

MSX Article

MARMSX

*Rotation and
Interpolation*

Summary

This article describes how to rotate 2D objects using any angle, as well as describes two pixel interpolation methods – nearest neighbor and bilinear.

1- Introduction

Image rotation is simply the application of a transform function to an image, in order to obtain new coordinates for each image pixel. In other words, after calculating a transform function to a pixel $P(X,Y)$, we obtain $P(X',Y')$. Once the color is the pixel identity, we simply copy the pixel color from $P(X,Y)$ to $P(X',Y')$.

The rotation of an object using angles multiple of 90 is quite easy to do, once it is only necessary to apply simple matrix transformations like transpose matrix. The next example rotates an image 90 degrees counter-clockwise.

```
10 SCREEN 5
20 COPY"BRASIL.IC5" TO (0,0),0
30 DX=0 : DY=80
40 FOR Y=0 TO 68 }
50 FOR X=0 TO 99 }
60 C=POINT(X,Y)
70 PSET(Y+DX,99-X+DY),C
80 NEXT X,Y
90 GOTO 90
```

Sweeps the image.

Get the color of pixel to be rotated.

Applies the rotation and copies the color.

The program above applies the transpose matrix operation on the original image and also applies a mirroring effect on axis Y , in order to avoid a reflected resulting image. Furthermore, the program adds a double shift DX and DY on the resulting image to avoid image overlapping.

Image rotation in any angle claims for a more complete coordinate system transformation.

2- Image rotation using any angle

The equation (1) describes a complete rotation system, which it is possible to use any angle around the coordinate system origin (coordinate 0,0).

$$\begin{aligned}x' &= x \cdot \cos(\theta) - y \cdot \sin(\theta) \\y' &= x \cdot \sin(\theta) + y \cdot \cos(\theta)\end{aligned}\tag{1}$$

Figure 1 shows the coordinate systems used to rotate the image: the screen coordinate system and the real world coordinate system. The first one is used to manipulate the image, while the second is the system used by the rotation system.

It is desirable that the coordinate system lies on the center of the image and not on the boundaries, as shown in figure 1. We must take in account that screen coordinate system has the Y axis reflected if compared to the real world coordinate system.

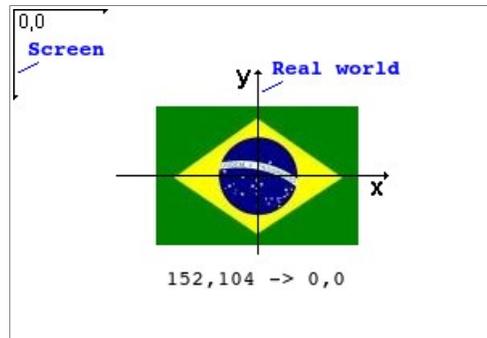


Figure 1 – Image, screen and the real world coordinate systems.

The first step is convert a pixel coordinate (in screen coordinate system) to real world coordinate system in order to apply the rotation. The equation (2) does this and also place this coordinate system on the center of the image.

$$\begin{aligned} x_r &= x_i - cx_i \\ y_r &= cy_i - y_i \end{aligned} \quad (2)$$

Where:

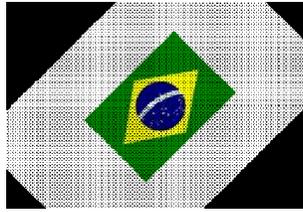
- x_r e y_r – are the real world (rotation system) coordinates.
- x_i e y_i – are the screen coordinates.
- cx_i e cy_i – are the image center coordinates on the screen coordinate system.

After applying the rotation, the calculated values x' and y' must be converted back to the screen coordinate system. Equation (3) does this operation.

$$\begin{aligned} x_i &= x'_r + cx_i \\ y_i &= cy_i - y'_r \end{aligned} \quad (3)$$

Although it is intuitive to rotate pixels from the source image, the correct way is to rotate pixels from the destiny image. This is due to the fact that if we rotate from the original image, some pixels in the destiny image may not be filled, once no correspondence was found. In the other hand, from the destiny image, all pixels are granted to be filled.

Image 2a shows the rotation from the source image. Notice that many pixels are empty (holes), once no correspondence was found. Image 2b shows the rotation from the destiny image. Now, the image is complete.



a) from the source image



b) from the destiny image

Figure 2. Rotation applied on the image.

The changes we must do to perform rotation from destiny image are:

1. Assume the angle theta (θ) in equation (1) as negative.
2. Take the color of the calculated pixel (source image).
3. Fill the read pixel (destiny image) using such color.

Once the MSX takes almost one second to calculate the sin and cos sin of an angle, we may optimize it by calculate these values before sweeping the image.

After calculating the rotation pixel coordinates in the source image, the result is composed by floating point numbers. This means that the calculated coordinate do not lie exactly on a pixel, as seen in figure 3.

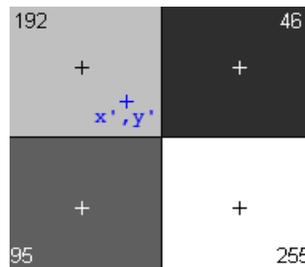


Figure 3. The color determination issue, from a coordinate using floating point values.

Once is the pixel color we are searching for, when the coordinate is not located exactly on a pixel but among neighbor pixels, what we have to do?

The next section will discuss this issue, presenting some interpolation methods which interpret this question in many ways.

3- Interpolation methods

3.1- Nearest Neighbor

The Nearest Neighbor interpolation assumes the color of the nearest pixel from that coordinate, ignoring the other pixels. In order to achieve that, we round the coordinates values. For example, the coordinate (98,5; 34,2) would be:

$$\begin{aligned} \text{round}(98.5) &= 99 \\ \text{round}(34.2) &= 34 \end{aligned}$$

After that, the color we assume is that from the pixel who coordinates are (99,34). This interpolation is quite simple, but present some image artifacts like “jigsaw” effect.

3.2- Bilinear

Bilinear Interpolation calculates the weighted average color from the four neighbor pixels around the coordinates. Equation (4) does Bilinear Interpolation.

$$f(x,y) = f(0,0) \cdot (1-x) \cdot (1-y) + f(1,0) \cdot x \cdot (1-y) + f(0,1) \cdot (1-x) \cdot y + f(1,1) \cdot x \cdot y \quad (4)$$

Where $f(x,y)$ is the color of each pixel, according to the following configuration:

$f(0,0)$	$f(1,0)$
$f(0,1)$	$f(1,1)$

For each neighbor pixel in relation to the coordinate (x',y') , it is considered the inverse distance, once the shorter distance from the pixel to the calculated coordinates, the greater contribution from that pixel to the final color.

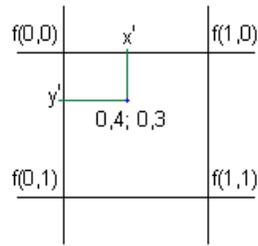
Suppose a rotation resulted coordinates are (150,4; 200,3). The four neighbor pixels are the following:

- $P_1(150,200)$
- $P_2(151,200)$
- $P_3(150,201)$
- $P_4(151,201)$

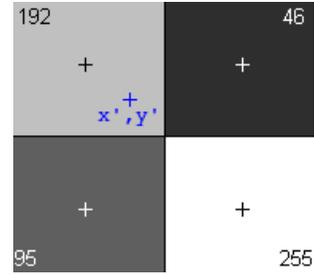
Assuming P_1 as origin at (0,0), we have:

- $P_1(0, 0)$
- $P_2(1, 0)$
- $P_3(0, 1)$
- $P_4(1, 1)$
- $P(0,4; 0,3)$

Figure 4a illustrates this coordinate system normalized to the Bilinear Interpolation calculation, while figure 4b shows the pixels gray levels.



a) normalized coordinate system



b) pixels intensities

Figure 4. Bilinear Interpolation.

Using equation (4), we have:

$$f(x, y) = 192 \times (1 - 0,4) \times (1 - 0,3) + 46 \times 0,4 \times (1 - 0,3) + 95 \times (1 - 0,4) \times 0,3 + 255 \times 0,4 \times 0,3$$

$$f(x, y) = 141.2200$$

$$f(x, y) = \text{round}(141.2200)$$

$$f(x, y) = 141$$

Once the pixel has an integer value, we round the result.

3.3- Comparing the results

Figure 5 presents two images, comparing the Nearest Neighbor and Bilinear interpolation.



a) Nearest Neighbor interpolation



b) Bilinear interpolation

Figure 5. Comparing interpolation methods.

We observe that Bilinear Interpolation results are better than Nearest Neighbor interpolation. Thus, Bilinear Interpolation results on greater computational cost. This is a critical problem to slow computers like MSX.

Another issue to use Bilinear Interpolation on MSX is due to the weighted colors. Only screen 8 is able to mix colors, once the other screens use indexed colors.

4- A program in Basic to rotate images

The following program rotates the Brazilian flag in 45 degrees counter-clockwise using Nearest Neighbor interpolation. For doing that, some procedures were made:

- The image was loaded in the page 1 from the screen 5, centered at 128,105.
- The program rotates the image from page 1 to page 0, also centered at 128,105.
- The drawing area takes in account the flags' diagonal, which is the image greatest possible length. So, this area is a square based on the flags' diagonal size.
- We use the floor instead of the round on floating numbers resulted from (x',y') calculation. Once MSX Basic do not support round function, the use of that would result on more operations.

```
10 SCREEN 5
20 SETPAGE 0,1:CLS
30 COPY"BRASIL.IC5" TO (77,71),1
40 SET PAGE 0,0
50 CX=128 : CY=105
60 PI=3.14159 : AN=-PI/4
70 ST=SIN(AN) : CT=COS(AN)
80 FOR Y=34 TO 175
90 FOR X=57 TO 198
100 XR = X-CX
110 YR = CY-Y
120 RX = FIX(XR*CT - YR*ST)
130 RY = FIX(XR*ST + YR*CT)
140 XI = RX+CX
150 YI = CY-RY
160 SETPAGE 0,1
170 C=POINT(XI,YI)
180 IF C<0 THEN C=0
190 SETPAGE 0,0
200 PSET(X,Y),C
210 NEXT X,Y
220 GOTO 220
```

Sin and cos sin
computed once.

Sweeps the image.

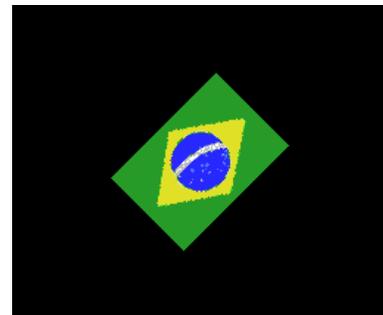
Converts from destiny image
coordinates to real world.

Rotates the pixel.

Converts the calculated coordinates to
image coordinate system.

Get pixel color from source image.

Copies the color to the destiny image.



This program takes around 30 minutes to complete the rotation in a MSX.

Add the next line to follow the process:

```
75 LINE(57,34)-(192,175),13,BF
```

Obs: the programs used in this article, as well as the pictures, are available for download at MarMSX Development page articles section – <http://marmsx.msxall.com>

5- Matlab and GNU-Octave fonts for tests

Nearest Neighbor	Bilinear
<pre> % Rotation % Marcelo Teixeira Silveira br=imread('flag.bmp'); [a b c]=size(br); br2 = zeros(a,b,c); br2=uint8(br2); theta = -0.78539816339744830961566084581988; % 45 degrees centerX=153; centerY=105; for x=1 : b for y=1 : a % From screen system to center system % Center coordinate system cx = x-centerX; cy = centerY-y; % Rotate nx = cx*cos(theta)-cy*sin(theta); ny = cx*sin(theta)+cy*cos(theta); % Nearest neighbor nx=ceil(nx); ny=ceil(ny); % Return old system cx=nx+centerX; cy=centerY-ny; if (cx>0) if (cy>0) if (cx<=b) if (cy<=a) br2(y,x,:) = br(cy,cx,:); end end end end end end; figure(1); subplot(1,2,1); imshow(br); subplot(1,2,2); imshow(br2); </pre>	<pre> % Rotation % Marcelo Teixeira Silveira br=imread('flag.bmp'); [a b c]=size(br); br2 = zeros(a,b,c); br2=uint8(br2); theta = -0.78539816339744830961566084581988; % 45 degrees centerX=153; centerY=105; for x=1 : b for y=1 : a % From screen system to center system % Center coordinate system cx = x-centerX; cy = centerY-y; % Rotate nx = cx*cos(theta)-cy*sin(theta); ny = cx*sin(theta)+cy*cos(theta); % Return old system cx=nx+centerX; cy=centerY-ny; % Pick up neighbors coordinates origin and float value of % coordinate Pox = floor(cx); Poy = floor(cy); xn = cx-floor(cx); yn = cy-floor(cy); if (cx>1) if (cy>1) if (cx<b) if (cy<a) % Bilinear br2(y,x,1) = ceil(br(Poy,Pox,1))*(1-xn)*(1-yn) +br(Poy,Pox+1,1)*xn*(1-yn)+br(Poy+1,Pox,1)*(1- xn)*yn+br(Poy+1,Pox+1,1)*xn*yn); br2(y,x,2) = ceil(br(Poy,Pox,2))*(1-xn)*(1-yn) +br(Poy,Pox+1,2)*xn*(1-yn)+br(Poy+1,Pox,2)*(1- xn)*yn+br(Poy+1,Pox+1,2)*xn*yn); br2(y,x,3) = ceil(br(Poy,Pox,3))*(1-xn)*(1-yn) +br(Poy,Pox+1,3)*xn*(1-yn)+br(Poy+1,Pox,3)*(1- xn)*yn+br(Poy+1,Pox+1,3)*xn*yn); end end end end end end; figure(1); subplot(1,2,1); imshow(br); subplot(1,2,2); imshow(br2); </pre>

6- Credits and references

This article was written originally in Portuguese and translated into English by Marcelo Silveira, Systems and Computer Engineer graduated at UERJ.

Written in: April 2007.

Reviewed: July 2017 and May 2018.

Site: <http://marmsx.msxall.com>

E-mail: flamar98@hotmail.com

References:

Wikipedia: <http://en.wikipedia.org>

- Article on Bilinear Interpolation

- Article on rotation.