MSX Article

Screen 2 Wonders

Summary

In this article we discuss some wonders that can be done with the limited MSX screen 2, which has only ready-made 15 colors and a 1x8 pixel grouping that can only display 2 colors at time.

1- A little trick to increase the number of screen 2 colors

Different from MSX 2 screens 2-7, the MSX 1 screen 2 has only 15 fixed colors. But, according to a little trick created by Daniel Vik [1], author of BlueMSX emulator and some incredible demos like Utopia and MSX Unleashed, we can increase MSX colors up to 120 colors (not 105 – see next section). How is it possible?

If we alternate two pixels with different colors very quickly on screen, we get an illusion as if their color were mixed. For example, if we alternate the color green (MSX color code 3) with cyan (MSX color code 7), we will see a new green color with the average colors of these two pixels. See figure 2.1.

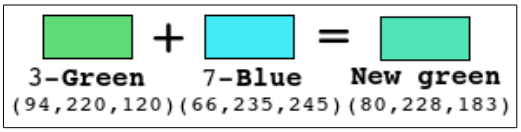


Figure 2.1. - Color combination.

1.1. How many colors we can generate by mixing the 15 MSX 1 native colors?

According to the Mathematics concept of combination, we have:

$$C(n,p)=\frac{n!}{(n-p)!\,p!}$$

$$C(15,2)=\frac{15!}{(15-2)!\,2!}=\frac{15\times14\times\cancel{13!}}{\cancel{13!}\,2!}=\frac{15\times14}{2}=15\times7=105$$

Here, we are talking about different colors combination. Nevertheless, we can also combine the same MSX native color, which results on itself. According to that, we have 105 + 15 = 120 colors.

Figure 2.2 shows all possible 120 colors generated by mixing MSX 1 native colors. The most outside diagonal presents the MSX 1 native colors.

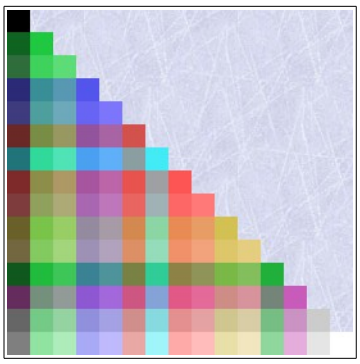


Figure 2.2. - The 120 achieved on screen 2.

1.2. How to generate images for the screen 2 with 120 colors format?

As we previously seen, these new colors are generated by mixing two MSX 1 colors. According to that, we must have two images that will mix up their colors in order to generate the final image with up to 120 colors. In addiction, this trick allow us to get another advantage: 4 colors per 1x8 pixels group. Let's see how.

As we already know, is it possible to display only two pixels per 1x8 block in screen 2 – front color and background colors. Once we have two images that share the same space and their colors can be mixed, we will have four different colors combinations for each block as follows:

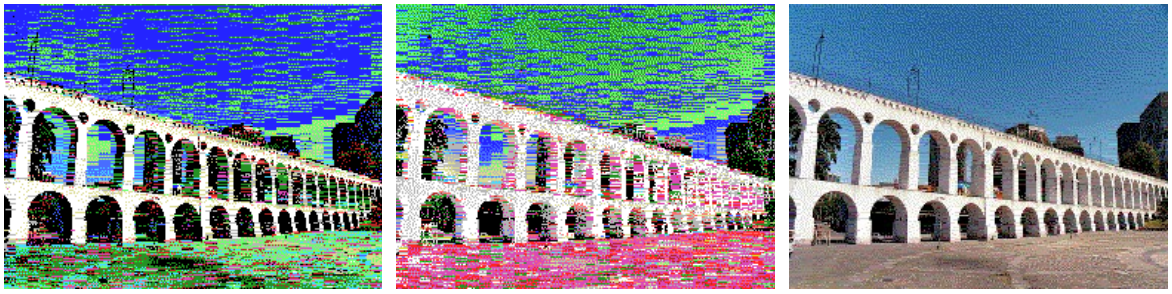
- *Image 1 front color with image 2 front color*
- *Image 1 front color with image 2 background color*
- *Image 1 background color with image 2 front color*
- *Image 1 background color with image 2 background color*

Figure 2.3. Color mixing.

By mixing only two MSX native colors, we will face a little problem: we can only generate three instead of four colors. For example, if we try to mix up the colors 1 and 2 in both images for a given 1x8 block, we will generate the colors 1-1, 2-2, 1-2 and 2-1. But colors 1-2 and 2-1 are the same colors.

The cost function [1] (seen on the next chapter) is responsible for quantize image and also finding out the best 4 colors combination for each 1x8 pixels group. This approach give better results than if we quantize image first and then found out the best 4 colors. In addition, it makes the task of finding out the best colors for each 1x8 block quite simple.

Figure 2.4 presents an example of image fusion by alternating two MSX 1 native colors images.



(a) (b) (c)

Figure 2.4. image color fusion: (a) + (b) = (c).

1.3. How to alternate image exhibition

Screen 2 has two tables that controls 1x8 pixels group: the pattern table, that controls the pixels pattern (front or background color), and the color table, that holds the color codes for the two colors.

The RAM to VRAM data transfer is quite slow for large images to achieve an illusory effect that we are searching for. In order to achieve a faster image switching, we will use another table from screen 2: the name table.

The name table relates screen physical 8x8 pixels blocks with both pattern and color tables 8x8 pixel blocks, formed by the vertically stacking of 1x8 pixels groups. This table divides vertically the screen into three parts, numbered from 0 to 255 each. The whole screen data transfer takes 12288 bytes, while name table only takes 768, which means that it takes 16 times less data transfer.

Let's see the next example. The program will rapidly switch two block of 8x8 pixels (blocks 0 and 2 from pattern/color tables) through the name table, at the physical location corresponding to the physical block 4, as seen on figure 2.5.

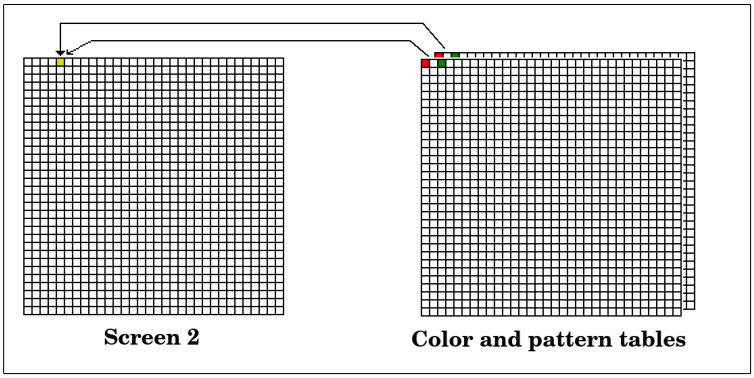


Figure 2.5. - An example of screen 2 color fusion.

Code in Assembly:

```
C000                10          ORG &HC000
C000 21 04 18       20 BEGIN:   LD HL,&H1804 ; VRAM block 004
C003 AF             30          XOR A        ; Pat+color to write: 0
C004 CD 4D 00       40          CALL &H4D    ; Send data to VRAM
C007 3E 02          50          LD A,2       ; Pat+color to write: 2
C009 CD 4D 00       60          CALL &H4D    ; Send data to VRAM
C00C CD 9C 00       70          CALL &H9C    ; Like A$=inkey$
C00F 28 EF          80          JR Z,BEGIN   ; If key not pressed, return
C011 C9             90          RET          ; Returns to Basic
```

Code in Basic, which incorporates the code in Assembly above:

```
10 COLOR 15,1:SCREEN 2
20 OPEN"grp:" AS #1
30 LINE(0,0)-(7,7),6,BF
40 LINE(16,0)-(23,7),2,BF
50 PRESET(9,0):PRINT#1,"+"
60 PRESET(25,0):PRINT#1,"="
70 DEFUSR=&HC000
80 E=&HC000
90 READ A$:IF A$="M" THEN 130
100 POKE E,VAL("&H"+A$)
110 E=E+1
120 GOTO 90
130 X=USR(0):END
200 DATA 21,04,18,AF,CD,4D,00,3E
210 DATA 02,CD,4D,00,CD,9C,00,28
220 DATA EF,C9,M
```

In order to make use of name table, the images must be already loaded in the VRAM. Images? Yes, both images share the pattern/color tables, and the trick is to alternate them using the name table.

Once both images share the same space on the VRAM, we can only use the half screen width for each image. One image takes the first half of pattern/color tables, while the other takes the rest. But, we must also reserve a block to hold the black color used on the screen's regions not used by the resulting image.

So, we reserved the block 0 to the black color and reduced image width from 128 to 120 to preserve image size balance. Each image has 120 x 192 pixels and takes 120 blocks of 8x8 pixels, as shown in figure 2.6.

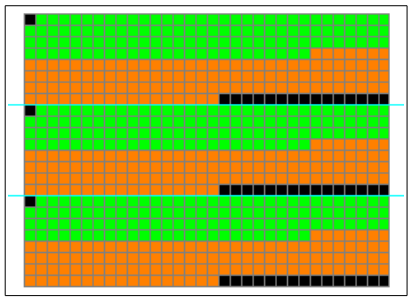
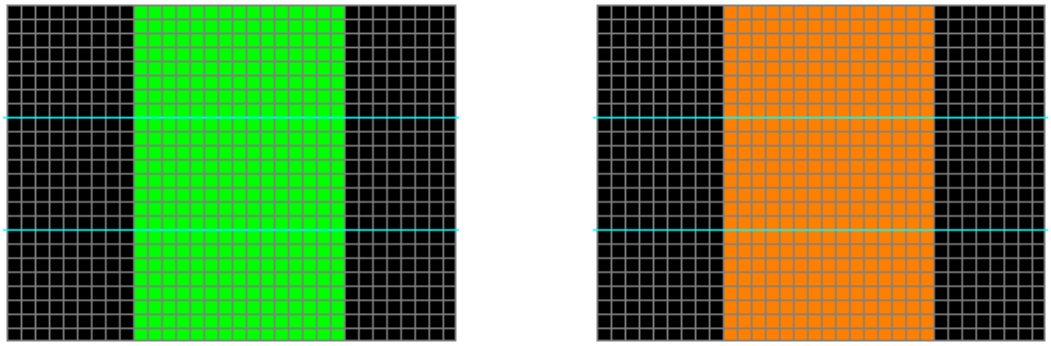


Figure 2.6. - Image layout on screen.

The green color represents the image A data, whereas the orange color represents the image B data.

Once loaded the images data, we will use the name table to set images on the right place and quickly switch them, as seen on figure 2.7.



(a) image A (a) image B

Figure 2.7. Image positioning and switching.

A simple way to achieve this is to create a map in the RAM with the block positions for both images A and B in the pattern/color tables. First we copy the image A map data to the VRAM name table, then we do the same for image B map. Repeat both operations until a key is pressed, for example. Do not load both maps at once or the image switch will not be seen.

This article is followed by an Assemlby source code for image switching, the map and some 120 colors image examples. To run an image in Basic, type:

```
BLOAD "IMAGE.120",R
```

The program changes automatically to screen 2 mode and shows the image. If you press a key, the program stops and returns to the screen 0.

2- Cost Function – the best 2/4 colors for an 1x8 pixels group

Daniel Vik [1] has used a cost function to convert 24-bit images to the 15 MSX 1 default colors. In this function, the objective is to calculate an error from each pixel color within an 1x8 pixel group to all possible MSX 1 color pairs. The pair who returns the shortest error from the 1x8 group being evaluated is then selected to represent the group.

This technique consists in creating a table with all two colors combinations from MSX colors without repetition, an then compare each group with each pair.

Index	Color 1	Color 2
1	1	2
2	1	3
3	1	4
4	1	5
...	...	...
104	13	14
105	14	15

Table 2.1 – Color combinations

The Squared Error is used to calculate the error for each group in relation to each color pair.

$$SE(r,n)=(R_r-R_n)^2+(G_r-G_n)^2+(B_r-B_n)^2$$

Where:

- r – the color to be tested from the original image.
- n – color 1 or 2 from table.
- R – red color component.
- G – green color component.
- B – blue color component.

After, we apply the Minor Squared Error:

$$MSE(r,n)=min[SE(r,1),SE(r,2)]$$

According to that, we assume only the error from one color from the pair which results in the least error. This prevents the testing pixel from the interference of the other color, mostly when such color is very different from the testing pixel.

At the end, we add the MSE from all 8 pixels to a given color pair and we get the error for that pair. The pair who returns the least error is selected to represent the group.

Let's see how it works.

Suppose an 1x8 group which the 1st pixel has the same color of MSX 1 index 7. When we compare this pixel with the line 4 from the table 2.1, we must calculate the SE from color 7 to color 1 and from the color 7 to the color 5.

```
Error 7-5: 12.100
Error 7-1: 118.315
```

After all, we find out that the error from 7 to 5 is less than the error from th 7 to 1. This means that the cyan color is closer to the blue than the black. So, we assume the error from 7 to 5 as the error for the 1st point. Notice that the error assumed 12 is quite smaller than the discarded value 118. In that case, we have eliminated a great interference on the overall error.

Now, we repeat this operation for the remaining 7 points, and still, at the line four. After that, we get the overall error for the evaluating color par.

$$MSE_{total} = \sum_{i=1}^{n} MSE_i$$

Once finished a line, we start another line until we reach the end of the table with color combinations. The pair who resulted on the least error is then selected to represent the pair.

Until here we only found out the best color pair to represent an 1x8 group. But, the group still having the original colors. For that, we may use the Euclidean Distance to calculate each pixel to each color of the selected pair. We assume the color pair who returns the shortest distance.

$$d = \sqrt{(R_r - R_n)^2 + (G_r - G_n)^2 + (B_r - B_n)^2}$$

2.1. Adding Error Diffusion to the process

On Jannone, Robsy and Ragozini's [2,3] work it was added the Error Diffusion [4] to the image conversion, resulting in impressive images to the limited MSX 1 screen 2.

This technique spreads an error to the original image's right-down pixels neighbors, minimizing the Mach Band Effects [4]. The error is calculated as the difference between the original color and the new calculated color.

According to the cost function characteristics, we will only get the new color from a pixel within an 1x8 pixel after processing all the pixels and finding out the best color pair. So, we can only apply error diffusion after all the process.

Thus, there is a little problem when we spread an error to a neighbor within the group: after changing the neighbor's color, we are changing the color of a group's member and, consequently, modifying the previous calculation we made to the whole group.

In order to solve that issue, after spreading an error to a pixel withing the evaluated group, we have to re-calculate the best color for that 1x8 group. Fortunately, the pixels located under the group are out of the group and the error spread to them do not affect it.

The pseudo-algorithm is presented next.

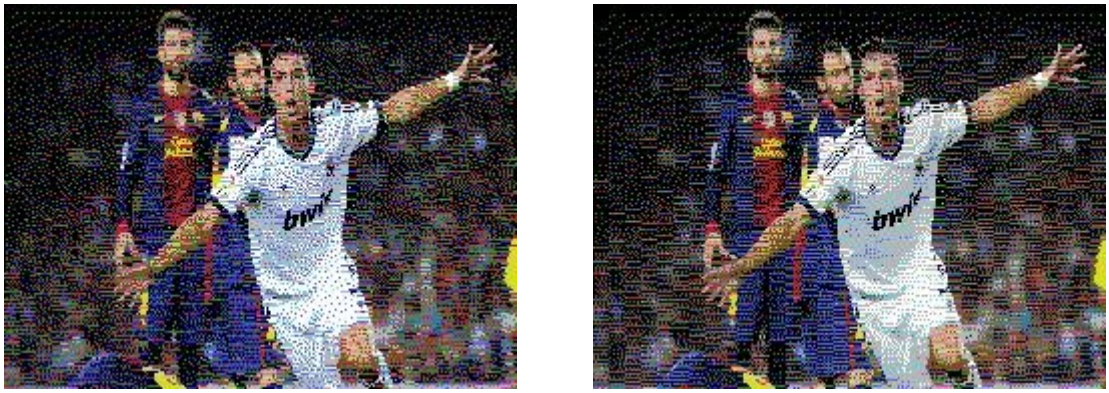
```
// For the group
for i ← 0 to 7 do
      [c1 c2] ← cost_function(group)
      quant_error ← pixel(x+i, y) – best_color(pixel(x+i, y), c1, c2)
      pixel(x+i+1, y) ← pixel(x+i+1, y) + quant_error * 7/16
end_for

// For the rest
for i ← 0 to 7 do
      quant_error ← pixel(x+i, y) – best_color(pixel(x+i, y), c1, c2)
      pixel(x+i, y) ← best_color(pixel(x+i, y), c1, c2)

      pixel(x+i-1, y) ← pixel(x+i-1, y) + quant_error * 3/16
      pixel(x+i, y) ← pixel(x+i, y) + quant_error * 5/16
      pixel(x+i+1, y) ← pixel(x+i+1, y) + quant_error * 1/16
end_for
```

Repeating eight times the cost function calculation for each 1x8 group compromises significantly the algorithm performance. If we ignore the re-calculation for the pixels within the group, we lie on the classic Error Diffusion algorithm and the result is quite similar.

Figure 3.1 compares the results from the normal algorithm with the optimized algorithm. We can notice on figure 3.1 (b) that some lines appears in comparison with figure 3.1 (a).



(a) (b)

Figure 3.1. Comparison between the original algorithm (a) and the optimized one (b).

Figure 3.2 shows more results obtained from the original algorithm with Error Diffusion applied on 24-bit images in order to convert them to MSX 1 screen 2 format.

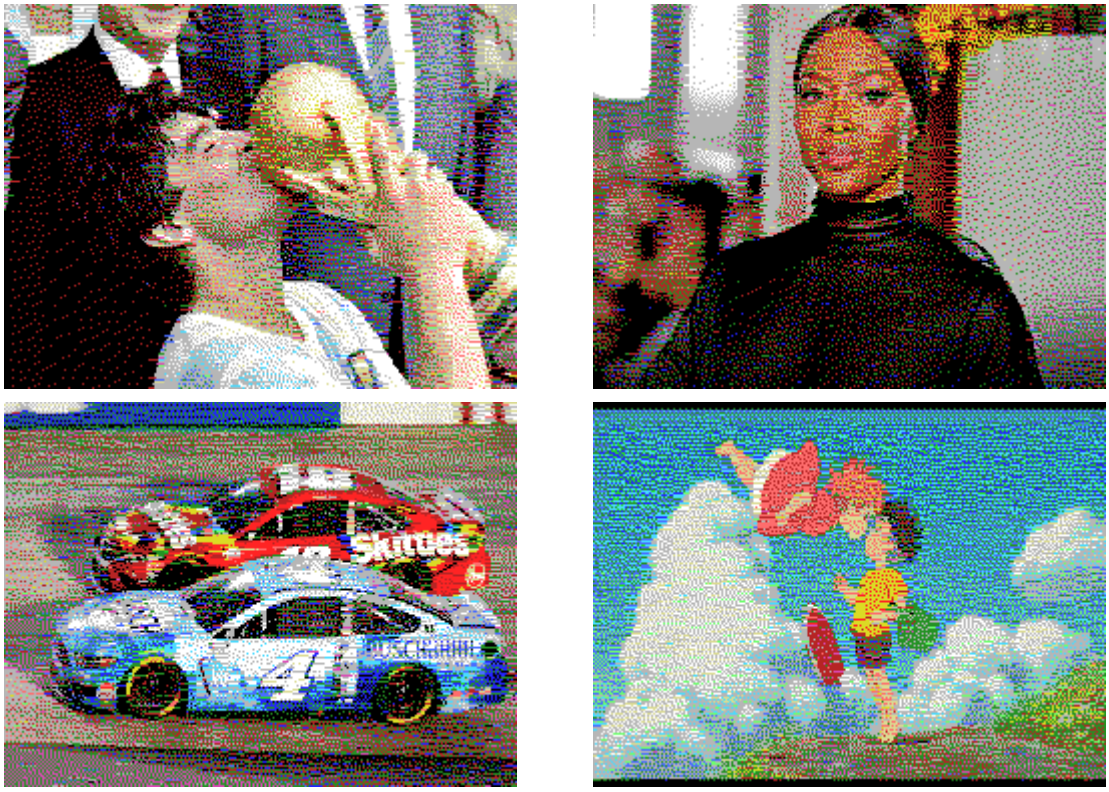


Figure 3.2. Results from the 24-bit images conversion to MSX 1 screen 2.

Credits and References

This article was written by Marcelo Silveira in December 2020.

Date: 12 / 2020
E-mail: flamar98@hotmail.com
Homepage: http://marmsx.msxall.com

References:

[1] – BMPto105, Daniel Vik, 2006. http://vik.cc/dvik-joyrex/download/105Colors.ppt
[2] – MSX Screen Convertor, Rafael Janone. http://msx.jannone.org/conv/
[3] – TMSOPT v.0.1, Eduardo Robsy, Arturo Ragozini and Rafael Janone, 2007.
[4] – Artigo Error Diffusion – Marcelo Silveira. http://marmsx.msxall.com/artigos
[5] – MSX Viewer 5 – Marcelo Silveira. http://marmsx.msxall.com/msxvw/msxvw5