Possibili titoli:

1. Amiga Game Programming in Assembly
2. Assembly Language Game Programming for Amiga
3. Amiga Assembly Game Programming
4. Retrogame programming for Amiga in Assembly
5. Old school game programming for Amiga
6. Bare metal game programming for Amiga
7. Amiga Bare Metal Game Programming
8. Programming a shoot’em up game for Amiga
9. Programming games in assembly for Amiga
10. Black art of assembly game programming for Amiga

Blank page

**Table of contents**

[1. Tools & Documentation 8](#_Toc181709679)

[Visual Studio Code 8](#_Toc181709680)

[Amiga Assembly Extension for VS Code 9](#_Toc181709681)

[First Project 10](#_Toc181709682)

[WinUAE Amiga Emulator 11](#_Toc181709683)

[WinUAE Configuration 13](#_Toc181709684)

[Taking Screenshots 19](#_Toc181709685)

[Screen recording 20](#_Toc181709686)

[Window size and video mode 21](#_Toc181709687)

[Joystick configuration 21](#_Toc181709688)

[Source code versioning with Git 21](#_Toc181709689)

[Online resources on Amiga programming 26](#_Toc181709690)

[2. Amiga hardware overview 27](#_Toc181709691)

[Amiga hardware architecture 27](#_Toc181709692)

[CPU 27](#_Toc181709693)

[RAM memory 28](#_Toc181709694)

[Custom Chips 29](#_Toc181709695)

[OCS features 29](#_Toc181709696)

[ECS Features 30](#_Toc181709697)

[AGA Features 30](#_Toc181709698)

[3. Motorola 68K Assembly Short Course 31](#_Toc181709699)

[Assembler directives 31](#_Toc181709700)

[Labels 32](#_Toc181709701)

[Comments 32](#_Toc181709702)

[Move instruction 33](#_Toc181709703)

[Debugging 34](#_Toc181709704)

[Data structures 37](#_Toc181709705)

[Arithmetic instructions 38](#_Toc181709706)

[Shift instructions 39](#_Toc181709707)

[Logical instructions 40](#_Toc181709708)

[Conditional statement 40](#_Toc181709709)

[Loops 41](#_Toc181709710)

[Subroutines 42](#_Toc181709711)

[4. Take control of Amiga hardware 44](#_Toc181709712)

[DMA 44](#_Toc181709713)

[Amiga Operating System and libraries 45](#_Toc181709714)

[Take System 45](#_Toc181709715)

[Release System 47](#_Toc181709716)

[Main program 48](#_Toc181709717)

[Main loop 48](#_Toc181709718)

[Run the program 48](#_Toc181709719)

[5. Game concept 50](#_Toc181709720)

[6. The Copper and copperlist 54](#_Toc181709721)

[Move Instruction 54](#_Toc181709722)

[Wait Instruction 54](#_Toc181709723)

[The Copperlist 55](#_Toc181709724)

[Save and restore the system copperlist 56](#_Toc181709725)

[7. How to display images on screen 59](#_Toc181709726)

[RAW format 59](#_Toc181709727)

[RAW Converter 60](#_Toc181709728)

[Palette 60](#_Toc181709729)

[Bitplanes 63](#_Toc181709730)

[Display window size 67](#_Toc181709731)

[8. Blitter 71](#_Toc181709732)

[Introduction 71](#_Toc181709733)

[DMA Channels 71](#_Toc181709734)

[Logic functions 72](#_Toc181709735)

[Blitter busy 72](#_Toc181709736)

[BLTSIZE and how to start Blitter 72](#_Toc181709737)

[Modulus 73](#_Toc181709738)

[Shift 74](#_Toc181709739)

[Masking 74](#_Toc181709740)

[Example of copying an image to the screen 74](#_Toc181709741)

[9. Tiles and tilemaps 79](#_Toc181709742)

[Tiles 79](#_Toc181709743)

[Tilemap 79](#_Toc181709744)

[Why are tiles used? 80](#_Toc181709745)

[Creating the tileset 80](#_Toc181709746)

[Creating the tilemap 87](#_Toc181709747)

[Alternative way to create a tileset/tilemap 93](#_Toc181709748)

[Export tilesets, palettes and map 94](#_Toc181709749)

[Import tilesets and tilemaps into our program 96](#_Toc181709750)

[Drawing a tile 97](#_Toc181709751)

[Drawing a column of tiles 100](#_Toc181709752)

[Let's fill the screen with tiles 101](#_Toc181709753)

[10. Scrolling background 104](#_Toc181709754)

[Blitter-based scrolling 104](#_Toc181709755)

[Background initialization 107](#_Toc181709756)

[Background Drawing Using Blitter 111](#_Toc181709757)

[Vertical Blank 112](#_Toc181709758)

[Background scrolling 113](#_Toc181709759)

[Hide the first 16 pixels to remove noise 115](#_Toc181709760)

[Double buffering 115](#_Toc181709761)

[Hardware scrolling 117](#_Toc181709762)

[Implementing hardware scrolling 119](#_Toc181709763)

[Vertical scrolling 123](#_Toc181709764)

[Implementing Vertical Scrolling 127](#_Toc181709765)

[11. Hardware sprites and joystick reading 136](#_Toc181709766)

[What are hardware sprites? 136](#_Toc181709767)

[Example 1: Joystick movement of an attached sprite 136](#_Toc181709768)

[Sprite DMA Channels 136](#_Toc181709769)

[Sprite size 137](#_Toc181709770)

[Sprite coordinates 137](#_Toc181709771)

[Sprites data structure 138](#_Toc181709772)

[Creating Sprite Data Structure 139](#_Toc181709773)

[Inspecting the Sprite Data Structure with Hex Editor 140](#_Toc181709774)

[Sprite pointers 143](#_Toc181709775)

[Sprite colors 144](#_Toc181709776)

[Sprites “attached” 145](#_Toc181709777)

[Priority between sprites and playfields 146](#_Toc181709778)

[Placing a Sprite 147](#_Toc181709779)

[Reading joystick 148](#_Toc181709780)

[Moving the Sprite with the Joystick 150](#_Toc181709781)

[Collisions 151](#_Toc181709782)

[AGA Sprites 155](#_Toc181709783)

[12. Multidirectional scrolling 156](#_Toc181709784)

[13. Blitter objects (Bobs) 157](#_Toc181709785)

[14. Player’s ship 158](#_Toc181709786)

[Enemies 159](#_Toc181709787)

[Firing bullets 160](#_Toc181709788)

[Collisions and explosions 161](#_Toc181709789)

[Text rendering 162](#_Toc181709790)

[Information panel 163](#_Toc181709791)

[Power-ups 164](#_Toc181709792)

[Satellite 165](#_Toc181709793)

[Game states 166](#_Toc181709794)

[Title screen 167](#_Toc181709795)

[Game over screen 168](#_Toc181709796)

[Keyboard and mouse input 169](#_Toc181709797)

[End level boss 170](#_Toc181709798)

[Sound 171](#_Toc181709799)

[Loading level data from disk 172](#_Toc181709800)

# 1. Tools & Documentation

At the Amiga times (late 80s – early 90s), video game programmers used the same Amiga, perhaps in the more professional versions Amiga 2000, 3000, 4000, to write code and debug it. It wasn't a very convenient method, because if the computer crashed during execution, you risked losing the unsaved code. To solve this problem, they began using another computer to write the code, often an IBM PC clone, and transfer the code to the Amiga via the serial port.

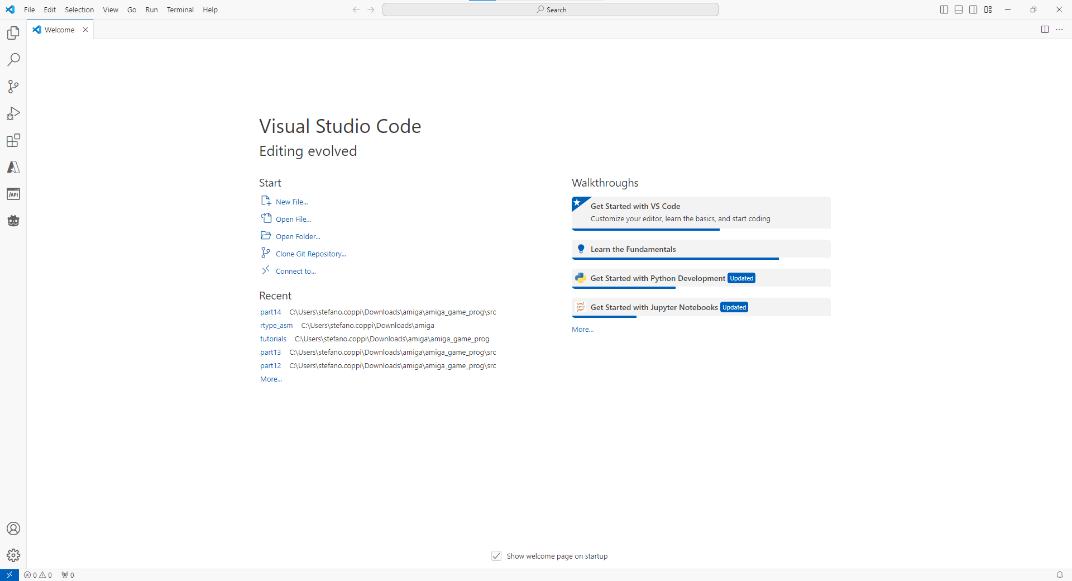
## Visual Studio Code

Today we are lucky because there are much more advanced development tools, which allow you to write code, format it, debug it and even test it. For writing the code we will use Microsoft's "Visual Studio Code", which can be downloaded free of charge from the following URL:

<https://code.visualstudio.com/download>

This tool is available for Windows, Mac Os and Linux.

After installation, when you launch Visual Studio Code, you will see the window shown in the following figure.



Visual Studio Code allows you to install extensions. To program in Assembly for Amiga, we'll install a fantastic extension that will make the task easier for us.

## Amiga Assembly Extension for VS Code

Press the “Extensions” button on the left (circled in green in the next figure). Visual Studio code will display a panel called "Extensions Marketplace", containing a list of available extensions. We type "amiga assembly" in the search box. An extension called "Amiga Assembly" will appear at the top of the list (highlighted in yellow in the next figure). Click on the "Install" button to install it.

Immagine che contiene testo, schermata, software, Icona del computer

Descrizione generata automaticamente

The extension we installed provides the following features:

* Compiling the assembly source code for the Motorola 68000 using the VASM assembler.
* Syntax highlighting and formatting for Motorola 68000 Assembly
* Calculator
* Color Editor
* Hexadecimal to Decimal Conversion
* Online Documentation
* Debugger
* sin/cos table generator

To learn more about using the extension, visit the following page:

<https://github.com/prb28/vscode-amiga-assembly/wiki/Getting-started>

## First Project

Now let's see how to create an example project. First, let's create a folder for our project, calling it "chapter1". Press the key combination CONTROL + SHIFT + P. The Command Palette will appear, shown in the following figure. Let's select the first item, "Amiga Assembly: Create example workspace". In the next dialog box, select the "chapter1" folder you created earlier. You will be asked for the name of the executable program, call it "chapter1".

Immagine che contiene testo, elettronica, schermata, schermo

Descrizione generata automaticamente

Visual Studio Code will show the image in the next figure. On the left we find the Explorer, which allows you to explore the folder. Let's see the contents of the subfolders:

* .vscode contains the files to configure builds and debug
* build contains compiled object code (\*.o file)
* include contains some files with the definitions of constants and macros, to be included in the assembly source
* uae contains a virtual hard disk that is mounted in the Amiga emulator and that contains the executable of our program.
* chapter1.s contains the source code for the sample.

If you click on the chapter1.s file, it will be displayed in the main pane, with the colored syntax.

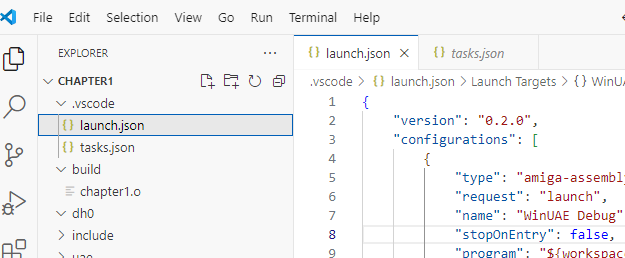
Immagine che contiene testo, schermata, software, numero

Descrizione generata automaticamente

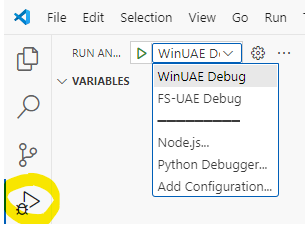
## WinUAE Amiga Emulator

To run the sample program in assembly, we use an Amiga emulator, called WinUAE and integrated into the Amiga Assembly extension for VS Code.

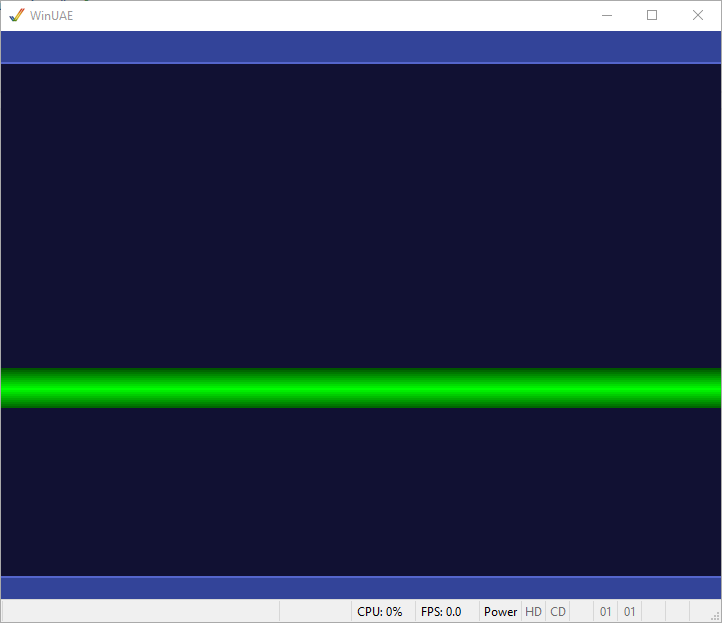
From the VS Code Explorer, open the "launch.json" file (see picture below). Let's change the value of the "stopOnEntry" attribute, putting it to "false". In this way we will prevent the program from stopping on the first statement when we execute it.



Now let's see how to run the program. By pressing the yellow circled button in the following figure, the "Run and Debug" panel is displayed. From the drop-down menu at the top, we can choose the debug configuration between WinUAE and FS-UAE (another emulator). We choose WinUAE and press the triangle-shaped key to activate debugging.



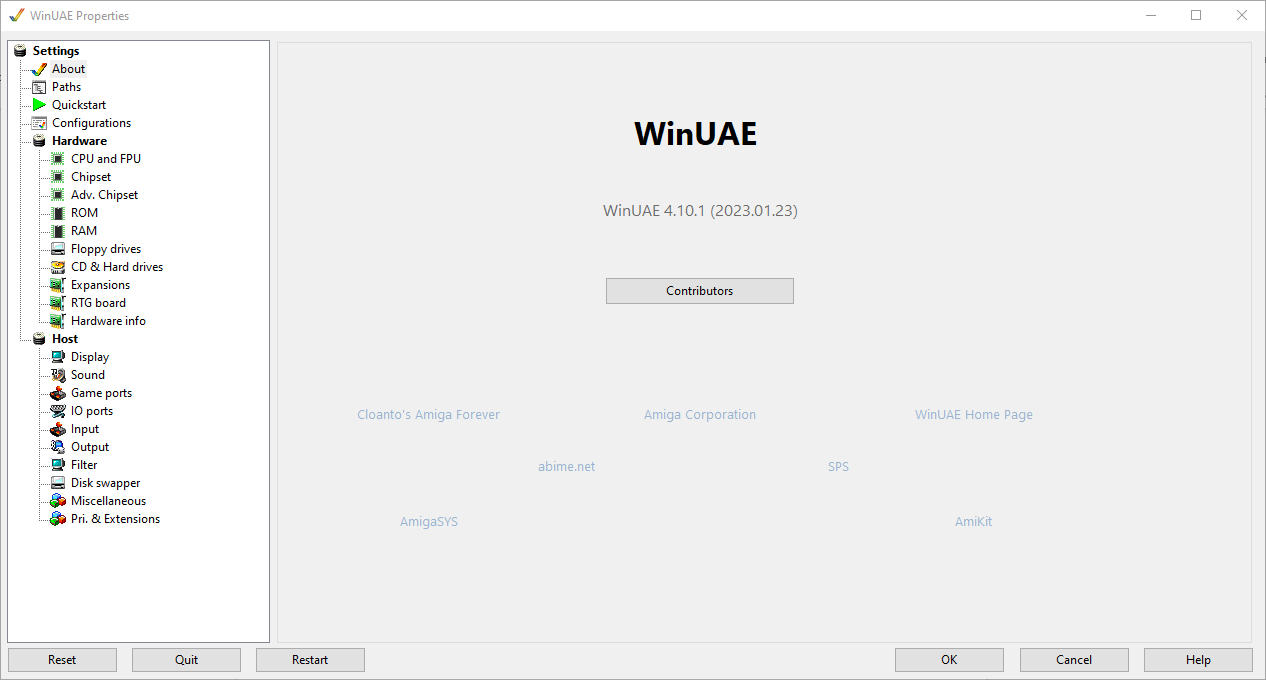
The emulator window will open, which will run the sample program, as in the following figure. The sample program shows a horizontal line moving up and down. If we press left mouse button, the program terminates and exits to Amiga OS.



## WinUAE Configuration

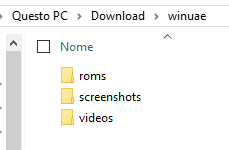
Let's see how to configure WinUAE to emulate an Amiga 500, an Amiga 1200 and an Amiga 4000.

Pressing F12 when WinUAE is open will take you to the properties screen, shown in the following figure.

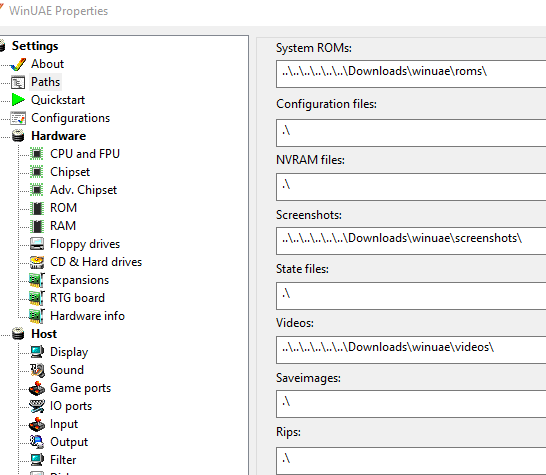


First of all, we should create a “winuae” folder in our downloads folder and create the following subfolders inside it (see following figure):

* roms
* screenshots
* videos



Now we can configure the file paths in WinUAE. We press the "Paths" item in the tree menu on the left of the WinUAE properties window. The screen shown in the following figure will appear.



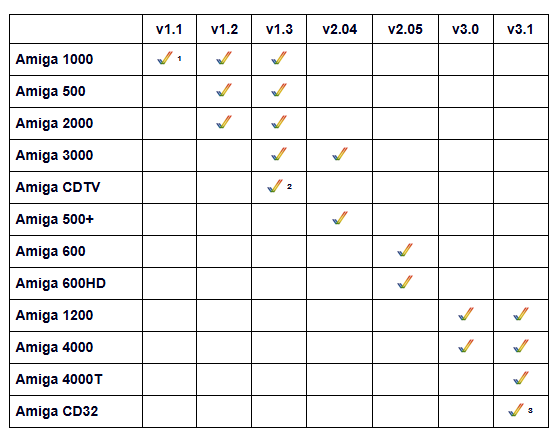
Let's change the paths of System ROMs, Screenshots and Videos so that they point to the 3 folders created earlier.

By default, the emulator uses a compatible AROS ROM. We want as much fidelity as possible, so we need to get the original ROMs.

The original ROMs, called Kickstart, can be purchased from the following site:

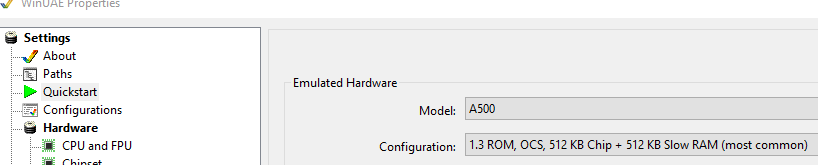
<https://www.amigaroms.com/>

The following figure shows a table with the Amiga models and compatible ROMs.

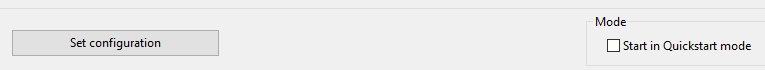


For our purposes we need Kickstart 1.3 for A500 emulation and Kickstart 3.1 for A1200 and A4000 emulation. Googling is not difficult to find files, although it is not legal. Once we have the files, we need to copy them to the “winuae/roms” folder.

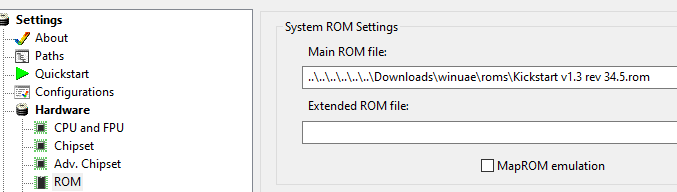
Now let's create the configuration for the Amiga 500. By pressing the "Quickstart" button from the tree menu, the screen in the following figure appears. We need to select A500 as the model.



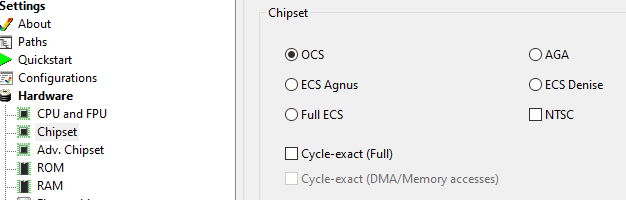
In addition (see next figure), we must uncheck the "Start in Quickstart mode" checkbox and press the "Set Configuration" button. A warning message will appear because the ROMs were not found. Press OK.



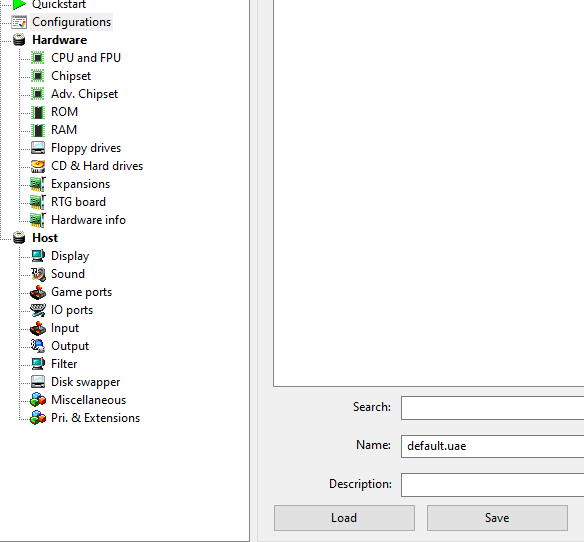
Now select the item "Hardware\ROM" from the tree menu. The screen in the following figure will appear. As the main ROM file we select the Kickstart 1.3 file that we got earlier.



Now select the item "Hardware\Chipset" and the window in the following figure will appear. We need to select the "Cycle-exact" checkbox to get the most faithful simulation of the Amiga chipset.

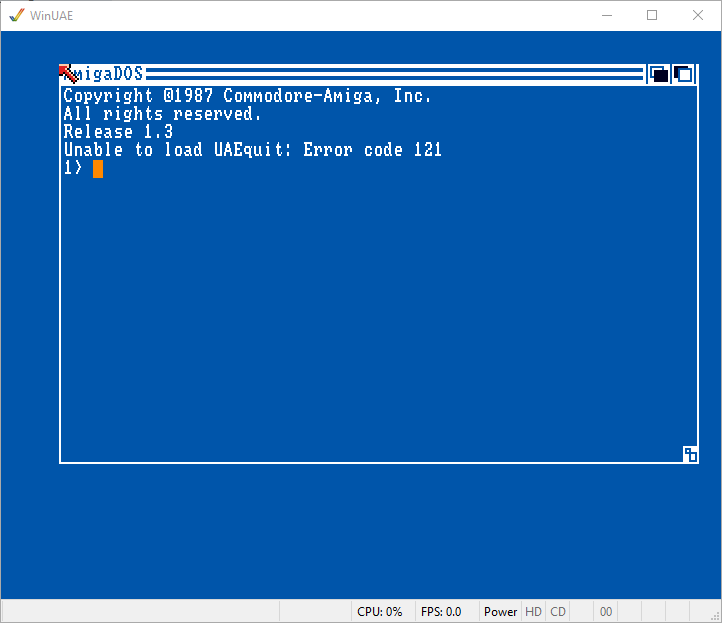


To save the configuration and make it the default, we need to select the "Configurations" item from the tree menu. In the window that appears (see following figure), we must press the "Save" button.



Now the default configuration of WinUAE will be the emulation of the Amiga 500. Press OK to close the properties window.

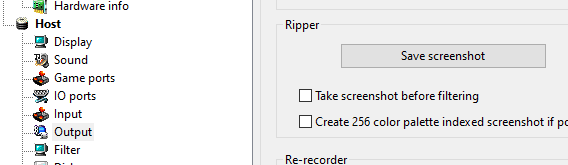
If you close the emulator now and try to run it again, you will see that it will be launched in A500 mode. Press the left mouse button to exit the program. You will see the window in the following, proving that WinUAE is emulating an Amiga 500. Now you can close the window, ignoring the error. In fact, the program that closes the emulator “UAEquit”, added by the Amiga Assembly extension, is not compatible with the A500. You have to close the emulator manually by pressing the close button of the window.



If you want to configure an Amiga 1200 or 4000, repeat the same steps, selecting the desired model. With the latter you must specify Kickstart 3.1 as ROM.

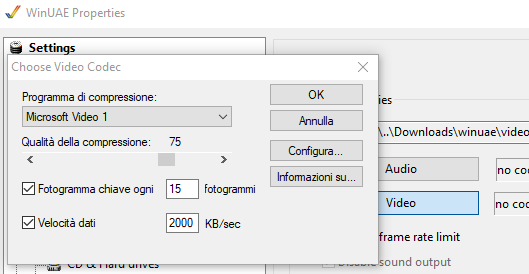
## Taking Screenshots

WinUAE allows you to take screenshots or record videos. Simply select the "Output" item from the tree menu. On the screen that appears (see following figure), press the "Save screenshot" button. The current screen of the emulator will be saved, in png format, in the screenshots folder that we created earlier.

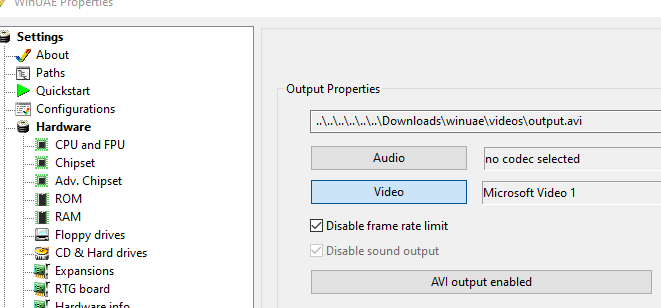


## Screen recording

To record a video with WinUAE output, again from the Output screen, configure the video format by pressing the "Video" button. In the dialog box that appears "Choose Video Codec" (see following figure), select "Microsoft Video 1" as the codec and set the data rate to 2000 KB/sec. Press OK.

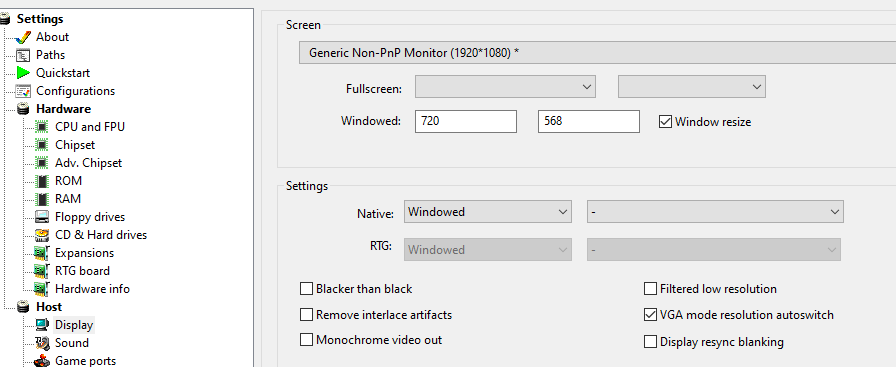


To start video recording, press the "AVI Output enabled" button (see following figure). To stop recording, press this button again. When you stop recording, you will find a .avi file in the videos folder that we created earlier.



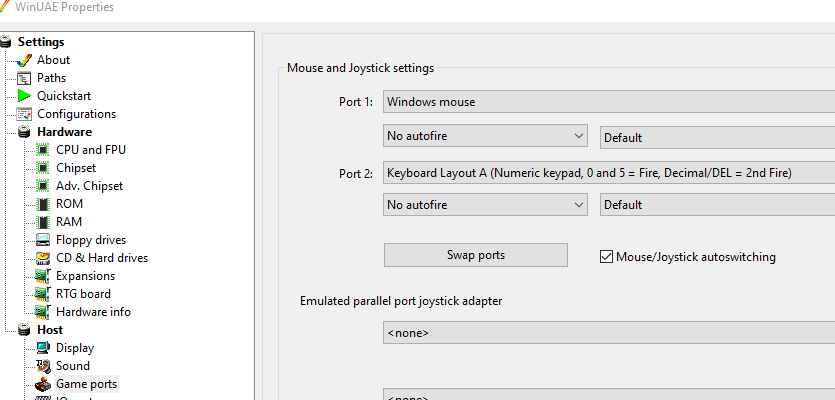
## Window size and video mode

By selecting the "Display" item from the WinUAE Settings tree menu, the screen in the following figure appears. You can change the size of the window, which by default is 720 x 568 pixels. It is possible to activate the fullscreen mode.



## Joystick configuration

By selecting the "Game ports" item in the Settings menu, the window in the following figure appears. In this window you can select the configuration of joysticks and mouse connected to the two ports available on the Amiga.

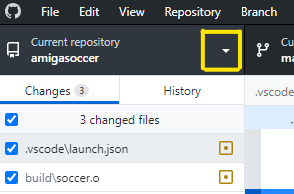


## Source code versioning with Git

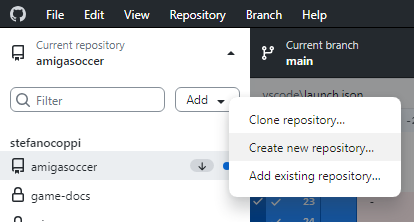
During the development of our video game, it is useful to keep track of the changes we make to the source code. To do this we will use a software called Github Desktop, which can be downloaded for free from the following url:

<https://desktop.github.com/download/>

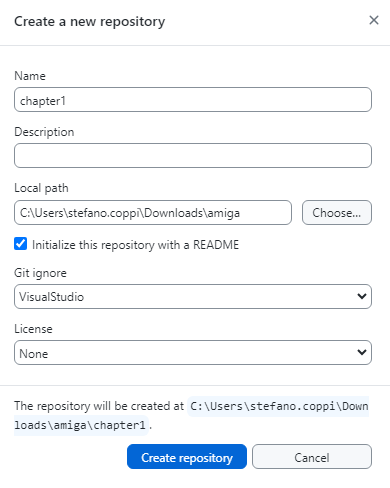
Let's see how to create a new source code repository for our video game. After launching Github Desktop, we press the button highlighted in yellow in the following figure:



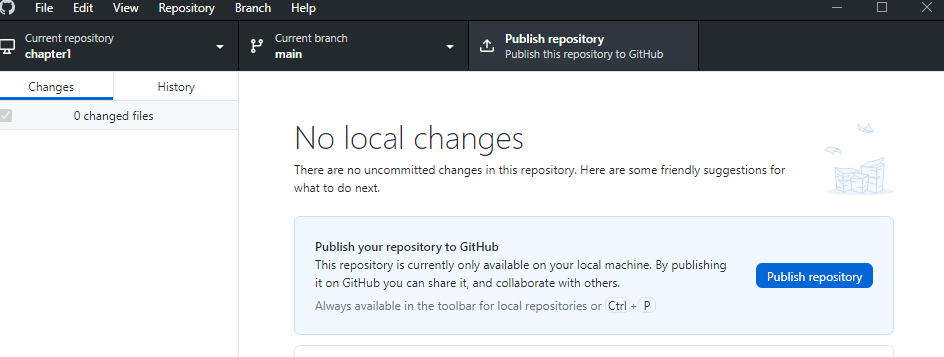
The window in the following figure will appear, in which we must select the item "Create new repository" from the "Add" menu.



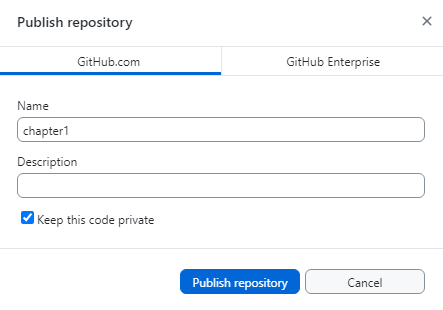
The window in the next figure will appear, in which we can set the name of the repository, a description, the folder that contains the files, if we want to add a .gitignore file that specifies the files to be ignored in version control. By pressing the "Create repository" button, the repository will be created locally to your computer.



After the new repository is created, you will see the screen in the following figure:

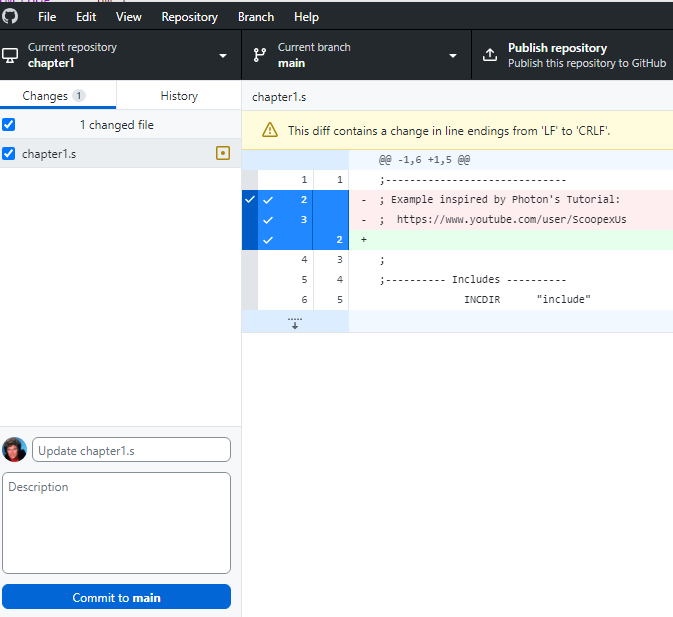


By pressing the "Publish repository" button, the screen in the following figure will appear to publish the repository to Github.



By pressing the "Publish repository" button, it will be published to the Github servers. You must have a Github account and you will be asked to log in. If you don't have the account, you need to create it first.

If you now try to edit the "chapter1.s" file, you will get the situation shown in the following figure. The left pane shows the files that have changed (in this case, chapter1.s). The middle pane shows the differences between the two versions of the files. In the box at the bottom left you can enter a description of the changes (in this case the text "Update chapter1.s" is suggested, but you can change it). Pressing the "Commit to main" button will record the changes in the repository. After a commit, you can publish your changes to the remote repository by pressing the "Publish repository" button.



By pressing the "History" button, the screen in the following figure is displayed, showing the history of all commits and also the differences, if you select a file.

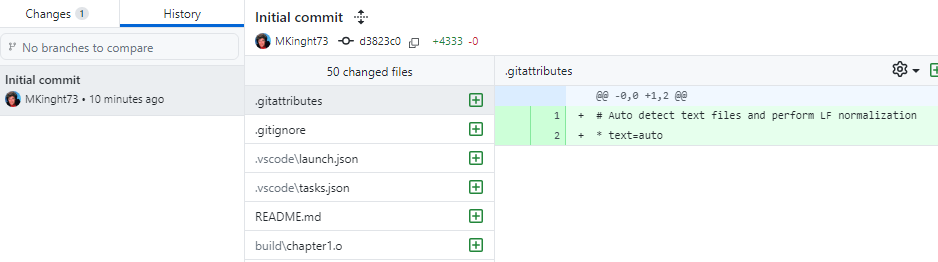
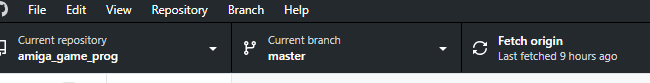


Figure 29

Let's say another developer has made changes and posted them to the remote repository. To update your local repository, simply select the "Fetch origin" item from the main screen (see next figure).



## Online resources on Amiga programming

A good reference for the Motorola 68000 Processor Assembly is given by the following document:

<https://www.nxp.com/docs/en/reference-manual/M68000PRM.pdf>

The official Amiga hardware manual is available online at the following url:

<http://amigadev.elowar.com/read/ADCD_2.1/Hardware_Manual_guide/node0000.html>

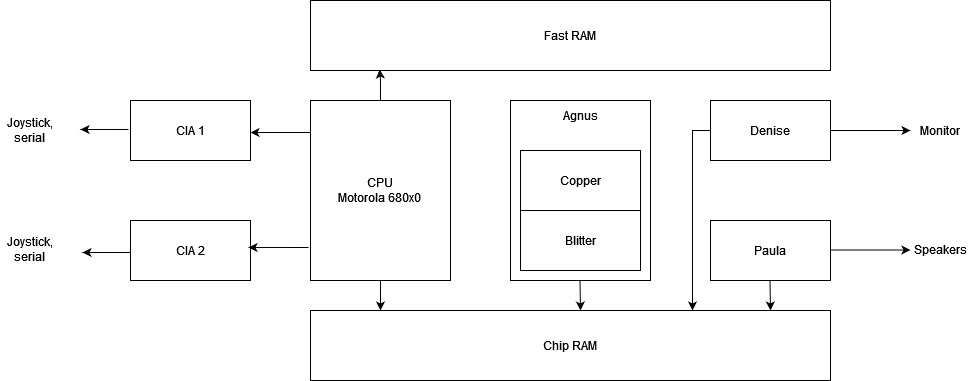
An online copy of Amiga Developer Documentation is available at the following url:

<http://amigadev.elowar.com/>

# 2. Amiga hardware overview

## Amiga hardware architecture

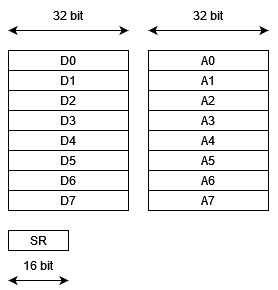
Since we want to program directly the Amiga hardware, we need to understand its architecture, which is shown in the following figure.



## CPU

The CPU is based on Motorola 680x0 family. The Amiga 500,1000 and 2000 were equipped with a 68000 running at about 7 Mhz. The Amiga 1200 was equipped with a 68EC020 at 14 Mhz. The Amiga 4000 was equipped with a 68030 or 68040 at 25 Mhz. The 68000 CPU has 32 bit registers, while the data bus and the ALU (Arithmetic Logic Unit) are 16 bits wide. The 68020,030 and 040 are full 32 bits CPUs with 32 bit data bus and ALU.

All the 680x0 CPUs have 8 data registers, 32 bit wide, called D0 - D7. These registers can also be used as 16 or 8 bit registers. 680x0 cpu also have 8 address registers, 32 bit wide, called A0 - A7. A7 is used as stack pointer and should not be changed directly. The following picture shows the registers:



There also is a status register SR, 16 bits wide. The bottom 8 bits are known as Condition Code Register (CCR), whose bottom 5 bits are flags that are set/cleared depending on the outcome of an instruction. The following table shows the meaning of the CCR flags:

|  |  |  |  |
| --- | --- | --- | --- |
| Bit pos. | Flag | Name | Function |
| 0 | C | Carry | This flag is set when the result of an instruction creates a carry from the most significant bit |
| 1 | V | Overflow | This flag indicates that there was an arithmetic overflow, which means that the result was either too large or too small for the destination |
| 2 | Z | Zero | This flag will be set when the result of the operation is zero |
| 3 | N | Negative | This flag will be set when the result of the operation is negative |
| 4 | X | Extend | Holds the carry for calculations with numbers larger than 32 bits |

## RAM memory

Amiga was equipped with two types of RAM memory:

FAST Ram: could be accessed only by the CPU

CHIP Ram: could be accessed by the custom chips or the CPU

For our purposes, the Chip RAM is very important, because into it we will store all the graphics and sound assets used by our game. We could store the program into Fast RAM, but only a few models had this type of memory.

The following table shows the amount of RAM memory for various Amiga models:

|  |  |  |
| --- | --- | --- |
| Model | Chip RAM | Fast RAM |
| A1000 | 256KB |  |
| A500 | 512KB |  |
| A2000 | 512KB | 512KB |
| A600 | 1 MB |  |
| A3000 | 1 MB | 1 MB |
| A1200 | 2 MB |  |
| A4000 | 2 MB | 2-4 MB |

## Custom Chips

The strength of Amiga architecture were the custom chips, designed by the legendary Jay Miner. The word "custom" means that the chips were designed specifically for Amiga and were not commercial off-the-shelf parts. There are 3 versions of this custom chipset:

* OCS : is the Original ChipSet available on A1000,A500,A2000. It was composed by 3 chips, named Agnus, Denise and Paula.
* ECS : Enhanced ChipSet available on A600,A3000. It was composed by 3 chips, with the same names of OCS.
* AGA : Advanced Graphics Architecture available on A1200,A4000,CD32. It was composed by 3 chips, named Alice,Lisa,Paula.

## OCS features

The chip Agnus contained the DMA controller, a coprocessor called "Copper" used to manage the screen configurations and special graphic effects, a coprocessor called "Blitter" used to transfer memory blocks.

Denise provided the interface with the monitor. It had a palette of 4096 colors. Various video modes:

* low-resolution 320x256 (200 for NTSC), with a maximum of 32 colors
* high resolution 640x256 (200 for NTSC), with a maximum of 32 colors
* interlaced video modes 320x512 (400 for NTSC) or 640x512 (400 for NTSC)
* HAM mode capable of displaying all the 4096 colors
* hardware horizontal and vertical scrolling
* 8 hardware sprites

The chip Paula provided four digital audio channels, each of them could play 8 bit samples with a maximum frequency of about 28Khz. Two CIA chips provided the input/output interfaces like joystick, serial, parallel ports.

## ECS Features

This chipset added the new following features:

* video mode VGA compatible (640x480)
* super-hires mode 1280x256

## AGA Features

This chipset added the following new features:

* palette with 16.8 million colors
* increased the number of colors of lowres and hires video modes to 256
* increased width of hardware sprites to 64 px

# 3. Motorola 68K Assembly Short Course

In this chapter we will try to explain the Assembly language programming of the Motorola 68000 Amiga processor. We'll explain the most frequently used instructions in video game programming, not the entire instruction set exhaustively.

A microprocessor is only capable of executing a limited set of instructions. Different microprocessors have different instruction sets. Instructions are nothing more than a sequence of bytes, often called "machine code". You will understand that it is inconvenient for a human being to write a program using machine code directly, because the sequence of bytes would be difficult to understand. For these reasons, the Assembly language was invented, which replaces machine code with alphanumeric symbols, maintaining a 1:1 correspondence between machine instructions and assembly instructions. So for example, instead of byte $7a we will have the assembly statement "ADD".

It is important to distinguish between the term "assembly" which is the name of the language and the term "assembler" which is the software that compiles a program into assembly into machine code.

## Assembler directives

In addition to instructions for the microprocessor, there are instructions for the assembler to tell the it how to create the assembly program. These instructions are called **DIRECTIVES.**

An assembly program must begin with the **SECTION directive**, which has the following syntax:

**SECTION** <name>,<type>

<name> name useful for identifying the segment

<type> type of memory, can take the following values:

CODE code segment in FAST RAM memory or CHIP in case of absence of FAST

CODE\_C code segment in CHIP RAM memory

DATA data segment in FAST RAM memory

DATA\_C data segment in memory CHIP RAM

BSS zeroed data segment in FAST RAM

BSS\_C zeroed data segment in CHIP RAM

This directive instructs the assembler to allocate all instructions that will follow to a portion of memory of the type specified by the <type> parameter.

An assembly program must end with the **END** directive.

## Labels

A generic line of an Assembly program has the following format:

<label> <instruction>.<size> <operands> <comment>

The label can be made up of letters, numbers and the symbol ".". To allow the use of the dot in labels, you need to add the "-ldots" option to the args in the tasks.json file, as shown in the following figure.

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

<size> indicates the size of the operands and can be as follows:

* b – byte, 8 bit
* w – word, 16 bit
* l – long, 32 bit

## Comments

To create a comment in Assembly, just use the ";" character. Everything that follows this character is a comment. It is a good idea to start the source code with a header comment, with the title and description of the program, information about the author and the target platform etc... An example is as follows:

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; Amiga Assembly Game Programming Book

;

; Chapter 3 - Motorola 68K Assembly Short Course

;

; (c) 2024 Stefano Coppi

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Now let's start to see the instructions of the Assembly language of the Motorola 68000 processor.

## Move instruction

Making an analogy with a high-level language, such as the C language, one of the simplest instructions is assignment, which is when we assign a constant value to a variable:

score = 255

The corresponding Assembly statement is MOVE, which is followed by two operands, called source and destination. This statement copies the value of the source operand to the destination.

Assuming that the score variable is stored in the d0 register, the assignment seen above results in:

move.b  #255,d0        ; copies byte 255 into least significant byte (lsb) of

; register d0

In this case, the value 255 is 1 byte in size. This form of MOVE uses what is called **immediate addressing** in jargon.

It is also possible to express numerical values in hexadecimal notation, by prefixing the $ symbol to the number:

move.w  #$1234,d0 ; copies word $1234 into least significant word (lsw) of d0

In this case, the variable in d0 will be assigned the hexadecimal value $1234, which is the size of a word.

Similarly, we can assign a value of long size to 32 bits by writing:

move.l  #$12345678,d0 ; copies a long word into d0

Now let's see how to declare a variable in Assembly. A variable is nothing more than a space in memory, capable of containing a value of a given size. This memory space is referenced by a symbolic name. In Assembly, we use the **dc** directivethat reserves a space in memory and initializes it with the constant value that follows.

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; VARIABLES

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

value1  dc.w    0  ; declares a variable, with a length of one word

It is possible to declare variables one byte long with dc.b or 32 bits with dc.l

Now let's see how to assign a value to a variable, using the MOVE statement and absolute addressing:

move.w  #$3456,value1 ; assigns value $3456 to the value1 variable

Another way of addressing is indirect. We use the Load Entire Address (LEA) statement to load the address of the variable into the address register a0. To assign a value to the variable, we use the notation (a0) which indicates the variable whose address is contained in a0.

; indirect addressing

  lea       value1,a0      ; loads address of value1 into a0

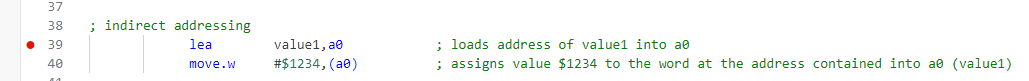
  move.w    #$1234,(a0)    ; assigns value $1234 to the word at the address

; contained into a0 (value1)

For those who know the C language, a0 is a pointer to the variable value1, which is a variable that contains the address of another variable.

## Debugging

To better understand how indirect addressing works, let's try using the VS Code debugger. To stop the execution of the program we must insert a breakpoint in the line where we want it to stop:



We must click in the empty space to the left of the line number. A red dot will appear, indicating the breakpoint.

To start debugging, press the F5 key. VS Code will start the emulator and stop running at the line where we inserted the breakpoint. This line will be highlighted in yellow.

Immagine che contiene testo, schermata, numero, Carattere

Descrizione generata automaticamente

In the left pane we find useful information for debugging. In fact, at the top we find the list of CPU registers, with their values in hexadecimal.

In the box in the center, called "Watch", we can add variables whose value we want to see. To add a variable, you have to press the "+" button next to the word Watch and then enter the name of the variable. Try adding the variable "value1".

In the bottom box, called "Disassembled Memory", we can see the memory location where the instruction is located, the assembly instruction and the translation into machine code, in hexadecimal notation.

For example, referring to the following figure, we can see that at memory location $1ba68 the instruction lea value1,a0 is stored, which in machine code corresponds to the sequence of bytes: 41 fa 00 78.

Immagine che contiene testo, Carattere, schermata, bianco

Descrizione generata automaticamente

At the top of the VS Code window, you will notice a buttons bar (see the following figure) that is used to command the execution of the program in debug.

Immagine che contiene testo, Carattere, logo, schermata

Descrizione generata automaticamente

To execute the program one instruction at a time, you must press the second button, called "Step Over" or the F10 key. If you are on a call to subroutines and want to enter the subroutine, you must press the third button, called "Step Into" or the F11 key. The fourth key is used to exit a subroutine. The fifth "Restart" button is used to restart the execution of the program from the beginning. The sixth key is used to end the execution of the program. The first "Continue" button will continue running the program normally.

After pressing F10 once, you see that the highlighted line moves to the next statement and that the value of register a0 has changed and now contains the address of the variable value1.

Immagine che contiene testo, schermata, software, numero

Descrizione generata automaticamente

If you press F10 again, you will take a step further to verify that the value1 variable takes the value $1234, using the watch box. At this point you can finish the execution of the program, by pressing the stop key or SHIFT + F5.

Immagine che contiene testo, software, Icona del computer, numero

Descrizione generata automaticamente

## Data structures

Now let's look at another type of addressing: **indirect with displacement**. To explain this addressing, let's introduce the concept of **data structure**.

If you know the C language, you know that you can declare a data structure with the **struct** keyword:

struct ship

{

  short int x;

  short int y;

}

The data structure is a container that aggregates a set of variables in an orderly manner, logically related. In the example seen it aggregates the x,y coordinates of the position of our spaceship.

In Assembly, a data structure is declared using the following syntax:

; declares a data structure

             rsreset

ship.x       rs.w       1

ship.y       rs.w       1

ship.length  rs.b       0

The RSRESET directive must always be placed at the beginning of the declaration and resets the counter inside the data offset in the structure. The label indicates the name of the field. The **rs.w** directive reserves the amount of word specified by the following numeric value. In this case 1 word. The last field represents the length of the structure itself, and no memory space is allocated for it, in fact the value is 0. We can also reserve space for a long using rs.l.

After declaring a structure, you need to create an instance of it in memory. This is done using the usual way you declare variables, with the dc directive. In fact, the fields are treated as if they were variable.

; instance of ship structure

ship  dc.w    0    ; ship.x

      dc.w    0    ; ship.y

To initialize the structure instance fields, you must assign the ship instance address to the a0 registry. Use the MOVE statement with indirect addressing with displacement to assign the value 100 to the ship.x field.

Indirect addressing with displacement adds the indicated displacement (ship.x and ship.y in this case) to the address contained in a0. Of course, it is possible to specify the displacement with an integer. Ship.y will be equal to 2 for example.

; indirect addressing with displacement

  lea       ship,a0            ; base address of ship structure instance in a0

  move.w    #100,ship.x(a0)    ; assigns value 100 to ship.x field

  move.w    #192,ship.y(a0)    ; assigns value 192 to ship.y field

There is also indirect addressing with post increment, which differs from the previous one in that after assigning the value $1234 to the address contained in a0, the latter is incremented by 2 to point to the next word.

; indirect addressing with post increment

  lea       value1,a0       ; loads address of value1 into a0

  move.w    #$1234,(a0)+    ; assigns value $1234 to the word at the address

; contained into a0 and then increments a0 of one

; word (2 bytes)

If we want to reset the bits of a register or a variable, we can use the CLR instruction.

  clr.b    d0    ; sets lsb of d0 to zero

  clr.w    d0    ; clears lsw of d0

  clr.l    d0    ; clears all 32 bits of d0

## Arithmetic instructions

Now let's look at the arithmetic statements, starting with the ADD, which sums the values of the source and destination operands. The result is stored in the target operand. Let's see an example below:

  move.w    #10,value1    ; value1 = 10

  move.w    #13,value2    ; value1 = 13

  move.w    value2,d0     ; d0 = value2

  add.w     value1,d0     ; d0 = value1 + value2

  move.w    d0,result     ; result = d0

The SUB instruction subtracts the source operand from the target, putting the result in the target. Below is an example:

  move.w    #100,value1    ; value1 = 100

  move.w    value1,d0      ; d0 = value1

  sub.w     #50,d0         ; d0 = d0 - 50 = value1 - 50

  move.w    d0,result      ; result = d0

The MULU instruction multiplies the 16 bit wide operands and puts the result in the destination, using 32 bits. MULU operates on unsigned numbers. To multiply two signed numbers, you need to use MULS. Below is an example:

  move.w    #30,d0       ; d0 = 30

  mulu      #3,d0        ; d0 = d0 \* 3 = 30 \* 3

  move.w    d0,result    ; result = d0

The DIVU instruction performs a division. The destination operand is divided by the source operand. The destination operand is the size of a long word (32 bits), while the source operand is the size of a word (16 bits). The quotient is inserted in the low word of d0, while the rest is inserted in the high word. Below is an example:

  move.w    #100,d0

  ext.l     d0           ; extends d0 value to a long word, because divu

; destination must be a long

  divu      #30,d0       ; d0 = d0 / 30  lower word = quotient, higher word =

; remainder

  move.w    d0,value1    ; value1 = quotient of 100/30 = 3

  swap      d0           ; swaps the lower and upper words of d0

  move.w    d0,value2    ; value2 = remainder of 100/30 = 10

The DIVU instruction operates on unsigned operands, while the DIVS operates on signed operands.

## Shift instructions

The ASL instruction shifts the target operand to the left, by a number of bits equal to the source operand. From an arithmetic point of view, the shift to the left is equivalent to the multiplication by powers of 2. There is the LSL instruction that operates on unsigned numbers. Below is an example:

  move.w    #20,d0    ; d0 = 20

  asl.w     #2,d0     ; shifts d0 2 bits to the left => d0 = d0 \* 2^2 = d0 \* 4 = 80

The ASR instruction shifts the destination operand to the right. The number of bits to be shifted is equal to the source operand. Arithmetically, the shift to the right is equivalent to the division by powers of 2. There is also the LSR instruction that operates on unsigned numbers, while ASR operates on signed numbers. Below is an example:

  move.w    #20,d0    ; d0 = 20

  asr.w     #1,d0     ; shifts d0 1 bit to the right => d0 = d0 / 2 = 10

## Logical instructions

Now let's see the logical instructions. Let's start with the AND that performs the logical operation “and” on the two operands, storing the result in the destination operand. The most frequent use is the masking operation. Suppose, for example, that you want to extract only the lower 4 bits of a byte. We can use a mask made in the following way: %00001111. By making the and between the byte of which we want to extract the lower 4 bits and the mask, we will get only the low 4 bits, as in the following example:

  move.w    #%10011100,d0    ; d0 = 156

  and.w     #%00001111,d0    ; d0 = 1100 = $c

The OR statement performs an or operation between the two operands, storing the result in the destination operand. An example is as follows:

  move.w    #%11110000,d0    ; d0 = 240 = $f0

  or.w      #%00001111,d0    ; d0 = %11111111 = $ff

The EOR instruction performs the exclusive OR between the two operands, storing the result in the destination operand. We will see later that it will be useful for reading the joystick.

  move.w    #%00001010,d0    ; d0 = %00001010 = $a

  eor.w     #%00001111,d0    ; d0 = %00000101 = $5

Finally, the NOT statement complements one of the only operand. It can be useful to reverse the bits of a word, as in the following example:

  move.w    #%00001111,d0    ; d0 = $000f

  not.w     d0               ; d0 = %1111111111110000 = $fff0

## Conditional statement

Now let's see how to implement the conditional **if** construct, present in high-level languages. Suppose you want to translate the following segment of C code into an assembly:

d0 = 123;

if (d0 >= 100)

    d0 = d1;

else

    d0 = d3;

d0 = 1;

To compare the value of a register or variable to a constant, you use the CMP statement. The comparison is made by subtracting the source operand from the destination operand. The result of this operation is to set the flags of the SR status register. These flags are used as input from the Bxx conditional jump instructions. To translate the segment of code seen earlier into an assembly, we need to use the BGE conditional jump statement, which jumps to the specified address if the result of the previous operation is greater than or equal to zero. Below is the complete example:

  move.w    #123,d0

  cmp.w     #100,d0              ; d0 >= 100?

  bge       .greater\_or\_equal    ; if d0 >= 100 jumps to .greater\_or\_equal

.else:

  move.w    d0,d3                ; else executes this instruction

  bra       .continue

.greater\_or\_equal:

  move.w    d0,d1

.continue:

  move.w    #1,d0

The other conditional jump instructions are as follows:

signed numbers

|  |  |
| --- | --- |
| BGE | branch on greater than or equal |
| BGT | branch on greater than |
| BLE | branch on lower than or equal |
| BLT | branch on less than |

unsigned numbers

|  |  |
| --- | --- |
| BHS | branch on higher than or same |
| BHI | branch on higher than |
| BLS | branch on lower than or same |
| BLO | branch on less than |

## Loops

Now let's see how to translate a loop with a fixed number of iterations, such as the **for** loop of high-level languages, into assembly. The following code snippet shows that loop.

The number of iterations minus one is stored in a data register, d7 in the example. Then you write the instructions to be repeated in a loop. In the example we make a byte-by-byte copy. To repeat the loop, you use the DBRA statement, which requires the register containing the number of iterations and the label that marks the beginning of the loop. This instruction repeats the loop until the register is non-zero.

; cycle with a fixed number of iterations

       moveq     #10-1,d7       ; number of iterations - 1 in d7

.loop  move.b    (a0)+,(a1)+    ; copies a byte from the address contained into

; a0 to the address contained in a1

       dbra      d7,.loop       ; repeats the loop until d7 <> 0

Another typical loop of high-level languages is the **while** loop:

while d0 <= 5

{

    d0 += 1;

}

This results in assemblies as shown in the following code snippet. The BRA instruction makes an **unconditional jump**, that is, the jump always happens, without any conditions. A comparison with the CMP statement and the BGT conditional jump is used to exit the loop.

.while\_loop:

       cmp.w    #5,d0

       bgt      .exit          ; if d0 > 5 exits the loop

       add.w    #1,d0          ; d0 = d0 + 1

       bra      .while\_loop    ; jumps to .while\_loop, repeating the loop

.exit  nop

## Subroutines

When you start writing long programs, it is essential to subdivide the code into **functions**, according to the dictates of structured programming. The C language function, or method, is called a "**subroutine**" in Assembly. Below you will find a subroutine declaration template.

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; Subroutine description.

;

; parameters:

; <register>.<size> - description

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

  movem    d0-a6,-(sp)    ; saves registers from d0 to a6 value into stack

  nop                     ; instructions here

  movem    (sp)+,d0-a6    ; restores registers value from stack

  rts                     ; returns to the instruction after the call

It is important to precede the code of the subroutine itself, with a header comment describing what the subroutine does and the input parameters. For each parameter we must list the register that contains it, its size (b,w,l) and the meaning of the parameter itself. Notice that the subroutine starts with a label that must match its name.

The first statement, MOVEM, saves the current values of the registers on the stack. Then we find the instruction body of the subroutine. At the end of the subroutine, we need to restore the values of the registers previously saved in the stack, again with a MOVEM statement. Finally, we have to return the execution of the program to the instruction following the call, through the RTS instruction.

The call to subroutines is normally made through the BSR statement followed by the label of the subroutine. This statement uses a displacement between the memory address of the subroutine and that of the call. This displacement is automatically calculated by the assembler. You can see it using the "Disassembled memory" box:

0x0001bb7e: bsr.b $6 61 04

0x0001bb80: bsr.b $6 61 02

0x0001bb82: rts 4e 75

0x0001bb84: movem.w d0-a6,-(a7) 48 a7 ff fe

The bytes of the instruction in machine code are 61 indicating the instruction and 04 indicating the operand. So the displacement is 04, in fact the subroutine starts 4 bytes after the instruction following the BSR, at the address 0x0001bb84.

There is also another way to call subroutines, which is with the JSR statement. The difference with BSR lies in the addressing, which in JSR is absolute, i.e. the memory address is indicated. We will see that the JSR will be used for calls to the operating system.

You can find the complete source code at the following url:

<https://github.com/stefanocoppi/amiga_game_prog_assembly/tree/main/chapter3>

# 4. Take control of Amiga hardware

Amiga has a sophisticated operating system (O.S.) that provides the programmer with all the tools to use the graphics and sound capabilities. However, historically, very few games have made use of the O.S. Virtually all commercial games that were developed for the Amiga bypassed the operating system and took direct control of the hardware and were written in assembly language. The reason is simple: in this way it is possible to get the maximum performance from the hardware. Since we want to create a video game in Assembly, programming the Amiga hardware directly, we must take control of the Amiga hardware, stopping the O.S. in a controlled manner. Before starting to write code, we will introduce some theoretical concepts necessary for a full understanding of the code we will write.

## DMA

DMA is an acronym for "Direct Memory Access". We know that both the custom chips and the CPU can access the Amiga chip memory. To increase the level of parallelism of the Amiga, the designers have created channels that allow the custom chips to access the memory directly, without the help of the CPU, leaving it free to do other things. This is one of the strengths of the Amiga architecture. Access to the DMA channels is regulated by a "DMA Controller" present in the Agnus chip. DMA channels can be enabled or disabled via the write-only DMACON register ($dff096). There is a DMACONR read-only register ($dff002) to read the status of these channels. The following table shows the meaning of the bits of these registers:

|  |  |  |
| --- | --- | --- |
| Bit # | Name | Description |
| 15 | SET/CLR | If it is 1 then the bits at 1 indicate enabling, if it is 0 then they indicate disabling |
| 14 | BlitBusy | read-only, 1 indicates that the Blitter is busy |
| 13 | BlitZero |  |
| 12 | X | not used |
| 11 | X | not used |
| 10 | BlitPri | Blitter Priority |
| 9 | Master | General switch for enabling all DMA channels |
| 8 | BPLEN | DMA channel for bitplanes |
| 7 | COPEN | Copper DMA channel |
| 6 | BLTEN | Blitter DMA channel |
| 5 | SPREN | Sprites DMA channel |
| 4 | DSKEN | Disk DMA channel |
| 3 | AUD3EN | DMA channel for voice 3 of the audio |
| 2 | AUD2EN | DMA channel for voice 2 of the audio |
| 1 | AUD1EN | DMA channel for voice 1 of the audio |
| 0 | AUD0EN | DMA channel for voice 0 of the audio |

## Amiga Operating System and libraries

The O.S. of Amiga, called AmigaOS and contained in a ROM memory called Kickstart, is composed of various libraries, which are not all present in memory at the same time, but are loaded dynamically. Its main component is called **Exec** and performs the functions of task scheduler, memory management, interrupt management, inter-process communication through messages, loading of dynamic libraries. At memory location $4, called **ExecBase**, there is the pointer to the library functions. In order to use a dynamic library, it must first be opened using the Exec's OpenLibrary function. This function returns a pointer to the base address of the library, which must be used to call all functions of the library itself. When you stop using a library, you need to close it using the CloseLibrary function.

## Take System

Now let's start writing a routine that takes full control of the Amiga hardware, disabling the O.S. in a controlled manner.

At first, we open the graphics.library and save its base address. This will be useful to us later.

So, let's disable O.S. multitasking first, so as to avoid giving up the use of the CPU to other tasks.

Then we disable the interrupts of the O.S. to prevent our video game from being interrupted.

Load the base address of the CUSTOM chips, $dff000, into register A5. This register **should not be modified** throughout our code. We will use this base address to access the registers of custom chips, through the indirect base + displacement addressing.

We save the state of the DMA channels and then set them according to our needs, so that we only enable the ones we use.

The last instructions are specific to the Amiga equipped with AGA chipsets, i.e. A1200 and A4000, while they have no effect on the others. They are used to make our video game work on the Amiga as well, AGA. I remember you that the game we are developing will target the Amiga 500, equipped with OCS chipset. Below is the complete code of the routine:

take\_system:

  move.l    ExecBase,a6            ; base address of Exec

  lea       gfx\_name,a1            ; OpenLibrary takes 1

; parameter: library name in

; a1

  jsr       OpenLibrary(a6)        ; opens graphics.library

  move.l    d0,gfx\_base            ; saves base address of

; graphics.library in a

; variable

  jsr       Forbid(a6)             ; disables O.S. multitasking

  jsr       Disable(a6)            ; disables O.S. interrupts

  lea       CUSTOM,a5              ; a5 will always contain

; CUSTOM chips base address

; $dff000

  move.w    DMACONR(a5),old\_dma    ; saves state of DMA channels

; in a variable

  move.w    #$7fff,DMACON(a5)      ; disables all DMA channels

  move.w    #DMASET,DMACON(a5)     ; sets only dma channels that

; we will use

  ; disables AGA features. only needed on A1200,A4000

  move.w    #0,$1fc(a5)            ; sets 16 bit FMODE

  move.w    #$c00,$106(a5)         ; disables 24 bit palette

  move.w    #$11,$10c(a5)          ; enables normal palette

  rts

The variables used by this routine are as follows:

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; VARIABLES

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; string containing the name of graphics.library

gfx\_name  dc.b    "graphics.library",0,0

; base address of graphics.library

gfx\_base  dc.l    0

; saved state of DMACON

old\_dma   dc.w    0

At the beginning of our source code, we must insert the following two directives, which allow the inclusion of some constants useful for accessing the hardware registers.

          incdir     "include"

          include    "hw.i"

Immediately after files includes, we need to insert the declaration of the following constants:

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; CONSTANTS

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; O.S. subroutines

ExecBase     equ $4

Disable      equ -$78

Forbid       equ -132

Enable       equ -$7e

Permit       equ -138

OpenLibrary  equ -$198

CloseLibrary equ -$19e

CIAAPRA      equ $bfe001

; DMACON register settings

                 ;5432109876543210

DMASET       equ %1000001000000000

## Release System

Now we have to see how to release control of the hardware to the O.S., so that at the end of our program, we can return to the Amiga O.S. without problems.

First we need to restore the state of the DMA channels saved in the old\_dma variable. Before copying the value to the DMACON register, we need to set bit 15 to 1. To set one bit to 1, leaving the others unchanged, just make the OR with a mask having all the bits at zero and only the bit to be set to one. In this case, the value of the form is %1000000000000000000 or $8000.

Now we need to re-enable O.S. multitasking and interrupts, using the Permit and Enable routines of the Exec.

Finally, we close the graphics.library, using the CloseLibrary routine of the Exec.

At this point the program can terminate, giving control back to the Amiga O.S. Below is the code of the procedure:

release\_system:

or.w       #$8000,old\_dma     ; sets bit 15

  move.w     old\_dma,DMACON(a5) ; restores saved DMA state

  move.l     ExecBase,a6        ; base address of Exec

  jsr        Permit(a6)         ; enables O.S. multitasking

  jsr        Enable(a6)         ; enables O.S. interrupts

  move.l     gfx\_base,a1        ; base address of

; graphics.library

  jsr        CloseLibrary(a6)   ; closes graphics.library

  rts

## Main program

The two routines created, take\_system and release\_system, must be called at the beginning and end of the main program, which we have not yet created.

Below is the main code:

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; MAIN PROGRAM

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

main:

nop

  nop

  bsr    take\_system     ; takes the control of Amiga's hardware

mainloop:

 btst   #6,CIAAPRA     ; left mouse button pressed?

  bne.s mainloop       ; if not, repeats the loop

  bsr    release\_system ; releases the hw control to the O.S.

  rts

The two NOP instructions at the beginning, are used to make the debug program stop on the "bsr take\_system" line. NOP is a null instruction, which does nothing.

## Main loop

Our video game will run in a loop, which we have called "mainloop". For now, this loop does not contain any instructions. We will add them later. To exit the loop, we control pressing the left mouse button. When you press this key, bit 6 of the CIAAPRA register becomes 0 and then you exit the loop.

## Run the program

If you try to compile and run the program, pressing the F5 key will bring up a completely black emulator screen. Since we haven't drawn anything on the screen, it is colored with the background color being black by default. Pressing the left mouse button will terminate the program and close the emulator. It's not a very exciting result, but the fact that the program closes without blocking the Amiga is already a good result, because it means that we can return to the O.S. in a correct way.

The source code created in this chapter will constitute the skeleton on which we will build our video game step by step, enriching it chapter after chapter. In the next chapters we will start to receive more interesting visual feedback than a black screen.

Immagine che contiene schermata, schermo, Rettangolo, portafotografie

Descrizione generata automaticamente

You can find the complete source code at the following url:

<https://github.com/stefanocoppi/amiga_game_prog_assembly/tree/main/chapter4>

# 5. Game concept

In questo capitolo ci concentreremo sull’ideazione del nostro videogioco.

**Generi di videogioco**

La prima cosa da fare per partire nell’ideazione di un videogioco è scegliere il genere di appartenenza. Durante il corso della storia dei videogiochi sono stati inventati molti generi. Alcuni dei generi più popolari sono:

* Shoot’em up: lo scopo è sparare ai nemici, per guadagnare punti ed avanzare di livello. Il primo è stato Space Invaders, nel lontano 1978.
* Platform: lo scopo è quello di attraversare livelli fatti da piattaforme, saltando da una all’altra, evitando i nemici e raccogliendo oggetti. Il più famoso è Super Mario Bros della Nintendo.
* Racing Games: lo scopo è guidare auto o moto, in sfide contro il tempo o contro gli avversari.
* Sport Games: simulano vari sport quali calcio, tennis, basket.
* Fighting Games (Beat’em up): lo scopo è quello di sfidare gli avversari in vari tipi di lotta: karate, judo, pugilato etc…
* Role Play Games: si interpreta un ruolo, cercando di far evolvere il proprio personaggio. Di solito hanno ambientazioni fantasy.

**Creiamo uno shoot’em up**

Per lo scopo principale di questo libro, che è quello di insegnare la programmazione di un videogioco per Amiga in linguaggio Assembly, ho scelto come genere quello degli shoot’em up. Questo perché è un tipo di videogioco che non richiede logiche complesse e neanche una fisica realistica. Un platform ad esempio richiede una fisica per il controllo dei salti, logiche per rilasciare ed usare oggetti etc… Un videogioco sportivo, ad esempio di calcio, richiede l’implementazione di tutte le logiche che regolano una partita di calcio. Uno shoot’em up richiede solo una logica per il movimento dei nemici e per la gestione di eventuali potenziamenti.

**Game Design**

Come facciamo a descrivere cosa deve fare il nostro videogioco?

Innanzitutto, dobbiamo definire qual è lo scopo del gioco. Nel nostro caso lo scopo ultimo è quello di difendere la Terra dagli attacchi dei malvagi invasori alieni.

Chi o cosa impersona il giocatore umano? Guidiamo una astronave, chiamata SX-9.

Dove è ambientato il gioco? È ambientato nella base che gli alieni hanno costruito nello spazio, in orbita intorno alla Luna. L’inquadratura è laterale e l’astronave volerà verso destra, sorvolando la base. Per il gioco didattico del libro, avremo 1 solo livello.

Come si completa un livello? Dobbiamo sconfiggere un’astronave che si trova a fine livello, posta a guardia di quel settore della base aliena. Serviranno molti colpi per distruggerla.

Come viene assegnato il punteggio? Ogni nemico distrutto assegna un punteggio differente.

Quando termina una partita? Il giocatore dispone inizialmente di 3 vite. Quando le perde tutte, la partita termina, game over!

Come faccio a perdere una vita? Quando la mia astronave tocca le strutture della base oppure un nemico o viene colpita da un colpo nemico.

Posso potenziare la mia astronave? Si, raccogliendo dei particolari oggetti, definiti “power-up”, posso potenziare le mie armi.

Come ottengo i power-up? Uccidendo un tipo particolare di nemico, che trasporta armi.

Perché dovrei scegliere questo gioco rispetto ad altri dello stesso genere? Perché è l’unico in cui posso usare un’arma molto potente, il “Beam”. Perché è l’unico in cui posso essere aiutato da un’astronave satellite. Perché consente il potenziamento delle armi della mia astronave.

Come funziona il “Beam”? Nella parte bassa dello schermo c’è una barra che indica il caricamento di questa arma. Si carica tenendo premuto il tasto di fuoco. Arrivato al termine del caricamento, quando la barra è completamente piena, rilasciando il tasto di fuoco si lancia questa arma. Bisogna prestare attenzione a non essere colpiti durante il caricamento.

Come posso essere aiutato dal satellite? Uccidendo un particolare nemico, posso ottenere un power up che attiva il satellite. Quest’ultimo viene agganciato davanti alla mia astronave ed è indistruttibile. Il satellite può distruggere i nemici toccandoli.

Elenco dei nemici, con nome, armi, colpi necessari a distruggerlo, punti

Elenco power up

Elenco armi

Pannello informativo

Schermata dei titoli

Sound design

**Technical game design**

Adesso affrontiamo il gioco dal punto di vista tecnico.

Per quale piattaforma vogliamo sviluppare il nostro videogioco? In questo caso abbiamo scelto il Commodore Amiga. Il modello minimo sarà l’A1200, in quanto vogliamo usare la grafica avanzata consentita dal chipset AGA. Sarà compatibile anche con l’A4000.

Che modalità video useremo? Useremo la modalità lowres 320 x 256 pixels.

Quanti colori? Useremo da 64 a 256 colori.

Come facciamo a memorizzare lo sfondo del livello? Dato che vogliamo uno sfondo largo diverse schermate (fino a 20), non possiamo memorizzarlo come un’unica immagine perché occuperebbe troppa memoria. Ad esempio una schermata 320x256 a 256 colori occupa circa 81.920 bytes. Se moltiplichiamo questo valore per 20, otteniamo 1.638.400 bytes ovvero circa 1.6 MB. Considerando che l’A1200 ha in totale 2MB di memoria RAM, capite come l’occupazione dello sfondo è eccessiva.

La soluzione è quella di usare una tilemap, in cui la mappa viene costruita a partire da un insieme di tile o blocchi di dimensione fissa, come se fosse un puzzle. In questo modo si deve memorizzare soltanto l’immagine contenente i tiles e poi la matrice che contiene l’indice dei tiles.

Possiamo usare gli sprite hardware? Dato che gli sprite hardware di Amiga sono 8 per riga di schermo, ed inoltre affinchè siano a 16 colori, dobbiamo sovrapporre due sprite, è difficile pensare di poter gestire i nemici con soli 4 sprite per riga.

Invece potremmo usare gli sprite hardware per la nostra astronave.

Per i nemici, i bullets e gli altri oggetti useremo i Blitter Objects.

Come effettuiamo lo scorrimento dello schermo? Dato lo scopo didattico del libro, useremo 2 tecniche e poi confronteremo i risultati:

1. Scrolling hardware
2. Scrolling usando il Blitter

Come controlliamo il gioco? Dato che si tratta di uno shoot’em up, il metodo di controllo ideale è il joystick. Non ha senso comandare l’astronave con il mouse o la tastiera. Quanti pulsanti del joystick usiamo? Sempre per mostrare come usare gamepad a più pulsanti, useremo 2 pulsanti.

Che tipo di audio usiamo? Useremo degli effetti sonori campionati.

Inoltre useremo una musica, in formato ProTracker, per la schermata dei titoli.

Abbiamo bisogno di visualizzare del testo? Si, per il punteggio, la schermata dei titoli e può essere utile per visualizzare delle informazioni di debug.

User Stories

Acceptance criteria

Epics

Definiamo le User Stories per il nostro videogioco

Agile Project Management con Github

Sprint

Roadmap

Release

# 6. The Copper and copperlist

The **Copper** is a coprocessor that is used to control the Amiga's graphics subsystem. It can modify the graphics processor registers even at each scan line of the screen. In this way it is possible to mix multiple video modes on the same screen and display more colors than the video mode allows.

Copper fetches instructions from RAM via DMA channels and can therefore operate independently from the CPU. Below we will look at the two most used Copper instructions. All Copper instructions consist of two consecutive words.

## Move Instruction

This instruction moves a constant numeric value into a custom chip register.

The Move instruction has the following format:

|  |  |  |  |
| --- | --- | --- | --- |
| First word | | Second word | |
| Bit.pos. | **Description** | **Bit.pos.** | **Description** |
| 0 | Always set to 0 | 0-15 | Value to be inserted into the register |
| 1-8 | address of the destination register |  |  |
| 9-15 | Unused, always set to 0 |  |  |

An example of a MOVE instruction is the following, which moves the value $fff to register $180, COLOR0:

dc.w $0180,$0fff

Basically, it sets the first color of the palette to white.

## Wait Instruction

This instruction causes Copper to wait until the electron beam reaches a given position.

The Wait instruction has the following format:

|  |  |  |  |
| --- | --- | --- | --- |
| First word | | Second word | |
| Bit.pos. | **Description** | **Bit.pos.** | **Description** |
| 0 | always set to 1 | 0 | always set to 0 |
| 1-7 | horizontal position | 1-7 | mask used for enabling horizontal comparison |
| 8-15 | vertical position | 8-14 | mask used for enabling vertical comparison |
|  |  | 15 | generally set to 1 |

An example is the following, which waits the line $40:

dc.w $4001,$fffe

The following statement is used to terminate a Copper program:

dc.w $ffff,$fffe

It tells Copper to wait for a position that doesn't exist, so Copper waits until the next frame. In fact, the horizontal position $FF is not reachable from the electron beam.

## The Copperlist

A set of instructions for Copper is commonly called a "**Copperlist**". After writing a copperlist, you need to tell Copper its address. The COP1LC ($dff080) register tells Copper the address of the copperlist. There is also a second COP2LC ($dff084) register that allows you to use a second copperlist, but we will not use it.

After loading the address of the copperlist into the COP1LC register, it is necessary to tell Copper to place its program counter at the beginning of the copperlist. This is done by writing any value into the COPJMP1 register ($dff088). In fact, this is a strobe register. In this way, at each frame, Copper will execute the copperlist.

Now let's try to write a very simple copperlist example, which colors the background half blue and half black.

We start with a MOVE statement that loads blue into the COLOR0 register, which is used for the background. Then we need to use a WAIT statement to wait for line 192. At this point we use another MOVE to change the COLOR0 color of the background to black. We end the program with the WAIT statement to terminate.

The complete copperlist is as follows:

; segment loaded in CHIP RAM

  SECTION    graphics\_data,DATA\_C

copperlist:

  ; BPLCON0 lowres video mode

  dc.w       $100,$0200

  ; puts blue value into COLOR0 register

  dc.w       $0180,$000f

  ; WAIT line 192 ($c0)

  dc.w       $c001,$fffe

  ; puts black value into COLOR0 register

  dc.w       $0180,$0000

  ; end of copperlist

  dc.w       $ffff,$fffe

Note that the first instruction enables the low resolution 320x256 video mode. This will be explained later.

The copperlist must be stored in the Amiga's CHIP memory, the only one Copper can access. To do this, you need to use the SECTION directive.

In this case we will use the following syntax to allocate a chip memory segment for the copperlist:

SECTION    graphics\_data,DATA\_C

This directive should be placed before the copperlist.

Now let's see the code needed to tell Copper where the copperlist is located and to position it at its beginning:

  ; sets our copperlist address into Copper

  move.l     #copperlist,COP1LC(a5)

  ; reset Copper PC to the beginning of our copperlist

  move.w     d0,COPJMP1(a5)

This code snipped must be placed at the end of take\_system routine.

## Save and restore the system copperlist

Before modifying the COP1LC register, we need to save the address of the system copperlist, so that it can be restored when our program exits.

To get the address of the system copperlist, you just need to know the base address of graphics.library. The address of the system copperlist is at offset $26. This address should be saved in a variable, for example old\_cop. Here is an example code, that must be included in the routine take\_system.

  ; gfx base

  move.l     d0,a0

  ; saves system copperlist address

  move.l     $26(a0),sys\_coplist

In the variables section, we must add the following variable:

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; VARIABLES

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

...

; address of system copperlist

sys\_coplist:

  dc.l       0

At the end of our program, we will have to restore the system copperlist, by loading the contents of the sys\_coplist variable into the COP1LC register and resetting the Copper PC, by writing to COP1JMP. The following code must be inserted at the beginning of the release\_system routine:

release\_system:

  ; restores the system copperlist

  move.l     sys\_coplist,COP1LC(a5)

  ; starts the system copperlist

  move.w     d0,COPJMP1(a5)

...

We must also remember to enable the DMA channel of the Copper, acting on the DMASET constant:

; DMACON register settings

; enables only copper DMA (bit 7)

                 ;5432109876543210

DMASET       equ %1000001010000000

The complete code for this chapter can be downloaded from the following url:

<https://github.com/stefanocoppi/amiga_game_prog_assembly/tree/main/chapter6>

If you try to run the program we have written, pressing F5, you will get an image similar to the following figure. You'll see that the background is blue up to line 192, then it turns black.

Immagine che contiene schermata, schermo, Rettangolo

Descrizione generata automaticamente

# 7. How to display images on screen

In this chapter we will see how to display images on the Amiga screen. Finally, by running the sample program, you will have the satisfaction of seeing something on the screen!

What are the most popular formats of graphic images? Nowadays, the most popular formats for bitmap images are JPEG and PNG. The first format is used for digital images, while the second is more used to store graphics for video games, as it also supports palette mode.

Game artists also use the PSD format, native of Photoshop, which also contains layers information, or the XCF format, native of GIMP.

Back in the days of the Amiga, the most popular format was IFF, native of the famous drawing software DeLuxe Paint.

## RAW format

The most convenient format to use when programming in Assembly is the **RAW** format, which is a sequence of bytes that directly represents the graphic data, without any metadata or header.

The VASM assembler provides a directive to include a RAW file in the source code.

The directive has the following syntax:

incbin <raw\_filename>

Please note that the graphic data must be loaded into the chip RAM, so you must create a specific section in the listing.

SECTION graphics\_segment,DATA\_C

incbin "image.raw"

## RAW Converter

But how can we convert PNG images in RAW format?

On Windows there is a command line tool, called **Amiga Image Converter** or **amigeconv**, which can be downloaded for free at the following url:

<https://github.com/tditlu/amigeconv>

Let's start to see how to use this tool. Suppose we have a PNG image, with 256 colors. To convert it to RAW format we must type the following command, in a Windows command prompt window:

amigeconv -f bitplane -d 8 space.png space.raw

In addition to the image graphics data, we need to export the palette. Just type:

amigeconv -f palette -p pal8 -c 256 -x space.png space.pal

This tool also allows the generation of masks for BOBs (which we will see later), export in the format required by hardware sprites, in the interleaved format.

After converting the image to raw format and extracting the palette, we can load them into memory, in the ram chip segment, with the following instructions:

palette incbin "gfx/space.pal" ; palette

img\_space6 incbin "gfx/space.raw" ; image 320 x 256 pixel

; 8 bitplanes

## Palette

Now we need to understand how Amiga handles colors and then load the palette into memory. Amigas with OCS/ECS chipsets have a palette of up to 32 colors. To set the colors of the palette, the registers called COLOR0 - COLOR31 are used, which correspond to the addresses $dff180 - $dff1be. The value to be written in the register has the following format:

|  |  |  |  |
| --- | --- | --- | --- |
| Bit 15-12 | Bit 11-8 | Bit 7-4 | Bit 3-0 |
| Not used | Red | Green | Blue |

In total we will have 12 bits to define the color, so Amiga can display 2^12 = 4096 different colors. As we saw in the lesson on Copper, to set the value of the custom chip registers we use the Copper MOVE instruction, creating a copperlist.

Amigas with AGA chipsets have 2 differences:

1. The values for the red, green, and blue components of the color use 8 bits. So, it uses 24 bits to represent a color. The color combinations that can be represented are 2^24 = 16M of colors!
2. The palette can have a maximum of 256 colors.

How do we enter an 8-bit value for the color components, since the color register has only 16 bits?

Let's represent the components of a color using uppercase letters for the upper nibble and lowercase letters for the low nibble:

RrGgBb

In the color register we will first insert the high nibbles, then:

dc.w $180,$RGB

Then we will set to 1 bit 9 of the BPLCON3 register ($dff106), which enables low nibbles.

dc.w $106,$200

Then we will insert the low nibbles:

dc.w $180,$rgb

How do we set the colors ranging from 32 to 256, given that there are only 32 color registers?

The 256 colors are divided into 8 banks of 32 each, numbered from 0 to 7. The active bank is selected using bits 13-15 of the BPLCON3 register.

|  |  |  |
| --- | --- | --- |
| Value of bit 13-15 | Bank | BPLCON3 value |
| 000 | 0: colors 0-31 | $0 |
| 001 | 1: colors 32-63 | $200 |
| 010 | 2: colors 64-95 | $400 |
| 011 | 3: colors 96-127 | $600 |
| 100 | 4: colors 128-159 | $800 |
| 101 | 5: colors 160-191 | $A00 |
| 110 | 6: colors 192-223 | $B00 |
| 111 | 7: colors 224-255 | $E00 |

Suppose you want to view a 256-color image on an Amiga AGA.

Let's take up the code of the previous chapter because we will start from it.

Let's start creating the copperlist.

Remember that the copperlist must be inserted in chip ram.

; segment loaded in CHIP RAM

SECTION graphics\_data,DATA\_C

copperlist:

...

palette\_coplist:

             ; colors 0-31 , high nibbles

             dc.w       BPLCON3,$000

             dc.w       $180,0,$182,0,$184,0,$186,0

             dc.w       $188,0,$18a,0,$18c,0,$18e,0

             dc.w       $190,0,$192,0,$194,0,$196,0

             dc.w       $198,0,$19a,0,$19c,0,$19e,0

             dc.w       $1a0,0,$1a2,0,$1a4,0,$1a6,0

             dc.w       $1a8,0,$1aa,0,$1ac,0,$1ae,0

             dc.w       $1b0,0,$1b2,0,$1b4,0,$1b6,0

             dc.w       $1b8,0,$1ba,0,$1bc,0,$1be,0

             ; colors 0-31 , low nibbles

             dc.w       BPLCON3,$200

             dc.w       $180,0,$182,0,$184,0,$186,0

             dc.w       $188,0,$18a,0,$18c,0,$18e,0

             dc.w       $190,0,$192,0,$194,0,$196,0

             dc.w       $198,0,$19a,0,$19c,0,$19e,0

             dc.w       $1a0,0,$1a2,0,$1a4,0,$1a6,0

             dc.w       $1a8,0,$1aa,0,$1ac,0,$1ae,0

             dc.w       $1b0,0,$1b2,0,$1b4,0,$1b6,0

             dc.w       $1b8,0,$1ba,0,$1bc,0,$1be,0

...

dc.w $ffff,$fffe ; end of copperlist

Notice how we first select the high nibbles of deck 0, then initialize the entire deck with zeros. Then we select the low nibbles of deck 0 and initialize them to zero. Next, we will load the palette instead of the values set to zero.

Repeat the same process for the other 7 color register banks.

Now we need to create a routine that loads the palette data from memory into the copperlist. The following routine creates 2 loops: an inner loop inserts the nibbles into the 32 registers of a bank, while the outer loop iterates over the 8 banks one time for the high nibble and a second time for the lower nibble.

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; Load palette into copperlist.

;

; parameters:

; a0 - address of palette

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

load\_palette:

movem d0-a6,-(sp) ; saves registers into stack

moveq #16-1,d6 ; number of external loop iterations

lea palette\_coplist,a1 ; pointer to palette data in copperlist

add.l #2,a0 ; points to first color value

add.l #2,a1

.ext\_loop:

add.l #4,a0 ; points to first color value

add.l #4,a1

moveq #32-1,d7 ; number of loop iterations

.loop:

move.w (a0),(a1) ; copy color value from memory to

; copperlist

add.l #4,a0 ; points to next color value in the

; palette

add.l #4,a1 ; point to the next value in the

; copperlist

dbra d7,.loop ; repeats the loop for all 32 colors of

; the color bank

dbra d6,.ext\_loop ; repeats the loop for all banks, for

; high and low nibbles

movem (sp)+,d0-a6 ; restores registers from stack

rts

## Bitplanes

Now let's see how Amiga displays images.

The Amiga video processor reads the image information from memory and drives the electron beam (raster) to draw the image on the monitor screen. An image on the screen is formed by moving the electron beam from left to right, one line at a time from top to bottom. This process is repeated 50 times per second in the European PAL system or 60 times on NTSC systems.

Immagine che contiene linea, cassettiera, cassetto

Descrizione generata automaticamente

But how are images encoded in memory? The concept of **Bitplane** is used. A bitplane is nothing more than a matrix of bits, having the same dimensions as the screen resolution. For example, if we have a low-resolution screen of 320x256 pixels, it will be represented by a bitplane made up of 256 lines of 320 bits each, or 40 bytes. With a single bitplane we can display images with only two colors, in fact the zero bits will represent pixels colored with the first color of the palette (COLOR0 register), while the 1 bits will represent pixels colored with the second color of the palette (COLOR1).

At this point the question might be: how do we represent images with more than 2 colors? The answer is: using more bitplanes. In fact, if we have 2 bitplanes, we have 2^2=4 combinations, corresponding to the first 4 colors of the palette. To know the color of a pixel, we must:

1. calculate the index N of the color in the palette
2. Read the element N of the palette

To calculate the index N of a pixel, simply read the value of the pixel in the various bitplanes and write them down, starting from the bitplane with the highest index, which will correspond to the most significant digit of the index.

Immagine che contiene testo, diagramma, linea, Parallelo

Descrizione generata automaticamente

Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamente

So, with 3 bitplanes, we can display 2^3=8 different colors, while with 4 bitplanes the colors will be 2^4=16 and so with 5 bitplanes we will have 2^5=32 different colors. The OCS/ECS chipsets support a maximum of 5 bitplanes. The AGA chipset supports up to 8 bitplanes. The bitplanes are normally stored one after the other. So, in the case of our 320x256 pixel resolution image with 16 colors, a bitplane is made up of 256 rows of 320 bits (40 bytes) each, with a total size of 10240 bytes. Having 16 colors, we will have to use 4 bitplanes. So, we will first have 10240 bytes representing bitplane0, then 10240 bytes of bitplane1, 10240 bytes of bitplane2 and 10240 bytes of bitplane3. In total the image takes up 10240\*4 = 40960 bytes.

There is another way to store bitplanes: the INTERLEAVED one, that is, the first line of bitplane0 is stored, the first line of bitplane1, ..., the first line of bitplane3, the second line of bitplane0, the second line of bitplane1, etc... This format is always managed by the amigeconv tool that we used, however we will not use it in this book.

At this point we have understood how an image is organized in memory and how it is drawn on the screen. But we still do not know how to tell the video processor where to read the image from in memory. There are registers called BPLxPT that contain the memory address of the various bitplanes.

|  |  |  |
| --- | --- | --- |
| Register | Address |  |
| BPL0PT | $dff0e0 | bitplane0 |
| BPL1PT | $dff0e4 | bitplane1 |
| BPL2PT | $dff0e8 | bitplane2 |
| BPL3PT | $dff0ec | bitplane3 |
| BPL4PT | $dff0f0 | bitplane4 |
| BPL5PT | $dff0f4 | bitplane5 |
| BPL6PT | $dff0f8 | bitplane6 |
| BPL7PT | $dff0fc | bitplane7 |

In the copperlist we will have a section where we will set the bitplane pointers.

Each pointer is composed of a high and a low word. We initialize the values to zero. Then we'll use a routine to initialize them.

bplpointers:

  dc.w    $e0,0,$e2,0    ; plane 0

  dc.w    $e4,0,$e6,0    ; plane 1

  dc.w    $e8,0,$ea,0    ; plane 2

  dc.w    $ec,0,$ee,0    ; plane 3

  dc.w    $f0,0,$f2,0    ; plane 4

  dc.w    $f4,0,$f6,0    ; plane 5

  dc.w    $f8,0,$fa,0    ; plane 6

  dc.w    $fc,0,$fe,0    ; plane 7

How do you specify the number of bitplanes to use? There is the BPLCON0 ($dff100) register, whose bits 12-14 indicate least significant bits of the number of active bitplanes while bit 4 indicates the most significant bit. So, to set 8 bitplanes, you must set to 0 bits 12-14 and set to 1 bit 4.

It is good practice to always set bit 9 of this register to 1.

In the copperlist we must add the following code:

; BPLCON0 ($100)

  ; bit 0: set to 1 to enable BLTCON3 register

  ; bit 4: most significant bit of bitplane number

  ; bit 9: set to 1 to enable composite video output

  ; bit 12-14: least significant bits of bitplane number

  ;                5432109876543210

  dc.w    BPLCON0,%0000001000010001

## Display window size

The visible portion of the screen, called the display window, is smaller than the entire monitor screen.

Immagine che contiene diagramma, linea, Rettangolo, Disegno tecnico

Descrizione generata automaticamente

Therefore, there are registers to set the start and end of the video window, DIWSTRT and DIWSTOP respectively.

|  |  |  |
| --- | --- | --- |
| register | address | value |
| DIWSTRT | $dff08e | bit 8-15 vertical position |
|  |  | bit 0-7 horizontal position |
| DIWSTOP | $dff090 | bit 8-15 vertical position (add $1 as MSB) |
|  |  | bit 0-7 horizontal position (add $1 as MSB) |

Typically a 320x256 lowres display in PAL format has the following values:

DIWSTRT $2c81 ($81,$2c) = (129,44)

DIWSTOP $2cc1 ($1c1,$12c) = (449,300)

Note that the DIWSTOP values ​​must be preceded by a $1 as the most significant bit.

There are also two registers to set the start and end position of reading data from memory, called DDFSTRT and DDFSTOP.

|  |  |  |  |
| --- | --- | --- | --- |
| register | address | format | default PAL lowres value |
| DDFSTRT | $dff092 | bit 0-1: always 0 | $38 |
|  |  | bit 2-7: horizontal position in pixel |  |
|  |  | bit 8-15: always 0 |  |
| DDFSTOP | $dff094 | bit 0-1: always 0 | $D0 |
|  |  | bit 2-7: horizontal position in pixel |  |
|  |  | bit 8-15: always 0 |  |

The value of DDFSTRT is calculated with the following formula:

DDFSTRT = DIWSTRT.x/2 - 8

In fact DIWSTRT.x = 129/2 - 8 = 64 - 8 = 56 ($38)

The value of DDFSTOP is calculated with the following formula:

DDFSTOP = DDFSTART + 8 \* (display\_window\_in\_word -1)

In fact we have DDFSTOP = $38 + 8\*(20 -1) = $38 + $98 = $D0

Let's see how to set the size of the display window in the copperlist:

dc.w DIWSTRT,$2c81 ; display window start at ($81,$2c)

dc.w DIWSTOP,$2cc1 ; display window stop at ($1c1,$12c)

dc.w DDFSTRT,$38 ; display data fetch start at $38

dc.w DDFSTOP,$d0 ; display data fetch stop at $d0

dc.w BPLCON1,0

dc.w BPLCON2,0

dc.w BPL1MOD,0

dc.w BPL2MOD,0

dc.w FMODE,0 ; 16 bit fetch mode

We will see the meaning of the BPLCON1 registers, etc. afterwards.

Now let's write a routine to initialize bitplane pointers.

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; Initializes bitplane pointers

;

; parameters:

; d0.l - address of bitplanes

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

init\_bplpointers:

movem d0-a6,-(sp)

lea bplpointers,a1 ; bitplane pointers in a1

move.l #(N\_PLANES-1),d1 ; number of loop iterations in d1

.loop:

move.w d0,6(a1) ; copy low word of image address into

; BPLxPTL (low word of BPLxPT)

swap d0 ; swap high and low word of image

; address

move.w d0,2(a1) ; copy high word of image address

; into BPLxPTH (high word of BPLxPT)

swap d0 ; resets d0 to the initial condition

add.l #DISPLAY\_PLANE\_SZ,d0 ; point to the next bitplane

add.l #8,a1 ; point to next bplpointer

dbra d1,.loop ; repeats the loop for all planes

movem (sp)+,d0-a6

rts

This routine, along with the load\_palette, must be called in the main, before entering the mainloop, as shown below.

main:

nop

nop

bsr take\_system ; takes the control of Amiga's hardware

lea palette,a0 ; pointer to palette data in memory

bsr load\_palette ; loads palette into copperlist

move.l #img\_space6,d0 ; address of image in d0

bsr init\_bplpointers ; initializes bitplane pointers to our

; image

mainloop:

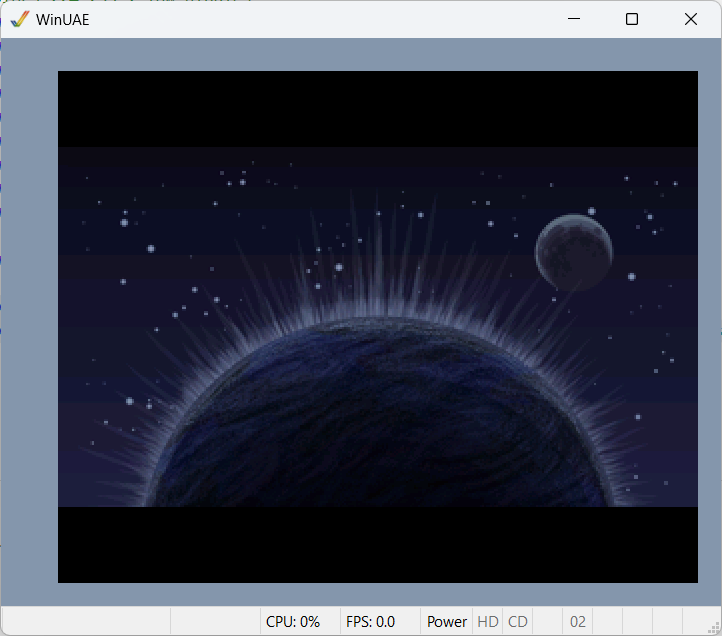
btst #6,CIAAPRA ; left mouse button pressed?

bne.s mainloop ; if not, repeats the loop

bsr release\_system ; releases the hw control to the O.S.

rts

Running the code we have created, we will get the display of a beautiful 256-color space image, in 320 \* 256 resolution.



The complete source code can be downloaded from the following url:

<https://github.com/stefanocoppi/amiga_game_prog_assembly/tree/main/chapter7>

# 8. Blitter

## Introduction

What is the **Blitter**? It is a coprocessor specialized in copying blocks of memory and drawing lines. It is in the Agnus chip. Its name comes from the acronym of "BLock Image Transfer" or transfer of image blocks. In jargon, the operations of copying image blocks are called "blit" operations or "blittings". Much of the graphics capabilities of the Amiga are due to this coprocessor, which can operate in parallel with the CPU, accessing the chip memory via DMA. We need to learn how to use it if we want to do graphics with the Amiga.

## DMA Channels

The Blitter accesses the chip ram memory via DMA, without the need for the CPU. In this way the CPU is free to do other things during image copying operations, which generally require long times. This parallelism between the CPU and the coprocessors is the secret of the Amiga's efficiency. The characteristic of the DMA channels used by the Blitter is that they have a width of 16 bits, so they can transfer one word at a time. For this reason, the DMA channels can only access memory addresses that are multiples of 16 bits, i.e. even ones. The Blitter can use 3 DMA channels for input, called "A, B, C". For output it has only one channel, called "D".

How do you enable DMA channels? There is a BLTCON0 (**Blitter Control Register 0**) register, which has the following format:

|  |  |  |  |
| --- | --- | --- | --- |
| Register | Address | Bit position | Usage |
| BLTCON0 | $dff040 | 12-15 | Shift value for channel A |
|  |  | 11 | Enable channel A |
|  |  | 10 | Enable channel B |
|  |  | 9 | Enable channel C |
|  |  | 8 | Enable channel D |
|  |  | 0-7 | Minterms selection |

How do you specify the memory location to read from (channels A, B, C) or write to (channel D)? You use the **BPLxPT** registers, which are pointers to each DMA channel of the Blitter.

|  |  |  |
| --- | --- | --- |
| Register | Address | Usage |
| BPLAPT | $dff050 | Pointer for Blitter Channel A |
| BPLBPT | $dff04c | Pointer for Blitter Channel B |
| BPLCPT | $dff048 | Pointer for Blitter Channel C |
| BPLDPT | $dff054 | Pointer for Blitter Channel D |

## Logic functions

We have seen that the Blitter can use up to three DMA channels as input that read from memory and a single D channel as output. But what operations can it perform on the read data?

These operations are called **logic functions** or **minterms** and are specified by bits 0-7 of the BLTCON0 register. The table below shows the values ​​for the most used logic functions.

|  |  |
| --- | --- |
| value of bit 0-7 of BLTCON0 | Logic function |
| $F0 | D = A |
| $CA | D = AB + (NOT A)C |
| $0F | D = NOT A |

The logic function in the second line is used to draw sprites, as we will see later. For now, we will use the first logic function.

## Blitter busy

Before modifying any Blitter registers, you need to check that the Blitter is not already operating. Bit 14 of the DMACONR register ($dff002) is called BBUSY and is set to 1 if the Blitter is performing an operation. We can write a simple routine that waits until the Blitter is free, that is, until the BBUSY bit is zero.

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; Wait for the blitter to finish

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

wait\_blitter:

.loop:

  btst.b    #6,DMACONR(a5)    ; if bit 6 is 1, the blitter is busy

  bne       .loop             ; and then wait until it's zero

  rts

## BLTSIZE and how to start Blitter

How do you tell the Blitter how much data to process? There is a register called BLTSIZE ($dff058). The Blitter specifies the size of the area to process as a rectangle. The height is measured in number of rows, while the width is measured in number of WORDS. Bits 0-5 of BLTSIZE indicate the width, while bits 6-15 indicate the height. How do you start the Blitter? The Blitter starts as soon as the BLTSIZE register is written. For this reason, it must be the last register to be set.

## Modulus

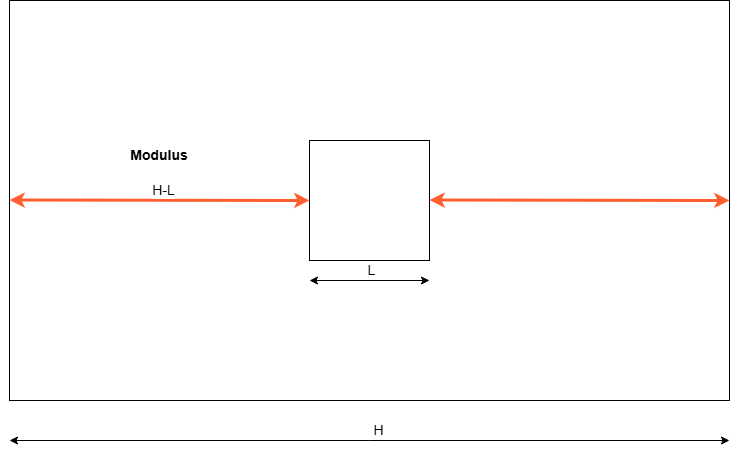
Suppose we want to copy a 32-pixel wide image onto a 320 pixel wide screen. When the Blitter has copied the first line of the image onto the screen, it must move on to transferring the second line. We must tell the Blitter how many bytes it must skip before writing the second line of the image. This quantity is called **Modulus** and is calculated with the following formula:

Modulus = (H-L)/8

where H = destination image width in pixel

L = source image width in pixel

If H=320 and L=32, the modulus of the D channel will be (320-32)/8 = 36. If the source image is exactly 32 pixels wide, then its modulus will be zero. The following image helps to understand the concept of modulus.



To set the moduli for each Blitter channel, the following registers are used:

|  |  |  |
| --- | --- | --- |
| Register | Address | Channel |
| BLTAMOD | $dff064 | A |
| BLTBMOD | $dff062 | B |
| BLTCMOD | $dff060 | C |
| BLTDMOD | $dff066 | D |

## Shift

We have seen that DMA channels operate on words and that their pointer addresses must be aligned to word, i.e. integer multiples of 16. How do we draw an object via the Blitter with an x ​​coordinate that is not a multiple of 16? We must use the ability to shift the source data of channels A, B up to a maximum of 16 pixels.

|  |  |  |  |
| --- | --- | --- | --- |
| Register | Address | Shift value | Channel |
| BLTCON0 | $dff040 | bit 12-15 | A |
| BLTCON1 | $dff042 | bit 12-15 | B |

We will see in detail how to use the shift in a next chapter on drawing Bobs.

## Masking

The blitter can mask the first and last word of each line that passes through the A channel. Masking means that the Blitter performs a logical AND operation between the read word and a mask. The mask is specified in two registers called BLTAFWM ($dff044) and BLTALWM ($dff046), and they serve respectively to indicate the mask of the first and last word of each line read through the A channel. This functionality will be used for drawing the Bobs and will be explained in detail later.

## Example of copying an image to the screen

For the example of this chapter, we will start from the listing of last chapter. Instead of pointing the bitplane pointers to a raw image, we point them to a completely cleared memory area. This will make the displayed screen completely blank. To create a cleared memory area in chip ram, without taking up space in the file, there is the BSS\_C segment. So, we will have to declare it as follows:

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; BSS DATA

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

        SECTION    bss\_data,BSS\_C

screen  ds.b       (DISPLAY\_PLANE\_SZ\*N\_PLANES)    ; visible screen

To learn how to use the Blitter, we will copy a 64x64 pixel 8-color image from memory to the screen.

To convert the PNG image to indexed format, we need to issue the following command, in a Windows Command Prompt window:

amigeconv -f bitplane -d 3 tile.png tile.raw

To extract the palette from the PNG image, with the colors in 8 bit format per component, in a format suitable for inclusion in the copperlist, we must issue the following command:

amigeconv -f palette -p pal8 -c 8 -x tile.png tile.pal

Let's start by including the image in memory, in the chip ram segment:

; segment loaded in CHIP RAM

          SECTION    graphics\_data,DATA\_C

img\_tile  incbin     "gfx/tile.raw"   ; image 64 x 64 pixel , 3 bitplanes

We include the palette directly in the copperlist, right after the bitplane pointers. Make sure that the palette has been exported in Copper format.

Compared to the previous chapter, here we will use 3 bitplanes, so in binary %0011. Of the binary number, the most significant bit must be inserted in bit 4 of BPLCON0. The three least significant bits are inserted into bits 12-14.

copperlist:

         ...

  ; BPLCON0 ($100)

  ; bit 0: set to 1 to enable BLTCON3 register

  ; bit 4: most significant bit of bitplane number

  ; bit 9: set to 1 to enable composite video output

  ; bit 12-14: least significant bits of bitplane number

  ; bitplane number: 3 => %0011

  ;                         5432109876543210

         dc.w      BPLCON0,%0011001000000001

         dc.w      FMODE,0                      ; 16 bit fetch mode

bplpointers:

         dc.w      $e0,0,$e2,0                  ; plane 1

         dc.w      $e4,0,$e6,0                  ; plane 2

         dc.w      $e8,0,$ea,0                  ; plane 3

palette  incbin    "gfx/tile.pal"               ; palette

         dc.w      $ffff,$fffe                  ; end of copperlist

Let's create a routine that copies this image to the visible screen:

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; Draws a 64x64 pixel tile using blitter.

;

; parameters:

; a0 - address of tile

; a1 - address where draw the tile

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

draw\_tile:

  movem.l    d0-a6,-(sp)                    ; saves registers into the stack

  moveq      #N\_PLANES-1,d1

  bsr        wait\_blitter

  move.w     #$ffff,BLTAFWM(a5)             ; don't use mask

  move.w     #$ffff,BLTALWM(a5)

  move.w     #$09f0,BLTCON0(a5)             ; enable channels A,D

                                        ; logical function = $f0, D = A

  move.w     #0,BLTCON1(a5)

  move.w     #0,BLTAMOD(a5)

  move.w     #(DISPLAY\_WIDTH-TILE\_WIDTH)/8,BLTDMOD(a5)  ; D channel modulus

.loop:

  bsr        wait\_blitter

  move.l     a0,BLTAPT(a5)                  ; source address

  move.l     a1,BLTDPT(a5)                  ; destination address

  move.w     #64\*64+4,BLTSIZE(a5)           ; blit size: 64 rows for 4 words

  add.l      #TILE\_PLANE\_SZ,a0              ; advances to the next plane

  add.l      #DISPLAY\_PLANE\_SZ,a1

  dbra       d1,.loop

  bsr        wait\_blitter

  movem.l    (sp)+,d0-a6                   ; restores registers from the stack

  rts

Notice that 3 blit operations are performed, one for each bitplane, using a loop. After each blit operation, the source and destination addresses are incremented to point to the next plane.

The draw\_tile routine must be called after the call to init\_bplpointers. We need to initialize the a0 register with the tile address and the a1 register with the address where we want to draw the tile. Suppose we want to display it at the center of the screen. The x and y coordinates must be loaded into the d0, d1 registers. We need to calculate the y\_offset and x\_offset to add to the base address of the screen. The vertical offset is simply the y position multiplied by the size in bytes of a display row, DISPLAY\_ROW\_SIZE. The horizontal offset is the x position divided by 8. Note that instead of division, we use a right shift, since the division instruction takes more clock cycles than the shift.

main:

  nop

  nop

  bsr       take\_system           ; takes the control of Amiga's hardware

  move.l    #screen,d0            ; address of screen in d0

  bsr       init\_bplpointers      ; initializes bitplane pointers to our image

  lea       img\_tile,a0

  move.w    #(DISPLAY\_WIDTH-TILE\_WIDTH)/2,d0      ; x position

  move.w    #(DISPLAY\_HEIGHT-TILE\_HEIGHT)/2,d1    ; y position

  mulu      #DISPLAY\_ROW\_SIZE,d1                  ; y\_offset = y \*

; DISPLAY\_ROW\_SIZE

  asr.w     #3,d0                                 ; x\_offset = x/8

  add.w     d1,d0                                 ; sum the offsets

  ext.l     d0

  lea       screen,a1

  add.l     d0,a1                                 ; sum the offset to a1

  bsr       draw\_tile

...

In the take\_system routine we added a call to the \_LVOOwnBlitter routine of graphics.library, which grants exclusive use of the Blitter to the application.

take\_system:

  move.l    ExecBase,a6               ; base address of Exec

  jsr       \_LVOForbid(a6)            ; disables O.S. multitasking

  jsr       \_LVODisable(a6)           ; disables O.S. interrupts

  lea       gfx\_name,a1               ; OpenLibrary takes 1 parameter: library

; name in a1

  jsr       \_LVOOldOpenLibrary(a6)    ; opens graphics.library

  move.l    d0,gfx\_base               ; saves base address of graphics.library

; in a variable

  move.l    d0,a6                     ; gfx base

  move.l    $26(a6),sys\_coplist       ; saves system copperlist address

  jsr       \_LVOOwnBlitter(a6)        ; takes the Blitter exclusive

...

In the release\_system routine we added a call to the \_LVODisownBlitter routine of graphics.library, to release the use of Blitter to other processes.

release\_system:

  move.l    sys\_coplist,COP1LC(a5)    ; restores the system copperlist

  move.w    d0,COPJMP1(a5)            ; starts the system copperlist

  or.w      #$8000,old\_dma            ; sets bit 15

  move.w    old\_dma,DMACON(a5)        ; restores saved DMA state

  move.l    gfx\_base,a6

  jsr       \_LVODisownBlitter(a6)     ; release Blitter ownership

  ...

The complete listing can be downloaded from the following url:

<https://github.com/stefanocoppi/amiga_game_prog_assembly/tree/main/chapter8>

By compiling the program and running it, you will get the image below, which shows a tile with which we will build the level of shoot’em up videogame.

Immagine che contiene schermata, testo, Rettangolo

Descrizione generata automaticamente

# 9. Tiles and tilemaps

## Tiles

A tile is nothing more than a square image, such as 64 x 64 pixels, that is used to compose the image of the game level. Making an analogy with a puzzle, the game level is the puzzle itself, made up of tiles.

The set of tiles used to make up a layer is called a **tileset**. Each tile is assigned a unique id, called a **tile index**.

Immagine che contiene schermata, testo, Rettangolo

Descrizione generata automaticamente

## Tilemap

How can the game level, made up of tiles, be represented? You can represent it with a matrix, whose elements represent the tile index. This matrix is called **Tilemap**.

To recap, to represent a game level, you need a tileset and a tilemap.



## Why are tiles used?

This question arises, in fact one might think that the easiest thing is to use a single large image of the level. The problem is in the size of that image. If we want a map 20 screens long, the image will have dimensions 20\*320=6400 horizontal pixels and 256 pixels vertical. A single bitplane of this image occupies (6400/8) \*256 = 204,800 bytes. If we use 8 bitplanes, the image will take up 1,638,400 bytes or 1.56 MBytes. You'll understand that we can't use all that memory for the background of the level alone, otherwise there wouldn't be enough memory for the rest of the graphics and sound. In addition, the video game could only run on Amiga AGA with more than 2 MB of chip memory.

Let's see how much a level takes up using tiles. Assuming tiles of 64 x 64 pixels, the tilemap will be a matrix of 6400/64 = 100 columns and 256/64 = 4 rows. Assuming you use a word for tile indexes, the tilemap will consist of 100 \* 4 = 400 words or 400 \* 2 = 800 bytes. To this must be added the image of the tileset, which we assume is 640 x 640 pixels. A bitplane occupies (640/8) \*640 = 51,200 bytes. Assuming you use 8 bitplanes, the tileset will take up 51,200 \* 8 = 409,600 bytes, which is 400KB or 0.39MB. So, with about 0.4 MB we can store the entire layer against 1.56 MB for the single image. We manage to reduce memory occupancy by a factor of almost 4. In this way we can use the remaining memory for other graphics or sound and the video game can also run on the base A1200. If we used fewer bitplanes, it could run on the A500 as well.

## Creating the tileset

As a tool for all graphics operations we will use Pro Motion NG, available at the following url:

<https://www.cosmigo.com/pixel_animation_software/downloads>

Both free and full paid versions of this tool are available. For our purposes, the free one is sufficient.

Now we should draw the tiles. Since I'm not an artist, I thought I'd buy a ready-made set of graphics for space shoot'em ups, costing about $10, from the following website:

<https://craftpix.net/>

Let's see how to prepare a tileset. Let's open a single tile with Pro Motion NG, using the "Load image" item from “File” menu. If the file is in RGB mode, the following dialog box will appear, to reduce the number of colors to 256, creating an optimized palette. In fact, Pro Motion NG works exclusively with palettes. Press OK to accept the default options.

Immagine che contiene testo, schermata, numero, Carattere

Descrizione generata automaticamente

The tile will be loaded and an image similar to the following will appear:

Immagine che contiene testo, schermata, software, Software multimediale

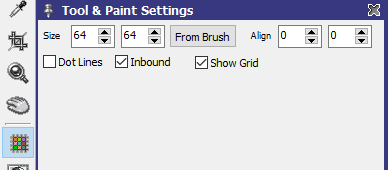
Descrizione generata automaticamente

To add the other tiles, we first need to enlarge the size of the image. From the “Frame” menu, select the "Resize Canvas" item. The following dialog box will appear. Set Width = 640 and Height = 640 and press OK.

Immagine che contiene testo, schermata, schermo, numero

Descrizione generata automaticamente

You will see that the image has been enlarged. To show a grid, you right-click on the grid-shaped button, highlighted in the figure below. In the "Tool & Paint Settings" box, set the grid size to 64 x 64 pixels and press the "Show Grid" checkbox. The grid will be displayed.



To add a second tile, from the "Brush" menu, select the "Load Brush..." item. Then select the tile file using the dialog box that will appear and press OK. The dialog box for optimizing the palette will appear. Press OK again. Now the tile will be loaded and you can place it on the grid with the mouse. You will observe anomalous colors. In fact, we must import the colors of the brush palette into the palette of our image. To do this, from the "Colors" menu, select the item "Import Colors from Brush...". The following dialog box will appear.

Immagine che contiene testo, schermata, schermo, software

Descrizione generata automaticamente

This window shows the number of colors required by the brush, those already present in the palette and those left to be imported. Press on the "Auto" button and we will see that the colors will be added to the palette. At this point we press the "Import" button. We will see that now the tile will take on the correct colors. Let's place it next to the first tile:

Immagine che contiene modello, arte, Rettangolo, piastrella

Descrizione generata automaticamente

Let's look at the box at the bottom right, called "Color Palette". This box shows the palette of the current image. The selected color is shown with a white triangle. At the bottom you can see the values of the selected color and you can change it using the sliders.

Immagine che contiene testo, schermata, schermo, software

Descrizione generata automaticamente

We continue to add the various tiles, repeating the same procedure. With each tile added, the palette will increase in size. To find out the number of colors used, from the "Colors" menu, select the item "Count Colors used":

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

The following dialog box will appear:

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Pro Motion NG also allows you to reduce the number of colors in an image.

From the "Colors" menu, select "Reduce Colors...":

Immagine che contiene testo, schermata, software, numero

Descrizione generata automaticamente

The following window will appear:

Immagine che contiene testo, schermata, schermo, numero

Descrizione generata automaticamente

The only value to change is the number of colors. By pressing "OK" we will be able to see the result. Obviously, by greatly reducing the number of colors, image quality is lost as it is not possible to reproduce the shades.

At the end we will have a tileset, in indexed PNG format, like the one in the figure.

Immagine che contiene modello, Rettangolo, arte

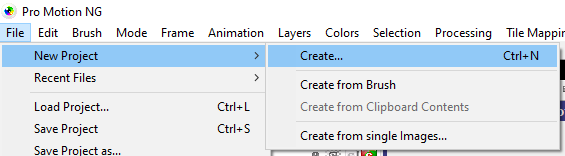
Descrizione generata automaticamente

Be careful to always leave the first tile empty so as not to have problems later. You can find the tileset file I used at this url:

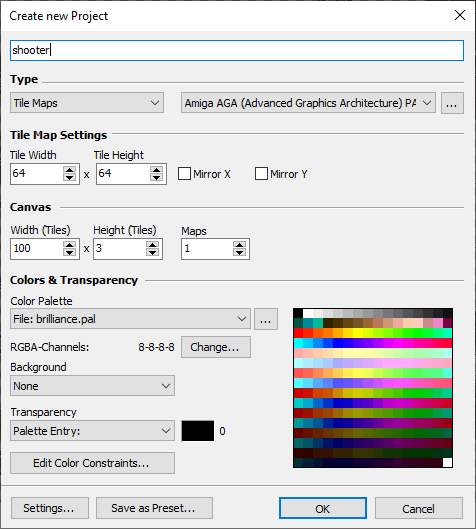
<https://github.com/stefanocoppi/amiga_game_prog_assembly/blob/main/chapter9/gfx/shooter_tiles.png>

## Creating the tilemap

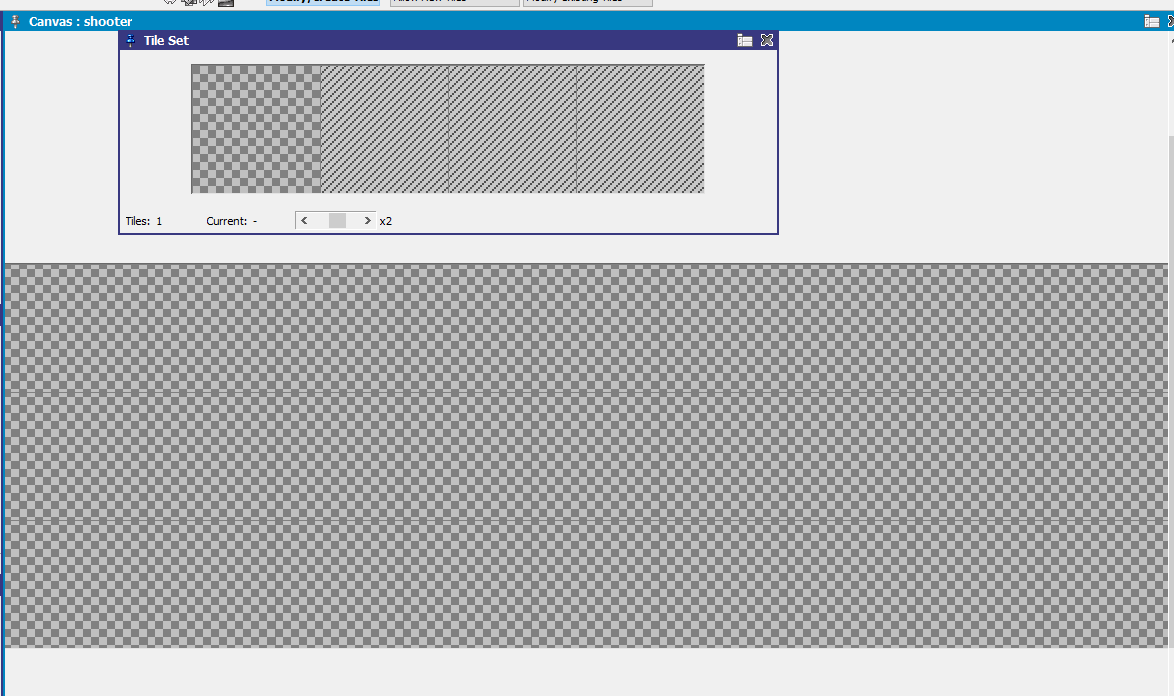
To create a tilemap with Pro Motion NG, from the "File" menu, select the "New Project" item and then "Create...".



The following window will appear:



Enter the project name, tile size (64 x 64), tile width and height (100 x 3), and leave the rest of the options as is. Press OK. The following image will appear:



In the upper part you will have the tileset, while in the central part you will have the tilemap. Now they are both empty.

Now let's import the tileset we created earlier. From the "Tile Mapping" menu, select the "Import Tiles..." option.

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

Select the PNG image containing the tileset and press OK. The following window will appear. Select "Append new Tiles" and press OK.

Immagine che contiene testo, schermata, schermo, numero

Descrizione generata automaticamente

The following window will appear to import the colors. Press "Auto" and then "Import".

Immagine che contiene testo, schermata, software, schermo

Descrizione generata automaticamente

If you now enlarge the "Tile Set" window, you will be able to see all the tiles, with the currently selected one highlighted in purple.

Immagine che contiene testo, schermata, modello, arte

Descrizione generata automaticamente

To create the map, select a tile in the tileset window and then draw it on the main map window, as shown in the following figure.

Immagine che contiene schermata, design

Descrizione generata automaticamente

Use a little imagination and creativity and complete the map. If you want to use a background color instead of a transparent one, select the "Background" box in the top left corner. You can choose the color or palette index for the background.

Immagine che contiene testo, schermata, software, schermo

Descrizione generata automaticamente

In the end, you will get something like the figure below.

Immagine che contiene schermata, linea

Descrizione generata automaticamente

You can find the complete project of the layer used in the book at the following url:

<https://github.com/stefanocoppi/amiga_game_prog_assembly/blob/main/chapter9/gfx/map_final.pmp>

## Alternative way to create a tileset/tilemap

There is an alternative way to create a tileset. Suppose you draw the level map directly and save it to a “map.png” image.

Load the map image into Pro Motion NG and then select the "Turn Project into Tile Maps" item from the "Tile Mapping" menu:



The following window will appear:

Immagine che contiene testo, schermata, schermo, numero

Descrizione generata automaticamente

Enter 32 for the width and height of a Tile and press OK.

Pro Motion NG will automatically create a tileset, with tiles measuring 32x32 pixels this time:

Immagine che contiene schermata, testo, modello, Rettangolo

Descrizione generata automaticamente

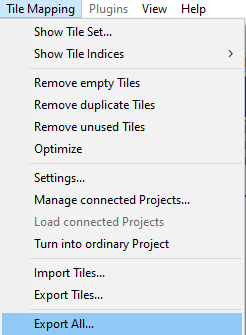
It will also automatically create a tilemap corresponding to the layer image we initially provided:

Immagine che contiene schermata, linea

Descrizione generata automaticamente

## Export tilesets, palettes and map

To export the tileset and the tilemap, from the "Tile Mapping" menu, select the "Export All..." item.



The following window will appear:

Immagine che contiene testo, schermata, schermo, numero

Descrizione generata automaticamente

You can enter the number of columns in the tileset. For "Tile Map File Type" select "TXM – Text Tile Map", which is basically a csv file. Select the destination directory and press "Export". In that directory you will find both the PNG file of the tileset and the TXM file with the tilemap.

You can find these files at the following url:

<https://github.com/stefanocoppi/amiga_game_prog_assembly/tree/main/chapter9/gfx>

## Import tilesets and tilemaps into our program

Let's convert the PNG image of the tileset to raw format and extract the palette using the amigeconv tool. Remember that the PNG image is 256 colors, so we need to use 8 bitplanes. To extract the palette, we use the pal8 format, with 8 bits per color, 256 total colors, and the -x option to export to Copperlist format.

amigeconv.exe -f bitplane -d 8 .\shooter\_tiles.png shooter\_tiles.raw

amigeconv.exe -f palette -p pal8 -c 256 -x .\shooter\_tiles.png shooter\_tiles.pal

Let's include the two files in a chip memory segment:

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; Graphics data

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; segment loaded in CHIP RAM

             SECTION    graphics\_data,DATA\_C

copperlist:

...

palette      incbin     "gfx/shooter\_tiles.pal"       ; palette

             dc.w       $ffff,$fffe                   ; end of copperlist

tileset      incbin     "gfx/shooter\_tiles.raw"       ; image 640 x 512 pixel ,

; 8 bitplanes

To create the tilemap file, we rename the \*.txm file to shooter\_tilemap.i. Then we open this file with Visual Studio Code. We add the dc.w directive at the beginning of each line, so that we can then include this file in the Assembly source.

dc.w 0,0,0,0,0,0,0,0,0, ...

dc.w 0,0,0,0,0,0,0,0,0, ...

 dc.w 0,0,0,0,2,3,3,69, ...

We save the file and include it in our source, at the end of the section reserved for variables:

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; VARIABLES

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

...

map  include    "gfx/shooter\_map.i"

## Drawing a tile

The routine that draws a tile using Bliiter is an evolution of the one created in the previous chapter. First we calculate the memory address where to copy the tile. At the base address of the screen we must add the x and y offsets. The offset\_y is obtained by multiplying the y by the size in bytes of the screen (DISPLAY\_ROW\_SIZE). The offset\_x is obtained by dividing the x by 8 (for efficiency reasons we use a right shift of 3).

We need to calculate the row and column of the tile in the tileset. By dividing the tile index by the number of columns in the tileset (TILESET\_COLS), we get the row in the quotient and the remainder is the column of the tile.

To get the x,y coordinates of the tile in the tileset, just multiply the row and column by 64 (tile width and height).

We then need to calculate the source address of the tile to be copied to the screen. We need to add the x and y offsets to the base address of the tileset.

At this point we set the Blitter registers and perform a loop to copy all 8 planes of the image. The complete routine code is the following:

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; Draws a 64x64 pixel tile using Blitter.

;

; parameters:

; d0.w - tile index

; d2.w - x position of the screen where the tile will be drawn (multiple of 16)

; d3.w - y position of the screen where the tile will be drawn

; a1   - address where draw the tile

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

draw\_tile:

  movem.l    d0-a6,-(sp)                  ; saves registers into the stack

  ; calculates the screen address where to draw the tile

  mulu       #DISPLAY\_ROW\_SIZE,d3         ; y\_offset = y \* DISPLAY\_ROW\_SIZE

  lsr.w      #3,d2                        ; x\_offset = x / 8

  ext.l      d2

  add.l      d3,a1                        ; sums offsets to a1

  add.l      d2,a1

  ; calculates row and column of tile in tileset starting from index

  ext.l      d0                           ; extends d0 to a long because the

; destination operand if divu must be

; long

  divu       #TILESET\_COLS,d0             ; tile\_index / TILESET\_COLS

  swap       d0

  move.w     d0,d1                        ; the remainder indicates the tile

; column

  swap       d0                         ; the quotient indicates the tile row

  ; calculates the x,y coordinates of the tile in the tileset

  lsl.w      #6,d0                        ; y = row \* 64

  lsl.w      #6,d1                        ; x = column \* 64

  ; calculates the offset to add to a0 to get the address of the source image

  mulu       #TILESET\_ROW\_SIZE,d0         ; offset\_y = y \* TILESET\_ROW\_S

  lsr.w      #3,d1                        ; offset\_x = x / 8

  ext.l      d1

  lea        tileset,a0                   ; source image address

  add.l      d0,a0                        ; add y\_offset

  add.l      d1,a0                        ; add x\_offset

  moveq      #N\_PLANES-1,d7

  bsr        wait\_blitter

  move.w     #$ffff,BLTAFWM(a5)           ; don't use mask

  move.w     #$ffff,BLTALWM(a5)

  move.w     #$09f0,BLTCON0(a5)           ; enable channels A,D

                                          ; logical function = $f0, D = A

  move.w     #0,BLTCON1(a5)

  move.w     #(TILESET\_WIDTH-TILE\_WIDTH)/8,BLTAMOD(a5)    ; A channel modulus

  move.w     #(DISPLAY\_WIDTH-TILE\_WIDTH)/8,BLTDMOD(a5)    ; D channel modulus

.loop:

  bsr        wait\_blitter

  move.l     a0,BLTAPT(a5)                ; source address

  move.l     a1,BLTDPT(a5)                ; destination address

  move.w     #64\*64+4,BLTSIZE(a5)         ; blit size: 64 rows for 4 word

  add.l      #TILESET\_PLANE\_SZ,a0         ; advances to the next plane

  add.l      #DISPLAY\_PLANE\_SZ,a1

  dbra       d7,.loop

  bsr        wait\_blitter

  movem.l    (sp)+,d0-a6                  ; restores registers from the stack

  rts

This routine must be called in the main, right after init\_bplpointers. We draw the tiles of index 2,3 side by side.

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; MAIN PROGRAM

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

main:

  nop

  nop

  bsr       take\_system         ; takes the control of Amiga's hardware

  move.l    #screen,d0          ; address of screen in d0

  bsr       init\_bplpointers    ; initializes bitplane pointers to our image

  lea       screen,a1           ; address where draw the tile

  move.w    #0,d2               ; x position

  move.w    #256-64,d3          ; y position

  move.w    #2,d0               ; tile index

  bsr       draw\_tile

  move.w    #64,d2              ; x position

  move.w    #256-64,d3          ; y position

  move.w    #3,d0               ; tile index

  bsr       draw\_tile

On the screen we will have the following output:

Immagine che contiene testo, schermata, design

Descrizione generata automaticamente

## Drawing a column of tiles

Now we need to write a routine that draws a column of tiles, starting from the top.

The routine calls draw\_tile in a loop. At each iteration the y position is incremented and the next line of the tilemap is read. The complete code is the following:

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; Draws a column of 3 tiles.

;

; parameters:

; d0.w - map column

; d2.w - x position (multiple of 16)

; a1   - address where draw the tile

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

draw\_tile\_column:

  movem.l    d0-a6,-(sp)

  ; calculates the tilemap address from which to read the tile index

  lea        map,a0

  lsl.w      #1,d0                   ; offset\_x = map\_column \* 2

  ext.l      d0

  add.l      d0,a0

  moveq      #3-1,d7                 ; number or tilemap rows - 1

  move.w     #0,d3                   ; y position

.loop:

  move.w     (a0),d0                 ; tile index

  bsr        draw\_tile

  add.w      #TILE\_HEIGHT,d3         ; increment y position

  add.l      #TILEMAP\_ROW\_SIZE,a0    ; move to the next row of the tilemap

  dbra       d7,.loop

  movem.l    (sp)+,d0-a6

  rts

To test this routine, add the following code in main, after init\_bplpointers:

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; MAIN PROGRAM

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

main:

  nop

  nop

  bsr       take\_system         ; takes the control of Amiga's hardware

  move.l    #screen,d0          ; address of screen in d0

  bsr       init\_bplpointers    ; initializes bitplane pointers to our image

  lea       screen,a1           ; address where draw the tile

  move.w    #11,d0              ; map column to draw

  move.w    #0,d2               ; x position (multiple of 16)

  bsr       draw\_tile\_column

The result is shown in the following figure:

Immagine che contiene testo, schermata

Descrizione generata automaticamente

## Let's fill the screen with tiles

Now let's try reusing the draw\_tile\_column routine to fill the entire screen. We write a routine called fill\_screen\_with\_tiles that takes as input the column of the map from which to start drawing the tiles. The routine calls in a loop the draw\_tile\_column. After each iteration the column of the map to draw and the x position are incremented. The code is the following:

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; Fills the screen with tiles.

;

; parameters:

; d0.w - map column from which to start drawing tiles

; a1   - address where draw the tile

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

fill\_screen\_with\_tiles:

       movem.l    d0-a6,-(sp)

       moveq      #5-1,d7             ; number of tile columns - 1 to draw

       move.w     #0,d2               ; position x

.loop  bsr        draw\_tile\_column

       add.w      #1,d0               ; increments map column

       add.w      #64,d2              ; increases position x

       dbra       d7,.loop

       movem.l    (sp)+,d0-a6

       rts

To test this routine, add the following statements to main:

main:

  nop

  nop

  bsr       take\_system               ; takes the control of Amiga's hardware

  move.l    #screen,d0                ; address of screen in d0

  bsr       init\_bplpointers          ; initializes bitplane pointers to our

; image

  lea       screen,a1                 ; address where draw the tile

  move.w    #11,d0                    ; map column to start drawing from

  bsr       fill\_screen\_with\_tiles

The result of running this routine is a screen that reproduces a piece of the game map, shown in the following figure. As you can see, we are starting to visualize something interesting and that looks more and more like a video game!

Immagine che contiene testo, schermata, Rettangolo, design

Descrizione generata automaticamente

You can find the complete source code at the following url:

<https://github.com/stefanocoppi/amiga_game_prog_assembly/tree/main/chapter9>

# 10. Scrolling background

In this chapter, we'll look at how to implement a scrolling background. First, we will see a horizontal scrolling solution based on the Blitter, then one based on the hardware scrolling of the Amiga playfield. Finally, we will see how to implement vertical scrolling, for purely educational purposes, since our game uses horizontal scrolling.

## Blitter-based scrolling

The main motivation for using this scrolling technique is that it is easy to combine scrolling with sprite drawing.

In fact, since we completely redraw the background at each frame, we can simply draw the sprites over the background without worrying about restoring the background. Also, managing hardware scrolling together with double buffering is more complex.

Now let's see how Blitter-based scrolling works. Referring to the following figure, we have an image of the entire game level, several screens long. The rectangle highlighted in blue represents the camera, which has the same size as the screen, and shows a part of the level. By moving the camera to the right, we get the level to scroll. To display the part shown by the camera on the screen, we need to blit the shown portion of the level on the screen.

Immagine che contiene diagramma, linea, Rettangolo, schermata

Descrizione generata automaticamente

We saw in the last chapter that we don't have enough memory to store the entire level as a single image. To overcome this problem we used a tilemap to represent the level.

So how do we achieve scrolling using tilemap?

Imagine we have an area the size of two screens, over which the camera can scroll. Refer to the figure below.

Immagine che contiene diagramma, linea, Parallelo, Piano

Descrizione generata automaticamente

We scroll the camera one pixel at a time to the right, as in the following figure.

Immagine che contiene diagramma, linea, Parallelo, Piano

Descrizione generata automaticamente

When the camera reaches an x-position that is a multiple of 16, we draw two columns of tiles at the left and right ends of the camera, taking the tiles from the tilemap (see the following figure).

Immagine che contiene diagramma, linea, Parallelo, Piano

Descrizione generata automaticamente

We continue scrolling and repeat the same steps (see following figure).

Immagine che contiene diagramma, linea, Parallelo, Piano

Descrizione generata automaticamente

When the camera reaches the edge of the background (see following figure), we notice that the left and right parts of the background have the same content.

Immagine che contiene diagramma, linea, Parallelo, Piano

Descrizione generata automaticamente

We can then reposition the camera to the initial position (see the following figure) without changing the image shown.

Immagine che contiene diagramma, linea, Parallelo, Piano

Descrizione generata automaticamente

At this point we can start over again.

Immagine che contiene diagramma, linea, Parallelo, Piano

Descrizione generata automaticamente

In this way we can scroll a tilemap of arbitrary length, using only a surface equal to twice the screen.

Obviously, to display the portion of the background shown by the camera we use the Blitter.

## Background initialization

Now it's time to put into practice the theoretical notions seen previously. For the example program we reuse part of the source code from the last chapter.

We need to allocate memory for the background surface, 2 times the width of the display + 2\*64 pixels for the two columns of tiles at the edges. We use the BSS\_C segment that can only contain zero-value data, allocated in chip memory. We define constants for the dimensions of this surface. We also define the map\_ptr variable that contains the index of the next column of the tilemap to be drawn.

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; CONSTANTS

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

...

; background

BGND\_WIDTH      equ 2\*DISPLAY\_WIDTH+2\*TILE\_WIDTH

BGND\_HEIGHT     equ 192

BGND\_PLANE\_SIZE equ BGND\_HEIGHT\*(BGND\_WIDTH/8)

BGND\_ROW\_SIZE   equ (BGND\_WIDTH/8)

; scroll

VIEWPORT\_HEIGHT equ 192

VIEWPORT\_WIDTH  equ 320

SCROLL\_SPEED    equ 1

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; VARIABLES

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

...

map\_ptr       dc.w       0                 ; current map column

bgnd\_x        dc.w       0                 ; current x coordinate

; of camera into background surface

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; BSS DATA

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

              SECTION    bss\_data,BSS\_C

dbuffer1      ds.b       (DISPLAY\_PLANE\_SZ\*N\_PLANES)    ; display buffers used

; for double buffering

dbuffer2      ds.b       (DISPLAY\_PLANE\_SZ\*N\_PLANES)

bgnd\_surface  ds.b       (BGND\_PLANE\_SIZE\*N\_PLANES)     ; invisible surface

; used for scrolling

; background

              END

The background surface must be initialized with the first portion of the tilemap. To do this, we create the init\_background routine that reuses the draw\_tile\_column routine seen in last chapter, with some slight modifications to adapt it to draw on the background surface.

We first draw the part that will be visible in the camera, composed of 5 columns 64 pixels wide each, for a total of 320 pixels.

Then we draw the column to the right of the camera, which contains the part of the map that will become visible during the scroll and we repeat the same column to the left of the camera, invisible, but which will serve to obtain the same content in the two parts of the background surface.

Below is the code:

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; Initializes the background, copying the initial part of the level map.

;

; parameters:

; d0.w - map column from which to start drawing tiles

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

init\_background:

       movem.l    d0-a6,-(sp)

; initializes the part that will be visible in the display window

       moveq      #5-1,d7                         ; number of tile columns - 1

; to draw

       lea        bgnd\_surface,a1                 ; address where draw the tile

       move.w     #TILE\_WIDTH,d2                  ; position x

.loop  bsr        draw\_tile\_column

       add.w      #1,d0                           ; increment map column

       add.w      #1,map\_ptr

       add.w      #TILE\_WIDTH,d2                  ; increase position x

       dbra       d7,.loop

; draws the column to the left of the display window

       add.w      #1,d0                           ; map column

       add.w      #1,map\_ptr

       move.w     #0,d2                           ; x position

       lea        bgnd\_surface,a1

       bsr        draw\_tile\_column

; draws the column to the right of the display window

       move.w     #DISPLAY\_WIDTH+TILE\_WIDTH,d2    ; x position

       lea        bgnd\_surface,a1

       bsr        draw\_tile\_column

       movem.l    (sp)+,d0-a6

       rts

The init\_background routine uses the draw\_tile\_column routine to draw a column of tiles. The latter uses the draw\_tile routine to draw a tile. Compared to the version seen in the last chapter, now draw\_tile must draw not directly on the visible screen, but on a surface in memory, called bgnd\_surface and having a width greater than that of the visible screen. It is necessary to make some changes to adapt it to draw on this surface:

* in the calculation of the source address, we must use BGND\_ROW\_SIZE, or the size in bytes of a background row to calculate the y offset.
* the module of the D channel becomes the following: (BGND\_WIDTH-TILE\_WIDTH)/8
* to advance to the next plane, we must add BGND\_PLANE\_SIZE to a1

The updated code is the following:

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; Draws a 64x64 pixel tile using Blitter.

;

; parameters:

; d0.w - tile index

; d2.w - x position of the screen where the tile will be drawn (multiple of 16)

; d3.w - y position of the screen where the tile will be drawn

; a1   - address where draw the tile

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

draw\_tile:

  movem.l    d0-a6,-(sp)                                  ; saves registers

; into the stack

; calculates the screen address where to draw the tile

  mulu       #BGND\_ROW\_SIZE,d3                            ; y\_offset = y \*

; BGND\_ROW\_SIZE

  lsr.w      #3,d2                                        ; x\_offset = x / 8

  ext.l      d2

  add.l      d3,a1                      ; sums offsets to a1

  add.l      d2,a1

; calculates row and column of tile in tileset starting from index

  ext.l      d0                                           ; extends d0 to a long because the destination operand if divu must be long

  divu       #TILESET\_COLS,d0           ; tile\_index / TILESET\_COLS

  swap       d0

  move.w     d0,d1                      ; the remainder indicates the tile column

  swap       d0                         ; the quotient indicates the tile row

; calculates the x,y coordinates of the tile in the tileset

  lsl.w      #6,d0                      ; y = row \* 64

  lsl.w      #6,d1                      ; x = column \* 64

; calculates the offset to add to a0 to get the address of the source image

  mulu       #TILESET\_ROW\_SIZE,d0       ; offset\_y = y \* TILESET\_ROW\_SIZE

  lsr.w      #3,d1                      ; offset\_x = x / 8

  ext.l      d1

  lea        tileset,a0                 ; source image address

  add.l      d0,a0                      ; add y\_offset

  add.l      d1,a0                      ; add x\_offset

  moveq      #N\_PLANES-1,d7

  bsr        wait\_blitter

  move.w     #$ffff,BLTAFWM(a5)         ; don't use mask

  move.w     #$ffff,BLTALWM(a5)

  move.w     #$09f0,BLTCON0(a5)         ; enable channels A,D

                                       ; logical function = $f0, D = A

  move.w     #0,BLTCON1(a5)

  move.w     #(TILESET\_WIDTH-TILE\_WIDTH)/8,BLTAMOD(a5)    ; A channel modulus

  move.w     #(BGND\_WIDTH-TILE\_WIDTH)/8,BLTDMOD(a5)       ; D channel modulus

.loop:

  bsr        wait\_blitter

  move.l     a0,BLTAPT(a5)              ; source address

  move.l     a1,BLTDPT(a5)             ; destination address

  move.w     #64\*64+4,BLTSIZE(a5)       ; blit size: 64 rows for 4 word

  add.l      #TILESET\_PLANE\_SZ,a0       ; advances to the next plane

  add.l      #BGND\_PLANE\_SIZE,a1

  dbra       d7,.loop

  bsr        wait\_blitter

  movem.l    (sp)+,d0-a6                ; restore registers from the stack

  rts

## Background Drawing Using Blitter

Now we need to create a routine that allows us to implement the camera seen in the previous figure, that is, one that can display a portion of the background surface, where we have drawn a part of the tilemap that represents the game level. We call this routine draw\_background and it takes as parameters:

* x coordinate of the camera in the background surface
* address of the buffer in which to draw

The routine uses the Blitter to copy a rectangular window of dimensions (VIEWPORT\_WIDTH+16) \* VIEWPORT\_HEIGHT from the background surface to the screen.

Since the x coordinate can be any and not necessarily a multiple of 16, we need to use the shift of the Blitter channel A.

We calculate the source address to which the channel A should point as seen in part 11, that is, by adding a horizontal offset to the base address of bgnd\_surface. The vertical offset is zero because we always start from y=0. The horizontal offset is obtained by dividing x by 8 and rounding to an even address.

To calculate the shift value, we select the first 4 bits of x using a mask, which represent the shift value S. Since we want to make a shift to the left, we must first initialize the shift value to the maximum and then decrement it. To do this, we calculate the shift value as 15-S. This value must be placed in the 4 most significant bits of the BLTCON0 register. To position it correctly we need to perform 2 shifts of 8 and 4 positions to the left, for a total of 12.

For the rest the routine makes a copy from channel A to channel D via Blitter. Pay attention to the values ​​of the modules, otherwise you will see a corrupted image.

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; Draws the background, copying it from background\_surface via Blitter.

;

; parameters:

;

; d0.w - x position of the part of background to draw

; a1   - buffer where to draw

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

draw\_background:

  movem.l    d0-a6,-(sp)

  moveq      #N\_PLANES-1,d7

  lea        bgnd\_surface,a0

; calculates the source image address

  move.w     d0,d2               ; copy of x

  asr.w      #3,d0               ; offset\_x = x/8

  and.w      #$fffe,d0           ; rounds to even addresses

  ext.l      d0

; calculates the shift value

  add.l      d0,a0               ; address of image to copy

  and.w      #$000f,d2           ; selects the first 4 bits, which correspond

; to the shift

  move.w     #$f,d3              ; since we want a left scroll,

  sub.w      d2,d3               ; we need to decrement the value of scroll,

; i.e. $f-scroll

  lsl.w      #8,d3               ; moves the 4 shift bits to the position they

; occupy in BLTCON0

  lsl.w      #4,d3

  or.w       #$09f0,d3           ; inserts the 4 bits into the value to be

; assigned to BLTCON0

.planeloop:

  bsr        wait\_blitter

  move.l     a0,BLTAPT(a5)       ; channel A points to background surface

  move.l     a1,BLTDPT(a5)       ; channel D points to draw buffer

  move.w     #$ffff,BLTAFWM(a5)  ; no first word mask

  move.w     #$0000,BLTALWM(a5)  ; masks last word

  move.w     d3,BLTCON0(a5)

  move.w     #0,BLTCON1(a5)

  move.w     #(BGND\_WIDTH-VIEWPORT\_WIDTH-16)/8,BLTAMOD(a5)

  move.w     #(DISPLAY\_WIDTH-VIEWPORT\_WIDTH-16)/8,BLTDMOD(a5)

  move.w     #VIEWPORT\_HEIGHT<<6+(VIEWPORT\_WIDTH/16)+1,BLTSIZE(a5)

  move.l     a0,d0

  add.l      #BGND\_PLANE\_SIZE,d0  ; points a0 to the next plane

  move.l     d0,a0

  move.l     a1,d0

  add.l      #DISPLAY\_PLANE\_SZ,d0 ; points a1 to the next plane

  move.l     d0,a1

  dbra       d7,.planeloop

  movem.l    (sp)+,d0-a6

  rts

## Vertical Blank

In a previous chapter we saw that the electron beam forms the video image in lines, starting from the top. A complete scan takes 1/50 of a second in the European PAL format. When the electron beam reaches the end of the screen, the electron beam is turned off and repositions itself at the top. This instant is called vertical blanking, since the electron beam is turned off (blank). We can use this signal to time our video game, in fact the vertical blank occurs every 1/50 of a second. Let's start writing a routine that waits for the electron brush to reach a given line. The position of the electron beam is reported in bits 8-16 of the VPOSR register. We use the and with the mask $1ff00 to select only bits 8-16. The value of the line to wait for is shifted to the left by 8 to align it with bits 8-16 of VPOSR. Then a comparison is made, and the loop is repeated until the comparison is zero.

The wait\_vblank routine, which waits for the vertical blank, uses the wait\_vline.

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; Waits for the electron beam to reach a given line.

;

; parameters:

; d2.l - line

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

wait\_vline:

  movem.l    d0-a6,-(sp)     ; saves registers into the stack

  lsl.l      #8,d2

  move.l     #$1ff00,d1

wait:

  move.l     VPOSR(a5),d0

  and.l      d1,d0

  cmp.l      d2,d0

  bne.s      wait

  movem.l    (sp)+,d0-a6     ; restores registers from the stack

  rts

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; Waits for the vertical blank

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

wait\_vblank:

  movem.l    d0-a6,-(sp)     ; saves registers into the stack

  move.l     #304,d2         ; line to wait: 304 236

  bsr        wait\_vline

  movem.l    (sp)+,d0-a6     ; restores registers from the stack

  rts

The wait\_vblank should be called in the main loop, so that each iteration of the loop occurs every 1/50 of a second.

## Background scrolling

To scroll the camera to the right, we create the routine scroll\_background. This first draws the background framed by the camera, using the routine draw\_background seen previously.

Every 16 pixels it draws two columns of new tiles, on the non-visible sides of the camera. The camera position is incremented by 16 and the tilemap pointer is incremented. If the end of the tilemap has been reached, the routine immediately returns to stop the scroll.

If the camera has reached the end of the background, it repositions itself to the beginning. Otherwise, the x position of the camera is incremented.

The constant SCROLL\_SPEED  defines the speed of the camera movement in pixels / 50th of a second.

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; Scrolls the background to the left.

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

scroll\_background:

         movem.l    d0-a6,-(sp)

         move.w     bgnd\_x,d0   ; x position of the part of

; background to draw

         move.l     draw\_buffer,a1         ; buffer where to draw

         bsr        draw\_background

         ext.l      d0                 ; every TILE\_WIDTH (64) pixels

; draws a new column

         divu       #TILE\_WIDTH,d0

         swap       d0

         tst.w      d0               ; remainder of bgnd\_x/TILE\_WIDTH is

; zero?

         beq        .draw\_new\_column

         bra        .check\_bgnd\_end

.draw\_new\_column:

         add.w      #1,map\_ptr

         cmp.w      #TILEMAP\_WIDTH,map\_ptr  ; end of map?

         bge        .return

         move.w     map\_ptr,d0              ; map column

         move.w     bgnd\_x,d2               ; x position = bgnd\_x - TILE\_WIDTH

         sub.w      #TILE\_WIDTH,d2

         lea        bgnd\_surface,a1

         bsr        draw\_tile\_column        ; draws the column to the left of

; the viewport

         move.w     bgnd\_x,d2               ; x position = bgnd\_x +

; VIEWPORT\_WIDTH

         add.w      #VIEWPORT\_WIDTH,d2

         lea        bgnd\_surface,a1

         bsr        draw\_tile\_column       ; draws the column to the right of

; the viewport

.check\_bgnd\_end:

         cmp.w      #TILE\_WIDTH+VIEWPORT\_WIDTH,bgnd\_x    ; end of background

; surface?

         ble        .incr\_x

         move.w     #SCROLL\_SPEED,bgnd\_x    ; resets x position of the part of

; background to draw

         bra        .return

.incr\_x  add.w      #SCROLL\_SPEED,bgnd\_x    ; increases x position of the part

; of background to draw

.return  movem.l    (sp)+,d0-a6

         rts

The scroll\_background  routine must be called in the main loop, just after the wait\_vblank. This way we are sure that it will be executed every 1/50 of a second.

mainloop:

  bsr      wait\_vblank          ; waits for vertical blank

  bsr      swap\_buffers

  bsr      scroll\_background

  btst     #6,CIAAPRA           ; left mouse button pressed?

  bne.s    mainloop             ; if not, repeats the loop

## Hide the first 16 pixels to remove noise

If you try to run the program now, you will notice that at the far left of the screen there are black pixels that go back and forth. They are due to the shift we make for scrolling. To avoid seeing this part, we can move the beginning of the display window 16 pixels forward, acting on the DIWSTRT register, as shown below:

dc.w       DIWSTRT,$2c91            ; display window start at ($91,$2c)

## Double buffering

If we make changes to the screen while the electron beam is drawing it, we see some noise in the image. To avoid this, the double buffering technique was invented.

To implement it, we create 2 video buffers, in the zeroed data segment, in CHIP RAM. We define two pointers to the video buffers, called view\_buffer and draw\_buffer. For all drawing operations we use the draw\_buffer, which is not visible. The view\_buffer instead is visible, in fact the bitplane pointers will point to it. At each vertical blank we will call the routine swap\_buffers, which will exchange the two pointers draw\_buffer and view\_buffer, causing the draw\_buffer to be displayed. The bitplane pointers are made to point to view\_buffer.

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; VARIABLES

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

view\_buffer  dc.l       dbuffer1              ; buffer displayed on screen

draw\_buffer  dc.l       dbuffer2              ; drawing buffer (not visible)

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; BSS DATA

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

             SECTION    bss\_data,BSS\_C

dbuffer1     ds.b       (DISPLAY\_PLANE\_SZ\*N\_PLANES)    ; display buffers used

; for double buffering

dbuffer2     ds.b       (DISPLAY\_PLANE\_SZ\*N\_PLANES)

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; Swaps video buffers, causing draw\_buffer to be displayed.

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

swap\_buffers:

  movem.l    d0-a6,-(sp)                ; saves registers into the stack

  move.l     draw\_buffer,d0             ; swaps the values ​​of draw\_buffer and

; view\_buffer

  move.l     view\_buffer,draw\_buffer

  move.l     d0,view\_buffer

  lea        bplpointers,a1             ; sets the bitplane pointers to the

; view\_buffer

  moveq      #N\_PLANES-1,d1

.loop:

  move.w     d0,6(a1)                   ; copies low word

  swap       d0                         ; swaps low and high word of d0

  move.w     d0,2(a1)                   ; copies high word

  swap       d0                         ; resets d0 to the initial condition

  add.l      #DISPLAY\_PLANE\_SZ,d0       ; points to the next bitplane

  add.l      #8,a1                      ; points to next bplpointer

  dbra       d1,.loop                   ; repeats the loop for all planes

  movem.l    (sp)+,d0-a6                ; restores registers from the stack

  rts

You can find the full source code at the following url:

<https://github.com/stefanocoppi/amiga_game_prog_assembly/tree/main/chapter10>

Running the program we created will scroll the entire level. You can adjust the scrolling speed by adjusting the constant SCROLL\_SPEED.

Immagine che contiene arte, Motivo, modello, design

Descrizione generata automaticamente

## Hardware scrolling

Now we want to implement the left scrolling of the background in a more optimized way, using the hardware scrolling feature provided by the Amiga.

The Amiga hardware provides the ability to scroll the playfield, i.e. the visible screen, up to a maximum of 15 pixels. The amount of the scrolling must be entered in register BPLCON1 ($dff102). More precisely, bits 0-3 contain the scroll for odd bitplanes, while bits 4-7 contain the scroll for even bitplanes.

How do you scroll the screen for more than 15 pixels? Once you reach 15 pixels, you must increase the bitplane pointers by 2 and reset the scroll value, so that you can scroll another 15 pixels and so on.

To scroll a tilemap, you can use a technique like the one seen earlier with the Blitter. You must create a playfield with a width of 2\*viewport\_width + 2\*tile\_width. You place the viewport tile\_width (64) pixels from the beginning of the playfield. Let's initialize the viewport with the first 5 columns of the tilemap. Then you draw the sixth column of the tilemap at either end of the viewport, as shown in the following picture. The bitplane pointer must be initialized at the beginning of the viewport, i.e. tile\_width pixels from the beginning of the playfield. Scroll values in BPLCON1 must be initialized to the maximum, i.e. $f. This is because to make a scroll to the left, we have to decrement this value to 0.

Immagine che contiene testo, linea, schermata, diagramma

Descrizione generata automaticamente

At this point we can act on the BPLCON1 register to decrement the scroll value until it brings it to 0. This causes the viewport to shift to the left by 1 pixel at a time. When the value of the scroll reaches 0, we need to move the Bitplane Pointer by 16 pixels, or 2 bytes. We also need to reset the scroll value to $f.

Immagine che contiene linea, testo, diagramma, schermata

Descrizione generata automaticamente

Every TILE\_WIDTH, or 64 pixels of displacement, we need to copy a new tilemap column at the ends of the viewport, as in the following figure:

Immagine che contiene testo, linea, diagramma, schermata

Descrizione generata automaticamente

Repeat the scrolling process until the viewport reaches the right end of the playfield minus the distance of one tile\_width. In this situation, we observe that the two halves of the playfield have identical content.

Immagine che contiene testo, linea, numero, Carattere

Descrizione generata automaticamente

You can then return the viewport to the beginning of the playfield, without changing the image on the screen for the player. From this point on, you can repeat the whole process, until you reach the end of the tilemap.

Immagine che contiene linea, diagramma, Diagramma, testo

Descrizione generata automaticamente

## Implementing hardware scrolling

Let's start by allocating memory for the playfield , larger than display window, in the chip ram zeroed data segment:

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; BSS DATA

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

              SECTION    bss\_data,BSS\_C

bgnd\_surface  ds.b       (BGND\_PLANE\_SIZE\*N\_PLANES)    ; surface used for

; scrolling background

Since we have a playfield larger than the display window, we need to set the odd and even bitplane modules. We also label with "scrollx" the value of BPLCON1, which contains the amount of scrolling. We initialize it at $00FF since to scroll to the left we must start from the maximum value and decrement it by 1.

copperlist:

...

         dc.w    BPLCON1

scrollx  dc.w    $00ff           ; bits 0-3 and 4-7 scroll value

         dc.w    BPLCON2,0

         dc.w    BPL1MOD,(BGND\_WIDTH-VIEWPORT\_WIDTH)/8

         dc.w    BPL2MOD,(BGND\_WIDTH-VIEWPORT\_WIDTH)/8

We also need to initialize the bitplane pointers to 64 pixels (8 bytes) from the beginning of the bgnd\_surface. To do this, we use the init\_bplpointers routine, in main. We also initialize the background, draw the first 5 columns of the tilemap in the viewport and draw the sixth column on the sides of the viewport itself. These operations are carried out by the init\_background routine.

main:

  nop

  nop

  bsr       take\_system             ; takes the control of Amiga's hardware

  move.l    #bgnd\_surface+8,d0    ; address of visible screen buffer

  move.l    #BGND\_PLANE\_SIZE,d1

  bsr       init\_bplpointers        ; initializes bitplane pointers to our

; image

  move.w    map\_ptr,d0

  bsr       init\_background

  move.w    #TILE\_WIDTH,bgnd\_x      ; x position of the part of background to

; draw

Running the program now, you will see the beginning of the tilemap displayed.

To make the tilemap scroll, we need to write a new routine scroll\_background. If the viewport's move value is a multiple of 64, draw a new column of tiles at the left and right ends of the viewport.

Then set the amount of the scroll in the BPLCON1 register of the copperlist. If the amount of the scroll is zero, then refresh the bitplane pointers and reset the value of the scroll to $00ff.

If the viewport shift has reached the maximum value of TILE\_WIDTH+VIEWPORT\_WIDTH, then it returns the bitplane pointers to the initial position and resets the scroll value again to $00ff.

Finally, increments the value of the viewport shift by 1, bgnd\_x.

Below is the complete code.

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; Scrolls the background to the left.

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

scroll\_background:

         movem.l    d0-a6,-(sp)

         move.w     bgnd\_x,d0     ; x position of the

; part of background

; to draw

         tst.w      d0

         beq        .set\_scroll

         ext.l      d0                                   ; every 64 pixels

; draws a new column

         divu       #TILE\_WIDTH,d0

         swap       d0

         tst.w      d0                                   ; remainder of

; bgnd\_x/TILE\_WIDTH is

; zero?

         beq        .draw\_new\_column                     ; if yes, draws new

; tile columns at the

; sides of viewport

         bra        .set\_scroll

.draw\_new\_column:

         add.w      #1,map\_ptr

         cmp.w      #TILEMAP\_WIDTH,map\_ptr               ; end of map?

         bge        .return

         move.w     map\_ptr,d0                           ; map column

         move.w     bgnd\_x,d2                            ; x position = bgnd\_x

; - TILE\_WIDTH

         sub.w      #TILE\_WIDTH,d2

         lea        bgnd\_surface,a1

         bsr        draw\_tile\_column                     ; draws the column to

; the left of the

; viewport

         move.w     bgnd\_x,d2                            ; x position = bgnd\_x

; + VIEWPORT\_WIDTH

         add.w      #VIEWPORT\_WIDTH,d2

         lea        bgnd\_surface,a1

         bsr        draw\_tile\_column                     ; draws the column to

; the right of the

; viewport

.set\_scroll:

         move.w     bgnd\_x,d0

         and.w      #$000f,d0                            ; selects the first 4

; bits, which

; correspond to the

; shift

         move.w     #$f,d1                               ; since we want a left

; scroll,

         sub.w      d0,d1                                ; we need to decrement

; the value of scroll,

; i.e. $f-scroll

         move.w     d1,d2                                ; copy

         move.w     d1,d0                                ; copy

         lsl.w      #4,d0

         or.w       d0,d1                                ; value of bits 0-3

; and 4-7 of BPLCON1

         move.w     d1,scrollx                           ; sets the BPLCON1

; value for scrolling

         tst.w      d2                                   ; scroll = 0?

         beq        .update\_bplptr                       ; yes, update bitplane

; pointers

         bra        .check\_bgnd\_end

.update\_bplptr:

         move.w     bgnd\_x,d1

         asr.w      #3,d1                                ; offset\_x = bgnd\_x/8

         and.w      #$fffe,d1                            ; rounds to even

; addresses

         ext.l      d1                                   ; extends to long

         move.l     #bgnd\_surface,d0

         add.l      d1,d0                                ; adds offset\_x

         move.l     #BGND\_PLANE\_SIZE,d1

         bsr        init\_bplpointers

         move.w     #$00ff,scrollx                       ; resets scroll value

.check\_bgnd\_end:

         cmp.w      #TILE\_WIDTH+VIEWPORT\_WIDTH,bgnd\_x    ; end of background

; surface?

         ble        .incr\_x

         move.w     #0,bgnd\_x                            ; resets x position of

; the part of

; background to draw

         move.l     #bgnd\_surface-2,d0

         move.l     #BGND\_PLANE\_SIZE,d1

         bsr        init\_bplpointers

         move.w     #$00ff,scrollx                       ; resets scroll value

         bra        .return

.incr\_x:

         add.w      #1,bgnd\_x                            ; increases x position

; of the part of

; background to draw

.return  movem.l    (sp)+,d0-a6

         rts

The scroll\_background routine must be invoked in the main loop. It must also be synchronized with the vertical blank signal, so that it is recalled 50 times per second.

mainloop:

  bsr      wait\_vblank          ; waits for vertical blank

  bsr      scroll\_background

  btst     #6,CIAAPRA           ; left mouse button pressed?

  bne.s    mainloop             ; if not, repeats the loop

When you run the program, you will see image noise on the left side of the screen. To remove it, simply change the DDFSTRT register value, making the data fetch start 32 pixels before the display window starts. This will no longer make the disturbing part visible. However, we must reduce the modules by 4 bytes (32 pixels).

copperlist:

...

  dc.w    DDFSTRT,$28                                ; display data fetch start at $28 to hide scrolling artifacts

...

  dc.w    BPL1MOD,(BGND\_WIDTH-VIEWPORT\_WIDTH)/8-4    ; -4 because we fetch 32 more pixels

  dc.w    BPL2MOD,(BGND\_WIDTH-VIEWPORT\_WIDTH)/8-4

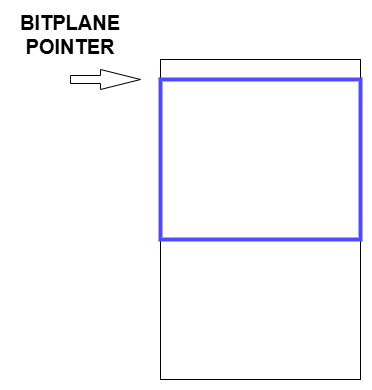
You can find the complete code at the following url:

<https://github.com/stefanocoppi/amiga_game_prog_assembly/tree/main/chapter10B>

## Vertical scrolling

Now let's see how to implement vertical scrolling of a tilemap. By learning this technique, you can create vertically scrolling shoot'em ups like the famous Xenon!

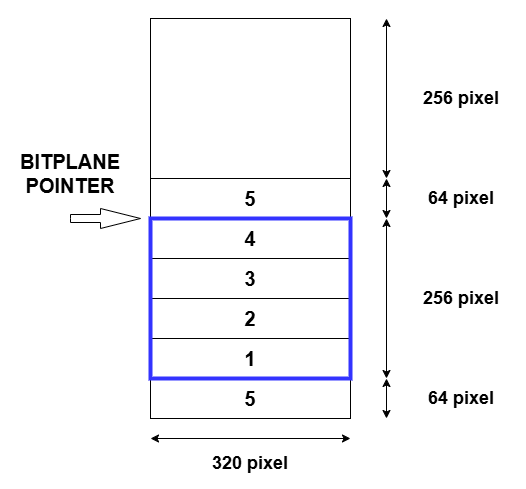
To achieve vertical scrolling on the Amiga, you must first have a playfield with a height greater than the viewport. To scroll the viewport, simply change the bitplane pointers. Incrementing them by a number of bytes equal to the size of a line will move the display window one line down, while decrementing them by the same amount will move the display window one line up. The following picture illustrates the concept.



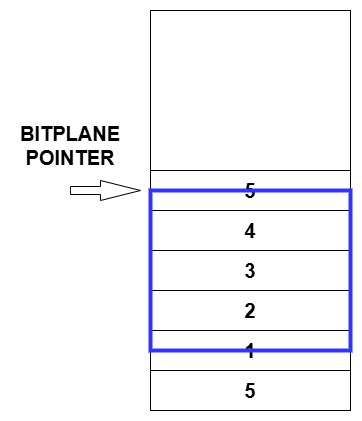
To achieve vertical scrolling down of a tilemap, we use a procedure similar to that seen for horizontal scrolling. Let's start with the situation shown in the following figure. We have a playfield with a width of 320 pixels while the height is:

2 \* (viewport height + tile\_height) = 2 \* (256 + 64) = 640 pixels

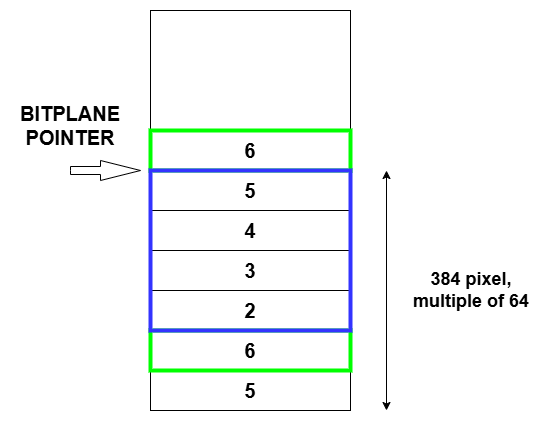
We place the viewport at a distance of (256 + 64) = 320 pixels from the bottom of the playfield, acting on the bitplanes pointers. Let's initialize the viewport by drawing the first 4 lines of the tilemap, numbered from 1 to 4 in the figure. Now let's draw row 5 of the tilemap at the top and bottom of the viewport.



Now let's move the viewport upwards, one line at a time, decrementing the bitplanes pointers, as shown in the following figure.



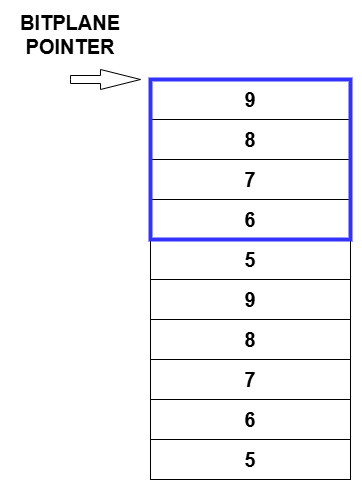
Whenever the viewport move is a multiple of tile\_height, i.e. 64, we need to draw a new row of the tilemap at the top and bottom of the viewport, as shown in the following figure.



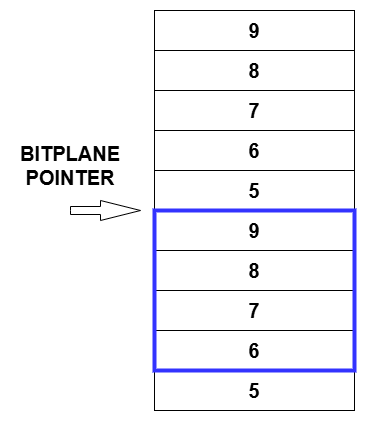
When the viewport moves relative to the bottom of the playfield will be:

2 \* viewport\_height + tile\_height = 2 \* 256 + 64 = 576 pixel

We will have the situation shown in the following figure. We notice that the two halves of the playfield have identical content. So, we can move the viewport down without changing the content and therefore without the player noticing.



Then we move the viewport down so that we can start the whole process again, as shown in the figure below.



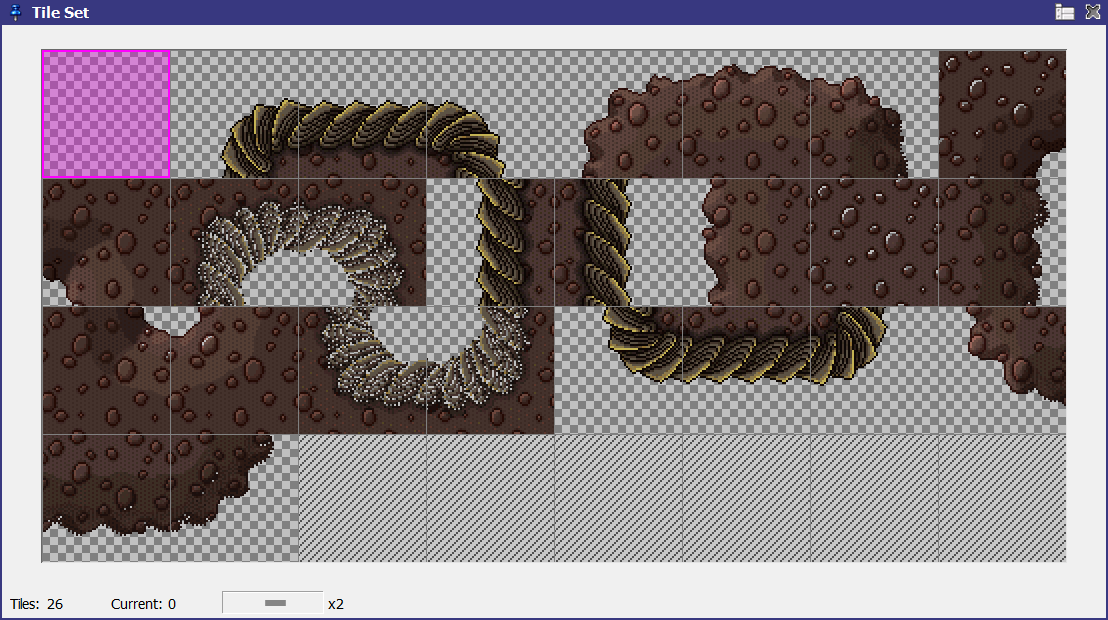
## Implementing Vertical Scrolling

Let's start creating a tilemap suitable for vertical scrolling. Let's use the same process used in chapter 9 for the horizontal tilemap.

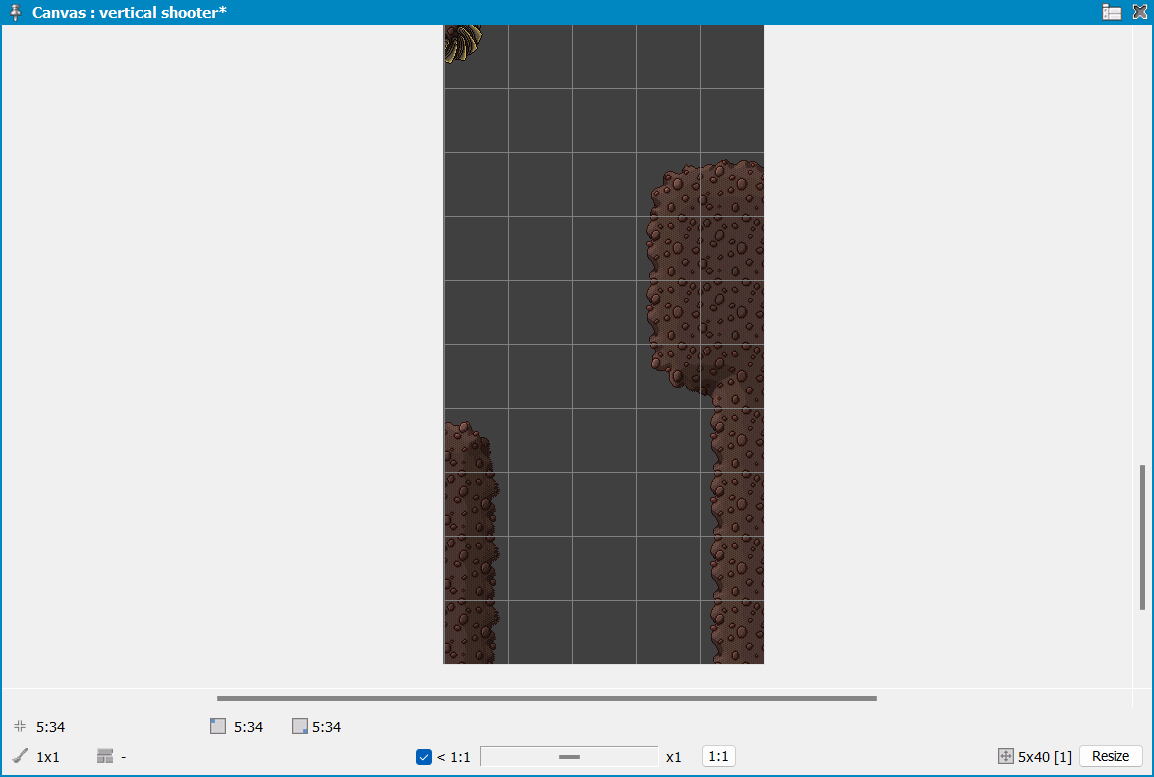
The dimensions of the tilemap are:

* Width: 320 pixels
* Height: 2560 pixels, or 10 screens

Let's reuse the tileset from chapter 9, shown in the following image:



Have fun creating the map, an example can be seen in the following figure.



You can find the complete tilemap project, in Pro Motion NG format, at the following url:

<https://github.com/stefanocoppi/amiga_game_prog_assembly/blob/main/chapter10C/gfx/vertical%20shooter%20map.pmp>

Export the tilemap and tileset as seen in chapter 9. You will get two files:

1. PNG image containing the tileset, vshooter\_tiles.png
2. text file containing the tilemap, vshooter\_map.i

At this point we need to convert the png image to raw format and extract the palette to copperlist format, using the amigeconv command line tool.

Type the following commands in a Windows terminal window.

amigeconv -f bitplane -d 8 .\vshooter\_tiles.png vshooter\_tiles.raw

amigeconv.exe -f palette -p pal8 -c 256 -x .\vshooter\_tiles.png vshooter.pal

Include the palette in the copperlist, while include vshooter\_tiles.raw in the chip memory segment.

copperlist:

...

bplpointers:

...

palette       incbin     "gfx/vshooter.pal"         ; palette

              dc.w       $ffff,$fffe               ; end of copperlist

tileset       incbin     "gfx/vshooter\_tiles.raw"   ; image 640 x 192 pixel , 8

; bitplanes

Include the tilemap in the variables section:

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; VARIABLES

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

...

map  include    "gfx/vshooter\_map.i"

Now you need to define some constants, which represent the dimensions of tiles, tilesets, and tilemaps. Notice that the tilemap is now 5 x 40 in size.

; tiles

TILE\_WIDTH       equ 64

TILE\_HEIGHT      equ 64

TILE\_PLANE\_SZ    equ TILE\_HEIGHT\*(TILE\_WIDTH/8)

TILESET\_WIDTH    equ 640

TILESET\_HEIGHT   equ 192

TILESET\_ROW\_SIZE equ (TILESET\_WIDTH/8)

TILESET\_PLANE\_SZ equ (TILESET\_HEIGHT\*TILESET\_ROW\_SIZE)

TILESET\_COLS     equ 10

TILEMAP\_WIDTH    equ 5

TILEMAP\_HEIGHT   equ 40

TILEMAP\_ROW\_SIZE equ TILEMAP\_WIDTH\*2

; scroll

VIEWPORT\_HEIGHT  equ 256

VIEWPORT\_WIDTH   equ 320

; background

BGND\_WIDTH       equ 320

BGND\_HEIGHT      equ 2\*VIEWPORT\_HEIGHT+2\*TILE\_HEIGHT

BGND\_PLANE\_SIZE  equ BGND\_HEIGHT\*(BGND\_WIDTH/8)

BGND\_ROW\_SIZE    equ (BGND\_WIDTH/8)

Returns the DDFSTRT register to the default value of $38, in the copperlist, as well as the zero scroll value in BPLCON1 and the zero modules in BPL1MOD and BPL2MOD.

copperlist:

  dc.w    DIWSTRT,$2c81    ; display window start at ($81,$2c)

  dc.w    DIWSTOP,$2cc1    ; display window stop at ($1c1,$12c)

  dc.w    DDFSTRT,$38      ; display data fetch start at $38

  dc.w    DDFSTOP,$d0      ; display data fetch stop at $d0

  dc.w    BPLCON1,0

  dc.w    BPLCON2,0

  dc.w    BPL1MOD,0

  dc.w    BPL2MOD,0

In the main you need to initialize the bitplanes pointers so that the viewport is at a distance of (256 + 64) pixels from the start of the playfield. This situation corresponds to the first figure in the previous paragraph.

main:

  nop

  nop

  bsr       take\_system                                ; takes the control of

; Amiga's hardware

  move.l    #bgnd\_surface+(256+64)\*BGND\_ROW\_SIZE,d0    ; address of visible

; screen buffer

  move.l    #BGND\_PLANE\_SIZE,d1

  bsr       init\_bplpointers                           ; initializes bitplane

; pointers to our image

In this case, the tilemap will be drawn in horizontal rows, instead of columns as in the previous case of horizontal scrolling.

We then need to create a routine that draws a row of tiles. The code is similar to that of the draw\_tile\_column procedure.

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; Draws a row of 5 tiles.

;

; parameters:

; d0.w - map row

; d3.w - y position

; a1   - address where draw the tile

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

draw\_tile\_row:

  movem.l    d0-a6,-(sp)

; calculates the tilemap address from which to read the tile index

  lea        map,a0

  mulu       #TILEMAP\_ROW\_SIZE,d0    ; offset\_y = map\_row \* TILEMAP\_ROW\_SIZE

  ext.l      d0

  add.l      d0,a0

  moveq      #TILEMAP\_WIDTH-1,d7     ; number or tilemap column - 1

  move.w     #0,d2                   ; x position

.loop:

  move.w     (a0),d0                 ; tile index

  bsr        draw\_tile

  add.w      #TILE\_WIDTH,d2          ; increment x position

  add.l      #2,a0                   ; move to the next column of the tilemap

  dbra       d7,.loop

  movem.l    (sp)+,d0-a6

  rts

Now you need to initialize the viewport, drawing the first 4 lines of the tilemap on it. Initialize the pointer to the tilemap line, map\_ptr, 4 lines before the end. This is because you will start showing the final part of the tilemap, moving the viewport upwards. To initialize the viewport, modify the init\_background routine, used for horizontal scrolling. Below is the code.

map\_ptr  dc.w       TILEMAP\_HEIGHT-4    ; current map row

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; Initializes the background, copying the initial part of the level map.

;

; parameters:

; d0.w - map row from which to start drawing tiles

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

init\_background:

         movem.l    d0-a6,-(sp)

; initializes the part that will be visible in the viewport

         moveq      #4-1,d7             ; number of tile rows - 1 to draw

         lea        bgnd\_surface,a1     ; address where draw the tile

         move.w     #(256+64),d3        ; position y

.loop    bsr        draw\_tile\_row

         add.w      #1,d0               ; increment map row

         add.w      #1,map\_ptr

         add.w      #TILE\_HEIGHT,d3     ; increase position y

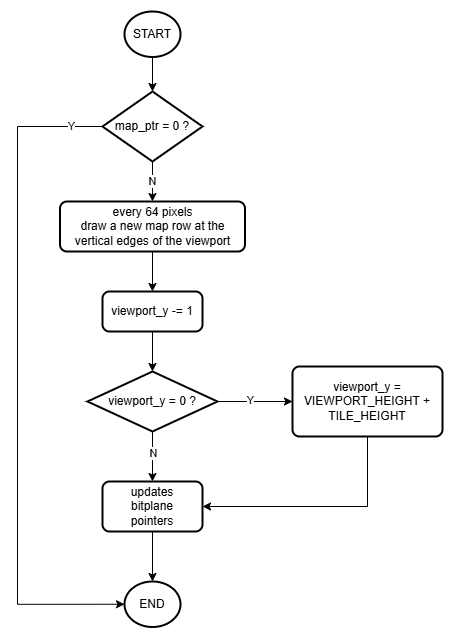
         dbra       d7,.loop

         move.w     #TILEMAP\_HEIGHT-4,map\_ptr

         movem.l    (sp)+,d0-a6

         rts

To scroll down the tilemap, you need to create a new routine, which you call scroll\_background. The following figure shows a flowchart of the routine.



The assembly code corresponding to the flowchart is as follows:

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; Scrolls the background downwards.

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

scroll\_background:

         movem.l    d0-a6,-(sp)

         tst.w      map\_ptr                ; end of map?

         beq        .return                ; if yes, returns

; every 64 pixels draws a new map row at at the upper and lower edges of the viewport

         move.w     viewport\_y,d0

         ext.l      d0

         divu       #64,d0                 ; viewport\_y / 64

         swap       d0

         tst.w      d0                     ; remainder = 0?

         beq        .draw\_new\_row          ; yes, draws new row

         bra        .scroll\_viewport

.draw\_new\_row:

         sub.w      #1,map\_ptr

         move.w     map\_ptr,d0             ; map row

         move.w     viewport\_y,d3          ; y = viewport\_y - TILE\_HEIGHT

         sub.w      #TILE\_HEIGHT,d3

         lea        bgnd\_surface,a1

         bsr        draw\_tile\_row          ; draws the row at the top of the

; viewport

         move.w     viewport\_y,d3          ; y = viewport\_y + VIEWPORT\_HEIGHT

         add.w      #VIEWPORT\_HEIGHT,d3

         bsr        draw\_tile\_row          ; draws the row at the bottom of the

; viewport

.scroll\_viewport:

         sub.w      #1,viewport\_y          ; decreases viewport y, to move it

; upwards

         tst.w      viewport\_y             ; viewport\_y = 0?

         beq        .reset\_viewporty       ; if yes, resets the viewport y

; position

         bra        .update\_bplpointers

.reset\_viewporty:

         move.w     #VIEWPORT\_HEIGHT+TILE\_HEIGHT,viewport\_y

.update\_bplpointers:

         move.w     viewport\_y,d1

         mulu       #BGND\_ROW\_SIZE,d1      ; offset\_y = viewport\_y \*

; BGND\_ROW\_SIZE

         ext.l      d1

         move.l     #bgnd\_surface,d0

         add.l      d1,d0                  ; adds offset\_y

         move.l     #BGND\_PLANE\_SIZE,d1

         bsr        init\_bplpointers       ; updates bitplane pointers

.return  movem.l    (sp)+,d0-a6

         rts

To achieve vertical scrolling, you need to invoke the scroll\_background routine in the main loop:

mainloop:

  bsr      wait\_vblank          ; waits for vertical blank

  bsr      scroll\_background

  btst     #6,CIAAPRA           ; left mouse button pressed?

  bne.s    mainloop             ; if not, repeats the loop

By running the program, you will get the vertical scrolling of the tilemap.

The complete source code is available at the following url:

<https://github.com/stefanocoppi/amiga_game_prog_assembly/tree/main/chapter10C>

# 11. Hardware sprites and joystick reading

## What are hardware sprites?

A sprite is a graphic object that is independent of the background and other sprites themselves. Amiga provides up to 8 hardware sprites, managed by as many DMA channels.

With OCS/ECS chipsets, the sprites are 16 pixels wide while they can be as tall as the entire screen. A sprite can use 3 different colors. By attaching two sprites, you get a 15-color sprite. The hardware can detect collisions between sprites and between sprites and playfields.

## Example 1: Joystick movement of an attached sprite

To teach you how to use hardware sprites, we'll create an example program that will display a hardware sprite in attached mode, with 15 colors. The user will be able to move the sprite with the joystick. In addition, the program will detect collisions between the sprite and an image displayed on the playfield, coloring the edge of the screen red.

The starting point for this program will be the code created in Chapter 7.

## Sprite DMA Channels

To use hardware sprites, we need to enable the corresponding DMA channel. In practice we must set bit 5 of the DMACON register. In the following code, we set the value of DMACON by means of a constant, which will then be assigned to DMACON in the take\_system routine.

; DMACON register settings

; enables sprites DMA (bit 5)

; enables copper DMA (bit 7)

; enables bitplanes DMA (bit 8)

           ;5432109876543210

DMASET equ %1000001110100000

## Sprite size

A hardware sprite, for Amigas with OCS/ECS chipset, has a width of 16 pixels, while the height can vary from 0 to the entire screen.

Immagine che contiene testo, schermata, Carattere, diagramma

Descrizione generata automaticamente

## Sprite coordinates

The origin of the sprite coordinate system is placed outside the visible screen. With reference to the following figure, the visible screen is 64 pixels horizontally and 44 vertically from the origin of the sprites. Then you need to add these quantities to the coordinates of the sprites on the visible screen.

Immagine che contiene testo, schermata, diagramma, linea

Descrizione generata automaticamente

## Sprites data structure

A hardware sprite has the data structure represented in the following figure.

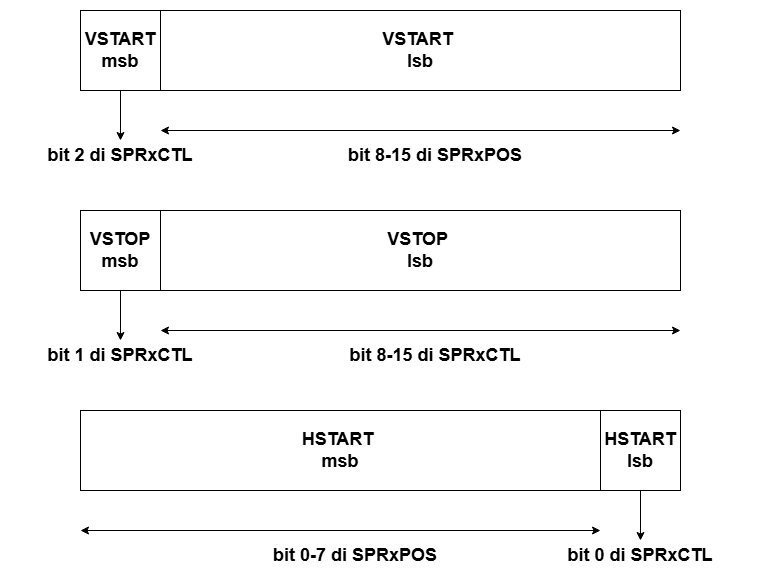
Immagine che contiene testo, schermata, numero, Carattere

Descrizione generata automaticamente

The structure begins with two control words, named SPRxPOS and SPRxCTL. It ends with two control words that are worth 0. After the initial two control words, two words follow that represent line 1, then line 2, line N.

The following figure shows how to calculate:

* VSTART, which is the line number at which the sprite begins
* VSTOP, the line number at which the sprite ends
* HSTART, the horizontal position at which the sprite begins



## Creating Sprite Data Structure

The data structure just seen will not be created manually, but the **amigeconv** tool will take care of it. For our example, we want to display a 15-color sprite. So, we create a 15-color, 16-pixel-wide PNG image and call it alien.png. The figure is shown, magnified by 800%, below.

Immagine che contiene modello, schermata, Policromia, quadrato

Descrizione generata automaticamente

To create the sprite's data structure, open a window with the Windows command prompt and type the following command.

amigeconv -f sprite -a -w 16 -t -d 4 .\alien.png alien.raw

The -a option indicates that the sprite is in attached mode.

The -w 16 option indicates the width 16 pixels.

The -t option indicates that control words should be created.

The -d 4 option indicates that 4 bitplanes are used.

## Inspecting the Sprite Data Structure with Hex Editor

The data structure will be created in the binary file alien.raw, which we can open with the Hex Editor of Visual Studio Code. Let's see how to do it.

First, you need to install the "Hex Editor" extension for Visual Studio Code.

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

Once the extension is installed, you need to right-click on the file name alien.raw. From the context menu that appears, select the "Open With..." item.

Immagine che contiene testo, Carattere, numero, linea

Descrizione generata automaticamente

The following window will appear. Choose the "Hex Editor" item.

Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamente

The following window will appear. On each line are shown 16 bytes of the file, in hexadecimal notation. In addition, the ascii representation of the bytes themselves is shown. Note that the first 4 bytes, highlighted in the figure, with a value of 0, represent the 2 control words of the 0 sprite.

Immagine che contiene testo, schermata, numero, Carattere

Descrizione generata automaticamente

The next word, highlighted in the following figure, represents plane 1 of the first line of sprite 0.



The next word, highlighted in the following figure, represents plane 2 of the first row of sprite 0.



Sprite 0 will end with another 4 bytes at 0, highlighted in the following figure.

Immagine che contiene testo, schermata, Carattere, numero

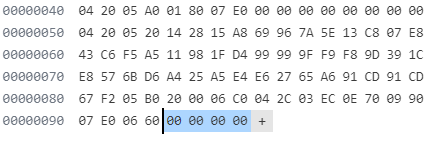
Descrizione generata automaticamente

Sprite 1 will begin with the next 2 control words, highlighted in the following figure.

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

Sprite 1 ends with 2 words to 0, highlighted in the following figure.



The alien.raw file must be included in the chip memory segment, as shown in the following code.

; segment loaded in CHIP RAM

              SECTION    graphics\_data,DATA\_C

...

alien\_sprite  incbin     "gfx/alien.raw"

## Sprite pointers

Each sprite has a pointer that should point to a data structure like the one seen in the previous paragraph. The following table shows the pointer sprite logs.

|  |  |  |
| --- | --- | --- |
| Register | Address | Description |
| SPR0PT | $dff120 | Sprite 0 pointer |
| SPR1PT | $dff124 | Sprite 1 pointer |
| SPR2PT | $dff128 | Sprite 2 pointer |
| SPR3PT | $dff12c | Sprite 3 pointer |
| SPR4PT | $dff130 | Sprite 4 pointer |
| SPR5PT | $dff134 | Sprite 5 pointer |
| SPR6PT | $dff138 | Sprite 6 pointer |
| SPR7PT | $dff13c | Sprite 7 pointer |

Normally, sprite pointers are set in the copperlist, as in the following code snippet.

copperlist:

  ; BPLCON0 ($100)

  ; bit 9: set to 1 to enable composite video output

  ; bit 12-14: least significant bits of bitplane number

  ;                           5432109876543210

             dc.w    BPLCON0,%0100001000000000

bplpointers  :

             dc.w    $e0,0,$e2,0                  ; plane 1

             dc.w    $e4,0,$e6,0                  ; plane 2

             dc.w    $e8,0,$ea,0                  ; plane 3

             dc.w    $ec,0,$ee,0                  ; plane 4

sprite\_pointers:

             dc.w    SPR0PTH,0,SPR0PTL,0

             dc.w    SPR1PTH,0,SPR1PTL,0

             dc.w    SPR2PTH,0,SPR2PTL,0

             dc.w    SPR3PTH,0,SPR3PTL,0

             dc.w    SPR4PTH,0,SPR4PTL,0

             dc.w    SPR5PTH,0,SPR5PTL,0

             dc.w    SPR6PTH,0,SPR6PTL,0

             dc.w    SPR7PTH,0,SPR7PTL,0

Notice that in the copperlist we set the BPLCON0 register to display a 4-bitplane playfield and we also declared bitplane pointers.

To initialize the sprite pointers we create the following routine, init\_sprite\_pointers.

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; Initializes sprite pointers

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

init\_sprite\_pointers:

  movem.l    d0-a6,-(sp)

  lea        sprite\_pointers,a1

  move.l     #alien\_sprite,d0

  move.w     d0,6(a1)                ; low word

  swap       d0

  move.w     d0,2(a1)                ; high word

  add.l      #8,a1                   ; next sprite pointer

  move.l     #alien\_sprite+76,d0     ; next sprite

  move.w     d0,6(a1)                ; low word

  swap       d0

  move.w     d0,2(a1)                ; high word

  bset       #7,alien\_sprite+76+3    ; sets sprite1 attached bit

  movem.l    (sp)+,d0-a6

  rts

The init\_sprite\_pointers routine must be invoked during the initialization phase, on the main, as shown in the following code snippet.

main:

  nop

  nop

  bsr       take\_system             ; takes the control of Amiga's hardware

  move.l    #bgnd,d0                ; address of screen in d0

  bsr       init\_bplpointers        ; initializes bitplane pointers to our

; image

  bsr       init\_sprite\_pointers

## Sprite colors

Sprites can use 3 colors, since the fourth is given by the transparent background. The following table shows the color registers associated with each pair of sprites.

|  |  |  |
| --- | --- | --- |
| Sprite no. | Color register | Address |
| Sprite 0,1 | Color17 | $dff1a2 |
| Color18 | $dff1a4 |
| Color19 | $dff1a6 |
| Sprite 2,3 | Color21 | $dff1aa |
| Color22 | $dff1ac |
| Color23 | $dff1ae |
| Sprite 4,5 | Color25 | $dff1b2 |
| Color26 | $dff1b4 |
| Color27 | $dff1b6 |
| Sprite 6,7 | Color29 | $dff1ba |
| Color30 | $dff1bc |
| Color31 | $dff1be |

## Sprites “attached”

To overcome the limitation of only 3 colors per sprite, you can use two "attached" sprites. In this mode, 15 colors can be used. Of course, the total number of sprites is reduced to 4. Sprites can only be stacked in pairs:

Pair 1: Sprite0 + Sprite 1

Pair 2: Sprite2 + Sprite 3

Pair 3: Sprite4 + Sprite 5

Pair 4: Sprite6 + Sprite 7

The colors are those ranging from:

Color 17 - $dff1a2 to Color 31 - $dff1be

We saw that we needed the -a flag to export the file with the data structure of our sprite in attached mode. Now we need to generate the palette and load it into memory.

To generate the palette, we use the amigeconv tool. Let's type the following command in a window with the Windows Command Prompt.

amigeconv.exe -f palette -p pal4 -c 16 -x .\alien.png alien.pal

We will get the alien.pal file, in binary copperlist format that will contain the palette of our sprite.

Since attached sprites use color registers 17 to 31, we need to edit the palette manually, using Hex Editor. After opening the file with Hex Editor, we notice that the palette uses color registers ranging from 0 to 15. Instead of the word $0180, which indicates the COLOR0 register, we must enter the value $01A0 corresponding to the COLOR16 register. Instead of $0182 (COLOR1) we replace $01A2 (COLOR17) and so on until COLOR31.

Immagine che contiene testo, Carattere, schermata

Descrizione generata automaticamente

After editing the alien.pal file, we include it in the copperlist, as in the following code snippet.

copperlist:

...

sprite\_pointers:

...

         dc.w      SPR7PTH,0,SPR7PTL,0

palette  incbin    "gfx/alien.pal"

## Priority between sprites and playfields

The priority between the sprites and playfield 1 is handled by bits 0-2 of register BPLCON2, while the priority between playfield 2 is handled by bits 3-5.

The following table shows the values to be entered in the BPLCON2 register.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Code | 000 | 001 | 010 | 011 | 100 |
| Max priority | Playfield | Pair 1 | Pair 1 | Pair 1 | Pair 1 |
|  | Pair 1 | Playfield | Pair 2 | Pair 2 | Pair 2 |
|  | Pair 2 | Pair 2 | Playfield | Pair 3 | Pair 3 |
|  | Pair 3 | Pair 3 | Pair 3 | Playfield | Pair 4 |
| Min Priority | Pair 4 | Pair 4 | Pair 4 | Pair 4 | Playfield |

On a practical level, you need to set the priority of all sprites on playfields, as in the following code snippet.

copperlist:

  ...

  dc.w    BPLCON2,%100100    ; sets sprites priority over playfield

## Placing a Sprite

We have seen on a theoretical level the meaning of the VSTART, VSTOP, HSTART fields of the sprite control words. Now let's create a routine that allows the positioning of a sprite, using the visible screen as a reference system. Below is the source code.

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; Sets the position of a sprite

;

; parameters:

; a1 - sprite address

; d0.w - y position (0-255)

; d1.w - x position (0-319)

; d2.w - sprite height

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

set\_sprite\_position:

  movem.l    d0-a6,-(sp)

  add.w      #$2c,d0                ; adds offset of screen beginning

  move.b     d0,(a1)                ; copies y into sprite VSTART byte

  btst.l     #8,d0                  ; bit 8 of y position is set?

  beq        .dontset\_bit8

  bset.b     #2,3(a1)               ; sets bit 8 of VSTART

  bra        .vstop

.dontset\_bit8:

  bclr.b     #2,3(a1)               ; clears bit 8 of VSTART

.vstop:

  add.w      d2,d0                  ; adds height to y position to get VSTOP

  move.b     d0,2(a1)               ; copies the value into sprite VSTOP byte

  btst.l     #8,d0                  ; bit 8 of VSTOP is set?

  beq        .dontset\_VSTOP\_bit8

  bset.b     #1,3(a1)               ; sets bit 8 of VSTOP

  bra        .set\_hpos

.dontset\_VSTOP\_bit8:

  bclr.b     #1,3(a1)               ; clears bit 8 of VSTOP

.set\_hpos:

  add.w      #128,d1                ; adds horizontal offset to x

  btst.l     #0,d1

  beq        .HSTART\_lsb\_zero

  bset.b     #0,3(a1)               ; sets bit 0 of HSTART

  bra        .set\_HSTART

.HSTART\_lsb\_zero:

  bclr.b     #0,3(a1)               ; clears bit 0 of HSTART

.set\_HSTART:

  lsr.w      #1,d1                  ; shifts 1 position to right to get the 8 most significant bits of x position

  move.b     d1,1(a1)               ; sets HSTART value

  movem.l    (sp)+,d0-a6

  rts

The set\_sprite\_position routine must be invoked at initialization time, immediately after the pointer sprite has been initialized, as shown in the following code snippet.

main:

  nop

  nop

  bsr       take\_system             ; takes the control of Amiga's hardware

  move.l    #bgnd,d0                ; address of screen in d0

  bsr       init\_bplpointers        ; initializes bitplane pointers to our

; image

  bsr       init\_sprite\_pointers

  lea       alien\_sprite,a1

  move.w    sprite\_y,d0             ; y position

  move.w    sprite\_x,d1             ; x position

  move.w    #SPRITE\_HEIGHT,d2       ; sprite height

  bsr       set\_sprite\_position

  lea       alien\_sprite+76,a1

  bsr       set\_sprite\_position

## Reading joystick

The Amiga has 2 joystick ports, called port0 and port1. Normally the mouse is connected to port0, while the joystick is connected to port1. Of course, it is also possible to connect a joystick to the port0.

To read the movement of the joystick, you need to use the JOY0DAT register for the joystick connected to port0, while JOY1DAT for the one connected to port1.

Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamente

Let's see how to detect the movement of the joystick lever. If we move the lever to the right, bit 1 of the JOYxDAT register is set. While if we move the lever to the left, bit 9 is set.

Immagine che contiene testo, linea, schermata, numero

Descrizione generata automaticamente

To detect the movement of the lever down, we need to make eor between bits 0 and 1 of JOYxDAT. If the result is 1 then the lever has been moved down. For the upward movement, we need to make the eor between bits 8 and 9.

From a practical point of view, just make a copy of the JOYxDAT register and make a shift to the right by 1 bit. Then the eor is made between the original register and the shifted copy.

The down state will correspond to bit 0 of the result, while the up state will correspond to bit 8.

Immagine che contiene diagramma, linea, testo, Rettangolo

Descrizione generata automaticamente

Normally Amiga joysticks had only 1 button. To detect the joystick button press in port0, just test bit 6 of the $bfe001 register. For the joystick button in port1, test bit 7. If the button is pressed, the bit is 0.

Immagine che contiene testo, linea, diagramma, numero

Descrizione generata automaticamente

The joystick ports of the Amiga provide support for 2 other buttons. The state of these buttons can be read through the bits of the POGGOR register, as shown in the following figure. Be careful, that the pressed state of the button corresponds to the 0 state of the bit.

Immagine che contiene testo, numero, linea, schermata

Descrizione generata automaticamente

Using the WinUAE emulator you can configure a 3-button joystick.

By pressing F12 and selecting the "Game Ports" item, the window shown in the following figure will appear. Select Gamepad from the drop-down menu. Configuration testing is also possible.

Immagine che contiene testo, schermata, software, Icona del computer

Descrizione generata automaticamente

## Moving the Sprite with the Joystick

We want to move the newly created sprite with the joystick. We create a new routine, move\_sprite\_with\_joystick, which reads the state of the joystick and changes the sprite coordinates accordingly.

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; Moves the sprite with the joystick

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

move\_sprite\_with\_joystick:

  movem.l    d0-a6,-(sp)

  move.w     JOY1DAT(a5),d0

  btst.l     #1,d0                     ; joy right?

  bne        .set\_right

  btst.l     #9,d0                     ; joy left?

  bne        .set\_left

  bra        .check\_up

.set\_right:

  add.w      #SPRITE\_SPEED,sprite\_x

  bra        .check\_up

.set\_left:

  sub.w      #SPRITE\_SPEED,sprite\_x

.check\_up:

  move.w     d0,d1

  lsr.w      #1,d1

  eor.w      d1,d0

  btst.l     #8,d0                     ; joy up?

  bne        .set\_up

  btst.l     #0,d0                     ; joy down?

  bne        .set\_down

  bra        .move\_sprite

.set\_up:

  sub.w      #SPRITE\_SPEED,sprite\_y

  bra        .move\_sprite

.set\_down:

  add.w      #SPRITE\_SPEED,sprite\_y

.move\_sprite:

  lea        alien\_sprite,a1

  move.w     sprite\_y,d0               ; y position

  move.w     sprite\_x,d1               ; x position

  move.w     #SPRITE\_HEIGHT,d2         ; sprite height

  bsr        set\_sprite\_position

  lea        alien\_sprite+76,a1

  bsr        set\_sprite\_position

  movem.l    (sp)+,d0-a6

  rts

The move\_sprite\_with\_joystick routine must be invoked in the main loop, at every frame.

mainloop:

  bsr      wait\_vblank                  ; waits for vertical blank

  bsr      move\_sprite\_with\_joystick

  bsr      check\_collisions

  btst     #6,CIAAPRA                   ; left mouse button pressed?

  bne.s    mainloop                     ; if not, repeats the loop

## Collisions

The Amiga hardware is capable of detecting collisions both between sprites and between sprites and playfields. Using the CLXCON register ($dff098) you can set the type of collisions to be detected. The following figure shows the register bits and their function. If we want to enable collisions of one of the odd sprites, we just need to set the corresponding bit. Collisions of even sprites are always detected. If we want to detect collisions between sprites and bitplanes, we need to enable planes. In addition, the color index must be loaded in bits 0-5. The collision will only be detected with the playfield bits of that color.

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

The CLXDAT register ($dff00e) indicates which collisions have been detected. Bit 1 is set if there is a collision between playfield1 and the sprite pair1, i.e. sprites 0 and 1. The following figure shows the meaning of all the bits in the register.

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

In our example program, we want to color the edge of the screen red when a collision between the sprite and the playfield is detected.

In the copperlist we set the CLXCON register to enable planes 1-4, enable sprite 1 and set the color number 8 as color index.

The check\_collisions routine will check if bit 1 of CLXDAT is set, which indicates a collision between playfield1 and pair 1, i.e. sprites 0 and 1. In this case, change the color 0 to red.

copperlist:

  ...

  ; Controls sprite-bitplane collisions

  ; bit 12: enable sprite 1

  ; bit 6-9: enable bitplanes 1-4

  ; bit 0-5: color index for collisions with playfield

  ;                  5432109876543210

  dc.w       CLXCON,%0001001111001000

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; Checks the collisions between sprite and playfield.

; If a collision is detected, change the border color to red.

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

check\_collisions:

  movem.l    d0-a6,-(sp)

  move.w     CLXDAT(a5),d0

  btst.l     #1,d0                       ; bit 1 checks collisions between playfield and sprites 0-1

  bne        .collision

  move.w     #$0000,bgnd\_palette+2

  bra        .return

.collision:

  move.w     #$0f00,bgnd\_palette+2

.return:

  movem.l    (sp)+,d0-a6

  rts

The check\_collisions procedure must be invoked in the main loop, as in the following code snippet.

mainloop:

  bsr      wait\_vblank                  ; waits for vertical blank

  bsr      move\_sprite\_with\_joystick

  bsr      check\_collisions

  btst     #6,CIAAPRA                   ; left mouse button pressed?

  bne.s    mainloop                     ; if not, repeats the loop

Running the example program, we will see a sprite in the shape of an alien's face that can move in 4 directions with the joystick in port 1. If we move the sprite to the playfield rock, we will see that the edge of the screen turns red, to detect the collision.

Immagine che contiene schermata, testo

Descrizione generata automaticamente

You can download the full code of the example at the following url:

<https://github.com/stefanocoppi/amiga_game_prog_assembly/tree/main/chapter11>

## AGA Sprites

# 12. Multidirectional scrolling

# 13. Blitter objects (Bobs)

Ffff

# 14. Player’s ship

# Enemies

Ggggg

# Firing bullets

Aaaa

# Collisions and explosions

Fffff

# Text rendering

Ggggg

# Information panel

Ffff

# Power-ups

Ffff

# Satellite

Ffff

# Game states

Ffff

# Title screen

Gggg

# Game over screen

Ffff

# Keyboard and mouse input

ffff

# End level boss

Sss

# Sound

Ffff

# Loading level data from disk