

Harrier Attack, by Durell Software Ltd, published by Amsoft, is an iconic game that is instantly recognizable as it came bundled with the machine. I remember spending many hours playing it with my dad when we got our first Amstrad, a CPC 464, back in 1987. I flew the plane and my dad pressed the space bar to bomb the targets. Many people have fond memories of growing up, playing this game.



Some are under the impression that Harrier Attack was written in the BASIC programming language. They may think this due to its slow speed, blocky movements and 8x8 character-sized sprites. But it is written in machine code. Its slow speed is due to the entire screen being scrolled one byte at a time. There are also program loops inserted by the programmer to deliberately slow the game down, presumably to make it easier for children to 'win' on the earlier levels.

As Harrier Attack is a fairly simple game, I wanted to see if I could disassemble it, to try to speed it up. I opened the BASIC loader in WinAPE to see where the main program was located. Once I had found this, I highlighted the code in WinAPE's disassembler and disassembled it.

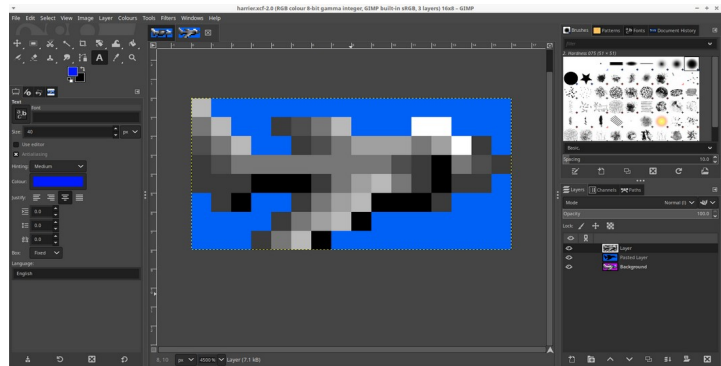
The first task was to identify which parts of the code were functions. I did this by creating a division between any commands ending in RETurn. This helped to make the code easier to read. The next job is to see which firmware functions were being used by the program by looking up their addresses and replacing them with proper names. Once the code was divided into functions, anything left over can be regarded as data, and it is just a case of working out whether they are variables, sounds or sprites.

I found the function that printed 8x8 tiles on the screen and used it to print every tile that is available in the game. Using this debug test, I was able to identify all of the tiles in the table. I also found the function that printed blocks of 8x8 tiles, and was able to identify the corresponding table that contained all the sprite definitions for clouds, tanks, lorries, buildings, etc. I identified the main program loop which takes you through all the stages of the level.

Once I had the code cleaned up, and it was recompiling and running in WinAPE without using the loader, I was able to start tinkering with it.

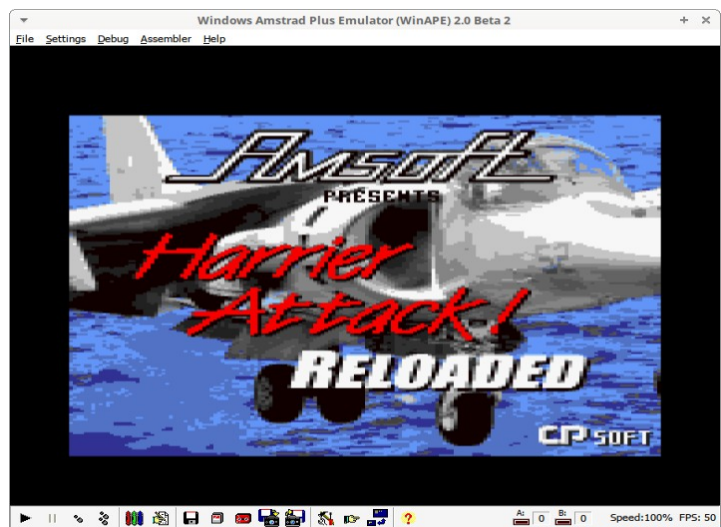
I identified the function that scrolled the screen and tried modifying it to see if I could improve it. Using some Z80 assembly language tutorials on CRTC scrolling, I found I was able to replace the byte by byte copy function with single CRTC scroll command. It worked! And it was fast! Although I found the screen gradually ascended vertically too, so I needed to modify the vertical positioning of the graphics to compensate. The flight instruments also scrolled with the screen, which was a big problem. But looking at some Amstrad Plus screen splitting examples, I found I was able to keep this portion of the screen stationary while still using the CRTC to scroll the main gaming area.

I found some examples of how to create hardware sprites on the Plus machines. Each sprite has its own palette of 16 colours, and can be displayed in any resolution. I designed a simple plane graphic in GIMP, not an easy task when you are working with a sprite 16x8 pixels in size, and then typed the values into WinAPE's assembler. It was fairly simple to display the Plus hardware sprite in place of the Amsoft one, although I needed to make the plane sprite in two halves, as sometimes part of the plane sprite disappears behind a cloud while the other part remains visible.



The new method of scrolling also created some problems. When the screen was scrolled byte by byte, planes would naturally move towards the left side of the screen without their coordinates being updated. It looks as if they are moving, but they are actually not. The hardware sprites stay in the same place until I tell them to move. So the code had to be modified to cater for this.

Getting the game to compile on cartridge was not too time consuming as I had already created my Mimo's Quest game on cartridge. So I borrowed the BASH script from it, created a new loading screen and palette, and eventually I had the game running from cartridge. The loading screen was created using a public domain image of a Harrier Jumpjet landing on an aircraft carrier - "U.S. Marine Corps Maj. M. J. Shulte lands an AV-8B Harrier aircraft on the flight deck of the amphibious assault ship USS Peleliu (LHA 5) May 15, 2008,



while under way in the Pacific Ocean." I converted the image in ConvImgCPC. It worked well with the Plus palette as most of the plane is greyscale and the background is all the one colour. The close up shot also shows plenty of detail on the aircraft. I used the original logos for the game title and publisher.

Below is my BASH script for creating the cartridge...

```
cd /home/chris/Desktop/MimoSource/HarrierAttack/
../makecartridge/rasm.exe -amper HarrierAttackSourceNew2_alt_CRTC_CART6.asm
../makecartridge/rasm.exe -amper AMSTRADFONT3.asm AMSFONT
../makecartridge/rasm.exe -amper HARR_SCR2.asm HARR_SCR

split -b 16k rasmoutput.bin game_harrier_ # SPLIT INTO 16K CHUNKS - THIS GETS COPIED TO RAM WHEN CART
LOADS
cd ../makecartridgeharrier

rm ./boot.bin
rm ./mycart.bin
```

```

./rasm.exe -amper boot2.asm boot
./rominject -p 0 -o 0 boot.bin ./mycart.bin
./rominject -p 1 -o 0 /home/chris/Desktop/MimoSource/HarrierAttack/HARR_SCR.bin ./mycart.bin
./rominject -p 2 -o 0 /home/chris/Desktop/MimoSource/HarrierAttack/game_harrier_aa ./mycart.bin #0000-3FFF CODE
./rominject -p 3 -o 0 /home/chris/Desktop/MimoSource/HarrierAttack/AMSFONT.bin ./mycart.bin #0100-3F00 CODE

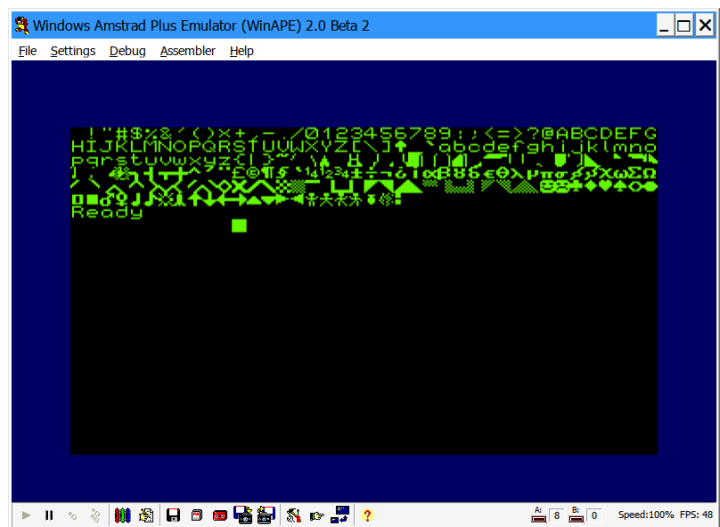
rm ./HarrierAttackPlus.cpr
rm /media/chris/SAMSUNG/Data/Emulator/WinAPE2021/ROM/HarrierAttackPlus.cpr
./buildcpr ./mycart.bin ./HarrierAttackPlus.cpr
cp ./HarrierAttackPlus.cpr /media/chris/SAMSUNG/Data/Emulator/WinAPE2021/ROM/HarrierAttackPlus.cpr

```

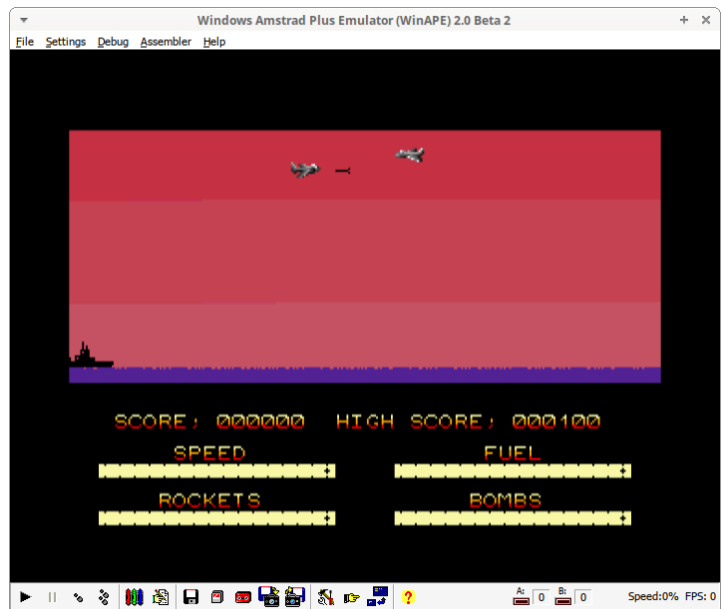
I had to rearrange the screen and code locations in memory. The instrument panel is located at &4000. The game screen is located at &8000. The main game code is located at &C000. And the extra functions and tables I created are located at &0000. As the ASIC is paged in at &4000, I moved the screen so the ASIC would not obscure my game code. As the split screen area of memory is the one that will be least used, I can safely have the ASIC paged in all the time until I need to update the flight instruments.

I thought it would be a good idea to improve the menu. The original Harrier Attack menu was really slow at redrawing. I don't know why the programmer used this method, but basically they used a firmware text command to 'clear' the entire screen. I replaced this function with a faster assembly one. As I am unable to use firmware for the Plus cartridge version of the game, I also had to replace the text function with my own one. I created a short assembly program to print the entire ASCII character set on screen, and then another function to rip those bytes from the screen and save it in a linear format in memory. This enabled me to quickly produce a table of sprites for my text printing function.

I also wanted to change the font so it didn't look like the game was created in BASIC. When I was a child, I remember being fascinated by a program I typed in from Amstrad Action that was able to turn plain text into multicoloured writing. I have no idea how it worked, as it was all machine code. But it looked amazing. I have an idea how it works now. I adapted my text writing function so that as the sprite data is copied to the screen line by line, the bits are changed depending on which line it is working on. It was a little complicated, but eventually I worked out which bits needed inverted or swapped until I was able to create multicoloured text. I also decided to use the font I created in BASIC when I was a child.



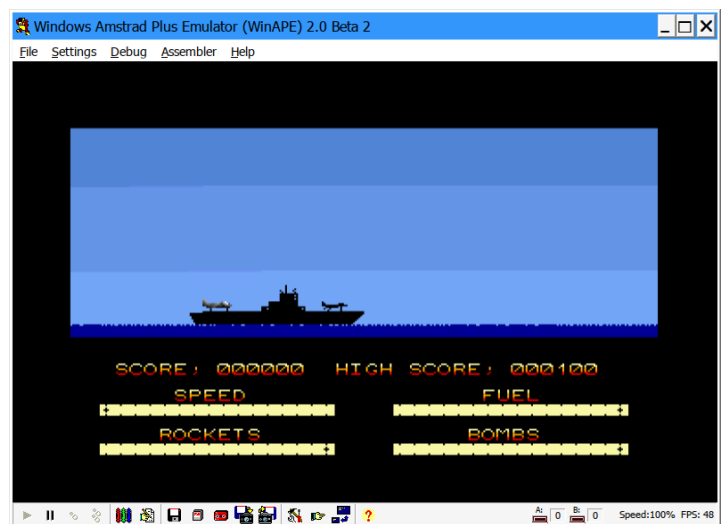
The Plus machines' colour palette is pretty impressive. I remember being amazed as a child, reading reviews of Burnin' Rubber and how the sky slowly changed from daylight, to dusk, to night time. It really added to the atmosphere of the game. So I decided to try and do the same for Harrier Attack. Using a couple of tables of integers, and a function to work out which integer to increment or decrement until we match another table, I was able to do a palette fade similar to Burnin' Rubber. Beginning the fade at certain points in the level adds to the realism.



Harrier Attack uses the `kl_time_please` function to record the passing of time so the game knows how long each part of the level should be. So I had to add an interrupt with my own function to record the time in a similar manner. Eventually I worked out how to do this using a 300th of a second interrupt and two 16bit numbers which are incremented sequentially.

The game also uses the keyboard scanning functions of the firmware. So these needed to be replaced too. I put a keyboard scanning function into the interrupt, to be called every 50th of a second, and then replaced the firmware calls to check for keypresses with ones to check the buffer created by this function instead. And it worked!

The palette fade to dusk and night created a new problem. The flight instruments would also change colour with the main game screen, and this looked bad. As I had a timer and keyboard interrupt running, I thought it might be a good idea to see if I could modify the palette in the interrupt too. Programmers used to do this to get extra colours on the screen. I left the ASIC unlocked and paged in by default. As the ASIC pages in over my split screen memory, I only need to page them out when I update the flight instruments. It worked, but the palette was unstable and flickered. I figured out that an interrupt was being triggered while I was updating the flight instruments, and consequently I was trying to write my palette data to the ASIC when it was paged out. I added a variable to let the interrupt function know the state of the ASIC, whether it is paged in or out, and the palette split started to function correctly.

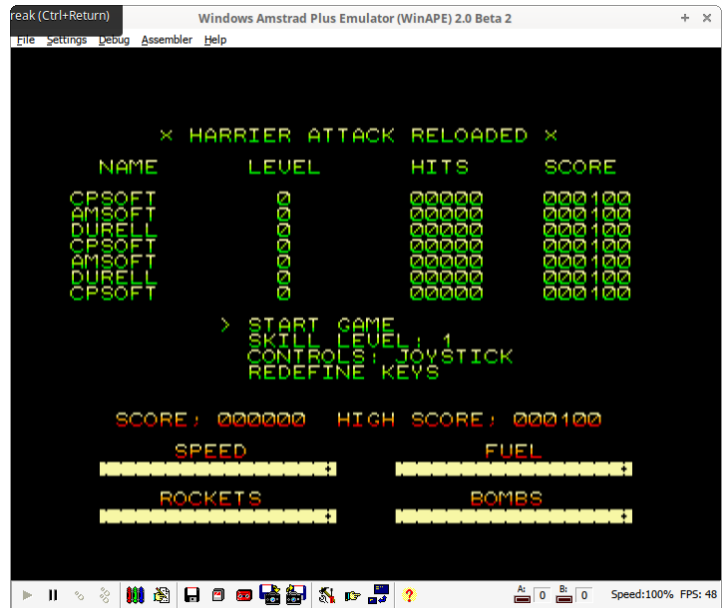


Harrier Attack uses the same pen for drawing the sea and clouds. A timer counts down until the plane is thought to be over land, and then the blue colour is changed to white. Using a split palette, I am able to have sea and clouds using the same pen without having to use a timer to switch the colour. And I may be able to use the other pens to do other things.

As I was creating a cartridge version for the Plus machines, it would be nice to be able to support the GX4000, which has no keyboard. Although Harrier Attack could be controlled by Joystick, you still needed to use the keys to drop bombs and the escape key to eject. A keyboard was also necessary to control the menu to start the game and enter your name into the highscore table.

I amended the highscore table so the player is able to press the up and down directions on the joystick to select a letter. Fire enters the letter in the scoreboard, the left direction deletes the last letter, and pressing fire when no letter has been selected completes the process. There is also a problem with the limitation of a two button joypad. As there

are missiles and bombs to fire, there are no more buttons to press for controlling the ejector seat. Some suggested using the pause button as an ejector button. I decided to create a function so the player could choose what keys they wanted to use. Harrier Attack's original keys were really awkward to use, so it is handy being able to choose keys that are familiar to you.



Harrier Attack used the km\_test\_key firmware function. As I have no firmware in the cartridge, this function had to be replaced. I tried creating my own function to read the keyboard, scan a table and then workout which line and bit had been triggered. Running it while the game was playing slowed everything down a lot. Instead I found that by using self-modifying code, I am able to insert a specific line address and bit-checking function directly into the routine for moving the sprites. A lookup table holds the 10 memory addresses of the lines populated by the keyboard scanner. There is a table of opcodes for checking each bit in a byte. When the player redefines their keys, we work out what line and bit has been triggered, and the appropriate line address and opcode are copied directly into function to detect the keypress. It saves a lot of processing time.

As Harrier Attack used the Amstrad firmware for sound, I had to recreate some of the sound effects by accessing the AY sound chip directly. It's quite complicated to understand at first, but after working out through extensive trial and error which registers produce which sounds, I was able to write a short routine for calling the sounds from the main program events. I was going to use the Arkos Tracker program, but unfortunately its player uses the shadow registers of the Z80. As I am already using the shadow registers to speed up my interrupts, I decided to try to code my own routine. There is a buffer for each sound channel, and every fiftieth of a second, this buffer is checked to see if it has been filled by a command. If so, the bytes are loaded into the registers and the sound is played. As the AY chip has no duration function, it plays your sound until you tell it to stop. This is not suitable for sound effects or music, so I had to add an extra byte in the buffer to act as a timer. This byte is incremented every time

we call the routine, every fiftieth of a second, and if it equals the duration set by the calling function, the channel is muted. I also made a 'waitchannel' function. It means the function will keep checking the buffer to see if the sound has finished playing before it moves on to the next command. This is useful for the ship's horn, that sounds at the start and end of the mission. And of course, the silencechannel command to mute the channels once the game has finished. It works well enough for Harrier's limited sound effects, though it would be handy to have proper envelopes for creating complicated sounds, though that would require a bit more thought.

I also changed the night palette so it has a green tinge. This makes it look like 'nightvision' has been activated. I altered some of the town building sprites to give the windows different colours to make it look like the rooms are occupied.

Dropping bombs on the sea or land just causes them to disappear. I thought it might be a nice effect for them to explode and leave a crater, so I adjusted the collision detection routine to allow for this. I created a small 8x8 sprite with a hole in it, and whenever a bomb is dropped on the land, it will make a low thud and leave a mark. It's a nice touch that adds a bit more realism.

One of the big problems of using the AY sound chip raw is the lack of envelopes. The AY does have an envelope function, which is quite handy, but the problem is if you use them, the volume parameter is ignored and it plays at full volume. I needed a way of softening some of the explosion noises, so I made my own volume fade routine. Using a status bit, I can tell my function whether to use a timer for the duration of the noise and then silence the channel, or to lower the volume each time a number of steps are counted down. This allows me to use both the volume control and envelopes, and have the sound dissipate. The same technique could be used for reverb, or for more complicated sound effects.