

Making of

**WRECKING BALL**

by Francesc Alcauer

## Content

Introduction .....	3
Argument .....	3
Graphics.....	4
Text hack in mode 0 .....	4
UDG editor for text <i>hack</i> .....	6
Evolution of the game's graphics .....	7
First graphics test on the CPC 464 model (it doesn't go well) .....	8
Seeking for professional help .....	9
Pixel screen scrolling vertical .....	11
Title screen .....	12
Loading and presentation screen.....	13
Moving elements during the game .....	13
Speed and gameplay .....	14
Moving things.....	14
Main loop .....	14
Bouncing pattern.....	15
Movement and action of the palette .....	16
Movement of rescue capsules .....	17
Collection of effect capsules .....	17
Sound and music .....	17
Sound effects.....	17
"Positional" stereophonic effect .....	18
Music .....	18
Loading system from tape.....	22
Fast loading of <i>extra-sized blocks</i> .....	22
Wink .....	23
Description of the parts of the program .....	23
Resources and tools used.....	24
Farewell.....	24

## Introduction

***Wrecking Ball* is a breakout** arcade video game developed in Locomotive **Basic** for **Amstrad CPC** computers. This game uses a graphics technique (which I thought was common in games in machine code), which manages through **UDG and some poke to firmware control addresses to define colorful graphics without using the effect of text transparency.**

Since the Basic in the *Amstrad CPC* is an interpreted programming language, in the code I have not added comments with the intention of not unnecessarily slowing down its execution, but here we will try to separate the main parts and give some explanations of their operation.

The conception of *Wrecking Ball* took place in the final stages of my previous development, also in Locomotive **Basic**, **CPC Invaders**, while thinking about the best way that the background could be repainted during the action of the game. That's when I thought of an *Arkanoid* style game and if it would not be possible to preserve the background of the screen by painting a char stored in a text-type array, choosing the char N from the X coordinate of the ball and being the element N' of the array the Y coordinate, running at a good speed.

The first thing I did was program a play area with simple obstacles and a bouncing ball (the 231st char of the *Amstrad CPC*). With the 4 colors of mode 1 of the CPC I did not think it was going to be very showy, dedicating one of those colors to the background (although I planned to use a repetitive pattern with plotting in the style of *Arkanoid*), there were only 2 more colors left for the rest of the action, I needed more color, which I would get with mode 0.

As I said above, **itis assumed** that to get multicolored UDG graphics you must use the text transparency mode and different chars with defined points according to the different colors that you want to show, but **with this project I have discovered that this is not so.**

Recalling the tests I had done 30 years ago with mode 0, in which **I poked the CPC records so that the firmware 'believed' that it was working in mode 1** (this had been used in some simple game, so that the markers did not look so huge with that horrible snout text font of mode 0, in the style of the scoreboard of the game *Rampage* of *Activision* in CPC), a surprising side effect is achieved: **the text are displayed in several colors;** instead of just the color of the foreground and background, others appear... Would it be possible to define which colors should appear or was it just an uncontrollable *bug*?

Another of the objectives that I set out to address with this project is to take special care in the sound and music, providing it with positional sound (ahem, more or less) and a more elaborate melody than in my previous project, in which only repetitive tones and sounds were emitted as drums at fixed times.

## Argument

We are in the year 3021 and the space debris around the earth has become so numerous that huge blocks of circulating material are forming in the immediate vicinity. These blocks pose not only a risk of collision with space stations, spacecrafts, or satellites around orbit, but could descend in altitude and reach the surface of the planet, with fatal results. You have been appointed to be the first operator of the space wrecking ball, a huge (and dangerous) spherical mass resulting from a secret project with a great capacity for disintegration...

## Graphics

### Text hack in mode 0

I will start by clarifying that **the development was begun emulating the model 6128 of the Amstrad CPC**. As a reference resource when I had the CPC, I bought (and still have) the book published in 1987 by Anaya **Multimedia, Amstrad CPC 464/664/6128. Manual de referencia avanzado**, by Rafael Sarmiento de **Sotomayor**, in this, among other topics, the routines and directions of the firmware of the CPCs are discussed. In the middle of page 153, the control memory addresses of the block referring to the screen appear.

Doing this: `MODE 0:POKE &B7C3,1` (this direction controls in the 6128 the active screen mode), the text cursor goes from rectangular to square, as in mode 1, and then doing `PEN 2`, we get an interesting and colorful *Ready* prompt:



If we test with different values for *PEN*, we observe that the behavior is like that of mode 1, showing text with the values from 1 to 3 and with value 4 equals *PEN 0* and the sequence is repeated in higher values. Every time we change to a value in which text is displayed, if we refine the view, we can see that it does so by showing points in 4 different inks (counting the background color):



Well, the ink value of the background is always the same, but if the others vary and 3 values are valid, we have 9 inks, what about the rest? Let's do the following test:

```
10 MODE 0:PEN #4,6:PEN #5,13:PEN #6,11:PEN #7,14
20 POKE &B7C3,1:PEN #1,1:PEN #2,2:PEN #3,3
30 FOR n=1 to 7:LOCATE #n,1,10+n:PRINT #n,"Ready":NEXT
```

With this we establish inks for *PEN* before and after *poke*. The result is as follows:



We see that two things have happened: while typing the cursor appears in the next line when it goes beyond the horizontal position 20 and we have texts with different combinations of inks. To solve the 20 columns what I do is redefine all the windows with `WINDOW #n,1,40,1,25` and

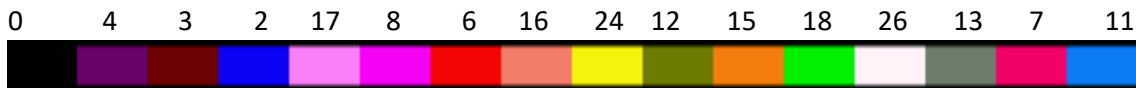
with that I already have the 40 columns. For the graphics window something similar happens, but it is solved with the instruction: *ORIGIN 0,0,0,640,400,0*

After different tests, I get the following table:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	X			X	X										
2		X						X		X					
3			X									X			X
4				X											
5	X			X	X										
6		X		X		X									
7			X	X			X								
8								X							
9	X							X	X						
10		X						X		X					
11			X					X			X				
12												X			
13	X											X	X		
14		X										X		X	
15			X									X			X

On the vertical axis we have the possible values for *PEN* and on the horizontal axis we have with which inks the chars that will appear with dada one will be shown. As we can see, there are two small problems and that is that there are inks that are repeated with different *PEN* values and that the values 4, 8 and 12 only show 1 ink.

I decide then to take the following strategy: load the inks so that they can represent chars with colors with some gradation, there are the colors in inks from 0 to 15:



Moving the color to the table above, we have the following multicolored combinations:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	X			X	X										
2		X						X		X					
3			X									X			X
4				X											
5	X			X	X										
W6		X		X		X									
7			X	X			X								
8								X							
9	X							X	X						
10		X						X		X					
11			X					X			X				
12												X			
13	X											X	X		
14		X										X		X	
15			X									X			X

UDG editor for text *hack*

To facilitate the task of investigating how the text is presented on the screen when I use this small *hack* and to create the first *graphics*, I make a simple UDG editor, being as follows:



In the upper-left part we have the char matrix of 8x8 points, just to its right we have the same char when we have the text *hack*, showing a char of 4x8 points.

If we look closely, we can see that each row of the char we define is divided into two *nibbles* that represent the two pixels on the left and the two on the right of the char, getting one ink or another, we will say that (with the color palette we have applied) of low, medium, or high intensity, depending on which bits we activate in each *nibble*.

With a little more effort and evolving the tool, I get the following result and now it allows me to export to file with BAS extension, formatted ASCII text with all *SYMBOL commands*:

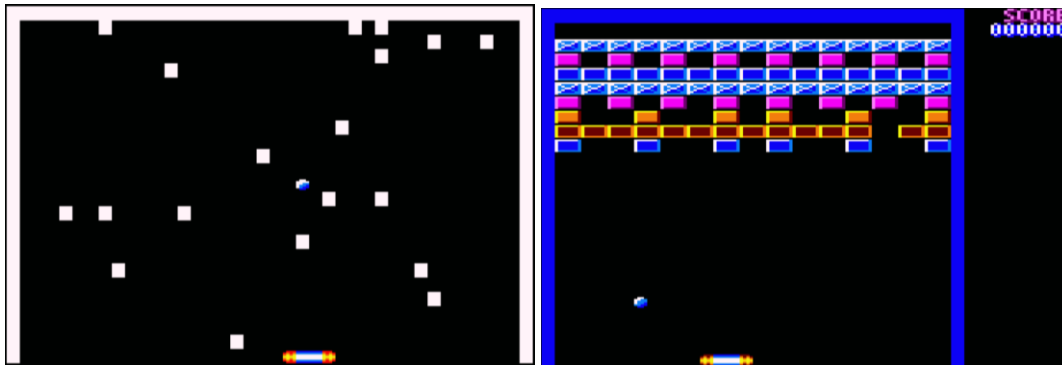


In conclusion, what I have is a UDG editor that allows me to generate graphics in 4 colors (3 foreground in addition to the background) thanks to the text *hack* of mode 0, to show '*sprites*' printing on the screen a single char with 4 colors, instead of having to print 3 chars with the text transparency mode. According to my tests, printing a single char is 1.9 times faster. I haven't tried it, but I imagine that with longer text strings there will be more difference.

Keep in mind that as it is logical to think, this trick can also be used from programs written in other languages, if the firmware is not deactivated.

### Evolution of the game's graphics

Here is the first and second model 😊:



I make a new UDG design of the ball, it goes from v1 to v3, which I like more, and I leave definitively since it seems to me that it has a metallic appearance:



I also redesign the edge of the play area and the *bricks* that will have to be destroyed with different characteristics of hardness, I add counter of those that remain to be eliminated and I give life to the point marker. At the moment the ball is never lost, so lives do not descend:



The thing is going well, I incorporate the pause, more elements of the marker and transparent bricks (the text *hack* produces a curious effect of transparency that does not look bad), after which the background is shown, background that consists of a variable type *text DIMensioned* of 40 chars in length for each element and that is redrawn with the movement of the ball or when destroying bricks, obtaining the char with *MID\$* of the dimensioned variable, without the speed suffering.



First graphics test on the CPC 464 model (it doesn't go well)

I am about to do the first test on the *model 464*. I generate the CDT and charge it, change the direction of the *POKE* address, from B7C3 to B1C8, which corresponds to 464 model, and I went into shock when I see this appear:

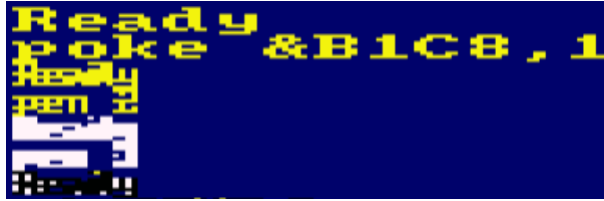


The horror. The firmware of the 464 model is different, apart from the basic instruction set. I had already noticed certain differences for example when printing chars in the lower-right corner, that in the 464 there is a carriage return and in the 664/6128 it does not happen like that and things like that, but this already ... anyway, why does he do this? Frustration and despair in equal parts...



## Seeking for professional help

A disaster, without the text *hack* there is no game, the visual impact it supposed was decisive and in the 464 model it did not seem to work... in this model it is different, it does strange things, there is no multicolored text, the pixels of the letters seem displaced:



Then I retrieve my old notebook, a folder of rings that is over 30 or 35 years old and that I still have. In that notebook I keep the design of the letters of the *Arkanoid II* that I played in *CPC Invaders* game and by which I had left my eyes copying pixel by pixel from the screen to the paper, it also contains some game passwords and formula annotations, small listings in Basic, tricks with the *OUT* and *POKE* instruction, among which the following is noted:

*Mode 0/1 special text design STR*

*10 mode 0*

*20 poke &B1C8,1:poke &B1D0,0:poke &B1D2,60*

The CPC model I had when I was a kid was the tape model, and all the tests I did were for that model. At that time, I had already been playing with this, but I had not exploited it, I had not done enough research on what was happening with the text and the different inks because I did not know how to realize anything. What I did do is play with those other directions (*B1D0* and *B1D2*) because it seemed like the text looked better:



I didn't remember it, but the text could also be seen in colors, and I liked how it looked, but there was the thing:



Back in the firmware manual, what did those memory addresses control? According to the book, the point masks and it is true that the points of the chars seemed displaced before touching the masks and after doing so the multicolored text was also more colorful, so they had something to do with all this.

The mask control memory addresses are those ranging from *B1CF* to *B1D6*, but apparently you only needed the first 4. I was testing different values and I could not make it look the same as in the *6128*, a model in which the masks also had no effect and in fact their values are at zero, so it did not serve as a reference. It seemed that in the *464* the values that I could assign to *PEN* behaved as if it were in mode 1, that is, from 0 to 3 and in 4 the sequence was repeated, so that the game would show a color lower than that of the *6128* model (remember the color table on page 4). Again, a situation of frustration...

I search the Internet and it does not seem that anyone has done anything similar with the firmware of the *CPC*, I am in a dead end, until I decide to contact "el profe", Fran Gallego... "If he doesn't help me with this, no one else can" I told myself.

I get in touch through the *CPCtelera* group on *Telegram*, I explain my problem to the group and the teacher immediately shows his interest in the matter, I jumped out of my chair and passed by private a photo of the screen where the project is seen in the two models, 6128 (left) and 464 (right):



*Profe* is asking me things and goes on to do his own tests and see the effects, and even reviews the CM routines of the firmware that are responsible for printing the chars on the screen to see how they work... As he continues explaining I's trying different things, then... surprise! we found that the default mask when mode 1 is on (&88, &44, &22, &11) is the one I had to use in MODE 0, but to set the ink to use I must not use the *PEN* statement, but a **POKE &B28F,ink**

To set the inks relative to another window (*WINDOW #n*), what I must do is change the current text **channel**, **B20C** memory address:

```
POKE &B20C,#window:POKE &B28F,ink
```

Well, everything is almost solved, a thousand thanks to the teacher for enlightening me!

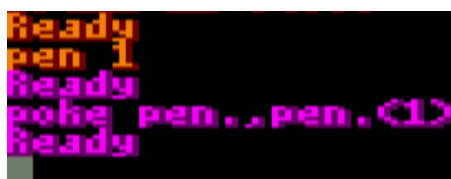
The next thing, since the same program must be able to run on all models, is to identify which firmware version the game is running on and set variables for the memory addresses that matter, which is easy:

```
IF &39=PEEK(&39)THEN
MODE.=&B1C8:CH.=&B20C:MASK.=&B1CF:PEN.=&B28F:REM 1.0
ELSE MODE.=&B7C3:CH.=&B6B5:MASK.=&B7CA:PEN.=&B72F:REM 1.1
```

I define an array of 15 elements (*DIM PEN.(15)*) with the ink values that interest me (and some extra) for the *POKE* to the direction that controls the ink, from 1 to 15:

```
&F0, &3C, &FC, &C3, &F, &CF, &F3, &3F, &FF, &3, &C, &CC, &C0, &30, &33
```

Only I must do this to change the ink of the current channel:



## Pixel screen scrolling vertical

Already before I got with *Wreckingball*, while working on *CPC Invaders*, I had made a small program that featured a vertical *scroll* with a starry background that moved to the pixel (up, all the stars at the same speed). The trick is very simple, you just must manipulate the CRTIC records properly to vary the position of the screen.

For example:

```
10 MODE 1
20 WHILE INKEY$="":GOSUB 100
30 PRINT#7,CHR$(11):FOR n=1 TO 2:LOCATE#7,1+RND*40,1:PRINT#7,".";
40 WEND
50 END
100 FOR y=1 TO 7:CALL &BD19:OUT &BC00,5:OUT &BD00,y
110 FOR n=1 TO 50:NEXT
120 NEXT:OUT &BC00,5:OUT &BD00,0:RETURN
```

For *CPC Invaders* game, I did not use it, because the entire screen moves and I did not have the opportunity to use it during the action, but this time I have introduced it in the sequence of introduction to the game and that gives way to the menu screen.



However, this trick has once left me hanging the *CPC* (emulating in *Retro Virtual Machine*), it has never happened to me with the emulated *CPC* newly initialized and loaded the game, but when it had been a long time since I restarted it, I do not know why it is. With WinAPE it always hung, by loading first the splash screen and then the game. That is why I only use it 2 times in the first execution, during the introduction.

## Title screen

For the menu screen where the game title is also displayed, I wanted it to be in large letters, without using the same font as the rest of the game. I had thought of making some cool text with 3D effect in Gimp, then passing it to the Amstrad, and this time yes, present it on the screen with the help of the *text hack* and transparencies to use all the colors that were necessary. I did some test and fast and it seemed to me that with the low resolution of mode 0 it was not going to look clear enough and on the other hand, I had to modify my UDG editor so that it interpreted a loaded image and converted it to char arrays with the pixels that needed to be shown for each color ... a waste.

I then considered presenting the title with chars so that, united together, they would make up the letters. The first test to see if it could fit everything was such this:



Aside from the lyrics looking ugly, the menu also looked too overloaded. I reduced the number of options to those strictly necessary and designed the chars as pieces that had to shape the 3x5 or 4x5 font, depending on which letter it is:

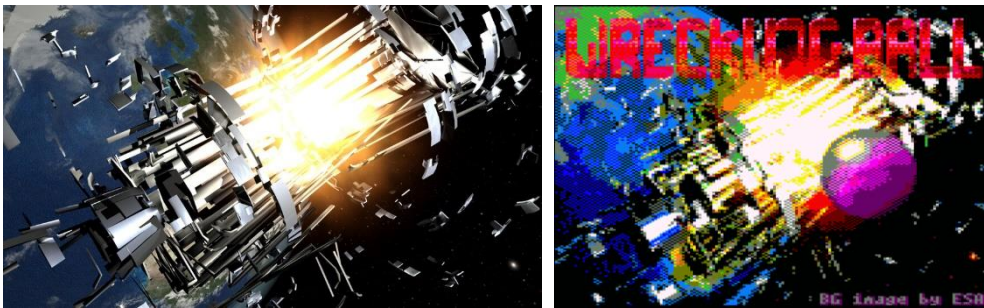


## Loading and presentation screen

This is a *breakout* style game, but what sense could it make that the action was in space? It occurred to me that what was to be destroyed was space junk.

During its development and by chance I have learned of news, etc. concerning the real problem with special garbage. It turns out that there are already companies that are responsible for cleaning the orbit of debris (e.g., *Airbus*, *Astroscale* and *ClearSpace*). In the final phase of the project, just before the presentation to the *CPCRetroDev*, I had the intention of including a loading screen and it occurred to me to visit the website of the European Space Agency (ESA) and more specifically, an article published talking about the problem of space debris (in it is commented that the first collision has already taken place in February 2009).

One of the images that illustrate it is the one I have used as the basis for the loading screen. With Paint.net I have increased the color saturation, I have raised the green component levels to the globe, I have added the ball and converted the resulting image with *ConvImgCPC*. Already in the CPC I have added the texts:

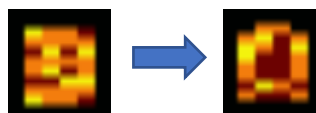


Note that the letters of the title are the same as on the menu screen of the game with the text transparency mode activated and that, with the *text hack* and mode, it is achieved that they really look as if they were translucent (in addition, by the inks resulting from the mixture the color palette applied to the inks seems intentional, but it's pure chance).



## Moving elements during the game

Towards the middle of development, I introduced elements that moved around the screen to hinder the player, diverting the trajectory of the ball and not being so monotonous. I used the char 224, which by default is a smiley face. This stayed that way until practically the end of the project since I had no idea what story to give to those objects. In the end it has remained that they are lost rescue capsules (I will not go into whether they are occupied or not) and the graph has emerged by taking advantage of one of the bytes of the face, since I use the *PEEK* command to detect collision between the ball and the object.



## Speed and gameplay

### Moving things

In the early stages of development, I have only a bouncing ball, a paddle that moves from side to side and some **obstacles**, embryo of what will later be the **bricks** of the game. The bounce pattern of the ball currently is as follows:

- When the ball collides laterally and/or vertically with something, the X and/or Y direction changes, respectively.
- When the ball collides diagonally, both directions, X and Y, *change*.

Immediately a movement problem is noticed: when the paddle is moved, the ball seems to move more slowly and so it is. To save this, the movement of the ball must be managed by interruptions, so that it is as fluid and constant as possible.

When I add the score indicator, and the brick counter that remains to be destroyed, the fact of having to erase the brick, putting in place the 2 chars that correspond to the background behind it and then updating the indicators reveal a new problem: updating all that causes the movement of the ball to slow down excessively.

Again, I make use of interruptions to update the indicators asynchronously, that is, when a brick is destroyed, an AFTER instruction is executed so that soon the number of bricks is updated and from this, another one that updates the score. If several bricks are destroyed very often, it will not give time for the update of the score and brick counter to be executed, so this task is postponed until a more propitious time (when the game will be more relaxed).

### Main loop

The main loop while running the game action is as straightforward as possible, chasing a keyboard response as fast as it is within reach:

```

100 IF NOT INKEY(kr) THEN 200
110 IF NOT INKEY(kL) THEN 210
120 IF 1 > CBR THEN 260
130 IF pw AND 2 THEN 170
150 GOSUB 300: IF INKEY(kp) THEN 100
160 GOSUB 900: GOTO 100

```

Line 150 makes a *GOSUB* to the routine of movement of enemies, so to speak, they will appear on the screen to hinder us and jumps to line 100 if the player has not pressed the **PAUSE** key (pause routine is located on line 900).

It should be noted that the *VARIABLE CBR* (*brick counter*) I do not use it to decide when the stage ends, but also to make the game jump when you have to subtract a life or other circumstances, since by the fact of using interruptions I have noticed that making certain changes of game situation from within a routine called by an interruption has unwanted effects, for example: faster ball movements for a certain time, updates of out-of-place screen objects, etc.


I use the *PW* variable activating and deactivating its bits to establish the player's state in terms of the items that have been collected, but I also use it at certain times when the ball, due to its trajectory, is placed on the player's paddle and it needs to be repainted, when it was not moving.

I also get *extra performance* when, after moving the palette, the same program line will continue to run while the player holds down the arrow key (and the end of the play area has not been reached):

```
200 IF pm>px THEN LOCATE#7,px,25:px=px+1:PRINT#7,pr$;:GOTO 100
ELSE 120
210 IF 2<px THEN px=px-1:LOCATE#7,px,25:PRINT#7,pL$;:GOTO 110
ELSE 120
```

### Bouncing pattern



The scheme of checks that performed the routine of movement of the ball (from line 3000) in each iteration initially was the following (we assume that it moves up and to the right):

	T1	T2
		T3




Collision is checked (bricks, like the background, are also stored in an array of text strings) in three possible positions: top, diagonal and side. If there is brick in each of those positions, the collision is processed and whatever must happen accordingly (lines 4000, 4100 and 4200) happens.




These checks were initially not too much of a problem, but when I added more content to the game and objects circulating on the screen to hinder the player, since the collision check with one of these objects was done by converting the X and Y coordinates of the ball to video memory address and examining the content, it became too heavy, so to speak, the routine of moving the ball and reacting to the player's keystrokes were no longer as fluid as I could want.


To reduce the number of checks to be carried out in each iteration (go from 3 to 1), I propose the possibility of checking only the diagonal position, in the first instance, if there is an obstacle, the vertical and lateral positions will also be checked:

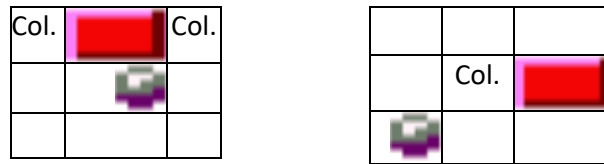
		
		

First the brick is detected in the position in which the green is located, if it exists, the vertical and lateral positions are also checked. But, in a case like these occurs, it would not bounce (remember, in the example, the ball moves up/right):

The way I came up with to alleviate this problem is to include in the array of bricks a value that indicates collision on the sides of the bricks where there is nothing, but on the screen, it is not shown, that is: C  C



It should be noted that, in the example on the right, the collision will be detected just one position before reaching the brick that causes it and will be saved for processing in the next iteration.

Another small change that I introduced is that the ball when colliding diagonally will not reverse its movement in both axes, but the vertical and randomly the horizontal will always be reversed, forcing the player more often to move, since otherwise it seemed to me that it was somewhat boring because there were cases in which you could keep the paddle in a certain position immobile sending the ball always in the same direction until destroying the intended brick.

## Movement and action of the palette

The palette, as usual in this type of game, moves only laterally and only obeys one of the directional keys simultaneously.

When the ball is at the Y coordinate 24 or higher, its X position is checked with respect to that of the paddle and its rebound will be decided based on:

1. If the ball approaches from the left and hits the far left, it will bounce in the opposite direction (the same applies to the opposite direction and side).
  - If the ball approaches from the left and hits the central part, it will bounce off in the same direction.



As I say, the bounce of the ball also takes place when it is at the vertical coordinate 25. The advantage that this has is that, since the paddle moves faster than the ball, it is possible to *rescue* the ball from the fall in *extremis*, that is, to arrive with the paddle and place it on the ball just before it falls to save the loss of a ball..



## Movement of rescue capsules

The capsules that appear and move around the screen are always four and have two movement patterns:










- Two of them will move according to a randomly chosen direction, never diagonally. When they collide with a wall, a new direction of movement is reconsidered. When they reach a certain "height" above the paddle, they will remain there and move laterally or move away.
- The other two will always move diagonally, looking for the interception of the ball and have no height limits.

When the capsules and the ball collide with each other, the capsule destroys and a new one will appear at the top of the screen.

## Collection of effect capsules

When a brick is removed, between 25 and 35% of the time it can (and I say can because then you check the number of bricks that are missing to be removed or the type of capsule) appear a capsule of different colors that alters the behavior of the ball or the palette. These capsules appear and remain static, until after a few seconds they disappear. Making them fall to the bottom to pick them up with the pallet was too much workload, so I ruled out doing so. Instead, it is the ball itself that collects them by colliding with them.

Effects are those that do not involve excessive additional workloads, for example: there is no *multiball* effect. The list is as follows:

	Increase the speed of ball travel.
	Reduce the size of the palette (to some extent).
	The ball can be "redirected".
	Reverse palette motion controls.
	Bonus score with 1000 pts. and pass level (the proposed wink is shown).
	Increase the size of the palette (to some extent).
	Causes the paddle to have some adhesion( <i>magnetism</i> )when it moves.
	Super destructive ball (a single impact is enough to remove the bricks).
	Reduce the speed of ball movement.

## Sound and music

### Sound effects

The premise is that anything that happens on the screen should produce some sound effect, otherwise the thing would be quite bland. Practically everything produces some effect, except the movement of the palette and the objects that swarm the screen, since I was not going to spend overcharging it either.

It is true that in a way the ball hitting effects are inspired by how they sound in the game *Arkanoid II*, but this type of sound was somehow more natural to me than any other. These are the configured envelopes:

```
ENV 1,15,-1,5:ENV 2,3,-5,1
ENT-1,1,-100,3,1,50,2,1,50,2
ENT-2,1,-100,3,2,25,2,2,25,2
```

Bounce effect on play area boundaries (short volume *fadeout*, metallic tone envelope):

```
SOUND 1,200,200,15,1,1
```

Rebound effect on bricks (different tones depending on the direction of vertical movement of the ball):

```
SOUND 1,200+50*sy,80,15,1,1
```

Palette bounce effect (same effect, with more bass tone):

```
SOUND 1,300,200,15,1,1
```

Remove brick effect (the sharpest of all, variation in tone envelope, with some arpeggio):

```
SOUND 1,100,80,15,1,2
```

### "Positional" stereophonic effect

I have provided the program with a certain stereophony for sound effects, according to the position of the ball, of two types:

- Channel number (right, left, or both)
- Volume level (higher volume in the left channel according to the action is closer to the left end of the play area and the same for the opposite case), while do that, I emit the same sound on the other channel with a little less volume to give some spatiality.

For the choice of sound channel, I use a user-defined function:

```
DEF FNst(n)=128+ABS(20>n)+4*ABS(10<n)
```

For the emission of the sound, which will do so by one or another channel or both, according to the previous function:

```
SOUND FNst(x),200,80,15,1,3
```

For the emission of sound based on volume, for example:

```
v=x\6:REM X es La coordenada de La bola
SOUND 129,200,5*(15-v),15-v,1,1:SOUND 132,200,5*(10+v),10+v,1,1
```

### Music

This new project, as in the previous one, is programmed 100% in Basic, so I did not want to use any kind of routines in CM for graphics or music (apart from the tools that the CPC firmware gives you. In addition, the recently published contest rules prohibit it.) From the first moment I was clear that for the incorporation of melody I was going to program a musical composition tool, although I am not a musician, nor do I have too much idea of solfeggio apart from what a pentagram is.

Even if I say I'm not a musician, I do have some ear and I play something with one hand (or I played, now I lack a lot of practice). It comes to me from the same time when I had the *Amstrad*

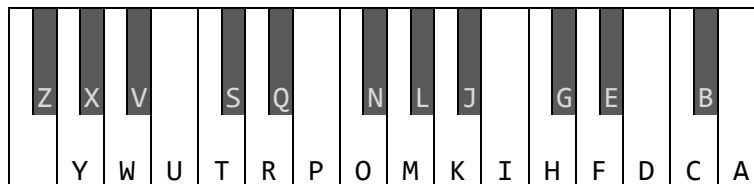
and an ex-brother-in-law who liked musical keyboards and let me mess around with them. Some of these synthesizers had a drum machine, resources more than fun to handle given my limitations. Later I bought a small Yamaha CS01III monophonic synthesizer (not MIDI) and a MIDI keyboard for the PC, with which I have also played some sequencers and MOD trackers.

The ultra-simple program that I have created to help me in the composition (and which I have called *Music Box*), is oriented to the operation of mod trackers, only, at a very basic level, without instruments or effects.

Again, I work with strings text *arrays* in which I store numerical values (1 to 5) to launch drum sounds (the drum machine) or alphabetic chars for the tones of the notes, based on the following formula (extracted from the CPC manual):

```
ma=ASC(char$)-65
SOUND 1,UNT(62500/(440*(2^((10-ma)/12)))),40,15
```

The alphabetical char - notes distribution would be as follows, transferred to a piano keyboard:



In *Music Box* the patterns are organized horizontally and on the screen (which seems a bit messy) the indicator of the tick or playback point (64 chars in length) is shown at the top, on the right the *tempo* (value that is passed to the EVERY that will invoke the interruption to sound the notes) and the vertical position of the cursor (which marks the current pattern, although in the capture this value does not match since it is in playback mode and as you can see, the cursor is lower, a position that corresponds to the pattern that is sounding at that moment).

The first pattern line (in which only numerical values from 0 to 5 are seen) corresponds to the drum machine, which will sound through the central channel and onwards, when they contain alphabetic chars, the odd lines correspond to the left channel and the even lines to the right. The three lower lines correspond to numerical values that will sound battery on both channels, right and left alternately and are played individually.



The program as well as the UDG editor allows you to export the work to file .BAS with ASCII format, containing the DATA lines:

```

20000 DATA BCECEGGCECEGGMNONIK
20010 DATA
1000303020003030103010302030443010003030200030301030103020342430
20020 DATA
Q000L0J0E0G0H0J0L0L0J0N0B0000000Q000L0J0E0G0H0J0L0J0G0N0Q0N0Q0000
20030 DATA
X0X0Q0Q0X0X0Q0Q0X0X0Q0Q0X0X0Q0Q0X0X0Q0Q0X0X0Q0Q0X0X0Q0Q0X0X0Q0Q0
20040 DATA
S000N0L0G0I0J0L0N0N0L0P0D0000000S000N0L0G0I0J0L0N0N0L0P0D0P0D0000
20050 DATA
Z0Z0S0S0Z0Z0S0S0Z0Z0S0S0Z0Z0S0S0Z0Z0S0S0Z0Z0S0S0Z0Z0S0S0Z0Z0S0S0Z0Z0S0S0
20060 DATA
L0L0E000L0L0X000L0L0E000L0L0X000N0N0G000N0N0B000N0N0G0G0N0N0B000
20070 DATA
0L0L0E000L0L0X000L0L0E000L0L0X000N0N0G000N0N0B000N0N0G0G0N0N0B000
20080 DATA
S000Z0Z0S0Z0Z0S000Z0Z0S000Z0Z0Q000X0X0Q0X0X0Q000X0X0Q000X0X0
20090 DATA
SSSSZZZZSSZZZZSSSSZZZZSSSSZZZZQQQQXXXXQQXXXXQQQQXXXXQQQQXXXX
20100 DATA
00000000000000000000000000000000E000G0H0J0L0J000G000H0J0L0J0Q000
20110 DATA
Q0Q0J000L0L0J000Q0Q0J000L0L0J000Q0Q0J000L0L0J000Q0Q0J000L0L0J000
20120 DATA
33300033300033355444000333000333000333552221122112212211111111
20130 DATA
4343400400304000300040300403000043434004003040003000403004030000
20140 DATA
111111111111111112222222200000000111111111111111111112342342300000000

```

The first *line DATA* indicates, by means of alphabetic chars, the number of the pattern that will be always reproduced, once I read in full the last char of that line it will go to the first position. The advantage of using alphabetic chars is that it has made it easier for me to read and introduce arrangements to the music so if they had been numerical codes.

The player is such that this:

```

30000 IF tk=1 THEN my=ASC(MID$(MUS$(0),mx,1))-65:mx=mx+1:IF
mx>LEN(MUS$(0))THEN mx=2
30010 ma=ASC(MID$(MUS$(my),tk,1))-65:IF ma>=0 AND ma<58 THEN
GOSUB 30300 ELSE IF ma<>-17 THEN ON ma+17 GOSUB
30200,30210,30220,30230,30240:GOTO 30050
30020 ON VAL(MID$(MUS$(1),tk,1))GOSUB
30100,30110,30120,30130,30140
30030 IF 1=my OR 14=my THEN 30050
30040 mc=ASC(MID$(MUS$(my+1),tk,1))-65:IF 0<=mc AND 58>=mc THEN
GOSUB 30400
30050 tk=1+tk MOD 64:RETURN
30100 SOUND 130,500,20,14,7,6:RETURN
30110 SOUND 130,500,20,14,7,,1:RETURN
30120 SOUND 130,5,5,14,6,0,6:RETURN

```

```

30130 SOUND 130,10,60,14,8,,3:RETURN
30140 SOUND 130,100,200,14,3,7:RETURN
30200 SOUND 129+3*(tk MOD 2),500,20,14,7,6:RETURN
30210 SOUND 129+3*(tk MOD 2),500,20,14,7,,1:RETURN
30220 SOUND 129+3*(tk MOD 2),5,5,14,6,0,6:RETURN
30230 SOUND 129+3*(tk MOD 2),10,60,14,8,,3:RETURN
30240 SOUND 129+3*(tk MOD 2),0,10,0,5,,1:RETURN
30300 SOUND 129,UNT(62500/(440*(2^((10-ma)/12)))),40,15,8:RETURN
30400 SOUND 132,UNT(62500/(440*(2^((10-
mc)/12)))),40,13,8,8:RETURN

```

And to start and sound the music:

```

30500 DIM MUS$(140):RESTORE 20000:FOR n=0 TO 14:READ
MUS$(n):NEXT tk=1:mx=1:EVERY 5 GOSUB 30000:RETURN

```

As options in the playback, I have available (of course) the possibility of playing the melody slower or faster (although not too fast, since I immediately run out of time). for example, when the game is over, I play a certain part of the melody more slowly to make it look sadder.

I can also add notes that are not actually in the *DATA* as an accompaniment to the melody, a few lower octaves, etc.

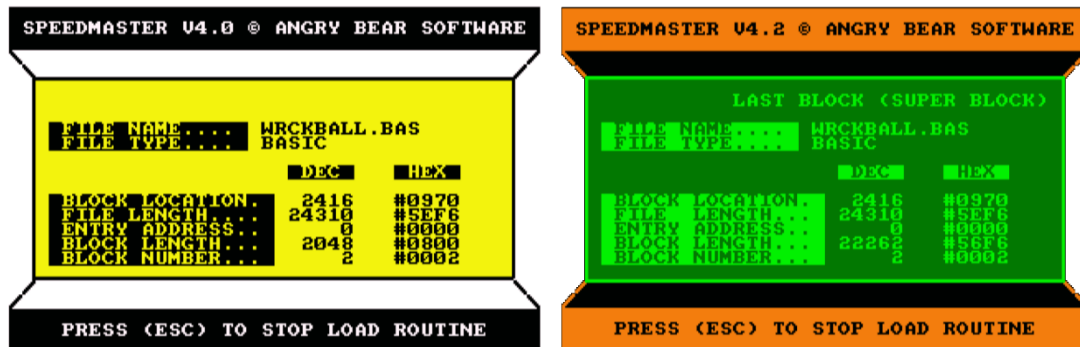
In total I have created 4 different melodies and 1 variation, to liven up the following parts of the game:

- Menu screen and top scores.
- Music at the beginning of each stage.
- Music for the wink.
- Music for the end-of-game message (extract from the menu melody at a slower speed).
- Music on the Initials Intro Screen by Getting One of the Top 10 scores.

## Loading system from tape

### Fast loading of *extra-sized blocks*

When I had my *Amstrad CPC 464* f.v., I had a lot of original games (most), but like every neighbor's son I also had my tapes copied, some copied with double plate, because they contained games with turbo charging and there was no choice, but I also had games with normal load, in blocks. To copy such programs you could use copycats, in which you entered the original tape, read all the blocks that could fit in memory and then introduced the virgin tape and dumped there the blocks read. The copy I used was *SpeedMaster* from *Angry Bear Software*.



This copy was programmed in Basic, and the loading/recording routine was CM. I don't know exactly how it happened, but tinning with it I realized that, while the first block of each file (which was made up of several blocks) had to be 2048 bytes, **from the second it could be larger**. This offered the advantage that to load any program saved me the time of loading the header of each block and the white space on the tape that separates each one, thus taking less time to load.

I modified *SpeedMaster* so that, in the loading phase, from the third blocks, the data in memory was moved, just at the end of the second block and modifying the header of this to correspond to the new block length, when recording this block on tape a block as large as it could have fit in the loading phase was obtained.

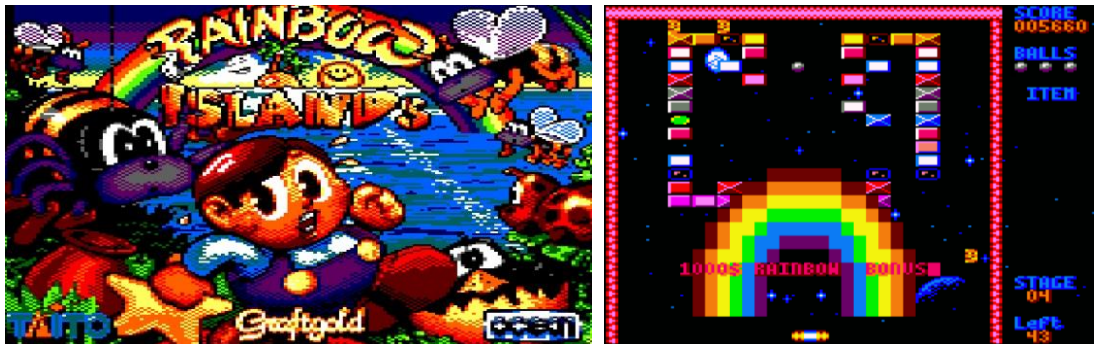
All this I have applied to the loading of *Wrecking Ball*: It is a program in Basic of loading by blocks, but except for the first, it is loaded as if it were turbo charging, but apart from its length, it can be considered that they are normal blocks, the firmware can load them perfectly by itself.

I also must comment that the speed at which it is recorded is 2000 b/s. (I have done tests with the *Retro Virtual Machine* emulators and with WinAPE and the second is not able to **correctly interpret** CDTs with blocks recorded in this way, so I had to insert a 1-byte block between the splash screen and the game to load correctly.

## Wink

This year's reference in the *CPC RetroDev* (2021) points out that a rainbow should appear during gameplay, as a nod to Ocean *Software's Rainbow Islands* game.

A rainbow effect has been included when collecting a golden capsule (🟡), or by **pressing R+ PAUSE**), which grants 1000 points and an extra ball. After showing the rainbow also sounds (in my own way) the final part of the main melody of the game *Rainbow Islands*, which takes place at the end of each phase of said game.



## Description of the parts of the program

1-50	License, initialization and start of the program.
60-70	Start of game
100-150	Main loop of the game.
160	Jump to pause of the game.
170-250	Movement and on-screen printing of the player's ship.
260-290	Jumps at the end of the stage, loss a ball or end of game.
300-470	Movement capsules moving around the screen.
500-580	Erasing, collision, printing and initializing moving capsules.
600-640	End current scenario, jump to generation of new scenario and main loop.
900-990	Pause loop input and output. Initiate interruptions to move ball, etc.
1000-1390	Generate scenario.
1400-1800	Initialize vars for bricks and background. Print background, bookmarks, etc.
3000-3810	Ball movement, collision detection, bounce on player ship, etc.
4000-4260	Process collision with bricks or item capsules.
4300-4650	Score when destroying brick and decide if item capsule appears.
4700-4930	Process collision with item capsule and apply effect of acquired item.
5100-5950	Char-by-char text on screen and other effects (rainbow, etc.)
6000-7390	Define character map.
7400-7930	Define functions, variable arrays, envelopes, and memory addresses.
8000-8980	Getting Started Screen and Options Menu.
9000-9990	Screen of best scores and introduction of initials.
10100-10240	Redefine game keys.
20000-20210	Music data
30000-30540	Melody player and melody initialization to play.
64000-64040	Program Output.

## Resources and tools used

Microsoft Word 2019 (Microsoft 365) © Microsoft Corporation

Paint.net. Copyright © 2014 dotPDN LLC, Rick Brewster, and contributors.

ConvImgCPC. Demoniak. (<http://deplanque.free.fr/ConvImgCpc/new/>)

RetroVirtualMachine v2.0 © 2018-19 Juan Carlos González Amestoy

WinAPE 2.0 Beta 2 © Richard Wilson

SpeedMaster v4. Angry Bear Software

European Space Agency (ESA), article on space debris on ESA's website:

[https://www.esa.int/Safety\\_Security/Space\\_Debris/About\\_space\\_debris](https://www.esa.int/Safety_Security/Space_Debris/About_space_debris)

[https://www.esa.int/ESA\\_Multimedia/Copyright\\_Notice\\_Images](https://www.esa.int/ESA_Multimedia/Copyright_Notice_Images)

GNU GPL License (<https://www.gnu.org/licenses/>)

## Farewell

I thank every reader for their interest in this project and in this document and I hope that my works will be enjoyed even a small fraction of what I have enjoyed creating them.

For me, the *Amstrad CPC* microcomputer has always been and has shown me that it is the best tool for learning in programming at all levels, both for its excellent user manual and reference and for, in my opinion, its unsurpassed exploitation characteristics in its programming.

*Best Regards.*