Appendix

Revision 0.8 – May 2017

Index

1. MSX Screens Memory and File Structure

The MSX video memory (VRAM) is another memory system, physically separated from the main memory system (RAM). The VRAM size depends on the MSX model, ranging from 16 Kb on MSX 1 to 128 Kb on MSX 2 and 2+.

The video memory has the information to be drawn on the screen, rendered by the video processor. The cycle of re-drawing the screen may vary from 50 or 60 Hz, depending on the region. MSX 2 and MSX 2+ have a built-in switch which allows to choose the right frequency.

If one wishes to save the screen contents in the MSX, he/she merely make a dump (copy) of the video memory corresponding to the screen area, with no compressing or image processing.

Nevertheless, the resulting file must have a 7-byte header containing some useful informations. This header is a standard MSX file header. The header structure is shown in the next table.

Byte	Description
00	Type of file description: FF = Basic program, FE = Binary program
01	Initial address LSB
02	Initial address MSB
03	Final address LSB
04	Final address MSB
05	Execution address LSB
06	Execution address MSB

Table 1. MSX Header file structure.

Note: LSB means the “Least Significant Byte” from a 16 bits number, while MSB stands for “Most Significant Byte”. For example, the 16-bit hexadecimal number D400 may be represented as:

LSB → 00
MSB → D4

The number D400H really appears inverted in the memory: 00 D4 *!*

For image files, the byte 00 is always set to “FE”. Initial and execution addresses are always set to “00”. The only ranging value is the final address, which depends on the size of the screen.

2. MSX Screens Layout[7, 8 , 9, 10]

Screen 2 / 4

Characteristics	Description
Dimensions	256 x 192 (width x height).
Colors	MSX 1 = 16 fixed MSX 2 = 16 out of 512 (palette)
VRAM initial address	Basic instruction base(11) for colors and base(12) for pattern Generally, pattern: 0, colors: 2000H
File header	FE 0 0 FF 37 0 0
Size of each table	6144 (1800H)
Total size	12288 (&H3000) calculated as 256 x 192 x 2 / 8
Pixel x Memory	2 bytes = 8x1 pixels block
Color representation	Index value from 0 to 15, according to the MSX 1 color table
To save the screen	*10 screen 2* *20 rem drawing code* ... *100 bsave "name.s02",base(12),base(11)+6143,s*
To load the screen	*10 screen 2* *20 bload "name.s02",s*

MSX 1 has a 16 pre-defined color set, referred each one by an index value, ranging from 0 to 15, in any screen mode (see figure 1). This set of values used to represent the colors take only a nipple (half of byte or 4 bits).

Color	Name	Index	Binary	Hexadecimal
	Invisible	0	0000	0
	Black	1	0001	1
	Green	2	0010	2
	Light green	3	0011	3
	Dark blue	4	0100	4
	Blue	5	0101	5
	Dark red	6	0110	6
	Cyan	7	0111	7
	Red	8	1000	8
	Light red	9	1001	9
	Dark yellow	10	1010	A
	Light yellow	11	1011	B
	Dark green	12	1100	C
	Magenta	13	1101	D
	Gray	14	1110	E
	White	15	1111	F

Figure 1. MSX 1 colors.

MSX Screen 2 has 49152 pixels and it takes 24576 bytes to store the color information from all pixels. Nevertheless, MSX 1 VRAM (video memory) is limited to only 16384 bytes. It is clear that it is not possible to store the color of all pixels together. Thus, the monochomatic storage (pixel on/off), which takes 6144 bytes, is perfectly possible.

In order to solve this issue, the MSX engineers applied the following trick: they grouped 8 pixels in line, using two bytes to represent the whole group. One byte stores the information about two different colors – the "front color" and the "background color". The remaining byte defines the pattern of each pixel – if its color is front or background.

The configuration of the byte representing the color is the following:

Bit	7	6	5	4	3	2	1	0
Color	F	F	F	F	B	B	B	B

Where F is the front color and B is the background color.

The pattern byte controls pixels individually, from left to right. The value “1” means that the pixel will assume the front color and “0” means that the pixels will assume the background color.

For example: suppose a 8x1 pixels block, where the colors are disposed in an alternated way. The front color is black (1) and the background is dark-blue (4). In that case, the color and pattern bytes are:

```
Color:   &B11110100
Pattern: &B10101010
```

The pattern table is stored in VRAM from 0000H to 17FFH, while the color is stored in VRAM from 2000H to 37FFH.

For example, if we want to paint the block beginning at the coordinates 0,0 using dark blue for one half and red for the other half, we must do:

Blue color code: 4 or 100 in binary.
Red color code: 8, or 1000 in binary.

VRAM (0000H) = &B11110000
VRAM (2000H) = &B01001000

Change VRAM in basic: vpoke *address*, *value*

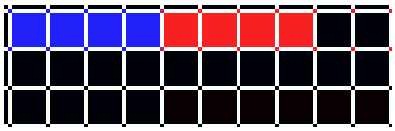


Figure 2. 8x1 pixel configuration example.

Until here we understand the way to configurate the 8x1 pixels block. Now, we must take in mind how MSX maps the 8x1 blocks in memory.

The next level of mapping consists in 8x8 pixels blocks, where each line is a 8x1 block. The mapping is done following the way from left to right and then from the top to bottom, using these 8x8 blocks, as seen in figure 3.

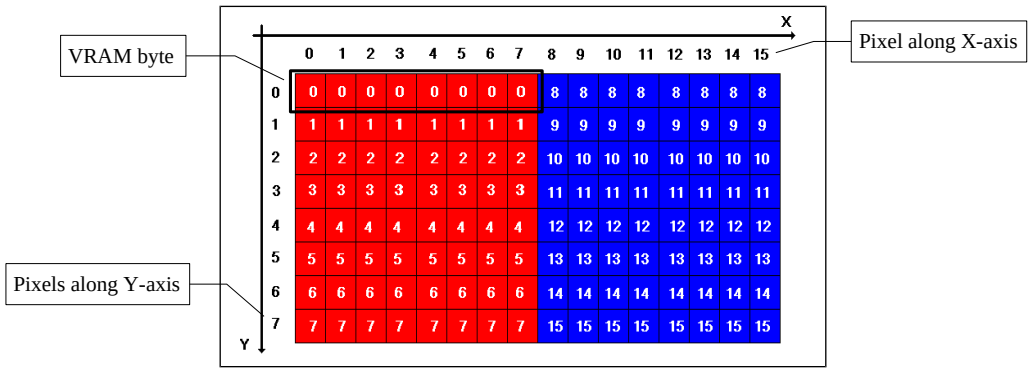


Figure 3. Screen structure x memory structure.

Figure 3 shows the first two 8x8 blocks, where the block 0 is marked as red and the block 1 is marked as blue. Notice that each line represents a VRAM address. The whole 8x8 block map is shown below:

0	1	2	3	4	5	6	7	...	31
32	33	34	35	36	37	38	39	...	63
...	...	...	...	...	...	...	...	...	...
736	737	738	739	740	741	742	743	...	767

The following program in C shows how to map from an image matrix to MSX VRAM.

Program in C to map from image matrix to screen 2 memory

```
for (y=0; y<192; y++)
{
  for (x=0; x<256; x+=8)
  {
    for (int i=0; i<8; i++)
      pixel_group[i] = img.getPixel(x+i, y);

    int color = find_color(pixel_group);
    int pattern = find_pattern(pixel_group, color);

    p = floor(x/8)*8 + y%8 + floor(y/8)*256;

    pattern_table[p] = pattern;
    color_table[p] = color;
  }
}
```

Group pixels in a 1x8 vector.

Find color: pick up the most two colors (histogram) and set them as front and back colors.

Find pattern: for a given front and back colors, use Euclidian Distance to find the most similar color to each pixel in the group. Set 0 to those closer to back color, and 1 those closer to front color.

As told before, the pattern table is located in VRAM address from 0000H to 17FFH, while the color table is located from 2000H to 37FFH. Notice that there is an area between the tables, precisely from 1800H to 1FFFH which contains some important information that must be preserved.

In that case, if we want to save a “dump” image from screen 2, we must save the VRAM data from 0000H to 3FFFH, including the area between the tables. In the same way, if we generate a screen 2 “dump” image format outside MSX, we must also include that data in our file. Another solution for that is to save each table in a separeted file and load each one alone.

If you want to deal with images using two files, do the following:

Save	Load
10 SCREEN 2 20 ' DRAWING CODE ... 200 BSAVE”PATTERN.S02”,0,&H17FF,S 210 BSAVE”COLOR.S02”,&H2000,&H37FF,S	10 SCREEN 2 20 BLOAD”PATTERN.S02”,S 30 BLOAD”COLOR.S02”,S 40 GOTO 40

Each file header contains the information about the correct VRAM address.

About the MSX image format, we have two common formats:

- ✔ Image dump – where the image is a deep VRAM copy and it is loaded directly to the VRAM by the Basic instruction **bload"file",s**.
- ✔ Binary executable file – where the file is an executable program which contains the image and it is responsible for loading the image to the VRAM. This file is loaded by the

Basic instruction **bload"file",r**. One of the advantages here is that some display animations are possible to be created like left-to-right, top-down etc.

There is a way to convert from binary image “,r” to dump image “,s”:

```
10 screen 2
20 bload “myscreen.scr”,r
30 bsave “mynewscreen.grp”, base(12),base(11)+6143,s
```

MSX 1 Real Colors Table

#	Color	Y	R-Y	B-Y	R	G	B	R	G	B
0	Transparent									
1	Black	0.00	0.47	0.47	0.00	0.00	0.00	0	0	0
2	Medium green	0.53	0.07	0.20	0.13	0.79	0.26	33	200	66
3	Light green	0.67	0.17	0.27	0.37	0.86	0.47	94	220	120
4	Dark blue	0.40	0.40	1.00	0.33	0.33	0.93	84	85	237
5	Light blue	0.53	0.43	0.93	0.49	0.46	0.99	125	118	252
6	Dark red	0.47	0.83	0.30	0.83	0.32	0.30	212	82	77
7	Cyan	0.73	0.00	0.70	0.26	0.92	0.96	66	235	245
8	Medium red	0.53	0.93	0.27	0.99	0.33	0.33	252	85	84
9	Light red	0.67	0.93	0.27	1.13	0.47	0.47	255	121	120
10	Dark yellow	0.73	0.57	0.07	0.83	0.76	0.33	212	193	84
11	Light yellow	0.80	0.57	0.17	0.90	0.81	0.50	230	206	128
12	Dark green	0.47	0.13	0.23	0.13	0.69	0.23	33	176	59
13	Magenta	0.53	0.73	0.67	0.79	0.36	0.73	201	91	186
14	Gray	0.80	0.47	0.47	0.80	0.80	0.80	204	204	204
15	White	1.00	0.47	0.47	1.00	1.00	1.00	255	255	255

This table was taken from the openMSX Project, version 0.7.0, file:VDP/Renderer.cc

Screen 3

Characteristics	Description
Dimensions	256 x 192
Colors	MSX 1 = 16 fixed / MSX 2 = 16 out of 512 (palette)
VRAM initial address	0
Header	FE 00 00 00 06 00 00
Size	1536 calculated as 32 x 48 small blocks
Pixel x Memory	1 byte for each 8x4 pixels block
Color representation	Index value from 0 to 15, according to the MSX 1 color table
To save the screen	10 screen 3 20 rem drawing code ... 100 bsave “name.s03”,0,1535 ,s
To load the screen	10 screen 3 20 bload “name.s03”,s

Like screen 2, screen 3 is also mapped by blocks. Each byte represents a 8x4 pixels block on the screen. The highest four bits represents the color of the left 4x4 block, while the lowest four bits represents the right 4x4 block, both composing the 8x4 block.

According to figure 4, the 8x4 block number 0 is represented by the binary value 01000110, where “0100” indicates the dark blue (4) and the “0110” indicates the dark red (6). According to the same figure, eight 8x4 blocks are stacked, forming a 8x32 block called “big block”.

The figure 5 shows how the big blocks are mapped: from left to right, then from top to bottom.

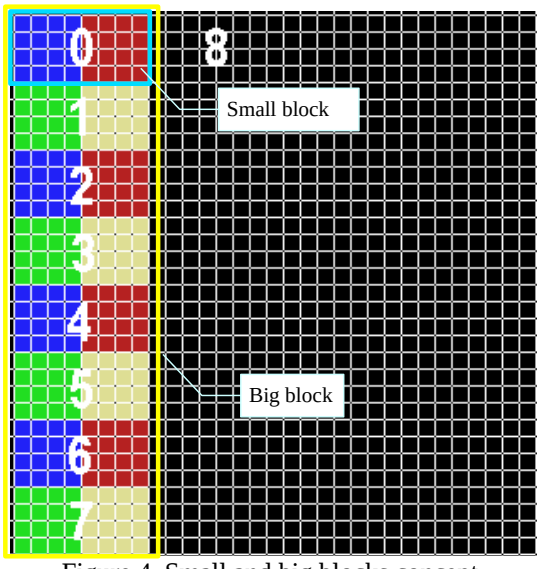


Figure 4. Small and big blocks concept.

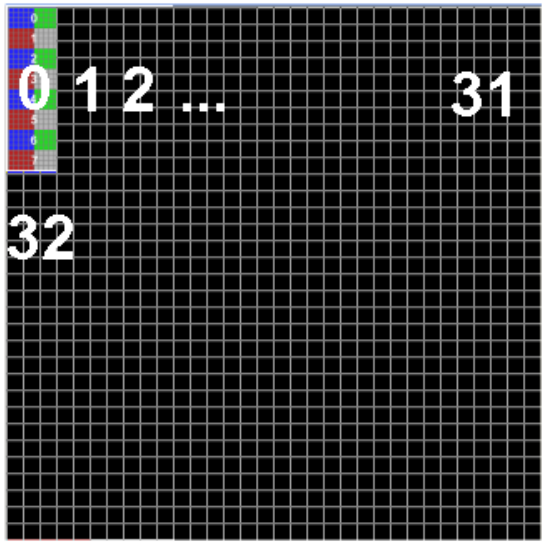


Figure 5. Mapping big blocks.

Formula to calculate the VRAM address using the coordinates x and y:

```
p = floor(x/8)*8 + floor(y/4) % 8 + floor(y/32)*256
```

Screen 5

Characteristics	Description
Dimensions	256 x 212
Interlaced mode	256 x 424
Pages	2
Colors	16 out of 512 (palette)
Header	FE 00 00 00 6A 00 00
VRAM address	Begin: 0 End: 27135 (&H69FF) Size: 256 x 212 / 2
Pixel x Memory	1 byte = 2 pixels (1 pixel = 4 bits)
Color representation	Palette index ranging from 0 to 15
To save the screen	10 screen 5 20 rem drawing code ... 100 bsave “name.s05”,0,27135,s
To load the screen	10 screen 5 20 bload “name.s05”,s

MSX 2 has innovated and increased considerably the size of the VRAM. Instead of 16 KB, now 128 KB were available. This capacity is not only enough to store all pixels data individually for one screen, but up to four screens.

In screen 5, each byte controls the color of two consecutive pixels (2x1 pixels block). The VRAM mapping here is simpler and more direct than the previous screens, as seen in table 2. As a consequence, the VRAM address calculation is faster than screen 2.

0	1	2	...	127
128	129	130	...	255
...	...	...	...	...
27008	27009	27010	...	27135

Table 2. Screen x memory arrangement in screen 5. The numbers represents the VRAM address.

The 2x1 pixels block is represented by one byte, as follows:

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
pixel *p*				pixel *p+1*			

Figure 6 illustrates the screen 5 mapping from pixel to VRAM.

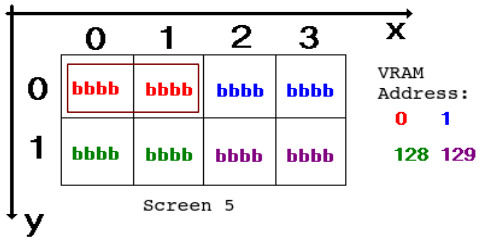


Figure 6. Mapping from screen 5 to VRAM. Symbol “b” means bit.

Formula to calculate the VRAM address using the coordinates x and y:

```
p = floor(x/2) + y*128
```

Once this screen structure can control individually the pixels, we face no color clash[1] problem as seen in screen 2.

MSX 2 introduced the concept of “pages”. As soon as the video memory increased to 128 KB, it was possible to segment it into pages. For example, the screen 5 uses only 27136 bytes of 128 KB to store data. According to that, we can store up to four screen 5 data in the VRAM. In fact, MSX divides the VRAM into pages of 32 KB. This resource is useful for animations and background work.

MSX 2 uses a little trick to increase the vertical size of the screen from 212 to 424 pixels called interlacing. The original image of 424 pixels height is divided into two images, one containing the even lines, and the other the odd lines. Each image is then loaded into different pages. Then, the hardware changes quickly the exhibition of the pages, giving us the impression to see only one image.

This code loads and shows an image in interlaced mode:

```
10 screen 5,,,,,3
20 bload “evenlinesscreen.sr5”,s
30 set page 1,1
40 bload “oddlinesscreen.sr5”,s
50 goto 50
```

First image has only even lines from original image.

Second image has only odd lines from original image.

Different from MSX 1, MSX 2 allow us to configurate colors. Each color is created by a combination of three additive primary colors: red (R), green (G) and blue (B). Each color component is a discrete value from 0 to 7, giving us a 7^3 or 512 different colors. On PCs, each color component ranges from 0 to 255, giving us 16 million colors.

There are two ways to represent a pixel:

- ✔ The direct RGB color values; and
- ✔ An index to a table containing the RGB color values.

In the direct RGB color, each pixel takes 9 bits to store data (3 bits for each color component). In the table system, called palette, each index holds the indirect color information about the pixel. This index represents a row from the nx3 table containing the real RGB colors. In that case, the number of bits to hold the data depends on the number of different colors represented by this table (number of rows).

MSX 2 palette holds 16 different colors and it is represented by a 16x3 table. In that case, MSX 2 screens based on palette can display only 16 colors at the same time, but each one using one of 512 possible colors. This palette takes 4 bits to represent an index.

The palette is also interesting for one reason: once changed the palette data from a given index, the color of all the pixels represented by that index are immediately changed.

Syntax in Basic to configurate palette:

```
COLOR=(index, r, g, b)
```

MSX 2 is also able to change screens 0, 1, 2, 3 and 4 RGB values using palette.

The table 3 shows the default palette configuration. These colors are the MSX 1 default colors (see figure 1).

Color index	R	G	B
0	0	0	0
1	0	0	0
2	1	6	1
3	3	7	3
4	1	1	7
5	2	3	7
6	5	1	1
7	2	6	7
8	7	1	1
9	7	3	3
10	6	6	1
11	6	6	4
12	1	4	1
13	6	2	5
14	5	5	5
15	7	7	7

Table 3. Corresponding color configuration for MSX1 color table.

Using table 3, we can understand that a pixel represented by the number 7 has, in fact, the red intensity of 2, green intensity of 6 and blue intensity of 7. If this pixel were represented by direct color, the corresponding binary value should be “0010 0110 0111”. Later, we will see that screen 8 uses direct RGB values.

MSX 2 screens 5, 6 and 7 use palette. In that case, the value represented for each pixel is the palette index (or table index).

Screen 6

Characteristics	Description
Dimensions	512 x 212
Interlaced mode	512 x 424
Pages	4
Colors	4 out of 512 (palette)
Header	FE 00 00 00 6A 00 00
VRAM address	Begin: 0 End: 27135 (&H69FF) Size: 512 x 212 / 4
Pixel x Memory	1 byte = 4 pixels (1 pixel = 2 bits)
Color representation	Palette index ranging from 0 to 3
To save the screen	10 screen 6 20 rem drawing code ... 100 bsave “name.s06”,0,27135,s
To load the screen	10 screen 6 20 bload “name.s06”,s

Screen 6 works like screen 5, but differs on the width resolution and the number of colors. Once each pixel is represented by 2 bits, it can only hold 4 different colors. This configuration allows the screen 6 to have a good resolution as screen 7 and also have 4 pages. Each byte represents 4 pixels, as follows:

b7	b6	b5	b4	b3	b2	b1	b0
pixel		pixel + 1		pixel + 2		pixel +3	

Figure 7 shows an example of binary value 00011011 for address 0.

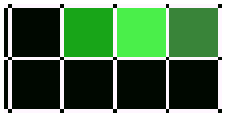


Figure 7. Screen 6.

Screen 7

Characteristics	Description
Dimensions	512 x 212
Interlaced mode	512 x 424
Pages	2
Colors	16 out of 512 (palette)
Header	FE 00 00 00 D4 00 00
VRAM address	Begin: 0 End: 54271 (&HD3FF) Size: 512 x 212 / 2
Pixel x Memory	1 byte = 2 pixels (1 pixel = 4 bits)
Color representation	Palette ranging from 0 to 15
To save the screen	10 screen 7 20 rem drawing code ... 100 bsave “name.s07”,0,54271,s
To load the screen	10 screen 7 20 bload “name.s07”,s

This screen is similar to screen 5, except for the fact that it has 512 pixels width.

Once it uses twice as much the pixels, it uses twice as much the video memory and, consequently, the number of video pages is decreased to 2.

This screen mode holds the best image resolution for MSX.

Screen 8

Characteristics	Description
Dimensions	256 x 212
Interlaced mode	256 x 424
Pages	2
Colors	256 fixed
Header	FE 00 00 00 D4 00 00
VRAM address	Begin: 0 End: 54271 (&HD3FF) Size: 256 x 212
Pixel x Memory	1 byte = 1 pixel
Color representation	1 byte controls 1 pixel Each byte represents RGB information: GGGRRRBB R and G value range from 0 to 7, while blue ranges from 0 to 3
To save the screen	10 screen 8 20 rem drawing code ... 100 bsave “name.s08”,0,54271,s
To load the screen	10 screen 8 20 bload “name.s08”,s

In screen 8, each pixel value represents directly the color. Thus, no more palette is used in this screen mode.

Here, all the possible 512 colors should be used, but it faces one problem: 9 bits are necessary to store pixel information, but there are only 8 bits available for each byte. To solve this issue, one bit was removed from the blue color component, once this is the least perceived color by the human eye. Screen 8 can show 256 colors at the same time.

The screen x memory structure is similar to screens 5 and 7, except that screen 8 uses 1 byte to represent 1 pixel.

Screen 8 color configuration:

b7	b6	b5	b4	b3	b2	b1	b0
g	g	g	r	r	r	b	b
pixel							

Examples of color representation:

Example 1:	Example 2:
R=5(&b101) G=6(&b110) B=2(&b10) Pixel: 11010110 = 214	R=2(&b010) G=0(&b000) B=3(&b11) Pixel: 00001011 = 11

This screen mode holds the best color resolution for MSX 2.

MSX Screen 8 to PC color mapping:

RED		GREEN		BLUE	
MSX	PC 24-bit	MSX	PC 24-bit	MSX	PC 24-bit
0	0	0	0	0	0
1	36	1	36	1	73
2	73	2	73	2	146
3	109	3	109	3	255
4	146	4	146	4	-
5	182	5	182	5	-
6	219	6	219	6	-
7	255	7	255	7	-

Program in C to create the above table

```
#include <stdio.h>
#include <math.h>

int main(void)
{
        double f = 255.0/7.0, b_f =7.0/3.0;

        for (unsigned int i=0; i<8; i++)
        {
                if (i<4)
                        printf("MSX(%d) – R=%d – G=%d – B=
%d\n",i,int(round(i*f)),int(round(i*f)),int(round(int(i*b_f)*f)));
                else
                        printf("MSX(%d) – R=%d – G=%d –
B=-\n",i,int(round(i*f)),int(round(i*f)));
        }

        return 0;
}
```

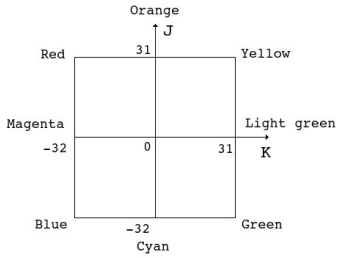
Screens 10, 11 and 12

Characteristics	Description
Dimensions	256 x 212
Interlaced mode	256 x 424
Pages	2
Colors	19268 fixed
Header	FE 00 00 00 D4 00 00
VRAM address	Begin: 0 End: 54271 (&HD3FF) Size: 256 x 212
Pixel x Memory	1 byte = 4x1 pixels
Color representation	YJK system, where each 4x1 pixels block are represented as: Byte 1 - (x,y) : Y_14 Y_13 Y_12 Y_11 Y_10 K2 K1 K0 Byte 2 - (x+1,y): Y_24 Y_23 Y_22 Y_21 Y_20 K5 K4 K3 Byte 3 - (x+2,y): Y_34 Y_33 Y_32 Y_31 Y_30 J2 J1 J0 Byte 4 - (x+3,y): Y_44 Y_43 Y_42 Y_41 Y_40 J5 J4 J3 The Y element is independent for each pixel and controls the brightness K and J control the color The big numbers mean the corresponding binary arragenment The small numbers (used in Y) mean the pixel
To save the screen	10 screen 12 20 rem drawing code ... 100 bsave “name.s12”,0,54271,s
To load the screen	10 screen 12 20 bload “name.s12”,s

Screens 10, 11 and 12 deals with a group of 4x1 pixels, which represents a YJK system:

Byte 1 - (x,y) :	Y_14	Y_13	Y_12	Y_11	Y_10	K2	K1	K0
Byte 2 - (x+1,y):	Y_24	Y_23	Y_22	Y_21	Y_20	K5	K4	K3
Byte 3 - (x+2,y):	Y_34	Y_33	Y_32	Y_31	Y_30	J2	J1	J0
Byte 4 - (x+3,y):	Y_44	Y_43	Y_42	Y_41	Y_40	J5	J4	J3

The Y component is independent for each pixel and controls the brightness.
The K and J components work together for all 4 pixels and control the color.
The RGB system must range from 0 to 31 each color component.



The conversion from YJK system to RGB is done as follows:

$R_n = Y_n + J$
$G_n = Y_n + K$
$B_n = 5/4*Y_n - J/2 - K/4$

Where n is one of the 4-pixel block.

We must keep in mind that:

```
if J > 31 then J=J-64
if K > 31 then K=K-64

if R < 0 then R = 0 else if R > 31 then R = 31
if G < 0 then G = 0 else if G > 31 then G = 31
if B < 0 then B = 0 else if B > 31 then B = 31
```

The conversion from RGB system to YJK is done as follows:

To calculate J and K:

$$Y_m = B_m/2 + R_m/4 + G_m/8$$
$$J = R_m - Y_m$$
$$K = G_m - Y_m$$

Where:

$$R_m = (R_1 + R_2 + R_3 + R_4)/4$$
$$G_m = (G_1 + G_2 + G_3 + G_4)/4$$
$$B_m = (B_1 + B_2 + B_3 + B_4)/4$$

To compose pixel bits, K and J must range from 0 to 63.

```
if J<0 then J=J+64
if K<0 then K=K+64
```

To calculate each Y value:

$$Y_1 = B_1/2 + R_1/4 + G_1/8$$
$$Y_2 = B_2/2 + R_2/4 + G_2/8$$
$$Y_3 = B_3/2 + R_3/4 + G_3/8$$
$$Y_4 = B_4/2 + R_4/4 + G_4/8$$

In order to verify the YJK system, use the following example:

```
10 SCREEN 12
20 FOR Y=0 TO 10
30 K=Y*256
40 FOR X=0 TO 255 STEP 4
50 VPOKE X+K  ,&B00000000
60 VPOKE X+K+1,&B00000000
70 VPOKE X+K+2,&B00000000
80 VPOKE X+K+3,&B00000000
90 NEXT X,Y
100 GOTO 100
```

Color	Y	J	K
Red	0	31	-32
Green	0	-32	31
Blue	0	-32	-32
Yellow	31	31	31
Magenta	31	0	-32
Orange	16	31	0
Light green	16	0	31
Cyan	31	-32	0
Black	0	0	0
White	31	0	0

According to the JK chart and the table seen before, the yellow color has K=32, J=32 and Y=0. In that case, K=&B011111 and J=&B011111. Change the program to:

```
50 VPOKE X+K  ,&B00000111
60 VPOKE X+K+1,&B00000011
70 VPOKE X+K+2,&B00000111
80 VPOKE X+K+3,&B00000011
```

Differences between screens 10, 11 and 12

For each pixel in screens 10 and 11 the bit 3 works like a switch. If this bit is set to 0, then the pixel is working with YJK system. If it is set to 1, then the pixel is working with RGB system.

Then, the configuration is:

```
Y3 Y2 Y1 Y0 B K2 K1 K0
Y3 Y2 Y1 Y0 B K5 K4 K3
Y3 Y2 Y1 Y0 B J2 J1 J0
Y3 Y2 Y1 Y0 B J5 J4 J3
```

For each pixel,
If B = 0, then YJK system is used
if B = 1, then RGB system is used 0-15 (and K or J is ignored)

To convert a screen 12 screen to screen 10 or 11:

Option 1	Option 2

`10 screen 12`
`20 bload "myscreen.pic",s`
`30 screen 10 : ' Or screen 11`
	`10 screen 11 : ' Only screen 11`
	`20 bload "myscreen.pic",s`
	`30 line(0,0)-(255,211),&B11110111,bf,and`

This will make sure that all pixels will work in YJK system.

For RGB system (B=1), only the Y is used. This pixel will work independently from the rest of the group, but the K or J information stills working for the whole group.
For example, to paint a pixel with magenta (color code 13 or &B1101):

1 1 0 1 1 X X X

The part marked with yellow background means the color palette index. The '1' marked with the cyan color means the RGB system set/unset. The area marked with green color means the J or K code that must be sometimes preserved. If there is no information to be preserved, use 000.
Screen 11 uses Basic instructions like LINE, CIRCLE, PAINT, using directly the byte color code (0-255). In the other hand, screen 10 uses MSX 2 index system (from 0 to 15).

Drawing a line in screen 11:

```
LINE (10,10)-(245,202),7,,AND       ' Reset Y
LINE (10,10)-(245,202),4*16+8,,OR   ' Set color 4 (0100 → 01000000 → 01001000)
```

Drawing a line in screen 10:

```
LINE (10,10)-(245,202),4
```

V9990

The V9990 is a powerful video processor developed by Yamaha and adapted for MSX in two different projects: GFX9000 by Sunrise[13] and Powergraph by Tecnobytes[14].

In both projects, the new video processor was not incorporated directly to MSX as MSX 2 and 2+ video processors did. In both projects, they work as an extra video processor attatched to the MSX via cartridge and having an independent output. In other words, MSX starts to count on two independent video processors.

Sunrise developed a Basic extension to MSX called G-Basic[15] in order to support the V9990. It defines 19 new video modes for MSX and all are supported by MSX Viewer 5.

Some new features are available:

- 32768 colors at the same time
- 5-bit palette (0-31) and 64 colors at the same time
- 32 x 32 pixels sprite
- 512 Kb VRAM
- Imagespace concept (pages diposed as a big screen)
- Tranparency

G-Basic file header description:

Byte	Description
00	0xFE = binary program
01	Initial address LSB
02	Initial address MSB
03	Final address LSB
04	Final address Mid-SB
05	Image space flag: 0x00 = single image / 0x80 = image space
06	Final address MSB – Final address bytes composed in this order: 06, 03, 02

Please, check out G-Basic manual[15] to a detailed description of each V9990 screen mode.

SymbOS SGX

SymbOS[17] is a multi-task operational system developed for MSX which supports free-size images. This specific format is called SymbOS Graphic File or SGX[16]. There are two versions of it: 4 or 16 colors.

MSX Viwer 5 also supports 4 and 16 SGX file formats. For details on SGX implementation, please check out SGX documentation[16].

openMSX

If you do not afford neither MSX 2 nor V9990, you can experience them using openMSX emulator.

3. Copy Instruction from Basic

Since MSX2 or higher, the Basic instruction COPY was introduced in order to copy any region (rectangle) of screen to another region, page or save it to a file.

Copy syntax:

COPY <origin> TO <destination>
COPY (x1, y1)-(x2, y2), page TO (x1, y1), page, operation
COPY (x1, y1)-(x2, y2), page TO “file”
COPY “file”, orientation TO (x1, y1), page, operation

We can save an image area, as seen in figure 8.

Figure 8. Image region.

This 8x2 image region can be saved in a file and loaded into any screen position or even another page. It is defined by the origin position (x,y) and the orientation. The default orientation to load the rest of the path is right and down.

The saved file has a different header from the common MSX image files. It has 4 bytes. The 2 first bytes indicate the width of the copy region, while the second pair of bytes indicates the height of the region. For the example shown in figure 8, the header should be: 08 00 02 00.

After the head, comes the screen data. If the region shown in figure 8 belongs to screen 5, the corresponding data should be: 01 23 45 67 FF FF FF FF.

The whole file is: 08 00 02 00 01 23 45 67 FF FF FF FF

4. ASCII Codes

Default ASCII table

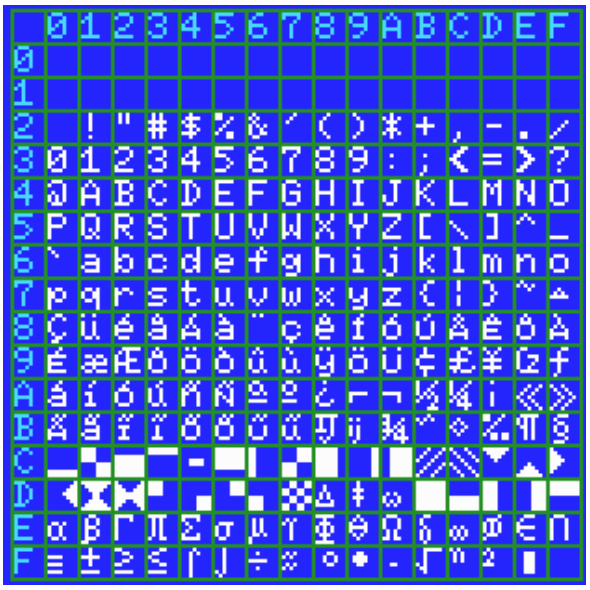
ASCII	Hex	Symbol	ASCII	Hex	Symbol	ASCII	Hex	Symbol	ASCII	Hex	Symbol
0	0	NUL	16	10	DLE	32	20	SPC	48	30	0
1	1	SOH	17	11	DC1	33	21	!	49	31	1
2	2	STX	18	12	DC2	34	22	“	50	32	2
3	3	ETX	19	13	DC3	35	23	#	51	33	3
4	4	EOT	20	14	DC4	36	24	$	52	34	4
5	5	ENQ	21	15	NAK	37	25	%	53	35	5
6	6	ACK	22	16	SYN	38	26	&	54	36	6
7	7	BEL	23	17	ETB	39	27	‘	55	37	7
8	8	BS	24	18	CAN	40	28	(	56	38	8
9	9	TAB	25	19	EM	41	29	)	57	39	9
10	A	LF	26	1A	SUB	42	2A	*	58	3A	:
11	B	VT	27	1B	ESC	43	2B	+	59	3B	;
12	C	FF	28	1C	FS	44	2C	´	60	3C	<
13	D	CR	29	1D	GS	45	2D	-	61	3D	=
14	E	SO	30	1E	RS	46	2E	.	62	3E	>
15	F	SI	31	1F	US	47	2F	/	63	3F	?

ASCII	Hex	Symbol	ASCII	Hex	Symbol	ASCII	Hex	Symbol	ASCII	Hex	Symbol
64	40	@	80	50	P	96	60	`	112	70	p
65	41	A	81	51	Q	97	61	a	113	71	q
66	42	B	82	52	R	98	62	b	114	72	r
67	43	C	83	53	S	99	63	c	115	73	s
68	44	D	84	54	T	100	64	d	116	74	t
69	45	E	85	55	U	101	65	e	117	75	u
70	46	F	86	56	V	102	66	f	118	76	v
71	47	G	87	57	W	103	67	g	119	77	x
72	48	H	88	58	X	104	68	h	120	78	w
73	49	I	89	59	Y	105	69	i	121	79	y
74	4A	J	90	5A	Z	106	6A	j	122	7A	z
75	4B	K	91	5B	[	107	6B	k	123	7B	(
76	4C	L	92	5C	\	108	6C	l	124	7C	|
77	4D	M	93	5D	]	109	6D	m	125	7D	)
78	4E	N	94	5E	^	110	6E	n	126	7E	~
79	4F	O	95	5F	_	111	6F	o	127	7F	␡

Description of ACII functions

ASCII	Hex	Symbol	Description
0	0	NUL	Null
1	1	SOH	Start of Heading
2	2	STX	Start of Text
3	3	ETX	End of Text
4	4	EOT	End of Tansmission
5	5	ENQ	Enquiry
6	6	ACK	Acknowledge
7	7	BEL	Bell
8	8	BS	Backspace
9	9	TAB	Horizontal Tabulation
10	A	LF	Line Feed
11	B	VT	Vertical Tabulation
12	C	FF	Form Feed
13	D	CR	Carriage Return
14	E	SO	Shift Out
15	F	SI	Shift In
16	10	DLE	Data Link Escape
17	11	DC1	Device Control 1
18	12	DC2	Device Control 2
19	13	DC3	Device Control 3
20	14	DC4	Device Control 4
21	15	NAK	Negative Acknowledge
22	16	SYN	Synchronous Idle
23	17	ETB	End of Transmission Block
24	18	CAN	Cancel
25	19	EM	End of Medium
26	1A	SUB	Substitute
27	1B	ESC	Escape
28	1C	FS	File Separator
29	1D	GS	Group Separator
30	1E	RS	Record Separator
31	1F	US	Unit Separator

Brazilian MSX (Hotbit)



Windows (Brazilian Layout)

121=y	136=ˆ	151=–	166=¦	181=µ	196=Ä	211=Ó	226=â	241=ñ
122=z	137=‰	152=˜	167=§	182=¶	197=Å	212=Ô	227=ã	242=ò
123={	138=Š	153=™	168=¨	183=·	198=Æ	213=Õ	228=ä	243=ó
124=|	139=‹	154=š	169=©	184=¸	199=Ç	214=Ö	229=å	244=ô
125=}	140=Œ	155=›	170=ª	185=¹	200=È	215=×	230=æ	245=õ
126=~	141=▯	156=œ	171=«	186=º	201=É	216=Ø	231=ç	246=ö
127=▯	142=Ž	157=▯	172=¬	187=»	202=Ê	217=Ù	232=è	247=÷
128=€	143=▯	158=ž	173=-	188=¼	203=Ë	218=Ú	233=é	248=ø
129=▯	144=▯	159=Ÿ	174=®	189=½	204=Ì	219=Û	234=ê	249=ù
130=‚	145=‘	160=	175=¯	190=¾	205=Í	220=Ü	235=ë	250=ú
131=ƒ	146=’	161=¡	176=°	191=¿	206=Î	221=Ý	236=ì	251=û
132=„	147=“	162=¢	177=±	192=À	207=Ï	222=Þ	237=í	252=ü
133=…	148=”	163=£	178=²	193=Á	208=Ð	223=ß	238=î	253=ý
134=†	149=•	164=¤	179=³	194=Â	209=Ñ	224=à	239=ï	254=þ
135=‡	150=–	165=¥	180=´	195=Ã	210=Ò	225=á	240=ð	255=ÿ

DOS

128=Ç	144=É	160=á	176=▒	192=└	208=ð	224=ó	240=–
129=ü	145=æ	161=í	177=▒	193=┴	209=Ð	225=ß	241=±
130=é	146=Æ	162=ó	178=▒	194=┬	210=Ê	226=Ô	242=‗
131=â	147=ô	163=ú	179=│	195=├	211=Ë	227=ò	243=¾
132=ä	148=ö	164=ñ	180=┤	196=─	212=È	228=õ	244=¶
133=à	149=ò	165=Ñ	181=Á	197=┼	213=ı	229=Õ	245=§
134=å	150=û	166=ª	182=Â	198=ã	214=Í	230=µ	246=÷
135=ç	151=ù	167=º	183=À	199=Ã	215=Î	231=Þ	247=¸
136=ê	152=ÿ	168=¿	184=©	200=╚	216=Ï	232=Þ	248=°
137=ë	153=Ö	169=®	185=╣	201=╔	217=┘	233=Ú	249=¨
138=è	154=Ü	170=¬	186=║	202=╩	218=┌	234=Û	250=·
139=ï	155=ø	171=½	187=╗	203=╦	219=█	235=Ù	251=¹
140=î	156=£	172=¼	188=╝	204=╠	220=▄	236=ý	252=³
141=ì	157=Ø	173=¡	189=¢	205=═	221=¦	237=Ý	253=²
142=Ä	158=×	174=«	190=¥	206=╬	222=Ì	238=¯	254=■
143=Å	159=ƒ	175=»	191=┐	207=¤	223=▀	239=´	255=

5. Converting Images from PC 24-bit to MSX

PC images are the state-of-art quality, having many different colors. Usually, such images are composed by 3 color channels, with 256 levels in each one. This color system is called true-type or 24-bit color.

The MSX also has 3 color channels, but with only 8 levels each one. This results on a small number of colors if compared to PC – 512 versus 16 millions. Once both computers the save color space discretized, we can see MSX colors as a subset of PC colors.

For those MSX screens which have few colors, such as the palette screens, the right color identification is not an easy task. Moreover, each type of image demands a suitable method to achieve good results. For example, drawings and poor color pictures generally have a few number of colors. In the other hand, high-quality photographies have millions of colors.

In MSX Viewer 5[6], four different methods to convert from PC 24-bit color are used: Color Quantization, Euclidean Distance, Error Diffusion and Ordered Dithering.

5.1. Color Quantization

This is a simple color mapping method, where each color component R,G,B is divided into N sets of colors, where 0 < N < 256. Each set may result or not on the same size, depending on the result of the division 256 / N.

For a given 24-bit pixel *P(x, y, c)*, where *c* is referred to one of the triplet R,G,B, we calculate the MSX corresponding color as follows:

```
f =  256.0 / N;
MSX[c] = floor (P(x, y, c) / f);
```

For MSX 2 palette system, N=8.

For example, lets take a pixel P(x,y) = { 255, 40, 128 }:

```
f = 256.0 / 8.0 = 32.0
MSX[red]   = floor(255.0 / 32.0) = floor(7.97) = 7
MSX[green] = floor(40.0 / 32.0)  = floor(1.25) = 1
MSX[blue]  = floor(128.0 / 32.0) = floor(4.00) = 4
```

It means that for the given pixel P(x,y) = { 255, 40, 128 }, the MSX palette equivalent R,G,B is (7, 1, 4).

Figure 9 illustrates the pixel quantization for N=8.

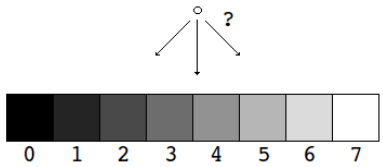


Figure 9: Color Quantization.

In Screen 8, the blue color is reduced by one bit. It leads to the following configuration:

```
Red:   N=8
Green: N=8
Blue:  N=4
```

MSX 2+ have a different palette system, where N=32 for each color component.

In order to map back from MSX colors to PC, use the following equations:

```
pixel(x, y, red)   = MSX[red]   * (255.0/N_red);
pixel(x, y, green) = MSX[green] * (255.0/N_green);
pixel(x, y, blue)  = MSX[blue]  * (255.0/N_blue);
```

5.2. Euclidean Distance

This method is used to find the best MSX palette color for a given PC color.

For each palette index, calculate the Euclidean Distance between the given PC color and the MSX palette color. The palette index where the smallest distance was found is then selected to represent that color.

The MSX palette must be converted to PC system, using MSX to PC mapping. See section 5.1.

The Euclidean Distance is calculated as follows:

$$distance = \sqrt{((palette(i,1) - pixel(x,y,1))^2 + (palette(i,2) - pixel(x,y,2))^2 + (palette(i,3) - pixel(x,y,3))^2)}$$

Where i is the i^{th} index of the palette, x,y the image pixel coordinate and the numbers 1, 2 and 3 the red, green and blue channels, respectively.

Program in C to find the best color for MSX 2 palette

```
double best_dist = sqrt(pow(palette(0,0) – pixel(x,y,0), 2) + pow(palette(0,1) – pixel(x,y,1), 2) + pow(palette(0,2) – pixel(x,y,2), 2));
int best_color = 0;
for (int i=1; i<256; i++)
{
  dist = sqrt(pow(palette(i,0) – pixel(x,y,0), 2) + pow(palette(i,1) – pixel(x,y,1), 2) + pow(palette(i,2) – pixel(x,y,2), 2));
  if (dist < best_dist)
  {
    best_dist = dist;
    best_color = i;
  }
}
```

Is it possible to convert MSX PC colors to Screen 8 using MSX palette system, even though the screen 8 do not use palette. In this case, we create a fake palette where the index is the real color. This method needs a table containing all possible values to compare and select the index of the best value.

Program in C to create screen 8 fake palette

```
double f = 255.0/7;
double b_f = 7.0 / 3.0;
int index = 0;

for (int r=0; r<8; r++)
{
  for (int g=0; g<8; g++)
  {
    for (int b=0; b<4; b++)
    {
      palette(index, 0) = floor(g * f);
      palette(index, 1) = floor(r * f);
      palette(index, 2) = floor(round(b * b_f) * f);
      index++;
    }
  }
}
```

5.3. Error Diffusion

In order to minimize the Mach Bands effects, the Error Diffusion[2,11] and Ordered Dither methods will generate some noise on the resulting image.

For a given pixel *(x,y)* in Error Diffusion, the current pixel is quantized. Then, for the pixel neighborhood, an error is calculated based on the color value and the color quantized. This error is then applied to the neighboring pixels. The error is never applied to already visited pixels. This is applied to each color channel separately.

Programs in C for Error Diffusion

```
for (int y=0; y<img.height(); y++)
{
  for (int x=0; x<img.width(); x++)
  {
    // Pixel quantization
    old_pixel = img(x,y);
    new_pixel = find_color(old_pixel);
    new_img(x,y) = new_pixel;

    // Calculate quantization error
    quant_error = old_pixel – new_pixel;

    // Spread error
    img(x+1,y) += quant_error * 7/16;
    img(x-1,y+1) += quant_error * 3/16;
    img(x,y+1) += quant_error * 5/16;
    img(x+1,y+1) += quant_error * 1/16;
  }
}
```

Quantization

Using quantization for the current pixel

```
for (int y=0; y<img.height(); y++)
{
  for (int x=0; x<img.width(); x++)
  {
    // Pixel quantization
    old_pixel = img(x,y);
    index = find_color(old_pixel);
    new_pixel = palette(index);
    new_img(x,y) = index;

    // Calculate quantization error
    quant_error = old_pixel – new_pixel;

    // Spread error
    img(x+1,y) += quant_error * 7/16;
    img(x-1,y+1) += quant_error * 3/16;
    img(x,y+1) += quant_error * 5/16;
    img(x+1,y+1) += quant_error * 1/16;
  }
}
```

Euclidean Distance

Using Euclidean Distance for the current pixel

Many pixels mask may be applied to apply error, but the most used is Floyd-Steinberg:

1/16			
	0	*	7
	3	5	1

Floyd-Steinberg mask.

The symbol '*' represents the current pixel *(x,y)*. This is where the mask should be placed.

It is important to notice that original image (img) must be converted from integer to double, while quantization error is a float point number. The new image is integer.

According to the programs, two strategies were adopted to calculate the current pixel: quantization (section 5.1) and Euclidean Distance (section 5.2). The resulting image for quantization is a RGB image, while for Euclidean Distance is an index image.

Important note: quantization must convert PC RGB to MSX RGB colors and return back to PC RGB colors (adjustment process). The first step is to fit the given color to the MSX 2 colors. The socond step is to return to PC colors that is being used by the converting program. See section 5.1 for more datails.

Quantization computation – $O(n^2)$ – is much more efficient than Euclidean Distance computation – $O(n^3)$. Nevertheless, MSX VRAM data from screens 2-7 are based on palette indexes. Once Euclidean Distance is applied, no changes are necessary to the resulting codes, which are really indexes. In the other hand, quantization needs an extra image processing using Euclidean Distance to find the indexes corresponding to PC colors.

Screen 8-12 do not use palette. They are based on MSX RGB colors. In that case, color quantization can be applied properly.

5.4. Ordered Dithering

Ordered Dithering[3,11] method uses a threshold map to generate noise on the resulting image. Bayer Matrix is used as a threshold map.

In MSX Viewer 5, a 4x4 Bayer Matrix is used. Like Error Diffusion, this method is applied to each color channel separately.

1/17				
	1	9	3	11
	13	5	15	7
	4	12	2	10
	16	8	14	6

4x4 Bayer Matrix

C++ code for Ordered Dithering

```
for (int y=0; y<img.height(); y++)
{
  for (int x=0; x<img.width(); x++)
  {
    old_pixel = img(x,y) + bayer[x%4][y%4];
    new_pixel = find_color(old_pixel);
    new_img(x,y) = new_pixel;
  }
}
```

Quantization

```
for (int y=0; y<img.height(); y++)
{
  for (int x=0; x<img.width(); x++)
  {
    old_pixel = img(x,y) + bayer[x%4][y%4];
    index = find_color(old_pixel);
    new_img(x,y) = index;
  }
}
```

Euclidean Distance

For *find_color(old_pixel)* function marked above and other considerations, see section 5.3.

5.5. Search for the Best Palette: K-Means

K-Means[5] is a iterative clustering algorithm used in data mining. The basic idea is to do the following: For a given a set $X=(x_1, x_2, \ldots, x_n)$, create K subsets of X; Each subset is represented by a centroid $C=(c_1, c_2, \ldots, c_k)$. Each iteration will assign elements to centroids (create subsets) and then re-calculate the centroids.

In other words, each iterations of K-Means will:

1. Assign elements to centroids: for each element of X, assign the cluster that has the shortest distance;
2. Re-calculate centroids: After assignments, calculate the new centroids values by the mean values of the elements assigned to each centroid.

The algorithm will stop when all centroids do not move between two iterations.

The given set X is always a mass of data. So, the initial centroid values are unknown. For solving that problem, we may initialize the cluster with random values.

Also, the number o K subsets is unknown. The user must give that number.

No matters how many dimensions have each element, it will be assigned to a centroid by a distance. The distance may be:

A. K-Means: Euclidean Distance: $d(P,C)=\sqrt{\sum_{i=1}^{n}(P_i-C_i)^2}$

B. K-Median: $d(P,C)=\sum_{i=1}^{n}|P_i-C_i|$

Figure 10 shows an example of K-Means in a two dimensional set of X. The number of subsets was set to K=3 and the number of elements to N=10. It took three iterations to converge.

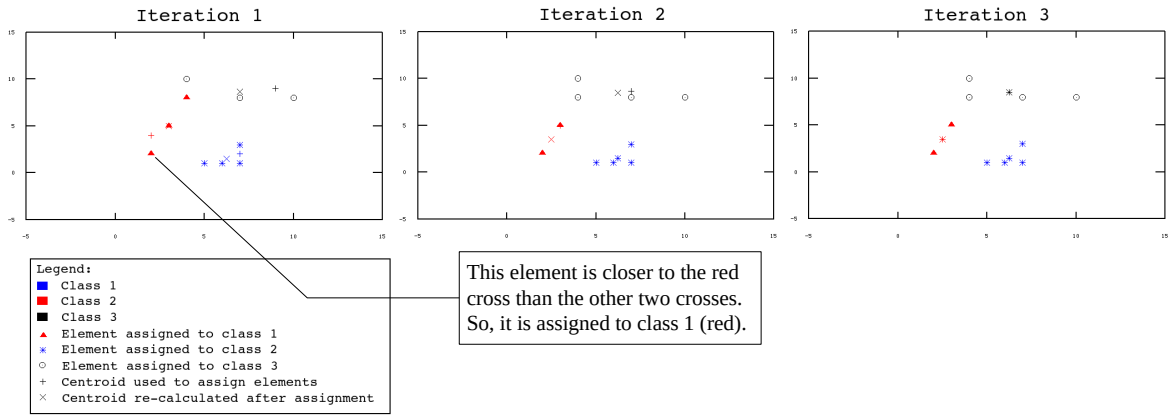


Figure 10: K-Means clustering.

As MSX 2 palette can be freely changed to any combination of RGB colors, ranging from 0-7 each color component, it will give us 512 (8^3) different colors.

In order to generate the best color combination palette for a given image, we will apply K-Means to find it.

Let's see how K-Means works for MSX 16 color palette:

The input set *X* is the PC input image. Let's adjust the image matrix to a single vector (see figure 11), where each element is a three-dimensional pixel data, represented by the RGB color system.

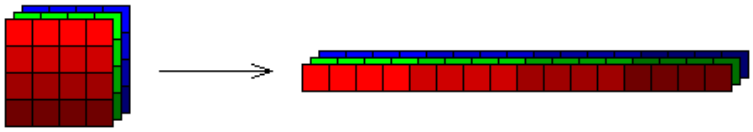


Figure 11: Conversion from image to vector.

The number of subsets or clusters are 16. Then, K=16.

The centroids are represented by each line of the palette (values R,G,B).

As said before, let's start the palette R,G,B values randomly.

Step 1: For each pixel, calculate the Euclidean Distance to each centroid:

$$d(P,C)=\sqrt{(P_R-C_R)^2+(P_G-C_G)^2+(P_B-C_B)^2}$$

Then, assign that pixel to the corresponding palette index found. Use 16 vectors to store pixels indexes, each one representing a centroid.

Step 2: Recalculate centroids after assign all pixels.

Repeat until centroids do not move (palette change) between two iterations.

For palette 64 out of 32K, use *K*=64.

Note: MSX palette must be converted to PC 24-bit format. See section 5.1.

During the K-Means execution, we may fall into two different traps:

A. Two or more centroids are overlapping; and

B. One or more centroids are hungry (no elements assigned).

These problems must be solved somehow. The “A” problem can be solved by moving centroids a little bit. The “B” problem can be solved by associating the hungry cluster to the most populated centroids and applying the solution for “A” problem.

In fact, overlapping situation lies on hungry situation, once just one of the centroids will be associated to the elements. According to that, we may apply another solution for the problem. As we may not re-calculate centroids for empty clusters, just repeat the values of these centroids. As associated centroid may move, this will naturally separate them.

On the next pages, figures 12 and 13 shows an example of these situations. See in figure 13 how the green cluster is empty, while red is full, enforcing that overlapping lies on hungry. Figure 14 shows the second solution using repeated centroids.

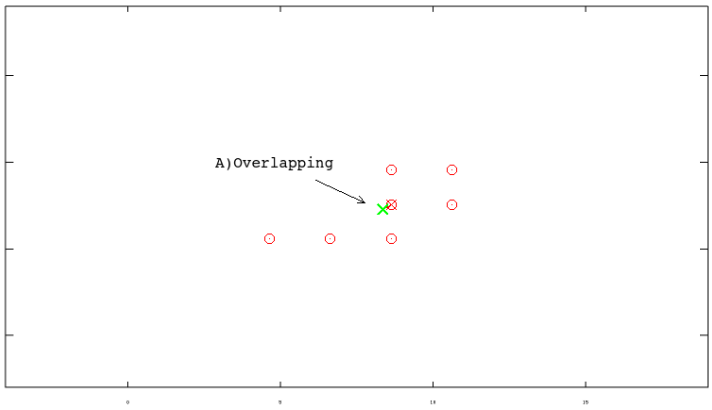


Figure 12. Problem “A” - Centroid overlapping.

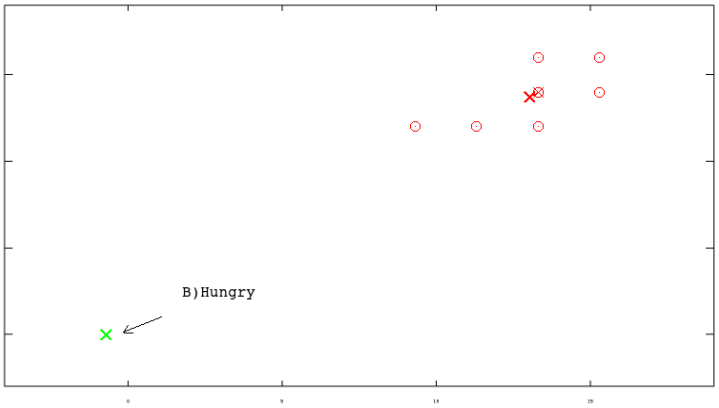


Figure 13. Problem “B” - Hungry element.

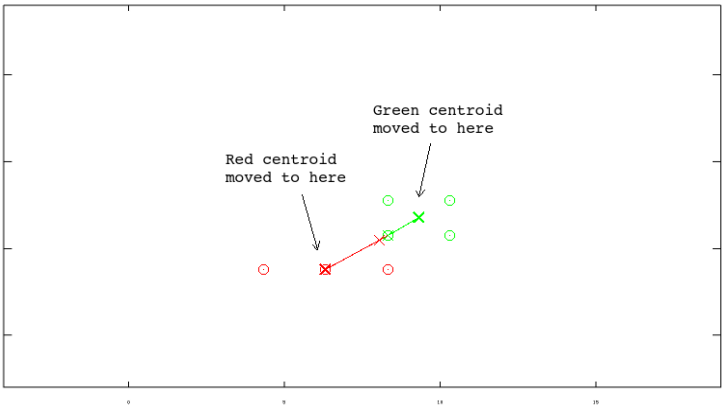


Figure 14. Solution using repeated centroid.

Important note:

if you are in PC color space (24-bit) using K-Means to find MSX Palette, you must apply color quantization and back on the palette (centroids). Otherwise, it will find palette colors between MSX intervals, something like “6.5, 5.4, 4.2”.
Even though converting to integer values, it will generate repeated values.

Fix it (for standard MSX), after each centroid re-calculation (step 2) or random initialization:

palette = floor(floor(palette * (8.0/256.0)) * 255.0/7.0);

5.6. Color Identification Using Fuzzy Logic

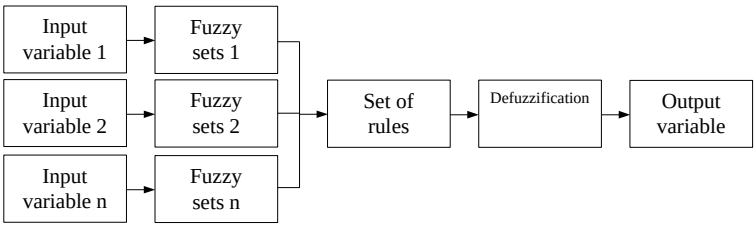
Fuzzy Logic is a form of many-valued logic in which the truth values of variables may be any real number between 0 and 1. By contrast, in Boolean logic, the truth values of variables may only be the integer values 0 or 1 – true or false[12].

Fuzzy logic has been employed to handle the concept of partial truth, where the truth value may range between completely true and completely false. Furthermore, when linguistic variables are used, these degrees may be managed by specific (membership) functions[12].

Human and animals often operate using fuzzy evaluations in many everyday situations. For example, when a person throws an object into a basket from distance, he/she does not compute exact values for the object weight, density, distance, direction, basket width etc to determine the force and angle to throw the object. Instead, the person instinctively applies quick fuzzy estimates, based upon previous experience, to determine what values of force, direction and angle to use to make the thrown[adapted from 12].

The MarMSX AI project introduced the concept of MSX 1 colors identification using Fuzzy Logic approach. The objective there was to introduce human experience to help on identifying colors.

The basic schema of a Fuzzy Machine process is the following:



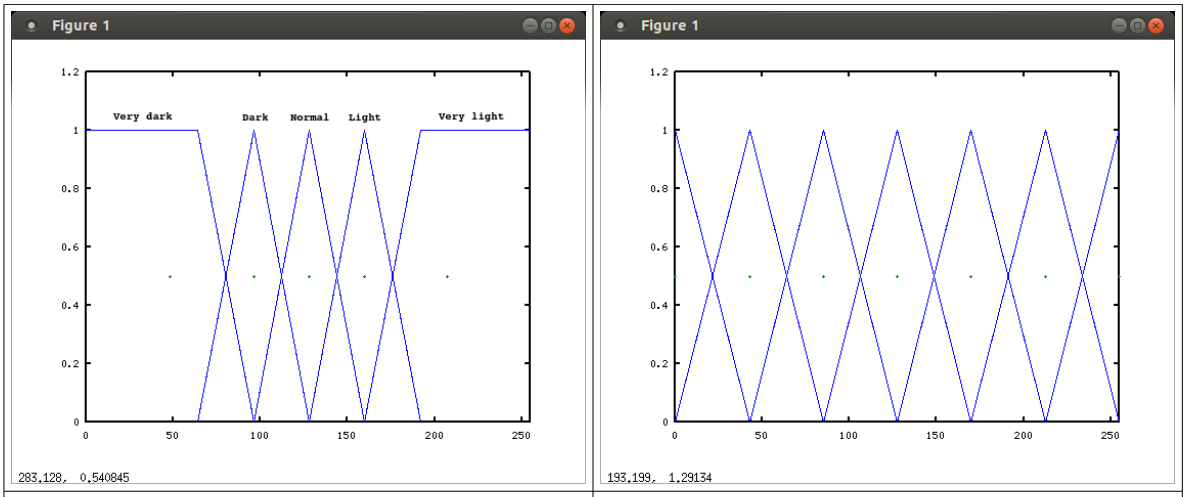
For color identification, the input variables are the three color components: red, greed and blue. Once the three variables are color intensities, the discussion universe ranges from 0 to 255.

Each input/output variable has its own fuzzy sets. The fuzzy sets are a set of membership functions (MF) where each one is responsible to represent a part of the discussion universe. The MFs will map a membership value between 0 and 1 for an input value.

Each fuzzy set is described as linguistic variable, using as adjectives and adverbs. For example, in a Fuzzy machine containing 5 fuzzy sets, we may describe each one as:

- very dark
- dark
- normal
- light
- very light

Figure 15 shows the fuzzy sets created for two different projects: AI and MSX Viewer. The first one has 5 fuzzy sets for each input variable using triangular and trapezoidal MFs, while the second has 7 fuzzy sets using only triangular MFs.



Fuzzy sets for AI project. | Fuzzy sets for MSX Viewer 5.

Figure 15. Fuzzy sets for different projects.

The x-axis values corresponding to *y=0* are:

- AI: { 64, 96, 128, 160, 192 }
- MSX Viewer 5: { 0.0, 42.5, 85.0, 127.5, 170.0, 212.5, 255.0 }

Notice that each MF has a ranging value representing an amount of color space intensity. For example, the AI fuzzy set “normal” ranges from 96 to 160 and it represents the medium color intensities.

In order to demonstrate how fuzzy sets evaluation works, lets take an example of an input variable having color intensity equal to 100. This value will activate two MFs: “dark” and “normal”. The MF “dark” will return a membership value equal to 0.88, while MF “normal” will return value equal to 0.12. In other words, an input color intensity of 100 is at the same time 88% dark and 12% normal. All the other MFs will return the value 0.

The previous example is depicted on figure 16.

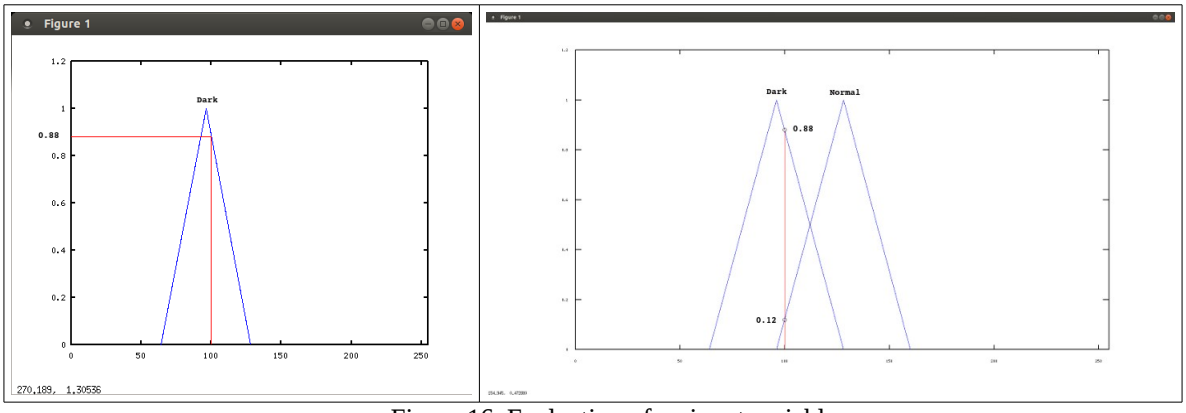


Figure 16. Evaluation of an input variable.

For the color identification case, each color component input RGB will be represented at output by a fuzzy set. The fuzzy set chosen is the one where the returned membership has the maximum value. For the input value 100, the “dark” set is the winner once it has the membership value of 0.88. The “dark” fuzzy set will then represent the value 100.

The next step is the inference using a set of rules. Whats are the rules?

In the previous step, each RGB color resulted in a fuzzy set representing it. Once each RGB value is an independent variable, many resulting fuzzy set combinations are possible. This combination is a Cartesian Product of each input fuzzy sets.

The rules are the formal description of decisions taken for each combination of fuzzy sets. They are created by a human specialist, bringing to the Fuzzy Machine the human capability of inference.

Suppose a Fuzzy Machine with three input variables and 2 fuzzy sets each one: dark and light. The rules are the Cartesian Product of the following tables: red, green and blue.

Red	Green	Blue
dark_red	dark_green	dark_blue
light_red	light_green	light_blue

Resulting rules:

Input 1	Input 2	Input 3	Inference
dark_red	dark_green	dark_blue	?
dark_red	dark_green	light_blue	?
dark_red	light_green	dark_blue	?
dark_red	light_green	light_blue	?
light_red	dark_green	dark_blue	?
light_red	dark_green	light_blue	?
light_red	light_green	dark_blue	?
light_red	light_green	light_blue	?

As mentioned before, the human specialist describes for each situation what he/she would do. In our case, each fuzzy set combination will result on a MSX 1 color index.

Each fuzzy set is represented by its centroid color intensity, generating a RGB color. The specialist then will compare the generated color with MSX 1 color set and choose the most alike color, establishing here a rule. Figure 17 shows an example of rules, using the AI project. The “C” line means the color generated by the combination of fuzzy sets centroids, while “H” line is the MSX 1 color chosen by a human specialist.

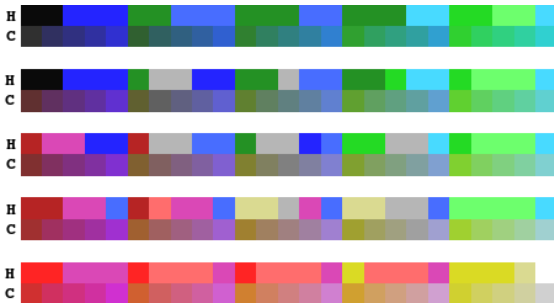
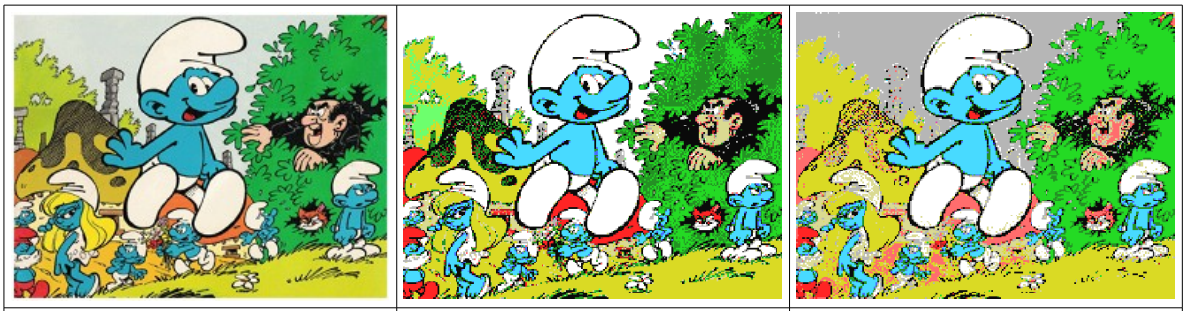


Figure 17. Creation of the rules.

For 5 fuzzy sets, there are 5^3 or 125 rules, while for 7 sets there are 7^3 or 343 rules.

Once our objective was achieved, where is to find the MSX 1 color palette, the last steps of Fuzzy Machine are not necessary.

For many years, MSX Viewer 5 used the fuzzy sets shown in figure 15, as an alternative to AI. Nevertheless, the AI project fuzzy sets presented better results, as seen on figure 18.



Original image.	Using AI project sets.	Using MSX Viewer 5 sets.

Figure 18. Examples of color identification using Fuzzy Logic.

From MSX Viewer version 5.0.1, AI rules replaced the original rules on MSX Viewer.

This approach for color identification using Fuzzy Logic is valid only for the same palette. In our case, we used the MSX 1 color palette. This is due to the fact of the creation of the rules is based on a specific palette. If palette is changed, the original color established for a rule decision is changed to another color and, consequently, another result comes up.

In that case, MSX Viewer 5 can only identify MSX 1 color palette using Fuzzy Logic.

6. Palette

MSX Viewer 5 has two different strategies to provide MSX palettes:

- First, create a Basic program which does everything.
- Second, create a binary program that loads palette into VRAM.

6.1. Palette Basic program

MSX Viewer creates automatically a Basic program, coding the palette adjustment and the image loading.

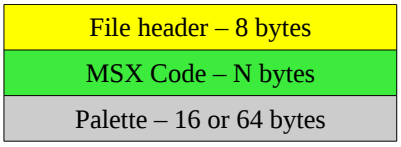
Below, we present an example of a Basic program created, called IMG07.BAS, which is responsible for loading an image called IMG.S07 and change MSX palette to the corresponding image palette.

```
10 SCREEN 7
20 COLOR=(0,6,6,6)
30 COLOR=(1,6,6,5)
40 COLOR=(2,6,5,4)
50 COLOR=(3,5,6,6)
60 COLOR=(4,6,4,3)
70 COLOR=(5,5,4,3)
80 COLOR=(6,4,3,3)
90 COLOR=(7,3,2,2)
100 COLOR=(8,1,1,1)
110 COLOR=(9,0,1,1)
120 COLOR=(10,0,0,0)
130 COLOR=(11,3,4,4)
140 COLOR=(12,2,5,6)
150 COLOR=(13,0,4,4)
160 COLOR=(14,0,3,3)
170 COLOR=(15,0,2,2)
180 BLOAD "IMG.S07",S
200 A$=INPUT$(1)
```

On MSX, the files IMG07.BAS and IMG07.S07 must be placed together to work.

6.2. Palette binary program

This second strategy loads palette into VRAM, by just running a binary program which changes palette. This program is created by MSX Viewer 5.



Palette program structure

There are two different palette programs: one for MSX 2 and other for MSX V9990.

MSX assembly code for 16 colors	MSX assembly code for 64 colors
0xFE, // Binary file identifier 0x00, 0xC0, // Program begins at C000 0x51, 0xC0, // Program ends at C051 (program + data) 0x00, 0xC0, // Program starts at C000 0x06, 0x10, // LD B,16 0x16, 0x00, // LD D,0 0x21, 0x22, 0xC0, // LD HL,&HC022 0x7E, // LP1: LD A,(HL) 0xCB, 0X27, // SLA A 0xCB, 0x27, // SLA A 0xCB, 0x27, // SLA A 0xCB, 0x27, // SLA A 0x4F, // LD C,A 0x23, // INC HL 0x5E, // LD E,(HL) 0x23, // INC HL 0x7E, // LD A,(HL) 0x81, // ADD A,C 0xDD, 0x21, 0x4D, 0x01, // LD IX,SETPLT 0xCD, 0x5F, 0x01, // CALL EXTROM 0x14, // INC D 0x23, // INC HL 0x10, 0xE6, // DJNZ LP1 0xC9, // RET 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0	0xFE, // Binary file identifier 0x00, 0xCE, // Program begins at CE00 (GBASIC is at C000!) 0xE1, 0xCE, // Program ends at C0E1 (C021 + 3*64) 0x00, 0xCE, // Program starts at CE00 0x21, 0x22, 0xCE, // LD HL,&HCE22 0x06, 0x40, // LD B,64 0x0E, 0x00, // LD C,0 0x3E, 0x0E, // LP1:LD A,&B00001110 0xD3, 0x64, // OUT (&H64),A 0x79, // LD A,C 0xCB, 0x27, // SLA A 0xCB, 0x27, // SLA A 0xD3, 0x63, // OUT (&H63),A 0x7E, // LD A,(HL) 0xD3, 0x61, // OUT (&H61),A 0x23, // INC HL 0x7E, // LD A,(HL) 0xD3, 0x61, // OUT (&H61),A 0x23, // INC HL 0x7E, // LD A,(HL) 0xD3, 0x61, // OUT (&H61),A 0x23, // INC HL 0x0C, // INC C 0x10, 0xE6, // DJNZ LP1 0xC9, // RET 0,0,0, 0,0,0

Naturally, the MSX Viewer will replace the 0's by the corresponding palette.

On MSX, run first the palette then load the image. For example:

```
10 SCREEN  7
20 BLOAD”IMG.PAL”,R
30 BLOAD”IMG.S07”,S
40 GOTO 40
```

Binary palette is preferred when displaying many images together like a slide show.

References

[1] Color Clash, Wikipedia, at http://en.wikipedia.org/wiki/Attribute_clash
[2] Error Diffusion, Wikipedia, at http://en.wikipedia.org/wiki/Error_diffusion
[3] Ordered Dithering, Wikipedia, at http://en.wikipedia.org/wiki/Ordered_dithering
[4] Mach Bands, Wikipedia, at http://en.wikipedia.org/wiki/Mach_bands
[5] K-Means Clustering, Wikipedia, at http://en.wikipedia.org/wiki/K-means_clustering
[6] MarMSX Development Page, at http://marmsx.msxall.com
[7] Marchi, J.; Tupimanbá, A.. CPU MSX Magazine #32, Bonus Editora, printed in Brazil, year 1993.
[8] Hans Otten. MCCM 72, translated by Alex Wulms, at http://www.msx-plaza.eu/home.php?page=mccm/mccm72/schermen_eng
[9] Retro 8 bits, at http://msx.retro8bits.com/msxsw.html
[10] OpenMSX Project, at http://openmsx.sourceforge.net/
[11] Velho, L.; Gomes, J. *Image Processing For Computer Graphics*, at http://w3.impa.br/~ipcg/
[12] Fuzzy Logic, Wikipedia, http://en.wikipedia.org/wiki/Fuzzy_logic
[13] Sunrise, at http://www.msx.ch/sunformsx
[14] Powergraph from Tecnobytes, at http://www.tecnobytes.com.br/p/v9990-powergraph.html
[15] G-Basic manual, at http://www.msx.ch/ftp/Products/Graphics9000/USER%20Manual/g-basic19990519.pdf
[16] SymbOS SGX format, at http://www.cpcwiki.eu/index.php/Format:SGX_(SymbOS_graphic_files)
[17] SymbOS, at http://www.symbos.de
[18] MSX Top Secret, Edson Moraes, at http://www.msxtop.msxall.com/Portugues/Projeto_msx_top_secret.htm

This appendix was written by:
Marcelo Teixeira Silveira – flamar98@hotmail.com
Rio de Janeiro, Brazil – August 2016

AKNOWLEGDEMENTS:

I would like to thank Julio Marchi and Andre Tupinamba for their excellent article on MSX2 and MSX2+ color systems in CPU MSX Magazine. Also, for Hans Otten for his magnificent articles.

Thanks Ricardo Oazem for testing MSX Viewer 5 on real V9990. Ricardo successfully used images generated by MSX Viewer 5 on his VSU project.

Thanks for all people from msx.org who found errors and suggested some stuff.