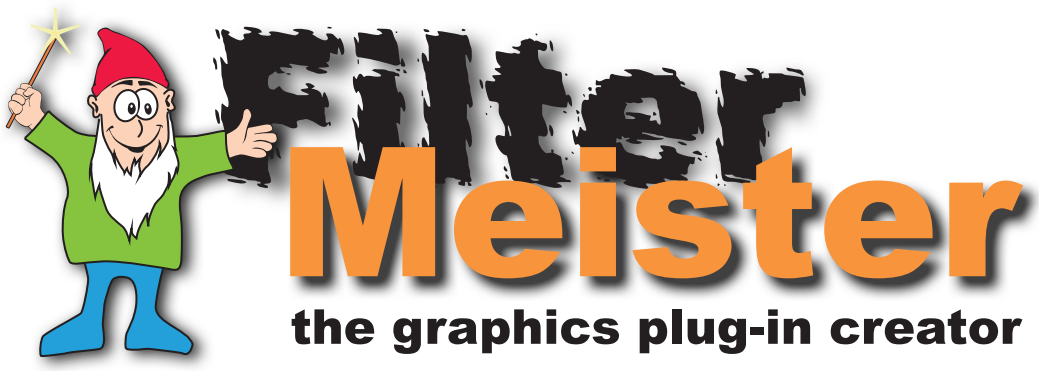



# Digital Image Processing with FilterMeister

*by Werner D. Streidt*

The creation for custom digital image filtering was based on the programming of propietary programs in various programming languages as found in the current literature about digital image processing. With the release of FilterMeister, a filter plug-in compiler, custom filter algorithms can easily be tested and applied within any existing graphics program which supports the Adobe Photoshop plug-in API. The programming content is wholly based on the filter algorithm; the programmer does not need to implement image file and screen/printer output routines. The Filter Factory Plus programming language is a superset of the original Adobe Filter Factory plug-in compiler and a subset of the C programming language. The major advantage of FilterMeister filters are: support of various image modes, floating-point arithmetic instead of purely integer arithmetic, up to 16 KB of source code memory and support of different user controls (pushbuttons, checkboxes, scrollbars, etc.). In addition to the programming possibilities, the programmer also has control over the filter dialog window's design. This thesis will examine FilterMeister's ability and limitations to generate plug-ins for digital image processing tasks.

*I hereby declare that this thesis represents my own work, except where due acknowledgement is made and that it has not been previously included in a thesis, dissertation or report submitted to the Fachhochschule Stuttgart – Hochschule für Druck und Medien or to any other institution for a degree, diploma or other qualification.*

Werner D. Streidt, October 1999

Correctors:
Dipl.-Ing. (FH) Ullrich Reiser
Dr. Thomas Hoffmann-Wahlbeck

# Table of Contents

## 5. Digital Operators



## 6. Plug-In artistry



## 7. Conclusion



## 8. Appendix



*This document was created with Quark XPress for Windows 3.3.2. The images were scanned into, created and/or retouched with Adobe Photoshop 5.0.2 and FilterMeister Beta 0.4.14. Vector graphics were created with Adobe Illustrator 7.0.1. The fonts used in this document were Adobe Minion Condensed, ITC Officina Sans, Letter Gothic 12 BT and CombiNumerals. The PDF files were created with Adobe Acrobat 4.0 Distiller.*

# 1. Introduction

This document contains several tutorials on creation of filters and plug-ins for digital image processing tasks with the help of the Plug-In compiler FilterMeister for Windows operating systems. Note that this document does not contain installation procedures or an introduction to the FilterMeister language programming – please consult the *Getting Started* and *User Guide* manuals(which are also part of this thesis) included in the CD-ROM. Also note that this document contains altered excerpts from the FilterMeister manuals mentioned above.

Chapter 2 will describe some basic knowledge on the functions and color spaces of a typical image editing program. Often needed preprocessing data like the mean value, deviation and histogram of an image are presented as FilterMeister programs.

Chapter 3 explains the necessity of image compression respectively reduction techniques. One example per technique is presented as a FilterMeister program.

Chapter 4 deals with image modifications such as translating, rotating and scaling an image, respectively typical color modifications such as desaturation, thresholding, inverting an image or the correction of brightness and contrast in an image. In the end, an easy example of image encryption and different calculation modes, as often needed for blending two images into one, are explained.

Chapter 5 describes the realization of digital operators with FilterMeister for preprocessing tasks such as blurring, sharpening, edge enhancement and morphological operations. The last section describes a special algorithm for object counting in bilevel images, the Hoshen-Kopelman algorithm.

Chapter 6 deals with the artistic possibilities of FilterMeister plug-ins.

Chapter 7 concludes the capabilites and limitations of the FilterMeister plug-in compiler as of October 1999.

The appendix lists the contents of the CD-ROM in this package and the references used for this thesis.

# 2. Image basics

This chapter describes the basic image characteristics images can have. Typical preprocessing functions are presented in form of FilterMeister programs.

## 2.1 Image metrics

Images are built up of pixels that contain color information and are aligned with the cartesian coordinate system. The zero point is found at the top-left corner of the image (in PostScript, for example, the zero point is found at the bottom-left corner of the page). The image's width is represented by the variable `X`, the image's height with the variable `Y`. Figure 2.1 (left side) shows the coordinates of an image with the width and height of 11 × 8 pixels.

FilterMeister also has built-in functions and variables that have access to the image in the polar coordinate system. The zero point in polar coordinates is found at the middle of the image. The two coordinate axes are the angle (or direction) and magnitude (or distance from the image's center) and are represented by the variables `d` and `m`, respecitvely. Figure 2.1 (right side) shows the computed polar coordinates of the same image.

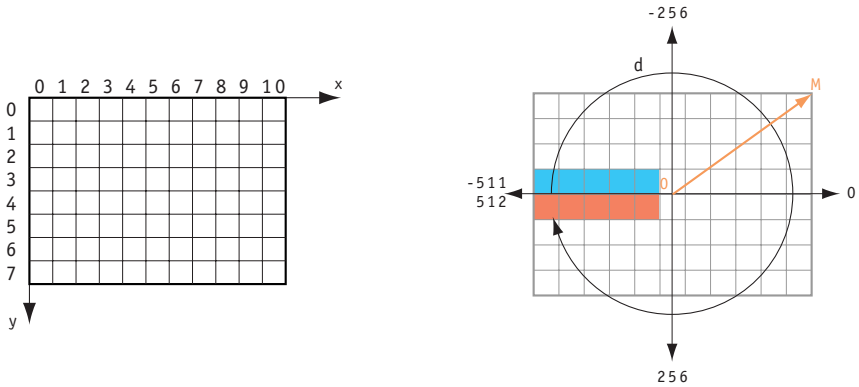



*Fig. 2.1: Visualization of the x, y, d, m variables and X, Y, M pseudo-constants*

The following table describes the variables and their respective ranges:

| Variable | Description | Value range |
|---|---|---|
| x | specifies the x-coordinate (horizontal position) of the current pixel | 0 – X-1 |
| y | specifies the y-coordinate (vertical position) of the current pixel | 0 – Y-1 |
| d | specifies the angle (or direction) of the current pixel around the center of the image | -512 – 511 |
| m | specifies the distance (or magnitude) of the current pixel from the center of the image | 0 – M |

*Note: X, Y and M are image constants that represent the image's width, height and half the image's diagonal measurement, respectively.*

## 2.1.1 Color Models *(Image Modes)*

When working with pixel-based images, one can work with either black-and-white (bitmap), grayscale and colored images. You have to understand that the pixel information between these image modes is different. This also plays a part in the image file's size and memory size needed.

### Bitmap images

Bitmaps are black-and-white images, whose pixel information can only be black or white and nothing in between. One bit can have the information 0 (Black) or 1 (White) and nothing else, and that is enough to describe one pixel in a bitmap.

The memory requirements for a 640 × 480 bitmap is 37.5 KB.

### Grayscale images

In Grayscale images, a pixel is described by one byte or 8 bits. The image consists of eight bit planes in one channel. You can combine the 8 bits in up to 256 combinations. So a pixel can, for example, have the following values:

0 (black), 64 (dark gray), 128 (gray), 192 (light gray) and 255 (white)

The memory requirements for a 640 × 480 Grayscale image is 300 KB.

### RGB images

Televisions, computer monitors, scanners and our eyes work with the emission respectively the absorption of Red, Green and Blue light rays. The combination of these three colors in different intensities can produce millions of different colors. These colors are actually light rays which have a certain frequency or wavelength. Imagine figure 2.1.1.1 as if one were in a dark room and were projecting three colored lamps onto the wall. When mixed together, the frequencies are added together. This is the additive color mixture.

*Fig. 2.1.1.1: Projection of three lamps with the colors Red, Green and Blue onto a wall in a dark room*

If you further imagine that each lamp can be set to 256 intensity settings, you could mix up to 256 × 256 × 256 = 16.7 million colors! Each color (red, green and blue) – in Photoshop called channels – is described thus in 8 bits or one byte. One pixel in your RGB-image is described by 3 × 8 bits = 24 bits!

The memory requirements for a 640 x 480 RGB image is 900 KB.

A gray tone can be achieved by setting equal intensities of the three color channels:

| | |
|---|---|
| (0, 0, 0) | black |
| (128, 128, 128) | gray |
| (192, 192, 192) | light gray |
| (255, 255, 255) | white |

RGB images are mostly used for web graphic creation, presentations in general and CD-ROM productions. Currently, FilterMeister supports only RGB images, so most of the plug-ins or filters in this paper are suitable for RGB images only.

### CMYK Images

These images are used specially for print products such as booklets, magazines, catalogs, etc. Most CMYK images are converted after scanning (with the scanner's driver) or when converting images with other color spaces to CMYK. CMYK stands for the inks Cyan (blue), Magenta (pinkish), Yellow and Black. These colors are printed on media and are not emitted colors (such as from the monitor). Thus, light is absorbed from the ink on paper and our eyes see only the reflected light rays (see figure 2.1.1.2).

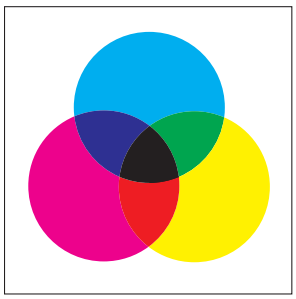


*Fig. 2.1.1.2: A print of three colored circles (Cyan, Magenta and Yellow) on paper*

Therefore, when nothing is printed, you see the media's color, which in most cases is the paper white. When you print Cyan with Magenta, you'll get a purplish Blue, Magenta with Yellow gets you Red and Yellow with Blue obviously Green. And when all colors are mixed together, all light rays are absorbed and you see a black area on the white paper.

Since the pixel intensity in each channel is still eight bits or one byte, each CMYK pixel is described by 32 bits. The memory requirements for a 640 x 480 CMYK image is 1200 KB or 1.2 MB.

### L*a*b* Images

If you work with Color Management, then the L*a*b* color system (see figure 2.1.1.3) should be well-known to you. This system is based on a standardized colorimetric measurement. This is based on the way the human eye perceives color. A pixel in L*a*b*-mode has three color values:

| | |
|---|---|
| L* | Lightness of the color |
| a* | green-to-red axis |
| b* | blue-to-yellow axis |

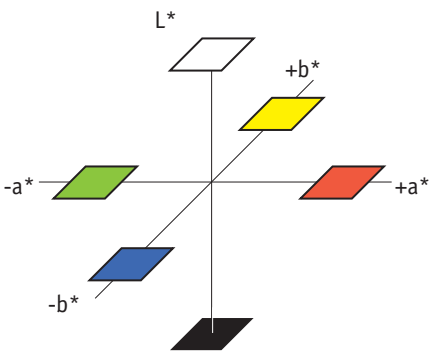



*Fig. 2.1.1.3: The L*a*b* color space has one lightness axis, one green-red axis and one blue-yellow axis*

**Color representation in image host progams**

In each channel of a color space (except for bitmap images, where a pixel is defined by one bit only), the pixel value ranges from 0 to 255. In RGB mode, one can see in Photoshop's information palette values ranging within the mentioned range. In CMYK mode, the information palette shows values between 0 and 100% and in L*a*b* mode, the information palette shows for the Lightness a value between 0 and 100 and for the color axes a value between -128 and 128.

The host program, in this case Photoshop, simply recalculates the values for the color spaces, but works internally within the byte-range (0,255). This can be easily be tested by creating a black-to-white gradient (which is actually a 0 to 255 gradient) in each channel and reading then the min/max values in the information palette.

## 2.1.2 Layers – Channels – Selections

It is assumed that FilterMeister is installed and working in Adobe Photoshop or JASC Paint Shop Pro. There are several other graphic programs which support Photoshop's image features such as layers, channels, selections, paths, etc. If your program supports one or more of these features, they might be called differently, so please consult your manual.

When any filter from the Filter Menu is called, the presented image data can be quite different. For example, when one is working in the *background layer* of an RGB image, the filter recognizes three values for a pixel: the Red, Green and Blue intensities of each pixel. If one is working in a *layer* other than the background layer and call the filter again, four pixel values are evaluated by the filter: Red, Green, Blue and the Alpha channel. Alpha stands for a transparency value and is represented by a byte like the other channels. Low alpha values indicate full transparency and high alpha values indicate full opacity.

Transparency is used with compositing techniques. For example, one has two images. In one image one can see a couple in the Netherlands and in the second image one sees a Hawaiian volcano. With a path, the couple is cut out and pasted as a new layer over the volcano image. The couple's pixels are opaque (not transparent) while the area surrounding the couple is completely transparent. That way, the background image can be seen through the layer.

With FilterMeister, one has access to all channels depending on the *active* layer when the plug-in was called. Calling FM on a background layer does not change any alpha information of a pixel because a background image has no alpha channel. *Note that FilterMeister accesses the active layer only.*

A layer like the one of the couple can be in two states: editable and protected transparency. In the first case, one can change (edit) the transparent areas of an image. In the latter case, Photoshop does not let one paint outside of the opaque areas. These states can be checked by calling the **layers palette** in Photoshop and looking at the small checkbutton called "Preserve Transparency."

Filtering can be affected by another situation. In an either rectangular, elliptical or free form (created with the Lasso or Text tool, for example) selection, the filter will change the pixel information only within the selection and not outside. Currently, FilterMeister takes pixel information from within the selection. The (selected) image area sent to the filter has a different size than the original image. The algorithms used in FilterMeister programs have to be programmed in an image size-independent way: the image seen in the preview window in the plug-in dialog shows the same result like the filter application to the original image (WYSIWYG = what you see is what you get).

### 2.1.3 The filtering method

Plug-ins need a lot of memory in order to be able to process images. When the image is small enough, the plug-in loads the whole image into memory and processes it with the filter algorithm. If the image's size is too large to be loaded into the available memory, then the image is divided up into rectangles which are loaded sequentially into memory to be processed by the plug-in. This process is called *tiling*. The tileability of a plug-in depends on the use of the functions `src()`, `rad()`, `cnv()`, `pset()`, `pget()` which can have access to all pixels in an image. If at least one of these functions is used, the image is *not* tiled.

We have now come to the stage where the image is loaded and the pixels within it are processed by the plug-in. With FM, one can choose from different filter processing functions, called handlers (`ForEveryTile`, `ForEveryPixel` and `RGBA`). The `ForEveryPixel` and `RGBA` handlers process the image:

- tile-by-tile
- within each tile, row-by-row (top-to-bottom)
- within each row, column-to-column (pixel-by-pixel from left-to-right)
- for each pixel, channel-by-channel (e.g., red, green, blue and transparency)

The `ForEveryTile` handler allows a custom plug-in processing direction as one needs it. One does not even have to process all rows, columns, channels or pixels. It is possible, for example, to program the plug-in to first process the columns and then the rows or process every n-th pixel in the width. The `ForEveryTile` handler is also good to do some pre-filtering calculations or for multi-filter processing (e.g., first blur the image, then equalize the colors and sharpen the image).

## 2.2 Image characteristics

### 2.2.1 Mean value and Deviation

The two image characteristics mean value and deviation are often needed for digital processing tasks or for statistical reasons.

The mean value $m_z$ of the channel z states if the channel is darker or lighter in the whole. A grayscale image with m=170, for example, is identified to be lighter. An RGB image having a red channel mean value of $m_R$=87, a green channel mean value of $m_G$=110 and a blue channel mean value of $m_B$=77 is seen as a dark green image (for example an image of a tree landscape). Note that the mean value cannot state if the image is contrasty or not – when, for example, one image contains the gray tone 128 and the other image contains a checkerboard pattern consisting of the two gray tones 0 and 255, the same mean value of m=127.5 is returned.

$$m_z = \frac{1}{X \times Y} \sum_{x=0}^{X-1} \sum_{y=0}^{Y-1} src(x,y,z) \tag{2.1}$$

The deviation $q_z$ returns the contrast value of the current channel z. Lower values specify that an image is in low contrast (flat) while high values specify that an image is high in contrast (contrasty).

$$q_z = \frac{1}{X \times Y} \sum_{x=0}^{X-1} \sum_{y=0}^{Y-1} \left( src(x,y,z) - m_z \right)^2 \tag{2.2}$$

The above equations were slightly modified and were taken from [Hab89], page 25. Listing 2.2.1 computes the mean value and the deviation of each channel in an RGB image.

```
// Listing 2.2.1
// Compute the mean value and deviation of each channel in an RGB image

%ffp

Category: "DIFP 2"
Title: "Mean value and Deviation"

ForEveryTile:
{
float sum_r=0, sum_g=0, sum_b=0;        // Variables for summing up all pixel values
float mw_r=0, mw_g=0, mw_b=0;           // Variables for mean value for each channel
float de_r=0, de_g=0, de_b=0;           // Variables for variance for each channel
```

```
for (y=0; y<Y; y++)
for (x=0; x<X; x++)
{
        sum_r=sum_r + (double) src(x,y,0);      // Sum up all red channel pixels
        sum_g=sum_g + (double) src(x,y,1);      // dito for green
        sum_b=sum_b + (double) src(x,y,2);      // dito for blue
}

mw_r=sum_r/(double) (X*Y);              // Calculate mean value (red)
mw_g=sum_g/(double) (X*Y);              // dito for green
mw_b=sum_b/(double) (X*Y);              // dito for blue

for (y=0; y<Y; y++)
for (x=0; x<X; x++)
{
        de_r=de_r + pow((double) src(x,y,0)-mw_r, 2.0);  // Deviation sum for red channel
        de_g=de_g + pow((double) src(x,y,1)-mw_g, 2.0);  // dito for green channel
        de_b=de_b + pow((double) src(x,y,2)-mw_b, 2.0);  // dito for blue channel
}

de_r=de_r/(double) (X*Y);               // Calculate deviation (red channel)
de_g=de_g/(double) (X*Y);               // dito for green channel
de_b=de_b/(double) (X*Y);               // dito for blue channel


Info("Image Dimensions = %d x %d\n\n"
"Mean value Red = %.1f\tDeviation Red = %.1f\n"
"Mean value Green = %.1f\tDeviation Green = %.1f\n"
"Mean Value Blue = %.1f\tDeviation Blue = %.1f\n"
,X,Y, mw_r, de_r, mw_g, de_g, mw_b, de_b);

return false;   // do not change image
}
```

*Fig. 2.2.1: The right image (b) is higher in contrast than the left image (a)*

The above plug-in computed for the images in figure 2.2.1 the following mean and deviation values:

| | *Image a)* | | *Image b)* | |
|---|---|---|---|---|
| **Channel** | **Mean value** | **Deviation** | **Mean value** | **Deviation** |
| Red | 110.9 | 3138.0 | 96.1 | 6367.8 |
| Green | 127.4 | 3444.4 | 122.9 | 7259.1 |
| Blue | 127.8 | 4734.2 | 124.7 | 10685.4 |

## 2.2.2 Line Histogram

Line histograms describe the color value of the pixels on a vertical or horizontal line. These can be used as a preprocessing task to later be able to measure lengths. One can imagine a robot camera photographing a drill hole in a workpiece. One of the preprocessing tasks would be the comparison between two line histograms at certain coordinates to see if the drill hole is within the tolerance.

The following program creates a row or column histogram in each RGB channel. Note that the column histogram is rotated by 90°. The check box "Show origin" shows the user from which row or column of the image the histogram is created.

*Note: The line histogram is not to be confused with the image histogram. The line histogram describes the color value on a line while the image histogram describes the relative color amount in the whole image.*

```
// Listing 2.2.2
// Line Histogram

Category: "DIFP 2"
Title: "Mean value and Deviation"

%ffp

ctl[0]: "Row", range=(0,100), Val=50
ctl[2]: RADIOBUTTON(GROUP), "Horizontal line histogram", Val=1
ctl[3]: RADIOBUTTON, "Vertical line histogram"
ctl[4]: GROUPBOX(GROUP), "Orientation", Color=CadetBlue, Pos=(260,17), Size=(110,*)
ctl[6]: CHECKBOX, "Show origin", Val=1

ForEveryTile:
{
// Change user control text depending on radio button
if (ctl(2)) setCtlText(0, "Row");
if (ctl(3)) setCtlText(0, "Column");
return false;
}

R,G,B:
ctl(6) ? // show origin...
ctl(2) ? scl(y,0,Y-1,0,100)==ctl(0) ? 255 : c*3/4              // row histogram
:        scl(x,0,X-1,0,100)==ctl(0) ? 255 : c*3/4              // column histogram

:       // ... or show line histogram in image
ctl(2)?
  scl(y,0,Y-1,0,255) >= 255-src(x, ctl(0)*Y/100, z) ? 0 : 255  // for row histogram
: scl(x,0,X-1,0,255) <= src( ctl(0)*X/100, y, z) ? 0 : 255     // for column histogram
```

## 2.2.3 Image histogram

The histogram describes the relative color amount in a channel of an image. Since a pixel can have a value from 0 to 255, one can use the 256 get/put memory cells which contain the total amount of pixels having that certain pixel value. The frequency of each pixel value in an image is computed with the following formula (see [Hab89], page 27):

$$p_z(c) = \frac{\text{get}(c)}{X \times Y} \qquad \text{where } c = 0, 1, ..., 255 \tag{2.3}$$

The visualization of a histogram is done with a bar graph. The following program creates a PostScript file which contains three histograms for the three channels Red, Green and Blue. It also states the individual channel mean value and deviation. The PostScript file will be created after clicking on OK and clicking on YES in the message window. The PostScript file can easily be converted to a PDF file with different popular software packages such as Adobe Acrobat, 5D NikNak or GhostScript.

```
// Listing 2.2.3
// Histogram PostScript file creator

%ffp

Category: "DIFP 2"
Title: "Histogram File Output"

ForEveryTile:
{
int PS_FILE, iMax;
float fCount, fp, fStart, sum=0, mw=0, de=0;

if (!doingProxy && YesNo("Create file?")==IDYES)
if (PS_FILE=fopen("c:\\FM_Test1.ps", "w"))
{
        fprintf(PS_FILE, "%%test.ps\n/cm { 28.34646 mul } def\n0.04 cm setlinewidth\n");
        fprintf(PS_FILE, "/Helvetica-Oblique findfont 18 scalefont setfont\n6.0 cm 27.5 cm
        moveto (Histogram for .....................) show\n\n");
        fprintf(PS_FILE, "/Helvetica-Oblique findfont 10 scalefont setfont\n8.0 cm 26.8 cm
        moveto (Image size = %d x %d) show\n\n", X, Y);
        fprintf(PS_FILE, "/Helvetica-Oblique findfont 7 scalefont setfont\n2.5 cm 2.0 cm
        moveto (Horizontal axis: Color value; Vertical axis: Pixel amount in %% relative to
        maximum cell content amount) show\n");
        fprintf(PS_FILE, "2.5 cm 1.5 cm moveto (Histogram document creator - Coypright 1999
        by Werner D. Streidt) show\n");

        for (z=0; z<3; z++) // only first three channels
        {
                if (z==0) fprintf(PS_FILE, "\n0.85 0.16 0.1 setrgbcolor\n\n");
                if (z==1) fprintf(PS_FILE, "\n0.3 0.7 0.06 setrgbcolor\n\n");
                if (z==2) fprintf(PS_FILE, "\n0 0.43 0.73 setrgbcolor\n\n");
```

```
        for (i=0; i<256; i++) put(0,i);  // reset cells
        for (y=0; y<Y; y++)
        for (x=0; x<X; x++)
 put( get(src(x,y,z))+1, src(x,y,z)); // count colors and save their number in cells

        fStart=2.53;
        iMax=0;

        for (i=0; i<256; i++) iMax=max(get(i), iMax); // get cell with highest value

        for (i=0; i<256; i++)
        {
                fp=6.0 * (double) get(i)/(double) iMax; // compute result of cell i
                fprintf(PS_FILE, "%.6f cm %.1f cm moveto 0 %.2f cm rlineto\n",
                fStart+i*0.039063, 19.7 - (double) z*8, fp);
        }
        fprintf(PS_FILE, "\nstroke\n\n");
}

fprintf(PS_FILE, "0 0 0 setrgbcolor\n0.6 setlinewidth\n/Helvetica findfont 9
scalefont setfont\n");

for (fCount=19.7; fCount>3.6; fCount-=8)
fprintf(PS_FILE, "\n\n"
"2.5 cm %.1f cm moveto 10 cm 0 rlineto -10 cm 0 rmoveto 0 6 cm rlineto\n"
"2.5 cm %.1f cm moveto 0 0.2 cm rlineto 2 cm 0 rmoveto 0 -0.2 cm rlineto 2 cm 0 "
"rmoveto 0 0.2 cm rlineto 2 cm 0 rmoveto 0 -0.2 cm rlineto 2 cm 0 rmoveto 0 0.2 cm "
"rlineto 2 cm 0 rmoveto 0 -0.2 cm rlineto\n"
"2.4 cm %.1f cm moveto 0.2 cm 0 rlineto 0 1.5 cm rmoveto -0.2 cm 0 rlineto 0 1.5 cm"
" rmoveto 0.2 cm 0 rlineto 0 1.5 cm rmoveto -0.2 cm 0 rlineto 0 1.5 cm rmoveto 0.2 "
"cm 0 rlineto 0 1.5 cm rmoveto\n"
"2.4 cm %.1f cm moveto (0) show 1.7 cm 0 rmoveto (51) show 1.5 cm 0 rmoveto (102) "
show 1.5 cm 0 rmoveto (153) show 1.5 cm 0 rmoveto (204) show 1.5 cm 0 rmoveto (255)"
" show\n"
"2.0 cm %.1f cm moveto (0) show -0.3 cm 1.5 cm rmoveto (25) show -0.35 cm 1.5 cm "
rmoveto (50) show -0.35 cm 1.5 cm rmoveto (75) show -0.47 cm 1.5 cm rmoveto (100) "
"show\n",
fCount, fCount-0.1, fCount, fCount-0.5, fCount-0.1);

fprintf(PS_FILE, "/Helvetica findfont 12 scalefont setfont\n14 cm 25 cm moveto (Red"
" channel) show\n14 cm 17 cm moveto (Green channel) show\n14 cm 9 cm moveto (Blue "
"channel) show\n\n/Helvetica findfont 9 scalefont setfont\n\n");

for (z=0; z<3; z++)
{
        sum=0;
        mw=0;
        de=0;

        for (y=0; y<Y; y++)
        for (x=0; x<X; x++)
                sum=sum + (double) src(x,y,z);          // Sum up all channel pixels
        mw=sum/(double) (X*Y);                          // Calculate mean value

        for (y=0; y<Y; y++)
        for (x=0; x<X; x++)
                de=de + pow((double) src(x,y,0)-mw, 2.0);// Deviation sum for channel
        de=de/(double) (X*Y);                            // Calculate deviation
```

```
                fprintf(PS_FILE, "14 cm %.1f cm moveto (Mean Value = %.1f) show\n", 24.0 -
                (double) z*8, mw );
                fprintf(PS_FILE, "14 cm %.1f cm moveto (Deviation = %.1f) show\n", 23.5 -
                (double) z*8, de );
        }
        fprintf(PS_FILE, "stroke\nshowpage\n");
        if (fclose(PS_FILE))
                Error("Cannot close PostScript file!");
}
else
        Error("Cannot write PostScript file\nDrive  is either full or write-protected!");

return false; // leave image unchanged
}
```

Figure 2.2.3 shows a PDF file showing the channel histograms of the left image in figure 2.2.1.

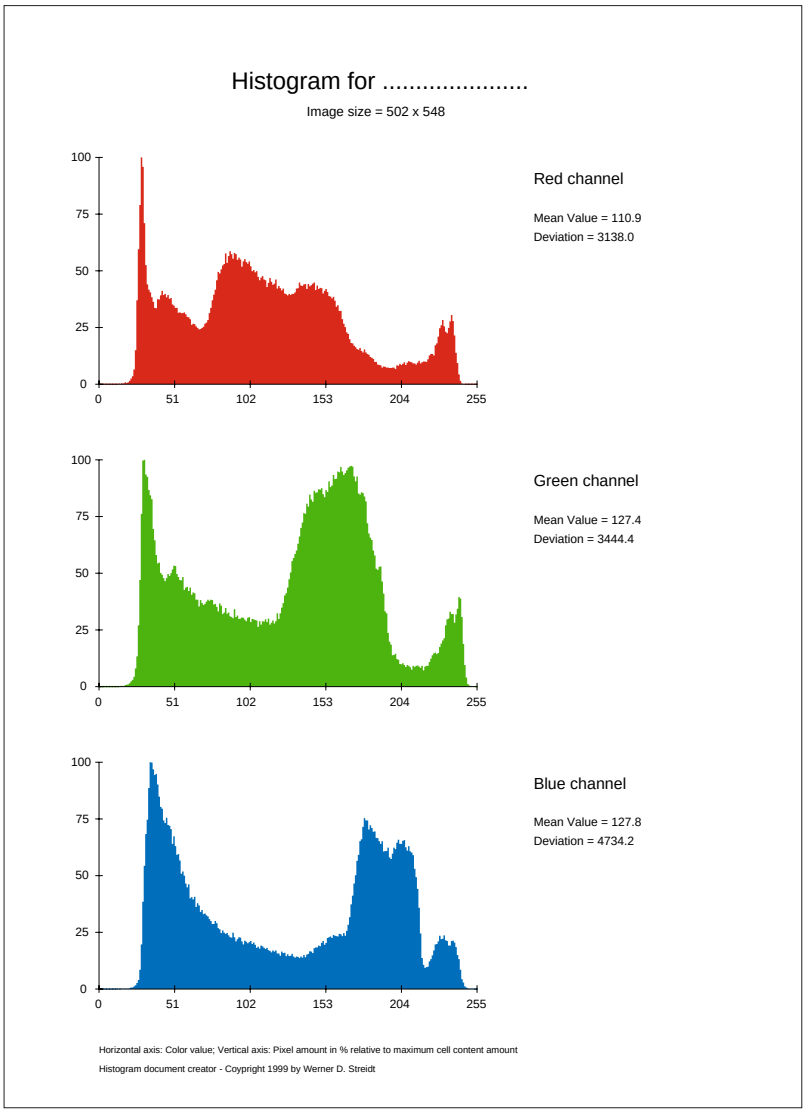



*Fig. 2.2.3*
*View of the PDF file*

# 3. Image Reduction and Compression

As shown in chapter 2.1.1 Color Models, the more channels an image has, the larger its file size on media. Several successful attempts to reduce the file size to a minimum have been accomplished. One can split the file types to image reducing techniques and image compressing techniques. An image reduction leads to a loss of pixel information – one typical example is the image file format JPEG. Image compression algorithms try to reduce the file size without loosing any pixel information – typical examples are the run-length encoding (in the BMP image file format) or Lempel-Ziv-Welch (or LZW, as used in the GIF and TIFF image file formats) algorithms.

In the following chapters, one example for image compression and one for image reduction are presented.

## 3.1 Run-Length Encoding

An uncompressed RGB-image file is saved as follows:

The **file head** contains:
- a code that host programs need in order to recognize the format (two bytes)
- two bytes for the image width and two bytes for the image height (images as large as 65,535 × 65,535 pixels can be saved and loaded)
- channel amount (one for grayscale, three for RGB, four for CMYK images plus any additional channels, for example for masks) saved in one byte

Note: Other head elements such as image resolution, comments, etc. are irrelevant for this paper.

The **file body** contains the values of all pixels contained in all channels.

Example:

An RGB image, being 640 × 480 pixels in size, requires:
3 channels × 640 width × 480 height = 921,600 bytes or 900 KB for the file body

The *run-length encoding* algorithm is fairly simple. The **file head** of a compressed image file resembles the one used above for uncompressed images. Only the format code differs (that way, host programs recognize that the following image file is RLE compressed).

The **file body** contains a byte pair for succeding pixels having the same value. The first byte contains the amount of pixels having the same value and the second byte contains the color value itself.

For example, the following 10 pixels in a row can be coded in only four bytes:

```
100     100     100     167     167     167     167     167     167     167     (uncompressed)

3       100     7       167     (RLE-encoded)
```

The uncompressed method requires 10 bytes, six bytes more than the RLE algorithm. As one can guess easily, RLE compression is useful for images with large areas with one color. If each pixel in an image differs in its value from its neighboring pixel (as most natural images such as photographs do), the file size can double with RLE compression in the worst case. For example, the file sizes for the left image in figure 2.2.1 are uncompressed 805 KB and compressed 1.44 MB.

The following program presents a dialog window with four check buttons. The user has the possibility of saving the current image to disk either RLE compressed or uncompressed. To prove that the saving methods work, the bottom two check buttons allow the user to load back the image file. The program works with RGB images and RGB images with alpha/transparency channel (i.e. any image layer other than the background layer).

```
// Listing 3.1
// Image compression with Run-Length encoding

%ffp

Category: "DIFP 3"
Title: "RLE Image compression"

ctl[0]: RADIOBUTTON(GROUP), "Save Image (uncompressed)", Size=(110,*), Pos=(190,30), Val=1
ctl[1]: RADIOBUTTON, "Save Image (RLE)", Pos=(190,40)
ctl[2]: RADIOBUTTON, "Load Image (uncompressed)", Size=(110,*), Pos=(190,55)
ctl[3]: RADIOBUTTON, "Load Image (RLE)", Pos=(190,65)
ctl[4]: GROUPBOX(GROUP), Pos=(180, 20), Size=(125,60)

ForEveryTile:
{
int IMG_FILE, iCount;
if (!doingProxy) // only perform functions on large image
{
        switch (ctl(0)*1+ctl(1)*2+ctl(2)*3)
        {
        case 1:         // save image uncompressed
                if (IMG_FILE=fopen("d:\\FM_image0.fmi", "wb"))  {
                        // file head
                        fputc(70, IMG_FILE);            // 'F'
                        fputc(48, IMG_FILE);            // '0' for uncompressed
                        fputc((X&65280)/256, IMG_FILE); // High bits for image width
                        fputc(X&255, IMG_FILE);         // Low bits for image width
                        fputc((Y&65280)/256, IMG_FILE); // High bits for image height
                        fputc(Y&255, IMG_FILE);         // Low bits for image height
                        fputc(Z, IMG_FILE);             // channel amount
                        // file body (pixel info)
                        for (z=0; z<Z; z++)
                        for (y=0; y<Y; y++)
                        for (x=0; x<X; x++)
```

```
                        fputc(src(x,y,z), IMG_FILE); //save each pixel individually
        if (fclose(IMG_FILE))   ErrorOk("Cannot close image file!");
        }
        else
        ErrorOk("Cannot write image file\nDrive is either full or write-protected!");
        break;

case 2:         // save image RLE compressed
        if (IMG_FILE=fopen("d:\\FM_image1.fmi", "wb"))
        {
                // file head
                fputc(70, IMG_FILE);            // 'F'
                fputc(49, IMG_FILE);            // '1' for compressed
                fputc((X&65280)/256, IMG_FILE); // High bits for image width
                fputc(X&255, IMG_FILE);         // Low bits for image width
                fputc((Y&65280)/256, IMG_FILE); // High bits for image height
                fputc(Y&255, IMG_FILE);         // Low bits for image height
                fputc(Z, IMG_FILE);             // channel amount

                // file body (pixel info)
                for (z=0; z<Z; z++)
                {
                iCount=1;
                put(src(0,0,z),0);      // put value of origin in cell 0
                for (y=0; y<Y; y++)
                for (x=0; x<X; x++)
        if ( (x!=X-1 ? src(x+1,y,z) : src(0,y+1,z)) == get(0)) // next pixel equal?
                        {
                        iCount++; // increment counter
        if (iCount==255) // are there more than 255 pixels having the same color?
                                {
                                fputc(iCount, IMG_FILE);
                                fputc(get(0), IMG_FILE);
                                iCount=0;
                                }
                        } else  {
                                fputc(iCount, IMG_FILE);        // amount
                                fputc(get(0), IMG_FILE);        // pixel value
                                iCount=1;                       // reset counter
        put( (x!=X-1 ? src(x+1,y,z) : src(0,y+1,z)), 0);        // next pixel value
                        }
if (iCount-1) // in case last pixel in channel is different from the fore-pixel
                        {
                                fputc(iCount-1, IMG_FILE);      // amount
                                fputc(get(0), IMG_FILE);        // pixel value
                        }
                }
        if (fclose(IMG_FILE)) ErrorOk("Cannot close image file!");
        }
else ErrorOk("Cannot write image file\nDrive is either full or write-protected!");
        break;

case 3:         // load uncompressed image
        if (IMG_FILE=fopen("d:\\FM_image0.fmi", "rb"))
        {       if (fgetc(IMG_FILE)==70 && fgetc(IMG_FILE)==48)
                {       // load image metrics
                        x=fgetc(IMG_FILE)*256+fgetc(IMG_FILE);
                        y=fgetc(IMG_FILE)*256+fgetc(IMG_FILE);
                        z=fgetc(IMG_FILE);
```

```
                                if ( (X==x) && (Y==y) && (Z==z) ) // correct metrics?
                                {       for (z=0; z<Z; z++)
                                        for (y=0; y<Y; y++)
                                        for (x=0; x<X; x++)
                                                pset(x,y,z, fgetc(IMG_FILE));
                                } else
ErrorOk("Your image's metrics have to be:\n\nWidth: %d\nHeight: %d\nChannels: %d", x, y, z);
                        } else
                        ErrorOk("Wrong file format!\nLoad an uncompressed .fmi file!");
                if (fclose(IMG_FILE)) ErrorOk("Cannot close image file!");
                } else  ErrorOk("Cannot load image file\nFile is non-existant or damaged");
                break;

        default:        // load RLE compressed image
                if (IMG_FILE=fopen("d:\\FM_image1.fmi", "rb"))  {
                if (fgetc(IMG_FILE)==70 && fgetc(IMG_FILE)==49) {
                        // load image metrics
                        x=fgetc(IMG_FILE)*256+fgetc(IMG_FILE);
                        y=fgetc(IMG_FILE)*256+fgetc(IMG_FILE);
                        z=fgetc(IMG_FILE);
                        if ( (X==x) && (Y==y) && (Z==z) ) // correct metrics?
                        {       x=0; y=0; z=0;
                                while ( (iCount=fgetc(IMG_FILE)) >=0 )
                                {       put(fgetc(IMG_FILE),0);
                                        for (i=0; i<iCount; i++)
                                        {       pset(x,y,z, get(0));
                                                if ((x+1) < X)  x++;
                                                else    {
                                                        x=0;
                                                        if ((y+1) < Y)  y++;
                                                        else    { y=0;  z++;    }
                                                        }
                                        }
                                }
                        } else
ErrorOk("Your image's metrics have to be:\n\nWidth: %d\nHeight: %d\nChannels: %d", x, y, z);
                } else  ErrorOk("Wrong file format!\nLoad a RLE compressed .fmi file");
                if (fclose(IMG_FILE)) ErrorOk("Cannot close image file!");
                } else  ErrorOk("Cannot load image file\nFile is non-existant or damaged");
                break;
        }       // end switch
}               // end if (!doingProxy)
return true;    // update image (important for file loading)
}               // end ForEveryTile
```

## 3.2 Bit plane Reduction

The bit plane reduction is a technique that does not preserve all of the image's pixel values. As explained in 2.1.1 Color Models, a pixel in a channel is one byte or eight bits large, allowing the pixel to have one out of 256 possible values. In binary annotation, the change of a bit in a byte can cause a subtle or a coarse change in value. The most significant bit or MSB in a byte is the one with the value 128 and the least significant bit or LSB in a byte is the one having the value 1:

| 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 |
| MSB | | | | | | | LSB |

The idea of the bit plane reduction is the equalization of the least significant bits (for example setting the LSB to '0') of a pixel value. For example, the value 255 would be changed to 254, or 127 would be changed to 126, etc., if the equalization of Bit 0 (with the value 1) was set to 0.

With program 3.2, the user can observe how the equalization of each bit plane changes the image's appearance. The dialog window contains eight check boxes which represent the eight bits in each channel. By default, all bit planes are preserved. The equalization is done by removing the checkmark in the box and the image's appearance can be observed in the preview window.

A special case comes into action when only one check box is active: the preview window shows the contents of that individual bit plane. If one views the LSB, the preview image apparently shows a random field. This means that the LSB does not contain much image relevance and an equalization leads to a subtle change. If one views the MSB, one can see that it contains a good amount of subject information.

The equalization of bits with low significance make these irrelevant for the image's subject. Instead of saving an image with 8 bits per channel, one could "remove" one or two LSBs and save the image with only 6 or 7 bits per channel – this is the reason why it is called a *reduction* or *lossy* technique.

The problem of most image file formats is that a pixel information in a channel is saved in a byte (in the physical medium) – a new algorithm is needed in order to combine the six or seven bit pixel values in a byte for the data file. For example, a 100 × 100 RGB image with 8 bits per pixel has a file size of 1,000 × 1,000 × 3 channels ÷ 1,024 bytes = 2,9 MB. The same RGB image with 6 bits per pixel has a file size (with the 6-to-8 bit reordering algorithm) of 1,000 × 1,000 × 3 channels × 6 bit ÷ 8 bit ÷ 1024 bytes = 2,1 MB.

This leads to a notable file size reduction (which was the sense of this chapter), but experiments with different file formats showed that only the LZW-compressed TIFF-format "supports" the bit plane equalization technique. Table 3.2 shows how much physical memory can be saved in one example (the image is a portrait and it is assumed that all images have the same size).

| | **RGB** | **CMYK** | **L*a*b*** | **Grayscale** |
|---|---|---|---|---|
| normal file size TIFF | 29,2 MB | 39 MB | 29,2 MB | 9,8 MB |
| TIFF LZW | 18,7 MB | 24,7 MB | 16 MB | 6,6 MB |
| Bit 0 gone | 14,7 MB (78%) | 19,2 MB (77%) | 11,6 MB (91%) | 5,1 MB (77%) |
| Bit 0 and 1 gone | 10,7 MB (57%) | 14,1 MB (57%) | 8,3 MB (72%) | 3,7 MB (56%) |
| CD-ROM (650 MB) image amount: | | | | |
| normal file size TIFF | 22 | 16 | 22 | 66 |
| TIFF LZW | 34 | 26 | 40 | 98 |
| Bit 0 gone | 44 (200%) | 33 (206%) | 56 (254%) | 127 (192%) |
| Bit 0 and 1 gone | 60 (270%) | 46 (287%) | 78 (355%) | 175 (265%) |

*Table 3.2: Comparison between normal and different LZW compressed images. Numbers in parentheses describe image size in relation to TIFF LZW*

*The bottom part describes the amount of images that can be saved on a CD-ROM with the above compressions*

The above seems very interesting for prepress companies who maintain image databases, but the problem with the color consistency (for those who work with color management) emerges.

```
// Listing 3.2
// Image reduction with bit plane equalization

%ffp

Category: "DIFP 3"
Title: "Bit plane equalization"

ctl[1]: CHECKBOX, "Bit 7 (MSB)", Val=1
ctl[2]: CHECKBOX, "Bit 6", Val=1
ctl[3]: CHECKBOX, "Bit 5", Val=1
ctl[4]: CHECKBOX, "Bit 4", Val=1
ctl[5]: CHECKBOX, "Bit 3", Val=1
ctl[6]: CHECKBOX, "Bit 2", Val=1
ctl[7]: CHECKBOX, "Bit 1", Val=1
ctl[8]: CHECKBOX, "Bit 0 (LSB)", Val=1

R,G,B:
(ctl(1)+ctl(2)+ctl(3)+ctl(4)+ctl(5)+ctl(6)+ctl(7)+ctl(8))!=1 ?
c & (ctl(8)*1+ctl(7)*2+ctl(6)*4+ctl(5)*8+ctl(4)*16+ctl(3)*32+ctl(2)*64+ctl(1)*128)
:
(c & (1<<(ctl(1)*7+ctl(2)*6+ctl(3)*5+ctl(4)*4+ctl(5)*3+ctl(6)*2+ctl(7)*1+ctl(8)*0)))*255
```

# 4. Image and Color Modifications

In this chapter, basic approaches to the modifications of pixel positions and pixel values are demonstrated. While the first two chapters deal with the problems of pixel position modification such as translation, rotation, mirroring and scaling, the last chapter explains different techniques to modify colors in images, such as the equalization of images in low contrast or special tasks like image inversion or color correction.

## 4.1 Translation and Rotation

The above transformation functions are often used in effect plug-ins (for example the displacement of pixel information in 3D landscapes for a more realistic effect) or for correct image alignment (such as the correction of scanned rotated images). This chapter will show several examples on image translation and rotation. The scaling function is shown in the following chapter 4.2 *Scaling techniques*.

### 4.1.1 Translation

The basic functions needed for pixel translation tasks are:

| | |
|---|---|
| `pset(x,y,z, a)` | to set a pixel in the destination image at the coordinate (x,y,z) with the value a |
| `tset(x,y,z, a)` | to set a pixel in the buffer 1 image at the coordinate (x,y,z) with the value a |
| `t2set(x,y,z, a)` | to set a pixel in the buffer 2 image at the coordinate (x,y,z) with the value a |
| | |
| `src(x,y,z)` | to fetch a pixel from the source image at the coordinate (x,y,z) |
| `pget(x,y,z)` | to fetch a pixel from the destination image at the coordinate (x,y,z) |
| `tget(x,y,z)` | to fetch a pixel from the buffer 1 image at the coordinate (x,y,z) |
| `t2get(x,y,z)` | to fetch a pixel from the buffer 2 image at the coorindate (x,y,z) |

Note that it is not necessary to copy the source image to the destination image, since this is done automatically by FilterMeister before the plug-in program is applied. The pixel values in both image buffers are set to 0 in all channels.

If a `(x,y,z)` coordinate is set out of range, the corresponding coordinate variable is clamped to the range's borders. This means that edge pixels are repeated, which is not necessarily a desired effect. When creating custom tile images for internet web page backgrounds, the edges have to be retouched to make the tile seamless. The best way to retouch the image is to offset it vertically and horizontally half the image's height respectively width. Additionally, the pixels moving off the image or selection have to be "wrapped around", so they can appear on the other side.

The FilterMeister program that does this task uses special conditions within a `src()`-function (see listing 4.1.1.a)

```
// Listing 4.1.1.a
// Image is offsetted by half the width and height and wrapped-around

%ffp

Category: "DIFP 4"
Title: "Image wrap-around"

R,G,B,A:   src(x<(X/2) ? X-(X/2)+x : x-(X/2), y<(Y/2) ? Y-(Y/2)+y : y-(Y/2), z)
```

As you can see in the listing above, the current pixel is tested if it lies in the first half of the width/height (`x<(X/2) ?`). If it is, then the wrap-around algorithm is used (`X-(X/2)+x`) . If the current pixel lies in the second half of the width/height, the offset algorithm is used (`x-(X/2)`).

The following program (listing 4.1.1.b) is also interesting for web tile creation. It takes an area one quarter the image's size, sets it at the top-left of the image and mirrors it vertically and horizontally (see figure 4.1.1, right image). The user can freely define the area by using the scroll bars. By clicking on the check box, the source area (i.e., the area that is mirrored) is shown by toning down the surrounding areas (see figure 4.1.1, left image). The `doingProxy` variable prevents the application of the *toning down*-algorithm of the final image.

*Figure 4.1.1: The left image shows the source area which is repositioned in the right image to the top-left and mirrored vertically and horizontally*

```
// Listing 4.1.1.b
// Takes an area one quarter the image's size and mirrors it verically and horizontally

%ffp

Category: "DIFP 4"
Title: "WebTileMeister"

ctl[0]: "Horizontal offset (in %)", Range=(0,100)
ctl[1]: "Vertical offset (in %)", Range=(0,100)
ctl[3]: CHECKBOX, "Show tile source"

R, G, B:
ctl(3) && doingProxy ?  // show tile source and is the filter applied to the preview window?
( x < ( 0 + scl(ctl(0),0,100,0,X/2) ) || x > ( X/2 + scl(ctl(0),0,100,0,X/2) ) ||
 y < ( 0 + scl(ctl(1),0,100,0,Y/2) ) || y > ( Y/2 + scl(ctl(1),0,100,0,Y/2) ) ) ? i/2 : c
:
src( (x<X/2 ? x:X-x-1)+scl(ctl(0),0,100,0,X/2), (y<Y/2 ? y:Y-y-1)+scl(ctl(1),0,100,0,Y/2), z)

A:
ctl(3) && doingProxy ? // show tile source and is the filter applied to the preview window?
a :
src( (x<X/2 ? x:X-x-1)+scl(ctl(0),0,100,0,X/2), (y<Y/2 ? y:Y-y-1)+scl(ctl(1),0,100,0,Y/2), z)
```

## 4.1.2 Rotation

The rotation of images or image parts can be done in polar coordinates with the `rad()`-function. In some special cases, the rotation of 90° and 180° can be done with the `src()`-function. The following terms can be simply tested by placing them within the `RGBA` handler.

### Rotation with the src()-function

| | |
|---|---|
| `src(X-x-1, y, z)` | mirrors image horizontally |
| `src(x, Y-y-1, z)` | mirrors image vertically |
| `src(y, x, z)` | rotates image 90° counterclockwise* |
| `src(y, X-x-1, z)` | rotates image 90° clockwise* |
| `src(X-x-1, Y-y-1, z)` | rotates image 180° (or simultaneous horizontal and vertical mirror) |

* *these codes only work correctly on images with equal width and height (assuming that no image information is wanted to be lost); currently, FilterMeister plug-ins cannot change the width and height of the current image*

### Rotation with the rad()-function

The easiest way to rotate an image in any angle around the image's center is to use the angle (or direction) variable d in the polar coordinate function `rad(d,m,z)`. Note that the range from zero rotation to full image rotation is (0, 1024). The following program lets the user rotate the image clockwise or counterclockwise by any angle in degrees, where the user control range is (-360°, 360°).

```
// Listing 4.1.2
// Rotate image by an angle between -360° and 360°

%ffp

Category: "DIFP 4"
Title: "Image rotation"

ctl[0]: "Angle", Range=(-360,360), Track

R,G,B,A:   rad(d- scl(ctl(0),-360,360, -1024,1024), m ,z)
```

*Figure 4.1.2: 30°-rotation with rad()-function*

After application of the above plug-in one can note that the image seems a bit "jaggy". Because of speed purposes, the `rad()`-function processes the image with integer arithmetic. The rounding problems lead thus to a mispositioning of pixel values and the image appears jagged (see figure 4.1.2).

## 4.2 Scaling techniques

The scaling of an image can be done in various ways. In this chapter, three techniques to zoom in and out of the image are described.

### 4.2.1 Polar coordinate zoom

A simple way to zoom in and out of an image is the usage of the polar coordinate function `rad()`. While in chapter 4.1.2 Rotation the variable `d` was used to rotate the image around the image's center, in this chapter the variable `m` (magnitude or distance from image's center) is interesting for zooming purposes (where the zooming direction is still the image's center).

*Note: While it is mathematically possible to zoom into infinity, the limit is physically set to the image's discrete metrics.*

In Listing 4.2.1, the zooming in and out with the rad()-function is demonstrated. The ForEveryTile handler is used to update the zoom factor text below the scroll bar. Analogous to the rotation, a scale with the rad()-function leads to a jagged image (see figure 4.2.1).

*Figure 4.2.1: 200% zoom with the rad()-function*

```
// Listing 4.2.1
// Zooming in and out with polar coordinates technique

%ffp

Category: "DIFP 4"
Title: "Radial zoom"

ctl[0]: "Zoom In/Out", Range=(1,100), Val=10
ctl[2]: STATICTEXT, ""

ForEveryTile:
{
setCtlTextv(2, "Zoom %s by %d %%", ctl(0)>10 ? "out" : "in", 1000/ctl(0));
return false; // Let RGBA-handler do the filtering
}

R,G,B,A: rad(d,m*ctl(0)/10,z)
```

## 4.2.2 Nearest Neighbor

A fast and better method than the Radial zoom plug-in in 4.2.1 is the image scaling with the nearest neighbor method, see [Reis97]. It takes the scaling factor and calculates the coordinates of the nearest source pixel. The ouput of this plug-in is good for downscaling (zoom out) and low upscaling (zoom in) settings (see figure 4.2.2), but high quality cannot be expected. A high setting in the scale factor leads to a "mosaic" image. Adobe Photoshop also offers this method to rescale images.

Listing 4.2.2 lets the user zoom in and out of the image and adds an offset adjustment in vertical and horizontal directions. A text under the scroll bars reveals the actual scaling factor (and is updated by the ForEveryTile handler).

*Figure 4.2.2: 200% zoom with the nearest neighbor algorithm*

The first part of the filter algorithm leaves the source image untouched if the user wants to zoom out (that way, no edge pixels are repeated) and two different codes are needed to either zoom in or out. WYSIWYG was also considered in the code, which means that the view of the preview window is the view seen in the final image.

```
// Listing 4.2.2
// Zooming in and out with nearest neighbor algorithm

%ffp

Category: "DIFP 4"
Title: "Nearest Neighbor example"

ctl[0]: "Scale factor (divided by 10)", Range=(1,100), Val=10
ctl[1]: "x Offset", Range=(0,100)
ctl[2]: "y Offset", Range=(0,100)
ctl[4]: STATICTEXT, ""

ForEveryTile:
{
setCtlTextv(4, "Zoom %s by %d %%", ctl(0)>10 ? "out" : "in", 1000/ctl(0));
return false;   // Let RGBA-handler do the filtering
}

R,G,B:
x*ctl(0)/10-ctl(1)*X/100*(ctl(0)-10)/10 < 0 || x*ctl(0)/10-ctl(1)*X/100*(ctl(0)-10)/10>X-1 ||
y*ctl(0)/10-ctl(2)*Y/100*(ctl(0)-10)/10 < 0 || y*ctl(0)/10-ctl(2)*Y/100*(ctl(0)-10)/10>Y-1 ?
c               // Leave image unchanged if zooming out and pixel is out of bounds
:
ctl(0)>=10 ?    // zoom out...
    src( x*ctl(0)/10-ctl(1)*X/100*(ctl(0)-10)/10, y*ctl(0)/10-ctl(2)*Y/100*(ctl(0)-10)/10, z)
    :           // ...or zoom in ?
    src( x*ctl(0)/10+ctl(1)*X/100, y*ctl(0)/10+ctl(2)*Y/100,z)
```

## 4.2.3 Linear Interpolation

The linear interpolation method is more complicated than the first methods mentioned in the past two chapters. As to [Reis97], the idea behind the interpolation is to add the missing pixels between the existing ones to simulate the restoration of details in the image. Nevertheless, there is currently no method that can reconstruct missing details. Figure 4.2.3.1 shows the differences between the scale methods nearest neighbor and linear interpolation with a scale factor of 3 respectively 300%.

Source image

| | |
|---|---|
| 100 | 51 |
| 78 | 192 |

Nearest neighbor method

| | | | |
|---|---|---|---|
| 100 | 100 | 100 | 51 |
| 100 | 100 | 100 | 51 |
| 100 | 100 | 100 | 51 |
| 78 | 78 | 78 | 192 |

Linear Interpolation

| | | | |
|---|---|---|---|
| 100 | 83 | 67 | 51 |
| 92 | 93 | 95 | 98 |
| 85 | 104 | 124 | 145 |
| 78 | 116 | 154 | 192 |

*Figure 4.2.3.1: Difference between nearest neighbor method and linear interpolation at a scale factor of 3.0*

Although this method is acceptable for medium scale factors (see figure 4.2.3.2), the bilinear and bicubic interpolation, as offered in Adobe Photoshop, lead to better results.

In Listing 4.2.3, the mix()-function (see the FilterMeister User Guide) is used to compute the value of the current pixel according to its distance from the two source image pixels. The program is divided in two loops. The first loop sets in the destination image the original source pixels and computes the columns with the linear interpolation method. The second loop computes the rows according to the newly sets column colors from the destination image.

*Figure 4.2.3.2: 300% zoom with the linear interpolation algorithm*

```
// Listing 4.2.3
// Zoom in image with linear interpolation technique

%ffp

Category: "DIFP 4"
Title: "Linear Interpolation"

ctl[0]: "Zoom Factor", range=(1,20), val=2, PageSize=1
ctl[1]: "x-Offset", range=(0,100)
ctl[2]: "y-Offset", range=(0,100)

ForEveryTile:
{
// First loop interpolates only columns
for (y=0; y<Y; y++)
for (x=0; x<X; x++)
for (z=0; z<Z; z++)
{
        if (!(x%ctl(0)))
        {
                if (!(y%ctl(0)))        // take original source pixel values
```

```
                        pset(x,y,z, src((x+ctl(1)*X/100)/ctl(0), (y+ctl(2)*Y/100)/ctl(0),z));
                else                    // compute pixel values between source values
                        pset(x,y,z, mix( src((x+ctl(1)*X/100)/ctl(0), (y+ctl(2)*Y/100+ctl(0)-
                        y%ctl(0))/ctl(0)-1, z),src((x+ctl(1)*X/100)/ctl(0), (y+ctl(2)*Y/100+
                        ctl(0)-y%ctl(0))/ctl(0), z), ctl(0)-(y%ctl(0)), ctl(0)));
        }
}
// Second loop interpolates rows
for (y=0; y<Y; y++)
for (x=0; x<X; x++)
for (z=0; z<Z; z++)
        if (x%ctl(0))
                pset(x,y,z, mix( pget(x-x%ctl(0),y,z), pget(x+ctl(0)-x%ctl(0), y, z), ctl(0)-
                x%ctl(0), ctl(0)));

return true;
}
```

## 4.3 Color value modifications

In this section, the modification of color in an image is the basis. First, some basic usages such as image inversion, threshold operations and desaturation are presented. The following techniques deal with the automatization of color correction and the image encryption. Finally, the code for most of the various calculation modes used in graphic programs are described.

### 4.3.1 Basic modifications

#### Image inversion

The inversion of an image is defined as the subtraction of the current color value from the maximum possible value a pixel can have. In common 24-bit RGB images, the maximum value of a byte is 255. The following program inverts any RGB-image.

```
// Listing 4.3.1.a
// Image inversion

%ffp

Category: "DIFP 4"
Title: "Image inversion"

R,G,B:  255-c
A:      a
```

### Image desaturation

The desaturation of an image is meant to reduce all its color saturation and can be used to simulate a RGB to Grayscale image conversion. If one thinks in mathematical logic, one would go and sum up the Red, Green and Blue values of a pixel and divide it by 3. The problem is that the human eye does not perceive color in such a clean mathematical way. It "sees" more green hues than red or even more than blue, so a correction according to the human eye perception is necessary. FilterMeister has the built-in variables `i`, `u` and `v` which represent the color values of the current pixel in the YUV color space, which is used in video technology. While the Y part represents the luminance, the u and v represent the two color axes yellow-blue and red-cyan. The Adobe manual "The Adobe Photoshop Filter Factory" describes in page 12 the color space conversion from RGB to YUV as to:

```
i  =  ((76 × r)  + (150 × g) +  (29 × b))/256       // Luminance axis
u  =  ((-19 × r) + (-37 × g) +  (56 × b))/256       // Yellow-Blue axis
v  =  ((78 × r)  + (-65 × g) + (-13 × b))/256       // Rot-Cyan axis
```

The simplest way to desaturate an image would be to use the variable `i`:

```
// Listing 4.3.1.b
// Desaturate RGB image with variable i

%ffp

Category: "DIFP 4"
Title: "Desaturation"

R,G,B:  i
A:      a
```

Figure 4.3.1 demonstrates what happens when the luminance (left side) and the mathematical average (right side) is computed for a spectral gradient.

*Figure 4.3.1: Desaturation of spectral gradient with luminance (left) and matthematical average (right)*

### Threshold operation

An often used preprocessing operation is the bilevelling of an image. This means that the pixel value range is converted to two possible values. For example, a pixel value within the 8-bit range of (0, 255) is scaled to the 1-bit range (0,1). The threshold technique takes a user-definable color value (threshold) within the 8-bit range and sets all values below the threshold to 0 and the values above the threshold to 1 (or any other value).

The threshold is a preprocessing operation required for the object counting plug-in shown in chapter 5.4. Listing 4.3.1.c takes the current image and lets the user define his/her own threshold for each channel.

```
// Listing 4.3.1.c
// Threshold example

%ffp

Category: "DIFP 4"
Title: "Channel Bilevelling"

ctl[0]: "Red threshold", Val=128
ctl[1]: "Green threshold", Val=128
ctl[2]: "Blue threshold", Val=128
ctl[4]: CHECKBOX, "Lock scroll bars"

ForEveryTile:
{
if (ctl(4))     // Lock scroll bars 1 and 2 to scroll bar 0?
{
        setCtlVal(1,ctl(0));
        setCtlVal(2,ctl(0));
}
return false;
}

R,G,B:  c<ctl(z) ? 0 : 255      // Bilevel channel according to user control z
A:      a                       // Leave alpha channel untouched
```

## 4.3.2 Color Correction

Often, images are scanned or grabbed with low contrast. This can either be a problem of lighting or hardware. A manual or automatic post-correction is required. The manual way is to offer the user the change of brightness and contrast, while the automatic way evaluates the lowest and highest values in the image and rescales the range to the maximum possible.

### Brightness/Contrast

The following program lets the user correct the brightness and contrast of the current image manually. In addition to the correction, the plug-in offers the user the possibility to save and load the current settings. That way, if a sequence of images were created in similar scanning/grabbing settings, the brightness/contrast correction is simply loaded and applied to the image. The algorithm for the correction is found in the RGBA-handler – the `ForEveryTile` is only used to save and load the .INI settings file.

```
// Listing 4.3.2.a
// Brightness/Contrast correction with Save/Restore function

%ffp

Category: "DIFP 4"
Title: "Brightness/Contrast"
```

```
ctl[0]: "Brightness", range=(-100,100), val=0
ctl[1]: "Contrast", range=(-100,100), val=0
ctl[2]: PUSHBUTTON, "Save", size=(34,14), pos=(280,50)
ctl[3]: PUSHBUTTON, "Restore", size=(34,14), pos=(315,50)

ForEveryTile:
{
        int SETTINGS_FILE;      // integer for file status

        if (ctl(2))             // User pushed Save
        {
                if (SETTINGS_FILE=fopen("C:\\Brightness_Contrast.ini", "w"))
                {
                        for (i=0; i<2; i++)
                        fprintf(SETTINGS_FILE, "%d\n",ctl(i));
                        if (fclose(SETTINGS_FILE))
                                Error("Can't close settings file.");
                }
                else
                        Error("Can't write to settings file.");
                setCtlVal(2,0); // Reset push button "Save"
        }

        if (ctl(3))             // User pushed restore
        {
                int iVal;
                if (SETTINGS_FILE=fopen("C:\\Brightness_Contrast.ini", "r"))
                {
                        for (i=0; i<2; i++)
                        {
                                fscanf(SETTINGS_FILE, "%d", &iVal);
                                setCtlVal(i,iVal);
                        }
                        if (fclose(SETTINGS_FILE))
                                Error("Can't close settings file.");
                }
                else
                        Error("Can't open settings file.");
                setCtlVal(3,0); // Reset push button "Restore"
        }

return false;
}       // end ForEveryTile

R,G,B: ctl(1)>0 ?
scl(c+ctl(0),127*ctl(1)/100, 255-127*ctl(1)/100,0,255) :
scl(c+ctl(0),0, 255,-ctl(1)*128/100,256+ctl(1)*128/100)
A:      a
```

**Automatic color correction**

The automatic color correction technique is fairly simple – the minimum and maximum pixel values from each channel is saved in memory cells and the plug-in scales the old range (min, max) to the new range (0,255), for example. Program 4.3.2.b, listed below, includes a checkbox to auto-level the image but also the option to set new min and max values, if the user so desires.

The text below the user control shows the minimum and maximum values for each channel. Figure 4.3.2 shows an example of the automatic color correction on an image.

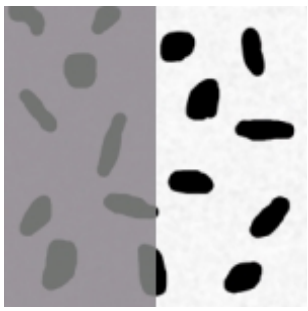


*Figure 4.3.2: Image before (left) and after (right) the automatic color correction*

```
// Listing 4.3.2.b
// Automatic Color Correction

%ffp

Category: "DIFP 4"
Title: "Auto-Level"

ctl[0]: "New min value"
ctl[1]: "New max value", Val=255
ctl[3]: CHECKBOX, "Auto-Level"
ctl[4]: STATICTEXT, Pos=(200,60), Size=(150,40)
ctl[5]: GROUPBOX, "Values", Color=CadetBlue, Pos=(190,50), Size=(150,40)

ForEveryTile:
{
put(255,0);     // Cell 0 for min value (red)
put(0,1);       // Cell 1 for max value (red)
put(255,2);     // Cell 2 for min value (green)
put(0,3);       // Cell 3 for max value (green)
put(255,4);     // Cell 4 for min value (blue)
put(0,5);       // Cell 5 for max value (blue)

for (y=0; y<Y; y++)
for (x=0; x<X; x++)
for (z=0; z<Z; z++)
{
        put(min(get(z*2), src(x,y,z)), z*2);            // save min value in cell z × 2
        put(max(get(z*2+1), src(x,y,z)), z*2+1);        // save max value in cell z × 2 + 1
}
setCtlTextv(4, "Red   min: %d\t\tRed   max: %d\nGreen min: %d\t\tGreen max: %d\n"
"Blue  min: %d\t\tBlue  max: %d", get(0), get(1), get(2), get(3), get(4), get(5));
return false;
}

R,G,B:
ctl(3) ?
scl(c, get(z*2), get(z*2+1), 0, 255  ) :                // do auto-levelling
scl(c, get(z*2), get(z*2+1), ctl(0), ctl(1) )           // user-defined levelling
```

## 4.3.3 Image Encryption

The watermarking of images is not secure enough, because all types of watermarking techniques depend on the image's metrics or certain image's areas. As soon as the image is resized or cut, the watermark information is gone. There is currently no effective solution for inserting a fix watermark for images. One solution to secure an image is to encrypt it.

In contrast to watermarking, image encryption is indeed possible. There are two possibilites of encrypting an image without losing any vital image information:

– displacing pixels in the image
– inverting pixels' color values

Program 4.3.3 shows an example of low image encryption by pixel value inversion. Low encryption means that the image can be recognized at some extent. The method is fairly simple: the conditional term returns either true or false and inverts the current pixel or not. In the conditional term I have used a combination of modulo operations with different values and the bit-wise OR operator. The usage of four scroll bars as encryption keys reveals a 32-bit encryption key.

```
// Listing 4.3.3
// Image Encryption

%ffp

Category: "DIFP 4"
Title: "Image hacker"

ctl[0]: "Key 1"
ctl[1]: "Key 2"
ctl[2]: "Key 3"
ctl[3]: "Key 4"


R: (x%ctl(3)+255-y%5 | y%ctl(2)+x%3)%15 ? c : 255-c
G: (x%ctl(1)+25-y%5  | y%ctl(1)+x%3)%2  ? 255-c : c
B: (x%ctl(0)+55-y%5  | y%ctl(2)+x%3)%5  ? c : 255-c
A: a
```

## 4.3.4 Calculation Modes

The calculation modes (such as Multiply, Difference, Lighten, etc.) in Photoshop are spread throughout the program. They can be set for the application of every tool or the combination of two or more layers. The application modes are also interesting for certain preprocessing tasks (difference between two images) but are also widely used to combine digital textures with the current image. The explanations derive from the Photoshop 4 (pages 208-209) and Photoshop 5 (pages 205-206) user guides.

It is not necessary, though, to incorporate the calculation modes into the algorithm, since Photoshop offers two calculation techniques: the first technique is to apply the texture to the image (so that the texture, but not the image is seen) andsequentially calling the menu command Fade... in the Filter-menu. The second technique leads to an application of the texture on a layer above the image, while setting different calculation modes in the layer palette. Nevertheless, the algorithms are presented here for special cases.

*Note: In the following formulas, the variables $p_1$ and $p_2$ represent two pixels from different channels in the same position. The layer 1 is on top of layer 2.*

**Multiply**

This mode computes the multiplication of the pixel values. The result is always darker than any of the colors. If one of the pixel colors is black, the result is black. If one of the pixel colors is white, no change is done. Picture this mode as if putting one color slide over another.

$$\frac{p_1 \times p_2}{255} \tag{4.1}$$

**Screen**

This mode computes the inversion of the multiplication of the inverse pixel values. The term leads to a lightening of the pixel color. If one of the pixel colors is black, the other color is not modified. If one of the pixel colors is white, the term results to white. Picture this mode as if putting one negative image over another and printing it on a negative photographic paper.

$$255 - \frac{(255 - p_1) \times (255 - p_2)}{255} \tag{4.2}$$

**Overlay**

If the bottom pixel color is darker than the mid-tone, the total combination is darker (multiply). If the bottom pixel color is lighter than the mid-tone, the total combination is lighter (screen).

$$p_2 < 128 ? \frac{2 \times p_1 \times p_2}{255} : 255 - \frac{2 \times (255 - p_1) \times (255 - p_2)}{255} \tag{4.3}$$

**Soft Light**

Depending on the top pixel color, the result is either darkened or lightened. The effect is similar to the burn and dodge tools in Photoshop. If the top pixel color is pure black, the result is somewhat lighter than black. If the top pixel color is pure white, the result is somewhat darker than white. The effect is similar to viewing the image with diffuse light.

$$\begin{aligned} &p_2 < 128 ? \frac{2 \times p_2 \times scl(p_1, 0, 255, 64, 192)}{255} : \\ &255 - \left( 2 \times \left( 255 - scl(p_1, 0, 255, 64, 192) \right) \times \frac{255 - p_2}{255} \right) \end{aligned} \tag{4.4}$$

**Hard Light**

Depending on the top pixel color, the result is darkened or lightened. The effect ist similar to the projection of glaring light onto the image. The resulting image is high in contrast.

$$p_1 < 128 ? \frac{2 \times p_1 \times p_2}{255} : 255 - \frac{2 \times (255 - p_1) \times (255 - p_2)}{255} \tag{4.5}$$

**Darken**

This mode selects the darker pixel color from both.

$$\min(p_1, p_2) \tag{4.6}$$

**Lighten**

This mode selects the lighter pixel color from both.

$$\max(p_1, p_2) \tag{4.7}$$

**Difference**

The difference mode computes the difference from the two values. In other words, it computes the absolute value of the subtraction from one another.

$$\mathrm{dif}(p_1, p_2) \quad \text{or} \quad \mathrm{abs}(p_1 - p_2) \tag{4.8}$$

Figure 4.3.4 demonstrates the combination of gradient with an image. The leftmost images are the original image (top) and the gradient image (bottom). The rest of the images show the combinations of the original and the gradient according to the different calculation modes mentioned above. The gradient was created with the following code:

```
R,G,B: (x*1024/X)%256   // create four linear b/w gradients
```

The calculation algorithms are described below.

```
c*((x*1024/X)%256)/255                  // multiply

255-((255-c)*(255-(x*1024/X)%256))/255  // screen

c < 128 ? 2*c*((x*1024/X)%256)/255 : 255-(2*(255-c)*(255-(x*1024/X)%256))/255    // overlay

c < 128 ? (2* ((x*1024/X)%256)*scl(c,0,255,64,192))/255 : 255- 2*(255-scl(c,0,255,64,192))*
          (255-(x*1024/X)%256)/255                                    // soft light

(x*1024/X)%256 < 128 ?
2*c*((x*1024/X)%256)/255 : 255-(2*(255-c)*(255-(x*1024/X)%256))/255     // hard light

min(c, (x*1024/X)%256)                  // darken

max(c, (x*1024/X)%256)                  // lighten

dif(c, (x*1024/X)%256)                  // difference
```

*Figure 4.3.4: Gradient and image (both leftmost images in first column) combined according to the following calculation modes: multiply, screen, overlay, soft light (top row), hard light, darken, lighten and difference (bottom row)*

# 5. Digital Operators

Digital operators perform low-level neighborhood operations that help restorate image acquiring errors (blurred or disturbed images), help detect edges, cornes and areas and preprocess images for further processing (object counting techniques). A neighborhood operation is an operation performed on a single pixel but that is dependant on surrounding or *neighboring* pixels. Since the operation is performed on a single pixel, the content of the image is changed – this is the reason why neighborhood operations are also called filters, because filters detect certain features (e.g., an edge or a disturbance). Filters are not to be confused with special effects plug-ins such as mirrors, neon glow, mirror effects and texture creation.

The following chapters introduce one- and two-dimensional operators as described in [Rei97], whereas the two-dimensional operators are divided into blur, sharpen, rank value and gradient filters. The last chapter introduces the Hoshen-Kopelman Algorithm that is used for object counting in bilevel images.

## 5.1 One- and Two-Dimensional Operators

The basic idea of filtering an image is the combination of the neighboring pixels' colors through a window or a filter mask, which is computed for all pixels in the image. The mask can be a one- or two-dimensional matrix whose elements describe the weighting factor for each pixel. The mask can be of any desired size and has a kernel where the result is put. Usually, the mask is distinctively smaller than the image.

There are two forms of the mask element combinations:
- for edge detection filters: the sum of the elements result to 0 (see 5.3 Gradient operators)
- for blur or sharpening filters: the sum of the elements is any value, but after weighting the pixel values, the whole result is divided by the sum of the elements (see next example)

An example of a three-point one-dimensional mask is shown in figure 5.1.1. The mask is applied to a row in an image and moves pixel-by-pixel from left to right. Since the sum of the elements is greater than 1, a division with the denominator is needed. The computation of the second pixel value (130) thus reads

$$(128\times1 + 130\times2 + 120\times1\ ) \div (1+2+1) = 127$$

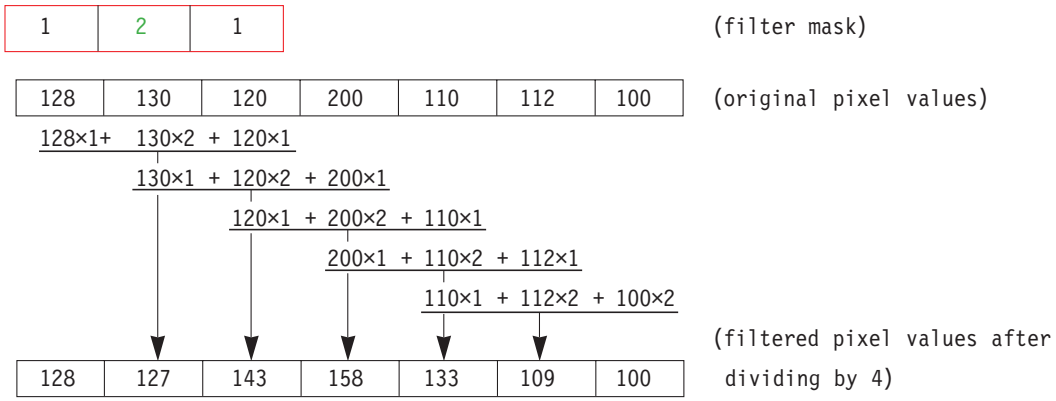



*Figure 5.1.1: Application of a three-point one-dimensional mask to a series of pixels in a row; the denominator is the sum of the mask elements*

If one looks at the resulting values, one can recognize that these values are mathematically "moved" nearer to each other, which means that high frequencies (a steep value change from one pixel to another) are blocked and low frequencies are let through (another reason for the usage of the term "filter"). The visual effect is a slight blur of the image.

A common problem for filters is the correct computation of the pixels at the edges of an image. When the kernel is above an edge pixel, one or more elements of the mask may lie outside the boundaries of the image. There are several proposals to solve this problem:

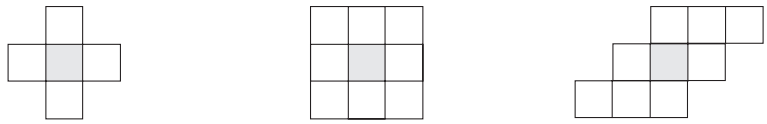
– leave edge pixels as they are
– repeat edge pixels in elements outside the image boundaries
– extrapolate new color values for the elements outside the image boundaries

As one can see in figure 5.1.1, the pixel values connected to the filter mask elements are taken from the source image row. If one wanted to use the newly filtered values, one would speak of filter recursiveness. This can easily be done in FilterMeister. The `src()`-function retrieves original pixels, while the `pget()`-function retrieves the pixels in the destination image. Since the output of a filter operation is set in the destination image, destination image information can be read with the `pget()`-function. The FilterMeister program 5.1.a demonstrates the usage of the above filter mask with and without recursion.

```
// Listing 5.1.a
// One-Dimensional three-point filter with and without recursion

%ffp

Category: "DIFP 5"
Title: "3-point blur filter"

ctl[0]: CHECKBOX, "Recursive filtering"

R,G,B:
ctl(0) ?
(pget(x-1,y,z) + 2*src(x,y,z) + src(x+1,y,z))/4         // recursive filter
:
( src(x-1,y,z) + 2*src(x,y,z) + src(x+1,y,z))/4         // "normal" filter
```

**Two-Dimensional Operators**

In the one-dimensional case, a pixel has two neighbors: one at the left and one at the right side. In the two-dimensional case, every pixel besides the edge pixels in an image have more than two neighbors, so the mask should at least be looking at top and bottom pixels, as well. The masks can change in size and form. Some people prefer to operate with four neighbors (top, left, right and bottom pixels), eight neighbors (top-left, top, top-right, left, right, bottom-left, bottom and bottom-right pixels) and other variations (see figure 5.1.2). Two-dimensional filter masks can have any size (for example 7×7).



*four-neighbor element* *eight-neighbor element* *diagonal element*

*Figure 5.1.2: Different two-dimensional structruing elements; kernel of the mask is colored in gray*

The most used filter mask is the eight-neighbor mask (3×3 mask shown as the middle image in figure 5.1.2). Like in the one-dimensional mask, the elements of the two-dimensional mask describe the weighting factor to be multiplied with the original neighbor pixels. Other structuring element forms are used mostly for rank value filtering as shown in chapter 5.2.

These filters are also called convolution filters and FilterMeister has a special function called `cnv()` for 3×3 filter masks and the function reads as follows:

```
cnv( m11, m12, m13, m21, m22, m23, m31, m32, m33, d )
```

where $m_{xy}$ describes the mask element as shown in figure 5.1.3 and `d` describes the denominator (sum of the mask elements). Note that `d` is always a number other than 0, or else a division by zero error will occur (FilterMeister intercepts this error automatically).

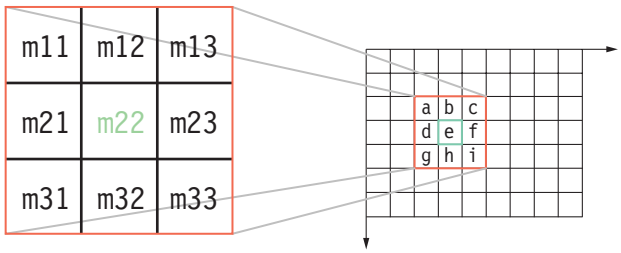



*Figure 5.1.3: 3×3 convolution filter mask (left) applied to a pixel in the image (right); the green color shows the position of the kernel*

The formula for the convolution kernel goes:

$$\frac{m11{\times}a + m12{\times}b + m13{\times}c + m21{\times}d + m22{\times}e + m23{\times}f + m31{\times}g + m32{\times}h + m33{\times}i}{d}$$

Examples:

```
cnv(1, 1, 1, 1, 1, 1, 1, 1, 1, 9)      // simple blur filter (also called low pass filter)
cnv(1, 2, 1, 2, 4, 2, 1, 2, 1, 16)           // simple Gaussian blur
cnv(-1, -1, -1, -1, 9, -1, -1, -1, -1, 1)     // sharpen filter
```

Note that a convolution filter with a matrix greater than 3×3 has to be computed by hand with the `src()`-function. Program 5.1.b demonstrates image blurring in three states:

1. Gaussian blur 3×3 (with the `cnv()`-function)
2. Normal blur (sum up all the neighbor and kernel values and divide by the amount of elements)
3. Gaussian blur 5×5 (with the `src()`-function and normal operation elements)

As described in [Rei97], the Gaussian filter is a binomial filter and a 5×5 convolution filter is shown in figure 5.1.4.

$$\begin{pmatrix}1 & 2 & 1\end{pmatrix}\times\begin{pmatrix}1\\2\\1\end{pmatrix}=\begin{pmatrix}1 & 4 & 6 & 4 & 1\\4 & 16 & 24 & 16 & 4\\6 & 24 & 36 & 24 & 6\\4 & 16 & 24 & 16 & 4\\1 & 4 & 6 & 4 & 1\end{pmatrix}$$

*Figure 5.1.4: matrix computation of a binomial 5×5 convolution filter mask*

```
// Listing 5.1.b
// Two-dimensional convolution filters

%ffp

Category: "DIFP 5"
Title: "Blur examples"

ctl[0]: RADIOBUTTON(GROUP), "Gaussian blur (radius 1)", Val=1
ctl[1]: RADIOBUTTON, "Normal blur"
ctl[2]: RADIOBUTTON, "Gaussian blur (radius 3)"

R,G,B:
ctl(0) ?
cnv(1,2,1,2,4,2,1,2,1,16) :     // Gaussian Blur (radius 1)
ctl(1) ?
cnv(1,1,1,1,1,1,1,1,1,9) :      // Average neighbor blur
// Gaussian blur (radius 3, i.e. 5x5 convolution filter)
( src(x-2,y-2,z)+src(x+2,y-2,z)+src(x-2,y+2,z)+src(x+2,y+2,z)+
4*(src(x-1,y-2,z)+src(x-1,y+2,z)+src(x+1,y-2,z)+src(x+1,y+2,z)+src(x-2,y-1,z)+
  src(x-2,y+1,z)+src(x+2,y-1,z)+src(x+2,y+1,z))+
6*(src(x,y-2,z)+src(x,y+2,z)+src(x-2,y,z)+src(x+2,y,z))+
16*(src(x-1,y-1,z)+src(x-1,y+1,z)+src(x+1,y-1,z)+src(x+1,y+1,z))+
24*(src(x,y-1,z)+src(x-1,y,z)+src(x+1,y,z)+src(x,y+1,z))+
36*src(x,y,z) )/256

A: a
```

## 5.2 Rank Value Filtering

The preprocessing task of rank value filters is the removal of disturbances or noise in an image. Common filters are the median filter, dilation and erosion filters (in Photoshop also known as minimum and maximum) described in this section.

#### Median filter

The median filter sort the values of the neighboring pixels in an ascending order and returns the value in the center of the value list. The main problem is a fast algorithm to sort the pixel values correctly. I will describe the usage of a 3×3 structuring element where the center value is to be evaluated and returned. For example, the matrix of the first 9 pixels in the top-left side of the image returned the following values:

100 89 92 110 100 99 50 94 102

One can see from the values that the pixel value 50 is the furthest from the other pixel values. These values in an ascending order:

50 89 92 94 99 100 100 102 110

The median or center value returned here is 99.

The easiest way to create such a median filter is to copy the 9 pixel values in memory cells and use some sorting method to sort the values in an ascending way (Bubblesort or Quicksort method). After sorting, the center value is placed in the current pixel.

While some sorting methods strongly depend on the amount of values, the fastest method is presented by [Smi98] which is visualized in figure 5.2.1. The graph shows the minimum network to determine the median of nine elements. Each node (black circle) is presented with two input values and sort the lower value to the left and the higher value to the right. Graphs not leading to a node simply ignore the corresponding value, because preceding comparison was enough.

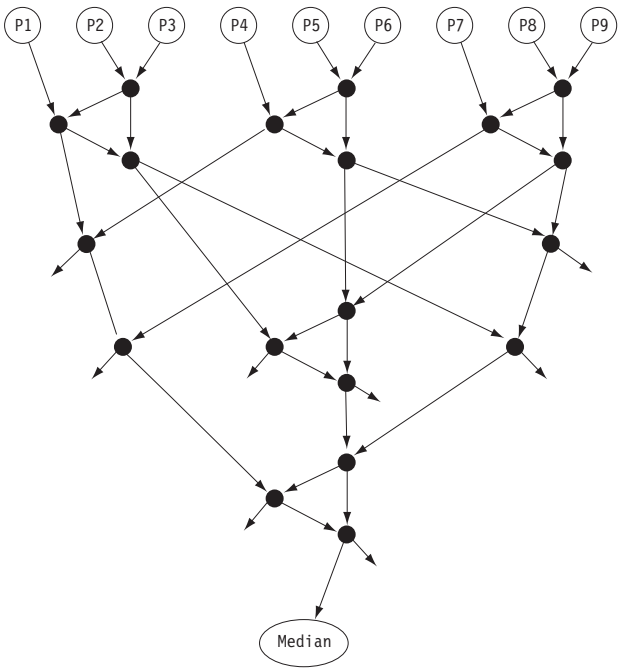



*Figure 5.2.1: fast median network algorithm*

```
// Listing 5.2.a
// Fast Median filter

%ffp

Category: "DIFP 5"
Title: "Fast Median filter"

ForEveryTile:
{
        //Loop through image without first and last row/column
        for (y=1; y<Y-1; y++)
        {
                for (x=1; x<X-1; x++)
                for (z=0; z<3; z++)
                {
                        // Put the values of the 3 x 3 matrix into cells 1 - 9
                        put(src(x-1,y-1,z),1);
                        put(src(x  ,y-1,z),2);
                        put(src(x+1,y-1,z),3);
                        put(src(x-1,y,z),4);
                        put(src(x  ,y,z),5);
                        put(src(x+1,y,z),6);
                        put(src(x-1,y+1,z),7);
                        put(src(x  ,y+1,z),8);
                        put(src(x+1,y+1,z),9);

                // Sort the cell values with the high-speed sort method by John L. Smith
                // http://www.eso.org/~ndevilla/median/
                        put(max(max(min(get(1),min(get(2),get(3))),
                        min(get(4),min(get(5),get(6)))),min(get(7),min(get(8),get(9)))),10);

                        put(min(min(max(max(get(8),get(9)),max(get(7),min(get(8),get(9)))),
                        max(max(get(5),get(6)),max(get(4),min(get(5),get(6))))),
                        max(max(get(2),get(3)),max(get(1),min(get(2),get(3))))),11);
```

```
                        put(min(max(min(max(get(8),get(9)),max(get(7),min(get(8),get(9)))),
                        min( min(max(get(5),get(6)),max(get(4),min(get(5),get(6)))),
                        min(max(get(2),get(3)),max(get(1),min(get(2),get(3)))) )),
                        max( min(max(get(5),get(6)),max(get(4),min(get(5),get(6)))),
                        min(max(get(2),get(3)),max(get(1),min(get(2),get(3)))) )),12);

                // put middle value into center pixel
                pset(x,y,z,min(max(get(10),min(get(11),get(12))),max(get(11),get(12))));
                }
                updateProgress(y,Y);
        }
return true;
}
```

Figure 5.2.2 shows the result after calling the median filter. One can see that the noise reduction is quite effective, but a certain blurring effect also comes into effect.

*Figure 5.2.2: fast median network algorithm example; original image (left) and filtered image (right)*

## Morphological Operators (Dilation and Erosion)

Mathematical Morphology is a field that introduces image processing operators used to analyze binary images and their tasks are to remove noise, enhance or segment images and edge detection, just to name a few. The most used morphological operators are dilation and erosion. A special technique is the combination of both operators to the techniques *opening* (which first erodes and then dilates the image) and *closing* (which first dilates and then erodes the image). As presented in figure 5.1.2, there are various structure elements forms which are used to extract the pixel values relative to a kernel pixel. Like all of the operators and filters described in this section, the structure element is passed along all the pixels in an image, in most cases from top to bottom and left to right.

### Dilation

Dilation takes the maximum pixel value in the structure element and colors the kernel with it. In general, objects are of light color while the background is of dark color. After dilation, objects (especially in bilevel images) seem to grow in size. Grayscale and color images tend to turn brighter than before. [HIPR] describes that dilation is also used as a basis for edge detection or region filling.

Dilating the left image in figure 5.2.2 results into dark noise removal, but bright noise amplification (see left image in figure 5.2.3).

Edge-detection on bilevel images with dilation is done by dilating the image and computing a difference image between the dilated and the original image. The result is an outline, being one pixel far from the original border.

### Erosion

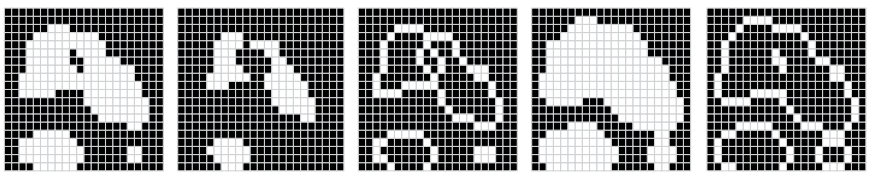
Erosion takes the minimum pixel value in the structure element and colors the kernel with it. After erosion, objects (especially in bilevel images) seem to shrink. Grayscale and color images tend to turn darker than before. [HIPR] describes that erosion is used as a preprocessing step for object labelling (counting white objects in an image – see also chapter 5.4 The Hoshen-Kopelman Algorithm).

Eroding the left image in figure 5.2.2 results into bright noise removal, but dark noise amplification (see right image in figure 5.2.3).

Edge-detection with erosion in bilevel images is done by eroding the image and computing the difference between the eroded and the original image. The result is an outline, where the outline pixels' positions coincide with the border pixels' positions of the original object.

*Figure 5.2.3: Left image in figure 5.2.2 was dilated (left) and eroded (right)*

Figure 5.2.4 shows the application of erosion and dilation with the corresponding edge finding routines on a bilevel image (leftmost image).



*Original image* *Erosion* *Erosion and difference* *Dilation* *Dilation and difference*

*Figure 5.2.4: Demonstration of erosion and dilation and edge-finding routines on original (leftmost image)*

Program listing 5.2.b lets the user erode and/or dilate the current image. Note that the four-neighbor structure element as shown in figure 5.1.2 is used and the results are shown in figure 5.2.4.

```
// Listing 5.2.b
// Erosion/Dilation example

%ffp

Category: "DIFP 5"
Title: "Erode or Dilate image"

ctl[0]: CHECKBOX, "", Pos=(220,20)
ctl[1]: STATICTEXT, "Dilate", Pos=(233,20)
ctl[2]: STATICTEXT, "Erode", Pos=(255,20)
ctl[3]: CHECKBOX, "Edge detect", Pos=(220,40)
```

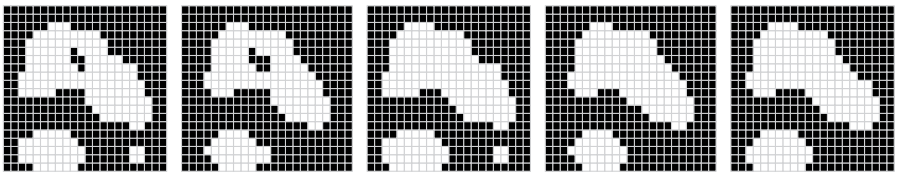
```
ForEveryTile:
{
if (ctl(0))     // Dilate
        {
        setCtlFontColor(1, COLOR(Blue));
        setCtlFontColor(2, COLOR(DarkGray));
        }
else            // Erode
        {
        setCtlFontColor(1, COLOR(LightGray));
        setCtlFontColor(2, COLOR(Blue));
        }
return false;
}

R,G,B:
!ctl(3) ?       // No edge detection?
   ctl(0) ?     // Dilate image
   max(max(max(src(x-1,y,z), src(x+1,y,z)), max(src(x,y-1,z), src(x,y+1,z))), src(x,y,z))
   :            // Erode image
   min(min(min(src(x-1,y,z), src(x+1,y,z)), min(src(x,y-1,z), src(x,y+1,z))), src(x,y,z))
:
   ctl(0) ?     // Dilate image and find edges
   dif(max(max(max(src(x-1,y,z), src(x+1,y,z)), max(src(x,y-1,z), src(x,y+1,z))), src(x,y,z))
   ,src(x,y,z))
   :            // Erode image and find edges
   dif(min(min(min(src(x-1,y,z), src(x+1,y,z)), min(src(x,y-1,z), src(x,y+1,z))), src(x,y,z))
   ,src(x,y,z))
```

### Opening and Closing

A simple application of an erosion or dilation operator is relative destructive, so they are applied in sequence. Depending on which operator is applied first, the sequential applications receive two different names: opening and closing.

*Opening* an image defines the erosion followed by a dilation. The *closing* of an image is defined by first dilating and then eroding an image. It is important to use the same structuring element for both operations. The effect can be increased by applying multiple erosions followed by the same amount of dilations (or vice-versa).

Figure 5.2.5 demonstrates various combinations of combined morphological operations. The leftmost image (original) is opened (second image shows the deletion of small white elements), closed (center image shows the deletion of small dark elements), opened & closed (fourth image) and closed & opened (last image). The last two images in figure 5.2.5 shows that the small white and dark elements are deleted.



*Original image* *Opening* *Closing* *Open & Close* *Close & Open*

*Figure 5.2.5: Demonstration of opening, closing, open & close and close & open on original image (left)*

Listing 5.2.c demonstrates the usage of opening and closing with a four-neighbor element. The results are shown in figure 5.2.5.

```
// Listing 5.2.c
// Opening/Closing example

%ffp

Category: "DIFP 5"
Title: "Open or Close image"

ctl[0]: CHECKBOX, "", Pos=(220,20)
ctl[1]: STATICTEXT, "Closing", Pos=(233,20)
ctl[2]: STATICTEXT, "Opening", Pos=(262,20)

ForEveryTile:
{
if (ctl(0))  // Closing
{
setCtlFontColor(1, COLOR(Blue));        // Highlight user control text "Closing"
setCtlFontColor(2, COLOR(DarkGray));

for (y=0; y<Y; y++)
for (x=0; x<X; x++)
for (z=0; z<Z; z++)
        tset(x,y,z,     // First dilate source image to tile buffer...
        max(max(max(src(x-1,y,z),src(x,y-1,z)),max(src(x+1,y,z),src(x,y+1,z))),src(x,y,z)) );

for (y=0; y<Y; y++)
for (x=0; x<X; x++)
for (z=0; z<Z; z++)     // ...then erode tile buffer to destination image
        pset(x,y,z,min(min(min(tget(x-1,y,z),tget(x,y-1,z)),min(tget(x+1,y,z),tget(x,y+1,z))),
        tget(x,y,z)));
}

else  // Opening

{
setCtlFontColor(1, COLOR(LightGray));
setCtlFontColor(2, COLOR(Blue));        // Highlight user control text "Opening"

for (y=0; y<Y; y++)
for (x=0; x<X; x++)
for (z=0; z<Z; z++)
        tset(x,y,z,     // First erode source image to tile buffer...
        min(min(min(src(x-1,y,z),src(x,y-1,z)),min(src(x+1,y,z),src(x,y+1,z))),src(x,y,z)) );

for (y=0; y<Y; y++)
for (x=0; x<X; x++)
for (z=0; z<Z; z++)
        pset(x,y,z,     // ...then dilate tile buffer to destination image
        max(max(tget(x-1,y,z),tget(x,y-1,z)),max(tget(x+1,y,z),tget(x,y+1,z)),tget(x,y,z)) );
}
return true;
}
```

## 5.3 Gradient Operators

Gradient operators are mostly used for edge-detecting techniques. As stated in 5.2 and shown in figure 5.2.4, morphological operators are also suitable for edge detection. The main difference between morphological and gradient operators is the image mode they are applied to. Morphological operators can detect edges in bilevel images best and often use structuring elements with different forms, while gradient operators detect edges on grayscale or color images and often use 3×3 convolution matrixes. The elements of the convolution matrix always sum up to 0.

An edge is defined as a sudden and steep value change of neighboring pixels. For example, a light object is placed onto a dark background (or vice-versa). A sudden and steep value change would mean a jump from one pixel with the value 5 (almost black) to 199 (very light), thus defining an edge.

The easiest edge finding technique is the usage of difference filters. The current pixel is set with the difference between the left and right (or top and bottom) pixel neighbor values. Such a filter would look like the following:

```
R,G,B:  -src(x-1,y,z) + src(x+1,y,z)    // left-to-right difference operator
A:      a
```

A convolution filter describing the above effect looks as follows:

```
R,G,B:  cnv(0,0,0, -1,0,1, 0,0,0, 1)    // left-to-right convolution difference operator
A:      a
```

The same problem with the image borders is found here. FilterMeister automatically clamps source pixel coordinates to the minimum (0) and maximum (X-1 or Y-1) values.

The Sobel operator, as deomnstrated in [Rei97], uses a combination of a difference and a blur filter (see chapter 5.1). While the convolution filter smooths some minor edges by blurring in horizontal direction, the vertical difference is then computed:

```
R,G,B:  cnv(1,2,1, 0,0,0, -1,-2,-1, 1)  // Sobel operator (North)
A:      a
```

The terms left-to-right and top-to-bottom (also used as East-to-West or North-to-South) describe a gradient with a certain direction. In order to detect all edges in all directions, four gradient operations have to be applied and combined. It is only necessary to create a filter that detects edges in any direction, in [Jähn97], page 292, described as isotropic edge detectors.

In [Hab89], page 138, the Laplace operator is presented as an isotropic edge detector for images with low noise. Noisy images result in a noise enhancement by the Laplace filter. The following programs present different variations of the Laplace operator:

```
R,G,B:  cnv(0,-1,0, -1,4,-1, 0,-1,0, 1)        // Laplace operator 1 (four-neighbor)
A:      a

R,G,B:  cnv(-1,-1,-1, -1,8,-1, -1,-1,-1, 1)    // Laplace operator 2 (eight-neighbor)
A:      a

R,G,B:  cnv(1,-2,1, -2,4,-2, 1,-2,1, 1)        // Laplace operator 3 with slight blur
A:      a
```

Applying the above Laplacian operators results to the images shown in figure 5.2.6. One can see that the four-neighbor result is acceptable, the eight-neighbor result is very good while the blurring eight-neighbor operator is rather poor to display edges.

## 5.4 The Hoshen-Kopelman Algorithm

The following chapter describes the object counting technique in bilevel images with the Hoshen-Kopelman algorithm (HKA). The objective is to effectively count the number of objects by giving them individual labels. The basic idea is best explained in [Rie97] with examples. That way, statistics dealing with object size and object amount can be done easily.

The HKA implemented here only works with bilevel images, so there are certain preprocessing steps that have to be done before running the object counting plug-in. One possible workflow would be:

1. Scan or photograph the image
2. Adjust brightness and contrast
3. Remove noise with median filter, for example
4. Convert the color or grayscale image to a bilevel image, where the objects are set to white and the background is set to black (invert the image, if necessary)
5. Apply any image correction/enhancement techniques (remove noise and/or segment objects with morphological operators)
6. Run the HKA plug-in to count the objects

The filter scans the image from top to bottom and from left to right. White pixels are labelled with a number, black pixels are ignored. A white pixel is labelled according to the following rules:

1. If the current pixel has no left and top white pixel neighbors, it gets a new label.
2. If the current pixel has a left *or* top white pixel neighbor, the current pixel gets the label of that neighbor.
3. If the current pixel has a left *and* a top white pixel neighbors, the current pixel gets the label of the neighbor with the lower value.
4. A diagonal neighbor is not part of the current object (four-neighbor theory applies here)

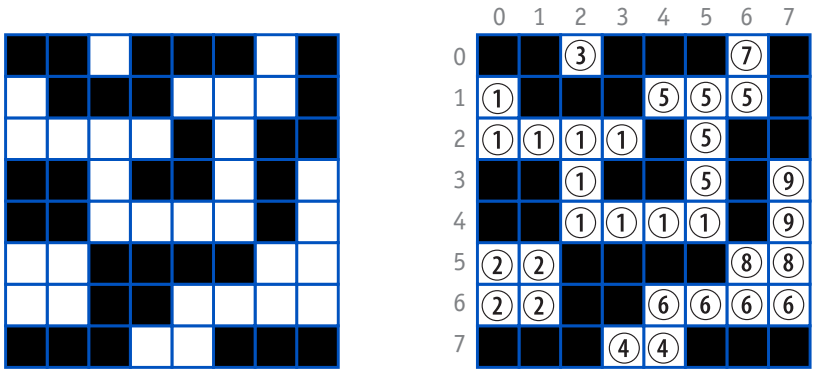


*Figure 5.4.1: Original image (left) is labelled (right) according to the HKA*

Figure 5.4.1 demonstrates the labelling of the objects. The first white pixel is encountered at (0,1) and is correctly labelled as to rule 1 (no neighbors means a new label). The bottom pixel at (0,2) is labelled as to rule 2 (neighbor at the top means to take his label). The following pixel at (0,5) is labelled again as to rule 1 and so on. Rule 3 (two neighbors?) is applied to the pixel at (4,7): it has a top neighbor with the label 6 and a left neighbor with the label 4. According to rule 3 it takes the label with the lower value. The dilemma now is that since the three pixels are interconnected and thus belong to one object, the top pixel at (4,6) needs to be relabelled. The image must be rescanned and the pixels relabelled until there are no conflicts.

The next step of the Hoshen-Kopelmann algorithm is to differ between correct and wrong labels. The wrong labels have to be relabelled. Hoshen and Kopelman introduce besides the label matrix the label-of-label (LOL) array. These LOLs can be seen as flags which state if the label is correct or has to be relabelled because of some conflict between two labels. A positive LOL defines that the label is correctly set and its number specifies the total amount of pixels having that label number. A negative LOL represents a wrong label and its absolute value points to the correct label number.

The first problem is encountered at pixel (4,7) as shown in the left image in figure 5.4.2. These are the current values of the LOLs:

LOL(1) = 9 LOL(2)=4 LOL(3)=1 LOL(4)=1 LOL(5)=1 LOL(6)=1

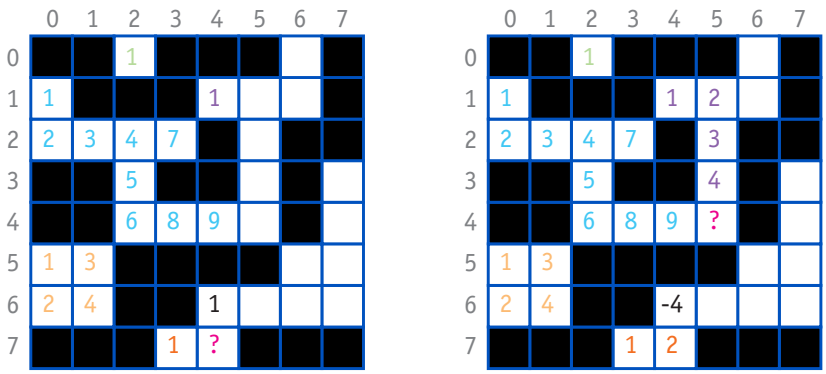


*Figure 5.4.2: Two label-of-label matrixes at two different stages; different colors represent different label-of-label indices*

The pixel to the left has the label 4 and the top pixel has the label 6, both having one pixel size at this time. The current pixel is added to the label with the lower value, in this case 4 (the LOL(4) is incremented

once). Since the top pixel belongs to the object, LOL(4) is incremented again and LOL(6) is negated and pointed to 4. After this whole operation, the LOLs look like the following:

LOL(1)=9 LOL(2)=4 LOL(3)=1 LOL(4)=3 LOL(5)=1 LOL(6)=-4

The next problem pixel is encountered at (5,4) as shown in the right image in figure 5.4.2. The values of the LOLs are now:

LOL(1)=9 LOL(2)=4 LOL(3)=1 LOL(4)=3 LOL(5)=4 LOL(6)=-4

The same procedure is used here. The current pixel takes the value of the lower label, and LOL(1) is incremented by 1. Since object 5 really belongs to object 1, its LOL value is added to LOL(1) and LOL(5) is negated and pointed to 1. After this operation, the LOLs look as follows:

LOL(1)=14 LOL(2)=4 LOL(3)=1 LOL(4)=3 LOL(5)=-1 LOL(6)=-4

When the algorithms reaches the last pixel in the image, the LOLs look as follows:

LOL(1)=16 LOL(2)=4 LOL(3)=1 LOL(4)=10 LOL(5)=-1 LOL(6)=-4
LOL(7)=-1 LOL(8)=-4 LOL(9)=-4

It is now easy to read the values of the LOL and count the amount of positive numbers, in the above case the algorithm returns a total amount of four objects. It is possible to continue with statistical computations such as average size of the objects, etc.

In conclusion, the Hoshen-Kopelman algorithm is an ingenious way to count objects in an image in one filter pass only. It also saves the size in pixels of the objects as well.

The HKA can be written as a FilterMeister plug-in and is shown in Listing 5.4. The plug-in has several limitations, though, which can be solved when FilterMeister offers the definition of arrays.

The two-dimensional label matrix is saved in the first channel of the tile buffer. At the end of the program, its contents are copied to the image, just to present the labels to those who want to understand the HKA at work. The first limitation here is that a tile buffer cell (which is actually a ”hidden” pixel) can only hold one byte of information (maximum label number is thus 255). Large images, a large amount of objects or even objects with irregular size can stop the plug-in from working correctly.

The one-dimensional LOL array is done with the memory cells, since these can save an integer between $-2^{31}$ and $2^{31}$-1. The limitation here is that the maximum amount of LOLs is set to 256. As said above, as soon as arrays are present in FilterMeister, this should be no longer a problem.

Listing 5.4 is divided into three parts. The first loop applies the HKA to the image. The second loop simply copies the data from the tile buffer to the destination image. The third and last part summarizes the image and object statistics.

The first loop with the HKA first tests if the current pixel is set to white. If it is, four different label and LOL definitions are possible:

1. Left and top pixel are direct neighbors to the current pixel
2. Left pixel is a direct neighbor to the current pixel
3. Top pixel is a direct neighbor to the current pixel
4. No direct neighbors present (new label)

The most extensive programming is needed for procedure 1 (left and top pixels are direct neighbors). The left and top pixels have to be tested in their label values (equal labels or different labels?) and the LOL also have to be tested in their values (are both LOLs equal or different, is one or both of them negative, etc.).

Figure 4.3.2 (file AMOEBA.TIF) was taken to test the correctness of the object counting algorithm. The image was created in Photoshop and a low contrast was simulated. Although not needed, the Auto-Level filter (shown in 4.3.1) was applied to increase contrast. Then, the Channel bilevelling filter (shown in 4.3.2) was used to create a bilevel image. Finally, the image inversion filter (shown in 4.3.1) was applied to invert the image, so the objects are white and the background is black (see figure 5.4.3).

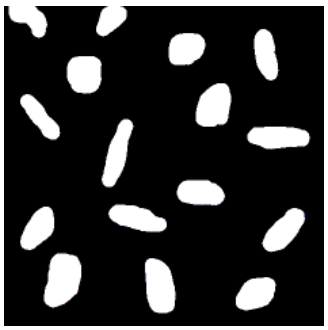


*Figure 5.4.3: The "digital amoebas" after several preprocessing steps described in the text*

The Object counter plug-in returned the following textfile `HKA.TXT`:

```
Object  1 has a size of  1641 pixels.
Object  2 has a size of  1783 pixels.
Object  3 has a size of  2079 pixels.
Object  4 has a size of  3340 pixels.
Object  5 has a size of  2419 pixels.
Object  6 has a size of  1392 pixels.
Object  7 has a size of  2429 pixels.
Object  8 has a size of  2345 pixels.
Object  9 has a size of  2922 pixels.
Object 10 has a size of  2103 pixels.
Object 11 has a size of  2159 pixels.
Object 12 has a size of  2643 pixels.
Object 13 has a size of  2166 pixels.
Object 14 has a size of  2489 pixels.
Object 15 has a size of  1971 pixels.
Object 16 has a size of  2303 pixels.

Totals 16 objects.
Average size: 2262 pixels
```

```
// Listing 5.4
// Object counting with the Hoshen-Kopelman algorithm

%ffp

Category: "DIFP 5"
Title: "Object counter"

ctl[0]: CHECKBOX, "Create text file"

ForEveryTile:
{
int iLabel=0, iDone=0, iCounter=0, iAverage=0;
int FILE;

for (x=0; x<X; x++)
for (y=0; y<Y; y++)
if (src(x,y,0)==255)
{
        iDone=0;
        if (tget(x-1,y,0) && tget(x,y-1,0))  // left pixel & top pixel are part of an object
        {
                if (tget(x-1,y,0)==tget(x,y-1,0))       // same labels
                {
                        tset(x,y,0,tget(x-1,y,0));
                        if (get(tget(x-1,y,0))>0)               // l_of_l positive
                        {       put(get(tget(x,y,0))+1,tget(x,y,0));
                                iDone=1;
                        }
                        if (get(tget(x-1,y,0))<0 && !iDone)     // l_of_l negative
                        {       put(get(abs(get(tget(x,y,0))))+1, abs(get(tget(x,y,0))) );
                                iDone=1;
                        }
                } // end same labels

                if (tget(x-1,y,0) != tget(x,y-1,0) && !iDone)  // different labels
                {
                        tset(x,y,0, min(tget(x-1,y,0),tget(x,y-1,0)));
                        // l_of_l positive (==,!=)
                        if (get(tget(x-1,y,0))>0 && get(tget(x,y-1,0))>0 && !iDone)
                        {       put(get(tget(x,y,0))+1+get(max(tget(x-1,y,0),
                                tget(x,y-1,0))),tget(x,y,0));
                                put(-tget(x,y,0), max(tget(x-1,y,0), tget(x,y-1,0)) );
                                iDone=1;
                        }
                        // 1 l_of_l pos, 1 neg, left > top
                        if (get(tget(x-1,y,0))>0 && get(tget(x,y-1,0))<0 && !iDone)
                        {       if (tget(x-1,y,0)<tget(x,y-1,0))
                                {       if (abs(get(tget(x,y-1,0)))==tget(x-1,y,0))
                                        {       put(get(tget(x,y,0))+1,tget(x,y,0));
                                                iDone=1;
                                        }
                                        else
                                        {       put(get(tget(x,y,0))+1+get(abs(get(max(tget(
                                                x-1,y,0), tget(x,y-1,0))))), tget(x,y,0));
                                                put(-tget(x,y,0), max(tget(x-1,y,0),
                                                tget(x,y-1,0)) );
                                                put(-tget(x,y,0), abs(get(max(tget(x-1,y,0),
```

```
                        tget(x,y-1,0)))) );
                        iDone=1;
                }
        }
        else
        {       if (abs(get(tget(x,y-1,0)))==tget(x-1,y,0))
                {       put(get(tget(x-1,y,0))+1,tget(x-1,y,0));
                        iDone=1;
                }
                else
                {       put(get(abs(get(tget(x,y-1,0))))+1+get(tget(
                        x-1,y,0)),abs(get(tget(x,y-1,0))));
                        put(get(tget(x,y,0)),tget(x-1,y,0));
                        iDone=1;
                }
        }
}
// 1 l_of_l pos, 1 neg, left < top
if (get(tget(x-1,y,0))<0 && get(tget(x,y-1,0))>0 && !iDone)
{       if (tget(x-1,y,0)<tget(x,y-1,0))
        {       if (abs(get(tget(x-1,y,0)))==tget(x,y-1,0))
                { put(abs(get(tget(x,y,0)))+1,abs(get(tget(x,y,0))));
                        iDone=1;
                }
                else
                {put(get(tget(x,y-1,0))+1+get(abs(get(tget(x,y,0)))),
                        abs(get(tget(x,y,0))) );
                        put(get(tget(x,y,0)), tget(x,y-1,0) );
                        iDone=1;
                }
        }
        else
        {       if (abs(get(tget(x-1,y,0)))==tget(x,y-1,0))
                {       put(get(tget(x,y,0))+1,tget(x,y,0));
                        iDone=1;
                }
                else
                {       put(get(abs(get(tget(x-1,y,0))))+1+
                        get(tget(x,y-1,0)),abs(get(tget(x-1,y,0))));
                        put(get(tget(x-1,y,0)),tget(x,y,0));
                        iDone=1;
                }
        }
}

// both l_of_l equal and negative
if (get(tget(x-1,y,0))<0 && get(tget(x,y-1,0))<0 &&
    get(tget(x-1,y,0))==get(tget(x,y-1,0)) && !iDone )
{       put(get(abs(get(tget(x,y,0))))+1, abs(get(tget(x,y,0))) );
        iDone=1;
}
// l_of_l different and negative
if (get(tget(x-1,y,0))<0 && get(tget(x,y-1,0))<0 &&
    get(tget(x-1,y,0))!=get(tget(x,y-1,0)) && !iDone)
{ if (get(max(tget(x-1,y,0),tget(x,y-1,0)))==min(get(
        tget(x-1,y,0)),get(tget(x,y-1,0))))
        {
        put(get(abs(get(tget(x,y,0))))+1+get(abs(get(max(tget(x-
        1,y,0), tget(x,y-1,0))))), abs(get(tget(x,y,0))));
```

```
                                put(get(tget(x,y,0)), max(tget(x-1,y,0), tget(x,y-1,0)));
                                put(get(tget(x,y,0)), abs(get(max(tget(x-1,y,0),
                                tget(x,y-1,0)))) );
                                        iDone=1;
                                }
                                else
                                { put(get(abs(get(tget(x,y,0))))+1+get(abs(get(max(tget(x-
                1,y,0), tget(x,y-1,0))))),abs(max(get(tget(x-1,y,0)),get(tget(x,y-1,0)))));
                                put(max(get(tget(x-1,y,0)),get(tget(x,y-1,0))),tget(x,y,0));
                                put(max(get(tget(x-1,y,0)),get(tget(x,y-1,0))),
                                abs(min(get(tget(x-1,y,0)),get(tget(x,y-1,0)))));
                                        iDone=1;
                                }
                        }
                } //end different labels
        }
        if (tget(x-1,y,0) && !iDone)    // left pixel is part of an object
        {
                tset(x,y,0, tget(x-1,y,0));
                if (get(tget(x,y,0))<0)
                        put(get(abs(get(tget(x,y,0))))+1,abs(get(tget(x,y,0))));
                else
                        put(get(tget(x,y,0))+1, tget(x,y,0) );
                iDone=1;
        }

        if (tget(x,y-1,0) && !iDone)    // top pixel is part of an object
        {
                tset(x,y,0, tget(x,y-1,0));
                if (get(tget(x,y,0))<0)
                        put(get(abs(get(tget(x,y,0))))+1,abs(get(tget(x,y,0))));
                else
                        put( get(tget(x,y,0))+1, tget(x,y,0) );
                iDone=1;
        }

        if (!iDone)  // new label
        {
                iLabel++;
                tset(x,y,0, iLabel);
                put(1, iLabel);
        }
} //end if_src==255, loops here

for (x=0; x<X; x++)
for (y=0; y<Y; y++)
        pset(x,y,0, tget(x,y,0)); // simply copies data from tile buffer to destination image

iCounter=0;

if (!ctl(0))    // write to file?
{       for (i=0; i<256; i++)
        if (get(i)>0)  // cell positive?
        {       iCounter++;
                tset(iCounter%X, iCounter/X, 1, tget(iCounter%X, iCounter/X, 1)+1);
                iAverage += get(i);
        }
Info("Total of %d objects.\nAverage size: %.f", iCounter, (double) iAverage/ (double)
iCounter);
```

```
}
else
{
// create file and write statistics
if (!doingProxy && (FILE=fopen("C:\\hka.txt", "w")))
        {
                for (i=0; i<256; i++)
                        if (get(i)>0)  // cell positive?
                        {
                        iCounter++;
                        tset(iCounter%X, iCounter/X, 1, tget(iCounter%X, iCounter/X, 1)+1);
                        iAverage += get(i);
                        fprintf(FILE, "Object %2d has a size of %5d pixels.\n", iCounter,
                        get(i));
                        }
                fprintf(FILE, "\nTotals %d objects.\nAverage size: %.f pixels", iCounter,
                (double) iAverage/ (double) iCounter);

                if (fclose(FILE)) ErrorOk("File could not be closed!");
        }
}
return true;
}
```

# 6. Plug-In Artistry

The nature of FilterMeister respectively plug-ins for image editing or painting programs in general is, among other things, to:

- modify image colors (e.g., color correction),
- modify pixel positions (e.g., auto-alignment, mirroring or other transformation of an image),
- digital image synthesis (e.g., find edges, morphological operations, preprocessing operations),
- work as utilities for internet-based tasks (e.g., seamless tile creation or GIF animation)
- compute statistical data from images (e.g., histograms and other preprocessing operations), and
- create artistic digital images

This chapter deals with the last subject: the creation of digital images in artistic form.

## 6.1 Color effects

The following effect is based on the idea of projecting the pixel value range to a sine curve. The user has the option to change the wavelength, if desired. Figure 6.1 shows the application of the effect to normal images and to blurred text. The results look like a pop art effect. While the color distortion of the image is not very intruguing, the blurred text's effect is much more interesting for graphic designers.

```
// Listing 6.1
// Pop Art effect

%ffp

Category: "DIFP 6"
Title:    "Pop Art"

ctl[0]: "Amount", Range=(-30,30)

R,G,B:  scl(sin(c*ctl(0)),-511,512,0,255)
A:      a
```


*Figure 6.1: The pop art effect is visualized on an image (left) and on a blurred text (right)*

## 6.2 Image mosaic

Throughout the years in modern art, there have been several attempts to display images by different types of mosaics. Well-known mosaic types are the typewriter or ASCII-mosaics (where the characters symbolize a different density and certain directions for better curve anti-aliasing). Or the image is represented by smaller images which are actually downscaled and recolored copies of the image in itself. Another attempt are the photomosaics (www.photomosaic.com) where an image is composed of much smaller images taken from a database, where the colors and structure of the small images basically coincide with the color and structure of part of the larger image.

The following program Image by Numbers reads the contents of an ASCII-text file and colors each individual letter according to the current color in the image. Before applying the plug-in, the image's height has to be resized to 70%, because of the font's proportions. Figure 6.2 shows an image being described by the greek number π, whose figures are colored according to the original image. The result is a PostScript file which needs to be printed or converted to PDF for viewing purposes. The plug-in offers the user the setting of points (1/72 inch), centimeters or inches and tells the user the final metrics of the image. The user control labelled with Color area describes the size of the mosaic, i.e., how many pixel colors are represented by a figure in the final image.

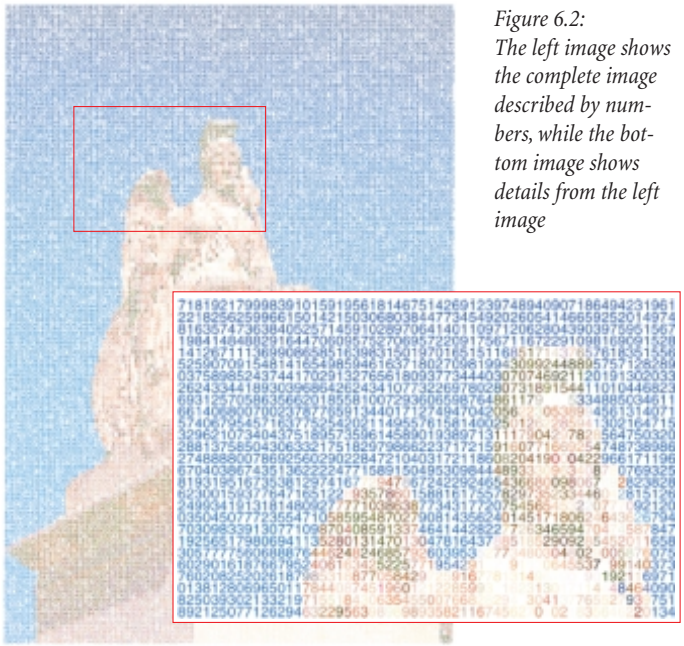


*Figure 6.2: The left image shows the complete image described by numbers, while the bottom image shows details from the left image*

```
// Listing 6.2
// Create image mosaic using a text file

%ffp

Category: "DIFP 6"
Title:    "Image by Numbers"

ctl[0]: "Color area", range=(1,50)
ctl[1]: COMBOBOX, "point\ncm\ninch", Val=1, Size=(40,40), Action=PREVIEW
ctl[3]: STATICTEXT, Size=(100,50)

ForEveryTile:
{
int PS_FILE, INP_FILE;
u = X*scaleFactor;
v = Y*scaleFactor;

switch(ctl(1))
{
```

```
case 0: setCtlTextv(3, "The character color is composed by %d × %d pixels.\n"
"The final output size is :\n%d × %d points", ctl(0), ctl(0), u*7/ctl(0)+30,
v*10/ctl(0)+30);
        break;

case 1: setCtlTextv(3, "The character color is composed by %d × %d pixels.\n"
"The final output size is :\n%.1f cm × %.1f cm", ctl(0), ctl(0),
(u*7.0/(ctl(0))+30.0)/28.3464567, (v*10.0/(ctl(0))+30.0)/28.3464567);
        break;

case 2: setCtlTextv(3, "The character color is composed by %d × %d pixels.\n"
"The final output size is :\n%.1f \" × %.1f \"", ctl(0), ctl(0), (u*7.0/(ctl(0))+30.0)/72.0,
(v*10.0/(ctl(0))+30.0)/72.0);
        break;
}

if (!doingProxy) // create file only when clicking on OK
{
        if ((PS_FILE=fopen("d:\\number1.ps", "w")) && (INP_FILE=fopen("d:\\pi.txt", "r")))
        {
                fprintf(PS_FILE, "%%Image by numbers\n\n/Helvetica findfont 12 scalefont "
                "setfont\n\n");
                for (y=0; y<=Y; y+=ctl(0))
                for (x=0; x<=X; x+=ctl(0))
                        fprintf(PS_FILE, "%0.2f %0.2f %0.2f setrgbcolor %d %d moveto (%c)"
                        " show\n", src(x,y,0)/255.0, src(x,y,1)/255.0, src(x,y,2)/255.0,"
                        " 10+x/ctl(0)*7, v*10/ctl(0)+10-y/ctl(0)*10, fgetc(INP_FILE));

                fprintf(PS_FILE, "\nshowpage");
                if (fclose(PS_FILE))    ErrorOk("Cannot close PostScript file.");
                if (fclose(INP_FILE))   ErrorOk("Cannot close text file.");
        }
        else
                ErrorOk("Cannot create PostScript file d:\\number1.ps (1) or load text file "
                "d:\\PI.TXT (2).\n\n(1) Disk may be full or write-protected.\n(2) File "
                "d:\\PI.TXT not found or corrupted.");
}
return false;
}
```

## 6.3 Wavy images

The following program displaces the pixels according to sine waves in both horizontal and vertical directions. The src()-function is used to locate the pixels and its coordinates are displaced by the sine value of the opposing axis, i.e., the x-coordinate is displaced by the sine value of the y-coordinate. The user has the ability to set the cycle or wavelength and amplitude of the sine waves. Small cycle values lead to a great image perturbance, while large cycle values lead to a "deep-ocean-wavy" image. Small amplitudes lead to subtle disturbances while large amplitudes produce large distortions in the image. Figure 6.3 shows an example of the "Sinus wave" plug-in.

*Figure 6.3: This image was perturbed by the "Sinus wave" plug-in in listing 6.3*

```
// Listing 6.3
// Displaces the pixels according to a sine wave in horizontal and vertical directions

%ffp

Category   :"Neology"
Title      :"Sinus waves"

ctl[0]:"Horizontal cycle", Val=128
ctl[1]:"Vertical cycle", Val=128
ctl[2]:"Horizontal amplitude", Val=20
ctl[3]:"Vertical amplitude", Val=20

R,G,B:  src(x+sin(y*1024/ctl(0))/(512/ctl(2)), y+sin(x*1024/ctl(1))/(512/ctl(3)),z)
A:      a
```

# 7. Conclusion

This paper was meant to provide the person who works with digital image processing tasks with information about the abilities and limitations of the plug-in compiler **FilterMeister** by AFH Systems Group.

FilterMeister offers the user a wide variety of commands for reading, processing and writing image information. Currently, FM is limited to process RGB images, although the full-working Beta version supports other image modes such as Grayscale, L*a*b*, Multiple Channels and CMYK and the creation of stand-alone plug-ins. The memory size for source code text is graciously large, so difficult or extensive plug-ins are compilable.

Although the FilterMeister language FF+ is a subset of the C programming language, some typical and important language elements are missing. The most important missing language element are arrays, which are necessary for co-occurence matrixes and other calculations. Although it is possible to use memory cells (as used for the histogram plug-in) and the two tile buffers (which each include three to four plane buffers, depending on the layer type), the amount of memory cells is limited to 256 cells and the tile buffer element can only save the information of a byte (0 – 255).

With FilterMeister, it is not possible to access pixel information outside the current layer or more than the standard color mode channel amount (which means that in an RGB-image, a fourth channel from the background layer cannot be accessed, although FilterMeister has access to all four channels in a CMYK image).

Unlike other commercial plug-ins, FilterMeister does not yet offer the ability of defining free definable cursor positions within the preview nor the ability to update the image in the background while the filter dialog is open. One also misses the ability to resize the image by FilterMeister.

The saving and loading of settings is limited to .INI-files instead to the Windows Registry. There is also currently no access to internal system data such as TrueType and/or PostScript fonts loaded by Windows or a Font Manager.

The limitations are not to stay – unlike Adobe's Filter Factory, AFH's FilterMeister is constantly being updated, so some of the above limitations may soon be a thing of the past. The foundation for digital filter and plug-in creation is present, as one can see in the chapters of this paper and the tutorials in the FilterMeister manuals *Getting Started* and *User Guide*.

To summarize the current capabilities of FilterMeister:

- Creation of standalone multi-pass filters/plug-ins
- Easy-to-understand programming language (superset of the Adobe's Filter Factory language and a subset of the programming language C)
- Typical image metrics are presented as variables
- Large variety of image-specific functions
- File input and output functions
- Memory cells and two tile buffers (as large as the image itself) for temporary image data
- User-configurable design of the plug-in dialog window
- Different user-definable user control types
- Support of integer and floating-point arithmetic
- Message window (in Microsoft Windows style) support
- Support of a large amount of color names
- Capable to be installed in various host programs (those who support plug-ins programmed according to the Adobe Plug-In Self-Development Kit (SDK))
- Runs under Microsoft Windows 95, Windows 98 and Windows NT operating systems

FilterMeister's main objective is to easen the burdeon of plug-in creation for Adobe Photoshop and Photoshop-Plug-in-compatible programs. The Adobe's SDK is not easy to understand and proficient knowledge of C respectively C++ is needed. The programmer should only think about the plug-in algorithm itself. In my opinion, FilterMeister will manage to stay as *the* plug-in compiling utility for digital image programs and will make the Adobe's SDK useless.

# 8. Appendix

## A. The CD-ROM

The CD-ROM in this paper includes the following directories:

```
Documentation
Source
Standalone
Images
```

The directory `Documentation` includes this paper and the FilterMeister manuals Getting Started (GS) and User Guide (UG) in the portable document format (PDF). It is recommended to read the GS and UG first. The newest versions of the GS and UG can be found at `http://www.filtermeister.com`.

The directory `Source` includes all programs in this paper in FilterMeister's source code (`.ffp`). The `README.TXT` lists all source files with a short description.

The directory `Standalone` includes all programs in this paper in the Photoshop filter format (`.8bf`), ready to be installed in the plug-in directory of the host program. The `README.TXT` lists the standalone files with a short description.

The directory `Images` includes the images found in this document in form of TIFF and EPS file formats.

## B. References

[APS4] Adobe Photoshop 4.0 manual, Adobe Systems, Inc.

[AFF] Adobe Filter Factory manual, Adobe Systems, Inc.

[Hab89] Haberäcker, Peter, "Digitale Bildverarbeitung – Grundlagen und Anwendung", 3rd Edition, '89

[HIPR96] Fisher, Perkins, et al., Hypermedia Image Processing Reference, Chapter **Image Processing Opreator Worksheets**, 1996
URL: `http://image.korea.ac.kr/HIPR/HTML/wksheets.htm`

[Reis96] Reiser, Prof. Dipl.-Ing., Lecture "Technologie Vorstufen – Grundlagen der Reproduktion", '96

[Reis97] Reiser, Prof. Dipl.-Ing., Lecture "Digitale Bildverarbeitung", 1997

[Rie97] Rieckmann, Christina, Dissertation "Lösung des Problems der Diffusion und Reaktion in dreidimensionalen Porennetzwerken für allgemeine Kinetiken", Chapter **Clusteranalyse mit dem Hoshen-Kopelman-Algorithmus**, 1997
URL: `http://pc50.vt4.tu-harburg.de/dissert/rieckmann/diss00.html`

[Smi96] Smith, John L., "Implementing Median Filters in XC4000E FPGAs", 1996
URL: `http://www.eso.org/~ndevilla/median/`

[Str99] Streidt, Werner D., "FilterMeister User Guide", PDF Manual, 1999

[Jähn97] Jähne, Bernd, "Digital Image Processing", 4th edition, 1997