

The following example illustrates the concept, opening a gate at coordinates (10, 12) of a size of 2 blocks, to pick up a key that is defined with the sprite 16.
Nothing else take the key opens the gate and the key is disabled for not evaluate more times the collision with it, ie, the command will return a 32 COLSP from the moment you pick up the key if you ever collide with it.
After opening the gate, if you move the character to the place occupied by said gate, collision with layout will result 0

```
----- This part is inside the loop logic ----
6410 | PRINTSPALL, 1.0
6411 | COLSP, 0, @ cs%: IF cs% <32 THEN IF cs%> = 15 then a GOSUB 6500
(... Further instructions...)
----------------------------------

Routine -------- ----------------- gate opening
6499 'it is found that your collision either with the key, which is
the sprite 16
6500 cleared $ = "MM": spaces $ = "": 'erase the sprite has been
defined as "M" (M is the sprite 18 in the "language" command | LAYOUT)
6501 if cs% = 16 then a | LAYOUT, 10.12, @ cleared $: | LAYOUT, 10.12,
$ @ spaces: | SETUPSP, 16,0,0
6502 return
```

8.4 A comecocos: LAYOUT bottom

Then we'll see an example using layout and overwriting, for which sets a threshold that allows ASCII colay command does not consider the collision with the background elements. Specifically as background element used the letter "Y", which corresponds to the sprite id = 30, and ASCII of the "Y" it is 89.

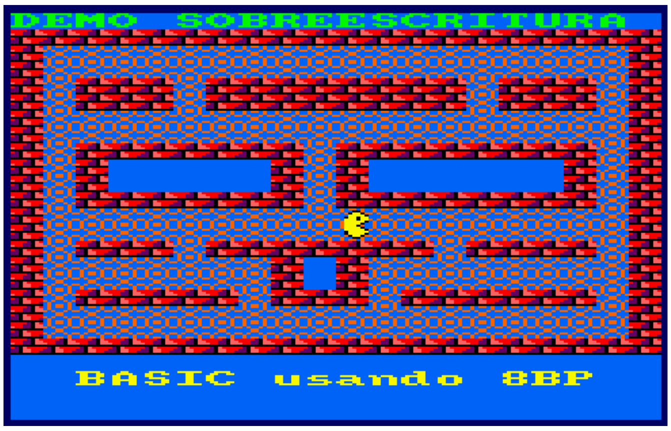


Fig. 28 Layout with a background pattern and overwriting

As you can see in the example just you need to invoke colay parameter threshold once, as successive invocations have already factored that threshold

Another interesting aspect is the management of this example keyboard. It is best to run the colay fewer operations and also gives a nice feeling to move through a corridor and connect with another having two keys pressed simultaneously

```
10 MEMORY 26999
20 MODE 0: DEFINT AZ: CALL & 6B78: 'install RSX
21 on break GOSUB 5000
25 Call & BC02: 'restores default palette case
26 GOSUB 2300: 'pallet with Overwriting
30 FOR j = 0 TO 31: | SETUPSP, j, 0, & X0: NEXT: 'reset sprites
40 | setLimits, 0,80,0,200 'limits of the game screen
45 | SETUPSP, 30.9, & 84d0 'grid background ( "Y")
46 | SETUPSP, 31.9, & 84f2: 'brick ( "Z")
50 dim c $ (25) for i = 0 to 24: $ c (i) = "": next
```

```
C $ 100 (1) = "zzzzzzzzzzzzzzzzzzzz"
C $ 110 (2) = "ZYYYYYYYYYYYYYYYYYYZ"
C $ 120 (3) = "ZYYYYYYYYYYYYYYYYYYZ"
C $ 125 (4) = "ZYZZZYZZZZZZZZYZZZYZ"
C $ 130 (5) = "ZYZZZYZZZZZZZZYZZZYZ"
C $ 140 (6) = "ZYYYYYYYYYYYYYYYYYYZ"
C $ 150 (7) = "ZYYYYYYYYYYYYYYYYYYZ"
C $ 160 (8) = "ZYZZZZZZZYZZZZZZZZYZ"
C $ 170 (9) = "ZYZ ZYZ ZYZ"
C $ 190 (10) = "ZYZ ZYZ ZYZ"
C $ 195 (11) = "ZYZZZZZZZYZZZZZZZZYZ"
C $ 200 (12) = "ZYYYYYYYYYYYYYYYYYYZ"
C $ 210 (13) = "ZYYYYYYYYYYYYYYYYYYZ"
C $ 220 (14) = "ZYZZZYZZZZZZZZYZZZYZ"
C $ 230 (15) = "ZYYYYYYYZ ZYYYYYYYYZ"
C $ 240 (16) = "ZYYYYYYYZ ZYYYYYYYYZ"
C $ 250 (17) = "ZYZZZZZYZZZYZZZZZZYZ"
C $ 260 (18) = "ZYYYYYYYYYYYYYYYYYYZ"
C $ 270 (19) = "ZYYYYYYYYYYYYYYYYYYZ"
C $ 271 (20) = "zzzzzzzzzzzzzzzzzzzz"
C $ 272 (21) = ""
C $ 273 (22) = ""
C $ 274 (23) = ""
300 'we print the layout
310 FOR i = 0 TO 20: | LAYOUT, i, 0, @ c $ (i): NEXT
311 locate 1.1: pen 9: print "DEMO OVERWRITE"
312 locate 3,23: pen 11: print "BASIC using 8BP"
320 | SETUPSP, 0,0, & x01000111 'collision detection with sprites and
layout
330 | SETUPSP, 0,7,1: dir = 1 'sequence = 1 (coco right)
340 x = 20 * 2: xn = xa: ya = 12 * 8: n = ya: 'character coordinates
350 | LOCATESP, 0, ya, xa: 'put the character (without printing)
360 | PRINTSPALL, 0,1,0 'prints sprites
361 cl% = 0: 'collision Variable
362 | colay, 89.0, cl% 'threshold chr $ ( "Y") is 89
400 'GAME BEGINS
401 | MUSIC, 0.5
402 'keyboard reading and collisions.
403 'if we go in the direction H (op), first check if a key is pressed
direction V (qa) and vice versa
404 if dirn <3 then GOSUB 450: GOSUB GOSUB 410 410 else: GOSUB 450
405 | LOCATESP, 0, n, xn: | PRINTSPALL
= 406 and n: x = xn
Goto 407 404

409 'keyboard horizontal direction
410 if INKEY (27) <0 then 430
420 xn = x + 1: Poke 27003, xn: | colay, 0, @ cl%: IF cl% = 0 then if
dir <> 1 then | SETUPSP, 0,7,1: DIR = 1: xn = xa: else return dirn =
1: return
421 xn = xa: Poke 27003, xn: return: 'There collision
430 if INKEY (34) <0 then return
```

```
440 xn = xa-1: Poke 27003, xn: | colay, 0, @ cl%: IF cl% = 0 then if
dir <> 2 then | SETUPSP, 0,7,2: DIR = 2: xn = xa: else return dirn =
2: return
441 xn = xa: Poke 27003, xn: 'There collision
442 return
449 'keyboard vertical direction
450 if INKEY (67) <0 then 480
460 n = ya-2: Poke 27001, n: | colay, 0, @ cl%: IF cl% = 0 then if dir
<> 3 then | SETUPSP, 0,7,3: DIR = 3: n = ya: else return dirn = 3:
return
461 n = ya: Poke 27001, n: 'There collision
480 if INKEY (69) <0 then return
490 n = and + 2: Poke 27001, n: | colay, 0, @ cl%: IF cl% = 0 then if
dir <> 4 Then | SETUPSP, 0,7,4: DIR = 4: n = ya: else return dirn = 4:
return
491 n = ya: Poke 27001, n: 'There collision
492 return

REM 2300 ---------- transparent sprites PADDLE MODE 0 ------------
2301 0.11 INK: light blue REM
1.15 INK 2302: REM orange
2303 2.0 INK: black REM
2304 3.0 INK:
2305 4.26 INK: white REM
INK 5,26 2306:
2307 6.6 INK: REM red
2308 7.6 INK:
2309 8.18 INK: Green REM
2310 9.18 INK:
2311 10.24 INK: yellow REM
2312 11.24 INK:
INK 2313 12.4: REM magenta
INK 2314 13.4:
14.16 INK 2315: REM orange
2316 INK 15, 16:
2317 YELLOW = 10
2420 RETURN

5000 | musicoff
5010 end
```

8.5 How to save memory in your layouts

If your game has many screens and you need to save space you can use many simple techniques to save memory
Imagine there is only one type of "brick" on a screen (a brick or lack thereof). This can be represented by a single bit, so that a byte holds 8 bricks. Since we can not write all ASCII codes because many of them are control, at least you can use half. This "would compact" screen at a ratio of 1: 7 (in the space of 10 screens you'll get 70 screens) and just before invoking | LAYOUT need to make the conversion between your bits (which will be in the form of characters) and characters you expect to receive the command.

This conversion would in the BASIC and even slow we talk about print the screen, which does not require excessive speed.
The same philosophy can be applied if there are only 4 types of bricks (two types and absence). You can use only 2 bits by brick and then put those bits in characters (1 to 128), so you can fit 3.5 blocks per character. In that case the ratio of savings is 2: 7 ie 3.5 times less. In the space of 10 screens you'll get 35 screens.

Another simple and effective solution is to define each layout with less information than necessary, "reducing" possibilities. It would be something like define the layout with less resolution you have. For example, we can think of to define a layout (which measures 20x25) with a 5x6 matrix, thus:

```
NNxNN
xxxxx
xxxMM
Mxxxx
xxMMx
xxxxx
MMxxM
```

Each character in this "microlayout" can then "expand" to something you predefined, for example, an "M" could represent three bricks with grass above and "x" could represent 3x2 blanks each. After expanding the definition of the screen we could have a maze that fills the layout, although logically we reduced the possibilities for layout. With this simple strategy can store 16 "simple" layouts in the space where before could only fit a single layout "complex". In summary, we have multiplied 16 times the storage capacity of layouts.
You can use a strategy less "aggressive" and define layouts simply half space, for example in "microlayouts" 10x12, which reduce the chances less, compared to the original layout of 20x25. And we are doubling the storage capacity. That if you have to program yourself a routine in BASIC to make the conversion between your microlayouts and "authentic" layouts.

9 Recommendations and advanced programming

9.1 Recommendations speed

The BASIC interpreter is running very heavy because not only runs every command it analyzes the number line, parses the command entered, validates their existence, number and type of parameters, their values q¡se find valid ranges (eg PEN 40 is illegal) and many more things. It is the syntactic and semantic analysis of each command what really weighs rather than execution. The case of the RSX commands is no exception. The BASIC interpreter checks the syntax and that weighs heavily, even though they are routines written in ASM, since before invoke the BASIC interpreter has done many things.

Therefore you have to save command executions, programming shrewdly to program logic passes the fewest possible instructions, even if it sometimes involves writing instructions. The use of GOTO is highly recommended, given the high speed of execution, as we shall see later in a comparative table.

A decisive factor when invoking a command is step parameter. The more parameters you have, the more expensive it is their interpretation by the BASIC, even even a ASM routine that is invoked by CALL, as the CALL command is still BASIC and before accessing routine in ASM, the number and type analyzes irretrievably parameter.

To evaluate the cost of executing a command you can use the following program. It will also serve to evaluate the performance of new functions in assembly that you incorporate the 8BP library if you want to.

```
10 MEMORY 25999
20 DEFINT az
30! = TIME
40 FOR i = 1 TO 1000
50 <here you put a command, such as PRINT "A">
60 NEXT
70 b! = TIME
80 PRINT (b! -a!)
900 c! = 1000 / ((b! -a!) * 1/300)
100 PRINT c, "fps"
110 d! = C! / 60
120 PRINT "can run" d !, "sweep commands (1/50 sec)"
125 rem if you leave the 50 empty line, it will take 0.47 milliseconds
130 PRINT "command takes"; ((b -a) / 300 to 0.47!) "Milliseconds"
```

Note: For experts in assembly language, you must keep in mind that if pretendeis measure the execution time of a routine that internally disables interrupts (using the instructions DI, EI) the time elapsed during deactivation is not measurable with this program BASIC. 8BP commands do not disable interrupts and are all measurable.

Let's see then the result of the performance of some commands. I must say that it is faster to execute a direct call to the address memory (CALL & XXXX) to invoke the RSX command. The following table obviously the smaller the result (expressed in milliseconds), faster is the command. The table presented here must have it in every present moment and take your programming decisions based on it.

Command	more	Commentary
PRINT "A"	3.63	Very slow. Definitely do not use it except promptly to change the number of lives but do not print in a game score for every enemy you kill
LOCATE 1.1 PRINT points	24.87	Place the text cursor LOCATE and print the variable value luna "points" is very expensive. If you upgrade points do it only occasionally and not on each game cycle
REM hello	0.20	Comments consume
' Hello	0.25	Save 2 bytes of memory but is slower !!
GOTO 60	0.196	Very fast!!! Faster even than REM. Use this pitiless command, use it !!!
| NOP	1.17	It does nothing, it's just a RET assembler. It gives us an idea of what it takes for the BASIC parser. This is the minimum time it will take to execute any command RSX
| NOP, 1,2,3	2.15	Parameter passing is costly
| LOCATESP, i, y, x	2.47	It is very fast considering the maximum achievable speed (2.15 NOP is what it takes 3 parameters) but if you do not use negative coordinates is better to use the BASIC POKE command. S want to locate a squadron of enemies is better to use AUTOALL and MOVERALL, because if for each enemy use a LOCATESP you be investing too long.
POKE & XXXX, value	0.71	Very fast! Use it to update the coordinates of the sprites
POKE d, value	0.85	Very fast considering that must translate the variable "d"
| POKE, & xxxx, value	1.87	Allows negative numbers only if you update a coordinate (X or Y) is better than LOCATESP
X = PEEK (& xxxx)	0.93	Very fast!
X = INKEY (27)	1.12	Very fast. Suitable for video games but must use it wisely as recommended in this book.
IF x> x = 0 THEN 50	1.42	Each weighs IF, we must try to save them because logic game will have many
If INKEY (27) = 1 then x = 1	1.75	Good combined use. It is faster to do b = INKEY (27) and then the IF ... THEN

A = A + 1:
IF A> 4 Then A = 0
 	
Versus
 	
A = A MOD 3 + 1
	2.6
	
	vs
	
	1.84
		This is a very clear example of how we schedule. It is much better to use the second option.
		On the other hand the use of MOD must be done with caution. If we do:
		A = (A + 1) MOD3
		it costs 2 ms and yet get the same (more or less). Parentheses cost

:	0.05	But it does not save much faster to use ":" rather than a new line number, and if you apply it just often saving significantly
| PRINTSP, 0,10,10	5.4	One 14 x 24 sprite. Eye, if you go to print multiple offsets much print all sprites hit with PRINTSPALL

CALL & xxxx, 0,10,10		It PRINTSP equivalent and is faster but less readable
| PRINTSPALL, 0.0 (6 sprites 14x24)	20.6	Printing 6 large sprites, you can almost all screen each sweep as 20.6 x 50 = 1030 ms and a second is 1000 ms Using "CALL" takes exactly 20.0 ms and therefore can with 6

| PRINTSPALL, 0.0
 	
(32 sprites 12x16)
	70.4	This means about 14fps at full load of sprites. What it takes is
		
		T = 3.3 + N x 2.1
		
		That is, a 2ms by sprite and a fixed cost of 3.1ms. This fixed cost is the cost of parsing BASIC adding to tour the table looking sprites which must be printed. If the parameters are ignored (and possible values of the last invocation would be taken), they are saved in the fixed 0.6ms, ie:
		
		T = 2.6 + 2.1 x N

| PRINTSPALL, N, 0,0
(No active sprite)
 	
N = 0
N = 10
N = 31
	2.6
	4.3
	5.9
		Cost of ordering the sprites:
		When N = 0, there being no sprite to print, the function must scan the table sequentially sprite. But cross it in an orderly manner is more expensive, as evidenced by the time consumed by increasing N. The time difference (5.9 -2.6 = 2.5ms) is what it takes to sort all sprites

| Colay, 0, @ x%	3.44	Acceptable. Use only with the character, not enemies or slow anger game. If the character measures multiples of 8 is faster. This example was logically 14x24 and 14 is not a multiple of 8 the greater the sprite takes more
CALL & XXXX, 0, @ x%	2.79	It's like invoke | colay. faster but less readable
GOSUB / RETURN	0.56	Acceptably fast. The test is a routine that only makes return.
| SETUPSP, id, param, value	3.5	OK although POKE is much better. For certain things SETUPSP is more readable and normally be used little in logic so that is not a problem to use it (for example when changing direction a sprite). In the command SETUPSP we see the number of sprite and parameter touched. In POKE not see anything, it is less readable even faster.
FOR / NEXT	0.6	You can use it to scroll through multiple enemies and each move according to the same rule. You should assess whether you can use AUTOALL or MOVEALL for your purposes as a single command would move all you want, which is much better than a loop.
| COLSP, 0, @ c%	4.97	It takes the same regardless of the number of active sprites. This routine will have to invoke the each cycle of the logic of your game, so they are almost 5ms which necessarily have to devote to this. A strategy such as

		running | COLSP only half the frames would not work because it requires incrementing a counter and check it with an IF, so although you save 2.5ms spend on the checks. If you have a ship or character and several shots is much more efficient to call upon COLSPALL instead of invoking several times COLSP
| animall	3.0	It is expensive but there is a way to invoke jointly by invoking | PRINTSPALL by a parameter that makes this function is invoked before printing the sprites. This saves the BASIC layer, that is what consumes send the command, which is 1.17ms (command | NOP). Therefore we can say that this command normally consume something less than 2ms
| AUTOALL	4.25	It is expensive but you can move while the 32 sprites
| MOVERALL, 1.1	5.14	It is expensive but you can move while the 32 sprites
SOUND	10	The sound command is "blocking" as the buffer fills 5 ratings. This means that your logic BASIC should not stringing more than 5 commands SOUND or stop until a note ends. In any case if you decide to use should be carefully time-consuming and execution (10 ms is a lot)

IF a> 1 and a> 2 THEN a = 2
 	
Versus
 	
IF a> 1 THEN IF at> 2 THEN a = 2
	2.52
	
	vs
	
	2.39
		A simple way to save 0.13 ms
		
		In everything programes note these details, every saving is important

Table 5 Relationship runtimes some instructions

When you make your program, try to see the cost of the commands you use and minimizes the best use of the CPU. For example if you can avoid going through an IF inserting a GOTO, it is always preferable. Or make use instead of LOCATESP POKE unless you use negative coordinates. When you lack speed and need a little more quickly and uses CALL use RSX leaves

Note that all the logic of your program must accumulate as much 20ms if you want to synchronize with sweeps screen, but the gameplay will remain well above acceptable, perhaps even 50ms, depending on the type of game. It is very difficult that you achieve 20ms unless there is not just sprites. But a target of 50ms is reasonably fast. If your game takes 50ms then generate 20fps (frames per second) and will be very acceptable. And if you get 40ms (25fps) even a professional will achieve smooth movement.

More important recommendations:

- Use DEFINT AZ at the beginning of the program. Performance will improve a lot. This is almost mandatory. This command deletes the variables that exist before and forces all new variables are integers unless otherwise indicated with modifiers like "$" or "!" (See Reference Guide Amstrad BASIC programmer)

- Delete Blanks
- Remove any comment on the game logic and if you leave it any REM (faster), do not use the quotation mark. If you use the quotation mark is to save 2 bytes of memory, and is suitable to discuss the rest of the program (initializations and stuff). If you want to comment on parts of logic you can do the following:

```
If x> 23 GOSUB 500
...
499 rem for this line is not passed and so this routine comment
500 if x> 50 THEN ...
...
550 RETURN
```

- Compacting in a line all that is logic game and can be compacted. For example, the following two lines:

```
10 if E1D = 0 then e1x = e1x + 1: if e1x> = 70 then a E1D = 1: |
SETUPSP, 1,7,10
20 if E1D = 1 then e1x = e1x-1: if e1x <= 4 then a E1D = 0: | SETUPSP,
1,7,9
```

You can change them (note the double else to apply to the first if):

```
10 if E1D = 0 then e1x = e1x + 1: if e1x> = 70 then a E1D = 1: |
SETUPSP, 1,7,10 else else if E1D = 1 then e1x = e1x-1: if e1x <= 4
then a E1D = 0: | SETUPSP, 1,7,9
```

- Avoid unnecessary line execution logic. This recommendation is the same as above but with a more elegant style of reading. Here is an example. With 30 ELSE we have avoided going through the line 20 in many cases

  In this example "E1D" is the address of the sprite 1 "e1x" is x coordinate

```
10 if E1D = 0 then e1x = e1x + 1: if e1x> = 70 then a E1D = 1: | SETUPSP,
1,7,10 else 30
20 if E1D = 1 then e1x = e1x-1: if e1x <= 4 then a E1D = 0: | SETUPSP, 1,7,9
30 | LOCATESP, 1, e1y, e1x
```

And now an even better version. We have eliminated unnecessary IF on line 20

```
10 if E1D = 0 then e1x = e1x + 1: if e1x> = 70 then a E1D = 1: | SETUPSP,
1,7,10 else 30
20 e1x = e1x-1: if e1x <= 4 then a E1D = 0: | SETUPSP, 1,7,9
30 | LOCATESP, 1, e1y, e1x
```

- BASIC never synchronized with the scanning screen because it slows down the game. That is, always use | PRINTSPALL, 1.0 instead of | PRINTSPALL, 1.1. If you compile the game with some compiler as "fabacom" then worth synchronize both to set the game speed to 50 frames per second for smoother movements.

- Once you have invoked the STARS command parameters or PRINTSPALL, or other commands 8BP command, the following times not invoke with parameters. The 8BP library has "memory" and use the last scaling parameters you used. This saves milliseconds to cross the layer parsing BASIC interpreter.

- Manages the keyboard (and in general this applies to whatever you do) running the fewest instructions. Here is an example (first wrong and then well done), where as much passes through 4 operations with corresponding IF INKEY $. Run it mentally and you'll see what I say. It is much faster the second

Poorly done (worst case = 8 executions "IF INKEY")

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

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

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

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

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

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

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

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

Well done (worst case = 4 executions "IF INKEY"):

```
1510 if INKEY (27) <> 0 goto 1520

1511 if INKEY (67) = 0 then IF dir <> 2 THEN | SETUPSP, 0,7,2: dir =
2: goto ELSE 1533 | ANIMA, 0: xn = x + 1: n = ya-2: goto 1533

1512 if INKEY (69) = 0 THEN IF dir <> 8 THEN | SETUPSP, 0,7,8: dir =
8: goto ELSE 1533 | ANIMA, 0: xn = x + 1: n = and + 2: goto 1533

IF 1513 dir <> 1 THEN | SETUPSP, 0,7,1: dir = 1: goto ELSE 1533 |
ANIMA, 0: xn = x + 1: goto 1533
```

```
1520 if INKEY (34) <> 0 goto 1530

1521 if INKEY (67) = 0 THEN IF dir <> 4 THEN | SETUPSP, 0,7,4: dir =
4: goto ELSE 1533 | ANIMA, 0: xn = x-1: n = ya-2: goto 1533

1522 if INKEY (69) = 0 THEN IF dir <> 6 THEN | SETUPSP, 0,7,6: dir =
6: goto ELSE 1533 | ANIMA, 0: xn = x-1 and n = + 2: goto 1533

IF 1523 dir <> 5 THEN | SETUPSP, 0,7,5: dir = 5: goto ELSE 1533 |
ANIMA, 0: xn = xa-1: goto 1533

1530 IF INKEY (67) = 0 THEN IF dir <> 3 THEN | SETUPSP, 0,7,3: dir =
3: goto ELSE 1533 | ANIMA, 0: n = ya-4: goto 1533

1531 IF INKEY (69) = 0 THEN IF dir <> 7 THEN | SETUPSP, 0,7,7: dir =
7: goto ELSE 1533 | ANIMA, 0: n = and + 4: goto 1533

1532 return
```

- In games where the logic of the enemies requires the use of a calculation function (such as cosine), precalcula everything and store in an array that you use during the execution of logic. Calculate during the game logic is cost prohibitive in BASIC

- In games of ships it is important not to use negative coordinates and if you want the ships appear around the edges and perceived clipping, reduce the screen a little setLimits

- In games of ships it avoids the checks with the layout. This is almost 3.5ms savings in each cycle of logic. Normally you will not require a layout in a game ships. The scenarios can be simulated with the sprites that have disabled the collision flag and represent craters, space bases, etc. and together with the mottled ground | STARS give the feeling of earth moving. Try to make your ship is the sprite 31 thus pass through "top" of the sprites that simulate the background, as your ship will be printed after

- Use the smallest number of parameters invocations 8BP commands. each parameter is time consuming and every millisecond is important in a game, especially arcade

- Testing alternative versions of the same operation
  A = A + 1: IF A> 4 Then A = 0: REM this consumes 2.6ms
  A = A MOD 3 +1: REM this consumes 1.84 ms

- Simplify: a complex logic is a slow logic. If you want to do something complicated, a complex trajectory, a mechanism of artificial intelligence ... do not do, try to "fake it" with a simpler model of behavior but produce the same visual effect. For example a ghost who is intelligent and chasing you, rather than make smart decisions, do try to take the same direction as your character, without any logic. This will make it appear that it is ready (just an idea)

- Do not use negative coordinates if you need to update the position of your ship or character very quickly. The POKE (the BASIC) command is very fast but only supports positive numbers, like PEEK. in case of use, use | POKE and | PEEK (8BP commands). Reserve the use of | LOCATESP for when you go to change both coordinates and can be positive and negative.

- If you need to check something, do not do at all levels of play. Perhaps enough that you check that "something" every 2 or 3 cycles, without requiring you to check each cycle. To choose when to run something, make use of the "modular artitmética". In BASIC you have the MOD instruction is an excellent tool. For example, to run one of every 5 times you can do: IF cycle MOD 5 = 0 THEN ...

- When programming a multi-shot (a ship that can fire three missiles simultaneously) uses the technique of massive logical and reduces logic. If each frame can only die an enemy, much will reduce the number of instructions executed. Use the command COLSPALL, that even lets you program more efficiently

- Make use of the "death sequences". This will save you instructions to check if a sprite that is exploding has reached its last frame of animation to disable it.

- Overwriting is expensive: if you can make your save game without overwriting milliseconds and colorful ganaras. Use it when you need it, but not without reason

- The animation macrosequences save you BASIC lines because they do not need to check the direction of movement of the sprite. Use them whenever you can.

9.2 Make the most memory

Each screen occupies a considerable memory, if it is a set of screens defined through the layout. In a simple game, each screen occupy a BASIC text of about 800 bytes and a code of about 2000 bytes, so it is difficult to make a game more than 10 screens (remember that you have for your game 26KB)

One of the fundamental things you should do is to reuse code logic instead of retyping enemies on each screen.
The logic of the enemies of each screen you can occupy more than the layout of the screen. In the game "montoya mutant" (performed in BASIC using 8BP), about one third of each screen is layout and the other two thirds are logical. If you need space for more screens it is best to "reuse" logic of enemies between screens, a mode selected. For example an enemy that moves from left to right only requires 3 parameters: where to start (X, Y), where it ends his career on the right (X max) and where it ends left (Xmin). Only you should initialize these 3 values and a GOSUB / RETURN BASIC lines would save on each screen where the enemy appears.

The key is to reuse some BASIC code in other screens, just as you do with the character movement routine. And within each screen if several enemies of the same type reuse code logic of the enemy, writing it only once.

Example: Three soldiers pacing the screen

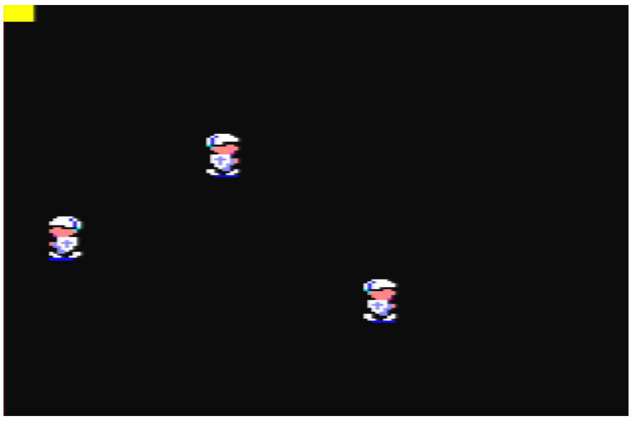


Fig. 29 three soldiers and a single logic routine

```
10 MEMORY 25999
20 MODE 0: DEFINT AZ: CALL & 6B78: 'install RSX
25 FOR j = 0 TO 31: | SETUPSP, j, 0, & X0: NEXT: 'reset sprites
26 | setLimits, 0,80,0,200
30 'parametrization 3 soldiers
40 Dim x (3)
50 x (1) = 10: xmin (1) = 10: xmax (1) = 60: y (1) = 60: Address (1) =
0: | SETUPSP, 1,7,9: | SETUPSP, 1, 0, & x111
60 x (2) = 20: xmin (1) = 15: xmax (2) = 40: and (2) = 100: address
(2) = 1: | SETUPSP, 2,7,10: | SETUPSP, 2, 0, & x111
70 x (3) = 30: xmin (1) = 5: xmax (3) = 50: and (3) = 130: address (3)
= 0: | SETUPSP, 3,7,9: | SETUPSP, 3, 0, & x111
80 for i = 1 to 3: | LOCATESP, i, and (i), x (i): next: 'put sprites

89 '----- MAIN LOOP GAME (GAME CYCLE) ----------------
90 for i = 1 to 3: GOSUB 100: next: 'call at 3 soldiers
91 | PRINTSPALL, 1.0: 'anima and prints 3 Soldiers
95 goto 90
96 'CYCLE END -------------- ------------------------- GAME
99 '------- ----- soldier routine
100 IF direction (i) = 0 THEN x (i) = x (i) +1: IF x (i)> = xmax (i)
THEN direction (i) = 1: | SETUPSP, i, 7.10 ELSE 120
110 x (i) = x (i) -1: IF x (i) <= xmin (i) THEN direction (i) = 0: |
SETUPSP, i, 7,9
120 27003 + i * Poke 16 x (i): 'we put in the new coordinate
130 return
```

And as general recommendations:

- You can overcome the memory limitations by algorithms that generate mazes, or screens without storage. Thus you can make many more screens. This requires creativity, of course, but possible.

- You can reuse the logic of enemies on a screen in another, saving many lines of code. Seize the GOSUB / RETURN mechanism for this, defining common parameters configurable logic. For example a guard that moves from left to right you can place it on many screens at different heights and with different limits in its path without programming again.

- You can also load phased games, so you do not have the game in memory at once. This is a bit annoying to the user belt (464) although it is not for the disk (CPC6128)

- Try to combine layouts consecutive screens. May variations from one to the next screen are only 10 lines of the layout (for example), saving 50% of memory to build (about 400bytes savings).

- If you want to make many screens, you compáctalas using the technique explained in section 8.4

9.3 Technique "massive logic" of sprites

Often you will need to move many sprites, especially in arcade style space or "commando" (the classic capcom 1985).

You could act separately in the coordinates of all the sprites and update them using POKE but be very slow, unfeasible if you want fluidity of movement. The best practice (and simple) is to make combined use of the auto motion and relative motion, which are | AUTOALL and | MOVERALL respectively.

The key to achieving many sprites speed is to use the technique I have dubbed "massive logic". This technique is primarily to implement less logic in each game cycle (what is called "reducing the computational complexity") and for this there are two options:

- Use a single logical that affects many sprites at once (using the flag automatic movement and / or relative)
- But use different logics run only one or a few in each game cycle.

Both options have the same goal: to run less logic in each cycle, allowing all sprites move at once but taking fewer decisions in each cycle of the game. This is called "reduce computational complexity," transforming a problem of order N (N sprites) a problem of order 1 (one logic to execute in each frame).
The key is to determine what logic or logical run in each cycle. In the simplest case, if we execute N sprites just one of the N logical. But in more complex cases we must be smart to determine what logic should run.

9.3.1 Simple example of mass logic

Returning to the example of the 3 soldiers. This time we will run only the logic of a soldier in every game cycle.

For despite this, the x coordinate of each soldier moving forward, we will use the automatic movement flag, instead of updating us.

```
10 MEMORY 25999
20 MODE 0: DEFINT AZ: CALL & 6B78: 'install RSX
25 FOR j = 0 TO 31: | SETUPSP, j, 0, & X0: NEXT: 'reset sprites
26 | setLimits, 0,80,0,200
30 'parametrization 3 soldiers
Dim x 40 (3): x% = 0
50 x (1) = 10: xmin (1) = 10: xmax (1) = 60: y (1) = 60: Address (1) = 0: |
SETUPSP, 1,7,9: | SETUPSP, 1, 0, & X1111: | SETUPSP, 1,5,0: | SETUPSP, 1,6,1
60 x (2) = 20: xmin (1) = 15: xmax (2) = 40: and (2) = 100: address (2) = 1:
| SETUPSP, 2,7,10: | SETUPSP, 2, 0, & X1111: | SETUPSP, 2,5,0: | SETUPSP,
2,6, -1
70 x (3) = 30: xmin (1) = 5: xmax (3) = 50: and (3) = 130: address (3) = 0: |
SETUPSP, 3,7,9: | SETUPSP, 3, 0, & X1111: | SETUPSP, 3,5,0: | SETUPSP, 3.6.1
80 for i = 1 to 3: | LOCATESP, i, and (i), x (i): next: 'put sprites
81 i = 0
89 '----- MAIN LOOP GAME (GAME CYCLE) --------------
90 i = i + 1: GOSUB 100
92 if i = 3 then i = 0
93 | AUTOALL
94 | PRINTSPALL, 1.0: 'anima and prints 3 Soldiers
95 goto 90
96 'CYCLE END -------------- ------------------------ GAME
99 '------- ----- soldier routine
100 | PEEK 27003 + i * 16% @ x: x (i) = x%
101 IF direction (i) = 0 THEN IF x (i)> = xmax (i) THEN direction (i) = 1: |
SETUPSP, i, 7.10: | SETUPSP, i, 6, -1 ELSE return
110 IF x (i) <= xmin (i) THEN direction (i) = 0: | SETUPSP, i, 7,9: |
SETUPSP, i, 6,1
120 return
```

Try it and see that multiplying the speed is almost 3. Each soldier has its own logic, but only run one at each game cycle, greatly easing cycle game.

The only limitation is that running the logic of each soldier one of every 3 times, the coordinate could exceed the limit we have set for two cycles. That means that we should be more careful when setting the limit, making sure to run it ever invades and erases a labyrinth wall of our screen, for example. I will try to explain this problem more precisely:

Suppose we have 8 sprites and our sprite moves in all cycles but only execute logic one of every 8 times.

Imagine a sprite that is in the position x = 20 and we want to move to the position x = 30 and turn around. Consider that the sprite has an automatic movement with Vx = 1. In that case we will check its position when x = 20, x = 28, x = 36. Upon reaching 36 we realize that we have gone !!! and change the speed of the sprite to Vx = -1

As you see the control limits the path is not necessary, unless we have this into account and we set the limit on something that we can control, which will xfinal = Xinicial + n * 8.

This limitation is minuscule when compared to the advantage of many sprites moving at high speed. With some cleverness we can even execute the logic fewer times, so that only one of every two cycles some kind of logic enemies run.

9.3.2 32 sprites moves with massive logic

Now let's look at a simple example to move 32 sprites simultaneously and gently (at 14fps). It is perfectly possible. Only a ghost will make decisions in each cycle, although it will move all the ghosts in all cycles. We can also encourage everyone (associating an animation sequence and using | PRINTSPALL, 1.0) and will continue being soft, but still seem that there is more movement as the flapping wings of a fly (for example) generates much sense of movement

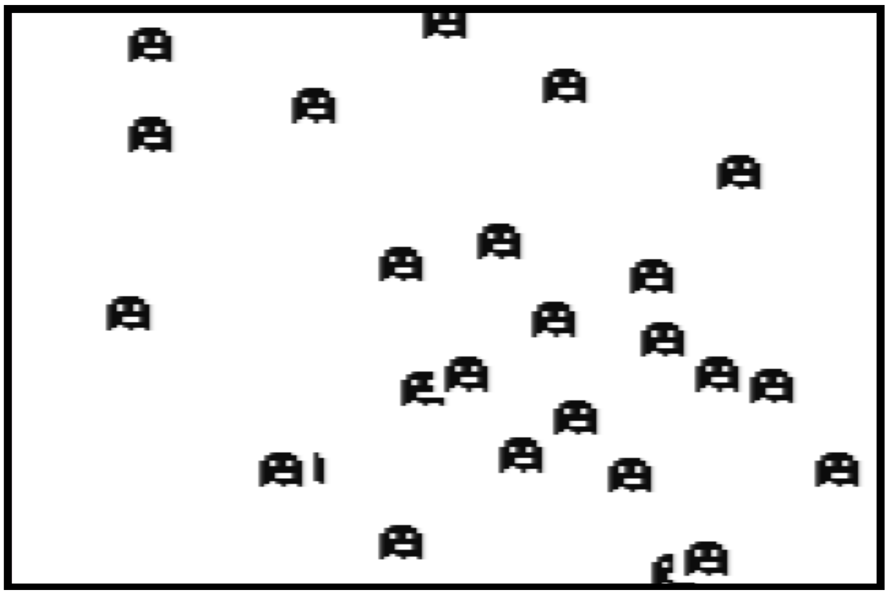


Fig. 30 with massive logic 32 sprites can move simultaneously

What we have done has been to reduce the computational complexity. We started with a problem of "order N", where N is the number of sprites. Assuming that each logic requires three instructions BASIC sprite in principle should be run N x 3 instructions per cycle. With the technique of "massive logic" we transform the problem of "Nth" a problem "order 1". It's called problems of "order 1" that involve a constant number of operations regardless of the size of the problem. In this case we have gone from operations Nx3 BASIC BASIC only 3 operations. This reduction of complexity is the key to high-performance logic massive technique.

```
1 MODE 0
10 MEMORY 25999: CALL & 6B78
20 DEFINT az
25 'reset enemies
30 FOR j = 0 TO 31: | SETUPSP, j, 0, & X0: NEXT
35 'num enemies of 12 x 16 (6bytes wide x 16 lines)
36 num = 32: x% = 0: y% = 0
40 FOR i = 0 TO num-1: | SETUPSP, i, 9, & 8ee2: | SETUPSP, i, 0, &
X1111:
41 | LOCATESP, i, rnd * 200 * 80 rnd
```

```
42 next
43 i = 0
45 GOSUB 100
46 i = i + 1: if i = num Then i = 0
50 | PRINTSPALL, 0.0
60 | AUTOALL
70 goto 45

100 | PEEK 27001 + i * 16, @ and%
110 | PEEK 27003 + i * 16, @ x%
120 if and% <= 0 then | SETUPSP, i, 5.2: | SETUPSP, i, 6.0: return
130 if and%> = 190 then a | SETUPSP, i, 5, -2: | SETUPSP, i, 6.0:
return
140 if x% <= 0 then | SETUPSP, i, 5.0: | SETUPSP, i, 6,1: return
150 if x%> = 76 then a | SETUPSP, i, 5.0: | SETUPSP, i, 6, -1: return

Random rnd = 160 * 3
170 random if = 0 then | SETUPSP, i, 5.2: | SETUPSP, i, 6.0: return
180 if random = 1 then | SETUPSP, i, 5, -2: | SETUPSP, i, 6.0: return
If 190 random = 2 then | SETUPSP, i, 5.0: | SETUPSP, i, 6,1: return
200 if random = 3 then | SETUPSP, i, 5.0: | SETUPSP, i, 6, -1: return
```

9.3.3 Massive logical technique in games like "Pacman"

If you have many enemies and must make decisions in each branch of a maze, you might think that the technique of mass logic is not accurate because every enemy does not check their position in each game cycle, but this can be solved with a simple "trick" . It is simply put enemies in well-chosen positions to start the game.

Suppose you have eight enemies and forks of the maze occur in multiples of 8.

If the enemy is in a first position 8 multiple touch you execute your logic. The second enemy touches you run your decision logic in the next cycle. If you are not in a position fork in the maze you can not change its course

To "fit" position with a multiple of 8 and so to decide which path to take at the fork, just start the game with this second enemy placed on a multiple of 8 minus one. Considering coordinates start at zero, multiples of 8 are:

First enemy position 0 or 8 or 16 or 24 or 32 or XX (in x or y axis, anyway)
Second enemy position 7 or 15 or 23 ...
Third enemy position 6 or 14 or 22 ...

And so on. You put your enemies following this rule:

Position = multiple of 8 - n, the number n being sprite

And every time you touch an enemy run your logic, you can be on a fork. That does not mean you should not check that is not in a position in the middle of a corridor without bifurcations. You should check and for this you can use PEEK against a character layout, as its coordinate divided by 8 is precisely a position of the layout. PEEK is very fast (about 0.9 ms) versus collision detection consuming more than 3ms. PEEK you check if the top position is occupied by a brick (for example) and if there via free then the enemy might take that new direction.

What if you have just 5 enemies? Really easy. There are cycles that can simply not perform any logic and so the five always make decisions to be in positions of bifurcation, multiples of 8.

9.3.4 Movement "block" squads

If you want to just move while a squadron in one direction, either of the following two functions of the library 8BP you will:

- If you use | AUTOALL have to put speed automatic sprites in the direction you want (in Vx, Vy or both) and of course activate the bit 4 of the status byte. The AUTOALL command has an optional parameter to invoke internally |ROUTEALL before moving sprites.

- If you use | you MOVERALL have to activate the bit 5 of the status byte to the sprites you go to move. This command requires as parameters enter as relative movement in Y and X want

The combined use of both strategies can allow you to move two squadrons with complex and different from each paths, we see an example

In this game we have defined the squad right with relative movement (using MOVERALL), following a circular path stored in an array. The group has left the flag of relative movement disabled but have activated the automatic movement flag, and using AUTOALL move down because they all have Vy = 2

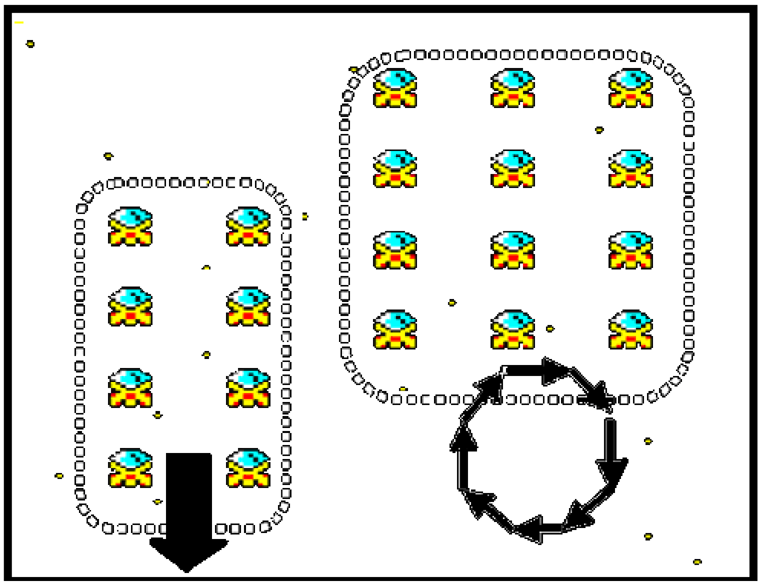


Fig. 31 Movement squads

Here is the list of example

```
1 MODE 0
```

```
10 MEMORY 25999
20 DEFINT az
25 REM path relative movement in the store ry arrays, rx
30 DIM rx (24): DIM r (24)
40 rx (0) = 1: r (0) = 2
50 rx (1) = 1: r (1) = 2
60 rx (2) = 1: r (2) = 2
70 rx (3) = 0: r (3) = 2
80 rx (4) = 0: r (4) = 2
90 rx (5) = 0: r (5) = 2
100 rx (6) = - 1: r (6) = 2
110 rx (7) = - 1: ry (7) = 2
120 rx (8) = - 1: r (8) = 2
130 rx (9) = - 1: ry (9) = 0
140 rx (10) = - 1: r (10) = 0
150 rx (11) = - 1: r (11) = 0
160 rx (12) = - 1: r (12) = - 2
Rx 170 (13) = - 1: r (13) = - 2
180 rx (14) = - 1: r (14) = - 2
Rx 190 (15) = 0: r (15) = - 2
200 rx (16) = 0: r (16) = - 2
Rx 201 (17) = 0: r (17) = - 2
Rx 202 (18) = 1: r (18) = - 2
Rx 203 (19) = 1: r (19) = - 2
Rx 204 (20) = 1: r (20) = - 2
Rx 205 (21) = 1: r (21) = 0
Rx 206 (22) = 1: r (22) = 0
Rx 207 (23) = 1: r (23) = 0
210 rem ------------ ------------ we initialize squads
220 FOR j = 0 TO 31: | SETUPSP, j, 0, & X0: NEXT
230 FOR j = 0 TO 16 STEP 4
240 FOR i = j j + 3 TO
250 | SETUPSP, i, 0, & X10111: | SETUPSP, i, 7.14: | SETUPSP, i, 6,1:
| SETUPSP, i, 5.1
260 | LOCATESP, i, 10 + (ij) * 28,3 * j + 10
270 NEXT
NEXT 280 j
281 rem initialize the second squadron with automatic movement
282 for i = 0 to 7: | SETUPSP, i, 0, & X01111: | SETUPSP, i, 5.2: |
SETUPSP, i, 6.0: next

310 rem loop logic ----- ------------------- program
420 | PRINTSPALL, 0.0
421 | AUTOALL
430 | MOVERALL, r (t) rx (t)
431 | STARS, 0,20,1,3,0
432 rem here index change route
440 t = t + 1: t = 24 THEN IF t = 0
470 GOTO 420
```

In the above example all ships move "block". At any time one can turn their flag relative motion and / or automatic and take life with a single specific logic that makes you fly to our ship and attack us.

But if what you want is something more advanced, such as having a group of ships do not move "block" but follow a path in "Indian" row, so that the ship is head makes movements that are imitating the other, you can use strategies like that I will describe below.

9.3.5 Mass Logics: Movement "single file"

A group of ships can have a horizontal automatic movement but have a logic so that when x coordinate exceeds a certain value, automatic and happen to have on the movement time. And when they pass another coordinate x again to deactivate the relative movement

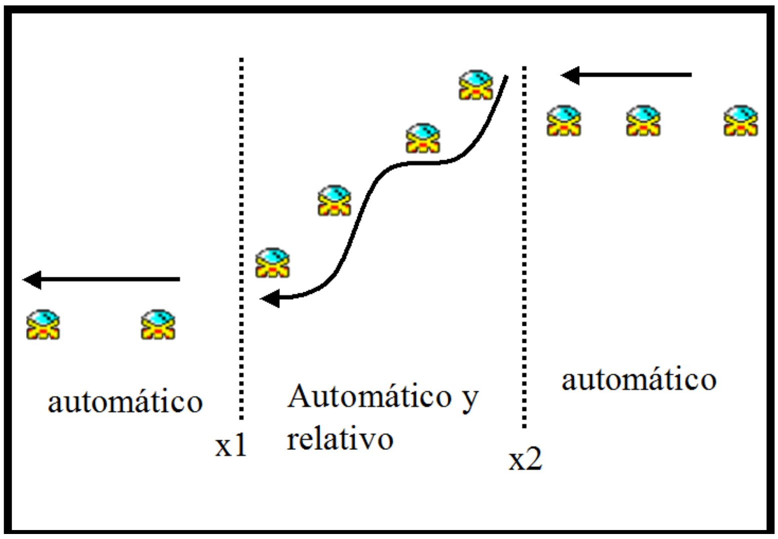


Fig. 32 Trajectories in single file

This simple strategy allows you to make the effect of "single file" of ships in a very efficient way, without acting on the coordinates of any individually.

The way is not controlling the x coordinate of each ship because that would imply a very slow loop checks. The quickest way is via a timer. In each game cycle increases. Something like what I show below

```
t = t + 1: if t> = 20 and t <= 25 then a | SETUPSP, t-20.0, & x10111
```

With that line within the logic of the game, each time you run the counter is incremented and when it reaches 25, one by one, corresponding to 0,1,2,3,4 ships sprites are changing the operating mode . When the counter reaches 30, it has passed the 5 ships to the new mode of behavior. The counter serves to control the x coordinate of all vessels but an infinitely more effectively. This strategy is essentially not run the logic of each ship in each cycle, but only the logic of the ship that interests run at all times. Imagine how costly it would be to check the x coordinate in 8 ships each cycle. According to the performance table, an IF takes 1.42ms and therefore at least eight ships would be taken about 12 ms to do the same as we have done with a single IF applied to the timer.

9.3.6 Massive logic: Complex Trajectories

If what you want is something even more complex, where ships describe a circle in single file, you must base it only on automatic movement. Think of the following figure

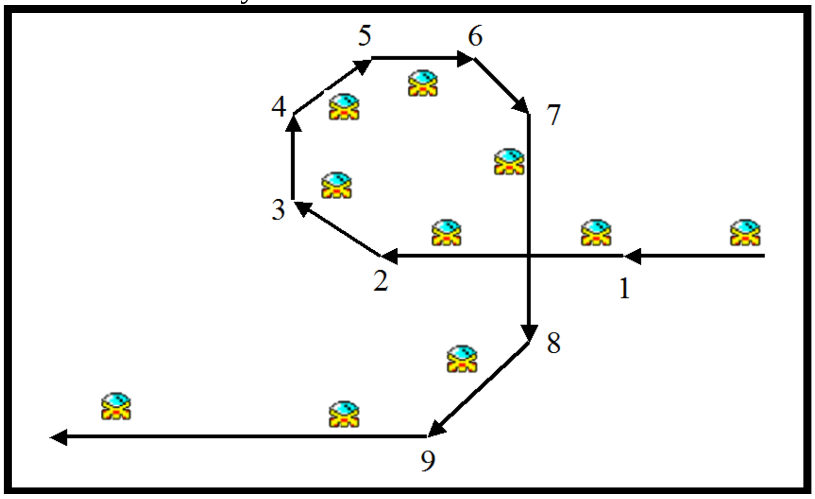


Fig. 33 Checkpoints paths

With a single counter-nine checkpoints you can go one by one changing Vy, Vx of each ship, so that in each cycle of your logic only acts as much on a ship

For example the checkpoint "2" would be something like

t = t + 1: if t> = 15 and t <= 20 then a | SETUPSP, t-15.5, -1: | SETUPSP, t-15.5, -1

Set 9 points for the counter control means establishing 9 IF statements through which passes the main program but you can use strategies to jump with GOTO, such as

if t <50 goto first group of IF
if t <100 goto second group
if t <100 goto third group
etc

Thus with only 3 IF in each cycle of the game you will control the 9 points of control.

Let's look at a simplified example. The following example works very soft with a squadron of eight ships. It has only 2 control points but is essentially the same strategy. I used two counters to prevent ships from changing operating mode too often. The counter which makes change is "t" and only advances each time the counter "tt" count to 10, at which time "tt" starts again. Numeral 10 is precisely the separation between sprites considering the x coordinate.

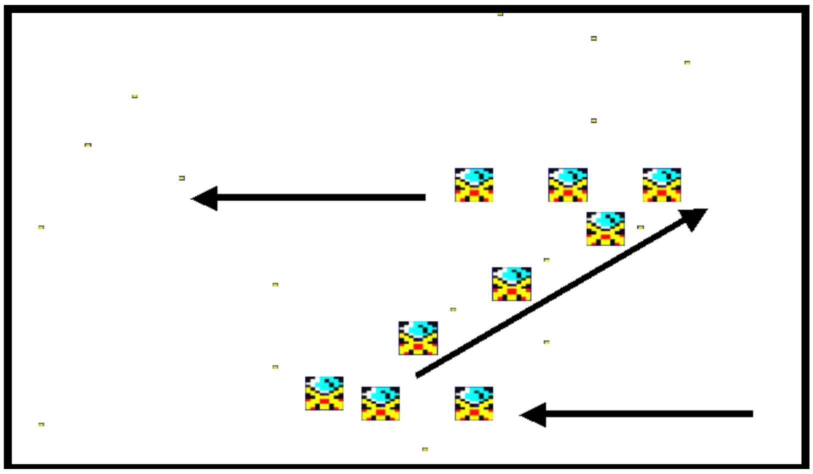


Fig. 34 complex paths with massive logic

```
1 MODE 0
10 MEMORY 25999
20 DEFINT az
100 rem initialisation --------------------------
220 FOR j = 0 TO 31: | SETUPSP, j, 0, & X0: NEXT
221 FOR j = 0 TO 7:
222 | SETUPSP, j, 0, & X1111: | setupsp, j, 7.14:
223 | setupsp, j, 5.0: | setupsp, j, 6, -1:
224 | locatesp, j, j * 150,40 + 10:
225 NEXT

400 t = 0: t = 0: rem loop logic game ----------------
420 | PRINTSPALL, 0.0
421 | autoall
431 | STARS, 0,20,1,0, -2
432 tt = tt + 1: if tt tt = 10 THEN = 0 GOSUB 1000
441 rem first installment (no logic)
Goto 442 420

499 rem ------------------ second tranche
500 | setupsp, t-1,5, -2: | setupsp, t-1,6,1:
510 return
599 rem Leg 3 ------------------------
600 | setupsp, t-5,5,0: | setupsp, t-5,6, -1:
610 return
C counter routine 999 -----------------
1000 t = t + 1: if t <= 8 then a GOSUB 500
1010 if t> 4 and t <= 13 then a GOSUB 600
1020 return
```

Let's look at another example to further clarify the concept of using a timer as a mechanism to alter the behavior of sprites, with no need to run independently for each logical.

In this example 8 enemies begin arranged in two diagonals of 4 each. Each group of 4 moves in the opposite direction and as they reach the end change direction, producing an effect of intertwining paths very attractive

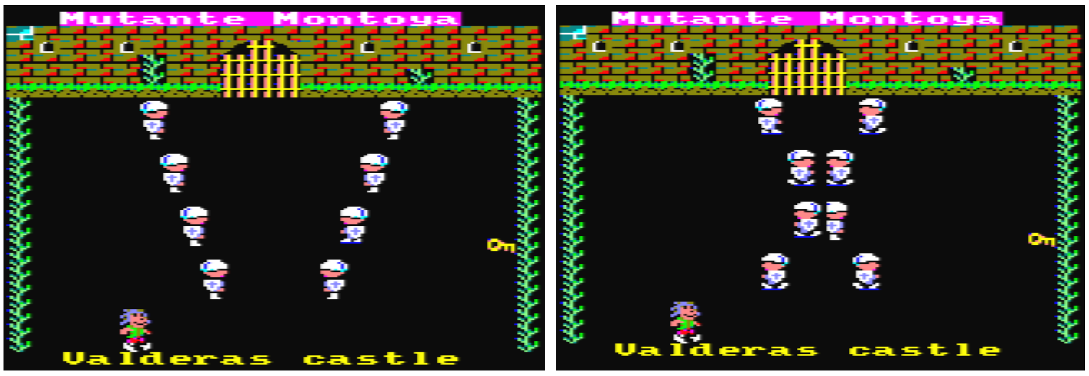


Fig. 35 use of massive Logics in "Mutant Montoya"

In this logic, every 4 cycles of the main loop (controlled by counter cc) increased runs counter c. And when c is between 10 and 15 (cycles 40 to 60) are going altering the behavior of the soldiers who are placed in opposite diagonals. Only two soldiers behavior is altered at a time. After a time corresponding to c = 20 (80 units of time) begin to turn around again.
The main loop also controls the collision between sprites and off the screen when the Y coordinate of the character (controlled by "n") is less than 26

```
4401 c1 = 10: c2 = 20: c = 0: cc = 0
4409 '----- --------- main logic loop
4410 GOSUB 1500: 'control character
4421 cc = cc + 1: if then a cc cc = 4 = 0 GOSUB 4700
4609 | AUTOALL 'automatic movement 8 sprites
4610 | PRINTSPALL, 1.0 'massive impression of sprites (9 = 8 +
character)
4611 | COLSP, 0, @ cs%: IF cs% <32 THEN GOSUB 600 GOTO 4020
4612 IF n <26 THEN RETURN
4620 GOTO 4410
4699 'control routines ----- ---- escuadron
4700 c = c + 1: if c> c1 and c <= c1 + 4 Then GOSUB 4800
4701 if c> c2 and c <= c2 + 4 Then GOSUB 4900
4702 Then if c = c = 10 30
4710 return
4799 'to turn ----- 2 2 --------
4800 | setupsp, c1 + 5-c, 7.10: | setupsp, c1 + 5-c, 6, -1
4801 | setupsp, c1 + 5-c + 4,7,9: | setupsp, c1 + 5-c + 4,6,1
4810 return
4899 '----- to turn again --------
4900 | setupsp, c2-c + 5, 7.9: | setupsp, c2-c + 5, 6.1
4901 | setupsp, c2 + 5-c + 4,7,10: | setupsp, c2 + 5-c + 4,6, -1
4910 return
```

9.3.7 massive logic: Controls the time, not space

Now let's go a step further, pushing the limits of this technique. In the above example we have differentiated the values of timer with several statements "IF", but we can do it all with a single statement, more efficient paths.

Begin imagining a path for a row of eight enemy ships. the ships will spend one to one by a series of "control nodes" which are places in space where they change direction, defined by their velocities in X and Y, ie (Vx, Vy)

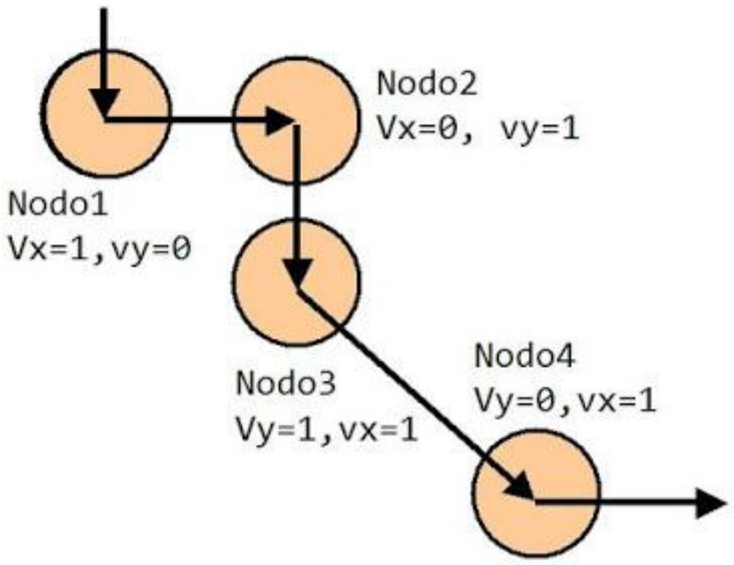


Fig. 36 Defined path to "control nodes"

One way to control that 8 ships change direction at these locations would compare their X, Y coordinates with each of the control nodes and if they match any of them, then apply the new speeds associated with the change in that node . Since we speak coordinates 2, 8 ships and 4 nodes, we are facing:

2 x 8 x 4 = 64 checks in each frame

This is not viable if we speed from BASIC. And it is not a computationally efficient strategy. Since we are facing a "deterministic" scenario, we could be sure in every moment of time where they will would find each of the ships and therefore instead of checks in space, we can only focus on the time coordinate (which is the frame number of the game or the so called number of "cycle play")

Since we know the speed at which ships move, we can know when the first of them will pass through the first node. At that moment we will call t (1). Also we assume that because of the separation between the vessels, ships the second pass through the node at time t (1) +10. the third in t (1) and the eighth +20 t (1) +70

Knowing this we can control the time with two variables: one count tens (i) and other units (j). To control the change of the 8 ships in the first node we can write:

```
j = j + 1: 10 THEN IF j = j = 0: i = i + 1
IF i> = t (1) AND i <t (1) +8 THEN [updated speed ship it (1) with the
velocity values of node 1]
```

As we see with a single line we can change the speed of each ship as they pass each by node 1. During the 8 seconds later at time (1) are updated each of the ships, just as They are passing through the node control

Now let's apply the same to the 4 nodes. We could run 4 checks instead of one, but would be inefficient. Also, if we had this many nodes would be many checks. We can do it only with one, considering that the first ship passes through a node at a time t (n) and the last ship passes by that node in t (n) +7

When the first ship passes through the first node, it makes sense to start checking node 2, but not the node 3 or 4. We have the largest node that will controlar.En as the child node, we can assume that although we have 20 nodes, are far enough apart so that no ships traversing more than 3 nodes at once (we will assume that and we will use this "3" as a parameter). Therefore the child to check node is the biggest - 3. At lower node we'll call "nmin" and more "nmax". (Nmin = nmax-3). If we want to have full freedom to define any path, nmin nmax must be less than the number of ships of the line.

```
j = j + 1: IF j = 10 THEN j = 0: i = i + 1: n = nmaxIF n <nmin THEN
50: 'there is no need to update more navesIF i> = t (n) AND i <t (n )
+8 THEN [updated it (n)]: IF it (n) = 0 THEN nmax = nmax + 1: nmin =
nmax-3n = n-1

50 'more game instructions
```

As you can see when increases at 1 dozen time, nodes are checked from "nmax" to "nmin" And in each node update ship it (n). Think that if t (3) is, for example, 23, then when we got to the node 3 at time i = 23, update the ship i-23 = 0, with the velocity values of node 3.

In the next cycle of play continues even check if "current" time interval node 2, and update the ship 1 with the velocity values of node 2,

and in the next cycle check node 1 and if still valid update the ship 2, and so on, always bearing in mind that the first ship may have reached nmax, while the next ship can be in nmax-1 and the following in nmax-2, etc.

In short, we have transformed 64 checks in only 1 using "massive Logics". And if the path had 40 instead of 4 nodes, we would have turned 640 operations in one!
Here is an excerpt of the video game "annunaki" that uses this technique to handle the paths of two symmetrical rows of six ships is included each

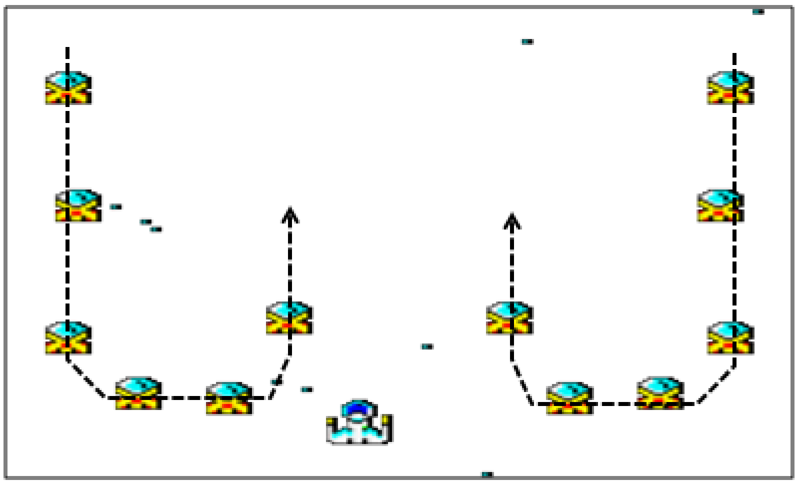


Fig. 37 Two rows with massive logic

Initialization of the control points of the trajectory. Different paths with different parameters are achieved

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

massive logic that manages the two rows. I have highlighted in red line that governs the movement of the 12 ships

```
2999 '4 ---- phase trajectories in two symmetrical rows configurable k
(i), kx (i), k (i). 10 are separated in 10 x -------------
3000 totaln = 12: Soft = 31-totaln: Hard = 100: ini = 31-totaln: |
COLSP, 32, ini, 30
For i = ini 3010 to 24
3020 | SETUPSP, i, 9 navemala1: | SETUPSP, i, 7,0: | SETUPSP, i, 5, k
(0) | SETUPSP, i, 0, & x01011: | SETUPSP, i, 6, kx (0 )
3021 | SETUPSP, i + 6.9, navemala1: | SETUPSP, i + 6,7,0: | SETUPSP, i
+ 6.5, k (0) | SETUPSP, i + 6.0, & x01011: | SETUPSP , i + 6.6, kx (0)
3030 | LOCATESP, i, -30- (i-ini) * 20.80 + (i-ini) * 10: | LOCATESP, i
+ 6 -30- (i-ini) * 20,0- (i- ini) * 06/10: '6 is the width in bytes of
the navemala and 26 high
3040 | STARS: GOSUB 500: GOSUB 750: | AUTO, 7: | AUTO 8: | AUTO, 9: |
PRINTSPALL, 1.0
3050 next
3060 cycle = 0: t = 0: col% = 32: d% = 200
3100 GOSUB 500: GOSUB 750
3109 | AUTOALL: | PRINTSPALL: | STARS
3120 | COLSPALL: if sp <32 then if sp = 31 then a GOSUB 300: | MUSIC
0.5: goto else GOSUB 3000 770
3130 cycle = cycle + 1: if then a cycle cycle = 10 = 0: t = t + 1: m =
inim
3131 PNIF if m = 1 then 3100 +
3132 IF t> = k (m) and t <k (m) +6 THEN i = tk (m) + ini: | SETUPSP,
i, 5, k (m), kx (m): | SETUPSP, i + 6 , 5, k (m), - kx (m): m = PNIF
THEN IF PNIF = m + 1: inim = PNIF-3 ELSE ELSE IF t> = k (12) THEN
RETURN
3133 m = m + 1: goto 3100
```

9.4 ROUTEALL

This is an "advanced" command available from the V25 version of the 8BP library. Greatly simplifies programming because you can define a path and make a sprite the scroll step by step through the ROUTEALL command.

First you need to create a route. To do this you need to edit the file routes_tujuego.asm. Each route has an unspecified number of segments and each segment has three parameters:

- How many steps we will take in that segment
- that Vy will be maintained during the segment
- that speed Vx will be maintained during the segment

At the end of the segment specification we must put a zero to indicate that the route has been completed and that the sprite should start down the path from the beginning.

Here is an example:

```
; LIST OF ROUTES
; ================
: Put the names of all routes do here
route_list
      dw ROUTE0
      dw ROUTE1
      dw route2
      dw Route3
      dw ROUTE4


; DEFINITION OF EACH ROUTE
; ==========================
ROUTE0; a circle
; -----------------
      db 5,2,0
      db 5,2, -1
      db 5,0, -1
      db 5, -2, -1
      db 5, -2.0
      db 5, -2.1
      db 5,0,1
      db 5,2,1
      db 0


ROUTE1; Left Right
; -------------------------
      10.0 db, -1
      db 10,0,1
      db 0


route2; up and down
; --------------------
```

```
      db 10 -2.0
      db 10,2,0
      db 0


Route3; eight
; ----------------

      db 15,2,0
      db 5,2, -1
      db 5,0, -1
      db 25, -2, -1
      db 5,0, -1
      db 5,2, -1
      db 15,2,0
      db 5,2,1
      db 5,0,1
      db 25, -2.1
      db 5,0,1
      db 5,2,1
      db 0

ROUTE4; a loop and goes to the left
; --------------------
      120.0 db, -1
      db 10 -2, -1
      db 20, -2.0
      db 10 -2.1
      db 5,0,1
      db 10,2,1
      db 20,2,0
      10.2 db, -1
      80.0 db, -1
      db 0

ROUTE5
segment_4_1 db 10 -2, -1
segment_4_2 10.2 db, -1
segment_4_3 db 0
```

Now to use routes from BASIC, simply we assign the path to a sprite with stating that we want to change the parameter 15, which is indicating the route SETUPSP command. In addition, the route flag (bit 7) must be activated in the status byte sprite and we'll put the flag with automatic movement and animation and printing.

```
10 MEMORY 25999
11 ON BREAK GOSUB 280
20 MODE 0: INK 0.0
21 LOCATE 1.20: PRINT "command | ROUTEALL and animation
macrosequences"
30 CALL & 6B78: az DEFINT
31 | setLimits, 0,80,0,200
40 FOR i = 0 TO 31: | SETUPSP, i, 0,0: NEXT
```

```
41 x = 10
50 FOR i = 1 TO 8
51 x = x + 20: IF x> = 80 THEN x = 10: y = y + 24
60 | SETUPSP, i, 0.143: rem with this flag active route
70 | SETUPSP, i, 7.2: | SETUPSP, i, 7.33: rem macrosequence animation
71 | SETUPSP, i, 15.3: rem assigned the route number 3
80 | LOCATESP, i, 30.70
81 REM | LOCATESP, i, 200 * RND RND 80 *
82 FOR t = 1 TO 10: | AUTOALL: | ROUTEALL: | PRINTSPALL, 1.0: NEXT
91 NEXT
100 | AUTOALL: | PRINTSPALL, 1.0
110 | ROUTEALL
120 GOTO 100
280 | MUSICOFF: MODE 1: 0.0 INK: PEN 1
```

Now we have everything. This advanced technology will greatly simplify programming with spectacular results.

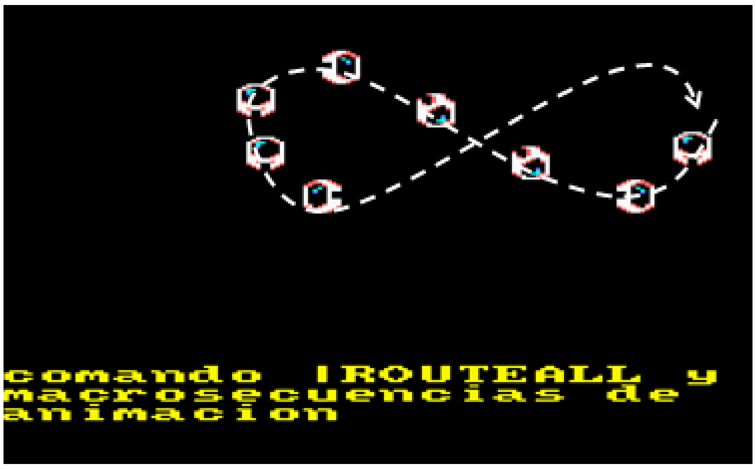


Fig. 38 a route as 8

10 Games with scroll

The library can scroll 8BP from the V24 version, by the combined use of a "world map" and a feature called | MAP2SP, which I will explain at the end of this chapter.

There are several techniques that can be used independently or combined to scroll. Let's see some examples

10.1 Scroll stars or dappled earth

In the library 8BP have an easy function used to create a background effect of stars moving, giving the feeling of scroll. It is the function | STARS.
This function is able to move up to 40 stars simultaneously without altering your sprites, so it is as if pasasen "below"

| STARS, <initial star>, <num stars>, <color>, <d>, <dx>

You have a bank of stars and you can combine several STARS commands for working with groups of stars at different speeds, giving a sense of depth.

The bank consists of 40 pairs of bytes representing coordinates (y, x). Taking from the direction 42540-42619 (are 80 bytes in total)

One way to generate random star 40 would

```
FOR dir = 42540 42618 TO STEP 2: POKE dir, RND * 200: POKE dir + 1, RND * 80:
NEXT
```

For a detailed description of the command, see the chapter "reference guide". In this chapter you will find different examples to simulate stars, earth, stars with two planes of depth, rain or even snow. Imaginatively probably possible to simulate more things with this same function.

For example if the stars you put in two or three sequences pixels diagonally instead of randomly distribute them, you can get a displacement movement of "segments", which could be ideal to simulate rain.

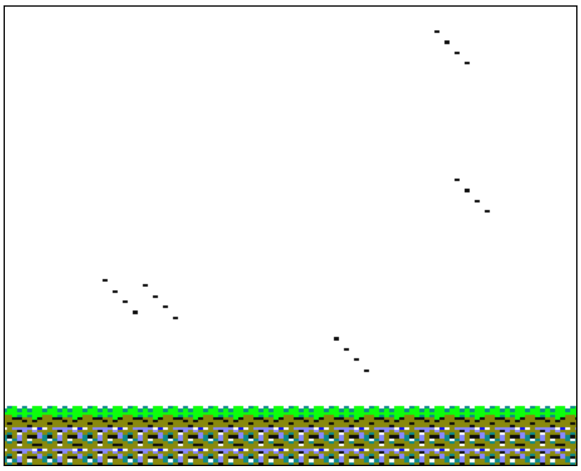
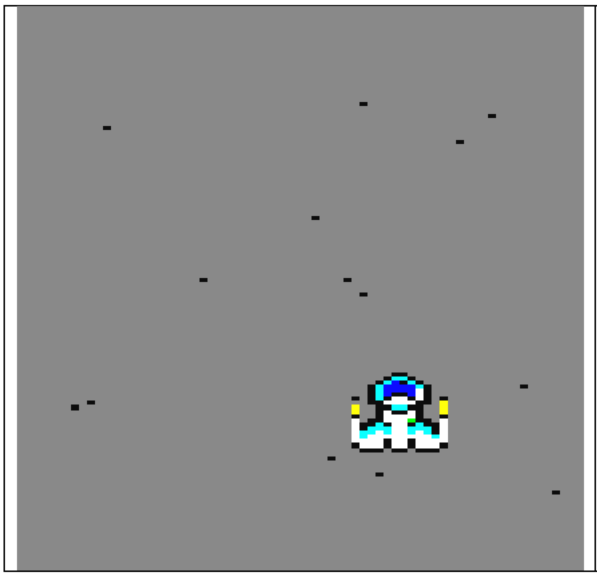


Fig. 39 Effect of rain STARS

```
10 MEMORY 25999
20 CALL & 6B78: 'install RSX
40 mode 0: CALL & BC02: 'restores default palette case
50 bank = 42540
FOR 60 TO dir = bank bank + 40 * 2 STEP 8:
70 y = INT (RND * 190): x = INT (RND * 60) +4
80 POKE dir, and POKE dir + 1, x:
90 POKE dir + 2 (y + 4): POKE dir + 3, x-1
100 POKE dir + 4, (y + 8): POKE dir + 5, x-2
POKE 110 dir + 6 (y + 12): POKE dir + 7, x-3
120 NEXT

140 'SCENARIO OF RAIN
141 '--------------------
150 | setLimits, 0,80,50,200 'limits of the game screen
151 lawn = & 84d0: | SETUPSP, 30.9, lawn 'letter Y is the sprite 31
152 rocks = & 84f2: | SETUPSP, 21,9, rocks: 'letter P is the sprite 21
160 $ string = "YYYYYYYYYYYYYYYYYYYY"
170 | LAYOUT, 22.0, $ @ string: 'This paints the grass
180 $ string = "PPPPPPPPPPPPPPPPPPPP"
190 | LAYOUT, 23.0, $ @ string: 'paints a row of rocks
200 | LAYOUT, 24.0, $ @ string: 'paints another row of rocks
210 'game cycle -------- -----
Defint 211 az
220 LOCATE 1.10: PRINT "DEMO OF RAIN"
221 LOCATE 1,11: PRINT "press ENTER"
230 | STARS, 0,40,4,2, -1
240 IF INKEY (18) = 0 THEN 300
250 GOTO 230
```

As the example of two planes of stars you have in chapter reference library, here we will see an example in which a spaceship flying over planet earth speckled with vertical scroll feeling



In this example it tries to see how the stars (or flecks of soil) literally pass under the ship, without affecting it.

Fig. 40 Effect of land peppered with STARS

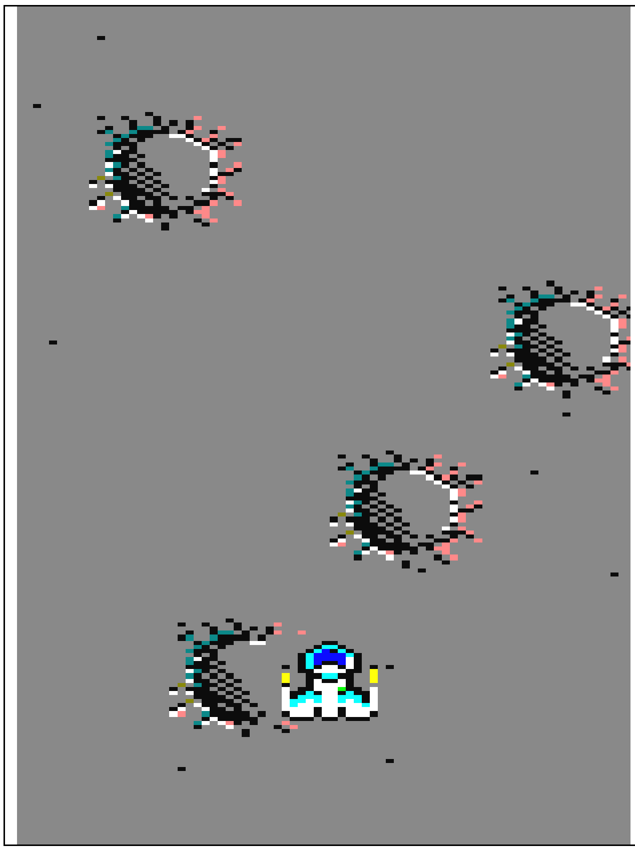
There is a way to invoke an optimized way the STARS command and simply to invoke a first time with the following parameters and times without parameters. The command assume that the parameter values are the same as those of the last invocation parameters and it saves time the interpreter BASIC spent processing parameters until 1.7ms

```
10 MEMORY 25999
11 'I put random stars
12 FOR dir = 42540 42618 TO STEP 2: POKE dir, RND * 200: POKE dir + 1,
RND * 80: NEXT
20 MODE 0: DEFINT AZ: CALL & 6B78: 'install RSX
25 Call & BC02: 'restores default palette case
26 0.13 ink: 'gray background
30 FOR j = 0 TO 31: | SETUPSP, j, 0, & X0: NEXT: 'reset sprites
40 | setLimits, 12,80,0,186: 'establish the limits of the game screen

41 'we will create a ship in the sprite 31
42 | SETUPSP, 31.0, & 1: 'status
43 ship = & a2f8: | SETUPSP, 31.9, ship 'image assigned to sprite 31
44 x = 40 y = 150: 'coordinates of ship

49 'game cycle -------- -----
50 | STARS, 0,20,5,1,0: 'black stars on gray ground
55 GOSUB 100: 'movement of the ship
60 | PRINTSPALL, 0.0
70 goto 50


99 'routine movement ship ------------
100 IF INKEY (27) = 0 THEN x = x + 1: GOTO 120
110 IF INKEY (34) = 0 THEN x = x-1
120 | LOCATESP, 31, and x
130 RETURN
```

10.2 Craters on the Moon

Now making combined use of the relative movement, the scroll of stars and the order in which the sprites are printed we will see an example of how to simulate a ship flies over the moon, that is, how to simulate a scroll screen with these elements

First we have chosen the sprite 31 for our ship, because that will make the final print. The sprites are printed in order, starting at zero and ending at 31. If a crater is a lower sprite to 31 will be printed before the ship and the ship will be "above", giving the impression that it is flying over.



In the picture the ship appears above a crater. As the bottom of the ship and the floor are the same color, it is "acceptable" though not overwriting

It is very important that the stars (specks of soil) and craters move at the same speed. To do so the command | MOVERALL as the command | STARS must have the same value of the parameter d

Fig. 41 flying over the moon

This is the BASIC code

```
10 MEMORY 25999
11 'I put random stars
12 FOR dir = 42540 42618 TO STEP 2: POKE dir, RND * 200: POKE dir + 1,
RND * 80: NEXT
20 MODE 0: DEFINT AZ: CALL & 6B78: 'install RSX
25 Call & BC02: 'restores default palette case
26 0.13 ink: 'gray background
30 FOR j = 0 TO 31: | SETUPSP, j, 0, & X0: NEXT: 'reset sprites
40 | setLimits, 12,80,0,186: 'establish the limits of the game screen

41 'we will create a ship in the sprite 31
42 | SETUPSP, 31.0, & 1: 'status
43 ship = & a2f8: | SETUPSP, 31.9, ship 'image assigned to sprite 31
45 x = 40 y = 150: 'coordinates of ship

46 'Now the craters
47 crater = & a39a: c% = 0
48 for i = 0 to 3: | SETUPSP, i, 9, Crater:
49 | SETUPSP, i, 0, & x10001: 'impression and relative movement
50 x (i) = rnd * 40 + 20: and (i) = i * 40
60 | locatesp, i, and (i), x (i)
70 next
```

```
71 t = 0

80 'game cycle -------- -----
81 | STARS, 0,20,5,3,0 'movement black stars
82 GOSUB 100: 'movement of the ship
83 | MOVERALL, 3.0 'movement craters
84 t = t + 1: if t> 10 THEN t = 0: GOSUB 200: 'control craters
90 | PRINTSPALL, 0,0 'impression ship and craters
91 goto 81


99 'routine movement ship -------------
100 IF INKEY (27) = 0 THEN x = x + 1: GOTO 120
110 IF INKEY (34) = 0 THEN x = x-1
120 | LOCATESP, 31, and x
130 RETURN

199 'reentry control cratering
200 c = c + 1
210 if c = 6 Then c = 0
220 | PEEK 27001 + c * 16, @ c%
If c% 230> 200 then a | POKE, 27001 + c * 16 -20
240 return
```

The following example has been a squadron of spaceships, craters and your spaceship (9 sprites in total and stars). Thanks to the MOVERALL and AUTOALL commands, you can move all this at an appropriate pace for an arcade game

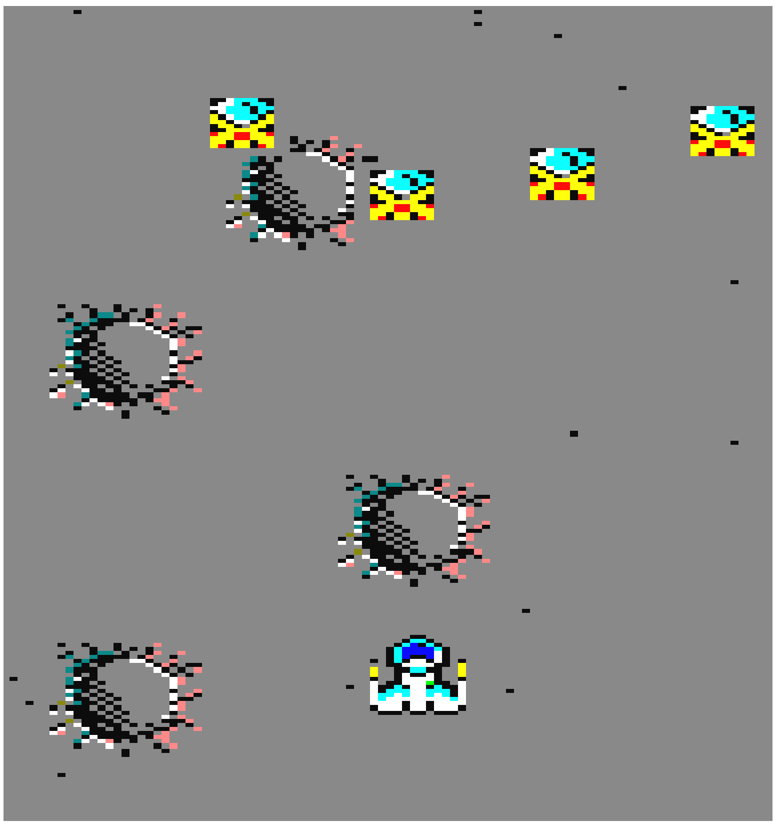


Fig. 42 scrolling arcade games using 8BP

```
10 MEMORY 25999
11 'I put random stars
12 FOR dir = 42540 42618 TO STEP 2: POKE dir, RND * 200: POKE dir + 1,
RND * 80: NEXT
20 MODE 0: DEFINT AZ: CALL & 6B78: 'install RSX
25 CALL & BC02: 'restores default palette case
INK 26 0.13: 'gray background
30 FOR j = 0 TO 31: | SETUPSP, j, 0, & X0: NEXT: 'reset sprites
40 | setLimits, 12,80,0,186: 'establish the limits of the game screen
41 'we will create a ship in the sprite 31
42 | SETUPSP, 31.0, & 1: 'status
43 ship = & A2F8: | SETUPSP, 31.9, ship 'image assigned to sprite 31
45 x = 40 y = 150: 'coordinates of ship
46 'Now the craters
47 crater = & A39A: c% = 0
48 FOR i = 0 TO 3: | SETUPSP, i, 9, Crater:
49 | SETUPSP, i, 0, & X1001: | SETUPSP, i, 5.3: | SETUPSP, i, 6.0
'printing and auto movement
50 x (i) = RND * 40 + 20: and (i) = i * 40
60 | LOCATESP, i, and (i), x (i)
70 NEXT
71 t = 0
75 'ships
76 FOR i = 4 TO 7: | SETUPSP, i, 7.14: | SETUPSP, i, 0, & X10101: |
LOCATESP, i, RND * 20, i * 10-20
77 NEXT:
78 inc = 1
80 'game cycle -------- -----
81 | STARS, 0,20,5,3,0 'movement black stars
82 GOSUB 100: 'movement of the ship
84 t = t + 1: IF t> 10 THEN inc = -inc: t = 0: GOSUB 200: 'control
craters
85 | MOVERALL, 2, inc
86 | AUTOALL
90 | PRINTSPALL, 0.0
91 GOTO 81
99 'routine movement ship -------------
100 IF INKEY (27) = 0 THEN x = x + 1: GOTO 120
110 IF INKEY (34) = 0 THEN x = x-1
120 | LOCATESP, 31, and x
130 RETURN
199 'reentry control cratering
200 c = c + 1
210 IF c = 4 THEN c = 0
220 | PEEK 27001 + c * 16, @ c%
IF c% 230> 200 THEN | POKE, 27001 + c * 16 -20
240 RETURN
```

10.3 Mountains and lakes: technique of "spotting"

The technique to paint mountains in games with horizontal scrolling and lakes in games with vertical scroll is the same

What we will do is to paint only the beginning of the mountain, through a sprite that helps us to paint your right side. We will put as many as we want. In this case I put three

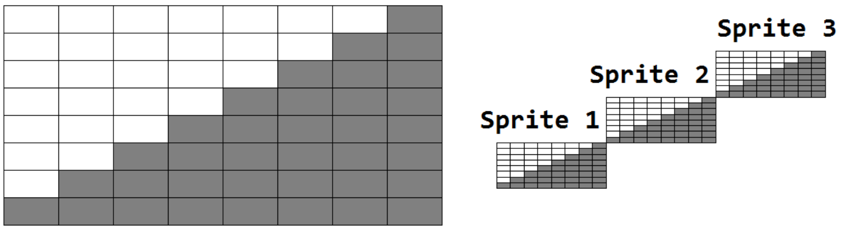


Fig. 43define the side of a mountain with several sprites

We do the same with a mirrored image that associate to 3 other sprites, and will place on the right, building the left side of the mountain. Careful that the mirror image has at least the last column of pixels to zero. This will allow itself to be erased move to the left.

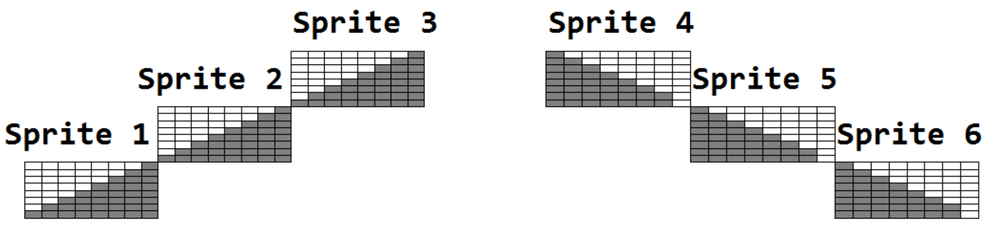


Fig. 44 Sprites available to build a mountain

By moving all sprites left by automatic or relative movement, sprites left begin to "stain" the background and thus "filling" the mountain, while the right sprites start cleaning. If the mountain appears gradually entering the screen will look like a sprite of a huge mountain, when in fact it is 6 small sprites.

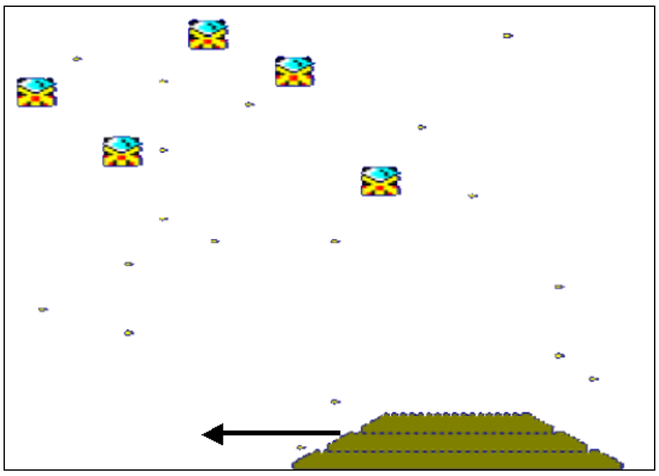


Fig. 45 a horizontal scroll of a mountain

Here you have the example illustrated

```
1 MODE 0
10 MEMORY 25999: CALL & 6B78
20 DEFINT az
30 FOR j = 0 TO 31: | SETUPSP, j, 0, & X0: NEXT
35 'sprites right side of the mountain
40 FOR i = 1 TO 3: | SETUPSP, i, 9, & 9102: | SETUPSP, i, 6, -1: |
SETUPSP, i, 0, & X1111: NEXT: 'r
45 'sprites for the left side of the mountain
50 FOR i = 7 TO 9: | SETUPSP, i, 9, & 9124: | SETUPSP, i, 6, -1: |
SETUPSP, i, 0, & X1111: NEXT: 'l
55 x = 80: inc = 1
60 | LOCATESP, 7,176, x: | LOCATESP, 8,184, (x-4): | LOCATESP, 9,192,
(x-8)
70 | LOCATESP, 1,176, x + 20: | LOCATESP, 2,184, (x + 24): | LOCATESP,
3,192, (x + 28)
71 'I place 5 ships
75 FOR i = 20 TO 24: | SETUPSP, i, 7.14: | SETUPSP, i, 0, & X11101: |
LOCATESP, i, RND * 100 (i-20) * 10: NEXT: 'relative

79 'main logic
80 | PRINTSPALL, 0.0
90 | AUTOALL
91 t = t + 1: GOSUB 200
92 | STARS, 0,20,1,0, -2
93 | MOVERALL, 0, inc
Tt tt = 94 + 1: GOSUB 300
100 GOTO 80

199 'routine repositions the mountain to the left of the screen
200 IF t <140 THEN RETURN
210 x = 100: | LOCATESP, 7,176, x: | LOCATESP, 8,184, (x-4): |
LOCATESP, 9,192, (x-8)
220 | LOCATESP, 1,176, x + 20: | LOCATESP, 2,184, (x + 24): |
LOCATESP, 3,192, (x + 28)
230 t = 0: RETURN
300 tt = 40 THEN IF inc = -inc: tt = 0
310 RETURN
```

In the case of a set of vertical scroll, if we paint a lake on a brown field, will do the same, some sprites that are "smearing" the terrain and other farther than they will "cleaning", pretending that this is a huge lake, in one piece.

You just have to have a precaution, and that the ships not flying over the lake or your "trick" will be revealed!

10.4 Tunel or rocky road

One way to do a rocky tunnel is simply concatenate the sprites that make wall. The following example of the video game "Anunnaki", 10 sprites are used on each side, 24 pixels high and 8 each wide. Although the speed of the scroll is not exciting, it is pretty decent.

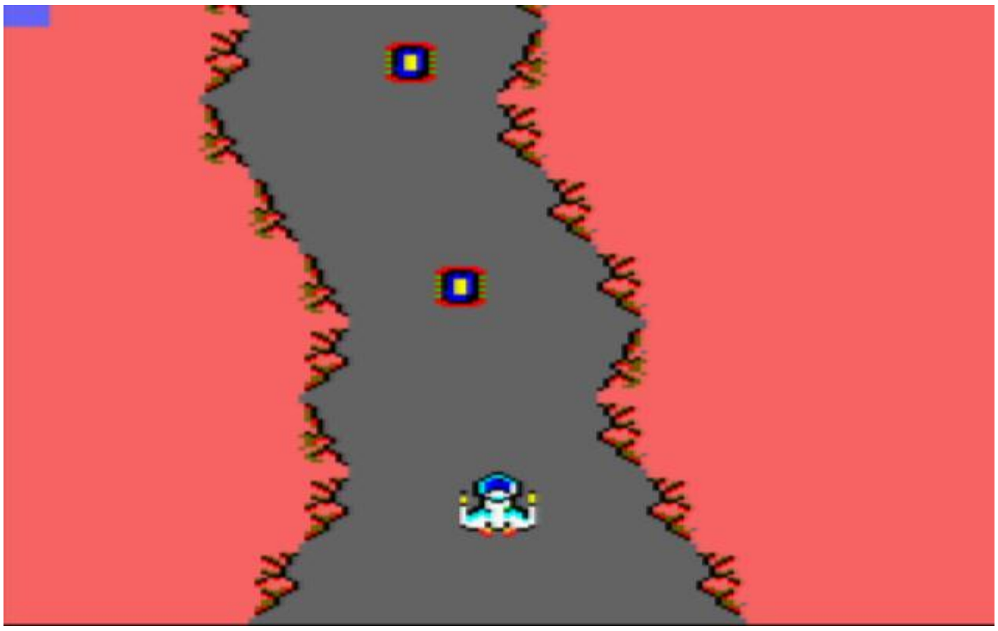


Fig. 46 Rocky tunnel effect

Periodically (by the following instruction) to the routine that paints the rocky tunnel is invoked. every 8 cycles of play is invoked because the walls move with automatic movement with Vy = 3

```
6555 cycle if mod 8 = 0 then GOSUB 6700

<...>

6699'logica Gorge
6700 des = des + 1: if then a des des = 20 = 10
6701 if xl> 30 THEN img = rocall: | SETUPSP, des, 9, img: | LOCATESP,
des, -24, xl: xl = xl-4: goto 6730
6702 if xl <0 THEN img = rocalr: xl = xl + 4: | SETUPSP, des, 9, img:
| LOCATESP, des, -24, xl: goto 6730
6703 random = int (rnd * 2)
6704 if random = 1 THEN img = rocall: | SETUPSP, des, 9, img: |
LOCATESP, des, -24, xl: xl = xl-4: goto 6730
6705 img = rocalr: | SETUPSP, des, 9, img: xl = xl + 4: | LOCATESP,
des, -24, xl
6710 'now the right side
6730 if xr <xl + 30 Then img = rocarr: | SETUPSP, des + 10.9, img: |
LOCATESP, des + 10, -24, xr: xr = xr + 4: return
Xr-xr 6740 = 4: img = rocarl: | SETUPSP, des + 10.9, img: | LOCATESP,
des + 10, -24, xr: return
```

10.5 Walking through the village

Let's look at one last example that uses relative motion to give the feeling of scroll, using sprites with drawings of houses, a mottled ground and a character located in the center in the direction you take, makes the whole environment to move on. It is a very basic example but it gives you an idea of the potential of these functions. Here it is what moves all the people!

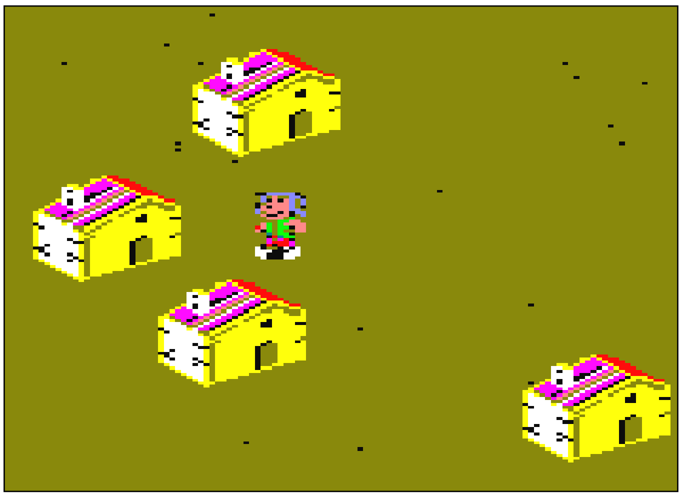


Fig. 47 The whole town moves

```
10 MEMORY 25999
20 MODE 0: Call & 6b78
30 DEFINT az
INK 240 0.12
241 border 7
250 FOR i = 0 TO 31
260 | SETUPSP, i, 0, & X0
270 NEXT
280 FOR i = 0 TO 3
290 | SETUPSP, i, 0, & X10001
300 | SETUPSP, i, 9, & A01C: rem houses
301 | LOCATESP, i, RND * 150 + 50 * 60 + 10 rnd
310 NEXT
320 | SETUPSP, 31,7,6: rem character
330 | LOCATESP, 31,90,38
340 | SETUPSP, 31.0, & X1111
400 x = 0: ya = 0
410 IF INKEY (27) = 0 THEN x = -1:
420 IF INKEY (34) = 0 THEN x = + 1:
430 IF INKEY (67) = 0 THEN ya = + 2
440 IF INKEY (69) = 0 THEN ya = -2
450 | MOVERALL, ya, xa
460 | PRINTSPALL, 1.0
470 | STARS, 1,20,5, ya, xa
480 GOTO 40
```

10.6 Scroll based on a world map

All the above techniques (STARS command to a mottled ground technique stained, relative movement of elements) are perfectly valid for scrolling, and even compatible with what we see now, which is the fundamental technique that will allow design a "world map" and make your character or your ship moves by with just one line of code.

The idea is simple: create a list of items that make up the map of the world (up to 64 items that call "map elements" or "map items"). Each element is described by the coordinates where it is located and the memory address where the image of the item in question (a house, a tree, etc.) is. The image associated with a map element may have the size you want. The coordinates of each element will be a positive integer from 0 to 32000.

Once created the map, we invoke the function:

```
| MAP2SP, I, Xo
```

This function analyzes the list of 64 items and determine which of them are being displayed if the world is observed by placing the bottom corner of the screen coordinates (Yo, Xo). Sprites function becomes the "map items", occupying the positions of the table sprite zero onwards. This can consume many or few sprites, depending on the density of map items you have. In another post the same function invocation, the map items that are no longer present in the scene will not consume sprites in the table, and other map items will take over. This means that the function consumes MAP2SP variable and undetermined number of sprites, which depends on the number of items visible map on the screen at all times. In the following example 3 sprites would use to invoke MAP2SP at designated coordinates.

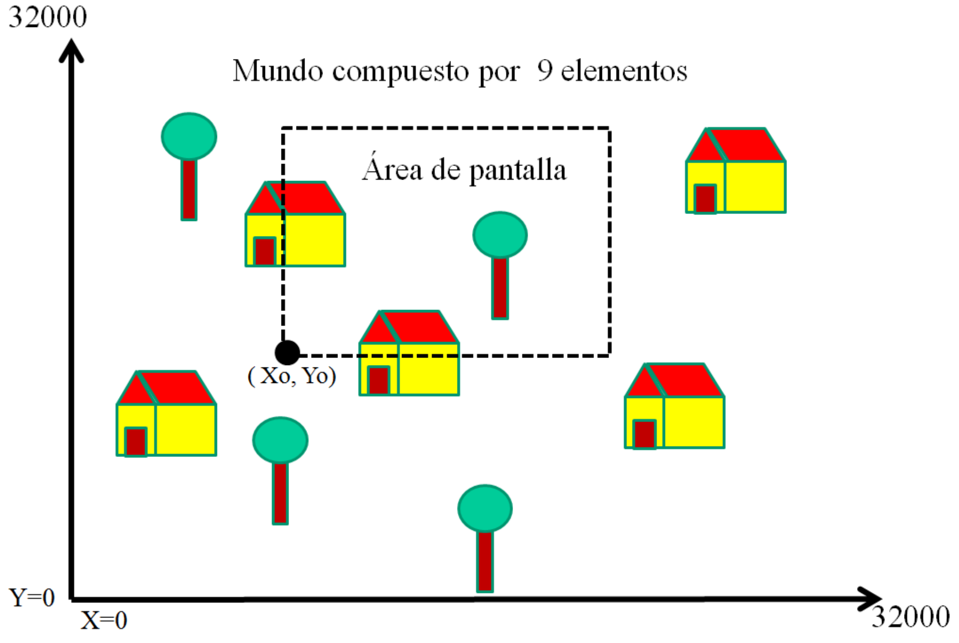


Fig. 48 World map and MAP2SP

If you use this mechanism, your character and enemies must use the sprites from 31 down thereby avoid possible clashes between sprites using the scroll mechanism and your characters.

You must invoke MAP2SP in each game cycle or at least every time you change the coordinates of the point of view from which you want to view the world.
Having clarified the concept we will review in detail the world map and an example of use of the function MAP2SP specified.

10.6.1 World map (Map Table)

The table where we will register all map elements is called MAP_TABLE and is specified in a .asm file called map_table_tujuego.asm

This table contains 64 entries defining images of the world map for your games with scroll. The table is assembled in the 26000 and contains 3 global parameters (occupying 5 bytes in total) and a list of "map items" which are described by three parameters each (x, y, image address)
The list can contain up to 64 items but may be limited to one of the global parameters.
The initial list occupies 5 bytes + 64 bytes items x 6 = 5 + 384 = 399 bytes

The table starts with 3 parameters:

- the maximum height of any map item
- maximum width of any map item (should expresearse as a negative number)
- number of items

The first two parameters are important to check when a sprite can be partially on-screen as the function MAP2SP not know or find out the width and height of each image. Only you know where is located the map item and assuming maximum height and width, see if that item may be entering the screen. If so, a sprite is created from the map item. If these two parameters are set to zero, it is necessary that the upper left corner of the map item is within the screen for that item is transformed into a sprite.

Here is an example of so-called file map_table_tujuego.asm

```
; MAP TABLE
; -----------------------
; first 3 parameters before the list of "map items"
dw 50; High maximum of a sprite if sneaks above and you have to paint
part of the
dw -40; maximum width of a sprite if strained left (negative number)
64 db; number of elements should be much mapa.como 64

; from here begin items
dw 100.10, CASA; 1
dw 50, -10, CACTUS, 2
dw 210.0, HOUSE, 3
dw 200,20, CACTUS, 4
dw 100.40, CASA; 5
dw 160,60, CASA, 6
dw 70.70, HOUSE, 7
dw 175.40, CACTUS, 8
dw 10,50, HOUSE, 9
dw 250,50, HOUSE, 10
dw 260,70, HOUSE, 11
dw 290.60, CACTUS, 12
dw 180.90, CASA; 13
dw 60,100, CASA; 14

...
```

To design your world I would suggest taking a notebook and you go checkered drawing on elements you want to have your world. Each square on the notebook can represent fixed as 8 pixels or 25 pixels an amount. The case is that you take your time to draw the world you want and how you are going to go. For example there gaunlet multidirectional type games and other vertical scroll as the command. You choose but in any case take your time and patience and the result will be worth it. Here is an example of the map of the game Commando. As you see each map is a phase. In 8BP you can change the map anytime using POKE functions, or have recorded files that you load 400 bytes in the 26000 direction maps, etc. There are many solutions to different phases without resorting to disk files, as each map takes up only 400 byets as much and can have several phases in RAM

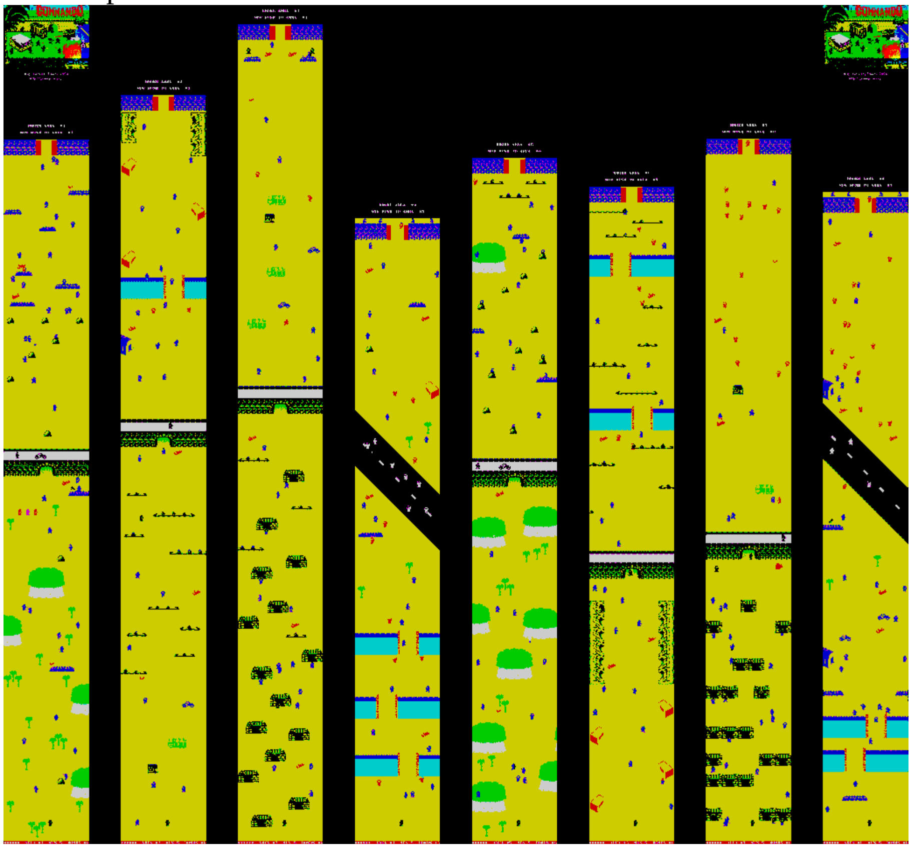


Fig. 49 Map of the classic game "Commando" (8 phases)

10.6.2 Using the function MAP2SP

Now let's see an example of using this function. You basically called once in each game cycle with the new coordinates of the origin from where the world is observed.

The function creates a variable number sprite sprites from 0 onwards and to create them will do with its screen coordinates adapted. That is, although an item map has a coordinate x = 100 if we place the mobile origin in the x = 90 then the sprite position will be created with the screen coordinate x '= x-90 = 10. The Y axis coordinates will note that the Y axis in the amstrad grows downward, while the world map grows up. Therefore the Y coordinate is adapted using the equation Y '= 200- (Y-yorig). But do not worry, this adaptation and makes MAP2SP function. You just have to go changing the mobile home where you should view the world map.

In this minigame has made a world composed of houses and cactus and our character walks between the elements. In this example, in case of collision (detected with COLSPALL), the character can not continue. In a game of aircraft in which map items are "sobrevolables" could parameterize the collision so that only collisions with enemies and shots are detected and not background elements, using COLSP, 32, <initial sprite>, <sprite_final> .

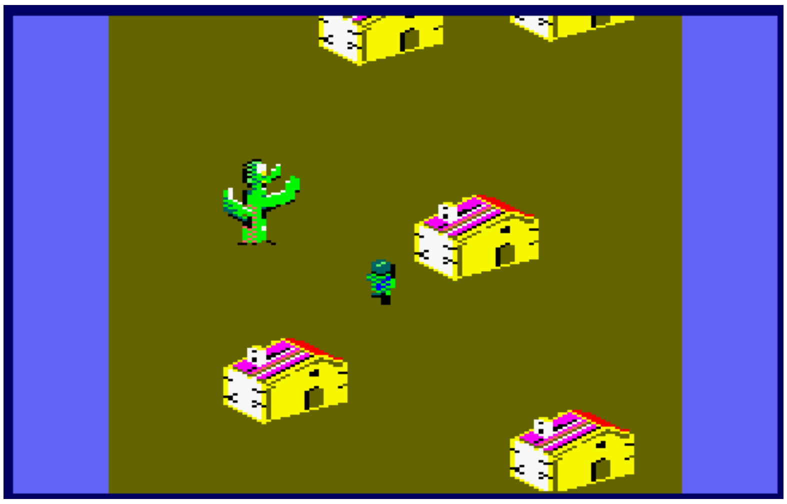


Fig. 50 Scrolling minigame inspired by "commando"

Now let's look at the list, you see, it is very small, but has everything: multidirectional scroll, reading keyboard, change the character animation sequences, collision detection, music ...

```
10 MEMORY 25999
20 MODE 0
30 ON BREAK GOSUB 280
40 CALL & 6B78
50 DEFINT az
INK 60 0.12
70 FOR y = 0 TO 400 STEP 2
PLOT 80 0, and 10: DRAW 78, and
90 PLOT 640-80, and 10: DRAW 640, and
100 NEXT
110 x = 0: y = 0
120 | SETUPSP, 31.0, & X100001
130 | SETUPSP, 31,7,1: dir = 1: 'initial upward direction
140 | locatesp, 31,100,36
150 | MUSIC, 1.5
160 | setLimits, 10,70,0,199: | PRINTSPALL, 0,1,0
170 col% = 32: sp = 32%: | COLSPALL, sp @% @ col%
180 | COLSP, 34, 0, 0: REM collision as there is a minimal overlap
190 'starts game cycle
```

```
200 IF INKEY (27) = 0 THEN x = x + 1: IF dir <> 3 then dir = 3: |
SETUPSP, 31,7,3: GOTO 220
210 IF INKEY (34) = 0 THEN x = x-1: IF x <0 x = 0 THEN-ELSE IF dir <>
4 THEN dir = 4: | SETUPSP, 31,7,4
220 IF INKEY (67) = 0 THEN y = y + 2: IF x = x AND dir <> 1 THEN dir =
1: | SETUPSP, 31,7,1: GOTO 240
230 IF INKEY (69) = 0 THEN y = y-2: IF y <0 THEN y = 0: x = xa ELSE IF
AND dir <> 2 THEN dir = 2: | SETUPSP, 31,7,2:
240 IF x = x = y THEN AND and dir = 0 ELSE | ANIMA, 31
250 | MAP2SP, y, x: | COLSPALL: IF col <32 THEN x = x: y = ya: |
MAP2SP, and x = x x ELSE: ya = y
260 | PRINTSPALL
270 GOTO 200
280 | MUSICOFF: MODE 1: 0.0 INK: PEN 1
```

Let's now look at another example of horizontal scroll, which has achieved an interesting effect map "infinite" world, making the end of the map is the same at the beginning and causing an abrupt jump when Xo reaches a certain value. In fact the world map only 13 items

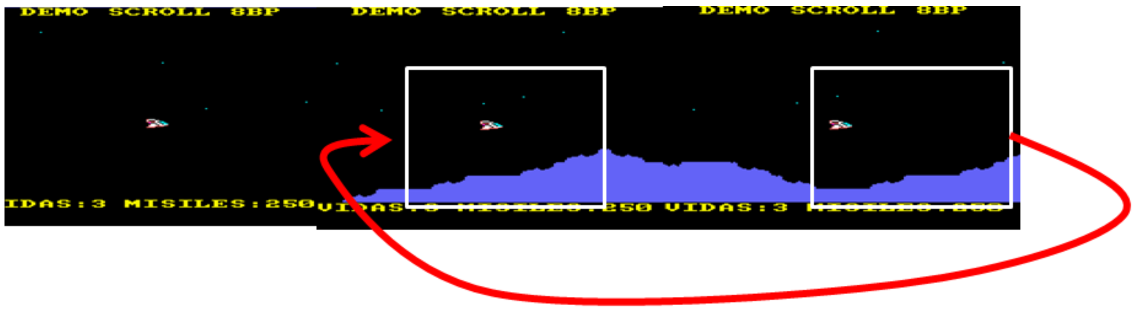


Fig. 51 World map "infinite"

This is the map that has been used

```
_MAP_TABLE
; first 3 parameters before the list of "map items"
dw 50; High maximum of a sprite if sneaks above and you have to paint
part of the
dw -18; maximum width of any map item. It must be expressed as a
negative number
db 13; 26; 13; number of map elements to consider. how much should be
64

; from here begin items
dw 36.80, MONTUP; 1
dw 48,100, MONTUP, 2
60.120 dw, MONTUP; 3
dw 72,130, MONTUP, 4
dw 72.140, MONTDW; 5
dw 60.160, MONTH, 6
60.180 dw, MONTDW; 7
48.190 dw, MONTDW; 8
; Here I repeat elements to fit the position 100
48.210 dw, MONTUP; 9
```

```
dw 60.230, MONTUP, 10
dw 72.240, MONTUP, 11
dw 72.250, MONTDW, 12
dw 60.270, MONTH, 13
; -------------------------
```

And this the BASIC program where I highlighted the line on which the world turns back without the player notice anything.

```
10 MEMORY 25999
11 FOR dir = 42540 42618 TO STEP 2: POKE dir, 20 + RND * 110: POKE dir
+ 1, RND * 80: NEXT
20 MODE 0
30 ON BREAK GOSUB 280
40 CALL & 6B78
50 DEFINT az
51 INK 0.0
52 | MUSIC, 0.5
110 x = 0: = 0
111 x = 36 y = 100
120 | SETUPSP, 31.0, & X100001
130 | SETUPSP, 31,7,1: dir = 1: 'initial upward direction
140 | LOCATESP, 31, and x
160 | setLimits, 0,80,0,176: | PRINTSPALL, 0,1,0
161 LOCATE 1,23: PEN 1: PRINT "LIVES: 3 MISSILES: 250"
162 LOCATE 1,1: PRINT "DEMO SCROLL 8BP"
170 col% = 32: sp = 32%: | COLSPALL, sp @% @ col%
180 | COLSP, 34, 0, 0: REM collision as there is a minimal overlap
190 'starts game cycle
200 IF INKEY (27) = 0 THEN x = x + 1: GOTO 220
210 IF INKEY (34) = 0 THEN x = x-1: IF x <0 THEN x = 0
220 IF INKEY (69) = 0 THEN y = y + 2: GOTO 240
230 IF INKEY (67) = 0 THEN y = y-2: IF y <0 THEN y = 0
240 IF x = x = y THEN AND and dir = 0 ELSE | ANIMA, 31
250 | MAP2SP, yo, xo: | COLSPALL: IF col <32 THEN END
260 | PRINTSPALL
+1 261 cycle = cycle: THEN IF cycle = 2 | STARS, 0,5,2,0, -1: cycle =
0
262 xo = x + 1: IF THEN xo xo = 210 = 100
263 | LOCATESP, 31, and x
270 GOTO 200
280 | MUSICOFF: MODE 1: 0.0 INK: PEN 1
```

11 Music

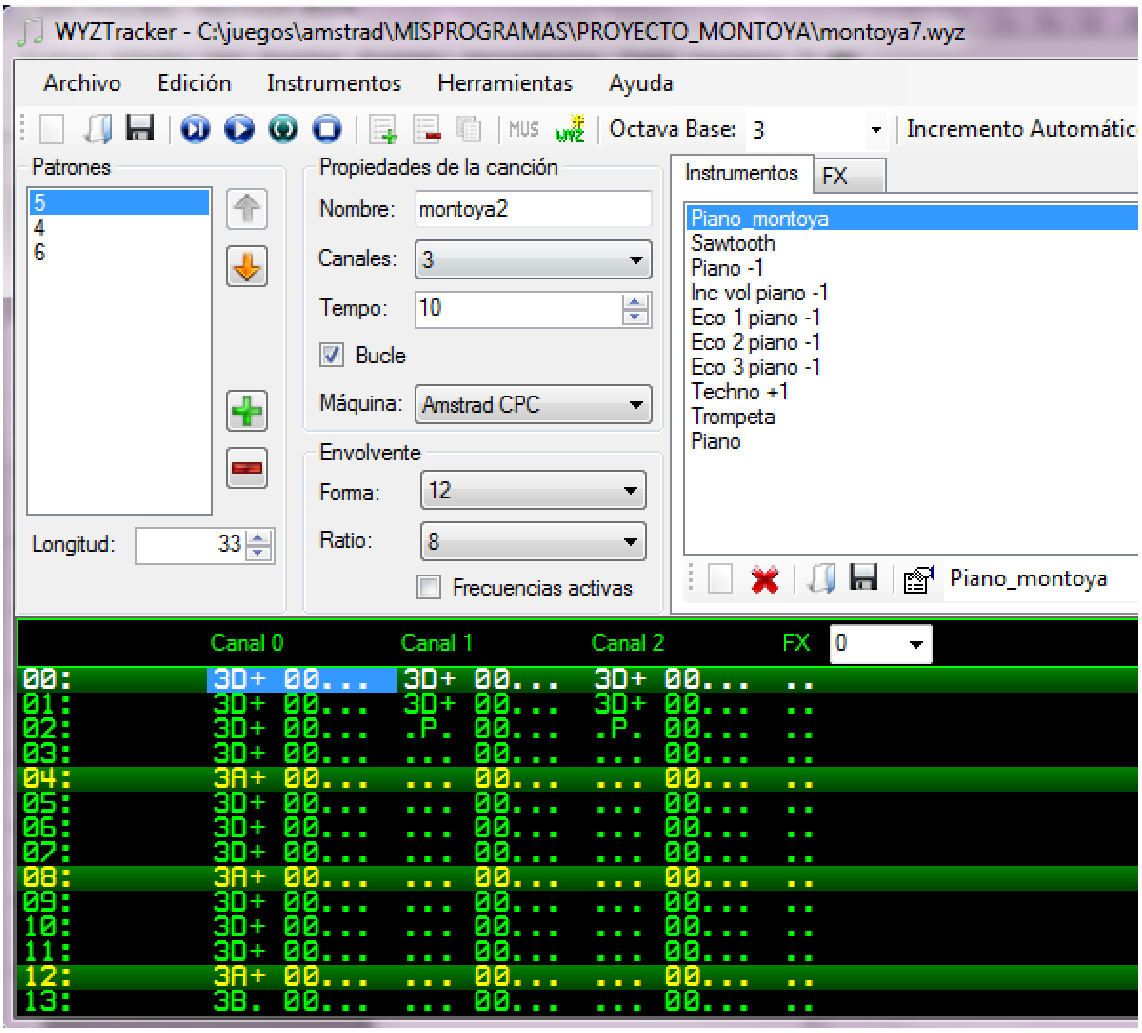
The tools we'll talk in this section do not I have, but are integrated into 8BP and are really good.

11.1 Edit music with tracker WYZ

This tool is a music sequencer for AY3-8912 sound chip. Musics generated can be exported and result in two files

- Instruments file ".mus.asm"
- A file of musical notes ".mus"

You can compose songs with this tool and you'll only limitation is that all the songs that you integrate into your game must share the same file instruments



WYZTracker - C:\juegos\amstrad\MISPROGRAMAS\PROYECTO_MONTOYA\montoya7.wyz

Octava Base: 3

Patrones: 5, 4, 6

Longitud: 33

Propiedades de la canción
Nombre: montoya2
Canales: 3
Tempo: 10
☑ Bucle
Máquina: Amstrad CPC

Envolvente
Forma: 12
Ratio: 8
☐ Frecuencias activas

Piano_montoya
Sawtooth
Piano -1
Inc vol piano -1
Eco 1 piano -1
Eco 2 piano -1
Eco 3 piano -1
Techno +1
Trompeta
Piano

Piano_montoya

FX 0

```
00:  3D+ 00...  3D+ 00...  3D+ 00...  ..
01:  3D+ 00...  3D+ 00...  3D+ 00...  ..
02:  3D+ 00...  .P. 00...  .P. 00...  ..
03:  3D+ 00...  ... 00...  ... 00...  ..
04:  3A+ 00...  ... 00...  ... 00...  ..
05:  3D+ 00...  ... 00...  ... 00...  ..
06:  3D+ 00...  ... 00...  ... 00...  ..
07:  3D+ 00...  ... 00...  ... 00...  ..
08:  3A+ 00...  ... 00...  ... 00...  ..
09:  3D+ 00...  ... 00...  ... 00...  ..
10:  3D+ 00...  ... 00...  ... 00...  ..
11:  3D+ 00...  ... 00...  ... 00...  ..
12:  3A+ 00...  ... 00...  ... 00...  ..
13:  3B. 00...  ... 00...  ... 00...  ..
```

Fig. 52 WYZTracker

This music sequencer is complemented by the WyZplayer which is integrated in the 8BP library.

Ls problems I've found you the detail here. you probably the same so you already have the solutions to these difficulties do not make you suffer happen

- When editing your song with WYZ tracker you must include an initial note in the 3 channels. This will include a first note is not part of the music should serve to initialize the 3 channels of sound in the WYZplayer. When you hear the song on the WyZtracker, that note will hinder you but will not ring in the integrated player in 8BP. You fijate in the screenshot that I have made, you'll see that first note played with the instrument id = 0 in this case is "piano_montoya"

- In addition, a channel should not change the instrument throughout the song. If you do not respect this limitation, the music will gradually accelerating. Be careful with this. Check that each channel only a single instrument.

These problems have no negative effect if we fulfill these rules. Otherwise it is possible that the song does not sound like it should or has not sound channels.

11.2 Assemble songs

Once you have made your song and you have the two files, just edit the file make_music.asm and include your music files as follows:

```
; if you join this independently
; should be at least where the code ends and the player 8bp, checking
; where _FIN_CODIGO tag is assembled.
; assuming that is less than 32000 (actually is somewhat less, you can
join in 32000)
; assemble after I save it with the save "musica.bin" b, 32000.1500
org 32000
;--------------------MUSIC---------------------------- --------
; It has the limitation of only being able to include a single file
; instruments for all songs
; the limitation is solved by simply putting all
; instruments into a single file.

; Instruments file. EYE HAS TO BE ONLY ONE
read  "Instrumentos.mus.asm"

; music files
; eye the first note should sound in the 3 channels and also will
never be repeated
; IMPORTANT this special note should be the instrument with id = 0
(edit using WYZ tracker)
; if another instrument gives me problems.
SONG_0:
INCBIN      "Micancion.mus";
SONG_0_END:

SONG_1:
INCBIN      "Otra_cancion.mus";
SONG_1_END:

SONG_2:
INCBIN      "Tercera_cancion.mus";
```

```
SONG_2_END:
SONG_3:
SONG_4:
SONG_5:
SONG_6:
SONG_7:
```

Finally re-8BP you join the library for the player of music (which is integrated into the library) knows the parameters of instruments and where the songs have been assembled.

To do so simply you join the make_all.asm file that looks like this

```
; Makefile for games using 8bit power
; if you alter only part you just have to assemble the corresponding
make
; for example you can join if you change the make_graficos drawings
; ---------------------------- -------------------- CODE -
And it includes the library 8bp and playerWYZ of music
read "make_codigo.asm"

;--------------------MUSIC---------------------------- --------
; It includes the songs.
read "make_musica.asm"

; ---------- GRAPHICS -------------------
; This part includes images and animation sequences
; sprites and table initialized with these images and sequences
read "make_graficos.asm"
```

and this already have everything assembled. Now you must generate your library 8BP as follows:
save "8BP.LIB", b, 27000.5000

and the musics:

save "MUSIC.BIN", b, 32000.1500

12 Reference Guide 8BP library

12.1 Library functions

12.1.1 | ANIMA

This command changes the frame of a sprite animation, taking into account its animation sequence assigned

Use:
| ANIMA, <sprite number>

Example:

| ANIMA, 3

The command does is check the sprite animation sequence, and if different from zero then goes to the animation sequence table (the first sequence is valid 1 and the last is 31). Choose the image whose position is following the current frame and update the frame field sprite attribute table.
If in the next frame secuenta it is zero then cyclized, ie, the first frame of the sequence is chosen.

In addition to changing the frame field, the image field is changed and assigned the memory address of where the new frame is stored.

| ANIMA not print the sprite but leaves it ready for when printed, so that printed the next frame in sequence

| ANIMA does not verify that the animation flag is active in the status byte sprite. In fact our character usually only we will want to cheer when moving and not always to be printed.

If the animation sequence is a "sequence of death" (includes a "1" in the last frame), then on reaching the frame whose image memory address is 1, the sprite becomes inactive.

The 8BP library allows you to "death sequences" that are sequences that at the end of roam around, the sprite becomes inactive. This is indicated by a simple "1" as the value of the memory address of the final frame. These sequences are very useful to define explosions enemies are animated with ANIMA or ANIMAALL. After catch up with your shot, you can associate a death animation sequence and subsequent cycles of the game will go through the various stages of animation of the explosion, and when you reach the last pass inactive, not in print anymore. This step-idle is done automatically, so you have to do is simply check the collision of your shot with enemies and if it collides with any of you change the state with SETUPSP so you can not collide more and assign the animation sequence death, also with SETUPSP

If you use a sequence of death, do not forget that the last frame before finding the "1" is a completely empty, so remove all traces of the explosion.

Example death sequence

```
dw EXPLOSION_1, EXPLOSION_2, ExPLOSION_3,1,0,0,0,0
```

12.1.2 | animall

This command encourages all sprites that have the flag enabled animation in the status byte. This command has no parameters

Use
| animall

It is advisable to use if you are going to animate many sprites as it is much faster than calling several times a command | ANIMA

As usually will want to invoke animall in each game cycle before printing sprites, there is a way to invoke more efficient and is set to "1", the corresponding parameter of the function PRINTSPALL, ie

| PRINTSPALL, 1.0

This function invokes internally before printing animall sprites, saving 1.17ms about what it would take to invoke separately | animall and | PRINTSPALL

12.1.3 | AUTO

This command moves a sprite (change its coordinates) according to their attributes Vy, Vx

Use:
| AUTO <sprite number>

Example:
| AUTO 5

What makes this command is to update the coordinates in Table sprite, adding the speed at current coordinate

The new coordinates are
new X = X + coordinate current Vx
Y new = Y coordinate Vy current +

It is not necessary that the sprite has the automatic movement active flag in the status field

12.1.4 | AUTOALL

This command moves all sprites that have the flag active automatic movement, according to their attributes Vy, Vx. This command has no parameters

Use:
| AUTOALL

12.1.5 | colay

Detects the collision of a sprite with the map screen (the layout). It takes into account the size of the sprite to see if it collides.

Use:
| Colay <threshold ASCII>, <sprite number>, @ collision%
O well:
| Colay, <num_sprite>, <@ collision%>

The optional parameter <threshold ASCII> just use in a first call to set the threshold collision in the colay command. This threshold represents the largest ASCII code layout element that is regarded as "no collision". The default is 32 (the blank)

Example:
| Colay, 0, @ collision%

The variable that you use for collision can be called as you want.

This routine modifies the collision variable (which must be whole and therefore the "%") putting it to 1 if the sprite collision indicated by the layout. If no collision the result is 0.

```
xanterior = x
x = x + 1
| LOCATESP, 0, y, x, 'we position the sprite in new position
| Colay, 0, @ collision%: 'check the collision
```

Now we check the collision and collision if we leave it in its previous location

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

If our character can move diagonally, often we will want that pressing right + up our sprite forward in one direction while the other has a wall and locks. This gives a greater sense of fluidity to the movement although complicated logic. Here we show how, from 1721. First line stands sprite with LOCATESP and then depending on collisions are detected colay repositions the sprite again with LOCATESP

In this example the variables have the following meaning
X: x coordinate above

and: coordinate and above
Xn: x coordinate new
n: coordinate and new

```
1500 character movement routine ------------------------
1510 if INKEY (27) <> 0 goto 1520

1511 if INKEY (67) = 0 then IF dir <> 2 THEN | SETUPSP, 0,7,2: dir =
2: goto ELSE 1533 | ANIMA, 0: xn = x + 1: n = ya-2: goto 1533

1512 if INKEY (69) = 0 THEN IF dir <> 8 THEN | SETUPSP, 0,7,8: dir =
8: goto ELSE 1533 | ANIMA, 0: xn = x + 1: n = and + 2: goto 1533

IF 1513 dir <> 1 THEN | SETUPSP, 0,7,1: dir = 1: goto ELSE 1533 |
ANIMA, 0: xn = x + 1: goto 1533

1520 if INKEY (34) <> 0 goto 1530
```

```
1521 if INKEY (67) = 0 THEN IF dir <> 4 THEN | SETUPSP, 0,7,4: dir =
4: goto ELSE 1533 | ANIMA, 0: xn = x-1: n = ya-2: goto 1533

1522 if INKEY (69) = 0 THEN IF dir <> 6 THEN | SETUPSP, 0,7,6: dir =
6: goto ELSE 1533 | ANIMA, 0: xn = x-1 and n = + 2: goto 1533

IF 1523 dir <> 5 THEN | SETUPSP, 0,7,5: dir = 5: goto ELSE 1533 |
ANIMA, 0: xn = xa-1: goto 1533

1530 IF INKEY (67) = 0 THEN IF dir <> 3 THEN | SETUPSP, 0,7,3: dir =
3: goto ELSE 1533 | ANIMA, 0: n = ya-4: goto 1533

1531 IF INKEY (69) = 0 THEN IF dir <> 7 THEN | SETUPSP, 0,7,7: dir =
7: goto ELSE 1533 | ANIMA, 0: n = and + 4: goto 1533

1532 return

1533 | LOCATESP, 0, n, xn: ynn = n: | colay, 0, @ cl%: IF cl% = 0 then
1450

1534 n = ya: | LOCATESP, 0, n, xn: | colay, 0, @ cl%: IF cl% = 0 then
1450

1535 xn = x: n = ynn: | LOCATESP, 0, n, xn: | colay, 0, @ cl%: IF THEN
cl% = 1 and n = ya: | LOCATESP, 0, n, xn

1536 and a = n: x = xn

1537 return
```

12.1.6 | COLSP

This command allows to detect the collision of a sprite with other sprites that have the flag active collision

use:
To configure:

```
| COLSP, 32, <initial sprite>, <end sprite>
| COLSP, 33, @ collision%
| COLSP, 34, dy, dx
```

To detect collisions:

```
| COLSP, <sprite number>, @ colsp%
```

Example:

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

The function returns the variable we pass as a parameter, the number of sprite collides with, or if no collision returns 32 because the sprite 32 does not exist (there are only 0 to 31)

As printing PRINTSPALL sprites, the function checks COLSP sprites starting at 31 and ending at zero. If collision flag have active sprites (bit 2 of byte status) then the collision is checked. Di two sprites collide at once with our sprite, the sprite number greater returns as it is being checked before.

Invocations to configure the command:
There is a way to set COLSP to perform less work checking the collision of sprites and thus save less runtime. The configuration using the indicate sprite 32 (which does not exist).

| COLSP, 32, <initial check sprite>, <sprite final check>
If for example the enemies of our character are the sprites 25 to 30, we can invoke (once) to so command:

```
| COLSP, 32, 25, 30
```

With that we will be indicating that any post-command invocation | COLSP should only cherquear the collision of sprites 25 to 30 (provided they are active collision flag).
This strategy reduces long time, for example if we are only 6 enemies in check, foreshadowing the command to only check from 25 onwards, we can save up to 2.5ms in each run. This is especially important in games where the character can shoot, because in each game cycle at least have to check the collision of character and shots.

Another interesting optimization, able to save 1.1 milliseconds on each invocation, is to tell the command that you always use the same variable BASIC to make the result of the collision. To do this we will tell you using as sprite 33, which does not exist

```
col = 0%
| COLSP, 33, @Col%
```

Once executed these two lines, the following invocations COLSP, will leave the result in the col variable, without indicate, for example:

| COLSP, 23

Finally it is possible to adjust the sensitivity of COLSP command, deciding whether the overlap between sprites must be several pixels or one to consider that there has been a collision.

For this you can set the number of pixels of overlap necessary both in the Y direction and the X direction, using the COLSP command and specifying the sprite 34 (which does not exist)

| COLSP, 34, dy, dx

The default values for dx and dy are 2 and 1 respectively. Note that in the Y direction are considered pixels but in the X direction are considered bytes (one byte is two pixels in mode 0)

For detection with minimal overlap (of a pixel vertically and / or horizontally byte) you do:

| COLSP, 34, 0, 0

12.1.7 | COLSPALL

Use
To configure:
    | COLSPALL, @ collider% @ collided%
To check for collisions
    | COLSPALL

This function checks who has collided (among the group of sprites that have a "1", the flag collider byte status) and who has collided (among the group of sprites that have a "1", the flag collision byte status).

It is a highly recommended feature when you have to manage your character collisions and several shots as it saves invocations COLSP and therefore speeds up your game.

12.1.8 | LAYOUT

use:
| LAYOUT, <and>, <x>, <@ $ string>

Example:
$ string = "XYZZZZ ZZ"
| LAYOUT, 0.1, @ string $

eye, use | LAYOUT 0.1, "XYZZZZ ZZ" would be incorrect in a CPC464 although it works in a CPC6128. In addition, CPC6128 you can avoid the use of the "@" but CPC464 is required.

This routine prints a row of sprites to build the layout or "labyrinth" of each screen. In addition to drawing the maze, or any graphic display built with small sprites 8x8, you can also detect collisions of a sprite with the layout, using the command | colay

Print sprites are defined with a string, whose characters (32 possible) represent one of the sprites by following this simple rule, where the only exception is the white space that represents the absence of sprite.

Character	**Sprite id**	**ASCII code**
""	ANY	32
";"	0	59
"<"	1	60
"="	2	61
">"	3	62
"?"	4	63
"@"	5	64
"TO"	6	65
"B"	7	66
"C"	8	67
"D"	9	68
"AND"	10	69
"F"	eleven	70
"G"	12	71
"H"	13	72
"I"	14	73
"J"	fifteen	74
"K"	16	75
"L"	17	76
"M"	18	77
"N"	19	78
"OR"	twenty	79
"P"	twenty-one	80
"Q"	22	81
"R"	2. 3	82
"S"	24	83
"T"	25	84
"OR"	26	85
"V"	27	86
"W"	28	87
"X"	29	88
"Y"	30	89
"Z"	31	90

Table 6 correspondence between characters and Sprites for the command | LAYOUT

IMPORTANT: After printing the layout you can change the sprites to be characters, so you'll still have 32 sprites

Coordinates and x are passed in character format. The library internally maintains a map of 20x25 characters, so the coordinates take the following values:

and takes values [0.24]

x takes values [0,19]

Sprites to print must be 8x8 pixels. are "bricks" ("bricks" in English) .To this kind of concept is also often called "tiles" (tiles)
If you use other sizes of sprite, this feature will not work well. Actually print the sprites but if a sprite is large you will have to place blanks to make room.

The library maintains an internal map layout and this function updates the data on the internal map layout so it will be possible to detect collisions. This map is an array of 20x25 characters, where each character corresponds to a sprite

The @string is a string variable. you can not go straight chain, although the CPC6128 passing parameters permits but would be incompatible with CPC464

precautions:
the function does not validate the string you pass it. If it contains sensitive or different character than allowed can cause unwanted, such as reboot or crash the computer effects. Nor can it be an empty string!

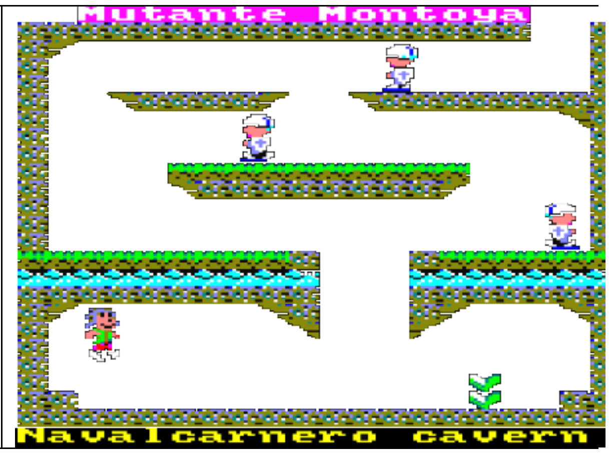
The limits established with setLimits should allow print anywhere you want. If you later want to do cliping in a smaller area can be invoked again when the entire layout setLimits this form

Example

```
2070 | setLimits, 0,80,0,200
2090 c $ (1) = "PPPPPPPPPPPPPPPPP P"
2100 $ c (2) = "PU P"
2110 c $ (3) = "P P"
C 2120 $ (4) = "P P"
C 2130 $ (5) = "P TPPPPU TPPPPPPPP"
2140 c $ (6) = "P TP"
C $ 2,150 (7) = "P P"
2160 c $ (8) = "P P"
C 2170 $ (9) = "P YYYYYYYYYY P"
2190 c $ (10) = "P TPPPPPPPPU P"
2195 c $ (11) = "P P"
2200 c $ (12) = "P P"
2210 c $ (13) = "P P"
2220 c $ (14) = "YYYYYYYYYP PYYYYYY"
2230 c $ (15) = "RRRRRRRRRR RRRRRRR"
2240 c $ (16) = "PPPPPPPPPP PPPPPPP"
2250 c $ (17) = "PU PU TP TP"
2260 c $ (18) = "OCT P"
2270 c $ (19) = "P P"
2271 c $ (20) = "P P"
2272 c $ (21) = "PW P"
2273 c $ (22) = "PP W PP"
2274 c $ (23) =
"PPPPPPPPPPPPPPPPPPPP"
```

In this example several bricks that have previously been created with the "spedit" tool used
; sprite 20 -> O bush
; sprite 21 -> P rock
; sprite 22 -> Q cloud
; sprite 23 -> R water
; sprite 24 -> S window
; sprite 25 -> T arch bridge right
; sprite 26 -> U arch bridge left
; sprite 27 -> V flag
; sprite 28 -> W plant
; sprite 29 -> X Peak Tower
; sprite 30 -> Y grass
; sprite 31 -> Z brick

```
2280 for i = 0 to 23
2281 | LAYOUT, i, 0, @ c $ (i)
2282 next
```



12.1.9 | LOCATESP

This command changes the coordinates of a sprite in the sprite attribute table

Use
| LOCATESP, <sprite number>, <and>, <x>

Example
| LOCATESP, 0,10,20

An alternative to this command, if you only want to change a coordinate is to use BASIC POKE command, inserting memory address occupied by the X coordinate or Y, the value we want. If we want to enter a negative coordinate function is required | POKE, because with the BASIC POKE would be illegal

The command | LOCATE not print the sprite, only positions it for when it printed.

12.1.10 | MAP2SP

This function runs the world map as described in the map_table.asm file and the map becomes sprites items that may be entering partially or completely screen.

Use
| MAP2SP, <and>, <x>

Example
| MAP2SP, 1500.2500

The parameters of the function are mobile source from which the world is displayed on the screen. There are three other parameters found in the MAP_TABLE, the table with which the world is defined. These parameters are the highest high, the maximum width (negative) and the number of items in the world (maximum 64)

```
; MAP TABLE
; ------------------------
; first 3 parameters before the list of "map items"
dw 50; High maximum of a sprite if sneaks above and you have to paint
part of the
```

```
dw -40; maximum width of a sprite if strained left (negative number)
64 db; number of map elements to consider. how much should be 64

; from here begin items
dw 100.10, CASA; 1
dw 50, -10, CACTUS, 2
dw 210.0, HOUSE, 3
dw 200,20, CACTUS, 4
dw 100.40, CASA; 5
dw 160,60, CASA, 6
dw 70.70, HOUSE, 7
dw 175.40, CACTUS, 8
dw 10,50, HOUSE, 9
dw 250,50, HOUSE, 10
dw 260,70, HOUSE, 11
dw 290.60, CACTUS, 12
dw 180.90, CASA; 13
dw 60,100, CASA; 14

...
```

12.1.11 | MOVER

This command moves a sprite relatively, that is, adding to its coordinates relative quantities a

Use:
| MOVE, <sprite number>, <d>, <dx>

Example:
| MOVER, 0.1, -1

Example moves the sprite 0 down and to the right at a time. It is not necessary that the sprite has the relative motion flag activated

12.1.12 | MOVERALL

This command moves so on all sprites that have the flag relative motion activated

Use
| MOVERLALL, <d>, <dx>

Example
| MOVERALL, 2.1

The example moves all sprites with flag of relative movement down (2 lines) and 1 byte to the right.

12.1.13 | MUSIC

This command allows a melody starts ringing

Use:
| MUSIC <numero_melodía>, <speed>

the number of the melody will be between 0 and 7
the "normal" speed is 5. If we use a higher number will play slower and if the number is lower will be played faster

Example:
| MUSIC, 0.5

Internally, the command does is install an interrupt is triggered 300 times per second. If we speed 5, one of every 5 times you shoot, the music playback function is executed

By relying on a break, there needs to be a running program so you can sound music, but as the BASIC interpreter is waiting to receive commands, such interruptions are not enabled. If you run just the command | MUSIC not hear anything, but if you run into a program like the one shown below, the music will sound

```
10 | MUSIC, 0.5
20 goto 20: 'infinite loop. To be running, music sounds
```

12.1.14 | MUSICOFF

This command paralyzes playing any melody. No parameters

Use:
| MUSICOFF

Internally it does is remove the interruption

12.1.15 | PEEK

This command reads the value of a 16 bit data from a given memory address. It is intended to check the coordinates of sprites that move with automatic or relative movement

Use
| PEEK, <address>, @ data%

Example
data% = 0
| PEEK, 27001, @ data%

if the coordinates are positive and just under 255 PEEK can use the BASIC command because it is something faster.

12.1.16 | POKE

This command introduces a 16-bit data (positive or negative) in a memory address. It is intended to modify coordinates of sprites as the POKE command can not handle negative coordinates or greater than 255 as bytes POKE operated while | POKE is a command that works with 16bit

Use:
| POKE, <address>, <value>

Example:
| POKE, 27003.23

This example the value 23 in the x coordinate of sprite 0.
It is a very fast function even if you manage only positive coordinates is better to use POKE it is even faster

12.1.17 | PRINTSP

Use:
| PRINTSP, <sprite id>, <and>, <x>

Example
prints the sprite 23 at coordinates y = 100, x = 40

| PRINTSP, 23,100,40

This routine prints a sprite on the screen, but not updated the coordinates of the sprite sprite in the table.
The coordinates are considered
Number of lines vertically [-32768..32768]. the corresponding inside the screen are [0..199]
Number of bytes in the horizontal [-32768..32768]. the corresponding inside the screen are [0..79]

Normally in the logic of a vieojuego you do use | PRINTSPALL because it is faster to print them all at once. But at other times of the game you might want to print sprites separately. In this example the descent of a "curtain" shown using a single sprite that is repeated horizontally and going down is "dying" red screen, giving the feeling of a curtain coming down

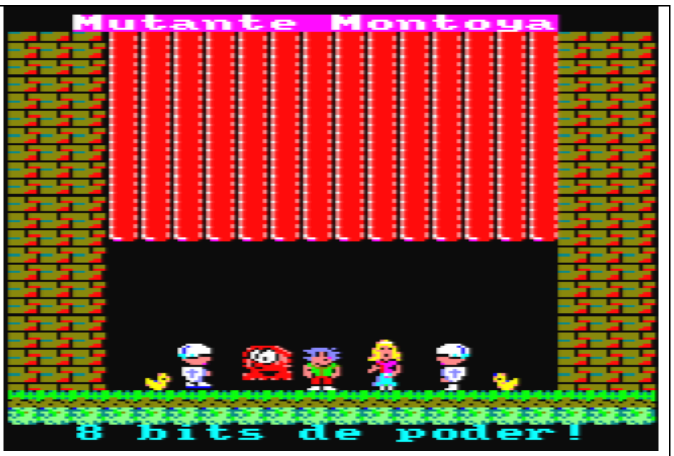
`7089 telon = & 8ec0`
`7090 │ setupsp, 1.9, telon`
`7100 for y = 8 to 168 step 4`
`7110 for x = 12 to 64 step 4`
`7111 │ PRINTSP, 1, y, x`
`7112 next`
`7113 next`
	

Fig. 53 An example of use of PRINTSP

12.1.18 | PRINTSPALL

Use:
| PRINTSPALL, <order>, <flag anima>, <flag sync>

Example:
prints all sprites encouraging them first and unsynchronized with sweeping, and unordered
| PRINTSPALL, 0, 1, 0

This routine prints once all sprites having bit0 active state. If set to 1, the flag animation, then before printing sprites frame is changed in your animation sequence, provided that the sprites have the active status bit 3

The <flag sync> is a flag synchronization with the scanning screen. It can be 1 or 0. The timing makes sense only if you compile the program with a compiler as "Fabacom". BASIC logic runs slowly and synchronize with the scanning produces small additional waits at each cycle of the game so that it is not convenient.
As a rule, it is only convenient if your game is able to generate 50fps per second, or what is the same, a game full cycle every 20 milliseconds. If you compile the game with a compiler as "fabacom", then it is recommended that you synchronize with sweeping screen because almost certain that you will achieve these 50fps and if you pass, your game will produce more frames than it can display the screen and then some can not be displayed and the movement will not be smooth.

few have more sprites on screen in print, the longer the command, although it is very fast. There are many sprites that can appear on screen but do not need to print (may have bit 0 OFF state) such as fruits, coins, bonus items in general and / or characters that do not move and do not have animation. Although not print may have bit active collision and thus affect the routine | COLSP

Finally we have the order parameter. We must indicate the number of sprites sorted by Y coordinate that we will print. For example if we put a 0, then the sprites are printed sequentially from the sprite sprite 0 to 31. If we put a 10 art print is from 0 to 10 ordered (11 sprites) and from 11 to 31 sequentially. If we all ordered 31 sprites are printed.

The system is very useful for games or golden renegade type AXE, where is necessary to give a depth effect.

| PRINTSPALL, 0, 1, 0: sequentially prints all sprites
| PRINTSPALL, 31, 1, 0: print neatly all sprites
| PRINTSPALL, 7, 1, 0: prints in an orderly and sequential 8 sprites rest

Print orderly computationally more costly to print sequentially. If you have just 5 sprites that must be ordered, pass a 4 as a parameter ordering, not spend a 31. Sort all sprites takes about 2.5ms but if you order just 5 you can save 2ms. Maybe you have many sprites and not worth ordering some, such as shots or sprites you know you are not going to overlap.

There is a very interesting behavior of this 1ms to save in execution function. It is to invoke parameters once and invoke the following times without parameters. In that case they assume that if no parameters are passed, their values are equal to the last to be passed. Thus the parser works less and reduces the runtime.

12.1.19 | ROUTEALL

This command allows you to route all sprites that have active route flag in its status byte through their assigned route (parameter 15 SETUPSP)

Use
| ROUTEALL

No parameters so it is very easy to invoke. This command does internally is to take account of steps by the segment that is studying each sprite, so that if the segment is finished, alters the speed of the sprite.
The command does not change the coordinates of the sprites, so they must be moved with AUTOALL and printed (and animated) with PRINTSPALL.

The routes are defined in the routes file routes_tujuego.asm

```
; DEFINITION OF EACH ROUTE
; ========================
ROUTE0; a circle
; -----------------
      db 5,2,0
      db 5,2, -1
      db 5,0, -1
      db 5, -2, -1
      db 5, -2.0
      db 5, -2.1
      db 5,0,1
      db 5,2,1
      db 0
```

The last segment is zero, indicating that the route has been completed and that the sprite should start from the princpio.

12.1.20 | setLimits

This command sets the boundaries of the area where they will be able to print sprites or stars.

Use:
| SetLimits, <xmin>, <xmax>, <ymin>, <ymax>

Example that sets the entire screen as permitted area
| SetLimits, 0,80,0,200

Outside these limits is carried clipping sprites, so that if a sprite is partially outside the permitted area, the functions | PRINTSP and | PRINTSPALL print only the part that is within the permitted area.

12.1.21 | SETUPSP

load data from a sprite this command in the SPRITES_TABLE
Use:
| SETUPSP, <id_sprite>, <param_number>, <value>

Example
| SETUPSP, 3, 7, 2

It allows for example to assign a new animation sequence when the sprite changes direction, or simply change their registration status flags.
With this function we can change any parameter of a sprite, less X, Y (done with LOCATE_SPRITE)
We can only change one parameter at a time. The parameter that will change is specified with param_number. The param_number is actually the relative position of the parameter in the SPRITES_TABLE

- param_number = 0 -> change the status (occupies 1 byte)
- param_number = 5 -> change Vy (takes 1 byte, value vertical lines). You can also change Vx at a time if we add the fnal as a parameter
- param_number = 6 -> change Vx (takes 1 byte, value in horizontal bytes)
- param_number = 7 -> change sequence (takes 1 byte, takes values 0..31)
- param_number = 8 -> change frame_id (takes 1 byte, takes values 0..7)
- param_number = 9 -> dir image changes (occupies 2 bytes)

Example
in this example we have given the 31 sprite image of a ship is assembled in the direction & a2f8

ship = & a2f8

| SETUPSP, 31.9, ship

In the case of param_number = 5, we can include Vx as a parameter to the end:
| SETUPSP, 31.5, Vx, Vx

Thus we will update the two speeds with a single command, which costs 3.73ms versus 6.8 would cost invoke two commands separately.

In case of param_number = 7, in addition to changing the animation sequence automatically command updates the frame, placing it in the initial (zero) and the direction of the image is updated, so you no longer need to invoke param_number = 9 to change the image of the sprite to the first image of the newly assigned sequence. If you are using |ANIMALL before printing or |PRINTSPALL with flag animation, although SETUPSP set you animation at frame zero, you are jumping to frame 1 before printing. This will not normally be a problem, but in case of a sequence of death in which for example the first frame is designed to delete the current sprite, you might not want to jump directly to frame 1. In that case a simple trick to solve this problem consists of repeat the zero frame in the definition of the death sequence. So you make sure that the frame zero is displayed. Another option is to remove the flag animation and use ANIMASP after printing.

12.1.22 | SETUPSQ

This command creates an animation sequence

Use:
| SETUPSQ <sequence number>, <address1>, <address N>, 0, ..., 0
Must be filled in 8 directions or complete up to 8 addresses with zeros
The sequence number must be between 1 and 31

Example that creates the sequence 1 with the 4-way where there assembled images (frames) of an animated character

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

Important: with the command SETUPSQ not be created macrosequences animation, you must create them define in the file sequences_tujuego.asm

12.1.23 | STARS

Move a bank of 40 stars on the screen (within the limits set by | setLimits), unpainted other sprites that already existed printed.

| STARS, <initial star>, <num stars>, <color>, <d>, <dx>

Example

| STARS, 0, 15, 3, 1, 0

Example 15 moves colored stars 3 (red) pixel vertically (as dx dy = 1 = 0). Invoked repeatedly gives a sense of background of stars that moves. When a star leaves the edge of the screen or set by | setLimits reappears on the opposite side, so there is a sense of continuity in the flow of the stars.

The bank-star hotel in the direction 42540 (= & A62C) and accommodates 40 stars, reaching the address 42619. Each star consumes 2 bytes, one for the Y coordinate and one for the X coordinate

You can move groups of stars separately, starting at the star you want. The initial coordinates of the stars must be initialized by the programmer.

Example initialization and use in a scroll of four planes of stars to give a sense of depth. Each plane will move at a different speed

```
10 CALL & 6b78: rem installs the RSX commands
20 bank = 42540
30 FOR star = 0 TO 39: 'loop to create 40 stars
40 POKE bank + star * 2 * 200 RND
50 POKE bank + star * 2 + 1, RND * 80
60 NEXT
70 MODE 0
80 REM now going to paint and move 4 stars planes 10 stars each
90 | STARS, 0,10,3,0, -1: '3 is red. The most distant move more slowly
91 | STARS, 10,10,2,0, -2 '2 is blue
92 | STARS, 20,10,1,0, -3 '1 is yellow
93 | STARS, 30,10,4,0, -4 '4 is white. Closest go more quickly
95 goto 90
```

The uses of this command can be very diverse.

- Using several banks of stars at once with different speed and color you can give a sense of depth
- If the direction of the stars is diagonal you can make a "rain effect"
- If the color is black and the background is brown or orange can give a sense of progress on a sandy territory
- If the movement is rolling and color of the stars is white can give a feeling of snow. The rocking motion you can achieve with a zigzag maintaining speed X to Y, or even using trigonometric functions such as cosine. Obviously if you use the cosine in the logic of a game will be very slow but you can store the precalculated cosine value in an array.
  Example snow effect:

```
1 MODE 0
30 'initialisation bank of 40 stars
40 FOR dir = 42540 42619 TO STEP 2
45 POKE dir, RND * 200: POKE dir + 1, RND * 80
48 NEXT
50 | STARS, 0,20,4,2, dx1
60 | STARS, 20,20,4,1, dx2
61 dx1 = 1 * COS (i): dx2 = SIN (i)
69 i = i + 1: 359 THEN IF i = i = 0
```

```
70 GOTO 50
```

There is a way to achieve faster execution, and is avoiding passing parameters. Throughout this book we have seen how parameter passing is expensive even if the invoked command does nothing. As we are well, before a command that requires 5 parameters which is especially costly. If we want to reduce the time required to interpret the BASIC parameters, we can simply invoke once the command with the following parameters and sometimes not pass parameters.

| STARS, 0,10,1,5,0
| STARS: 'this invocation without parameters assumes the same values of the last invocation

This possibility is especially useful in games where we want to invoke the command on each game cycle to move stars, because we will save about 1.7 ms

13 Assembled from the bookstore, graphics and music

Whether you want to make changes in the library as if you add music and graphics you need reassembling, This is because for example, the player music is integrated into the library and need to know where (memory address) begins each song, so necessary reassembling and save the library version specified for your game as well as the assembly file graphics and music file assembly.

As I explained in the section on "steps" you should give, this will be a version of the library specified for your game. For example, the command | MUSIC, 3.5 will sound the melody number 3 you yourself have made. The melody number 3 can be completely different in another game. The same applies to the file data instrument. There are certain dependencies between the code of the music player and addresses where data from instruments and melodies are assembled.
It's simple but you have to understand the structure of the library to do

The first thing to be clear is that your game is composed of three binary files:

- 8BP library (it is a binary file), including sprite attribute table
- binary file of music, with the melodies of your game
- binary image file sprites, including animation sequence table

And two BASIC files:

- Charger (load library, music and sprites and finally your game)
- BASIC program (your game)

In this section we will focus on how to generate 3 binary files you need. To generate the 3 files must first assemble all (now say I like) and then you have all assembled in memory you execute these commands to generate files. You see these commands simply take fragments of memory and saved in separate files.

```
SAVE "8BP.LIB", b, 26000.5000
SAVE "MUSIC.BIN", b, 32000.1500
SAVE "SPRITES.BIN", b, 33500.8500
```

It now remains to understand how to assemble everything (library, music and graphics) memory. To do this you must understand the structure of the .asm files that you must handle and its dependencies. A single file .BIN actually going to require more than one .asm file to generate it.

The following diagram shows all files .asm a game that use 8BP well as the dependencies between them.

Gray those files you have to edit appear, such as:

- file songs and instruments, which generate the WYZtracker
- where you indicate the file make_musica be assembled files .mus
- the image file that you create with the SPEDIT
- sprites table where you assign images to the sprites (although not strictly necessary because you have the command | SETUPSP)
- the sequence table where you define that images form a sequence (although not strictly necessary because you have the command | SETUPSQ)
- the world map: where you define up to 64 elements of the world
- the file paths are paths where you define the sprites you want

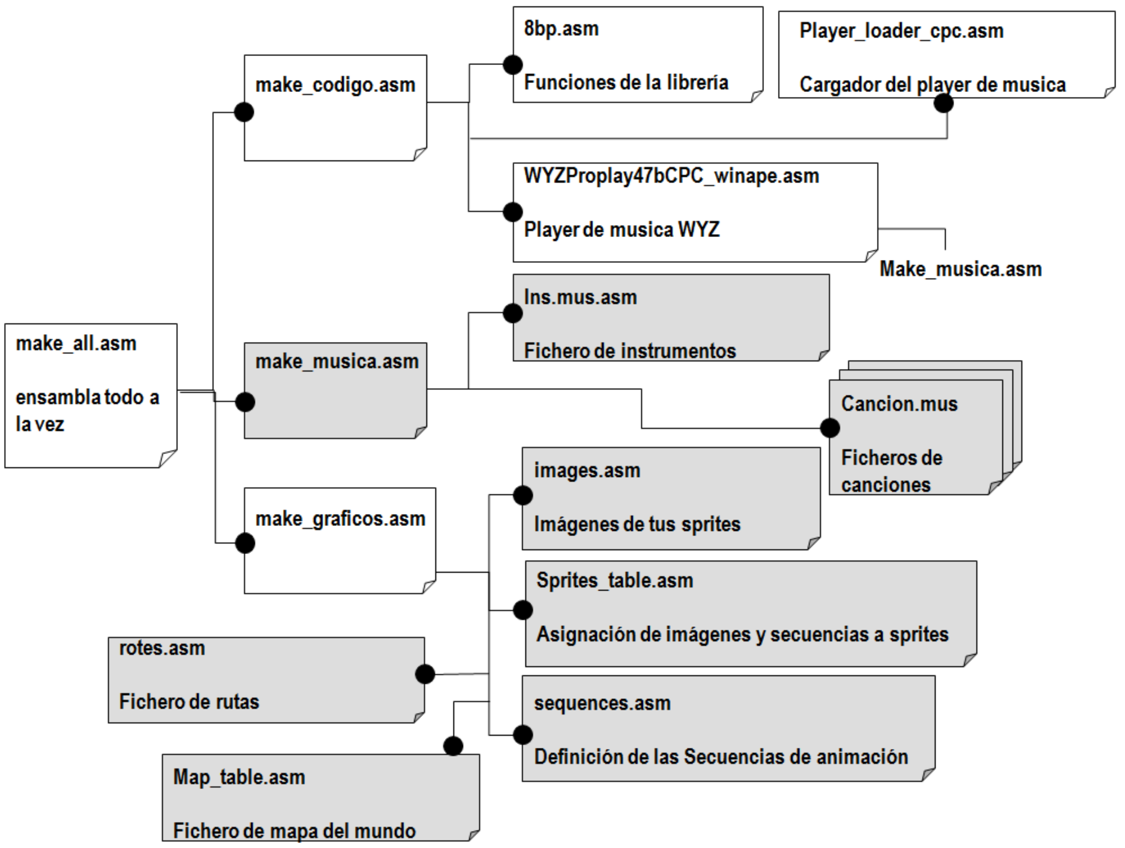


Fig. 54 Files to assemble

You can assemble all with make_all and then use the SAVE command to save the images, music and library 8BP in different binary files, as we have seen. To do this simply open the file make_all WinAPE within the editor and press "assemble"

If you just change the graphics you can assemble them separately, selecting the file "make_graficos.asm" and pressing assemble

If you change the musics you must re-assemble the code in the library because there is a dependency between the code and the songs, because the code needs to know where each song begins. So if you change or add songs you should join with make_all.asm and re-save your files "8BP.LIB" and "MUSIC.BIN"

14 Possible future improvements to the library

The 8BP library is improved, adding new features that could open up new possibilities for the programmer. Here are some suggestions to make

14.1 Memory to locate new features

Currently, the library occupies 6250 Bytes, of which 75 bytes are free for future expansion.

The space for music is currently 1250bytes and should not reduce it, so if in the future more memory apart from those 300 bytes is required, there are two options:

- Reduce the space available for the programmer (26KB).
- 8KB intended to reduce graphics

14.2 Save graphics memory

One possible operation:

| FLIP sequence, direction

It could turn all frames of an animation sequence, saving a lot of graphics memory. It would be an easy instruction to program and very profitable in savings. The programmer would force him to invoke when a sprite changes direction or to avoid any slowdown for that reason, it could make FLIP all types of enemies that were to appear on a screen just before entering it, reusing memory in other screens with other different enemies.

14.3 Printing pixel resolution

Currently used resolution and coordinates byte bytes that are 2 pixels in mode 0

Actually a way to overcome this limitation by defining 2 images for the same sprite that a single pixel are displaced. Moving that sprite across the screen you can switch between simply change the image for the displaced and move the sprite a byte. That way you get a movement pixel by pixel.

14.4 Layout mode 1

The current layout functions as a character buffer 20x25 = 500Bytes
It can be used in games mode 1 without problems but there will be things we can not do, how to define a piece that occupies 3 characters wide mode 1 because the character mode 0 occupy twice the width of mode 1. It is not a problem, but it is a limitation.

A layout mode 1 1KB occupy as 40x25 = 1000. Since the layout mode 0 and the mode 1 would not be used simultaneously, they would overlap in memory and taking into account that of mode 0 is in 42000-42500, simply to mode 1 it situaríamos between 41500 and 42500, "stealing "500bytes to the memory of 8KB sprites, between 34000 and 42000

Changes to support this improvement are minimal, affecting only two functions | LAYOUT and | colay should be aware of the mode screen, using a variable to act as a flag (layout0 / layout1) and manipulásemos from BASIC with POKE

14.5 Hardware scrolling functions

There are few games Amstrad CPC with a smooth scroll quality, programmed using the capabilities of the video controller chip M6845. Currently the library has a scroll mechanism based on CPU (not hardware but is efficient and versatile games with scroll in any direction of movement).

The fact that there are so many games due to the fact that in the 80s video game programmers did not have much information and also in many cases were fans
Among the few games that have smooth scrolling highlight 2 firebird

- Mission genocide (firebird, 1987, by Paul Shirley, an excellent programmer who also invented a technique of ultrafast overwriting without using masks)
- Warhawk (firebird, 1987)

Fig. 55 Firebird games with fast and smooth scrolling

Scroll technique of these two games is the same, known as "vertical shear". The breaking technique is to control exactly the moment when the sweep screen occurs. At that time we deceive the 6845 CTRC saying that the display ends earlier than normal. That if, before completing that section of the screen, we say that incorporates less scanlines of a section corresponding to that size. Then, in a moment we must control the microsecond, we tell the chip to start a new screen without vertical sync signal occurred. This allows us to draw a second display area (markers for example) and compensate the number of scanlines of the first section. If we do well the compensation mechanism number of scanlines, we can make one of two screen sections move with extraordinary smoothness. The problem of bringing this technique to a command to be used from

BASIC is that control of interruptions is imprecise due to the execution of the interpreter, and here we need a very precise control.

The problem of hardware scrolling (which also affects a scroll by software to move across the screen) is that "drag" the sprites present with him, so that to resell'll notice an unwanted vibration enemies and/or our character. To solve this you can use double buffering and switching between two blocks of 16KB each time a frame is ready. This will prevent you can see every frame "as is". In 8BP the doblebuffer I dismissed to leave a good space of RAM the programmer and why these techniques have not been implemented.

For all the above reasons, the scroll in 8BP is based on a world map that does not drag move sprites and therefore is more efficient because less memory moves while allowing multidirectional movement.

14.6 8BP migrate the library to other microcomputers

This library seía easily portable to other Z80 based microcomputers like the Sinclair ZX Spectrum. In the case of ZX spectrum would have to rewrite the routines painting on display because the video memory is handled differently.

The migration of the library to a Commodore 64 would also be feasible, although it could not reuse the assembly code, since it is based on another microprocessor. In addition, in the case of commodore 64, migration 8BP library should take advantage of the characteristics of the machine and its 8 sprites hardware, so it should internally incorporate 8BP library would be a sprite multiplexer, offering 32 sprites but internally 8 sprites using hardware.

Fig. 56 Sinclair ZX and Commodore 64, two classic

15 Some games made 8BP

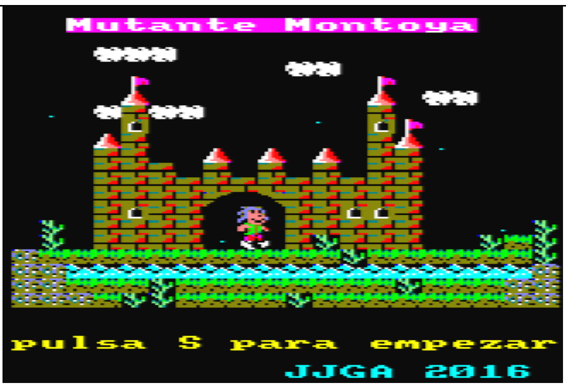
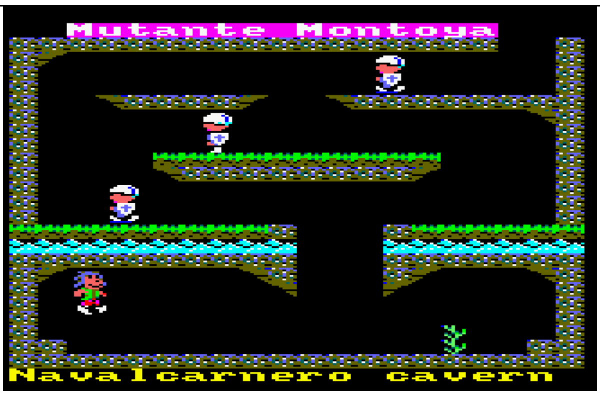
In this chapter I will describe as facts are some games you can find on the web https://github.com/jjaranda13/8BP made with 8BP

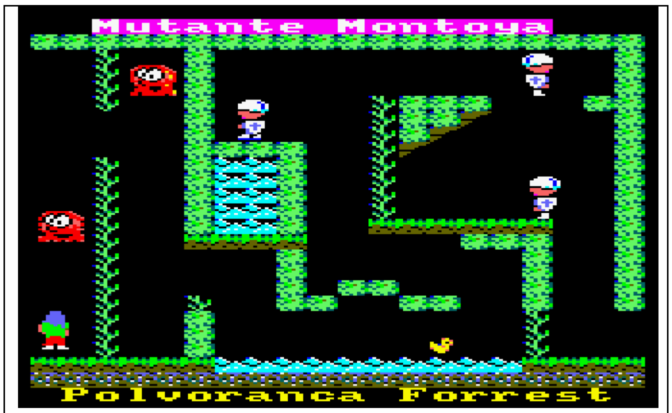
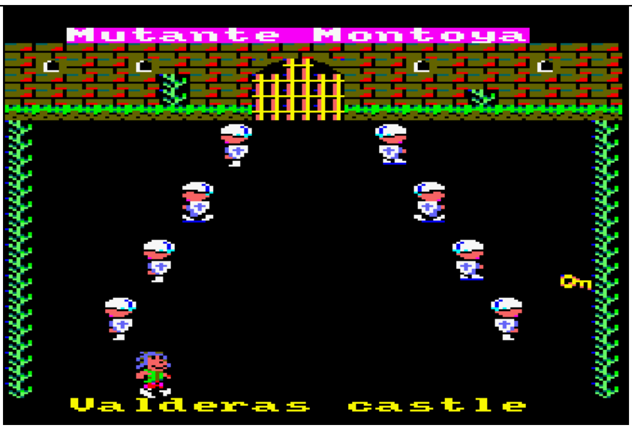
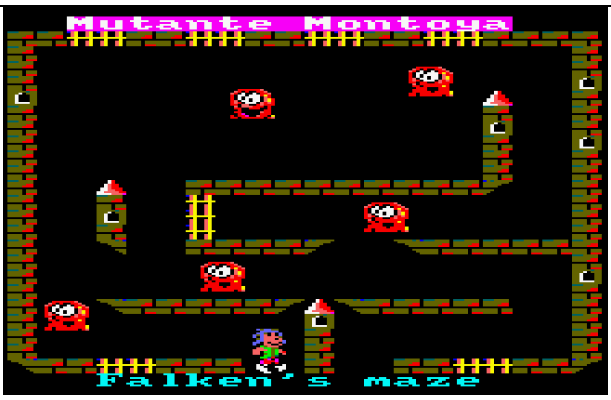
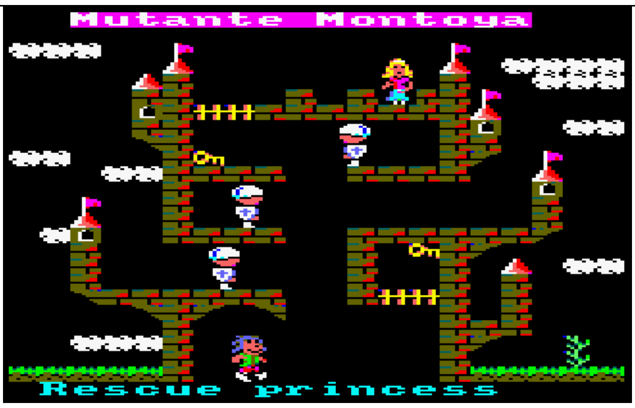
- **Anunnaki:** a game of ships, arcade genre
- **Mutant Montoya:** a game of pass screens could fit into gender platforms

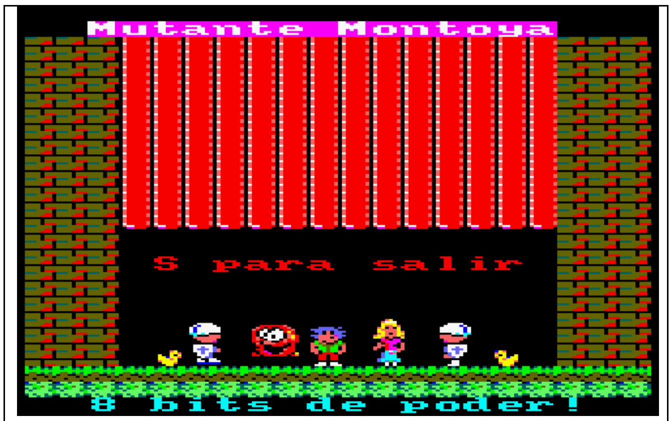
15.1 Montoya mutant

A first tribute to the Amstrad CPC, with a title inspired by the classic "mutant monty"

Available as part of the freeware distribution of library 8BP in dsk and wavhttps://github.com/jjaranda13/8BP
It is a simple game of 5 levels. It is based on the use of layout 8BP to build each screen. Here are some "clues" as is done, although the basic list is accessible

	The presentation simulates rain with the command | STARS. Try to see as they pass below sprites and layout elements without damaging them. The castle is built as a layout. In the list you can distinguish BASIC built with letters associated with sprites
	The first level is very simple. the output condition is that the Y coordinate of the character is less than zero (when you get out above). All sprites are printed simultaneously with the command 8BP "| PRINTSPALL"

	The second level uses no special technique. Do not use massive logical and yet you can see the power and speed of the library 8BP. there are 7 sprites You can also appreciate the "clipping" of the print routine if you move the character off screen to the left. In that case it will be shown only partially
	The third level uses the technique of "massive logic" to move 8 soldiers. Although there are 9 sprites on the screen at once, and be running in BASIC, everything works at an interesting pace adecuado.Es analyze how the gate opens. It acts on the layout when montoya take the key.
	The fourth level uses "massive logic". He could have gotten eight ghosts and would have continued to operate with little speed difference, it would have been very difficult to move the screen!
	This screen is not massive logic used even if there are 5 sprites, it is fast enough. the same technique as in the third level to open the floodgates used

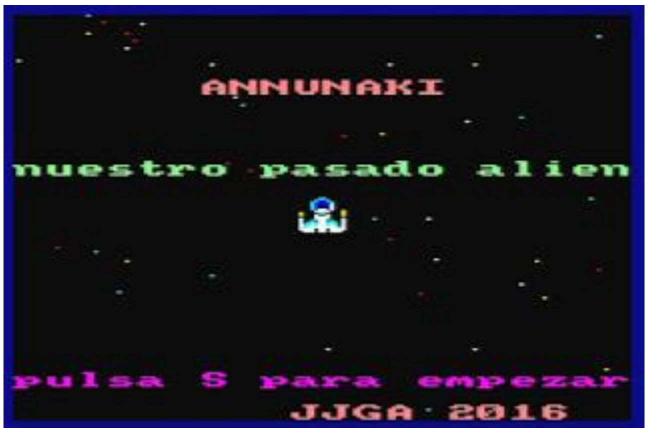
	At the end all the characters come out to say hello. The effect of the falling curtain is made with a single sprite, printing it in a loop and invoking the command | PRINTSP. It is very simple but interesting effect. The sprite is "smearing" while lowering the screen, giving the feeling of a curtain moving

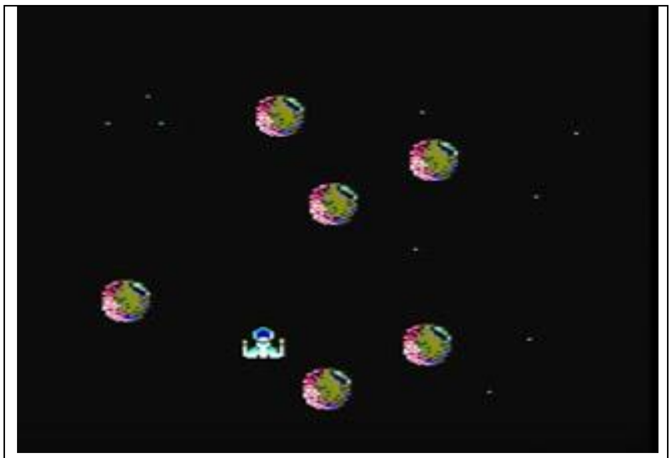
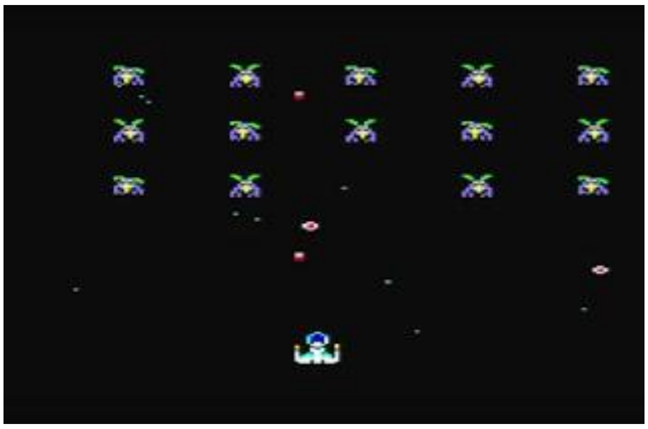
15.2 Anunnaki, our past alien

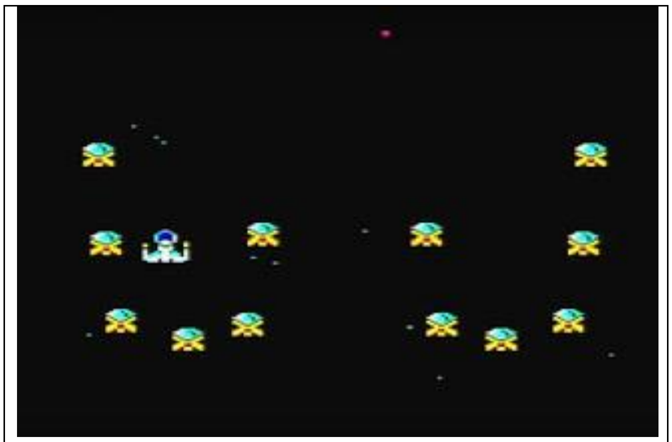
This is a very interesting to analyze and delve into the programming technique of "massive logic" game. The annunaki is a video game arcade showing intensive technique massive logic and simulation of vertical scroll

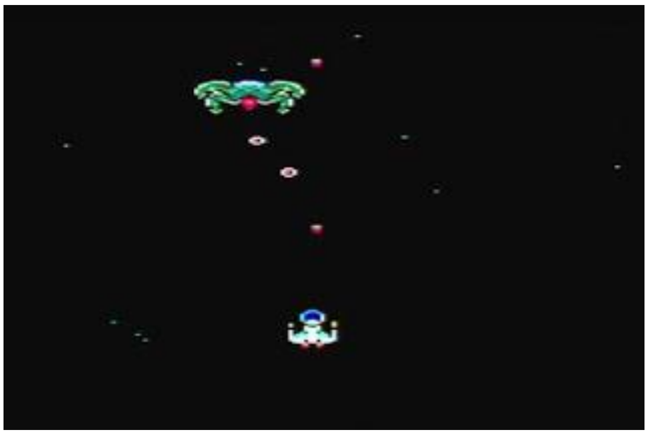
Unlike the "Mutant Montoya," the videogame "Anunnaki" does not use the layout as it is a game where it comes forward and destroy enemy ships, is not a game of mazes or go pantallas.Este game also makes use of techniques scroll "simulated" very interesting. It is available as part of the freeware distribution of 8BP library. Available in dsk and wavhttps://github.com/jjaranda13/8BP

You Enki, a commander anunnaki who faces alien races to conquer planet Earth and thus subjecting humans to their will.The game consists of 2 levels, but if you lose a life, continuous at the point of the level you find not back to the beginning of level.The first level is a phase in interstellar space, where you must dodge meteors and kill hordes of space ships and big birds. At the end of the level you must destroy a "boss" The second level is developed on the Moon, where you must destroy hordes of ships, after which you must cross a mine-infested until you find three that must destroy tunnel.

 ANNUNAKI nuestro pasado alien pulsa S para empezar JJGA 2016	the display screen shows a scroll of stars of 4 layers at different speeds to give a sense of depth. To do this use the command STARS the controls are QAOP and the spacebar to disparar.Durante the game, pressing Q, the ship goes up and shows its thrusters fire. This is done simply by changing the animation sequence associated with the ship with a POKE the corresponding to the sequence of animation of the ship in the direction sprites table
	meteorites are indestructible. By using the instruction COLSPALL collision it is determined that collided is a sprite

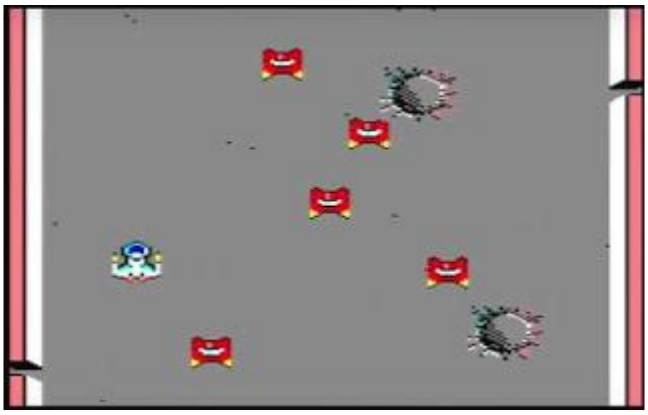
	whose identifier is greater than a certain number and thus concluded that it is a "tough" enemy that can not be destroyed. massive logic is used, so that in each frame only a meteorite may decide to change its direction of movement the collision of multiple shooting is done by COLSPALL, since the 3 shots have flag active in the status byte "collider" and to save cycles can only start dying an enemy in the same frame, which does not represent any limitation in the game, and we will kill one by one enemy.
	pajarracos all have relative motion flag in its status byte, and move with the MOVERALL command a trajectory stored in an array. Thus all they move at once with a single instruction there is a horde of ships that fall into "zig-zag" which also use the same technique, the displacements calculated with the cosine function is stored in an array for later use in MOVERALL | MOVERALL, 0, rx (i): i = (i + 1) MOD 15 as you can see, while there are background stars passing "below" all enemies, using the STARS command. explosions are "death sequences", so that after finishing the animation of the explosion, the enemy in question is disabled and does not print more.
	the different hordes of enemy ships in formation follow different paths, always in two rows of six ships each. To move the technique of controlling logical mass at each instant of time that two ships must change its direction of motion is used. The movement is done by assigning each ship one Vx, Vy and invoking AUTOALL pair, since they all have the flag of automatic movement





the final enemy (called "They arkaron") is semi-hard, it means you have a counter shots that reach and dies by receiving ten impacts.

music during these times change, one more "distressing" melody. 3 melodies that have the game take up little, no more than 200 bytes each



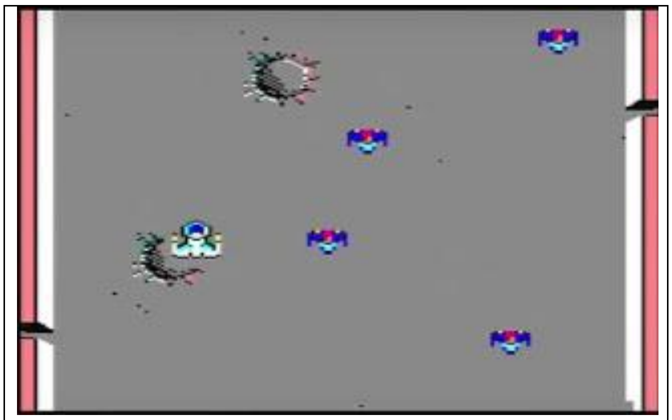
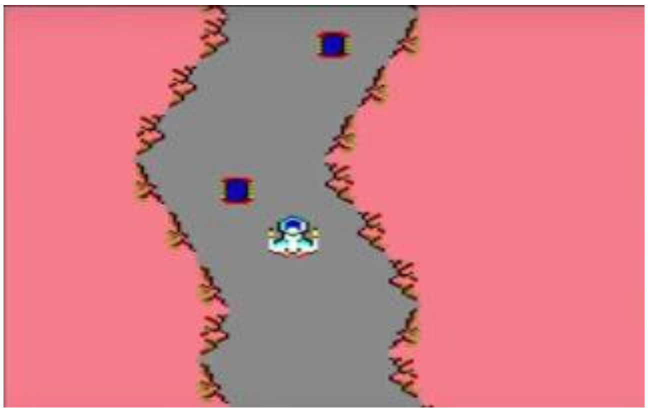
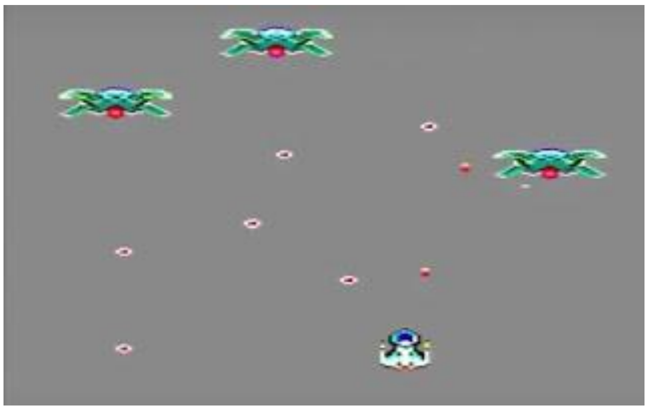
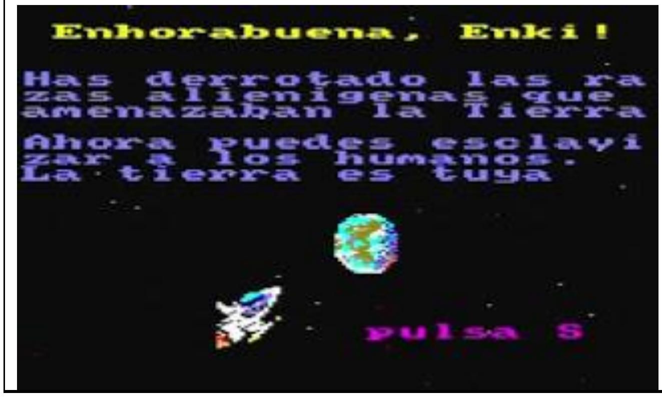
On the moon ("hollow moon" means "hollow moon" because it is an alien artificial satellite) scroll is simulated with "specks" and craters that move at the same speed. The speckles are made with STARS and craters are sprites move downward.

In addition, two sprites on the sides that "stain" the screen giving the effect of side walls are used.

all hordes using "massive logic" to move

more hordes, this time with a path that changes direction past few moments.

notice how the ship passes "above" the crater although not in use overwriting. It is because the ship has a handle above the crater and therefore sprite is printed later. Obviously "erase" a piece of

	crater but the effect is acceptable
	the tunnel is constructed with two rows of sprites that simulate the edges of the tunnel, giving an effect of vertical scroll It is really heavy because sprites are very large but the effect is acceptable overcome the tunnel is difficult, randomly generated so that sometimes it is more difficult and sometimes easier.
	the end must be destroyed three leaders whose difficulty is that no one dies independently but each must receive at least 10 shots for the 3 die at once. if you shoot a thousand times the same will get not overcome this challenge. The number of sprites on screen is very high if we add all the shots of the ship and enemies but the movements of the shots have little logic and everything moves fast enough
 Enhorabuena, Enki! Has derrotado las ra zas alienigenas que amenazaban la Tierra Ahora puedes esclavi zar a los humanos. La tierra es tuya pulsa S	By destroying the three leaders, we arrived at the end of the game screen, with a different melody and a message of congratulations. Enki is good? ... Or bad? Sumerian texts describe it as the good god and his brother Enlil was bad ... but what if it had been a hoax?

16 APPENDIX III: Organization of video memory

16.1 The human eye and the resolution of the CPC

The video memory Amstrad CPC has 3 modes of operation. The most used mode for games is the mode 0 (160x200) by having more color, but also has quite used the mode 1 (320x200) to schedule games.

Since the amount of video memory is the same resolution to win in many colors sacrifices, but curiously horizontally, that is the side of the longer screen, there is less resolution than vertical (160 horizontal and 200 in vertical). You wonder why. Also not the only microcomputer did that, many other computers also used the same strategy with the horizontal side

The reason has to do with the workings of the human visual system. The eye perceives more details vertical than horizontal, so that "harm" the resolution in the horizontal axis is not as serious as damage vertically. Subjectively is more acceptable result.

16.2 Video Memory

The most comprehensive and clear information is in the firmware amstrad manual. This information will be useful if you want to build an improved sprite editor or want to dive deeper into the assembler routines and schedule print overwriting or anything else.

16.2.1 Mode 2

In mode 2, each pixel is represented by one bit. So a byte represents 8 pixels. If we take any byte of video memory, correspondence with pixels is 1 bit per pixel, this table bits are represented as pixels (pi) correspond

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
p0	p1	p2	p3	p4	p5	p6	p7

a byte bit 7 as the most to the left is numbered. The pixel 0 is precisely also the pixel located more to the left, that is, there is nothing "backwards". Everything is correct

16.2.2 Mode 1

In mode 1 have 4 colors (represented by 2 bits). A byte therefore represents 4 pixels. The correspondence between pixels and bits is somewhat more complex. The pixel 0 for example is encoded with bits 7 and 4

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

16.2.3 Mode 0

Here we have a small mess. Each byte represents only two pixels, of which the bits corresponding with byte is: Pixel 0 is coded with the bits 7,5,3 and 1

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
p0(0)	p1(0)	p0(2)	p1(2)	p0(1)	p1(1)	p0(3)	p1(3)

In the next picture you certainly becomes clearer:

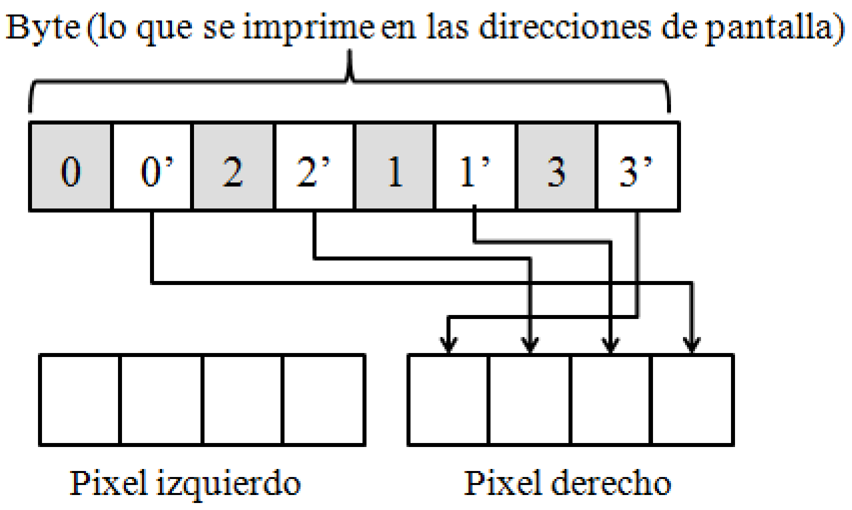


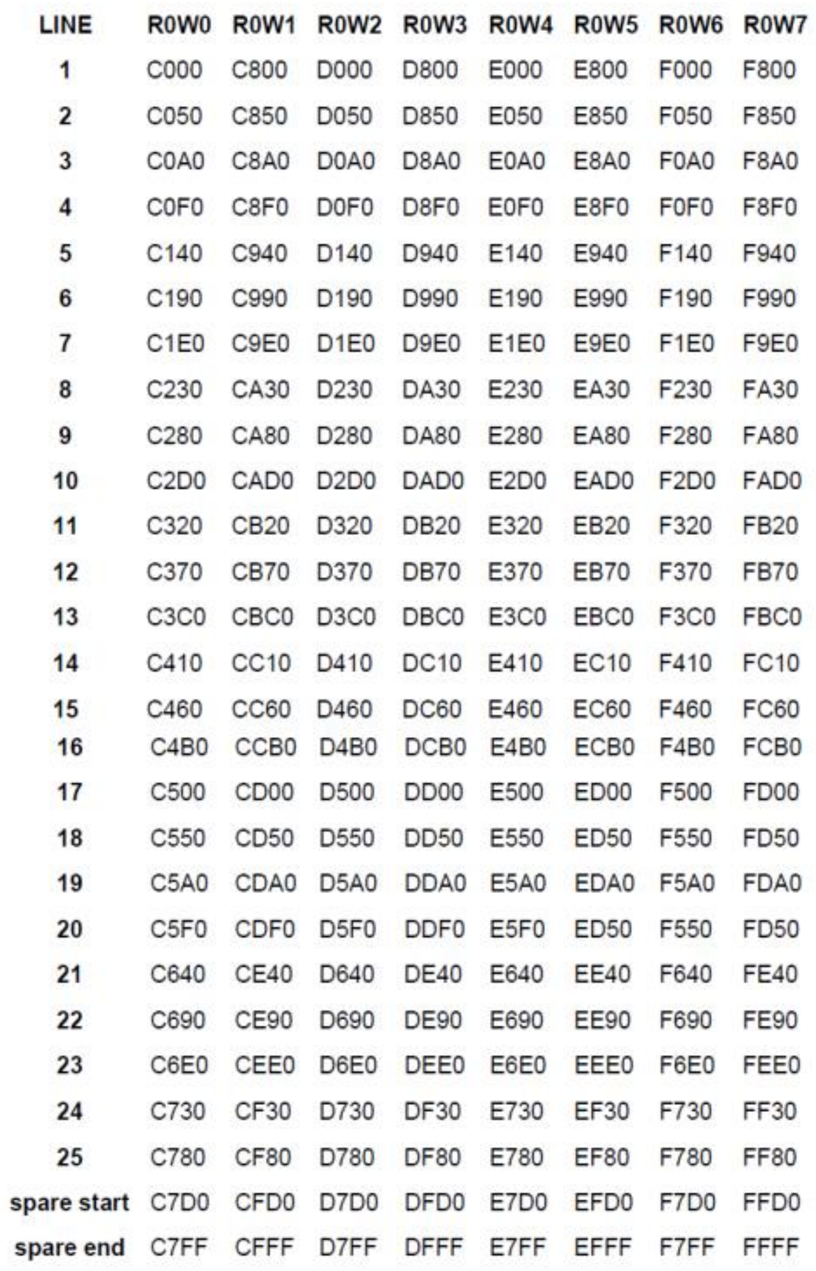
Fig. 57 pixels and bits in mode 0

I can not say what dark reason to be well organized memory but I imagine that the cause is in the GATE ARRAY, the chip that translates these bits to video signal. I imagine the designer reduced circuitry with this twisted design.

16.2.4 Screen memory

The pixels of the screen belonging to the same line are encoded in the bytes that are also contiguous. However, from one line to another no jumps.
If we move in memory addresses, when the end of a line jump to a line that is 8 lines below. And if we continue on the next line we have to jump in memory addresses are 2048 positions.

The following table displayed video memory. In the left line of characters (from 1 to 25) and for each line, the starting address of each of the eight scanlines that compose (expressed as ROW7 ROW0 ...) appears



LINE	R0W0	R0W1	R0W2	R0W3	R0W4	R0W5	R0W6	R0W7
1	C000	C800	D000	D800	E000	E800	F000	F800
2	C050	C850	D050	D850	E050	E850	F050	F850
3	C0A0	C8A0	D0A0	D8A0	E0A0	E8A0	F0A0	F8A0
4	C0F0	C8F0	D0F0	D8F0	E0F0	E8F0	F0F0	F8F0
5	C140	C940	D140	D940	E140	E940	F140	F940
6	C190	C990	D190	D990	E190	E990	F190	F990
7	C1E0	C9E0	D1E0	D9E0	E1E0	E9E0	F1E0	F9E0
8	C230	CA30	D230	DA30	E230	EA30	F230	FA30
9	C280	CA80	D280	DA80	E280	EA80	F280	FA80
10	C2D0	CAD0	D2D0	DAD0	E2D0	EAD0	F2D0	FAD0
11	C320	CB20	D320	DB20	E320	EB20	F320	FB20
12	C370	CB70	D370	DB70	E370	EB70	F370	FB70
13	C3C0	CBC0	D3C0	DBC0	E3C0	EBC0	F3C0	FBC0
14	C410	CC10	D410	DC10	E410	EC10	F410	FC10
15	C460	CC60	D460	DC60	E460	EC60	F460	FC60
16	C4B0	CCB0	D4B0	DCB0	E4B0	ECB0	F4B0	FCB0
17	C500	CD00	D500	DD00	E500	ED00	F500	FD00
18	C550	CD50	D550	DD50	E550	ED50	F550	FD50
19	C5A0	CDA0	D5A0	DDA0	E5A0	EDA0	F5A0	FDA0
20	C5F0	CDF0	D5F0	DDF0	E5F0	ED50	F550	FD50
21	C640	CE40	D640	DE40	E640	EE40	F640	FE40
22	C690	CE90	D690	DE90	E690	EE90	F690	FE90
23	C6E0	CEE0	D6E0	DEE0	E6E0	EEE0	F6E0	FEE0
24	C730	CF30	D730	DF30	E730	EF30	F730	FF30
25	C780	CF80	D780	DF80	E780	EF80	F780	FF80
spare start	C7D0	CFD0	D7D0	DFD0	E7D0	EFD0	F7D0	FFD0
spare end	C7FF	CFFF	D7FF	DFFF	E7FF	EFFF	F7FF	FFFF

Fig. 58 Memory Map screen

Amstrad screen has 200 lines x 80 bytes wide each, so the memory that is displayed is 200 x80 = 16,000 bytes. However the video memory is 16384 bytes. There are 384 bytes "hidden" in 8 segments of 48 bytes each, which are not shown on screen even if part of the video memory. These 8 segments are what in the table above are called "spare". Each segment is 48 bytes because as I said before to jump from one line to the next line must be added 2048 but in reality the 25 lines that separate contiguous memory only occupy 25x80bytes = 2000 bytes.

```
From the & C7D0 to C7FF inclusive
From the & CFD0 to CFFF inclusive
From the & D7D0 to D7FF inclusive
From the & DFD0 to DFFF inclusive
From the & E7D0 to E7FF inclusive
From the & EFD0 to EFFF inclusive
From the & F7D0 to F7FF inclusive
From the & FFD0 to FFFF inclusive
```

You can check this by POKE on these memory addresses, and see that will not alter the content of the screen.
It is tempting to think of using these "hidden" areas of memory to store small assembly routines or variables. However it is dangerous because a MODE command issued from BASIC completely erase those memory segments, so if you use it you must be aware of it. LIBERIA 8BP in these segments are used to store local variables of some functions whose value can be deleted without risk.

16.3 Sweeps screen

Amstrad generates 50 frames per second. That means that every 20ms should produce about a new sweep screen.

We might think that perhaps the sweep, which is to paint the screen consumes a fraction of the 20 ms but it is not. Paint the screen takes you to complete Amstrad 20ms, so that even sync your impression of sprites with sweeping display is very likely that you get caught, leading to two known effects:

- Flickering: (flicker) occurs when you delete the sprite before printing in your new position. To avoid this there is a very simple solution: do not delete it. Just do the sprite go erasing their own trail, leaving a border on the sprite to fulfill that function. The sprite is bigger but not blink, though he caught sweeping half, then do not go away

- Tearing: (jog): occurs when we caught sweeping half the sprite. Half is printed with the new position (head and trunk) and the other half is not enough time (legs). Then the sprite is printed "evil", although it is corrected in the next frame but for a moment as if it deform or went bankrupt. THE tearing a bad effect but much more acceptable than the flickering. The perfect solution involves controlling every millisecond where the sweep for each sprite print without our reach.

A typical recommendation is to print the sprites from the bottom up, to minimize these effects. Thus it is only possible to sweep once you get caught in one of the sprites, while printing from top to bottom, you can catch in several sprites sweep that both (CPU and Cathode Ray) work in the same direction. Unfortunately most interesting it is sort of up and down to give effect deep in the sprites (useful in certain games like Golden Axe, double dragon, renegade, etc)

Time consuming screen are as follows. Note that since the scanning interruption occurs, you have to paint 3,5ms possible without getting caught. But at that time you can print as much 2 small sprites.

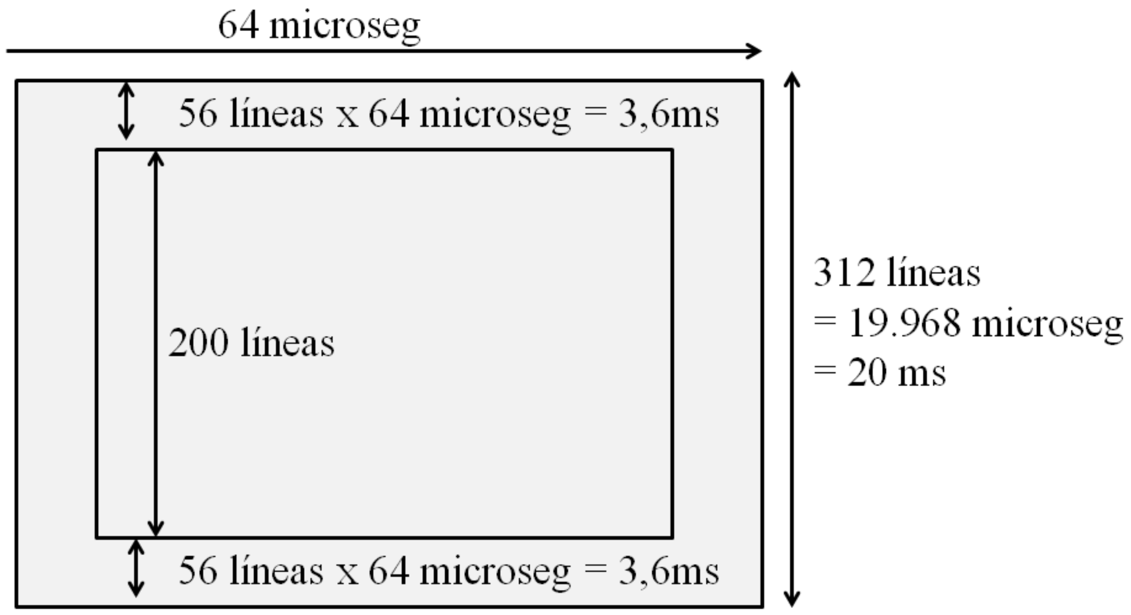

64 microseg
56 líneas x 64 microseg = 3,6ms
200 líneas
312 líneas
= 19.968 microseg
= 20 ms
56 líneas x 64 microseg = 3,6ms

Fig. 59 times in a screen sweep

16.4 Making a loading screen for your game

There are many ways to do it. A very simple is to construct a graph modified to allow you to paint across the screen without menus show and at the end press a key to launch a SAVE command you spedit program like this

SAVE "mipantalla.bin", b, & C000, 16384

As you see the command saves 16KB from the start address of the display, which is the & C000
How to load it would be

LOAD "mipantalla.bin", & C000

And you'd see slowly as while charging is being drawn on the screen, since it is precisely there where these charging into the video memory.

Another way is to create a well-worked layout and save it using the command above SAVE

Finally you can use a tool like ConvImgCPC (there are others), an image converter that works under Windows. This tool allows you to transform any image (which can be a scan of one of your drawing) into a binary file (with extension .scr) suitable for CPC

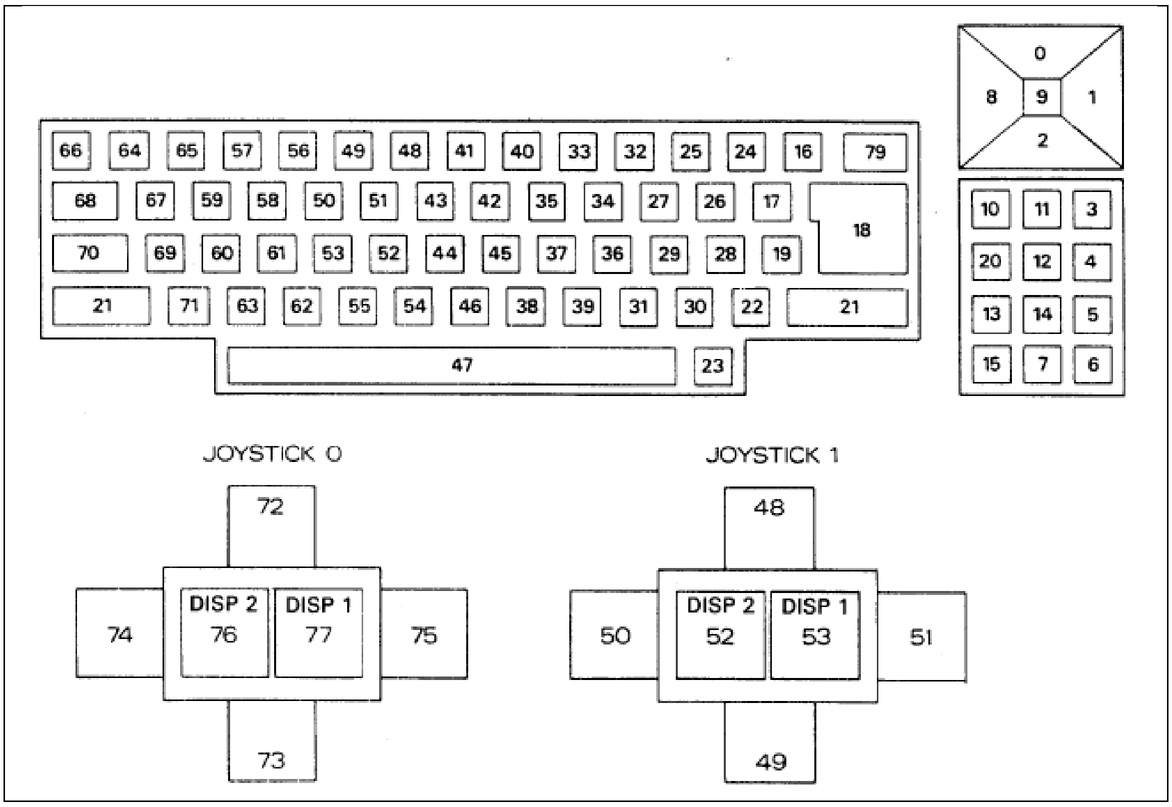
To put this file on a disk (in a .dsk file) you must use the CPCDiskXP which is another tool that allows you to put files into files .dsk

Once inside the .dsk you can load with LOAD "mipantalla.bin", & C000

However the colors may not look right because ConvImgCPC adjusts the palette to close as possible to the original colors. If you invoke CALL & C7D0 get to see the image with the colors selected by convImgCPC

To leave the palette in its default, use CALL & BC02, a firmware routine.

17 APPENDIX IV: INKEY codes



18 APPENDIX V: Palette

The 27 colors are:

0 - Negro (5)	1 - Azul (0,14)	2 - Azul claro (6)	3 - Rojo	4 - Magenta	5 - Violeta	6 - Rojo claro (3)	7 - Púrpura	8 - Magenta claro (7)
9 - Verde	10 - Cyan (8)	11 - Azul cielo (15)	12 - Amarillo (9)	13 - Gris	14 - Azul pálido (10)	15 - Anaranjado	16 - Rosa (11)	17 - Magenta pálido
18 - Verde claro (12)	19 - Verde mar	20 - Cyan claro (2)	21 - Verde lima	22 - Verde pálido (13)	23 - Cyan pálido	24 - Amarillo claro (1)	25 - Amarillo pálido	26 - Blanco (4)

The values of the default palette in each mode are:

Modo 2:

0: Azul (paleta 1)	1: Amarillo intenso (paleta 24)

Modo 1:

0: Azul (paleta 1)	1: Amarillo intenso (paleta 24)
2: Cyan claro (paleta 20)	3: Rojo claro (paleta 6)

Modo 0:

0: Azul (paleta 1)	1: Amarillo intenso (paleta 24)	2: Cyan claro (paleta 20)	3: Rojo claro (paleta 6)
4: Blanco (paleta 26)	5: Negro (paleta 0)	6: Azul claro (paleta 2)	7: Magenta claro (paleta 8)
8: Cyan (paleta 10)	9: Amarillo (paleta 12)	10: Azul pálido (paleta 14)	11: Rosa (paleta 16)
12: Verde claro (paleta 18)	13: Verde pálido (paleta 22)	14: Parpadeo Azul/Amarillo	15: Parpadeo azul cielo/Rosa

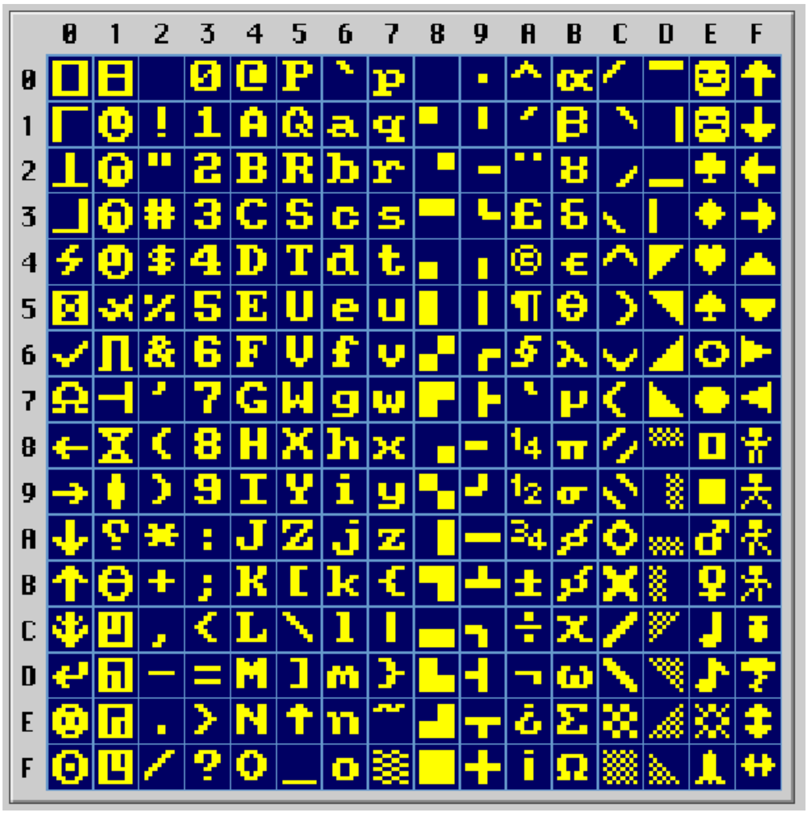
The palette values in each mode are managed with the INK command, see the reference manual Amstrad BASIC for more information.

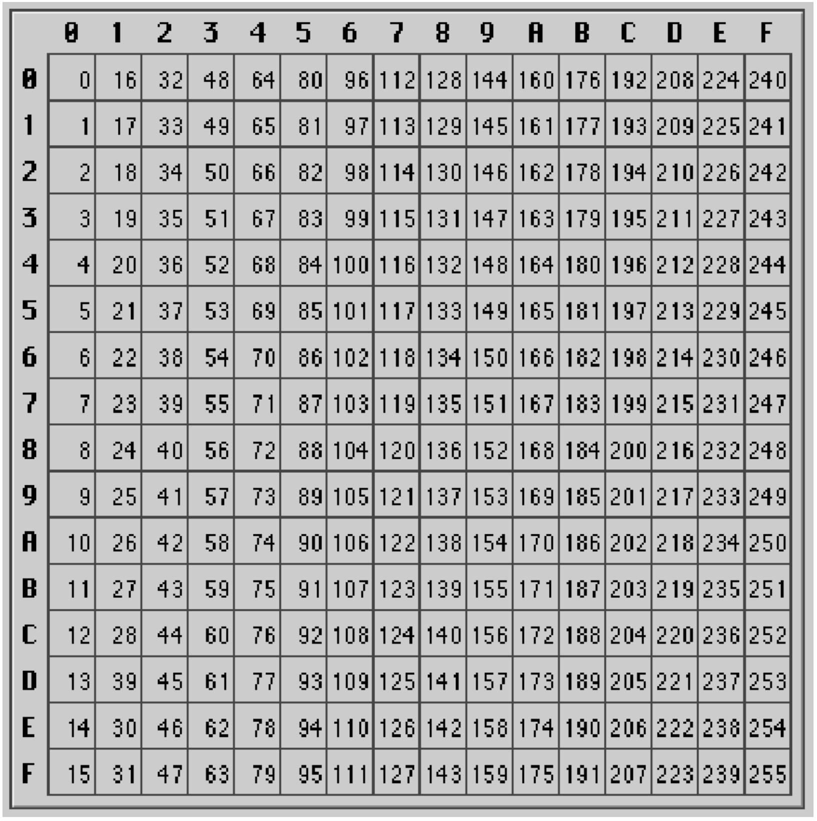
For example, to set the zero color like red, consult the palette of the 27, we see that red is the sixth color and write

INK 0.6

And we have set the zero color to be red

19 APPENDIX VI: ASCII Table Amstrad CPC





	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240
1	1	17	33	49	65	81	97	113	129	145	161	177	193	209	225	241
2	2	18	34	50	66	82	98	114	130	146	162	178	194	210	226	242
3	3	19	35	51	67	83	99	115	131	147	163	179	195	211	227	243
4	4	20	36	52	68	84	100	116	132	148	164	180	196	212	228	244
5	5	21	37	53	69	85	101	117	133	149	165	181	197	213	229	245
6	6	22	38	54	70	86	102	118	134	150	166	182	198	214	230	246
7	7	23	39	55	71	87	103	119	135	151	167	183	199	215	231	247
8	8	24	40	56	72	88	104	120	136	152	168	184	200	216	232	248
9	9	25	41	57	73	89	105	121	137	153	169	185	201	217	233	249
A	10	26	42	58	74	90	106	122	138	154	170	186	202	218	234	250
B	11	27	43	59	75	91	107	123	139	155	171	187	203	219	235	251
C	12	28	44	60	76	92	108	124	140	156	172	188	204	220	236	252
D	13	39	45	61	77	93	109	125	141	157	173	189	205	221	237	253
E	14	30	46	62	78	94	110	126	142	158	174	190	206	222	238	254
F	15	31	47	63	79	95	111	127	143	159	175	191	207	223	239	255

20 APPENDIX VII: Some interesting firmware routines

In this section I will include some firmware routines that can be called from BASIC and that might be interesting in your programs

CALL 0: produces a reset computer

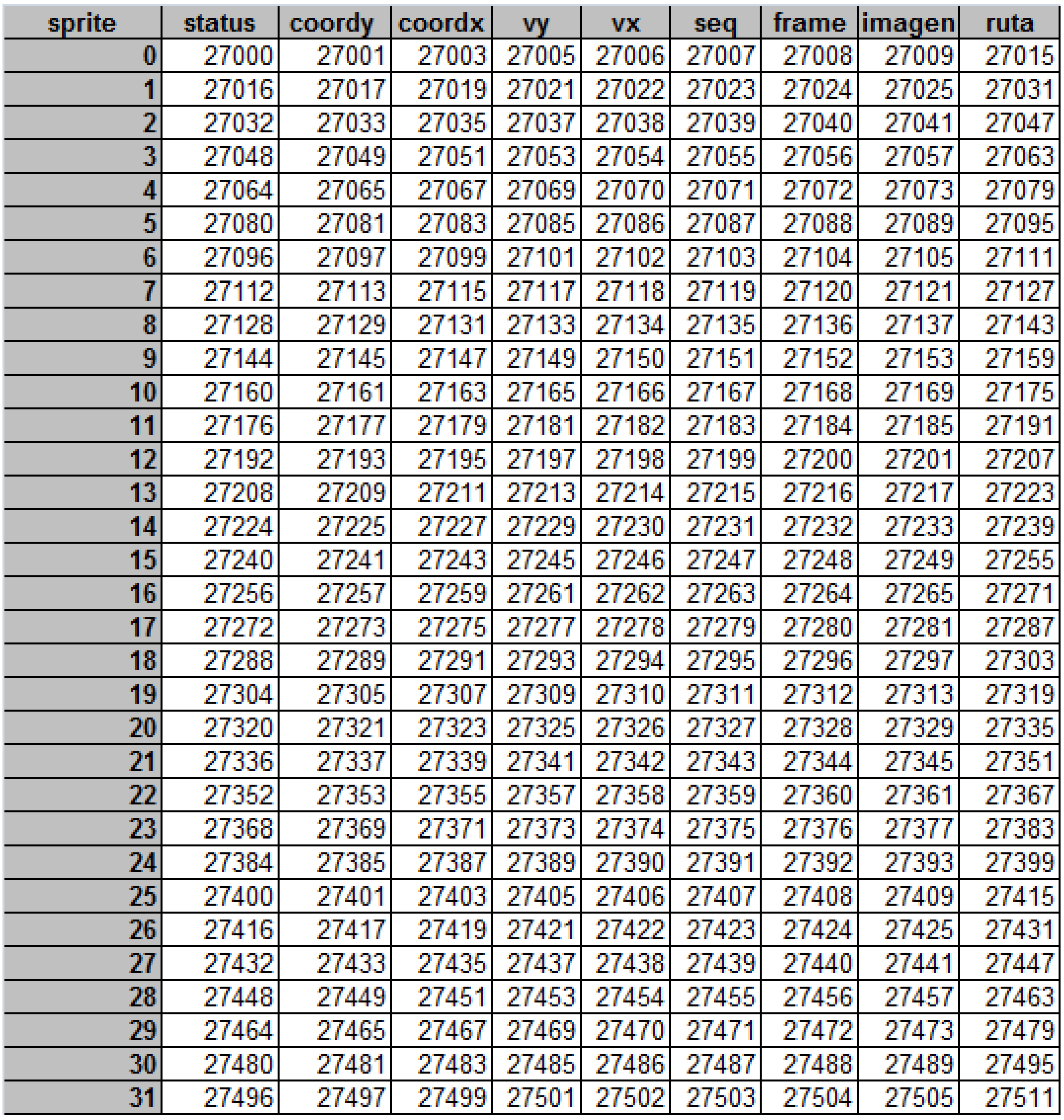
CALL & BC02: initializes the palette to its default value. It is good practice to invoke at the beginning of your program if you would find altered

CALL & BD19: synchronized with the scanning screen. If you drive a few sprites can get a smoother motion but keep in mind that this instruction will slow down much your program.

CALL & BB48: BREAK disables the mechanism, preventing stop the program if it is running.

CALL & BD21, BD22 &, & BD23, BD24 &, & BD25: produces an effect of flash on the screen

21 APPENDIX VIII: Table of attributes of Sprites



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

22 APENDIX IX: 8BP memory map

```
             AMSTRAD CPC464 MEMORY MAP OF 8BP
;
; AMSTRAD CPC464 memory map of 8BP
;
; & FFFF + ----------
;        | screen(16000 Bytes) + 8 hidden screen segments each 48bytes
; & C000 + ----------
;        | system (definible symbols, etc.)
;  42619 + ----------
;        | bank of 40 stars (from 42540 to 42619 = 80bytes)
;  42540 + ----------
;        | map layout of characters (25x20 = 500 bytes)
;  42040 + ----------
;        | sprites (up to drawings 8.5KB)
;        |
;        + ----------
;        | Route definitions (variable length each)
;        + ---------
;        | animation sequences of 8 frames (16 bytes each)
;        | and groups of animation sequences (macrosequences)
;  33500 + ----------
;        | songs (1.25kB to music)
;        |
;  32250 + ----------
;        | 8BP routines (5860 bytes)
;        | Here are all the routines and table sprites
;        | It includes music player "wyz"
;  26390 + ----------
;        | world map (389 bytes)
;  26000 + ----------
;        | | BASIC variables
;        | V
;        |
;        | ^ BASIC (program text)
;        | |
;      0 + ----------
```

23 APENDICE X: Available commands in 8BP library

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

Tabla 7 Available commands in 8BP library