FilterMeister

Book One - Getting Started

Book Two - User Guide

Book Three - Reference Manual

Table of Contents

1. Introduction

We welcome you to FilterMeister by AFH Systems Group! FilterMeister is a plug-in for Adobe Photoshop and plug-in compatible programs (such as Corel PhotoPaint, JASC Paint Shop Pro, Macromedia Freehand, etc.) with which you can create your own astounding filter effects such as fractals, explosions, 3D-effects, and lots more! *Note: Currently, FilterMeister is available only for the Windows 95/NT Operating Systems.*

The entire plug-in window is fully configurable to meet your needs. You can:
* use sliders, checkbuttons for user interaction,
* load background bitmap images,
* use radio buttons and pull-down menus to select predefined settings or multiple-filters in one plug-in,
* save and load image or settings data to and from disk,

and lots more!

In FilterMeister (short: FM), you will be working with the FF+ (FF plus) language, which is a superset of the Adobe Filter Factory (FF) language. FF+ is a subset of the C language, with extensions that include:

- Additional built-in variables and functions for image processing and filter design
- Additional syntax to describe the design of the User Interface
- Predefined event handlers, which are special purpose functions for processing specific filter events and User Interface actions

You won't need advanced programming skills, since we will be introducing the language to you bit by bit. With the FilterMeister package you received three manuals in the PDF file format:

```
GettingStarted.pdf
UserGuide.pdf
ReferenceManual.pdf
```

In this manual, we will introduce you to the basics of image representation in computers. You'll be learning how to install and run FM and create your first small filters. The User Guide will deepen your knowledge of Filter Programming with FM in the form of tutorials. We recommend that you read the manuals thoroughly and study the algorithms presented. The FM language is so powerful that we can't teach you all the possibilites, functions, variables, handlers, etc. This is where you can consult the Reference Manual to learn the elements not discussed in the first two manuals.

2. Images

In this chapter you will be presented with the basics of image representation in computers. If you are a pro or know what vector images, pixel images, channels, color spaces (or image modes) are, then skip this chapter and go ahead and install FilterMeister as described in Chapter 3.

2.1 Vector-based/Pixel-based images

There are two different kinds of graphic programs:

a) Drawing programs
b) Painting programs

Drawing and painting are not synonyms. Drawing programs are *vector-based* programs. Usually you use predefined objects such as lines, circles, rectangles, etc., and combine them on your worksheet (the worksheet's width and height are usually measured in inches or millimeters). You can assign each object an individual color, filling, thickness, etc. This means you can edit each object without losing information, even if the objects are 'hidden' behind others. The good thing about vector-based graphics is that you can scale them to any size you want without losing information as in pixel-based graphics; each object's data is a mathematical expression – scaling the picture only rearranges the mathematical formula. If you could zoom into these drawings to infinity, you would not notice any “jaggies,” also known as *aliasing*. Typical drawing programs are Corel DRAW!, Adobe Illustrator or Macromedia Freehand.

Painting programs are *pixel-based* programs. Images edited here are usually scanned pictures and retouched or painted with different tools. Tools are available to manipulate your image in any way you want. You can use brushes, selections, smudge, eraser, paintbucket, etc. Your image's width and height are always measured in pixels – the painting program helps you by telling you how wide/tall your image will be when printed. Each pixel contains intensity information and all pixels have the same width/height. Depending on your image the pixels contain grayscale or color information. Another possible piece of information is an alpha value, giving the pixel a transparency option. This is great when compositing two or more images into one (e.g., clipping a person from an image and putting him/her into another background). The difference from a vector-based picture is that once you paint over a pixel, the original pixel information is gone. Scaling a pixel-based image is problematic. Scaling down means that the image's pixel width is reduced, meaning that the new width is composed of fewer pixels. Some pixels get lost. Scaling up means that the image's width and height are increased. Since some pixels are newly created, their color information has to be computed. This process is called interpolation. Typical painting programs are Adobe Photoshop, JASC Paint Shop Pro and Fractal Design Painter. There are also some graphic

programs on the market which can combine vectors and pixels, but we won't be needing information about these.

Let's check the paint programs a bit more.

For example, a common image size has a width of 640 pixels and a height of 480 pixels. Normally you say the image is 640 × 480 pixels large. In order to be able to work with the pixels, you need their adresses. This is solved by a Cartesian coordinate system. The variable x will give you the horizontal position and the variable y will give you the vertical position of the pixel. The origin (0,0) of the coordinate system is on the top left of the image. The biggest values for x and y in our image would be (639,479). Did I say 639 and 479? Yes! This is because the origin is also a part of our image, meaning x can have a value from 0, 1, 2, 3, ..., 637, 638, 639 (a *total* of 640 pixels). Another common image size would be 800x600 and you can address all the pixels between (0,0) and (799,599), inclusive.

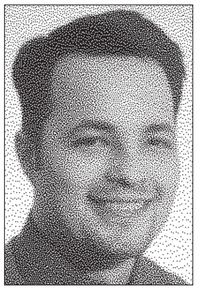
By the way, Photoshop (and several other programs) also works with vector-based objects. Yup, in case you never worked with them, I am talking about *paths*... (oh, those). Paths are dots (or anchors) which are connected by Bezier curves and they are ideal for clipping and some other effects.

2.2 Color Models *(Image Modes)*

When working with pixel-based images, we have the possibility of working 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.



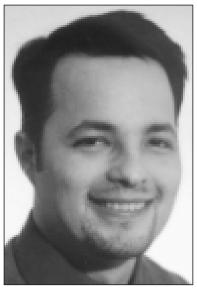
Memory requirements:
How large (in kilobytes) is a 640 × 480 bitmap?

640 × 480	= 307,200 pixels
307,200 ÷ 8 bits	= 38,400 bytes
38,400 ÷ 1024 bytes	= 37.5 KB

Grayscale images

In grayscale images, a pixel is described by one byte or 8 bits. You can combine or rearrange 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)



Memory requirements:
How large (in kilobytes) is a 640 x 480 Grayscale image?

640 × 480	= 307,200 pixels
307,200 × 8 bits	= 2,457,600 bits
2,457,600 ÷ 8 bits	= 307,200 bytes
307,200 ÷ 1024 bytes	= 300 KB

RGB images

Now let's add some color to our images. When you were a kid (or still are one) you probably checked out how colors are emitted from dad's TV or computer monitor. When you got really near you probably saw a matrix of three colored dots: Red, Green and Blue. With these three colors, you can create most of the colors the eye can see. These colors are actually colored light rays which have a certain frequency or wavelength. When mixed together, the frequencies are added together. This is the additive color mixture. Imagine the following image as if you were in a dark room and were projecting three colored lamps onto your wall. By the way, all scanners work by scanning the red, green and blue parts of a picture, photograph or slide.

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!

Memory requirements:
How large (in kilobytes) is a 640 x 480 RGB image?

640 × 480	= 307,200 pixels
307,200 × 8 bits	= 2,457,600 bits (each channel)
2,457,600 × 3 channels	= 7,372,800 bits
7,372,800 ÷ 8 bits	= 921,600 bytes
921,600 ÷ 1024 bytes	= 900 KB

How can gray be described in a RGB image? By simply using an equal number in the three channels, for example:

(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. Except for some special cases, the FM tutorial algorithms will be programmed for RGB-images.

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 separating images in 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. The wavelengths are subtracted from each other.

Therefore, when nothing is printed, you see the media's color, which will in most cases be 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, we have a total of 32 bits per CMYK pixel.

Memory requirements:
How large (in kilobytes) is a 640 x 480 CMYK image?

640 × 480	= 307,200 pixels
307,200 × 8 bits	= 2,457,600 bits (each channel)
2,457,600 × 4 channels	= 9,830,400 bits
9,830,400 ÷ 8 bits	= 1,228,800 bytes
1,228,800 ÷ 1024 bytes	= 1200 KB or 1.2 MB

L*a*b* Images

If you work with Color Management, then the L*a*b* color system 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*	red-green axis
b*	yellow-blue axis

This is somewhat difficult to understand and should not frighten you, because this mode won't be interesting for our objectives in the FM manuals.

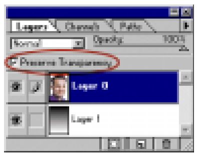
2.3 Layers - Channels - Selections

We will assume that you are working with 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 we are 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 we are 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 mean fully transparent and high alpha values indicate full opacity.

Transparency is used when you work with compositing techniques. For example, we have two images. In one image you see a couple in the Netherlands and in the second image you see a Hawaiian volcano. With a path, we can cut out the couple and paste these people 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, you have access to all channels depending on the *active* layer when the plug-in was called. Calling FM on a background layer won't let you change any alpha information of a pixel because a background image has no alpha channel. *With FM, you currently have access to only one layer at a time.*



A layer like the one of the couple can be in two states: editable and protected transparency. In the first case, you can change (edit) the transparent areas of an image. In the latter case, Photoshop won't let you paint outside of the opaque areas. You can check if you have one of these states by calling the **layers palette** in Photoshop and looking at the small checkbutton called "Preserve Transparency."

Filtering can be affected by another situation. If you have a selection, either rectangular, elliptical or free form (created with the Lasso or Text tool, for example), the filter will in most cases change the pixel information only within the selection and not outside. At the moment, FM only takes pixel information from the selection. The (selected) image area sent to the filter has a different size than the original image. In further lessons we will teach you how to modify your algorithm in a way that it will be independent of an image's size.

A filter can handle seven states:

1. Flat image with no selection mask (Background layer without selection)
2. Flat image with selection mask (Background layer with selection)
3. Floating selection
4. Editable Transparency with no selection (Layer with transparency and no selection)

5. Editable Transparency with selection (Layer with transparency and a selection)
6. Protected Transparency with no selection (Layer where transparent areas can't be edited; no selection)
7. Protected Transparency with selection (Layer with uneditable transparency and selection)

2.4 How does Filtering work?

Filters need a lot of memory in order to be able to process images. When the image is small enough, the filter 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 filter. This process is called *tiling*.

The tileability of a filter depends on the use of certain functions which have access to all pixels in an image (*for example when mirroring the image horizontally*). A tiling restricition is in such a case not desirable. In a future lesson we will show you how to override this feature and force the filter to tile anyway.

We have now come to the stage where the image is loaded and the pixels within it are processed by the filter. With FM, you can choose from different filter processing functions, called handlers. Some of them tell the filter to 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 other handlers give you more freedom; you can determine the filter processing direction as you need it. You don't even have to process all rows, columns, channels or pixels! You could, for example, tell the filter to first process the columns and then the rows (*good for windy effects*).

3. Installing FilterMeister

In this chapter, you will learn how to install the FilterMeister plug-in properly into your host graphics application directory and run a small filtering test. Finally, there are some Internet resources you should know of where you can get support and exchange ideas and knowledge.

3.1 Plug-In Installation

You have received an EXE-file that self-extracteds to a directory `FilterMeister` containing these files:

`AfhFM04b.8bf`	FilterMeister plug-in
`FmTemplates`	Directory containing several algorithm templates
`FmSamples`	Directory containing FM algorithms and plug-ins
`FmDocumentation`	Directory containing documentation in PDF-format
`Readme.txt`	First file you should have read

Please copy the whole FilterMeister directory to the following directories in accordance with your host program:

Adobe Photoshop 4+	`Plugins` or `Plug-Ins`
Adobe Illustrator 7.0.x	`Plug-Ins`
Macromedia Freehand 8.0.x	`Xtras`

When these programs are restarted, FilterMeister is recognized by the host program. If you intend to use FilterMeister with the following programs and/or want to test FM's behavior with different hosts without copying the FM directory more than once, then either leave the current FM directory at its place or move it to a more convenient location. All you now need is to go to the host program's plug-in preferences and reset them to the FM directory. Previously recognized plug-ins may no longer appear in the filter menu because the host program no longer searches the directory that contained them. Please consult your host program's manual for correct installation of Photoshop-compatible plug-ins.

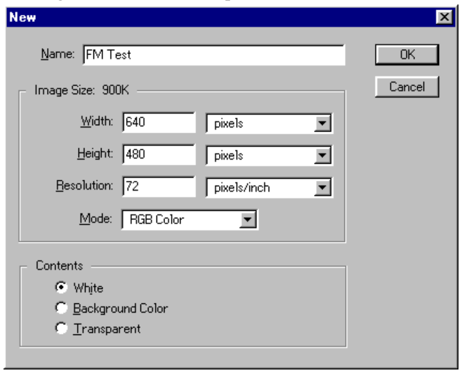
Macromedia FireWorks 2
JASC Paint Shop Pro 4+ can load up to three different directories with plug-ins
Ulead PhotoImpact 4
Corel PhotoPaint 8

3.2 Short Introduction

Let us test if our installation worked correctly. If you haven't restarted your host program after installation, please do so now. *Paint Shop Pro users don't need to restart the program.*

All filters are grayed out because there is no active image. Let's create a new RGB-image. In Photoshop and Paint Shop Pro, select `File` → `New...` creating a new 640 × 480 RGB image. In Adobe Illustrator, for example, you can create any object (or a group of objects) and convert it to a pixel image. You can then run the FilterMeister plug-in when this pixel object is activated.

Calling New... in Photoshop


New

Name: FM Test

Image Size: 900K

Width: 640 pixels

Height: 480 pixels

Resolution: 72 pixels/inch

Mode: RGB Color

Contents

[✓ White] Background Color Transparent

and in Paint Shop Pro

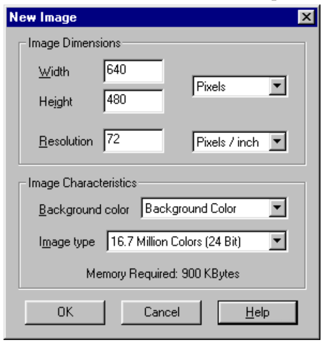

New Image

Image Dimensions

Width 640

Height 480 Pixels

Resolution 72 Pixels / inch

Image Characteristics

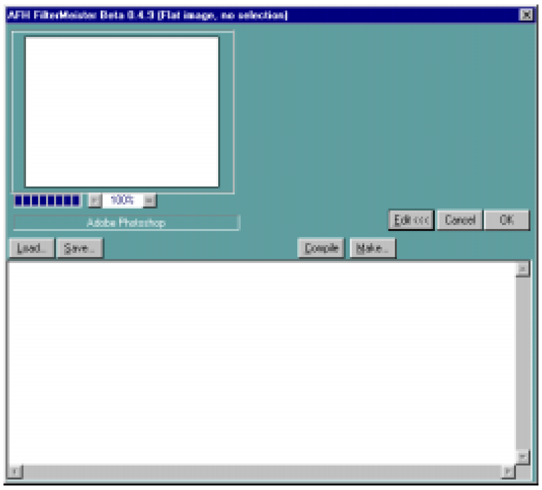
Background color: Background Color

Image type: 16.7 Million Colors (24 Bit)

Memory Required: 900 KBytes

In **Photoshop**, select the category `FilterMeister` from the `Filter`-menu; then select `FilterMeister 0.4.1b...`. In **Paint Shop Pro**, from the main `Image` menu select the `Plug-in Filters` submenu, then the category `FilterMeister` and finally `FilterMeister 0.4.1b...`. This will run FM. You'll see nothing at first, since there is no code loaded. Don't worry, you're only a couple of pages from your first filter experiences!

When you click on `Edit...` you'll see the following window with different interface elements:


AFH FilterMeister Beta 0.4.3 [Flat image, no selection]

Preview window	with zoom buttons and progress bar
Editing window	the place where FM progams are loaded or typed
`Load...` button	to load Filter Factory text file format (`afs`), Filter Factory stand-alone filters (`8bf`), Filter Factory Wizard source files (`ffw`), FilterMeister source files (`ffp`) and normal ASCII text files
`Save...` button	to save in the FilterMeister format `ffp`
`Compile` button	after loading, typing or correcting the FM program in the editing window, you need to compile it first before you can run the program
`Cancel`	Do not apply the algorithm to the current image; the contents in the editing window won't be affected*
`OK`	Apply the algorithm to the current image

* *Applies only to Adobe Photoshop, Illustrator, Macromedia Fireworks*

3.3 Internet Resources

The homepage for FilterMeister is

`http://www.filtermeister.com`

Here you can download the newest versions of the FilterMeister plug-in. Online tutorials will further your programming knowledge, whether you are still a novice or a pro. There will also be an online FAQ (Frequently Asked Questions) that will be updated whenever needed. We recommend that you visit our site periodically.

Contact us by e-mail at `support@filtermeister.com` for direct support.

We also suggest that you join our FilterMeister Mailing List (FMML), where FM filter designers can exchange ideas, comment on our plug-in, etc. Although you can expect support by e-mailing us directly, we also recommend that you post your problems to the Mailing List. The FM group will be available for you there and help you in any way they can.

a) To subscribe to the FMML, send an e-mail message to `fmml-request@lists.best.com` with
`subsingle`
in the body of the message.

b) To unsubscribe from the FMML, send an E-Mail message to `fmml-request@lists.best.com` with
`unsubscribe`
in the body of the message.

c) If you ever need to get in contact with the owner of the list, (if you have trouble unsubscribing, or have questions about the list itself) send an e-mail message to `fmml-owner@filtermeister.com`.

d) Rules:
1. Stay on topic
2. No selling of services or wares
3. No flames
4. If you take, give something back
5. Do not mail files to the list.

4. First Filters with FilterMeister

In the next few chapters, you will be introduced to filter programming with the most common variables, constants and functions. The tutorials presented here are designed to give you an idea of the possibilities FilterMeister can offer. Don't forget that the filters presented here are very basic – if you yearn for more, please consult the *User Guide*. Due to the vast possibilites of FilterMeister, we won't be able to present you with all FF+ elements. Please consult the *Reference Manual*, where all programming elements are listed.

4.1 Modifying Images

In this chapter, you will learn the different basic structures an FM-program can have and how to modify image colors and pixel positions.

Run your graphics program and load any RGB image of your choice. Call up FilterMeister in the `Filter` menu. Type in the editing window the following FM program:

```
%ffp

R:      r
G:      g
B:      b
A:      a
```

This program structure is based on the Filter Factory (FF) structure and does not seem really complicated. Actually, it won't affect your image when applied. All FM programs start with the (optional) header line `%ffp`. It mainly indicates that the following program is written in the Filter Factory Plus (FFP or FF+) language. FM can compile filter programs written in several different source languages such as Filter Factory, Filter Factory Wizard and Filter Factory Plus language and others. But that should not interest us at the moment.

The next four lines imply that each channel has its own formula. In the above case the formula consists of only one variable. These variables (in lower case) are the channel pixel intensities of the current image in the actual (x,y)-position when the filter is applied. The above program will simply read the red, green, blue and alpha (if a non-background layer is active) values of the current pixel and write them back. Picture it as if your image was composed of colored stone mosaics and you lift them to see their individual color and place them back in their original position. So, no need to compile it and apply it to your image (although you can do that if it makes you happy).

Let's alter the image with the following program:

```
%ffp

R:      r+rnd(-10,10)
G:      g*2
B:      b-r
A:      a
```

When finished typing the code, click on `Compile` and you'll see that the filter is applied to the proxy image in the Preview window. Click on OK and the filter is applied to your image. What does the above program do?

In the red channel, FM will take the original red color and add some random number between -10 and 10, inclusive. The green pixel intensities are multiplied by two while in the blue channel the red pixel intensities are subtracted from the blue pixel intensities. There is no rule that you have to use the r, g, b and a variables only in their respective channels.

A couple of chapters ago, we mentioned that a channel is described by one byte. One byte can contain a value from 0 to 255, inclusive (i.e., 256 possible values). In the above program, we multiplied, added and subtracted values and it is possible that a computed value is out of range, i.e., less than 0 (negative) or greater than 255. For example, let's say we have a pixel with the RGB-intensity (0,150,10). The green channel is multiplied by 2; in our case it would lead to 150 × 2 = 300. But the maximum value a byte can have is 255, so the value is "clamped" to that maximum value 255. No matter how far away we are from the range 0 to 255, the values computed are clamped to these maximum boundary values when the filter is applied. The same applies to computed values below 0. All negative values are clamped to 0. In our example, the blue channel formula for our pixel results in 0-150 = -150 clamped to 0.

Note that the variables r, g, b and a always refer to the original unmodified pixel values. A corresponding set of uppercase variables R, G, B and A refers to the newly modified values that will be written to the output image. Let's take an example of the above program applied to a pixel with RGB value (0,150,10). Possibly, the red channel formula computed a 9. The blue channel formula subtracts the red channel from the current blue channel. The equation would then still read 10 - 0 = 10, because we are working with the original pixel values. If the B channel formula is changed to b-R, then the equation would be 10 - 9 = 1, since R refers to the newly modified value of the red channel. This is crucial knowledge for filter creation!

Comments

These codes are fairly easy to understand. Nevertheless, you should try to insert some description of what each line does. This can be done with single-line or multi-line remarks or comments. For example, the above program can be written like this:

```
%ffp

/* This little program created by me will
color the image in a psychedelic way*/

R: r+rnd(-10,10)   //Add to red a random number between -10 and 10, inclusive
G: g*2             //Multiply the green channel value by 2
B: b-r             //Subtract the red channel value from the blue channel value
A: a               //Don’t change the transparency
```

We advise you to use comments as detailed as possible. Think of projects where more than two persons are working on a filter algorithm. Or think of getting back from a 10-day trip and now try deciphering an uncommented algorithm... That can be quite quite time-consuming!

As we said, there are single-line and multi-line commenting possibilites. This is an example of poor multi-line commenting:

```
/* Let’s jam a bit
R: r*rnd(-255,255)
filter finished */
```

The comment in the above code starts from `/*` and ends at `*/`, so the red channel formula is never applied. Here is an example of poor single-line commenting:

```
B: r+b*g  //Psychedelic blue   G: b+r-g  //Musical green
```

Everything after the first `//`-comment is *ignored* by the FM-compiler, so no green channel formula is computed.

ForEveryPixel

Another program structure is the usage of the handler (or function) `ForEveryPixel`. It resembles the FF structure and looks like this:

```
%ffp

ForEveryPixel:          //Define the ForEveryPixel handler
{

R = r+rnd(-10,10);      //Each line ends with a semicolon
G = g*2;
B = b-r;

}
```

The above program looks a lot like a C-program. The `ForEveryPixel`-handler structure is very similar to the Filter Factory structure. The difference is that it is a special function which is intoduced with a colon and the formulas are enclosed by braces `{}`.

ForEveryTile

The following handler, the `ForEveryTile`-handler, processes the piece of the image currently loaded in memory. In 2.4 How does Filtering work? we explained the two different processing modes. If there is enough memory, load the whole image into memory, or else load parts of it (tiles). It looks quite cumbersome to program, but you have more possibilites than the `ForEveryPixel`-handler or the Filter Factory structure can offer. Here is a clean `ForEveryTile`-template without formulas:

```
%ffp

ForEveryTile:
{
for (y=y_start; y<y_end; y++)          //Loop through all rows
        {
        for (x=x_start; x<x_end; x++)  //Loop through all columns
                {
                for (z=0; z<Z; z++)    //Loop through all channels
                { ... }                //Insert the filter algorithm here
                }                      //end x-loop
        }                              //end y-loop

return true;                           //Tile has been completely processed
}                                      //end ForEveryTile
```

Let's see what the FOR-loops do in one example: We are using the variable `y` to go through all rows of the current image tile. The first row of the tile is `y_start`, and the last row is `y_end - 1`. y++ simply means that y is incremented by 1 (we could have written y=y+1 as well).

Our program would look like the following:

```
%ffp

ForEveryTile:
{
for (y=y_start; y<y_end; y++)                        //Loop through all rows
        {
        for (x=x_start; x<x_end; x++)                //Loop through all columns
                {
                pset(x,y,0,src(x,y,0)+rnd(-10,10));  //Change red (channel 0) pixel value
                pset(x,y,1,src(x,y,1)*2);            //Change green (channel 1) pixel value
                pset(x,y,2,src(x,y,2)-src(x,y,0));   //Change blue (channel 2) pixel value
                }                                    //end x-loop
        }                                            //end y-loop

return true;                                         //Tile has been completely processed
}                                                    //end ForEveryTile handler
```

Don't worry if you don't understand it now, we'll explain it thouroughly in the User Guide. The difference from the original template is that we did not add a `z`-loop (`z` being the variable for the current channel value). We are addressing the channels individually within the `x`-loop. There are two new functions used in the listing:

`pset(x, y, z, c)` and `src(x, y, z)`

The `src(x, y, z)`-function tells the filter to read the pixel intensity from the position `(x,y)` and from the specific channel `z`.

The `pset(x, y, z, c)`-function tells the filter to set a pixel at position `(x,y)` and channel `z` to the value specified by formula c.

In `pset(x,y,0,src(x,y,0)+rnd(-10,10));` we are changing the current pixel `(x,y)` in channel 0 (red) with the formula `src(x,y,0)+rnd(-10,10)` which takes the current pixel `(x,y)` in channel 0 (red) and adds to it a random number between -10 and 10, inclusive.

Note: Certain functions can address all of the image's pixels and would not work correctly if their addressing boundaries are changed. Therefore, when these functions are called in an FM program, tiling is deactivated by default, thus obliging the filter to load the whole image as a single large tile. Two examples of such functions are the `src(x, y, z)` *and* `pset(x, y, z, c)` *functions.*

Modifying pixel positions

The examples shown have only changed the pixels' colors in our image. Now we will move some pixels around. Load any RGB-image and run FM with the following FF+ source code:

```
%ffp

// This filter will mirror the image horizontally in all channels

R:      src(X-x-1,y,z)
G:      src(X-x-1,y,z)
B:      src(X-x-1,y,z)
A:      src(X-x-1,y,z)
```

The idea of the horizontal mirroring is to get the pixel from the opposite side of the image. Let's say the filter is processing the leftmost pixel of each row in a 300 × 200 RGB-image, i.e., `x=0`. The image's width is `X`=300. Therefore the expression `X-x-1` reads 300-0-1 = 299 and the function reads `src(299,y,z)`. The filter will take the pixel from (299,y) and place it in the current coordinate (0,y). At position `x`=25, the function will take the pixel from (274,y). In the rightmost pixel of each row, i.e., `x`=299, the expression reads 300-299-1=0. The filter will take the pixel from (0,y) and place it in the position (299,y).

4.2 User Interaction

In most cases, the filter designer (you) wants to give the user the possibility of choosing his/her own filter control settings. In this chapter, you will learn how to implement basic control elements such as sliders or buttons. For further exploration, please consult the User Guide or the Reference Manual.

Slider Interaction

We will create some colored borders around the images. Load any RGB-image and call FM from the `Filter`-Menu. Click on the `Edit...`-button if the editing window is not visible and type in the following listing in FF format:

```
%ffp

ctl[0]: "Border thickness"

R:      x<ctl(0) ? 0 : r
G:      x<ctl(0) ? 0 : g
B:      x<ctl(0) ? 0 : b
```

After clicking on `Compile`, you will see that you just gave birth to a slider (and gave it a name)! Feel free to drag it around as you like and you'll notice how the preview window changes according to your slider setting. The second line in the code, `ctl[0]: ″Border thickness″` activates the user control indexed with the number 0, in our case the first slider. You can activate up to sixty user controls in FM!

Let's examine the code. There are three new elements:

Control function `ctl(x)`

With the `ctl(x)`-function, FM returns the current value setin any user control. For example, if you drag the slider to 137, then this function returns the value 137.

Relational operator <

Relational operators are used to compare two values. In our example, we compared if the current pixel position x is LESS THAN the value set in the slider:

```
x < ctl(0)
```

This operator returns a Boolean value, either 0 (false) or 1 (true)

Conditional operator ? :

This is a very compact IF ... THEN ... ELSE function. Preceding the question mark is a condition, which returns a Boolean value (true or false). *If* the condition true, *then* execute the code between the question mark and the colon, *else* execute the code after the colon. The code can be thus read as:

If the current pixel's position (while filtering) is less than the current slider setting, then set the pixel intensity to 0 else leave it as it is (red, green or blue, depending in which channel these variables are used).

A border is generally used on all four sides of an image, so the condition has to be extended from one side to four sides:

```
x < ctl(0) || y < ctl(0) || x > (X-ctl(0)-1) || y > (Y-ctl(0)-1)
```

The new element here is the *logical operator* `||` for logical OR operations. The above code tests:

- is the current pixel's horizontal position x less than the current slider setting OR
- is the current pixel's vertical position y less than the current slider setting OR...

We have to explain the code for the left and lower borders. Let's say the slider is set to 10 for a 10 pixel border. The last ten pixels can be calculated by subtracting the slider setting from the image's width X and height Y. The expression `x > (X-ctl(0)-1)` tests if the current pixel position is part of the last ten pixels. *The subtraction of 1 is needed because the last possible pixel position does not equal the image's width/height.* Compile and test the complete algorithm:

```
%ffp

ctl[0]: "Border thickness"

R:      x < ctl(0) || y < ctl(0) || x > (X-ctl(0)-1) || y > (Y-ctl(0)-1)  ? 0 : r
G:      x < ctl(0) || y < ctl(0) || x > (X-ctl(0)-1) || y > (Y-ctl(0)-1)  ? 0 : g
B:      x < ctl(0) || y < ctl(0) || x > (X-ctl(0)-1) || y > (Y-ctl(0)-1)  ? 0 : b
```

A black border is rather dull, so let's give the user the possibility of defining his own border color with slider interaction:

```
%ffp

ctl[0]: "Border thickness"
ctl[5]: "Border color (red)"
ctl[6]: "Border color (green)"
ctl[7]: "Border color (blue)"

R:      x < ctl(0) || y < ctl(0) || x > (X-ctl(0)-1) || y > (Y-ctl(0)-1)  ? ctl(5) : r
G:      x < ctl(0) || y < ctl(0) || x > (X-ctl(0)-1) || y > (Y-ctl(0)-1)  ? ctl(6) : g
B:      x < ctl(0) || y < ctl(0) || x > (X-ctl(0)-1) || y > (Y-ctl(0)-1)  ? ctl(7) : b
```

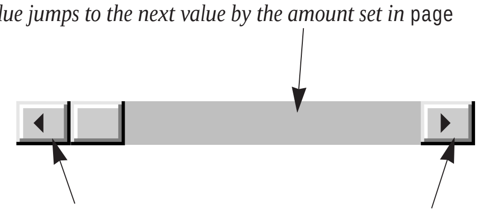
A slider is one example of a user control. A user control can be defined to have different properties, which can change the current look-and-feel of the respective user control. For example, we can redefine the value range a slider can return, or the value set on the first filter call (the default value when an FM program is compiled is 0), and lots more!

Take the last filter we programmed and change the following `ctl[x]’s` to:

```
ctl[0]: "Border thickness", range=(0,100), val=10
ctl[5]: "Border color (red)", page=4
ctl[6]: "Border color (green)", line=2
```

The first line activates the slider 0 with the label ”Border thickness”, sets the possible slider range to 0-100 and sets the initial value to 10. The user control property `page` defines by how much the slider will jump when you click on the slider paging area. The user control property `line` defines by how much the slider will jump when you click the left/right buttons.

When clicking here, the slider value jumps to the next value by the amount set in `page`



When clicking here, the slider value jumps to the next value by the amount set in `line`

The slider (in FF+ notation called scrollbar) is just one of the available user controls. You can implement trackbars, combo boxes, pushbuttons and other user controls as well! Consult the User Guide or Reference Manual for further information on these user controls.

4.3 Changing the User Interface's Appearance

In this chapter, you will learn how to change the filter window's appearance. Each element can be placed and colored. You can even load a background image into your filter window.

Dialog appearance

Type in the following listing and compile it under FM:

```
%ffp

Dialog:  Text = "Super Duper Power Tools", Color = mediumaquamarine, Pos = Screen(15,20),
         Drag = background

ctl[0]: "Scaling"
```

The property `Text` defines the text to be put into the *Title bar*. The background color of the filter window is defined with a *color* of the HTML color space. *Currently, FM supports the color name space X11 (which is part of the HTML JavaScript standard), older HTML-color name spaces, the Java color name space, the Netscape "safe" color space, Microsoft Windows 95 color name spaces, the Win32 system UI color name space and user defined colors (defined as a RGB-triplet or in Hexadecimal annotation). See also the* ***Reference Guide*** *for a complete listing of the supported colors.* The window has also been repositioned with `Pos` to the coordinate (15,20). Finally, `Drag` allows the user to move the whole window by dragging not only the title bar (default), but also the window background.

Notice that all properties are separated with a comma.

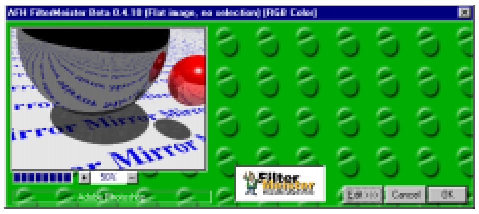
Not only is the filter window color definable, you can also use instead a gradient or a background image! Background gradients are created with the property Gradient, for example:

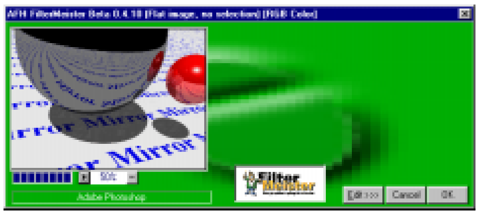
```
Gradient = ( RGB(210,10,80), #EEFF00, H)
```

The first color is defined as a RGB-triplet and the second in HTML-hexadecimal notation. The third sub-key `H` tells FM to orient the gradient in the horizontal direction. The third subkey is *optional*, though. If it is not defined (or by using `V` instead of `H`), then a vertical gradient is used.

Background images can be loaded with the property `image`. Currently, you can load BMP images without RLE (run length encoding). You also have the possibility to tile or stretcha background image with the respective properties `TILED` and `STRETCHED`. Examples:

```
Dialog: image = "byteme.bmp", TILED
Dialog: image = "byteme.bmp", STRETCHED
```


AFH FilterMeister Beta 0.4.18 [Flat image, no selection] [RGB Color]
50%
Edit >>
Cancel
OK


AFH FilterMeister Beta 0.4.18 [Flat image, no selection] [RGB Color]
50%
Adobe Photoshop
Edit >>
Cancel
OK

The left 50 × 50 BMP file has been ***tiled****...* *...and* ***stretched*** *in the filter window*

Please observe the following points:

* *The gradient or bitmap is redrawn when the dialog window is resized (for example, if you press the* `Edit...` *button).*
* *If the BMP file is not located using the PATH list, then the list specified by the FM_PATH environment variable is tried.*
* *If the background image is not found, the most recent Gradient or Color specification is applied.*

User controls

In page 20, you learned how to set the user control properties `range`, `val`, `page` and `line`. Let's examine the following listing:

```
%ffp

ctl[0]: STANDARD, Text = "Color pressure", size = (100,20), pos = (250,30),
        fontcolor=Blue, disabled
```

The first property tells FM to use the default user control, a *scrollbar (also known as a slider)*. You have the option of using scrollbars, trackbars, checkboxes, radio buttons, etc. The property `Text` defines the text to be placed in front of the user control. If you compare the above listing with past listings, you will notice that it is not necessary to include `Text =`; the quoted text string suffices. The `size` property defines the *user control size in DBUs** and the `pos` property defines the *position* of the user control. *The user control origin does not begin with the user control text, but the user control itself.* The `fontcolor` defines the color of the user control text. Finally, the property `disabled` does not allow user interaction for that specific user control.

* *The size is specified in abstract Dialog Box Units (DBUs), which scale according to the system font size. For a VGA display with standard fonts, 1 DBU is approximately 2 pixels.*

4.4 Transferring Data

FM gives the filter programmer the possibility of saving and loading data to and from disk and opening information windows to the user. With FM, one can, for example:

- give the user the possibility of saving control values into an .ini file
- save images in your own file format (WIFF, KPEG, CIA, etc.)
- load and interpret other file formats (create your own Raytracer or maybe a Fractal engine which supports FRACTINT .map color map files)
- save text files with image statistics
- pop up windows with information (or a shareware fee reminder?)

As you can see, there is (almost) nothing FM can't do. The boundary of possibilites is your imagination! Think of the Adobe slogan and start dreaming!

Pop it up

We'll start with popping up a window containing FM's first words. Type in the following source code:

```
%ffp

ForEveryTile:
{
Info("Hello, world!");  //pop up a window containing the text string "Hello, world!"
return false;           //don't change the image
}                       //end ForEveryTile
```

If this program is run within the FM Dialog, an info window containing the text Hello, world! will appear only once. After clicking on OK to "apply" the filter it is possible that the info window appears several times. The reason lies in the handler we are using, ForEveryTile. When the image is tiled, the info window will appear every time a tile is processed. For example, in one of our test machines, an 800 × 600 RGB image is tiled six times.

The following source code should not be run in FM:

```
%ffp

ForEveryPixel:
{
Info("Hello, world!");  //pop up a window containing the text string "Hello, world!"
}                       //end ForEveryPixel
```

Can you guess why we warned you? Each time a pixel is processed, an information window appears. In a 300 × 200 RGB-image, the user would have to press the OK Button of the info box 60,000 times. You must be a mad sadist if your shareware-reminder would appear after processing each pixel. Even a ForEveryTile reminder can be quite annoying...

Input/Outputting data

The following filter resembles the brightness/contrast-function used in most painting programs. The filter features the option of saving and loading the user control settings. In addition to the two scrollbars, two pushbuttons are added. For better reading, comments have been ”lowlighted”.

```
%ffp

// User control definitions

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, i;           //Define file and counter variables

        if (ctl(2))                     //If user clicked on Save
        {
                if (SETTINGS_FILE=fopen("Brightness_Contrast.ini", "w"))
                {
                        for (i=0; i<2; i++)                     //Read user control values
                        fprintf(SETTINGS_FILE, "%d\n",ctl(i));  //Save data to the file
                        if (fclose(SETTINGS_FILE))
                                Error("Can't close settings file.");
                }
                else
                        Error("Can't write to settings file.");
                setCtlVal(2,0);         //Reset pushbutton
        }

        if (ctl(3))                     //If user clicked on Restore
        {
                int iVal;       //Variable where user control data is saved temporarily
                if (SETTINGS_FILE=fopen("Brightness_Contrast.ini", "r"))
                {
                        for (i=0; i<2; i++)
                        {
                                fscanf(SETTINGS_FILE, "%d", &iVal);     //Read data from file
                                setCtlVal(i,iVal);           //and assign it to user controls
                        }
                        if (fclose(SETTINGS_FILE))
                                Error("Can't close settings file.");
                }
                else
                        Error("Can't open settings file.");
                setCtlVal(3,0);         //Reset Pushbutton
        }

return false;           //Leave image as it is
}                       //End ForEveryTile
```

```
// Apply the following filter to the image

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)
```

On first glance, it might look all too complicated, so we will examine it here. The program is divided into three sections: The user control definitions, the `ForEveryTile` handler and the filter code in FF format.

In the *first* section, the user controls are defined. The user controls 0 and 1 are *scrollbars* and the user controls 2 and 3 are *pushbuttons.* We suggest you to play a bit with the user control properties.

In the *second* section, the `ForEveryTile` handler is called. Notice that the handler returns the *boolean* value `false`. Since the handler does not perform any image filtering, it is set to return false. That way, the `ForEvery`**`Pixel`** handler or the *FF-style filter function* (as used above) processes the image. Set the return value to `true` and you will notice that a scrollbar movement does not change the image in the preview window.

All the `ForEveryTile` handler does is check if one of the pushbuttons has been clicked. The variable `SETTINGS_FILE` is used as a file handle. Let's examine the following expression:

```
SETTINGS_FILE=fopen("Brightness_contrast.ini", "w"))
```

A file with the name `Brightness_contrast.ini` is opened for write (`"w"`) purposes and the return value is saved in the `SETTINGS_FILE` variable. The variable can tell if the file has been opened (any nonzero number or true) or not (0 or false). When the variable is assigned with 0, an error has occured. This can mean that the disk is either full or no media is inserted (if saved to a diskette) or the file's attribute `read-only` is active. If you change the `"w"` to `"r"` in the above expression, the file is opened for reading purposes. This is used later for the **Restore**-button. If the `SETTINGS_FILE` variable is set to 0, then an error occurred. This can mean that no file has been found or the file is corrupt.

As you may know, the `.ini`-file format describes an ASCII-Text file. It is possible to read/write files in binary mode. All you need to do is change `"w"` to `"wb"` for writing purposes and `"r"` to `"rb"` for reading purposes.

When the **Save** pushbutton is clicked, the settings-file is opened for writing purposes and testedto make sure if the `SETTINGS_FILE` variable returns a nonzero number. A zero would lead to an error window. A nonzero number would start a for-loop which saves the user control settings to the file. Let's examine the following expression:

```
fprintf(SETTINGS_FILE, "%d\n", ctl(i));
```

`fprintf` is the command for a *formatted file print*. Consult the **Reference Manual** for further knowledge of the `fprintf`-command. Between the parentheses, the `SETTINGS_FILE` handle is used to tell the filter which file the following commands are written to. Next, a string `"%d\n"` is defined. The `%d` is a placeholder for a **d**ecimal variable number which is defined after the string (in this case `ctl(i)`). In plain language: Write the current setting of user control `i` in decimal form. The `\n` part is an ASCII *escape command* which tells the writing procedure to enter a new line in the file. If the first scrollbar was set to 65 and the second scrollbar was set to -54, the loop would write in the file the following two lines:

65
-54

Try setting these values in the FM dialog and save them. Go to the Explorer and edit the file `Brightness_contrast.ini` with the a text editor such as the Notepad.

The `fclose` command tests if the file was successfully closed after writing all the values. If the disk was full, then an error window appears containing `"Can't close settings file."`.

If the writing procedure was successful, the pushbutton's value is set back to 0. If this not done, clicking the pushbutton once would lead to a constant file writing. You can picture this as if the button was jammed.

When the **Restore** button is clicked, a new variable `iVal` is defined. First, the file is opened and checked for any errors. If no errors were encountered, a loop reads the data from the file. The command `fscanf` does the opposite of what `fprintf` does. It scans or reads the file data and assigns it to the `iVal` variable. Within the loop the user controls are assigned to this variable with the following expression:

```
setCtlVal(i, iVal);             //Set the current user control i to the setting iVal
```

If the file is corrupt, an error window appears containing `"Can't close settings file."`.

If an error was encountered while opening the file, an error message appears containing `"Can't open settings file."`

Finally, the pushbutton's value is reset to 0. If this is not done, clicking the pushbutton would lead to a constant file opening and user control value assignment (or constant error messages).

In the *third* section of the listing, the actual filter code is written in FF format. You don't have to understand the code at the moment. All you need to know here is that with the first scrollbar, you can brighten or darken the image. The second scrollbar modifies your image's contrast.

5. Filter Factory and FilterMeister

This chapter is intended for users with knowledge of Filter Factory programming.

5.1 Historical notes

Filter Factory was introduced in 1994 by Joe Ternasky, an Adobe employee who was inspired by Niklaus Wirth to create a small and easy filter compiler. FF was included in the Photoshop 3.0 Deluxe CD-ROM for Macintosh and Windows as a ”Goodie”. A couple of tutorials and a small 15-page manual in PDF format introduced the user to a whole new world of filter creation. Although users had the possibility of exploring the Photoshop CD-ROM, nobody paid much attention to FF back then. Even the author of this manual remembers the first installation of FF and running it under Photoshop – the plug-in was deinstalled a couple of minutes later...

The looks of the interface, the programming possibilites haven't changed since then. There are two different FF versions known, though. The only difference seemed to be the size of the standalone filter files. The older version created 56 KB size files, the newer version, as of 1995 created 48 KB size files. The reason was no internal code optimizing, but rather the call of functions from the PLUGIN.DLL file installed in the Photoshop directory. That is the reason why newer FF filters need this file in order to run in non-Adobe programs (such as Corel PhotoPaint or JASC Paint Shop Pro). But the programming possibilities given to the users weren't changed.

The structure of the standalone plug-ins created with FF was very easy for FF to build. FF is a JIT-compiler (a just-in-time compiler). It created standalone plug-ins by copying itself with the filtering algorithm in a file, including the current scrollbar settings. The standalone plug-in compiled the filter algorithm before applying it to the image. The reason that all FF standalone filters are the same size is that FF allocates fixed-size data structures to hold variable-length info such as the filter source code.

Alfredo Mateus, a Brazilian who was then a chemistry student, founded in 1995 with Mario Klingemann, a German graphic designer, the Filter Factory Discussion Group (FFDG), a mailing list for FF-fanatics. Soon, many people subscribed to the list and lots of information about FF started to go around. In the end of 1996, a Filter Factory FAQ (Frequently-Asked-Questions) was introduced by Werner D. Streidt, a German student of printing technology. The FAQ evolved, due to its subject matter, to a 51-page Filter Factory Programming Guide (FFPG).

During these years, different Internet sites presented FF-relevant contents. Filters were given for free and their amount sprang up to about 1,000! In 1997, different utilities like PlugIn Manager and PlugIn Commander appeared. These two utilities by Michael Johannhanwahr and Harald Heim, respectively, import, decompile, compile and manage FF filters.

Thanks to some hints by some users did the FFDG community notice that FF was also available for Adobe Premiere, a video editing program. The Premiere version was so to say the animated Photoshop FF-version.

From November 1997 on, German computer magazines started introducing FF-based articles. Some of them introduced the FF programming language while others presented filter collections in the Internet. In December 1997, Alexander F. Hunter, a professional programmer, had worked on a successor plug-in for FF and presented his first filter "Lift the Cover" (based on Mario Klingemann's filter wth the same name) created with the Filter Meister (back then written separately). During 1998, people in the FFDG made suggestions for new features in Filter Meister. After a five-month beta-test, FilterMeister (now written together) was presented to the public in June 1999.

In the fourth quarter of 1998, another FF-successor was presented to the public: Filter Formula by ATS/Graphics, an Austrian company. The FF programming language and dialog were expanded by useful functions and elements.

5.2 Compatibility issues

Although FilterMeister has its own new language, the FF+ language, it has successfully managed to integrate the original FF language and combine it with the new FF+ structures. In order to make it easier for FF users to port to FM, full FF language compatibility has been the first goal for FM. This way, you do not need to change your filter codes in order for them to work in FM.

FilterMeister can import the FF source code files and standalone plug-in files created with Filter Factory. This is done when you run FilterMeister and click on the Load... button. Simply select the file of your choice. There are differences between these two file formats. While the standalone plug-ins (.8bf-files) are imported with source code, scrollbars with start settings and filter descriptions (Author, Category, Copyright, Filename, etc.), the import of source code files (.afs-files) loads the source code and activates all scrollbars. You will have to check which scrollbars are used in the filter code and name them.

Making FF source code FM-compliant

With the possibilites of FM, you ought to change your FF source code. Prior to FM, the FF user had the following problems:

- scrollbar range did not change when using the val-function
- scrollbars were used as toggles

Here is an example of a very simple FF filter being made FF+ compliant:

```
%ffp

ctl[0]: "Line occurrence"
ctl[7]: "Horizontal/Vertical Toggle"

R, G, B:
ctl(7)>128 ?             //scrollbar 7 used as a toggle
y%val(0,1,20) ? c : 255  //horizontal white lines with a max. spacing of 20 pixels
:                        //following line is executed if scrollbar 7 smaller or equal to 128
x%val(0,1,20) ? c : 255  //vertical white lines with a max. spacing of 20 pixels
```

turns into:

```
%ffp

ctl[0]: "Line occurrence", range=(1,20)
ctl[7]: CHECKBOX, "Horizontal/Vertical Toggle", size=(95,*)

R, G, B:
ctl(7) ?               //Is checkbox active?
y % ctl(0) ? c : 255   //Horizontal white lines
:
x % ctl(0) ? c : 255   //Vertical white lines
```

The listing is easier to read now. The scrollbar range is displayed now in the dialog and a checkbox is used as a toggle. Notice that the size of the checkbox area had to be enlarged due to the text length. The sign * tells FM to use the default value.

Please consider the following aspects, as well. In the `ForEveryPixel` *handler the* `c` *and* `z` *variables are not defined, since each pixel is processed in its entirety (all channels at once) and not one channel within a pixel. In the* `ForEveryTile` *handler, all the FF-variables (such as x, y, z, m, d, etc.) are not defined. The constants (X, Y, Z, M, D, etc.) and functions, though, can still be used.*