Coloring ConTeXt

explaining luatex and mkiv

Hans Hagen

PRAGMA ADE

Contents

Introduction

1 Basics

1.1 Color models 5
1.2 Using color 5
1.3 Using cmyk or rgb 6
1.4 Conversion 8
1.5 Definitions 10
1.6 Freezing colors 13
1.7 Color groups 14
1.8 Palets 14
1.9 Transparency 15
1.10 Interpolation 17
1.11 PDF 18
1.12 Unboxing 19
1.13 Color intents 20
1.14 Collections 21
1.15 Text color 23
1.16 Tikz 23
1.17 Implementation details 24
1.18 Grouping 25
1.19 Commands 25

2 Metafun

2.1 Defining and using 31
2.2 Passing colors 32
2.3 Grouping 33
2.4 Transparency 35
2.5 Shading 36
2.6 Text 40
2.7 Helpers 40

3 Graphics

3.1 Conversion 45
3.2 Recoloring 46
3.3 Profiles 48
3.4 Masks 49

2

Contents

Introduction

This manual fits in the series where we discus fundamental subsystems like fonts and languages. Here we will collect the more technical backgrounds. This document is not meant as a manual for users who start with ConTEXt, for that we have other manuals.

Color has a rather long history in ConTEXt because we supported it right from the start. In the times that dvi backend drivers were used, specials were the way to force color in the result. However, each driver had different demands: some expected specific color directives, others a sequence of for instance PostScript commands. When pdf showed up, resource management entered the game. Because ot always used a backend driver model in ConTEXt, it could easily be adapted. All management, for instance of nested colors, was done in TEX code. If advanced color support hadn't been available right from the start, we'd probably not be using TEX now.

In MkIV color support was implemented from scratch but in a for the user downward compatible way. In that respect this manual is not going to reveal anything revolutionary. Much of the work is now delegated to Lua and because of that directives are no longer part of the (expanded) input stream. As a result color is now more robust and less intrusive.

Because MetaPost support is well integrated, we also communicate colors to MetaPost. In MkIV the communication between the two engines was upgraded and hopefully evolved into an (even) more convenient interface.

External graphics are in fact islands in the document flow: they manage their resources like colors themselves. However, there are some ways to deal with the demands of publishers and printers with respect to colors. These will be discussed too.

This document is still under construction. The functionality discussed here will stay and more might show up. Of course there are errors, and they're all mine. The text is not checked for spelling errors. Feel free to let me know what should get added.

Hans Hagen
PRAGMA ADE, Hasselt NL
2016

Introduction

1 Basics

1.1 Color models

When you work with displays, and most of us do, the dominant color model is rgb. As far as I know cmyk electrowetting displays are still not in production and even there the cmyk seems to have made place for rgb (at least in promotion movies). This is strange since where rgb is used in cases where colors are radiated, cmyk shows up in reflective situations (and epub readers are just that). But rgb and cmyk being complementary is not the only difference: cmyk has an explicit black channel, and as a consequence you cannot go from one to the other color space without loss.

In print cmyk is dominant but in order to get real good colors you can go with spot colors. The ink is not mixed with others but applied in more or less quantity. A mixture of spot colors and cmyk is used too. You can combine spot colors into a so called multitone color. Often spot colors have names (for instance refering to Pantone) but they always have a specification in another color space in order to be shown on screen. Think of “gold” being a valid ink, but hard to render on screen, so some yellowish replacement is used there when documents get prepared on screen.

In ConTEXt all these models are supported, either or not at the same time. In MkII you had to turn on color support explicitly, if only because of the impact of the overhead on performance, but in MkIV color is on by default. You can disable it with:

```
\setupcolors
  [state=stop]
```

The three mentioned models are controlled by keys, and by default we have set:

```
\setupcolors
  [rgb=yes,
   cmyk=yes,
   spot=yes]
```

Spot colors and their combinations in multitone colors are controlled by the same parameter. You can define colors in the hsv color space but in the end these become and behave like rgb.

1.2 Using color

Normally you will use colors grouped. Most environments accept a `color` parameter (some have `textcolor` or similar longer names too). In a running text you can use:

```
\color[red]{This will show up red.}
```

or:

```
\startcolor[red]
    This will show up red.
\stopcolor
```

In case you don't want the grouping you can use:

```
\directcolor[red]
```

You can even use:

```
\colored[r=0.5]{also red}
```

In which case an anonymous color is used. An ungrouped variant of this is:

```
\directcolored[r=0.5]
```

You will seldom use these direct variants, but they might come in handy when you write macros yourself where extra grouping starts interfering. In fact, it often makes sense to use a bit more abstraction:

```
\definehighlight
  [important]
  [color=red]

First \highlight[important]{or} second \important {or} third.
```

This gives:

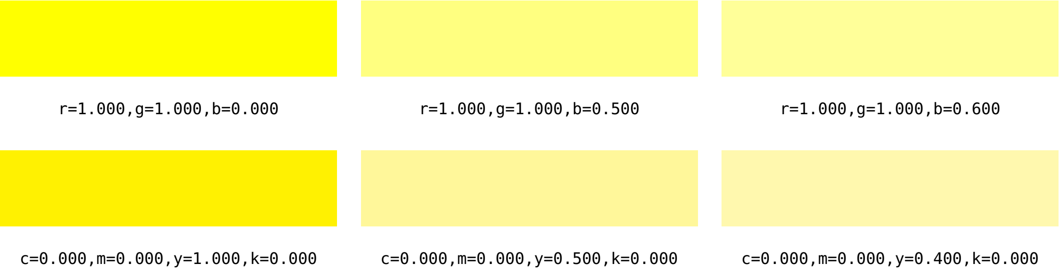
First or second or third..

1.3 Using cmyk or rgb

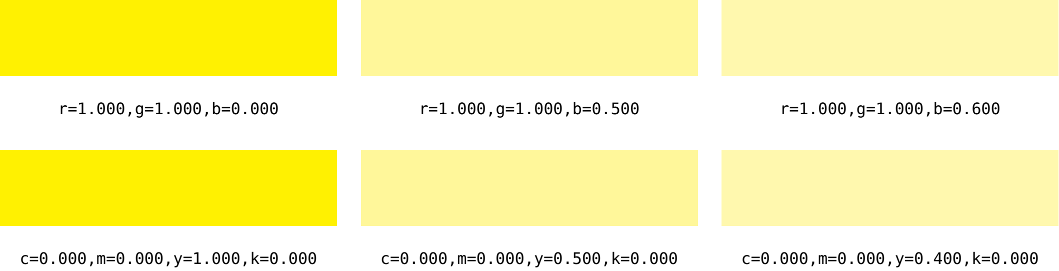
When you compare colors in different color spaces, you need to be aware of the fact that when a black component is used in cmyk, conversion to rgb might give the same results but going back from that to cmyk will look different from the original. Also, cmyk colors are often tuned for specific paper.

```
\definecolor[demo:rgb:1][r=1.0,g=1.0]
\definecolor[demo:rgb:2][r=1.0,g=1.0,b=0.5]
\definecolor[demo:rgb:3][r=1.0,g=1.0,b=0.6]
\definecolor[demo:cmy:1][y=1.0]
\definecolor[demo:cmy:2][y=0.5]
\definecolor[demo:cmy:3][y=0.4]
```

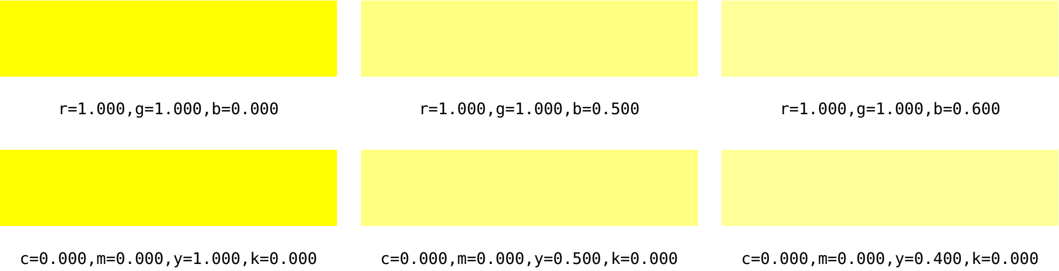
In these definitions we have no black component. In figure 1.1 we see how these colors translate to the other color spaces.


r=1.000,g=1.000,b=0.000
r=1.000,g=1.000,b=0.500
r=1.000,g=1.000,b=0.600
c=0.000,m=0.000,y=1.000,k=0.000
c=0.000,m=0.000,y=0.500,k=0.000
c=0.000,m=0.000,y=0.400,k=0.000

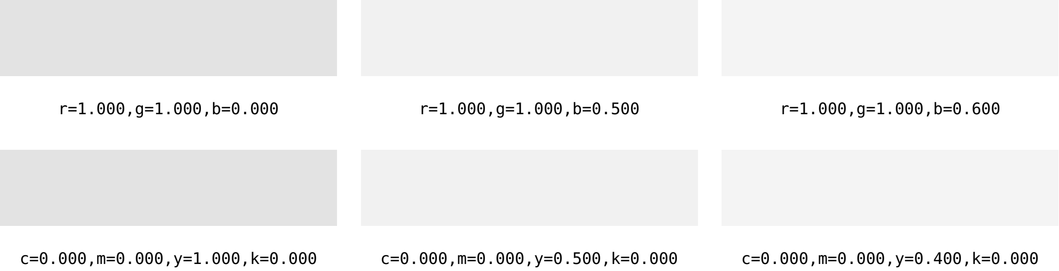
Both rgb and cmyk enabled


r=1.000,g=1.000,b=0.000
r=1.000,g=1.000,b=0.500
r=1.000,g=1.000,b=0.600
c=0.000,m=0.000,y=1.000,k=0.000
c=0.000,m=0.000,y=0.500,k=0.000
c=0.000,m=0.000,y=0.400,k=0.000

Only cmyk enabled.


r=1.000,g=1.000,b=0.000
r=1.000,g=1.000,b=0.500
r=1.000,g=1.000,b=0.600
c=0.000,m=0.000,y=1.000,k=0.000
c=0.000,m=0.000,y=0.500,k=0.000
c=0.000,m=0.000,y=0.400,k=0.000

Only rgb enabled.


r=1.000,g=1.000,b=0.000
r=1.000,g=1.000,b=0.500
r=1.000,g=1.000,b=0.600
c=0.000,m=0.000,y=1.000,k=0.000
c=0.000,m=0.000,y=0.500,k=0.000
c=0.000,m=0.000,y=0.400,k=0.000

Both rgb and cmyk disabled.

Figure 1.1 What happens when we disable color spaces.

```
\definecolor[demo:rgb:1][r=0.5,g=0.6,b=0.7]
\definecolor[demo:rgb:2][r=0.5,g=0.6,b=0.7]
\definecolor[demo:rgb:3][r=0.5,g=0.6,b=0.7]
\definecolor[demo:cmy:1][c=0.5,m=0.4,y=0.3]
\definecolor[demo:cmy:2][c=0.4,m=0.3,y=0.2,k=0.1]
\definecolor[demo:cmy:3][c=0.3,m=0.2,y=0.1,k=0.2]
```

When we define the colors as above, you can see a difference between the rgb and cmyk values, but also between a black component versus black distributed over the colorants. This is seen best in figure 1.2 when we compare the first and third colors alongside. In figure 1.3 you see the whole repertoire.

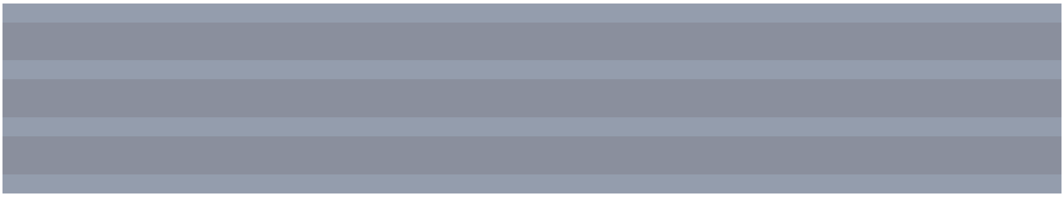


Figure 1.2 The impact of black on a cmyk color.

1.4 Conversion

A conversion to gray happens when rgb and cmyk are both disabled. The following setting forces conversion. It disables both rgb and cmyk:

```
\setupcolors
  [conversion=always]
```

The default setting is `yes` which means that colors will be reduced to gray in the backend. This is an optimization which can result in slightly smaller output:

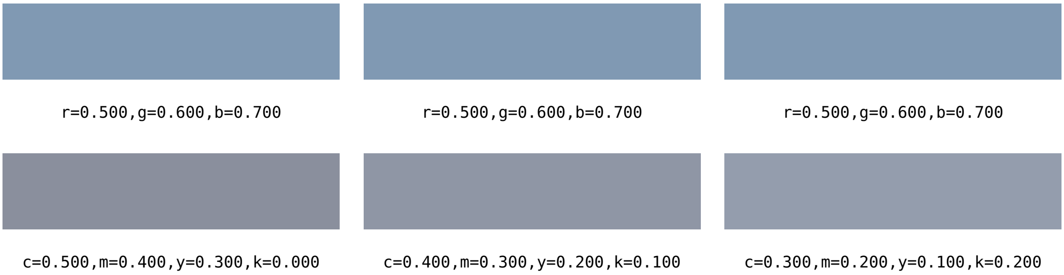
	#	**pdf sequence**
cmyk	23	`0 0 0 0.5 k 0 0 0 0.5 K`
rgb	29	`0.5 0.5 0.5 rg 0.5 0.5 0.5 RG`
gray	11	`0.5 g 0.5 G`

The conversion to gray is controlled by:

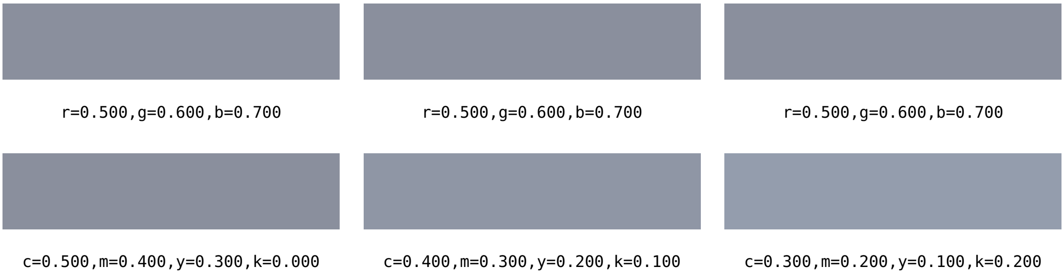
```
\setupcolors
  [factor=yes]
```

Like `conversion` the `factor` is a global setting. You can play with the factor values. The default (`yes`) uses the factors used by color television:

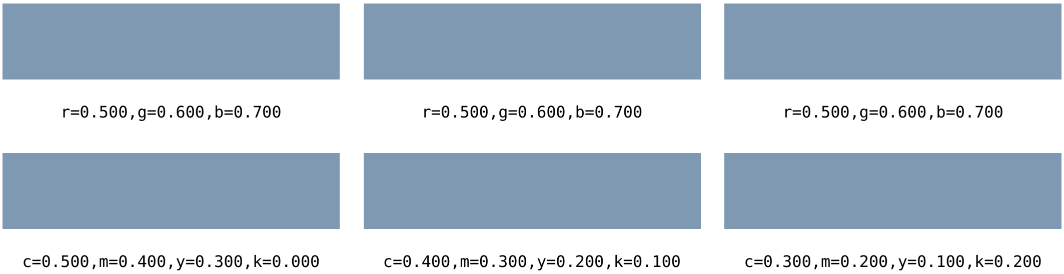
$$s = 0.30r + 0.59g + 0.11b$$


r=0.500,g=0.600,b=0.700
r=0.500,g=0.600,b=0.700
r=0.500,g=0.600,b=0.700
c=0.500,m=0.400,y=0.300,k=0.000
c=0.400,m=0.300,y=0.200,k=0.100
c=0.300,m=0.200,y=0.100,k=0.200

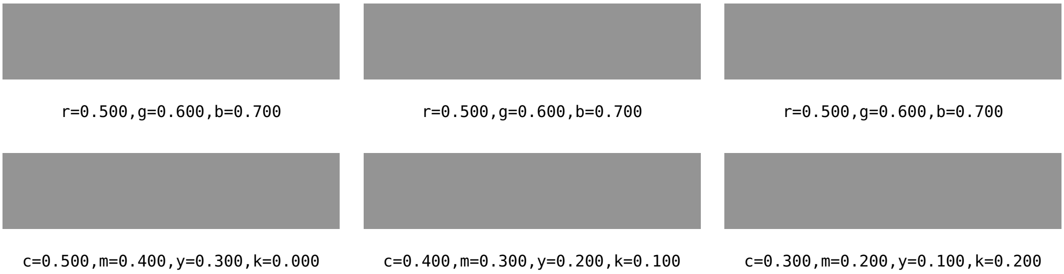
Both rgb and cmyk enabled


r=0.500,g=0.600,b=0.700
r=0.500,g=0.600,b=0.700
r=0.500,g=0.600,b=0.700
c=0.500,m=0.400,y=0.300,k=0.000
c=0.400,m=0.300,y=0.200,k=0.100
c=0.300,m=0.200,y=0.100,k=0.200

Only cmyk enabled.


r=0.500,g=0.600,b=0.700
r=0.500,g=0.600,b=0.700
r=0.500,g=0.600,b=0.700
c=0.500,m=0.400,y=0.300,k=0.000
c=0.400,m=0.300,y=0.200,k=0.100
c=0.300,m=0.200,y=0.100,k=0.200

Only rgb enabled.


r=0.500,g=0.600,b=0.700
r=0.500,g=0.600,b=0.700
r=0.500,g=0.600,b=0.700
c=0.500,m=0.400,y=0.300,k=0.000
c=0.400,m=0.300,y=0.200,k=0.100
c=0.300,m=0.200,y=0.100,k=0.200

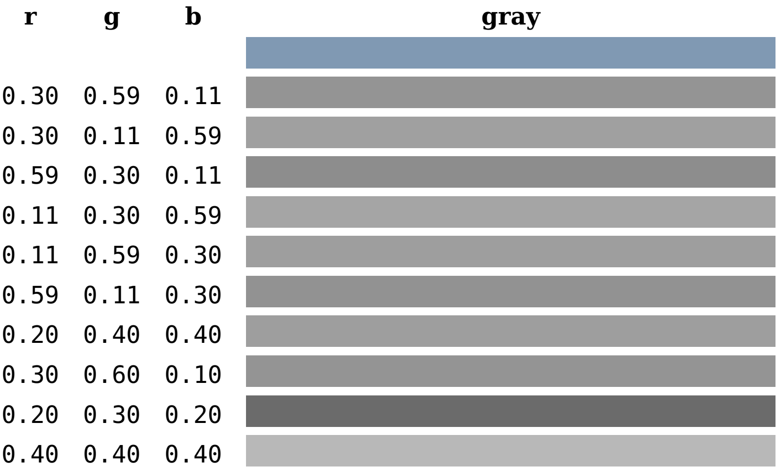
Both rgb and cmyk disabled.

Figure 1.3 What happens when we disable color spaces (black component).

In figure 1.4 we demonstrate what happens when you use different values. Normally you won't change the defaults but for experimenting we do provide the option:

```
\setupcolors
  [factor=0.20:0.40:0.40]
```

There is one pitfall. Colors are finalized per page and as this is a backend feature the value current when a page is shipped out is used. An exception are MetaPost graphics, as they have local resources and are finalized immediately. This is hardly a limitation because one will never set these numbers in the middle of a document.



r	**g**	**b**	**gray**
			▨
0.30	0.59	0.11	▨
0.30	0.11	0.59	▨
0.59	0.30	0.11	▨
0.11	0.30	0.59	▨
0.11	0.59	0.30	▨
0.59	0.11	0.30	▨
0.20	0.40	0.40	▨
0.30	0.60	0.10	▨
0.20	0.30	0.20	▨
0.40	0.40	0.40	▨

Figure 1.4 Color to gray conversion using factors.

1.5 Definitions

The mostly used color definition command is `\definecolor`. Here we define the primary colors:

```
\definecolor [red]     [r=1]
\definecolor [green]   [g=1]
\definecolor [blue]    [b=1]
\definecolor [yellow]  [y=1]
\definecolor [magenta] [m=1]
\definecolor [cyan]    [c=1]
```

These can be visualized as follows:

```
\showcolorcomponents[red,green,blue,yellow,magenta,cyan,black]
```

```
color         name     transparency  specification

white black   red                    r=1.000,g=0.000,b=0.000
white black   green                  r=0.000,g=1.000,b=0.000
white black   blue                   r=0.000,g=0.000,b=1.000
white black   yellow                 c=0.000,m=0.000,y=1.000,k=0.000
```

white black	magenta	c=0.000,m=1.000,y=0.000,k=0.000
white black	cyan	c=1.000,m=0.000,y=0.000,k=0.000
white	black	s=0.000

Transparency is included in these tables but is, as already noted, in fact independent. It can be defined with a color:

```
\definecolor [t:red]   [r=1,a=1,t=.5]
\definecolor [t:green] [g=1,a=1,t=.5]
\definecolor [t:blue]  [b=1,a=1,t=.5]
```

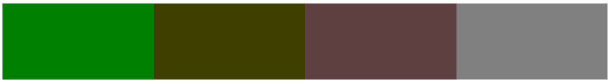
This time the transparency values show up too:

color	name	transparency	specification
white black	t:red	a=1.000,t=0.500	r=1.000,g=0.000,b=0.000
white black	t:green	a=1.000,t=0.500	r=0.000,g=1.000,b=0.000
white black	t:blue	a=1.000,t=0.500	r=0.000,g=0.000,b=1.000

Because transparency is separated from color, we can define transparent behaviour as follows:

```
\definecolor[t:only] [a=1,t=.5]

\dontleavehmode
\blackrule[width=4cm,height=1cm,color=darkgreen]%
\hskip-2cm
\color[t:only]{\blackrule[width=4cm,height=1cm,color=darkred]}%
\hskip-2cm
\color[t:only]{\blackrule[width=4cm,height=1cm]}
```

We skip back to create an overlay, so we get:



In the section about transparency a bit more will be said about the relation between color and transparencies and how to cheat.

As soon as you need to typeset something for professional printing, spot colors will show up so they are supported too. A spot color is not really a color but related to the substance that gets put on the paper. This can be ink but also something metallic, like silver, gold or some texture. In these cases we need something to represent it when not printed on a suitable device so again we end up with a color. This is reflected in the way spot colors are set up.

```
\definecolor     [parentspot]              [r=.5,g=.2,b=.8]
\definespotcolor [childspot-1] [parentspot] [p=.7,e=fancy]
```

```
\definespotcolor [childspot-2] [parentspot] [p=.4]
```

The three colors, two of them are spot colors, show up as follows:

color	name	transparency	specification
white black	parentspot		r=0.500,g=0.200,b=0.800
white black	childspot-1		p=0.700
white black	childspot-2		p=0.400

The `p` is comparable to the `s` in gray scales. The `e` parameter can be used to specify a name for the color. In the pdf file that name will become the separation name (a popular commercial naming scheme is Pantone).

A combination of spotcolor is called a multitone color. These are defined as follows (we also define a few spotcolors and use transparency):

```
\definespotcolor [spotone]   [darkred]   [p=1]
\definespotcolor [spottwo]   [darkgreen] [p=1]

\definespotcolor [spotone-t] [darkred]   [a=1,t=.5]
\definespotcolor [spottwo-t] [darkgreen] [a=1,t=.5]

\definemultitonecolor
    [whatever]
    [spotone=.5,spottwo=.5]
    [b=.5]

\definemultitonecolor
    [whatever-t]
    [spotone=.5,spottwo=.5]
    [b=.5]
    [a=1,t=.5]
```

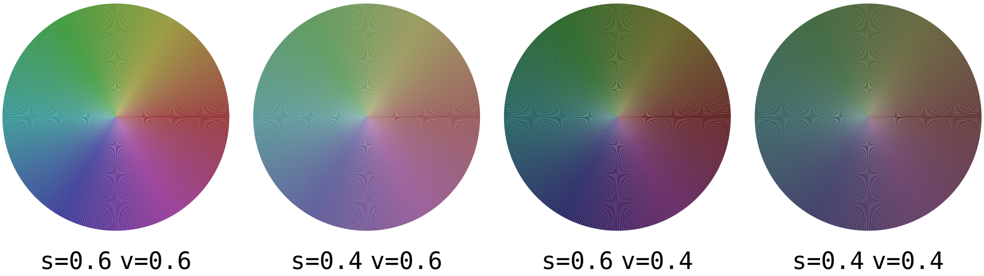
color	name	transparency	specification
white black	spotone		p=1.000
white black	spottwo		p=1.000
white black	spotone-t	a=1.000,t=0.500	p=1.000
white black	spottwo-t	a=1.000,t=0.500	p=1.000
white black	whatever		p=.5,.5
white black	whatever-t	a=1.000,t=0.500	p=.5,.5

Transparencies combine as follows:

```
\blackrule[width=3cm,height=1cm,color=spotone-t]\hskip-1.5cm
\blackrule[width=3cm,height=1cm,color=spotone-t]
```

In case you want to specify colors in the hsv color space, you can do that too. The hue parameter (`h`) is in degrees and runs from 0 upto 360 (larger values get divided). The saturation (`s`) and value (`v`) parameters run from 0 to 1. The `v` parameter is mandate. In figure 1.5 we show what the last two variables do.

```
\definecolor[somecolor][h=125,s=0.5,v=0.8]
```



`s=0.6 v=0.6` `s=0.4 v=0.6` `s=0.6 v=0.4` `s=0.4 v=0.4`

Figure 1.5 Four hsv color circle running from 0 to 360 degrees, with zero at the right.

If you need to use hexadecimal color specifications you can use these definitions:

```
\definecolor[mycolor][x=4477AA]
\definecolor[mycolor][h=4477AA]
\definecolor[mycolor][x=66]
\definecolor[mycolor][#4477AA]
```

The # is normally not accepted in TEX source code but when you get the specification from elsewhere (e.g. xml) it can be convenient.

1.6 Freezing colors

We can clone colors and thereby overload color dynamically. You can however freeze colors via the setup option `expansion`.

```
\definecolor[green]   [r=.5]{({\green green ->   red})}
\definecolor[green]   [g=.5]{({\green green -> green})}
\definecolor[green]   [blue]{({\green green ->  blue})}
\definecolor[blue]     [red]{({\green green ->   red})}
\setupcolors[expansion=yes]%
\definecolor[blue]     [red]%
\definecolor[green]   [blue]%
\definecolor[blue]    [r=.5]{({\green green ->  blue})}
```

(green -> red) (green -> green) (green -> blue) (green -> red) (green -> blue)

1.7 Color groups

Nowadays we seldom use colorgroups but they are still supported. Groups are collections of distinctive colors, something we needed in projects where many graphics had to be made and consistency between text and image colors was important. The groups can be translated into similar collections in drawing programs used at that time.

```
\definecolorgroup
  [redish]
  [1.00:0.90:0.90, % 1
   1.00:0.80:0.80, % 2
   1.00:0.70:0.70, % 3
   1.00:0.55:0.55, % 4
   1.00:0.40:0.40, % 5
   1.00:0.25:0.25, % 6
   1.00:0.15:0.15, % 7
   0.90:0.00:0.00] % 8
```

The redish color is called by number:

```
\blackrule[width=3cm,height=1cm,depth=0pt,color=redish:1]\quad
\blackrule[width=3cm,height=1cm,depth=0pt,color=redish:2]\quad
\blackrule[width=3cm,height=1cm,depth=0pt,color=redish:3]
```

The number of elements is normally limited and eight is about what is useful and still distinguishes good enough when printed in black and white.

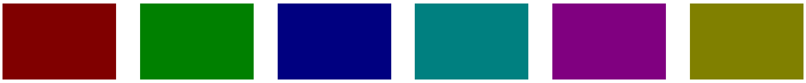
1.8 Palets

Color palets are handy when you want to use a set of (named) colors but also want to switch efficiently between different definitions:

```
\definepalet
  [standard]
  [darkred=darkcyan,
   darkgreen=darkmagenta,
   darkblue=darkyellow]
```

The `\setuppalet` commands switches to a palet. When a requested color is not part of a palet, a regular lookup happens. This is used as:

```
\blackrule[width=15mm,height=10mm,depth=0mm,color=darkred]\quad
\blackrule[width=15mm,height=10mm,depth=0mm,color=darkgreen]\quad
\blackrule[width=15mm,height=10mm,depth=0mm,color=darkblue]\quad
\setuppalet[standard]%
\blackrule[width=15mm,height=10mm,depth=0mm,color=darkred]\quad
\blackrule[width=15mm,height=10mm,depth=0mm,color=darkgreen]\quad
\blackrule[width=15mm,height=10mm,depth=0mm,color=darkblue]
```



Here we use color names but often you end up with more symbolic names:

```
\definepalet
  [standard]
  [important=darkred,
   notabene=darkgreen,
   warning=darkyellow]
```

As with the regular color commands, the palet mechanism is an old one but it is well integrated. Instead of inheriting you can also use definitions:

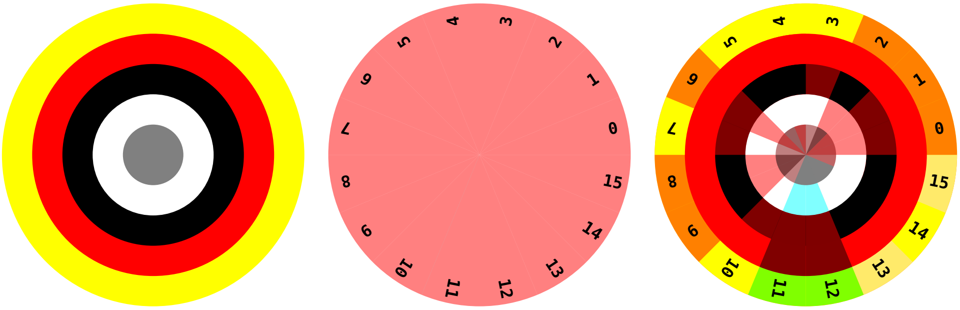
```
\definepalet
  [standard]
  [important={r=.5},
   notabene={g=.5},
   warning={r=.5,g=.5}]
```

1.9 Transparency

We already discussed transparency as part of colors. In most cases we will choose type normal (or 1) as transparency type, but there are more:

0	none
1	normal
2	multiply
3	screen
4	overlay
5	softlight
6	hardlight
7	colordodge
8	colorburn
9	darken
10	lighten
11	difference
12	exclusion
13	hue
14	saturation
15	color
16	luminosity

In figure 1.6 we compare these variants. Not all are as effective as their effect depends on several factors. You can read more about it in the pdf specification.



backgrounds used

transparencies used

transparency overlayed

Figure 1.6 The differences between the transparency options. The center is explicitly filled with white.

Colors and transparencies are coupled by definitions. We will explain this by some examples. When we say:

```
\definecolor[clr1][r=.5]
```

A non-transparent color is defined and when we say:

```
\definecolor[clr2][g=.5,a=1,t=.5]
```

We defined a color with a transparency. However, color and transparency get separated attributes. So when we nest them as in:

```
\color[clr1]{\bf RED   \color[clr2] {GREEN}}
\color[clr2]{\bf GREEN \color[clr1] {RED}  }
```

we get:

RED GREEN
GREEN RED

The transparency of the outer color is also applied to the inner color. If you don't want that, you explicitly need to set them:

```
\definecolor[clr3][b=.5,a=1,t=1]
```

```
\color[clr1]{\bf RED   \color[clr2] {GREEN} \color[clr3]{BLUE} }
\color[clr2]{\bf GREEN \color[clr1] {RED}   \color[clr2]{GREEN}}
\color[clr3]{\bf BLUE  \color[clr1] {RED}   \color[clr2]{GREEN}}
```

we get:

RED GREEN BLUE
GREEN RED GREEN
BLUE RED GREEN

If you define a transparent-only color, you get transparent black:

```
\definecolor[clr4][a=1,t=.5]
```

So:

```
\color[clr1]{\bf RED    \color[clr4] {RED}}
\color[clr4]{\bf BLACK  \color[clr1] {RED}}
```

gives:

RED RED
BLACK RED

In addition to the already discussed definers and setters we also have a few special ones. Personally I never needed them but they are the for completeness.

```
\definetransparency[tsp1][a=1,t=.5]
```

We apply this to some text:

```
\color      [clr1]{\bf RED   \transparent[tsp1] {RED}   }
\transparent[tsp1]{\bf BLACK \color      [clr1] {RED}   }
\transparent[tsp1]{\bf BLACK \transparent[reset]{BLACK} }
```

and get:

RED RED
BLACK RED
BLACK BLACK

We can also only switch color:

```
\color[clr1]{\bf RED \color    [clr2] {GREEN}}
\color[clr1]{\bf RED \coloronly[clr2] {GREEN}}
```

So the second line has no transparency:

RED GREEN
RED GREEN

The `\starttransparent` and `\startcoloronly` commands are the complements of `\transparent` and `\coloronly`.

1.10 Interpolation

You can define intermediate colors in a way comparable with MetaPost `.5[red,green]` kind of specifications. Here are some examples:

```
\definecolor [mycolor1] [.5(red,green)]
\definecolor [mycolor2] [.8(red,green)]
\definecolor [mycolor3] [.4(red,white)]
\definecolor [mycolor4] [.4(white,red)]

\showcolorcomponents[red,green,mycolor1,mycolor2,mycolor3,mycolor4]
```

```
color          name      transparency  specification

white black    red                     r=1.000,g=0.000,b=0.000
white black    green                   r=0.000,g=1.000,b=0.000
white black    mycolor1                r=0.500,g=0.500,b=0.000
white black    mycolor2                r=0.200,g=0.800,b=0.000
white black    mycolor3                r=1.000,g=0.400,b=0.400
white black    mycolor4                s=0.720
```

An older method, still available is:

```
\defineintermediatecolor[mycolor5][0.5,red,green]
```

A variation on this are complementary colors:

```
\definecolor[mycolor1][.5(blue,red)]
\definecolor[mycolor2][-.5(blue,red)]
\definecolor[mycolor3][-(blue)]
\definecolor[mycolor4][-(red)]

\showcolorcomponents[blue,red,mycolor1,mycolor2,mycolor3,mycolor4]
```

```
color          name      transparency  specification

white black    blue                    r=0.000,g=0.000,b=1.000
white black    red                     r=1.000,g=0.000,b=0.000
white black    mycolor1                r=0.500,g=0.000,b=0.500
white black    mycolor2                r=0.500,g=1.000,b=0.500
white black    mycolor3                r=1.000,g=1.000,b=0.000
white black    mycolor4                r=0.000,g=1.000,b=1.000
```

1.11 PDF

Although it is not perfect, pdf evolved in such a way that it will stay around for a while. One reason is that it has become a standard, or more precisely a set of standards. Depending on what variant you choose color support is limited.

format	**gray**	**rgb**	**cmyk**	**spot**	**transparency**
`pdf/a-1a:2005`	⋆	⋆	⋆	⋆	

`pdf/a-1b:2005`	⋆	⋆	⋆	⋆	
`pdf/a-2a`	⋆	⋆	⋆	⋆	⋆
`pdf/a-2b`	⋆	⋆	⋆	⋆	⋆
`pdf/a-2u`	⋆	⋆	⋆	⋆	⋆
`pdf/a-3a`	⋆	⋆	⋆	⋆	⋆
`pdf/a-3b`	⋆	⋆	⋆	⋆	⋆
`pdf/a-3u`	⋆	⋆	⋆	⋆	⋆
`pdf/ua-1`	⋆	⋆	⋆	⋆	⋆
`pdf/x-1a:2001`	⋆		⋆	⋆	
`pdf/x-1a:2003`	⋆		⋆	⋆	
`pdf/x-3:2002`	⋆	⋆	⋆	⋆	
`pdf/x-3:2003`	⋆	⋆	⋆	⋆	
`pdf/x-4`	⋆	⋆	⋆	⋆	⋆
`pdf/x-4p`	⋆	⋆	⋆	⋆	⋆
`pdf/x-5g`	⋆	⋆	⋆	⋆	⋆
`pdf/x-5n`	⋆	⋆	⋆	⋆	⋆
`pdf/x-5pg`	⋆	⋆	⋆	⋆	⋆

When you have set the `format` with `\setupbackend` to one of the known formats mentioned in the previous table, the color conversions will automatically kick in.

1.12 Unboxing

This paragraph is somewhat complex, so skip it when you don't feel comfortable with the subject or when you've never seen low level ConTEXt code.

Colors are implemented using attributes. Attributes behave like fonts. This means that they are kind of frozen once material is boxed. Consider that we define a box as follows:

```
\setbox0\hbox{\bf default {\darkred red \darkgreen green} default}
```

What do you expect to come out the next code? In MkII the 'default' inside the box will be colored yellow but the internal red and and green words will keep their color.

```
\bf default {\darkyellow yellow {\box0} yellow} default
```

This is what we get in MkIV: **default** **yellow** **default** **red** **green** **default** **yellow** **default**

When we use fonts switches we don't expect the content of the box to change. So, in the following the 'default' texts will *not* become bold.

```
\setbox0\hbox{default {\sl slanted \bi bold italic} default}
```

```
default {\bf bold {\box0} bold} default
```

Now we get: default **bold** default *slanted* ***bold italic*** default **bold** default.

Redoing a box with a new font is sort of tricky as by then all kind of manipulations have been applied and the original inputs is long gone. But colors are easier to deal with and therefore in MkIV we have a trick to make sure the outer color gets applied to the box:

```
default {\bf \darkyellow yellow {\attributedbox0} yellow} default
```

So, we get: default **yellow yellow** default. In MkIV you need to enable inheritance first with:

```
\enabledirectives % only mkiv
  [attributes.inheritance]
```

There is also an `\attributedcopy` macro. These macros signal the attribute resolver that this box is to be treated special.

In MkII we have a similar situation which is why we had the option (only used deep down in ConTEXt) to encapsulate a bunch of code with

```
\startregistercolor[foregroundcolor]
some macro code ... here foregroundcolor is applied ... more code
\stopregistercolor
```

This is for instance used in the `\framed` macro. First we package the content, foregroundcolor is not yet applied because the injected specials of literals can interfere badly, but by registering the colors the nested color calls are tricked into thinking that preceding and following content is colored. When packaged, we apply backgrounds, frames, and foregroundcolor to the whole result. Because nested colors were aware of the foregroundcolor they have properly reverted to this color when needed.

In MkIV the situation is reversed. Here we definitely need to set the foregroundcolor because otherwise attributes are not set. This is no problem because contrary to MkII colors don't interfere (no extra nodes). We could have flushed the framed content using `\attributedbox`, but we don't want to enable inheritance by default because it comes with some overhead.

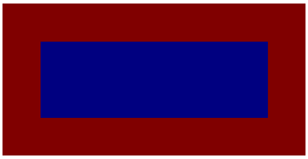
The `\attributedbox` command is considered obsolete. In LMTX there is a `\recolorbox` command that recolors a box. Because these commands are probably never needed it made more sense to move the burden to a specific command than to add additional overhead to the whole color mechanism. My experience is that unboxing and copying is very rare in ConTEXt.

1.13 Color intents

If we do this:

```
\startoverlay
    {\blackrule[color=darkred, height=2cm,width=4cm]}
    {\blackrule[color=darkblue,height=1cm,width=3cm]}
\stopoverlay
```

we get:



The blue rectangle is drawn on top of the red one. In print, normally the printing engine will make sure that there is no red below the blue. In case of transparent colors this is somewhat tricky because then we definitely want to see part of what lays below.

You can control this process with the following commands:

```
\setupcolors
  [intent=...]
```

The default setting is `none` but you can set the intent to `overprint` or `knockout` as well. In a running text you can use the following commands:

```
\startcolorintent[overprint|knockout]
    ...
\stopcolorintent

\startoverprint
    ...
\stopoverprint

\startknockout
    ...
\stopknockout
```

In practice you will probably seldom need to deal with this so best leave the defaults as they are.

1.14 Collections

Collections are predefined sets of colors. You find them in the files `colo-imp-*.mkiv` and you can make such files yourself. When you define a color a command is generated by default. When you load a collection, there is a danger that you redefine commands unintended. For that reason most collections are wrapped in:

```
\startprotectedcolors
    % definitions
```

```
\stopprotectedcolors
```

This prevents commands being defined and assumes that colors are accessed by using the `color` parameter of setup commands or in the text with:

```
\color[somecolor]{this gets colored}

\startcolor[somecolor]
    this gets colored
\stopcolorintent
```

The default set (`rgb`) is already preloaded with:

```
\usecolors[rgb] % preloaded
```

You can get a list of colors with

```
\showcolor[rgb]
```

This generates the list:

■	■	0.000	0.000	black
■	■	0.110	0.000 0.000 1.000	blue
■	■	0.700	0.000 1.000 1.000	cyan
■	■	0.066	0.000 0.000 0.600	darkblue
■	■	0.280	0.000 0.400 0.400	darkcyan
■	■	0.400	0.400	darkgray
■	■	0.354	0.000 0.600 0.000	darkgreen
■	■	0.164	0.400 0.000 0.400	darkmagenta
■	■	0.238	0.400 0.200 0.000	darkorange
■	■	0.180	0.600 0.000 0.000	darkred
■	■	0.356	0.400 0.400 0.000	darkyellow
■	■	0.900	0.900	gray
■	■	0.100	0.100	gray-1
■	■	0.200	0.200	gray-2
■	■	0.300	0.300	gray-3
■	■	0.400	0.400	gray-4
■	■	0.500	0.500	gray-5
■	■	0.600	0.600	gray-6
■	■	0.700	0.700	gray-7
■	■	0.800	0.800	gray-8
■	■	0.900	0.900	gray-9
■	■	0.590	0.000 1.000 0.000	green
■	■	0.555	0.500 0.500 1.000	lightblue
■	■	0.850	0.500 1.000 1.000	lightcyan
■	■	0.850	0.850	lightgray
■	■	0.795	0.500 1.000 0.500	lightgreen

▨	🟪	0.705	1.000 0.500 1.000	lightmagenta
▨	🟧	0.595	1.000 0.500 0.000	lightorange
▨	🟥	0.650	1.000 0.500 0.500	lightred
▨	🟨	0.945	1.000 1.000 0.500	lightyellow
▨	🟪	0.410	1.000 0.000 1.000	magenta
▨	🟦	0.088	0.000 0.000 0.800	middleblue
▨	🟦	0.420	0.000 0.600 0.600	middlecyan
▨	▨	0.625	0.625	middlegray
▨	🟩	0.472	0.000 0.800 0.000	middlegreen
▨	🟪	0.246	0.600 0.000 0.600	middlemagenta
▨	🟫	0.357	0.600 0.300 0.000	middleorange
▨	🟥	0.240	0.800 0.000 0.000	middlered
▨	🟨	0.534	0.600 0.600 0.000	middleyellow
▨	🟧	0.595	1.000 0.500 0.000	orange
▨	▨	0.750	0.750	palegray
▨	🟥	0.300	1.000 0.000 0.000	red
		1.000	1.000	white
▨	🟨	0.890	1.000 1.000 0.000	yellow

These are the collections shipped with ConTEXt. Some names are

`crayola`	crayon colors
`dem`	a demo set of groups and palets
`ema`	an old coming from an Emacs user
`rainbow`	a series of color groups by Alan
`ral`	a set often used in industry (from Germany)
`rgb`	a basic set of colors defined in the rgb color space
`x11`	(most of the) standard X11 rgb colors

You can look in these files to see what gets defined. Even if you don't use them they might be illustrative,

1.15 Text color

Setting the color of the running text is done with:

```
\setupcolors
  [textcolor=darkgray]
```

If needed you can also set the `pagecolormodel` there but its default value is `none` which means that it will obey the global settings.

1.16 Tikz

In case you use the TikZ graphical subsystem you need to be aware of the the fact that its color support is more geared towards LATEX. There is glue code that binds the ConTEXt color system to its internal representation but there can still be problems. For instance, not all color systems are supported so ConTEXt will try to remap, but only when it knows that it has to do so. You can best not mix colorspaces when you use TikZ. If you really want (and there is no real reason to do so) you can say:

```
\enabledirectives[colors.pgf]
```

and then (at the cost of some extra overhead) define colors as:

```
\definecolor[pgfcolora][blue!50!green]
\definecolor[pgfcolorb][red!50!blue]
```

1.17 Implementation details

The low level implementation of colors in MkIV is fundamentally different from MkII. In MkIV something like this happens:

```
one \color[red]{two} three
```

becomes (with grouping):

```
one {<start color: red>two<stop color>} three
```

the start and stop points are in fact injections in the input: a special (for dvi) or literals (for pdf) is inserted that turns the color on and off, but also information is carried along about the state of color, so that we can properly nest as well as pick up the current color after a page break. We never had real problems with this mechanism but one had to keep in mind that injections like this could interfere with typesetting. This mechanism didn't rely on the engine for housekeeping, all was done at the TEX end using so called marks.

In MkIV we use attributes. This means that the sequence now looks like:

```
one {<set color attribute to red>two} three
```

The actual handling of color happens when a page is shipped out and there is no interference with typesetting. The work is mostly done in Lua.

Colorspaces (rgb, cmyk, spot) were already supported in MkII and of course still are in MkIV. However, the colorspace is now a more independent property. At some point in MkII we also implemented transparency as a property of a color. In MkIV transparency is still defined with a color but handled independently. This means that where in MkII color is just one axis, in MkIV we have three: colorspace (model), color and transparency. This of course has a bit of a performance and memory hit, but in practice a user won't notice it.

1.18 Grouping

The `\color` and `\startcolor` command group their arguments. There might be cases where this interferes with your intentions, for instance when you want to set some variable and use its value later on.

```
1 \scratchcounter=1
plus
1 \advance \scratchcounter by 1
equals
\the\scratchcounter
```

The summation works out okay: **1 plus 1 equals 2**.

```
\color[darkblue]{1 \scratchcounter=1}
plus
\color[darkblue]{1 \advance \scratchcounter by 1}
equals
\color[darkgreen]{\the\scratchcounter}
```

Here the final result depends on the value of `\scratchcounter`: **1 plus 1 equals 3**.

```
\start
    \pushcolor[darkblue]1 \scratchcounter=1 \popcolor
    plus
    \pushcolor[darkblue]1 \advance \scratchcounter by 1 \popcolor
    equals
    \pushcolor[darkgreen]\the\scratchcounter \popcolor
\stop
```

Here we get: **1 plus 1 equals 2**. The `\pushcolor` and `\popcolor` commands can be used nested which give a bot of overhead. The `\savecolor` and `\restorecolor` commands are variants that don't stack. They are a bit more efficient but if you use them nested you probably also will use some grouping. Where the push–pop pair needs to be matched, the save–restore pair doesn't impose that restriction.

1.19 Commands

There are quite some commands that relate to colors but you probably only need `\definecolor`, `\color` and `\startcolor ...\stopcolor`. Here we show the complete list. Some commands are redundant, for instance `\definenamedcolor` is the same as `\definecolor`.

```
\color {...}
        *
*   CONTENT
```

```
\colorcomponents
```

```
               1         2
\colored [..,..=..,..] {...}
1   inherits: \definecolor
2   CONTENT
```

```
           *
\colored {...}
*   CONTENT
```

```
             *
\coloronly {...}
*   CONTENT
```

```
\colorvalue
```

```
\comparecolorgroup
```

```
\comparepalet
```

```
                   *
\definecolor [..,..=..,..]
*   r  =  NUMBER
    g  =  NUMBER
    b  =  NUMBER
    c  =  NUMBER
    m  =  NUMBER
    y  =  NUMBER
    k  =  NUMBER
    h  =  NUMBER
    s  =  NUMBER
    v  =  NUMBER
    w  =  NUMBER
    x  =  NUMBER
    a  =
    t  =  NUMBER
```

```
                    1       2
\definecolorgroup [...] [x:y:z,..]
1   gray rgb cmyk spot    OPT
2   TRIPLET
```

```
\definecolor
```

```
                         *
\defineglobalcolor [..,..=..,..]
*   inherits: \definecolor
```

```
\defineglobalcolor
```

```
                              1           2
\defineintermediatecolor [...,...] [..,..=..,..]
                                         OPT
1   COLOR NUMBER
2   a  =  NUMBER
    t  =  NUMBER
```

```
                             1               2               3
\definemultitonecolor [..,..=..,..] [..,..=..,..] [..,..=..,..]
                                                        OPT
1   COLOR  =  NUMBER
2   inherits: \definecolor
3   inherits: \definespotcolor
```

```
                          *
\definenamedcolor [..,..=..,..]
*   inherits: \definecolor
```

```
\definenamedcolor
```

```
                    *
\definepalet [..,..=..,..]
*   NAME  =  COLOR
```

```
\definepalet
```

```
                            *
\defineprocesscolor [..,..=..,..]
*   inherits: \definecolor
```

```
                         *
\definespotcolor [..,..=..,..]
*   a  =
    t  =  NUMBER
    e  =  TEXT
    p  =  NUMBER
```

```
\definetransparency
```

```
                            *
\definetransparency [..,..=..,..]
*   a  =
    t  =  NUMBER
```

```
\definetransparency
```

```
\directcolor
```

```
                       *
\directcolored [..,..=..,..]
*   inherits: \definecolor
```

```
\directcolored
```

```
\doifblackelse
```

```
\doifcolor
```

```
\doifcolorelse
```

```
\doifdrawingblackelse
```

```
\doifelseblack
```

```
\doifelsecolor
```

```
\doifelsedrawingblack
```

```
\getpaletsize
```

```
                  *
\graycolor {...}
*   CONTENT
```

```
\grayvalue
```

```
\MPcolor
```

```
\MPcoloronly
```

```
\MPoptions
```

```
\MPtransparency
```

```
\negatecolorbox
```

```
\paletsize
```

```
\processcolorcomponents
```

```
\pushcolor ... \popcolor
```

```
\savecolor ... \restorecolor
```

```
                   *
\setcolormodell [...]
*   black bw gray rgb cmyk all none
```

```
\setupcolor
```

```
                     *
\setupcolors [..,..=..,..]
*   state           =  start stop
    spot            =  yes no
    expansion       =  yes no
    factor          =  yes no
    rgb             =  yes no
    cmyk            =  yes no
    conversion      =  yes no always
    pagecolormodel  =  auto none NAME
    textcolor       =  COLOR
    intent          =  overprint knockout none
```

```
\setuppalet
```

```
\showcolor
```

```
\showcolorbar
```

```
\showcolorcomponents
```

```
                        *
\showcolorgroup [...,...]
                       OPT
*   horizontal vertical number value name
```

```
\showcolorset
```

```
                  *
\showpalet [...,...]
                 OPT
*   horizontal vertical number value name
```

```
\startcolor ... \stopcolor
```

```
                    *
\startcolorintent [...] ... \stopcolorintent
*   knockout overprint none
```

```
\startcoloronly ... \stopcoloronly
```

```
\startcolorset ... \stopcolorset
```

```
\startcurrentcolor ... \stopcurrentcolor
```

```
\startknockout ... \stopknockout
```

```
\startoverprint ... \stopoverprint
```

```
\startprotectedcolors ... \stopprotectedcolors
```

```
\starttextcolor ... \stoptextcolor
```

```
\starttextcolorintent ... \stoptextcolorintent
```

```
\starttransparent ... \stoptransparent
```

```
\switchtocolor
```

```
\transparencycomponents
```

```
               *
\transparent {...}
*   CONTENT
```

```
\usecolors
```

2 Metafun

2.1 Defining and using

In MetaPost itself colors are defined as numbers or sets:

```
color     red  ; red  := (1,0,0) ;
cmykcolor cyan ; cyan := (1,0,0,0) ;
numeric   gray ; gray := 0.5 ;
```

You don't need much fantasy to see that this fits well in the data model of MetaPost. In fact, transparency could be represented as a `pair`. The disadvantage of having no generic color type is that you cannot mix them. In case you need to manipulate them, you can check the type:

```
if cmykcolor cyan : ... fi ;
```

because MetaFun is tightly integrated in ConTEXt you can refer to colors known at the TEX end by string. So,

```
string mycolor ; mycolor := "red" ;
```

and then:

```
fill fullcircle scaled 4cm withcolor mycolor ;
```

is quite okay. For completeness we also have `namedcolor` but it's not really needed:

```
fill fullcircle scaled 4cm withcolor namedcolor("red");
```

You can define spot colors too but normally you will refer to colors defined at the TEX end.

```
\startMPcode
    fill fullcircle scaled 3cm withcolor
        .5 * spotcolor("whatever",(.3,.4,.5)) ;
    fill fullcircle scaled 2cm withcolor
             spotcolor("whatever",(.3,.4,.5)) ;
    fill fullcircle scaled 1cm withcolor
             spotcolor("whatever",(.3,.4,.5)/2) ;
\stopMPcode
```

Multitones are defined as:

```
\startMPcode
    fill fullcircle scaled 3cm withcolor
        .5 * multitonecolor("whatever",(.3,.4,.5),(.5,.3,.4)) ;
    fill fullcircle scaled 2cm withcolor
             multitonecolor("whatever",(.3,.4,.5),(.5,.3,.4)) ;
    fill fullcircle scaled 1cm withcolor
             multitonecolor("whatever",(.3,.4,.5)/2,(.5,.3,.4)/2) ;
\stopMPcode
```

Some pdf renderers have problems with fractions of such colors and even display the wrong colors. So, in these examples the third alternative in the sets of three might be more robust than the first. The result is shown in figure 2.1.

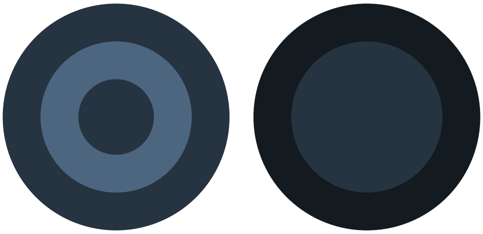


Figure 2.1 Spot and multitones directly defined in MetaFun.

2.2 Passing colors

Originally TeX and MetaPost were separated processes and even in LuaTeX they still are. There can be many independent MetaPost instances present, but always there is Lua as glue between them. In the early days of LuaTeX this was a one way channel: the MetaPost output is available at the TeX end in Lua as a table and properties are used to communicate extensions. In today's LuaTeX the MetaPost library has access to Lua itself so that gives us a channel to TeX, although with some limitations.

Say that we have a color defined as follows:

```
\definecolor[MyColor][r=.25,g=.50,b=.75]
```

We can apply this to a rule:

```
\blackrule[color=MyColor,width=3cm,height=1cm,depth=0cm]
```

From this we get:

In TeX (code) we can do this:

```
\startMPcode
    fill unitsquare xyscaled (3cm,1cm) withcolor \MPcolor {MyColor} ;
```

```
\stopMPcode
```

When the code is pushed to MetaPost the color gets expanded, in this case to `(0.25, 0.50, 0.75)` because we specified an rgb color but the other colorspaces are supported too.

Equally valid code is:

```
\startMPcode
    fill unitsquare xyscaled (3cm,1cm) withcolor "MyColor" ;
\stopMPcode
```

This is very un-MetaPost as naturally it can only deal with numerics for gray scales, triplets for rgb colors, and quadruplets for cmyk colors. In MetaFun (as present in ConTeXt MKIV) the `withcolor` operator also accepts a string, which is resolved to a color specification.

For the record we note that when you use transparent colors, a more complex specification gets passed with `\MPcolor` or resolved (via the string). The same is true for spot and multitone colors. It will be clear that when you want to assign a color to a variable you have to make sure the type matches. A rather safe way to define colors is:

```
def MyColor =
    \MPcolor{MyColor}
enddef ;
```

and because we can use strings, string variables are also an option.

2.3 Grouping

The reason for discussing these details is that there is a rather fundamental concept of grouping in TeX which can lead to unexpected side effects. The reason is that there is no grouping at the Lua end, unless one uses a kind of stack, and that in MetaPost grouping is an explicit feature.

```
\scratchcounter=123
\bgroup
    \scratchcounter=456
\egroup
```

After this TeX code is expanded the counter has value 123. In MetaPost you can do the following:

```
scratchcounter := 123 ;
```

```
\begingroup
    scratchcounter := 456 ;
\endgroup
```

but here the counter is 456 afterwards! You explicitly need to save a value:

```
scratchcounter := 123 ;
\begingroup
    save scratchcounter ;
    numeric scratchcounter ; % variables are numeric by default anyway
    scratchcounter := 456 ;
\endgroup
```

The difference perfectly makes sense if you think about the kind of applications TEX and MetaPost are used for. In Lua you can do this:

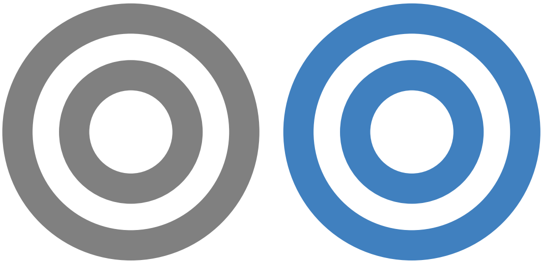
```
scratchcounter = 123
do
    local scratchcounter = 456
end
```

and in fact, a `then`, `else`, `while`, `repeat`, `do` and function body also behave this way.

So, what is the impact on colors? Imagine that you do this:

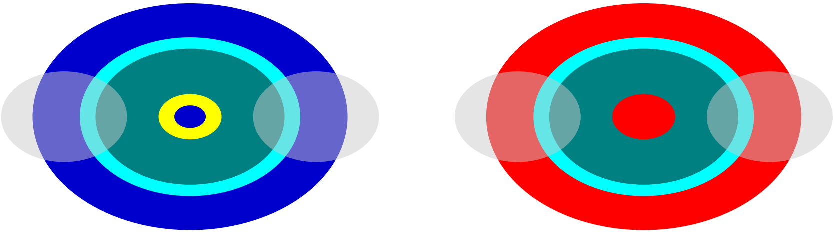
```
\bgroup
    \definecolor[MyColor][s=.5]
    \startMPcode
        pickup pencircle scaled 4mm ;
        draw fullcircle scaled 30mm withcolor \MPcolor{MyColor} ;
        draw fullcircle scaled 15mm withcolor "MyColor" ;
    \stopMPcode
\egroup
\quad
\startMPcode
    pickup pencircle scaled 4mm ;
    draw fullcircle scaled 30mm withcolor \MPcolor{MyColor} ;
    draw fullcircle scaled 15mm withcolor "MyColor" ;
\stopMPcode
```

We get the following colors:



Because `\MPcolor` is a TEX macro, its value gets expanded when the graphic is calculated. After the group (first graphic) the color is restored. But, in order to access the colors defined at the TEX end in MetaPost by name (using Lua) we need to make sure that a defined color is registered at that end. Before we could use the string accessor in MetaPost colors, this was never a real issue. We kept the values in a (global) Lua table which was good enough for the cases where we wanted to access color specifications, for instance for tracing. Such colors never changed in a document. But with the more dynamic MetaPost graphics we cannot do that: there is no way that MetaPost (or Lua) later on can know that the color was defined inside a group as clone. For daily usage it's enough to know that we have found a way around it in ConTEXt at neglectable overhead. In the rare case this mechanism fails, you can always revert to the `\MPcolor` method.

The following example was used when developing the string based color resolver. The complication was in getting the color palets resolved right without too much overhead. Again we demonstrate this because border cases might occur that are not catched (yet).

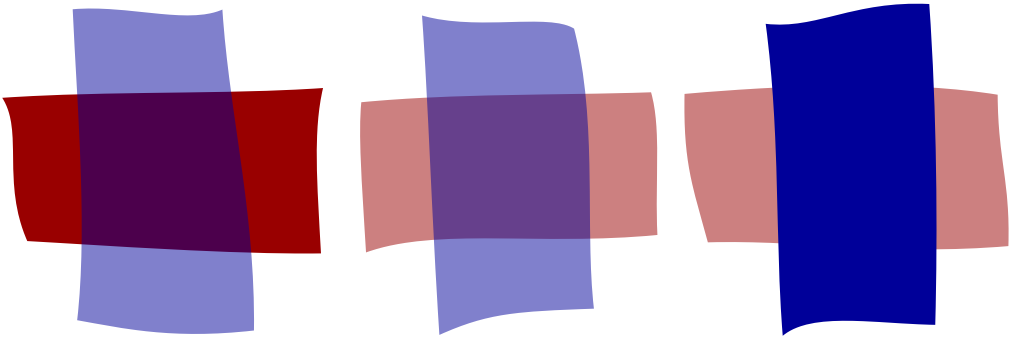


2.4 Transparency

Transparency is supported at the TEX end: either or not bound to colors. We already saw how to use colors, here's how to apply transparency:

```
\startMPcode
    fill fullsquare xyscaled (4cm,2cm) randomized 5mm
        withcolor "darkred" ;
    fill fullsquare xyscaled (2cm,4cm) randomized 5mm
        withcolor "darkblue" withtransparency ("normal",0.5) ;

    fill fullsquare xyscaled (4cm,2cm) randomized 5mm shifted (45mm,0)
        withcolor "darkred"  withtransparency ("normal",0.5) ;
    fill fullsquare xyscaled (2cm,4cm) randomized 5mm shifted (45mm,0)
        withcolor "darkblue" withtransparency ("normal",0.5) ;

    fill fullsquare xyscaled (4cm,2cm) randomized 5mm shifted (90mm,0)
        withcolor "darkred"  withtransparency ("normal",0.5) ;
    fill fullsquare xyscaled (2cm,4cm) randomized 5mm shifted (90mm,0)
        withcolor "darkblue" ;
\stopMPcode
```

We get a mixture of normal and transparent colors. Instead of `normal` you can also pass a number (with 1 being `normal`).

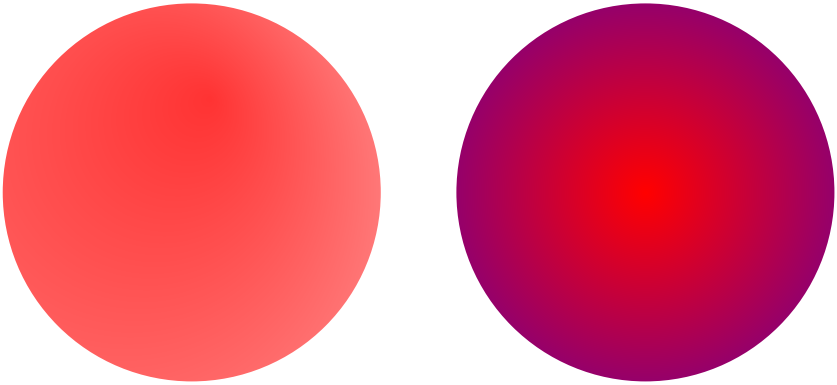


2.5 Shading

Shading is available too. This mechanism is a bit more complex deep down because it needs resources as well as shading vectors that adapt themselves to the current scale. We will not go into detail about the shading properties here.

```
\startMPcode
    comment("two shades with mp colors");
    fill fullcircle scaled 5cm
        withshademethod "circular"
        withshadevector (2cm,1cm)
        withshadecenter (.1,.5)
        withshadedomain (.2,.6)
        withshadefactor 1.2
        withshadecolors (red,white)
        ;
    fill fullcircle scaled 5cm shifted (6cm,0)
        withshademethod "circular"
        withcolor "red" shadedinto "blue"
    ;
\stopMPcode
```

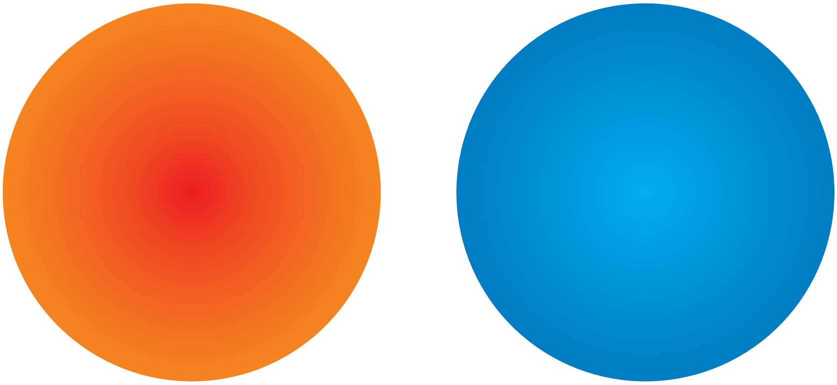
You can use normal MetaPost colors as well as named colors.



The color space of the first color determines if the second one needs to be converted, so this is valid:

```
\startMPcode
    comment("two shades with named colors");
    fill fullcircle scaled 5cm
        withshademethod "circular"
        withshadecolors ((1,0,0),(0,0,1,0))
    ;
    fill fullcircle scaled 5cm shifted (6cm,0)
        withshademethod "circular"
        withcolor (1,0,0,0) shadedinto "blue"
    ;
\stopMPcode
```

The first circle is in rgb colors and the second in cmyk.

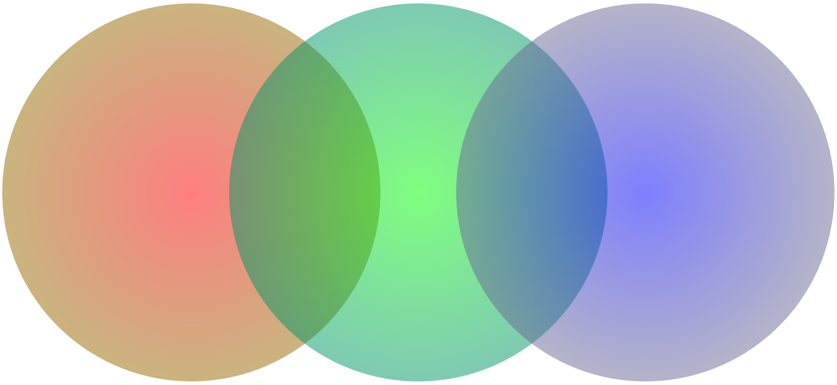


You cannot use spot colors but you can use transparency, so with:

```
\startMPcode
    comment("three transparent shades");
    fill fullcircle scaled 5cm
```

```
        withshademethod "circular"
        withshadecolors ("red","green")
        withtransparency ("normal",0.5)
    ;
    fill fullcircle scaled 5cm shifted (30mm,0)
        withshademethod "circular"
        withshadecolors ("green","blue")
        withtransparency ("normal",0.5)
    ;
    fill fullcircle scaled 5cm shifted (60mm,0)
        withshademethod "circular"
        withshadecolors ("blue","yellow")
        withtransparency ("normal",0.5)
    ;
\stopMPcode
```

we get:

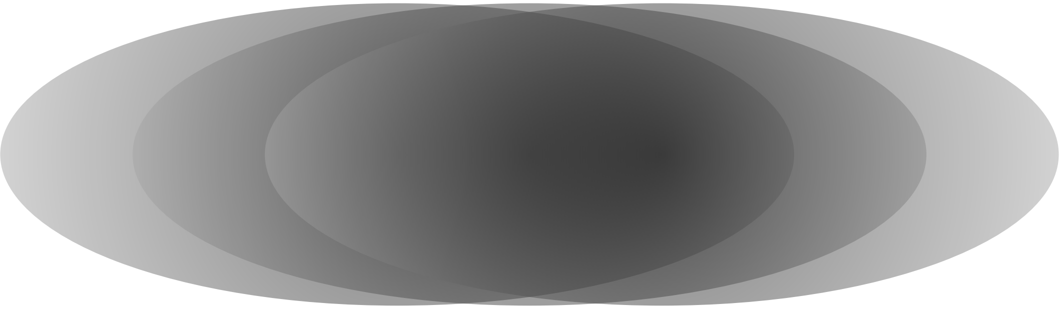


You can define a shade and use it later on, for example:

```
\startMPcode
    defineshade myshade
        withshademethod "circular"
        withshadefactor 1
        withshadedomain (0,1)
        withshadecolors (black,white)
        withtransparency (1,.5)
    ;

    fill fullcircle xyscaled(.75TextWidth,4cm)
        shaded myshade ;
    fill fullcircle xyscaled(.75TextWidth,4cm) shifted (.125TextWidth,0)
        shaded myshade ;
    fill fullcircle xyscaled(.75TextWidth,4cm) shifted (.25TextWidth,0)
```

```
        shaded myshade ;
\stopMPcode
```

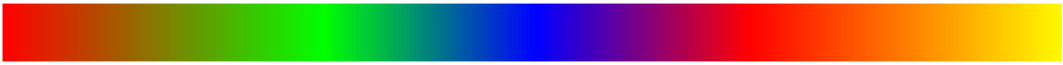
This gives three transparent shaded shapes:



A very special shade is the following:

```
\startMPcode
    fill fullsquare yscaled 5ExHeight xscaled TextWidth
        withshademethod "linear"
        withshadevector (0,1)
        withshadestep (
            withshadefraction .3
            withshadecolors (red,green)
        )
        withshadestep (
            withshadefraction .5
            withshadecolors (green,blue)
        )
        withshadestep (
            withshadefraction .7
            withshadecolors (blue,red)
        )
        withshadestep (
            withshadefraction 1
            withshadecolors (red,yellow)
        )
    ;
\stopMPcode
```

The result is a colorful band:

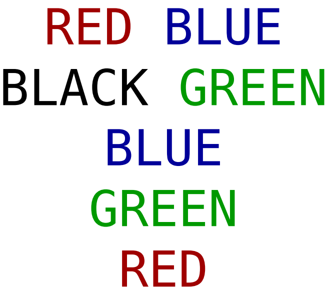


2.6 Text

The text typeset with `textext` is processed in TEX using the current settings. A text can of course have color directives embedded.

```
\startMPcode
numeric u ; u := 8mm ;
draw thetextext("RED",                     (0,0u)) withcolor darkred ;
draw thetextext("\darkgreen GREEN",        (0,1u)) ;
draw thetextext("\darkblue   BLUE",        (0,2u)) withcolor darkred ;
draw thetextext("BLACK {\darkgreen GREEN}",(0,3u)) ;
draw thetextext("RED   {\darkblue   BLUE}",(0,4u)) withcolor darkred ;
\stopMPcode
```

In this example we demonstrate that you can also apply a color to the resulting picture.



2.7 Helpers

There are several color related macros in MetaFun and these are discussed in the MetaFun manual, so we only mention a few here.

```
\startMPcode
    fill fullsquare xyscaled(TextWidth,4cm)
        withcolor darkred ;
    fill fullsquare xyscaled(TextWidth,3cm)
        withcolor complementary darkred ;
    fill fullsquare xyscaled(TextWidth,2cm)
        withcolor complemented darkred ;
    fill fullsquare xyscaled(TextWidth,1cm)
        withcolor grayed darkred ;
\stopMPcode
```

This example code is shown in figure 2.2. The `complementary` operator subtracts the given color from white, the `complemented` operator calculates its values from opposites (so a zero becomes a one). In figure 2.3 a more extensive example is shown.

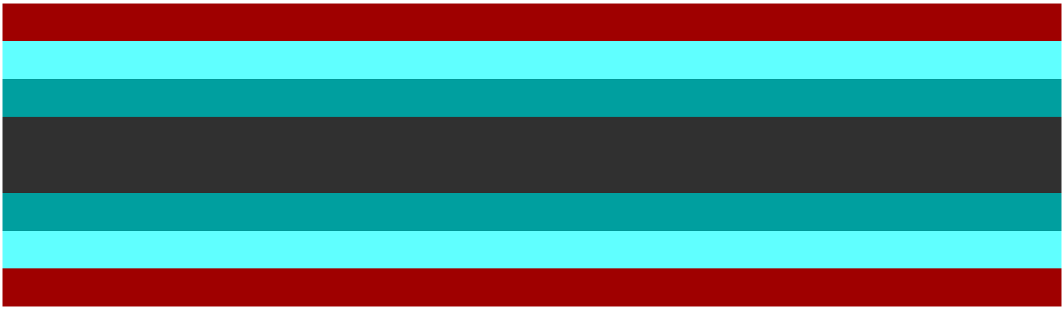
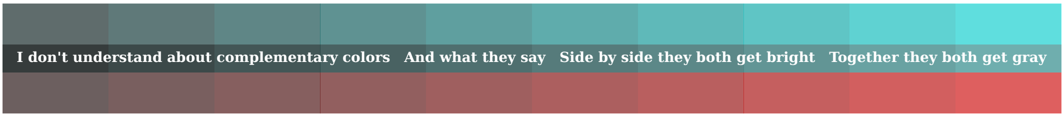
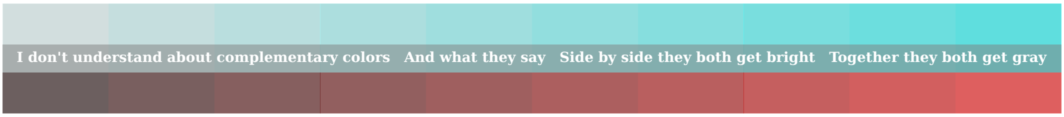


Figure 2.2 The `complementary`, `complemented` and `grayed` methodscompared.


I don't understand about complementary colors
And what they say
Side by side they both get bright
Together they both get gray

complemented


I don't understand about complementary colors
And what they say
Side by side they both get bright
Together they both get gray

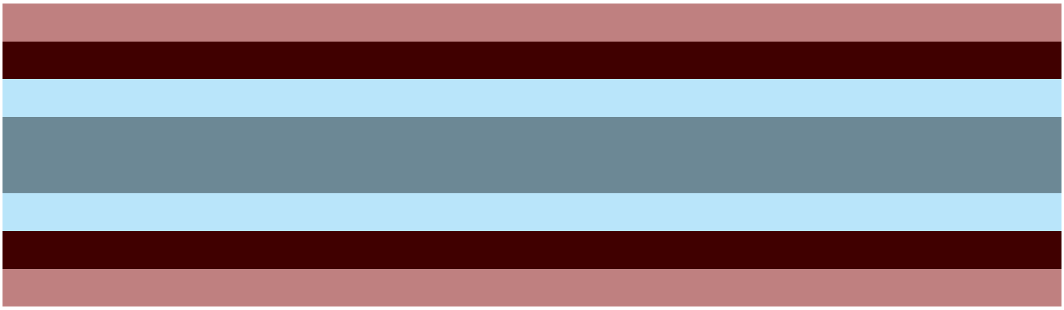
complementary

Figure 2.3 Two methods to complement colors compared (text: Fiona Apple).

As we discussed before, the different color models in MetaPost cannot be mixed in expressions. We therefore have two macros that expand into white or black in the right colorspace.

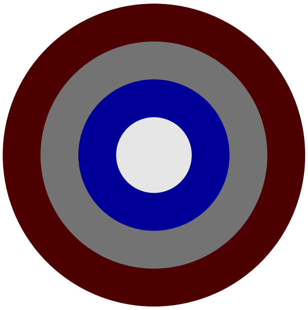
```
\startMPcode
    fill fullsquare xyscaled(TextWidth,4cm)
        withcolor darkred ;
    fill fullsquare xyscaled(TextWidth,3cm)
        withcolor complementary darkred ;
    fill fullsquare xyscaled(TextWidth,2cm)
        withcolor complemented darkred ;
    fill fullsquare xyscaled(TextWidth,1cm)
        withcolor grayed darkred ;
\stopMPcode
```

```
\startMPcode
    fill fullsquare xyscaled(TextWidth,4cm)
        withcolor .5[(.5,0,0),   whitecolor (.5,0,0)] ;
    fill fullsquare xyscaled(TextWidth,3cm)
        withcolor .5[(.5,0,0),   blackcolor (.5,0,0)] ;
    fill fullsquare xyscaled(TextWidth,2cm)
        withcolor .5[(.5,0,0,0), whitecolor (.5,0,0,0)] ;
    fill fullsquare xyscaled(TextWidth,1cm)
        withcolor .5[(.5,0,0,0), blackcolor (.5,0,0,0)] ;
\stopMPcode
```



There are two macros that can be used to resolve string to colors: `resolvedcolor` and `namedcolor`. A resolved color is expanded via Lua while a named color is handled in the backend, when the result is converted to pdf. The resolved approach is more recent and is the same as a string color specification.

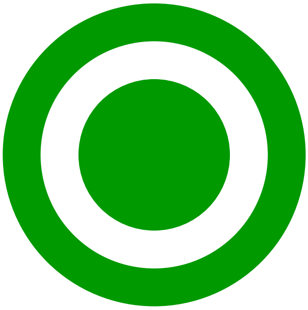
```
\startMPcode
    fill fullcircle scaled 4cm withcolor .5 resolvedcolor "darkred" ;
    fill fullcircle scaled 3cm withcolor .5 resolvedcolor "gray" ;
    fill fullcircle scaled 2cm withcolor .5 namedcolor    "darkblue" ;
    fill fullcircle scaled 1cm withcolor .5 namedcolor    "gray" ;
\stopMPcode
```



There is a `drawoptions` macro that can be used to define properties in one go.

```
\startMPcode
    drawoptions(withcolor "darkgreen");
    fill fullcircle scaled 4cm  ;
    fill fullcircle scaled 3cm withcolor white ;
    fill fullcircle scaled 2cm ;
\stopMPcode
```

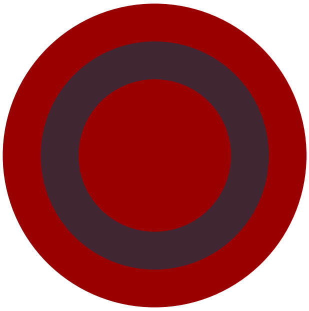
We get:



The drawback of this approach is that, because we use so called pre- and postscripts for achieving special effects (like spotcolors and shading) successive `withcolor` calls can interfere in a way that unexpected results turn up. A way out is to use properties:

```
\startMPcode
    property p[] ;
    p1 = properties(withcolor "darkred") ;
    p2 = properties(withcolor "white") ;
    fill fullcircle scaled 4cm withproperties p1 ;
    fill fullcircle scaled 3cm withproperties p2 ;
    fill fullcircle scaled 2cm withproperties p1 ;
\stopMPcode
```

This results in:



3 Graphics

3.1 Conversion

There is not that much to tell about graphics and color simply because from the perspective of TeX a graphic is just a blob with dimensions that travels through the system and in the backend gets included as-is. This means that when there is a problem with an image you have to go back to the source of that image and fix it there.

It can happen that you need to manipulate an image and in a fully automated workflow that can be cumbersome. For that reason ConTeXt has a mechanism for converting graphics.

original	**target**
bmp	default pdf
eps	default gray.pdf pdf
gif	default pdf
jpg	cmyk.pdf gray.pdf
png	cmyk.pdf gray.pdf recolor.png
ps	default gray.pdf pdf
svg	default pdf png
svgz	default pdf png
tif	default pdf

Some of these converters are applied automatically. For instance if you include an `eps` image, ConTeXt will try to convert it into a pdf file and only do that once (unless the image changed). Of course it needs a conversion program, but as long as you have GhostScript, GraphicMagick and InkScape on your machine it should work out well.

You can also define your own converters (we use a verbose variant):

```
\startluacode
    -- of course we need options

    local resolutions = {
        [interfaces.variables.low]    = "150x150",
        [interfaces.variables.medium] = "300x300",
        [interfaces.variables.high]   = "600x600",
    }

    figures.programs.lowrespng = {
        command  = "gm",
        argument = [[convert -resample %resolution% "%oldname%" "%newname%"]],
    }
    figures.converters["png"]["lowres.png"] = function(oldname,newname,resolution)
        runprogram (
            figures.programs.lowrespng.command,
            figures.programs.lowrespng.argument,
            {
                oldname    = oldname,
                newname    = newname,
                resolution = resolutions[resolution] or "150x150"
            }
        )
    end
\stopluacode
```

Usage is as follows:

```
\externalfigure[mill.png][conversion=lowres.png]
```

3.2 Recoloring

You can think of more complex conversions, like converting a gray scale image to a colored one.

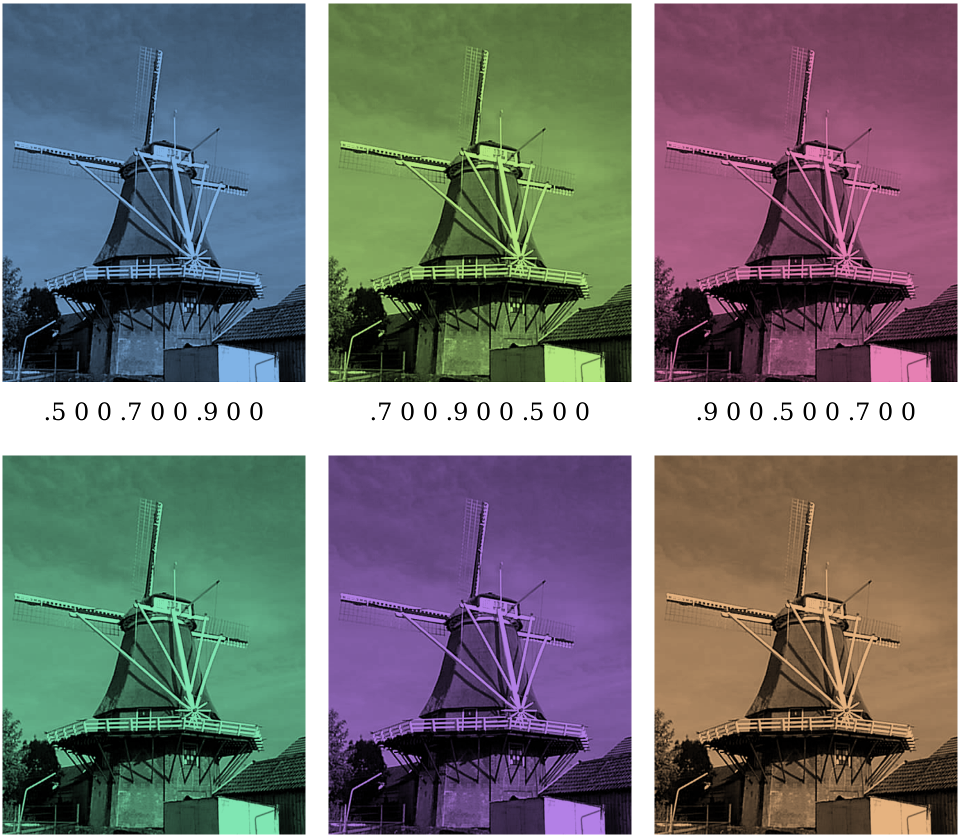
```
\startluacode
    figures.programs.recolor = {
        command  = "gm",
        argument = [[convert -recolor "%color%" "%oldname%" "%newname%"]],
    }

    figures.converters["png"]["recolor.png"] =
        function(oldname,newname,resolution,arguments)
            figures.programs.run (
                figures.programs.recolor.command,
                figures.programs.recolor.argument,
                {
                    oldname = oldname,
                    newname = newname,
                    color   = arguments or ".5 0 0 .7 0 0 .9 0 0",
                }
            )
        end
\stopluacode
```

This can be applied as follows. The `resolution` and `color` parameters get passed to the converter. This method is actually built in already.

```
\useexternalfigure[mill][mill.png][conversion=recolor.png]
```

```
\startcombination[3*2]
  {\externalfigure[mill][arguments=.5 0 0 .7 0 0 .9 0 0]}{\figurefilearguments}
  {\externalfigure[mill][arguments=.7 0 0 .9 0 0 .5 0 0]}{\figurefilearguments}
  {\externalfigure[mill][arguments=.9 0 0 .5 0 0 .7 0 0]}{\figurefilearguments}
  {\externalfigure[mill][arguments=.5 0 0 .9 0 0 .7 0 0]}{\figurefilearguments}
  {\externalfigure[mill][arguments=.7 0 0 .5 0 0 .9 0 0]}{\figurefilearguments}
  {\externalfigure[mill][arguments=.9 0 0 .7 0 0 .5 0 0]}{\figurefilearguments}
\stopcombination
```

The results are shown in figure 3.1. In this case we pass the colors to be use in a kind of matrix notation that GraphicMagick needs.



.5 0 0 .7 0 0 .9 0 0

.7 0 0 .9 0 0 .5 0 0

.9 0 0 .5 0 0 .7 0 0

.5 0 0 .9 0 0 .7 0 0

.7 0 0 .5 0 0 .9 0 0

.9 0 0 .7 0 0 .5 0 0

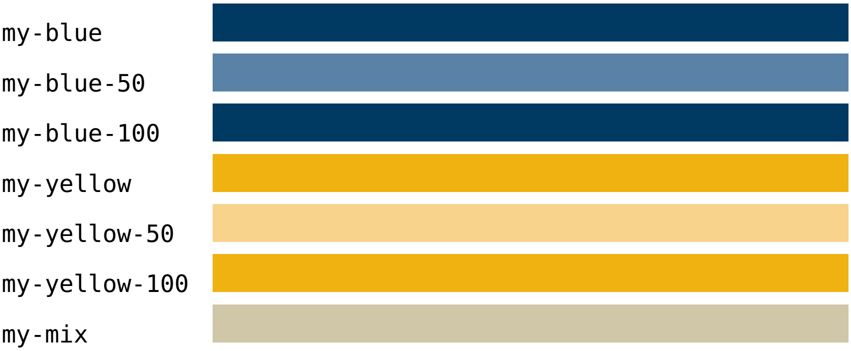
Figure 3.1 Recoloring bitmap images.

Recoloring an image this way is actually not the best solution because there is an internal mechanism that does the same. This trick (currently) only works with spot colors.

```
\definecolor [my-blue]   [c=1,m=.38,y=0,k=.64] % pms 2965 uncoated m
\definecolor [my-yellow] [c=0,m=.28,y=1,k=.06] % pms  124 uncoated m
```

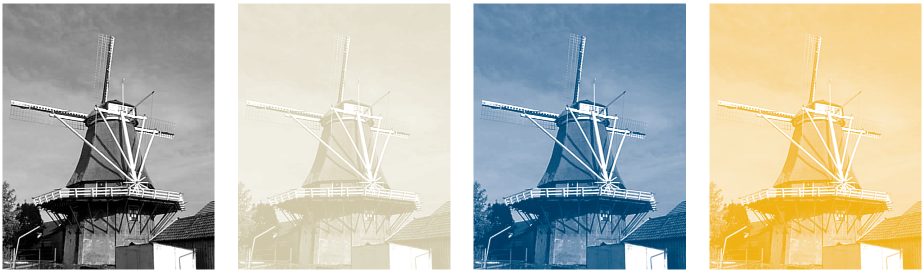
```
\definespotcolor [my-blue-100]   [my-blue]   [p=1]
\definespotcolor [my-yellow-100] [my-yellow] [p=1]
\definespotcolor [my-blue-50]    [my-blue]   [p=.5]
\definespotcolor [my-yellow-50]  [my-yellow] [p=.5]

\definemultitonecolor [my-mix] [my-blue=.12,my-yellow=.28] [c=.1,m=.1,y=.3,k=.1]
```

These colors show up as:



```
\startcombination[4*1]
  {\externalfigure[demofig]}                      {no color}
  {\externalfigure[demofig][color=my-mix]}        {indexed duotone}
  {\externalfigure[demofig][color=my-blue-100]}   {spot color}
  {\externalfigure[demofig][color=my-yellow-100]} {spot color}
\stopcombination
```

This time we don't call an external program but we add an indexed color map to the image. The result can be seen in figure 3.2.



no color

indexed duotone

spot color

spot color

Figure 3.2 Reindexing bitmap images.

3.3 Profiles

Color profiles are used to control the printing process. There is some (limited) support for that built in. An example of a setup that we use in a project is the following:

```
\setupexternalfigures
  [order={pdf,eps,png,jpg},
   conversion=cmyk.pdf,
   method=auto]
```

So, we prefer pdf vector images, if needed converted from eps. When there is no vector image we check for a png and as last resort for a jpg. The `method` is set to `auto` which means that we check if the image file indeed is reflected in the suffix. This is needed because in a workflow with tens of thousands of images there can be bad ones.

The `conversion` parameter will make ConTEXt check if there is a `cmyk.pdf` converter defined and when that is the case, it's applied. That specific converter will add a color profile to the image. You can set the profiles with:

```
\enabledirectives[graphics.conversion.rgbprofile=srgb.icc]
\enabledirectives[graphics.conversion.cmykprofile=isocoated_v2_eci.icc]
```

and these happens to be the defaults. You have to make sure that the files are present, preferable in `t:/texmf/colors/icc/context`. If you add profiles you need to make sure that `colorprofiles.lua` is updated accordingly.

Just for completeness, in our situation, we also have set:

```
\enabledirectives[graphics.conversion.eps.cleanup.ai]
\enabledirectives[graphics.extracheck]
```

The first directive will make sure that confusing sections (for instance meant to the drawing program) are stripped from an eps file, and the second one forces some extra checking on the image (just to make sure that the engine doesn't exit on bad images).

3.4 Masks

A png bitmap image can have a mask that permits a background to shine through but you can also apply that effect to a regular png image. The next examples use two (pre)defined masks:

```
\registerfiguremask [mymask1] {
    {
        {   0, 100, 0x00 },
        { 101, 200, 0x7F },
        { 201, 255, 0xFF },
    }
}

\registerfiguremask [mymask2] {
    210
}
```

The first mask maps the (grayscale) image values onto a mask value by range while the second just passes a criterium. The argument to `\registerfiguremask` is a number, table or string in Lua speak

For the examples we define two colors:

```
\definecolor[mymaskcolor1][darkred]
\definecolor[mymaskcolor2][.75(darkblue,white)]
```

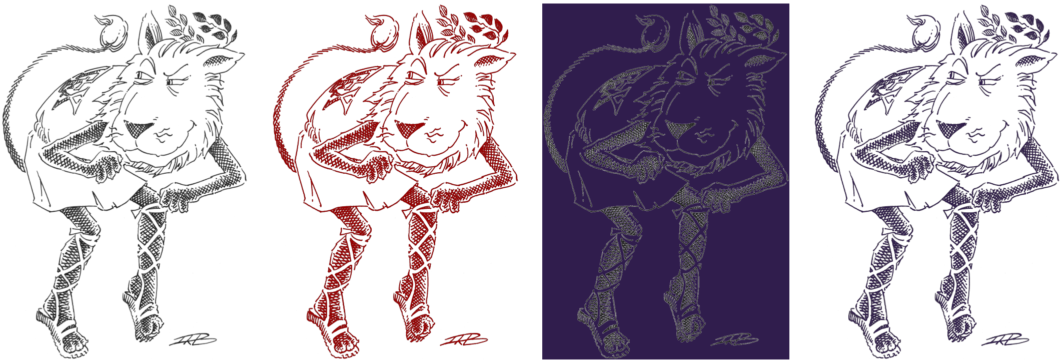
We now include two images:

```
\externalfigure
  [2019-sneaky-bw-lowres.png]
  [background=color,
   backgroundcolor=mymaskcolor1,
   mask=mymask1,
   width=\measure{combination}]
```

and

```
\externalfigure
  [2019-sneaky-bw-lowres.png]
  [background=color,
   backgroundcolor=mymaskcolor2,
   mask=mymask2,
   width=\measure{combination}]
```

The result is shown in figure 3.3 and shows that one has probably experiment a bit with the values. The first shows the original and the last the predefined 'demomask' that uses a table with four ranges.



original table number demomask

Figure 3.3 Masks

We can also use an image as mask. Take these three definitions:

```
\externalfigure
  [mill.png]
  [height=5cm]

\externalfigure
  [2019-sneaky-bw-lowres.png]
  [height=5cm]

\externalfigure
  [mill.png]
  [mask=2019-sneaky-bw-lowres.png,height=5cm]
```

In figure 3.4 the third example has both images stacked.

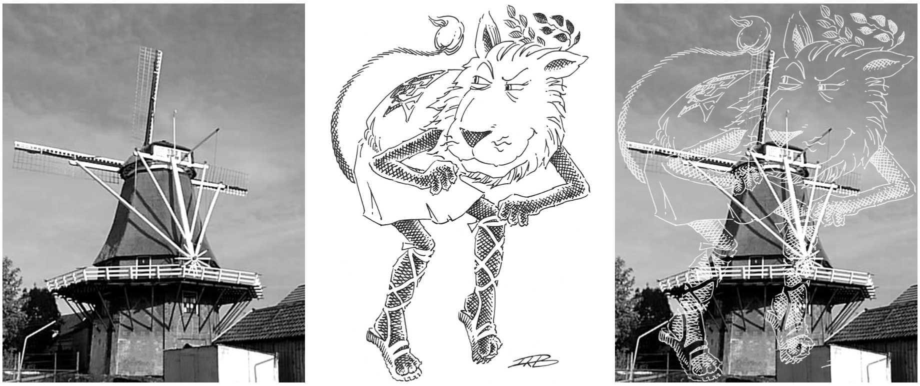
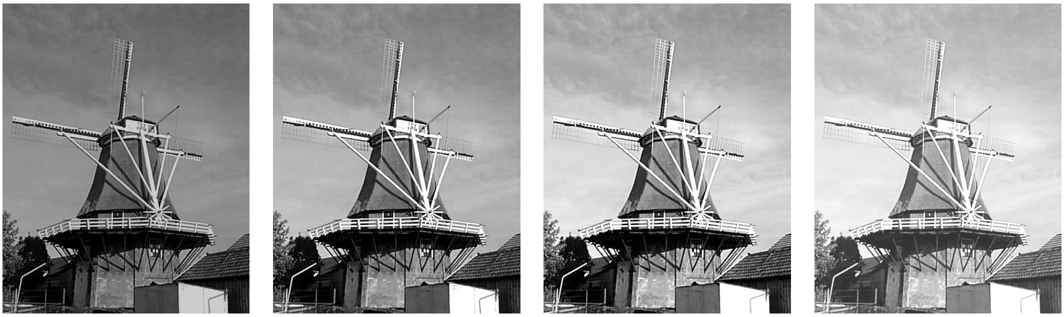


Figure 3.4 Masks

Next we show how to make an image lighter or darker. For this we use the `range` key. It can be assigned a number (fraction) or a name that serves as lookup in a registry. As with masks these are Lua definitions. AN example of a range definition is:

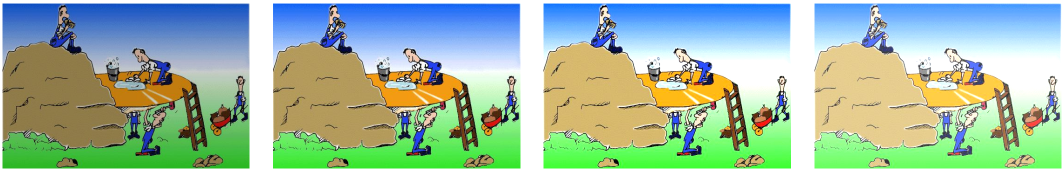
For an rgb you can provide two or six values. In figure 3.5 we show a lighter, normal, darker and limited example. In figure 3.6 we apply them to a jpeg image.

```
\registerfigurerange [myrange] {
    { 0.2, 1.2 }
}
```



`range=0.80` default `range=1.20` `range=myrange`

Figure 3.5 Ranges



`range=0.80` default `range=1.20` `range=myrange`

Figure 3.6 Ranges

This book is about colors and how to use them in ConTeXt MkIV, including MetaFun. Although the basics are not that complex, a bit of insight in how they are implemented and what can be done might help in creating more interesting looking documents.

work in progress