The *AmsGraph* library

INTRODUCTION

The *AmsGraph* library is a FreeBASIC library giving access to the graphic functions of the old Amstrad CPC computers. Most functions are implemented, sometimes with extended features, and some new functions are provided.

INSTALLATION

The library is provided as source code in the `src` subdirectory. It must be compiled with FreeBASIC into a static library :

```
fbc amsgraph.bas -lib
```

Then, place the resulting file `libamsgraph.a` in the appropriate subdirectory of your FreeBASIC installation (e. g. `lib\win32`), and the header file `amsgraph.bi` in the `inc` subdirectory.

Some functions need the FBImage library by D. J. Peters. You should therefore install this one, too.

There are 15 sample programs, placed in the `examples` subdirectory.

NOTATION

In what follows, we have used suffixes to indicate the types of the function parameters : `%` for integers (`long`), `$` for strings and none for real numbers (`double`) .

GRAPHIC MODES

The Amstrad CPC computer has 3 graphic modes: 0, 1 and 2.

All 3 modes have the same resolution (640 × 400 pixels) and can display 25 lines of 20, 40 or 80 characters respectively.

Since the total width of the screen is constant, the character size must adapt: 32 ×16 pixels in mode 0, 16 × 16 in mode 1 and 8 ×16 in mode 2.

The *AmsGraph* library provides the following extensions:

1. An optional parameter: the title of the graphic window

```
MODE 1, "Graphics in mode 1"
```

2. A custom mode: mode 3 (or higher) which allows to choose the sizes of the screen and characters:

```
MODE 3, Title$, w%, h%, w1%, h1%
```

with :

(`w, h`) = width and height of the window in pixels

(`w1, h1`) = width and height of a character element in pixels (each character consists of 8 × 8 elements)

Example:

```
MODE 3, "Custom mode", 850, 100, 4, 10
```

These parameters are optional, their default values being those of mode 1 (640, 400, 2, 2)

WRITING TEXT

The following statements have been renamed to avoid confusion with the FreeBASIC statements :

- `GCLS` (Graphic `CLS`) clears the graphic screen
- `CRLOCATE` (Column/Row `LOCATE`) places the cursor at a given position
- `APRINT` (Amstrad `PRINT`) displays a text, using the Amstrad font :

  ```
  APRINT txt$, x%, y%
  ```

  `txt` is the text to be written; (`x, y`) is the position of the text (upper left corner) in pixels.

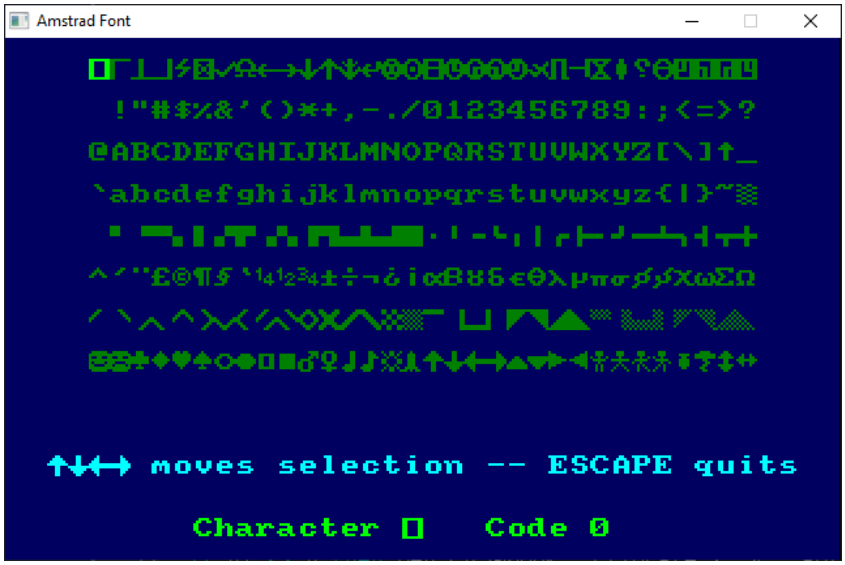
  The last two parameters are optional; if they are absent, the position defined by `CRLOCATE` will be used.

- `FBPRINT` (FreeBASIC `PRINT`) displays a text, using the FreeBASIC font.

  The syntax is similar to that of the `APRINT` statement, but the position of the text must be defined in pixels, as the `CRLOCATE` statement concerns only the Amstrad font.

  The FB font is always 8 × 16 and is not affected by character redefinitions or control characters.

The Amstrad font has 256 characters corresponding to ASCII codes 0 to 255. The example program `caract.bas` displays all these characters. By scrolling through the table with the arrow key, the code of the chosen character is obtained.



ALPHABETICAL INKEY FUNCTION

The `AINKEY` function is an alternative to the FreeBASIC `INKEY` function. It returns a string of characters corresponding to the keystroke. Alphanumeric characters are returned as is, for example :

```
if ainkey() = "A" then ...
```

For "special" keys (function keys, arrows etc.) the returned string is the description of the key :

```
if ainkey() = "F1" then ...      ' Function key
if ainkey() = "DOWN" then ...    ' Down arrow
if ainkey() = "ESCAPE" then ...  ' Escape key
```

The predefined strings are :

```
"CTRL+A" ... "CTRL+F", "CTRL+J" ... "CTRL+L", "CTRL+N" ... "CTRL+Z"

"F1" ... "F12"

"BACKSPACE", "TAB", "ENTER", "ESCAPE", "HOME", "END"

"UP", "DOWN", "LEFT", "RIGHT" (Arrows)

"PAGEUP", "PAGEDOWN"

"INSERT", "DELETE"
```

Note: The function reads the keyboard continuously and returns the empty string if no key is pressed. It can therefore be used to make waiting loops:

```
while ainkey() = "" : wend
```

COLOR SYSTEMS

Colors are usually expressed in ARGB format, just like in FreeBASIC.

The HSV (*Hue*, *Saturation*, *Value*) system is another way of representing colors that is often more effective than the classic RGB system.

The following statements allow the conversion between the two systems :

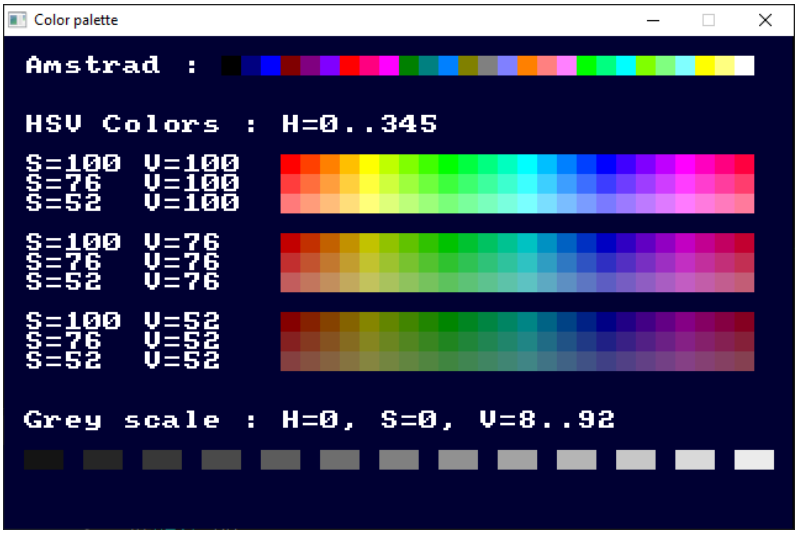
```
RGBtoHSV R%, G%, B%, H, S, V

HSVtoRGB H, S, V, R%, G%, B%
```

where R, G, B are integer numbers (0 .. 255), H is a real angle (0 .. 360°), S and V are real numbers in (0 .. 1)

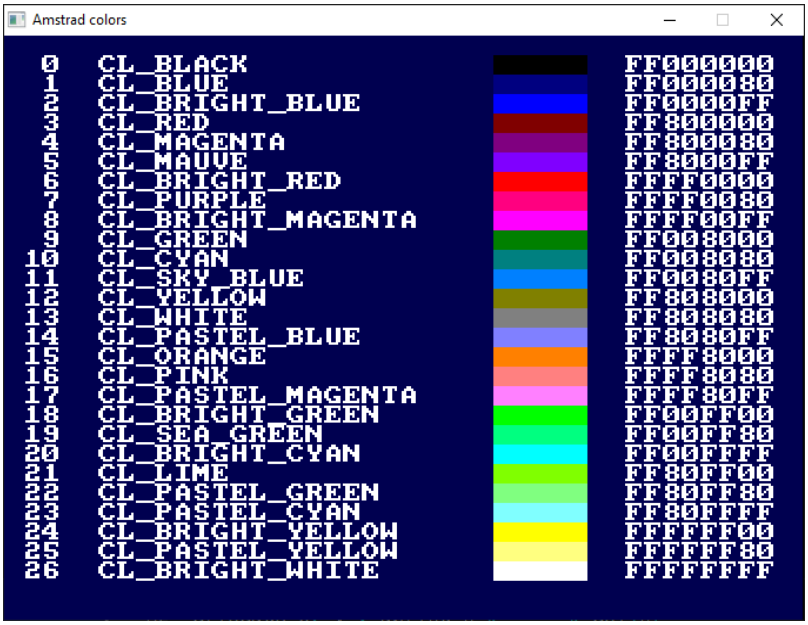
PREDEFINED COLORS

The library has a palette of 256 predefined colors, adapted from FreeBASIC and displayed on this image generated by the example program `palette.bas` (the values of S and V are noted here in percentages):



Each color has a code, from 0 to 255.

Codes 0 to 26 correspond to the 27 colors of the Amstrad CPC, displayed on the following image, generated by the example program `colors.bas` :



These colors are generated by taking 3 levels (0%, 50% and 100%) of each R, G, B color, resulting in $3^3 = 27$ colors.

Codes 27 to 242 are defined by :

- 24 shades (H) at 15° intervals on the color wheel
- 3 values for saturation (S) and brightness (V)

Codes 243 to 255 correspond to grey levels.

COLOR MANAGEMENT

The `PAPER` and `PEN` statements allow to change the background and foreground colors. For example, to have a dark red background, you can use one of the following syntaxes:

```
PAPER CL_RED       PAPER &hFF800000       PAPER RGB(128, 0, 0)
```

The `INK` statement allows to change one of the predefined colors:

```
INK 10, &hFF00FF00 ' color 10 becomes a bright green
```

The `GET_INK` function returns an integer corresponding to one of the predefined colors.

```
GET_INK(1)         ' returns 4278190208
HEX(GET_INK(1))    ' returns FF000080
```

REDEFINITION OF CHARACTERS

The `SYMBOL` statement allows to redefine a character, according to the syntax :

```
SYMBOL code%, n1%, n2%, n3%, n4%, n5%, n6%, n7%, n8%.
```

where `code` is the ASCII code of the character to be modified, `n1` to `n8` are integers corresponding to the binary encodings of the 8 lines constituting the character.

The example program `symbol.bas` redefines the letter `i` (code 105) with :

```
n1 = 255 ' 11111111 in binary
n2 = 129 ' 10000001 in binary
n3 = 189 ' 10111101 in binary
n4 = 153 ' 10011001 in binary
n5 = 153 ' 10011001 in binary
n6 = 189 ' 10111101 in binary
n7 = 129 ' 10000001 in binary
n8 = 255 ' 11111111 in binary
```

The character then takes on the following appearance:

```
********
*      *
* **** *
*  **  *
*  **  *
* **** *
*      *
********
```

MULTICOLORED CHARACTERS

The `SYMBOL` statement creates only monochrome characters. To have multicolored characters, *AmsGraph* adds the `SYMBCOL` statement whose syntax is the following:

```
SYMBCOL code%, colstr$
```

where `colstr` is a string whose ASCII codes represent the entries in the color palette.

For example, to create the following character with the default Amstrad palette :

we need the colors:

- CL_BLUE (entry 1)
- CL_BRIGHT_BLUE (entry 2)
- CL_MAGENTA (entry 4)
- CL_BRIGHT_RED (entry 6)
- CL_PINK (entry 16)
- CL_BRIGHT_WHITE (entry 26)

If we agree to represent the background color by a space (ASCII 32), the string will be of the form :

```
colstr = chr(32, 32, 32, 32, 32, 32, 32, 32) + _
         chr(32,  4,  4,  4,  4,  4, 32, 32) + _
         chr( 4,  4,  4,  4, 16,  4,  4, 32) + _
         chr( 4, 16,  1, 16, 16,  1,  4, 32) + _
         chr( 4, 16, 16, 16, 16, 16,  4, 32) + _
         chr( 4,  4,  2,  2,  2,  4, 32, 32) + _
         chr(32, 32, 26, 26, 26, 32, 32, 32) + _
         chr(32, 32,  6, 32,  6, 32, 32, 32)
```

We have split the string into 8 segments to show the structure of the graph. Note that the FreeBASIC `CHR` function seems limited to 32 parameters.

The creation of the character (e.g. code 200) and its display will be done by :

```
SYMBCOL 200, colstr
APRINT chr(200)
```

The colors can also be assigned to alphanumeric characters, which simplifies the writing of the string. For example :

```
INK ASC("B"), CL_BLUE
INK ASC("L"), CL_BRIGHT_BLUE
INK ASC("M"), CL_MAGENTA
INK ASC("P"), CL_PINK
INK ASC("R"), CL_BRIGHT_RED
INK ASC("W"), CL_BRIGHT_WHITE

colstr = "        " _
         " MMMMM  " _
         "MMMMPMM " _
         "MPBPPBM " _
         "MPPPPPM " _
         "MMLLLM  " _
         "  WWW   " _
         "  R R   "
```

The example program `symbcol.bas` shows this possibility.

GRAPHIC CHARACTER EDITOR

The program `caredit.bas` allows to modify and colorize the graphic characters (codes ≥ 127).

The 16-color palette is the one used by the PICO-8 software but you can change it.

The strings that make up the parameters of the `SYMBCOL` statement are stored in the random access file `caredit.dat`, each string consisting of 64 characters.

The characters thus modified can be used in any program. The following example reads the 4 characters of ASCII codes 248 to 251 from the file and displays them on the screen.

```
open "caredit.dat" for random as #1 len=64

dim as string car
dim as long i

mode 1

LoadPalette

for i = 0 to 3
  car = read_char(248 + i)
  crlocate 2 * i + 2, 2
  aprint car
next i

while ainkey() = "" : wend

close #1

end
```

```
sub LoadPalette ()
  ' Insert the color definitions here
  ' as in the LoadPalette subroutine
  ' of the caredit.bas program

  ink asc("R"), &hFF004D ' Red
  ........................................
end sub

function read_char (nchar as long) as string
  dim as string*64 ch
  get #1, nchar - 127, ch
  symbcol nchar, ch
  return chr(nchar)
end function
```

Note that in the `read_char` function the `ch` variable that reads a record must be 64 characters long.

CONVERTING IMAGES TO CHARACTERS

The `PICtoCHAR` statement converts an image into a sequence of characters that can be displayed by `APRINT`. The syntax is:

```
PICtoCHAR filename$, code%
```

where `filename` is the name of the graphics file and `code` is the ASCII code of the first character.

The example programs `pictochar.bas` and `maze.bas` show this usage.

The characters are read line by line, up to the maximum code of 255.

The limitations are as follows:

- The image dimensions must be multiples of 8
- The graphic mode must correspond to the size of the characters. For example, for characters of 24 × 32 pixels, we will choose the custom mode (mode 3), the last two parameters of the `MODE` statement being equal to 3 and 4 (i.e. 24/8 and 32/8).

CONTROL CODES

The character codes 0 to 31 can be interpreted as control characters:

```
APRINT CHR(0)     Allows control characters (default)
APRINT CHR(1)     Print control characters

APRINT CHR(7)     Ring the bell (equivalent to BEEP)
APRINT CHR(8)     Shift cursor to the left
APRINT CHR(9)     Shift cursor to the right
```

```
APRINT CHR(10)    Move cursor down one line
APRINT CHR(11)    Move cursor up one line
APRINT CHR(12)    Clear the screen
APRINT CHR(13)    Return to line
APRINT CHR(14, n) Equivalent of PAPER with n = color index
APRINT CHR(15, n) Equivalent of PEN with n = color index

APRINT CHR(17)    Erase the line until the last character
APRINT CHR(18)    Erase the line from the last character
APRINT CHR(19)    Erase from the screen top to the last character
APRINT CHR(20)    Clear from the last character to the screen bottom

APRINT CHR(22, n) Set interaction with background ("bit blit"):

  n = 0 ==> PSET mode (normal)
  n = 1 ==> XOR mode
  n = 2 ==> AND mode
  n = 3 ==> OR mode
  n = 4 ==> ALPHA mode, with transparency = A of RGBA
  n = 5 ==> TRANS mode, with transparency = &hFF00FF (magenta)

APRINT CHR(30)     Equivalent of CRLOCATE 1, 1
APRINT CHR(31,x,y) Equivalent of CRLOCATE x, y
```

The example program `graffiti.bas` is adapted from a series of articles published in the french magazine *Amstrad magazine* under the title *Amstradian graffitis*. It shows how to create *sprites* by combining 4 characters in a 2 x 2 matrix. The code characters 251, 252, 253, 254 are redefined by the `SYMBOL` statement. We use control characters : 31 to place the *sprite*, then 8 and 10 to position the second row of characters. A simple `APRINT` statement is sufficient to display the *sprite*:

```
APRINT CHR(31, x%, y%, 251, 252, 8, 8, 10, 253, 254)
```

The example program `transpar.bas` shows the writing of a text at different transparency levels, set by the `RGBA` function after activating the transparent mode by the control character (code 22).

PLOTTING AREA

The `ORIGIN` statement sets the origin of the axes and defines a rectangular graphics area (*viewport*) inside the window. All subsequent plots will be limited to this area: points outside it will not be plotted (*clipping*).

The syntax of this statement is:

```
ORIGIN x%, y%, lft%, rgt%, top%, bottom%, fill_color%, border_color%
```

- `x, y` define the position of the origin (coordinates 0, 0)
- `lft, rgt, top, bottom` are the coordinates of the rectangle delimiting the drawing area.
- `fill_color` is the fill color
- `border_color` is the color of the frame

The parameters `lft, rgt, top, bottom` are optional; if they are absent the plot will use the entire graphics window

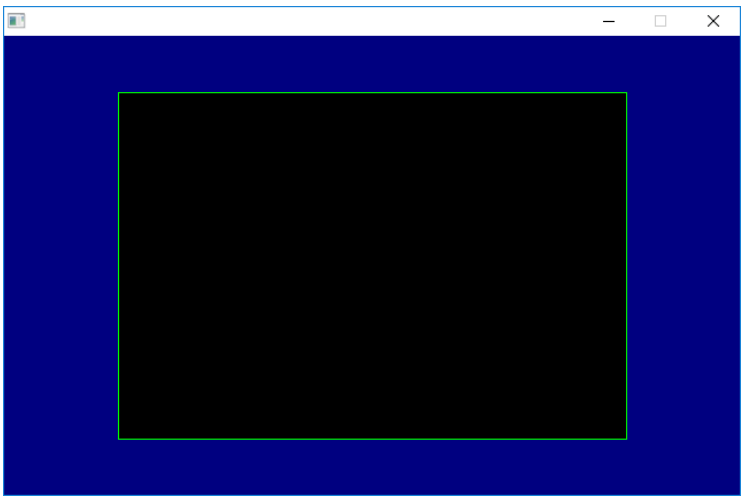
The coloring parameters are also optional; if they are absent, the corresponding graphic elements will not be drawn.

For example, the two instructions :

```
MODE 1

ORIGIN 320, 200, 100, 540, 350, 50, CL_BLACK, CL_BRIGHT_GREEN
```

generate the following image:



The origin (0,0) is placed in the centre. The scale goes from -220 to 220 on Ox and from -150 to 150 on Oy (the Oy axis points upwards, according to the Amstrad convention).

Using `ORIGIN` without parameters restores the initial screen.

The sample program `origin.bas` shows a more complete example with 2 windows.

POINTS AND LINES

In the following :

- `x, y` are the coordinates of a point (in pixels)

- `dx`, `dy` are the displacement from the current point (in pixels)
- `col` is the color (optional parameter)

The following statements are available:

```
MOVE x%, y%           ' Move to (x,y)

PLOT x%, y%, col%     ' Plot point (x, y)

LDRAW x%, y%, col%    ' Draw a line from the current point
                        to the point (x,y)

TEST(x%, y%)          ' Function: moves the point to (x,y)
                      ' and returns the color of the point
```

Note : The original Amstrad statement `DRAW` has been renamed as `LDRAW` (Line Draw) to avoid confusion with the FreeBASIC `DRAW` statement.

The variants use the displacement from the current point:

```
MOVER dx%, dy%

PLOTR dx%, dy%, col%

LDRAWR dx%, dy%, col%

TESTR(dx%, dy%)
```

The sample program `plotfunc.bas` is a function plotter using these statements.

RECTANGLES, CIRCLES AND ELLIPSES

The following statements draw rectangles :

```
RECTANGLE x%, y%, w%, h%

RECTANGLE_FILL x%, y%, w%, h%
```

where (`x, y`) are the coordinates of the upper left corner, `w` and `h` the width and height of the rectangle. The second statement fills the rectangle with the foreground color.

The `ARC` statement draws a circular or elliptical arc:

```
ARC xc%, yc%, rx%, ry%, a1%, a2%
```

The arc is centered at (`xc`, `yc`) with radii `rx` and `ry`, and is drawn between the angles `a1` and `a2` (in degrees)

The last 3 parameters are optional, the default values are 0, 0 and 360. The value `ry = 0` causes a circle to be drawn.

To draw a sector, give negative values to the angles, without zero values (use for example -0.01 instead of 0).

The `PIE` statement draws a circle or ellipse filled with the foreground color:

```
PIE xc%, yc%, rx%, ry%
```

This statement does not accept angular parameters, so it can only draw complete circles or ellipses.

FILLING SURFACES

The `FILL` statement fills an area with a color. Before using this statement, the pointer must be placed inside the surface (e. g. with `MOVE`).

```
FILL paint_col%, border_col%
```

- `paint_col` is the fill color
- `border_col` is the color of the line delimiting the surface; this parameter is optional, its default value is the foreground color

The `FILL_PATTERN` statement fills with a pattern created by `SYMBCOL` and assigned to a character :

```
FILL_PATTERN numchar%, border_col%
```

where `numchar` is the character number ; `border_col` works as with the `FILL` statement.

The sample program `fill_pattern.bas` creates a red cross pattern and uses it to fill a circle.

MOUSE TEST

The `GET_MOUSE` statement returns the mouse parameters:

```
GET_MOUSE x%, y%, btn%, wheel%
```

- `x`, `y` : position of the mouse, expressed in pixels, taking into account if necessary the coordinates defined by `ORIGIN`
- `btn` : button code, according to the table below
- `wheel` : position of the wheel: 0 at the start of the program, negative if the wheel is turned towards the user, positive otherwise.

The mouse buttons are denoted by symbolic constants :

```
BUTTON_LEFT   (1)
BUTTON_RIGHT  (2)
BUTTON_MIDDLE (4)
```

The sample program `get_mouse.bas` shows this usage.

GRAPHIC FILES

The following statements load an image in BMP, PNG, JPG, TGA or DDS format (non-interlaced formats only) :

```
IMG_LOAD filename$

IMG_LOAD_TRANS filename$
```

where `filename` is the name of the image file (the extension can be omitted for a BMP file). The image is displayed in the area defined by the `ORIGIN` statement.

The second version makes the background of the image transparent. The background color is defined by the color of the pixel located at the top left of the image.

The sample program `img_load.bas` shows the loading of an image with or without a transparent background.

The `IMG_SAVE` statement saves the image to a BMP or PNG file. Only the area defined by the `ORIGIN` statement is saved.

```
IMG_SAVE filename$, trans%
```

where `trans` is an optional parameter indicating if the transparency information must be saved in the case of a PNG file (default value = FALSE).