Hans Hagen
Ridderstraat 27, 8061GH
Hasselt
pragma@wxs.nl

metapost
MetaFun
Chapter 3: Embedded graphics

keywords
METAPOST, MetaFun, ConTEXt, graphics

abstract
This article is a nearly 100% copy of chapter 3 of the MetaFun manual. This chapter discusses a few alternative ways to define and include METAPOST graphics in a document.
This article contains some colors, so if you want to get the real picture, you should fetch the MetaFun manual from www.pragma-ade.com (beta manual download page). The MetaFun macros (METAPOST as as well as TEX) are part of the regular ConTEXt distribution.

Introduction

This is chapter 3 of the METAFUN manual and introduces the basic definition and inclusion macros. In the previous chapters, METAPOST was introduced, while the following chapters discuss how to enhance the layout, how to achieve special effects, and how to define and debug graphics. The METAFUN manual fully covers METAPOST and the way it can be used in TEX.

At the time of this writing, the METAFUN manual spans some 260 pages. The manual is available in two versions: a reduced A4 format, and a version designed for reading from computer screens. The manual is not yet finished, but can be fetched from the CONTEXT beta manuals download page at www.pragma-ade.com or one of its mirrors.

Getting started

From now on, we will assume that you have CONTEXT running on your platform. Since PDF has full graphics support, we also assume that you use PDFTEX, or know how to go from DVI to PDF. Since this document is not meant as a CONTEXT tutorial, we will limit this introduction to the basics needed to run the examples.

A simple document looks like:

```
\starttext
  Some text.
\stoptext
```

You can process this document with the PERL based command line interface to CONTEXT. If the source code is embedded in the file `mytext.tex`, you can say:

```
texexec --pdf mytext
```

As an alternative to `-pdf`, you can explicitly set the output driver in your document:

```
\setupoutput[pdftex]
\starttext
  Some text and/or graphics.
\stoptext
```

Yet another alternative is:

```
% interface=english output=pdftex
\starttext
  Some text and/or graphics.
\stoptext
```

Here the `interface` directive tells TEXEXEC that it should force the english user interface.

We will use color, and since traditionally TEX is rather unaware of color, this feature is turned off by default, so, if you want to see color, you should type:

```
\setupcolors[state=start]
\starttext
  Some \color[blue]{text} and/or \color[green]{graphics}.
\stoptext
```

As an alternative, you can run TEXEXEC like:

```
texexec --pdf --color mytext
```

In later chapters we will occasionally see some more CONTEXT commands show up. If you want to know more about what CONTEXT can do for you, we recommend the beginners manual and the reference manual, as well as the manual that comes with TEXEXEC.

External graphics

Since TEX has no graphic capabilities built in, a graphic is referred to as an external figure. A METAPOST graphic often has a number as suffix, so embedding such a graphic is done by:

```
\externalfigure[graphic.123][width=4cm]
```

An alternative method is to separate the definition from the inclusion. An example of a definition is:

```
\useexternalfigure[pentastar][star.803][height=4cm]
\useexternalfigure[octostar] [star.804][pentastar]
```

Here, the second definition inherits the characteristics from the first one. These graphics can be summoned like:

```
\placefigure
  {A five||point star drawn by \METAPOST.}
  {\externalfigure[pentastar]}
```

Here the stars are defined as stand–alone graphics, in a file called `star.mp`. Such a file can look like:

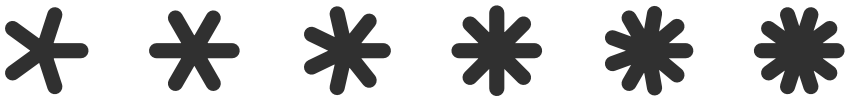
```
def star (expr size, n, pos) =
  for a=0 step 360/n until round(360*(1-1/n)) :
    draw (origin -- (size/2,0))
      rotatedaround (origin,a) shifted pos ;
  endfor ;
enddef ;

beginfig(803) ;
  pickup pencircle scaled 2mm ; star(2cm,5,origin) ;
endfig ;

beginfig(804) ;
  pickup pencircle scaled 1mm ; star(1cm,8,origin) ;
  pickup pencircle scaled 2mm ; star(2cm,7,(3cm,0)) ;
endfig ;
```

```
end.
```

This `star` macro will produce graphics like:



Integrated graphics

An integrated graphic is defined in the document source or in a style definition file. The most primitive way of doing this is beginning with the definition of the graphic.

```
\startMPgraphic
  fill fullcircle scaled 200pt withcolor .625white ;
\stopMPgraphic
```

Next the graphic can be loaded, using:

```
\loadcurrentMPgraphic{optional setups}
```

Finally, the graphic is placed in the document with:

```
\placeMPgraphic
```

The optional setups are passed on to the figure inclusion macro, which in ConTeXt is the command `\externalfigure`.

Since every definition replaces the previous one, this method forces you to embed the definitions in the running text. In this document we also generate graphic while we finish a page, so there is a good change that when we have constructed a graphic which will be called the next page, the wrong graphic is placed.

Therefore you may as well forget these commands, since there are more convenient ways of defining and using graphics, which have the added advantage that you can predefine multiple graphics, thereby separating the definitions from the usage.

The first alternative is a *usable* graphic. Such a graphic is calculated anew each time it is used. An example of a usable graphic is:

```
\startuseMPgraphic{name}
  fill fullcircle scaled 200pt withcolor .625yellow ;
\stopuseMPgraphic
```

When you put this definition in the preamble of your document, you can place this graphic anywhere in the file, saying:

```
\useMPgraphic{name}
```

As said, this graphic is calculated each time it is placed, which can be time consuming. Apart from the time aspect, this also means that the graphic itself is incorporated many times. Therefore, for graphics that don't change, ConTeXt provides *reusable* graphics:

```
\startreusableMPgraphic{name}
  fill fullcircle scaled 200pt withcolor .625yellow;
\stopreusableMPgraphic
```

This definition is accompanied by:

```
\reuseMPgraphic{name}
```

Imagine that we use a graphic as a background for a button. We can create a unique and reusable graphic by saying:

```
\def\MyGraphic%
  {\startreusableMPgraphic{name:\overlaywidth:\overlayheight}
      path p ; p := unitsquare
        xscaled \overlaywidth yscaled \overlayheight ;
      fill p withcolor .625yellow ;
      draw p withcolor .625red ;
   \stopreusableMPgraphic
   \reuseMPgraphic{name:\overlaywidth:\overlayheight}}
```

After this we can say:

```
\defineoverlay[my graphic][\MyGraphic]
\button[background=my graphic,frame=off]{Go Home}[firstpage]
```

Say that we have a 30pt by 20pt button, then the identifier will be `name:30pt:20pt`. Different dimensions will lead to other identifiers, so this sort of makes the graphics unique.

We can bypass the ugly looking `\def` by using a third class of embedded graphics, the *unique* graphics.

```
\startuniqueMPgraphic{name}
  path p ; p := unitsquare
    xscaled \overlaywidth yscaled \overlayheight ;
  fill p withcolor .625yellow ;
  draw p withcolor .625red ;
\stopuniqueMPgraphic
```

Now we can say:

```
\defineoverlay[my graphic][\uniqueMPgraphic{name}]
\button[background=my graphic,frame=off]{Go Home}[firstpage]
```

You may wonder why unique graphics are needed when a single graphic might be used multiple times by scaling it to fit the situation. Since a unique graphic is calculated for each distinctive case, we can be sure that the current circumstances are taken into account. Also, scaling would result in incomparable graphics. Consider the following definition:

```
\startMPgraphic
  draw unitsquare
    xscaled 5cm yscaled 1cm
    withpen pencircle scaled 2mm
    withcolor .625red ;
\stopMPgraphic
```

Since we reuse the graphic, the dimensions are sort of fixed, and because the graphic is calculated once, scaling it will result in incompatible line widths.

These graphics were placed with:

```
\hbox \bgroup
  \loadcurrentMPgraphic{width=5cm,height=1cm}\placeMPgraphic \quad
  \loadcurrentMPgraphic{width=8cm,height=1cm}\placeMPgraphic \egroup
```

Imagine what happens when we add some buttons to an interactive document without taking care of this side effect. All the frames would look different. Consider the following example.

```
\startuniqueMPgraphic{right or wrong}
  pickup pencircle scaled .075 ;
  fill unitsquare withcolor .8white ;
  draw unitsquare withcolor .625red ;
  currentpicture := currentpicture
    xscaled \overlaywidth yscaled \overlayheight ;
\stopuniqueMPgraphic
```

Let's define this graphic as a background to some buttons.

```
\defineoverlay[button][\uniqueMPgraphic{right or wrong}]
\setupbuttons[background=button,frame=off]
```

```
\hbox
  {\button {previous}          [previouspage]\quad
   \button {next}              [nextpage]\quad
   \button {index}             [index]\quad
   \button {table of contents} [content]}
```

The buttons will look like:

Compare these with:

Here the graphic was defined as:

```
\startuniqueMPgraphic{wrong or right}
  pickup pencircle scaled 3pt ;
  path p ; p := unitsquare
    xscaled \overlaywidth yscaled \overlayheight ;
  fill p withcolor .8white ;
  draw p withcolor .625red ;
\stopuniqueMPgraphic
```

The last class of embedded graphics are the *runtime* graphics. When a company logo is defined in a separate file `mylogos.mp`, you can run this file by saying:

```
\startMPrun
  input mylogos ;
\stopMPrun
```

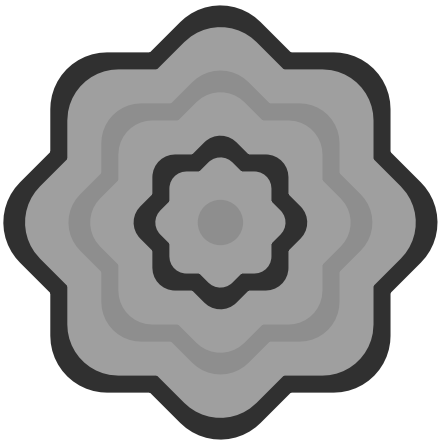
The source for the logo is stored in a file named `mylogos.mp`.

```
beginfig(21) ;
  draw fullsquare            withcolor .625red ;
  draw fullsquare rotated 45 withcolor .625red ;
  picture cp ; cp := currentpicture ;
  def copy = addto currentpicture also cp enddef ;
  copy scaled .9 withcolor .625white  ;
  copy scaled .7 withcolor .625yellow ;
  copy scaled .6 withcolor .625white  ;
  copy scaled .4 withcolor .625red    ;
  copy scaled .3 withcolor .625white  ;
  fill fullcircle scaled .2 withcolor .625yellow ;
  currentpicture := currentpicture scaled 50 ;
```

```
endfig ;
end .
```

In this example the result is available in the file `mprun.21`. This file can be included in the normal way, using:

```
\externalfile[mprun.21][width=5cm]
```



Figuur 1 The logo is defined in the file `mylogos.mp` as figure 21 and processed by means of the `mprun` method.

Graphic buffers

In addition to the macros defined in the previous section, you can use ConTeXt's buffers to handle graphics. This can be handy when making documentation, so it makes sense to spend a few words on them.

A buffer is a container for content that is to be (re)used later on. The main reason for their existence is that they were needed for typesetting manuals and articles on TeX. By putting the code snippets in buffers, we don't have to key in the code twice, since we can either show the code of buffers verbatim, or process the code as part of the text flow. This means that the risk of mismatch between the code shown and the typeset text is minimized.

```
\startbuffer
You are reading the \METAFUN\ manual.
\stopbuffer
```

This buffer can be typeset verbatim using `\typebuffer` and processed using `\haalbuffer`, as we will do now:

An other advantage of using buffers, is that they help you keeping the document source clean. In a many places in this manual we put table or figure definitions in a buffer and pass the buffer to another command, like:

```
\placefigure{A very big table}{\haalbuffer}
```

Sometimes it makes sense to collect buffers in separate files. In that case we give them names.

This time we should say `\typebuffer[mfun]` to typeset the code verbatim. Instead of TeX code, we can put METAPOST definitions in buffers.

Buffers can be used to stepwise build graphic. By putting code in multiple buffers, you can selectively process this code.

```
\startbuffer[red]
drawoptions(withcolor .625red) ;
\stopbuffer

\startbuffer[yellow]
drawoptions(withcolor .625yellow) ;
\stopbuffer
```

We can now include the same graphic in two colors by simply using different buffers. This time we use the special command `\processMPbuffer`, since `\haalbuffer` will typeset the code fragment, which is not what we want.

```
\startregelcorrectie[blanko]
\processMPbuffer[red,graphic]
\stopregelcorrectie
```

The line correction macros take care of proper spacing around the graphic. The `[blanko]` directive tells ConTeXt to add more space before and after the graphic.

```
\startregelcorrectie[blanko]
\processMPbuffer[yellow,graphic]
\stopregelcorrectie
```

Which mechanism you use, (multiple) buffers or (re)usable graphics, depends on your preferences. Buffers are slower but don't take memory, while (re)usable graphics are stored in memory which means that they are accessed faster.

Communicating color

Now that color has moved to the desktop, even simple documents have become more colorful, so we need a way to consistently apply color to text as well as graphics. In ConTeXt, colors are called by name.

The next definitions demonstrate that we can define a color using different color models, RGB or CMYK. Depending on the configuration, ConTeXt will convert one color system to the other, RGB to CMYK, or vice versa. The full repertoire of color components that can be set is as follows.

```
\definecolor[color one]  [r=.1, g=.2, b=.3]
\definecolor[color two]  [c=.4, m=.5, y=.6, k=.7]
\definecolor[color three][s=.8]
```

The numbers are limited to the range 0..1 and represent percentages. Black is represented by:

```
\definecolor[black 1] [r=0, g=0, b=0]
\definecolor[black 2] [c=0, m=0, y=0, k=1]
\definecolor[black 3] [s=0]
```

Predefined colors are passed to METAPOST graphics via the `\MPcolor`. First we define some colors.

```
\definecolor[darkyellow][y=.625] % a CMYK color
\definecolor[darkred]   [r=.625] % a RGB color
\definecolor[darkgray]  [s=.625] % a gray scale
```

These are the colors we used in this document. The next example uses two of them.

```
\startuseMPgraphic{color demo}
  pickup pencircle scaled 1mm ;
  path p ; p := fullcircle xscaled 10cm yscaled 1cm ;
  fill p withcolor \MPcolor{darkgray} ;
  draw p withcolor \MPcolor{darkred} ;
\stopuseMPgraphic

\useMPgraphic{color demo}
```

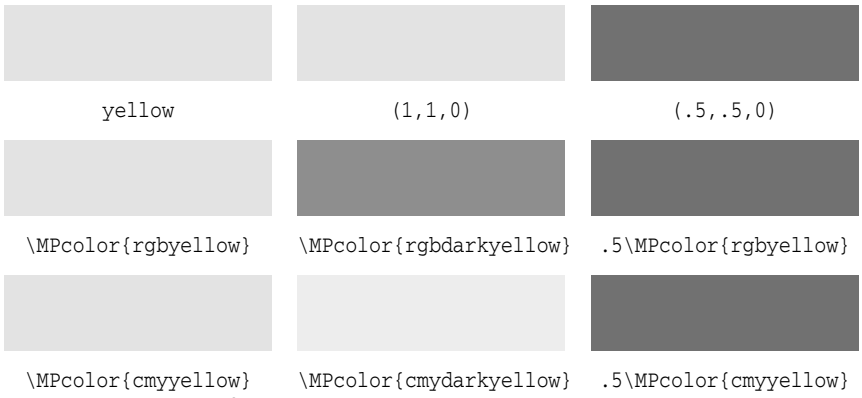
The previous example uses a pure RGB red shade, combined with a gray fill.

Since METAPOST does not support the CMYK color space and native gray scales — although gray colors are reduced to the more efficient PostScript `setgray` operators in the output— the macro `\MPcolor` takes care of the translation from CMYK to RGB as well as gray to RGB. However, there is a fundamental difference between a yellow as defined in ConTeXt using CMYK and an RGB yellow in METAPOST.

```
\definecolor[cmyyellow]    [y=1]
\definecolor[rgbyellow]    [r=1,g=1]

\definecolor[cmydarkyellow][y=.625]
\definecolor[rgbdarkyellow][r=.625,g=.625]
```

Figure 2 demonstrates what happens when we multiply colors by a factor. Since we are not dealing with real CMYK colors, multiplication gives different results for CMYK colors passed as `\MPcolor`.



Figuur 2 All kinds of yellow.

So, `.625red` is the same as `[r=.5]`, but `.625yellow` is not the same as `[y=.5]`, but matches `[r=.5,g=.5]`. Figure 3 shows the pure and half reds.

Figuur 3 Some kinds of red.

In order to prevent problems, we advise you to stick to RGB color specifications when possible. That way you prevent not only conversion problems, but the also (often obscure) ways printing and viewing devices handle CMYK.

Common definitions

When using many graphics, there is a chance that they share common definitions. Such shared components can be defined by:

```
\startMPinclusions
  color mycolor ; mycolor := .625red ;
\stopMPinclusions
```

All METAPOST graphics defined in the document end up in the files `mpgraph.mp` and `mprun.mp`. When processed, they produce (sometimes many) graphic files. When using TEXEXEC to process documents, these two files are processed automatically after a run so that in a next run, the right graphics are available.

When you are using the `web2c` distribution, CONTEXT can call METAPOST at runtime and thereby use the right graphics instantaneously. In order to use this feature, you have to enable `\write18` in the file `texmf.cnf`. Also, in the file `cont-sys.tex`, that holds local preferences, or in the document source, you should say:

```
\runMPgraphicstrue
```

This enables runtime generation of graphics using the low level TEX command `\write18`. First make sure that your local brand of TEX supports this feature. A simple test is making a TEX file with the following line:

```
\immediate\write18{echo It works}
```

If this fails, you should consult the manual that comes with your system, locate an expert or ask around on the CONTEXT mailing list. Of course you can also decide to let TEXEXEC take care of processing the graphics afterwards. This has the advantage of being faster but has the disadvantage that you need additional TEX runs.

If you generate the graphics at run time, you should consider to turn on graphic slot recycling, which means that you often end up with fewer intermediate files:

```
\recycleMPslotstrue
```

There are a few more low level switches and features, but these go beyond the purpose of this manual. Some of these features, like the option to add tokens to `\everyMPgraphic` are for experts only, and fooling around with them can interfere with existing features.

One page graphics

Although all of what is demonstrated in this document is done in CONTEXT, some of the features discussed here can also be done in plain TEX. In the METAFUN distribution there is a file called `plainfun.tex`, which loads the appropriate CONTEXT modules.

Many low level macros are rather generic, and can be used in plain TEX without problems. However, the big advantage of using CONTEXT is, that graphics can be part of the text flow and that you can put them on layers. If you don't want this, and only want to make stand alone graphics, you may still consider using CONTEXT for that purpose.

Another advantage is that when using CONTEXT you don't have to bother about specials, font inclusion and all those nasty things that can spoil a good day. An example of such a graphic is the file `mfun-888` that resides on the computer of the author.

```
[file mfun-888 bestaat niet]
```

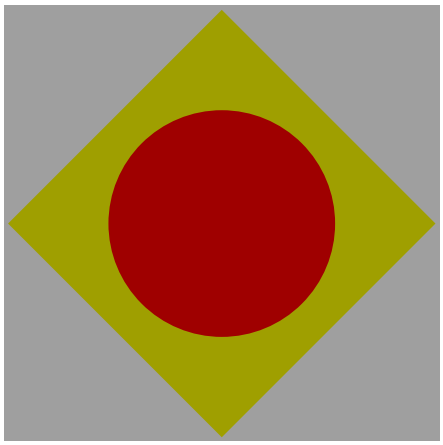
Given that the ConTeXt english interface format is present on your system, you can process this file with TeXexec, for instance using pdfTeX. The `-once` directive saves some runtime.

```
texexec --once --pdf mfun-888
```

You can define many graphics in one file. The TeXexec manual describes how to selectively process pages. If you use pdfTeX, you can include individual pages from pdf files:

```
\placefigure
  {A silly figure, demonstrating that stand||alone||graphics
   can be made.}
  {\externalfigure[mfun-888][page=1]}
```

In this case the `page=1` specification is not really needed. You can scale and manipulate the figure in any way supported by the macro package that you use.



Figuur 4 A silly figure, demonstrating that stand–alone–graphics can be made.

Managing resources

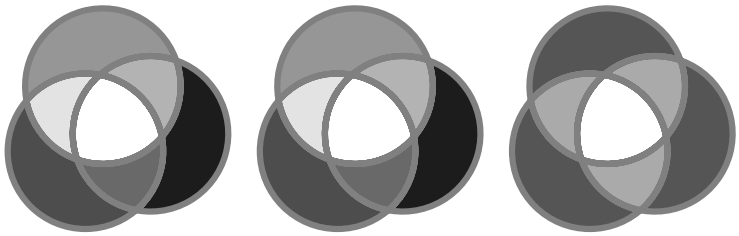
A graphic consists of curves, either or not filled with a given color. A graphic can also include text, which means that fonts are used. Finally a graphic can have special effects, like a shaded fill. Colors, fonts and special effects go under the name resources, since they may demand special care or support from the viewing or printing device.

When fonts are used, a METAPOST file is not self contained. This means that the postprocessing program has to deal with the fonts. In ConTeXt, the special driver —and pdfTeX support is considered as such— takes care of this. We will discuss text related issues in chapter ??.

Special effects, like shading, are supported by dedicated METAPOST modules. These are included in the ConTeXt distribution and will be discussed later in chapter ??.

Since METAPOST supports color, an embedded graphic can be rather colorful. However, when color support is disabled or set up to convert colors to gray scales, ConTeXt will convert the colors in the graphic to gray scales.

You may wonder what the advantage is of weighted gray conversion. Figure 5 shows the difference between natural colors, weighted gray scaled and straightforward, non–weighted, gray scales.



full color weighted gray linear gray

Figuur 5 The advantage of weighted gray over linear gray.

When we convert color to gray, we use the following formula. This kind of conversion also takes place in black and white televisions.

$$G = .30r + .59g + .11b$$

Kluwer Academic Publishers (KAP) is an internationally operating company, publishing 700 journals and 800 books per year. For our rapidly developing electronic publishing activities we have a vacancy for a

(La)TeX Developer

In this technically challenging position you will be responsible for various (La)TeX−related development and production activities, and for the development and implementation of software solutions at the front and back end of the production process of scientific information. The (La)TeX developer will be responsible for bridging the gap between SGML/XML content and (La)TeX−based material.

KAP is looking for a team player with, besides (La)TeX expertise, a broad technical knowledge. You should be able to demonstrate knowledge in SGML/XML/HTML, databases and programming languages such as Java, Perl, OmniMark, etc. Experience in working on different computing platforms and operating systems would be an advantage. You should be able to communicate your ideas to other members of the team and support both internal and external users, and be a champion in the production department for new technology.

Kluwer Academic Publishers, with offices also in the USA (Boston and New York) is part of the Wolters Kluwer organisation, and offers a challenging position with good conditions and career opportunities in an excellent working environment.

Send your CV and a covering letter explaining why you are the right person for this position, in English, to

Mr. Rob Doornebal
Kluwer Academic Publishers
Achterom 119
3311 KB Dordrecht

For more information on this function you can contact Mr. Rob de Jeu at (0)78 6392524 (phone) or Rob.deJeu@wkap.nl (email).