metafun xl

Hans Hagen

Contents

Introduction

For quite a while, around since 1996, the integration of MetaPost into ConTeXt became sort of mature but, it took decades of stepwise refinement to reach the state that we're in now. In this manual I will discuss some of the features that became possible by combining Lua and MetaPost. We already had quite a bit of that for a decade but in 2018, when LuaMetaTeX showed up a next stage was started.

Before we go into details it is good to summarize the steps that were involved in integrating MetaPost and TeX in ConTeXt. It indicates a bit what we had and have to deal with which in turn lead to the interfaces we now have.

Originally, TeX had no graphic capabilities: it just needed to know dimensions of the graphics and pass some basic information about what to include to the dvi post processor. So, a MetaPost graphic was normally processed outside the current run, resulting in PostScript graphic, that then had to be included. In pdfTeX there were some more built in options, and therefore the MetaPost code could be processed runtime using some (generic) TeX macros that I wrote. However, that engine still had to launch MetaPost for each graphic, although we could accumulate them and do that between runs. Immediate processing means that we immediately know the dimensions, while a collective run is faster. In LuaTeX this all changed to very fast runtime processing, made possible because the MetaPost library is embedded in the engine, a decision that we made early in the project and never regret.

With pdfTeX the process was managed by the `texexec` ConTeXt runner but with LuaTeX it stayed under the control of the current run. In the case of pdfTeX the actual embedding was done by TeX macros that interpreted the (relatively simple) PostScript code and turned it into pdf literals. In LuaTeX that job was delegated to Lua.

When using pdfTeX with independent MetaPost runs support for special color spaces, transparency, embedded graphics, outline text, shading and more was implemented using specials and special colors where the color served as reference to some special extension. This works quite well. In LuaTeX the pre- and postscript features, which are properties of picture objects, are used.

In all cases, some information about the current run, for instance layout related information, or color information, has to be passed to the rather isolated MetaPost run. In the case if LuaTeX (and MkIV) the advantage is that processing optional text happens in the same process so there we don't need to pass information about for instance the current font setup.

In LuaTeX the MetaPost library has a `runscript` feature, which will call Lua with the given code. This permitted a better integration: we could now ask for specific information (to the TeX end) instead of passing it from the TeX end with each run. In LuaMetaTeX another feature was added: access to the scanners from the Lua end. Although we could already fetch some variables when in Lua this made it possible to extend the MetaPost language in ways not possible before.

Already for a while Alan Braslau and I were working on some new MetaFun code that exploits all these new features. When the scanners came available I sat down and started working on new interfaces and in this manual I will discuss some of these. Some of them are illustrative, others are probably rather useful. The core of what we could call LuaMetaFun (or MetaFun XL when we use the file extension as indicator) is a key-value interface as we have at the TeX end. This interface relates to ConTeXt LMTX development and therefore related files have a different suffix: `mpxl`. However, keep in mind that some are just wrappers around regular MetaPost code so you have the full power of traditional MetaPost at hand.

We can never satisfy all needs, so to some extent this manual also demonstrates how to roll out your own code, but for that you also need to peek into the MetaFun source code too. It will take a while for this manual to complete. I also expect other users to come up with solutions, so maybe in the end we will have a collection of modules for specific tasks.

There is some duplication between this manual and the MetaFun manual which is mostly a side effect of some functionality not being present in MkIV and discussing it in the LuaMetaFun manual would be confusing. Also, from an educational point of view it doesn't hurt to read about it twice.

Because Mikael Sundqvist and I both like MetaPost and we work together on improving existing and exploring new features in the engine as well as MetaFun. Some of the examples in this manual result from that. We have a lot of fun working on this and a side effect this manual benefits a lot from our collaboration.

Hans Hagen
Hasselt NL
August 2021 (and beyond)

1 Technology

The MetaPost library that we use in LuaMetaTEX is a follow up on the library used in LuaTEX which itself is a follow up on the original MetaPost program that again was a follow up on Don Knuths MetaFont, the natural companion to TEX.

When we start with John Hobbies MetaPost we see a graphical engine that provides a simple but powerful programming language meant for making graphics, not the freehand kind, but the more systematic ones. The output is PostScript but a simple kind that can easily be converted to pdf.[1] It's output is very accurate and performance is great.

As part of the LuaTEX development project Taco Hoekwater turned MetaPost into mplib, a downward compatible library where MetaPost became a small program using that library. But there is more: there are (when enabled) backends that produce png or svg, but when used these also add dependencies on moving targets. The library by default uses the so called scaled numbers: floats that internally are long integers. But it can also work in doubles, decimal and binary and especially the last two create a dependency on libraries. It is good to notice that as in the original MetaPost the PostScript output handling is visible all over the source. Also, the way Type1 fonts are handled has been extended, for instance by providing access to shapes.

At some point a Lua interface got added that made it possible to call out to the Lua instance used in LuaTEX, so the three concepts: TEX, MetaPost and Lua can combine forces. A snippet of code can be run, and a result can be piped back. Although there is some limited access to MetaPost internals, the normal way to go is by serializing MetaPost data to the Lua end and let MetaPost scan the result using `scantokens`.

The library in LuaMetaTEX is a bit different. Of course it has the same core graphic engine, but there is no longer a backend. In ConTEXt MkIV the PostScript (and other) backends were not used anyway because it operates on the exported Lua representation of the result. Combined with the `prescript` and `postscript` features introduced in the library that provides all we need to make interesting extensions to the graphical engine (color, shading, image inclusion, text, etc). The MetaPost font support features are also not used because we need support for OpenType and even in MkII (for pdfTEX and XƎTEX) we used a different approach to fonts.

It is for that reason that the library we use in LuaMetaTEX is a leaner version of its ancestor. As mentioned, there is no backend code, only the Lua export, which saves a lot, and there are no traces of font support left, which also drops many lines of code. We forget about the binary number model because it needs a large library that also occasionally changes, but one can add it if needed. This means that there are no dependencies except for decimal but that library is relatively small and doesn't change at all. It also means that the resulting mplib library is much smaller, but it's still a substantial component in LuaMetaTEX. Internally I use the future version number 3. The original MetaPost program is version 1, so the library got version 2, and that one basically being frozen (it's in bug-fix mode) means that it will stick to that.

Another difference is that from the Lua end one has access to several scanners and also has possibilities to efficiently push back results to the engine. Running scripts can also be done more efficient. This permits a rather efficient (in terms of performance and memory usage) way to extend the language and

---

[1] For that purpose I wrote a converter in the TEX language for pdfTEX, and even within the limitations of TEX at that time (fonts, number of registers, memory) it worked out quite well.

add for instance key/value based interfaces. There are some more additions, like for instance pre- and postscripts to clip, boundary and group objects. Internals can be numeric, string and boolean. One can use utf input although that has also be added to the ancestor. Some redundant internal input/output remapping has been removed and we are more tolerant to newlines in return values from Lua. Error messages have been normalized, internal documentation cleaned up a bit. A few anomalies have been fixed too. All in- and output is now under Lua control. Etcetera. The (now very few) source files are still cweb files but the conversion to C is done with a Lua script that uses (surprise) the LuaMetaTEX engine as Lua processor. This give a bit nicer C output for when we view it in e.g. Visual Studio too (normally the cweb output is not meant to be seen by humans).

Keep in mind that it's still MetaPost with all it provided, but some has to be implemented in macros or in Lua via callbacks. The simple fact that the original library is the standard and is also the core of MetaPost most of these changes and additions cannot be backported to the original, but that is no big deal. The advantage is that we can experiment with new features without endangering users outside the ConTEXt bubble. The same is true for the Lua interface, which already is upgraded in many aspects.

2 Text

2.1 Typesetting text

The MetaFun `textext` command normally can do the job of typesetting a text snippet quite well.

```
\startMPcode
    fill fullcircle xyscaled (8cm,1cm) withcolor "darkred" ;
    draw textext("\bf This is text A") withcolor "white" ;
\stopMPcode
```

We get:

You can use regular ConTEXt commands, so this is valid:

```
\startMPcode
    fill fullcircle xyscaled (8cm,1cm) withcolor "darkred" ;
    draw textext("\framed{\bf This is text A}") withcolor "white" ;
\stopMPcode
```

Of course you can as well draw a frame in MetaPost but the `\framed` command has more options, like alignments.

Here is a variant using the MetaFun interface:

```
\startMPcode
    fill fullcircle xyscaled (8cm,1cm) withcolor "darkred" ;
    draw lmt_text [
        text  = "This is text A",
        color = "white",
        style = "bold"
    ] ;
\stopMPcode
```

The outcome is more or less the same:

Here is another example. The `format` option is actually why this command is provided.

```
\startMPcode
    fill fullcircle xyscaled (8cm,1cm) withcolor "darkred" ;
    draw lmt_text [
```

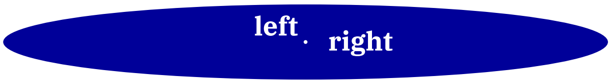
```
        text   = decimal 123.45678,
        color  = "white",
        style  = "bold",
        format = "@0.3F",
    ] ;
\stopMPcode
```

The following parameters can be set:

name	type	default	comment
`offset`	`numeric`	`0`	
`strut`	`string`	`auto`	adapts the dimensions to the font (`yes` uses the the default strut)
`style`	`string`		
`color`	`string`		
`text`	`string`		
`anchor`	`string`		one of these `lft`, `urt` like anchors
`format`	`string`		a format specifier using `@` instead of a percent sign
`position`	`pair`	`origin`	
`trace`	`boolean`	`false`	

The next example demonstrates the positioning options:

```
\startMPcode
    fill fullcircle xyscaled (8cm,1cm) withcolor "darkblue" ;
    fill fullcircle scaled .5mm withcolor "white" ;
    draw lmt_text [
        text     = "left",
        color    = "white",
        style    = "bold",
        anchor   = "lft",
        position = (-1mm,2mm),
    ] ;
    draw lmt_text [
        text   = "right",
        color  = "white",
        style  = "bold",
        anchor = "rt",
        offset = 3mm,
    ] ;
\stopMPcode
```



2.2 Strings

Those familiar with TeX probably know that there’s something called catcodes. These are properties that you assign to characters and that gives them some meaning, like regular letters, other characters, spaces, but also escape character (the backslash) or math shift (the dollar). Control over catcodes is what makes for instance verbatim possible.

We show a few possibilities and start by defining a macro:

```
\def\foo{x}
```

```
\framed\bgroup
    \startMPcode
        interim catcoderegime := vrbcatcoderegime ;
        draw textext("stream $\string\foo$") withcolor "darkred" ;
    \stopMPcode
\egroup
```

stream $\foo$

```
\framed\bgroup
    \startMPcode
        draw textext("stream $\foo$") withcolor "darkblue" ;
    \stopMPcode
\egroup
```

stream x

```
\framed\bgroup
    \startMPcode
        interim catcoderegime := vrbcatcoderegime ;
        draw textext(stream "!" $\string\foo$) withcolor "darkgreen" ;
    \stopMPcode
\egroup
```

stream "!" $\foo$

```
\framed\bgroup
    \startMPcode
        draw textext(stream "!" $\foo$) withcolor "darkyellow" ;
    \stopMPcode
\egroup
```

stream "!" x

```
\framed\bgroup
    \startMPcode
        draw textext(\btx stream "!" $\string\foo$\etx) withcolor "darkgreen" ;
    \stopMPcode
```

```
\egroup
```

stream "!" x

The `vrbcatcodesregime` switches to a verbatim catcode regime so the dollars remain dollars. But because we do expand control sequences we have to put `\string` in front.

The (expandable) `\btx` and `\etx` commands are aliases for the control characters `0x02` and `0x03`. These are valid string fences in LuaMetaTEX's MetaPost and thereby permit embedding of the double quotes.

3 Axis

The axis macro is the result of one of the first experiments with the key/value interface in MetaFun. Let's show a lot in one example:

```
\startMPcode
    draw lmt_axis [
        sx =   5mm, sy =   5mm,
        nx =  20,   ny =  10,
        dx =   5,   dy =   2,
        tx =  10,   ty =  10,

        list = {
            [
                connect = true,
                color   = "darkred",
                close   = true,
                points  = { (1, 1), (15, 8), (2, 10) },
                texts   = { "segment 1", "segment 2", "segment 3" }
            ],
            [
                connect = true,
                color   = "darkgreen",
                points  = { (2, 2), (4, 1), (10, 3), (16, 8), (19, 2) },
                labels  = { "a", "b", "c", "d", "e" }
            ],
            [
                connect = true,
                color   = "darkblue",
                close   = true,
                points  = { (5, 3), (8, 8), (16, 1) },
                labels  = { "1", "2", "3" }
            ]
        },

    ] withpen pencircle scaled 1mm  ;
\stopMPcode
```

This macro will probably be extended at some point.

name	type	default	comment
nx	numeric	1	
dx	numeric	1	
tx	numeric	0	
sx	numeric	1	
startx	numeric	0	
ny	numeric	1	
dy	numeric	1	

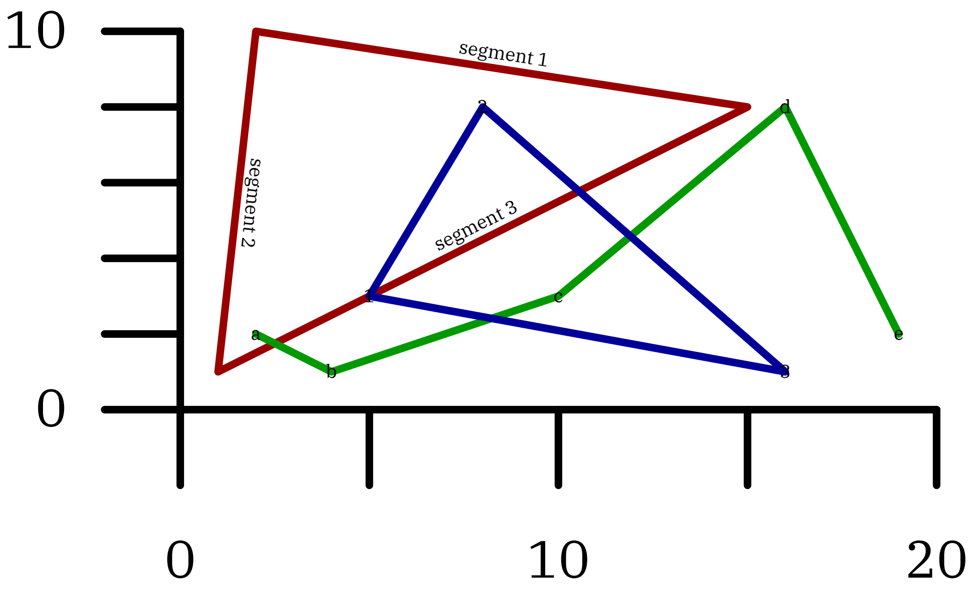


Figure 3.1

ty	numeric	0
sy	numeric	1
starty	numeric	0

samples	list	
list	list	
connect	boolean	false
list	list	
close	boolean	false
samplecolors	list	
axiscolor	string	
textcolor	string	

4 Outline

In a regular text you can have outline characters by setting a (pseudo) font feature but sometimes you want to play a bit more with this. In MetaFun we always had that option. In MkII we call `pstoedit` to turn text into outlines, in MkIV we do that by manipulating the shapes directly. And, as with some other extensions, in LMTX a new interface has been added, but the underlying code is the same as in MkIV.

In figure 4.1 we see two examples:

```
\startMPcode
    draw lmt_outline [
        text      = "hello"
        kind      = "draw",
        drawcolor = "darkblue",
    ] xsized .45TextWidth ;
\stopMPcode
```

and

```
\startMPcode
    draw lmt_outline [
        text          = "hello",
        kind          = "both",
        fillcolor     = "middlegray",
        drawcolor     = "darkgreen",
        rulethickness = 1/5,
    ] xsized .45TextWidth ;
\stopMPcode
```

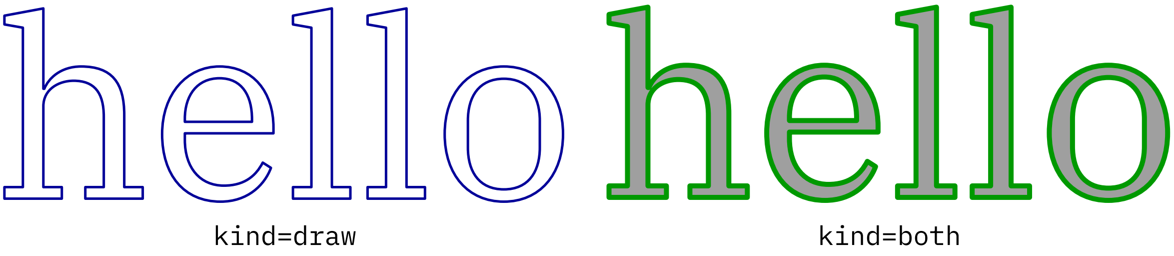
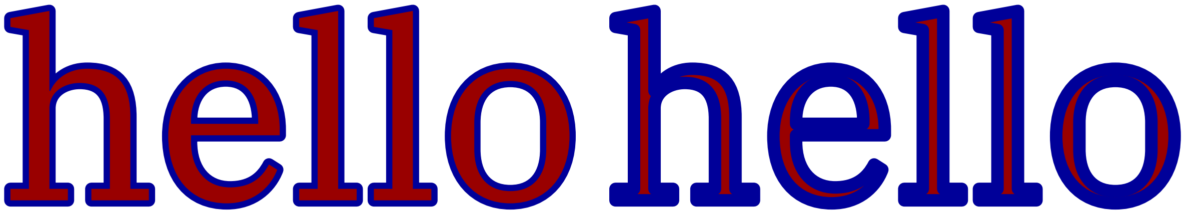


Figure 4.1 Drawing and/or filling an outline.

Normally the fill ends up below the draw but we can reverse the order, as in figure 4.2, where we coded the leftmost example as:

```
\startMPcode
    draw lmt_outline [
        text          = "hello",
        kind          = "reverse",
        fillcolor     = "darkred",
        drawcolor     = "darkblue",
        rulethickness = 1/2,
    ] xsized .45TextWidth ;
\stopMPcode
```



`kind=reverse` `kind=both`

Figure 4.2 Reversing the order of drawing and filling.

It is possible to fill and draw in one operation, in which case the same color is used for both, see figure 4.3 for an example fo this. This is a low level optimization where the shape is only output once.

`kind=fillup` `kind=fill`

Figure 4.3 Combining a fill with a draw in the same color.

This interface is much nicer than the one where each variant (the parameter `kind` above) had its own macro due to the need to group properties of the outline and fill. Let's show some more:

```
\startMPcode
    draw lmt_outline [
        text      = "\obeydiscretionaries\samplefile{tufte}",
        align     = "normal",
        kind      = "draw",
        drawcolor = "darkblue",
    ] xsized TextWidth ;
\stopMPcode
```

In this case we feed the text into the `\framed` macro so that we get a properly aligned paragraph of text, as demonstrated in figure 4.4 and ??. If you want more trickery you can of course use any ConTEXt command (including `\framed` with all kind of options) in the text.

We thrive in informationathick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthesize, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discriminate, distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synopsize, winnow the wheat from the chaff and separate the sheep from the goats.

Figure 4.4 Outlining a paragraph of text.

```
\startMPcode
    draw lmt_outline [
        text      = "\obeydiscretionaries\samplefile{ward}",
        align     = "normal,tolerant",
```

```
        style     = "bold",
        width     = 10cm,
        kind      = "draw",
        drawcolor = "darkblue",
    ] xsized TextWidth ;
\stopMPcode
```

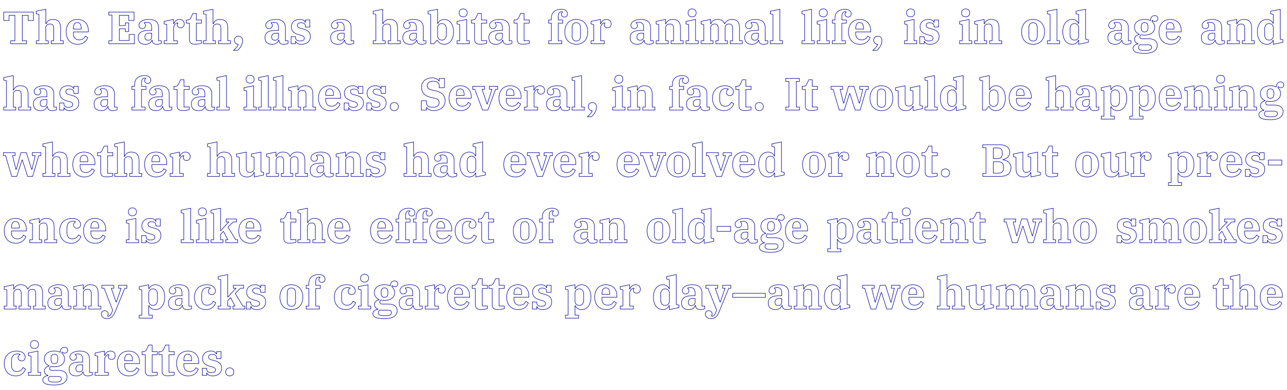


Figure 4.5 Outlining a paragraph of text with a specific width.

We summarize the parameters:

name	type	default	comment
`text`	`string`		
`kind`	`string`	`draw`	One of `draw`, `fill`, `both`, `reverse` and `fillup`.
`fillcolor`	`string`		
`drawcolor`	`string`		
`rulethickness`	`numeric`	`1/10`	
`align`	`string`		
`style`	`string`		
`width`	`numeric`		

Here is a bonus:

```
\startMPcode
    draw lmt_outline [
        kind = "outline",
        text = "\bf To foo or to bar, that's the question.",
    ] xsized TextWidth
        withshademethod "linear"
        withshadedirection down
        withshadecolors ("lightred","lightblue")
    ;
\stopMPcode
```

Here we get a path back instead of a picture with multiple elements:

5 Followtext

Typesetting text along a path started as a demo if communication between TEX and MetaPost in the early days of MetaFun. In the meantime the implementation has been modernized a few times and the current implementation feels okay, especially now that we have a better user interface. Here is an example:

```
\startMPcode{doublefun}
    draw lmt_followtext [
        text   = "How well does it work {\bf 1}! ",
        path   = fullcircle scaled 4cm,
        trace  = true,
        spread = true,
    ] ysized 5cm ;
\stopMPcode
```

Here is the same example but with the text in the reverse order. The results of both examples are shown in figure 5.1.

```
\startMPcode{doublefun}
    draw lmt_followtext [
        text    = "How well does it work {\bf 2}! ",
        path    = fullcircle scaled 4cm,
        trace   = true,
        spread  = false,
        reverse = true,
    ] ysized 5cm ;
\stopMPcode
```

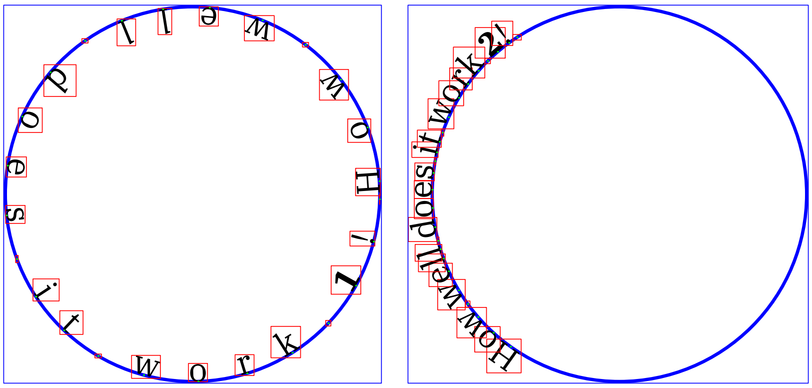


Figure 5.1

There are not that many options. One is `autoscale` which makes the shape and text match. Figure 5.2 shows what happens.

```
\startMPcode{doublefun}
    draw lmt_followtext [
        text        = "How well does it work {\bf 3}! ",
        trace       = true,
        autoscaleup = "yes"
```

```
    ] ysized 5cm ;
\stopMPcode

\startMPcode{doublefun}
    draw lmt_followtext [
        text         = "How well does it work {\bf 4}! ",
        path         = fullcircle scaled 2cm,
        trace        = true,
        autoscaleup = "max"
    ] ysized 5cm ;
\stopMPcode
```

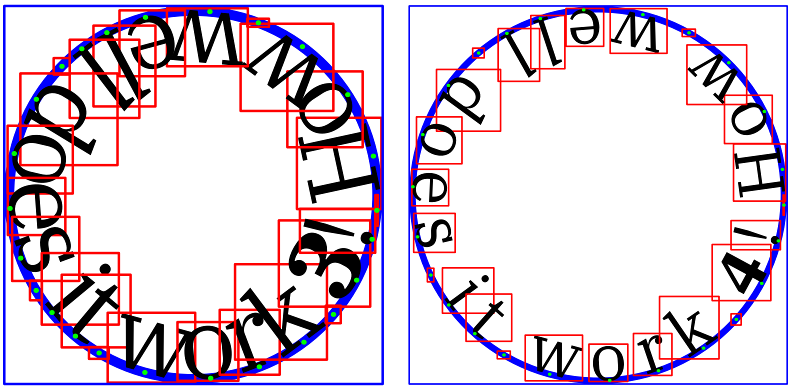


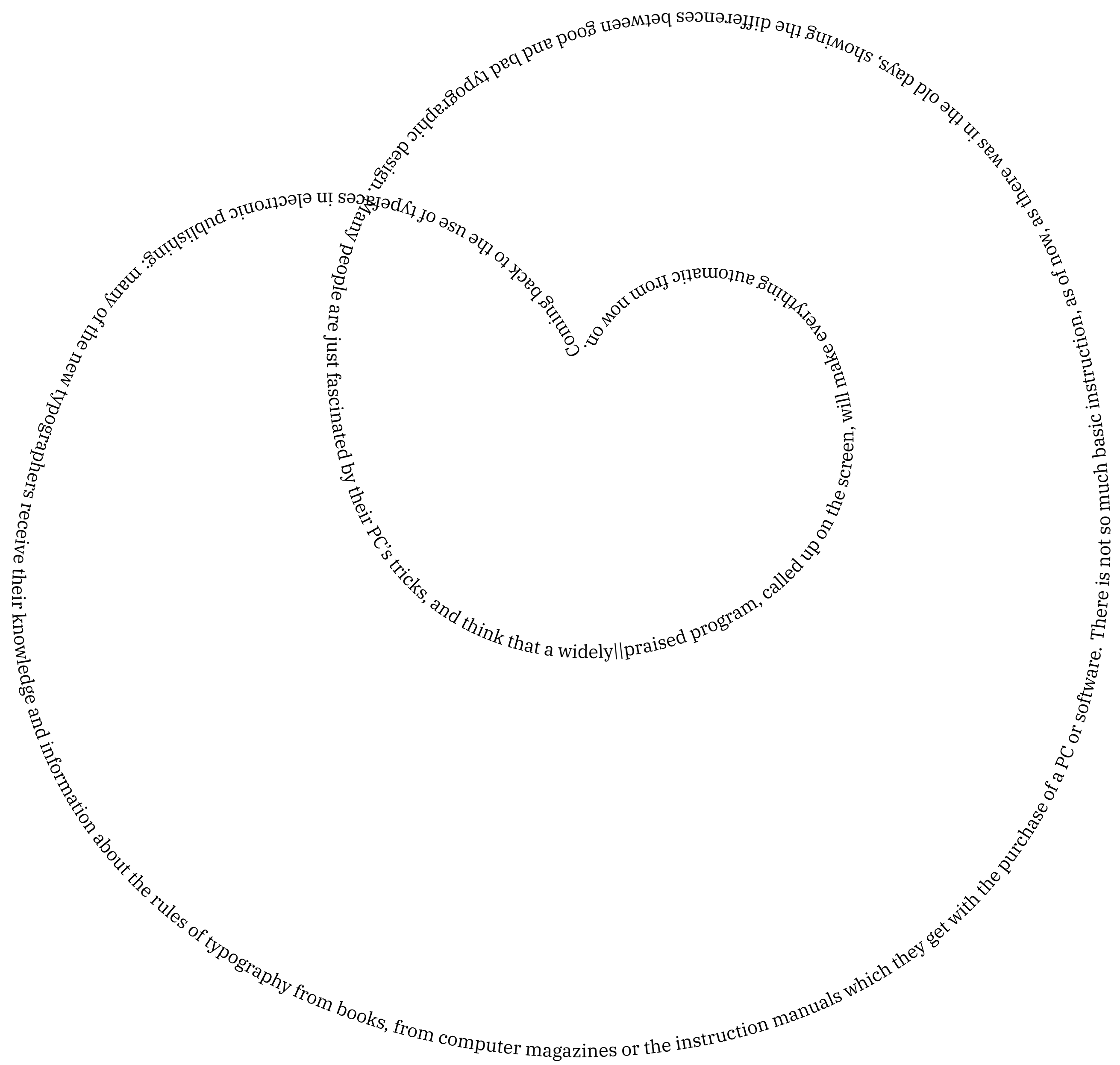
Figure 5.2

You can use quite strange paths, like the one show in figure 5.3. Watch the parenthesis around the path. this is really needed in order for the scanner to pick up the path (otherwise it sees a pair).

```
\startMPcode{doublefun}
    draw lmt_followtext [
        text         = "\samplefile {zapf}",
        path         = ((3,0) .. (1,0) .. (5,0) .. (2,0) .. (4,0) .. (3,0)),
        autoscaleup = "max"
    ] xsized TextWidth ;
\stopMPcode
```

The small set of options is:

name	type	default	comment
`text`	`string`		
`spread`	`string`	`true`	
`trace`	`numeric`	`false`	
`reverse`	`numeric`	`false`	
`autoscaleup`	`numeric`	`no`	
`autoscaledown`	`string`	`no`	
`path`	`string`	`(fullcircle)`	



Many people are just fascinated by their PC's tricks, and think that a widely||praised program, called up on the screen, will make everything automatic from now on. Coming back to the use of typefaces in electronic publishing: many of the new typographers receive their knowledge and information about the rules of typography from books, from computer magazines or the instruction manuals which they get with the purchase of a PC or software. There is not so much basic instruction, as of now, as there was in the old days, showing the differences between good and bad typographic design.

Figure 5.3

6 Placeholder

Placeholders are an old ConTEXt features and have been around since we started using MetaPost. They are used as dummy figure, just in case one is not (yet) present. They are normally activated by loading a MetaFun library:

```
\useMPLibrary[dum]
```

Just because it could be done conveniently, placeholders are now defined at the MetaPost end instead of as useable MetaPost graphic at the TEX end. The variants and options are demonstrated using side floats.

Figure 6.1

```
\startMPcode
    lmt_placeholder [
        width       = 4cm,
        height      = 3cm,
        color       = "red",
        alternative = "circle".
    ] ;
\stopMPcode
```

In addition to the traditional random circle we now also provide rectangles and triangles. Maybe some day more variants will show up.

Figure 6.2

```
\startMPcode
    lmt_placeholder [
        width       = 4cm,
        height      = 3cm,
        color       = "green",
        alternative = "square".
    ] ;
\stopMPcode
```

Here we set the colors but in the image placeholder mechanism we cycle through colors automatically. Here we use primary, rather dark, colors.

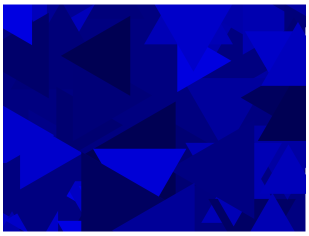
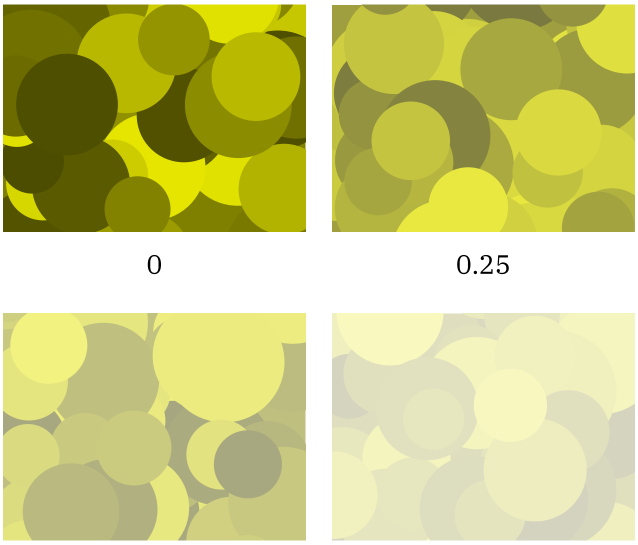


Figure 6.3

```
\startMPcode
    lmt_placeholder [
        width       = 4cm,
        height      = 3cm,
        color       = "blue",
        alternative = "triangle".
    ] ;
\stopMPcode
```

If you want less dark colors, the `reduction` parameter can be used to interpolate between the given color and white; its value is therefore a value between zero (default) and 1 (rather pointless as it produces white).



0 0.25

0.50 0.75

Figure 6.4

We demonstrate this with four variants, all circles. Of course you can also use lighter colors, but this option was needed for the image placeholders anyway.

```
\startMPcode
    lmt_placeholder [
        width       = 4cm,
        height      = 3cm,
        color       = "yellow",
        alternative = "circle".
        reduction   = 0.25,
    ] ;
\stopMPcode
```

There are only a few possible parameters. As you can see, proper dimensions need to be given because the defaults are pretty small.

name	type	default	comment
`color`	`string`	`red`	
`width`	`numeric`	`1`	
`height`	`numeric`	`1`	
`reduction`	`numeric`	`0`	
`alternative`	`string`	`circle`	

7 Arrow

Arrows are somewhat complicated because they follow the path, are constructed using a pen, have a fill and draw, and need to scale. One problem is that the size depends on the pen but the pen normally is only known afterwards.

To some extent MetaFun can help you with this issue. In figure 7.1 we see some variants. The definitions are given below:

```
\startMPcode
draw lmt_arrow [
    path = (fullcircle scaled 3cm),
]
    withpen pencircle scaled 2mm
    withcolor "darkred" ;
\stopMPcode

\startMPcode
draw lmt_arrow [
    path   = (fullcircle scaled 3cm),
    length = 8,
]
    withpen pencircle scaled 2mm
    withcolor "darkgreen" ;
\stopMPcode

\startMPcode
draw lmt_arrow [
    path = (fullcircle scaled 3cm rotated 145),
    pen  = (pencircle xscaled 4mm yscaled 2mm rotated 45),
]
    withpen pencircle xscaled 1mm yscaled .5mm rotated 45
    withcolor "darkblue" ;
\stopMPcode

\startMPcode
pickup pencircle xscaled 2mm yscaled 1mm rotated 45 ;
draw lmt_arrow [
    path = (fullcircle scaled 3cm rotated 45),
    pen  = "auto",
]
    withcolor "darkyellow" ;
\stopMPcode
```

There are some options that influence the shape of the arrowhead and its location on the path. You can for instance ask for two arrowheads:

```
\startMPcode
    pickup pencircle scaled 1mm ;
```

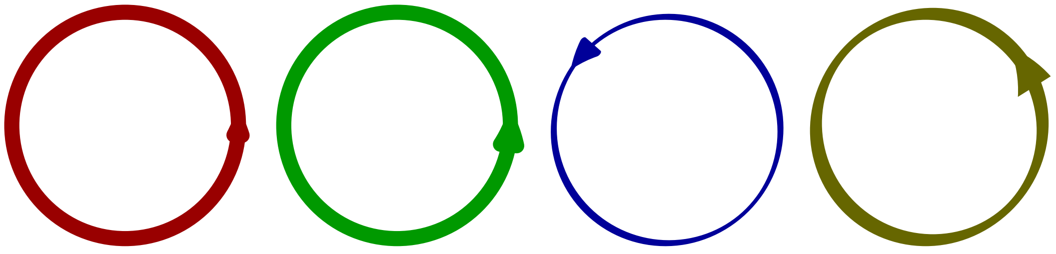
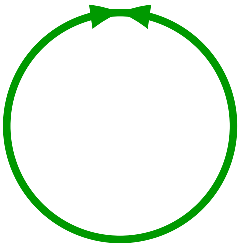


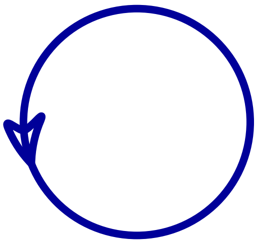
Figure 7.1

```
    draw lmt_arrow [
        pen      = "auto",
        location = "both"
        path     = fullcircle scaled 3cm rotated 90,
    ] withcolor "darkgreen" ;
\stopMPcode
```



The shape can also be influenced although often this is not that visible:

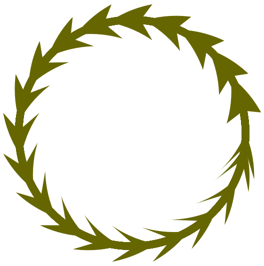
```
\startMPcode
    pickup pencircle scaled 1mm ;
    draw lmt_arrow [
        kind        = "draw",
        pen         = "auto",
        penscale    = 4,
        location    = "middle",
        alternative = "curved",
        path        = fullcircle scaled 3cm,
    ] withcolor "darkblue" ;
\stopMPcode
```



The location can also be given as percentage, as this example demonstrates. Watch how we draw only arrow heads:

```
\startMPcode
```

```
    pickup pencircle scaled 1mm ;
    for i = 0 step 5 until 100 :
        draw lmt_arrow [
            alternative = "dimpled",
            pen         = "auto",
            location    = "percentage",
            percentage  = i,
            dimple      = (1/5 + i/200),
            headonly    = (i = 0),
            path        = fullcircle scaled 3cm,
        ] withcolor "darkyellow" ;
    endfor ;
\stopMPcode
```



The supported parameters are:

name	type	default	comment
path	path		
pen	path		
	string	auto	
kind	string	fill	fill or draw
dimple	numeric	1/5	
scale	numeric	3/4	
penscale	numeric	3	
length	numeric	4	
angle	numeric	45	
location	string	end	end, middle or both
alternative	string	normal	normal, dimpled or curved
percentage	numeric	50	
headonly	boolean	false	

8 Shade

8.1 Shading operators

see MetaFun manual.

8.2 Shading interface.

This interface is still experimental!

Shading is complex. We go from one color to another on a continuum either linear or circular. We have to make sure that we cover the whole shape and that means that we have to guess a little, although one can influence this with parameters. It can involve a bit of trial and error, which is more complex that using a graphical user interface but this is the price we pay. It goes like this:

```
\startMPcode
definecolor [ name = "MyColor3", r = 0.22, g = 0.44, b = 0.66 ] ;
definecolor [ name = "MyColor4", r = 0.66, g = 0.44, b = 0.22 ] ;

draw lmt_shade [
    path      = fullcircle scaled 4cm,
    direction = "right",
    domain    = { 0, 2 },
    colors    = { "MyColor3", "MyColor4" },
] ;

draw lmt_shade [
    path      = fullcircle scaled 3cm,
    direction = "left",
    domain    = { 0, 2 },
    colors    = { "MyColor3", "MyColor4" },
] shifted (45mm,0) ;

draw lmt_shade [
    path      = fullcircle scaled 5cm,
    direction = "up",
    domain    = { 0, 2 },
    colors    = { "MyColor3", "MyColor4" },
] shifted (95mm,0) ;

draw lmt_shade [
    path      = fullcircle scaled 1cm,
    direction = "down",
    domain    = { 0, 2 },
    colors    = { "MyColor3", "MyColor4" },
] shifted (135mm,0) ;
\stopMPcode
```

Normally this is good enough as demonstrated in figure 8.1 because we use shades as backgrounds. In the case of a circular shade we need to tweak the domain because guessing doesn't work well.

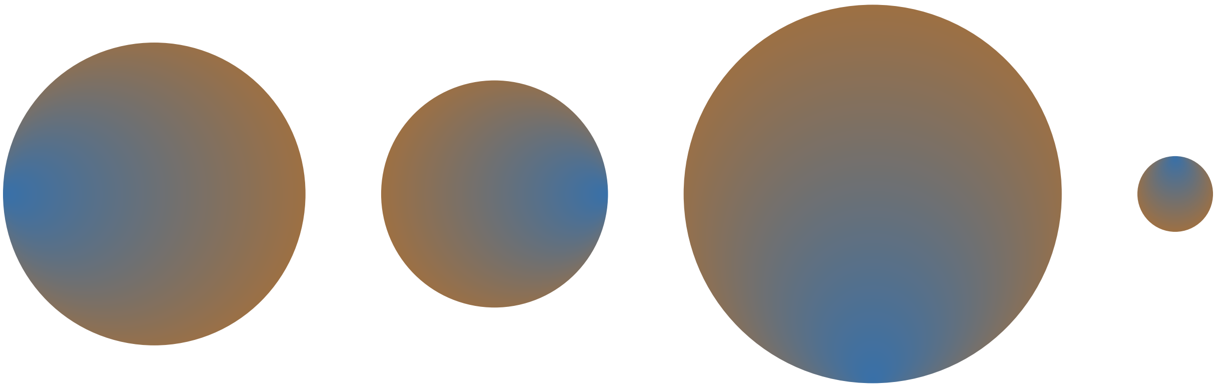


Figure 8.1 Simple circular shades.

```
\startMPcode
draw lmt_shade [
    path        = fullsquare scaled 4cm,
    alternative = "linear",
    direction   = "right",
    colors      = { "MyColor3", "MyColor4" },
] ;

draw lmt_shade [
    path        = fullsquare scaled 3cm,
    direction   = "left",
    alternative = "linear",
    colors      = { "MyColor3", "MyColor4" },
] shifted (45mm,0) ;

draw lmt_shade [
    path        = fullsquare scaled 5cm,
    direction   = "up",
    alternative = "linear",
    colors      = { "MyColor3", "MyColor4" },
] shifted (95mm,0) ;

draw lmt_shade [
    path        = fullsquare scaled 1cm,
    direction   = "down",
    alternative = "linear",
    colors      = { "MyColor3", "MyColor4" },
] shifted (135mm,0) ;
\stopMPcode
```

The `direction` relates to the boundingbox. Instead of a keyword you can also give two values, indicating points on the boundingbox. Because a boundingbox has four points, the up direction is equivalent to `{0.5,2.5}`.

The parameters `center`, `factor`, `vector` and `domain` are a bit confusing but at some point the way they were implemented made sense, so we keep them as they are. The center moves the center of the path that is used as anchor for one color proportionally to the bounding box: the given factor is multiplied by half the width and height.

Figure 8.2 Simple rectangular shades.

```
\startMPcode
draw lmt_shade [
    path      = fullcircle scaled 5cm,
    domain    = { .2, 1.6 },
    center    = { 1/10, 1/10 },
    direction = "right",
    colors    = { "MyColor3", "MyColor4" },
    trace     = true,
] ;
\stopMPcode
```

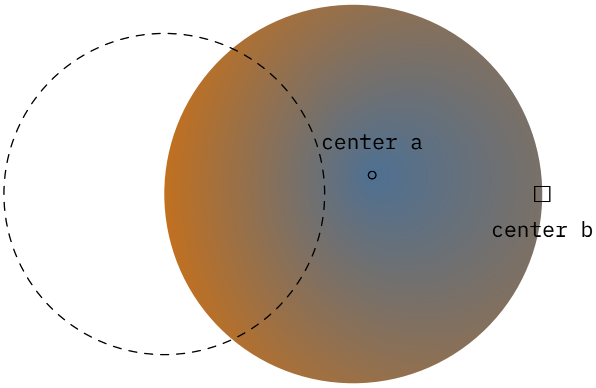


Figure 8.3 Moving the centers.

A vector takes the given points on the path as centers for the colors, see figure 8.4.

```
\startMPcode
draw lmt_shade [
    path      = fullcircle scaled 5cm,
    domain    = { .2, 1.6 },
    vector    = { 2, 4 },
    direction = "right",
    colors    = { "MyColor3", "MyColor4" },
    trace     = true,
] ;
\stopMPcode
```

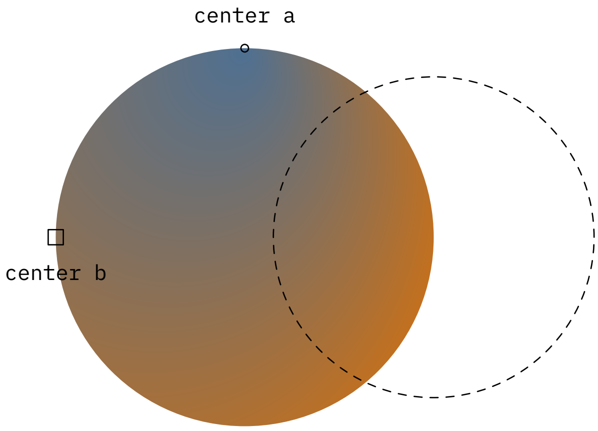


Figure 8.4 Using a vector (points).

Messing with the radius in combination with the previously mentioned domain is really trial and error, as seen in figure 8.5.

```
\startMPcode
draw lmt_shade [
    path      = fullcircle scaled 5cm,
    domain    = { 0.5, 2.5 },
    radius    = { 2cm, 6cm },
    direction = "right",
    colors    = { "MyColor3", "MyColor4" },
    trace     = true,
] ;
\stopMPcode
```

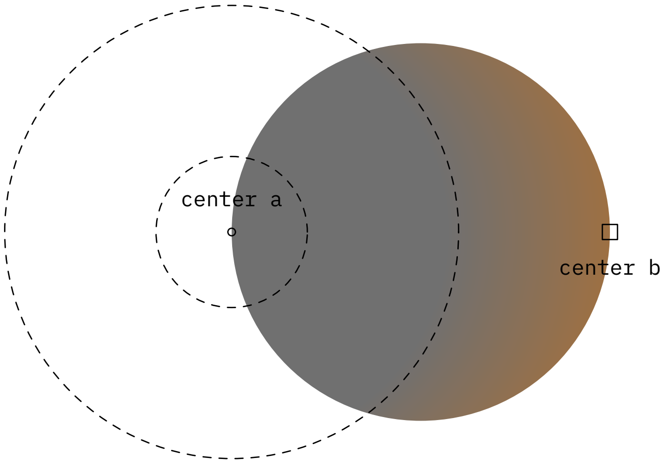


Figure 8.5 Tweaking the radius.

But actually the radius used alone works quite well as shown in figure 8.6.

```
\startMPcode
draw lmt_shade [
    path        = fullcircle scaled 5cm,
    colors      = { "red", "green" },
    trace       = true,
] ;
```

```
draw lmt_shade [
    path        = fullcircle scaled 5cm,
    colors      = { "red", "green" },
    radius      = 2.5cm,
    trace       = true,
] shifted (6cm,0) ;

draw lmt_shade [
    path        = fullcircle scaled 5cm,
    colors      = { "red", "green" },
    radius      = 2.0cm,
    trace       = true,
] shifted (12cm,0) ;
\stopMPcode
```

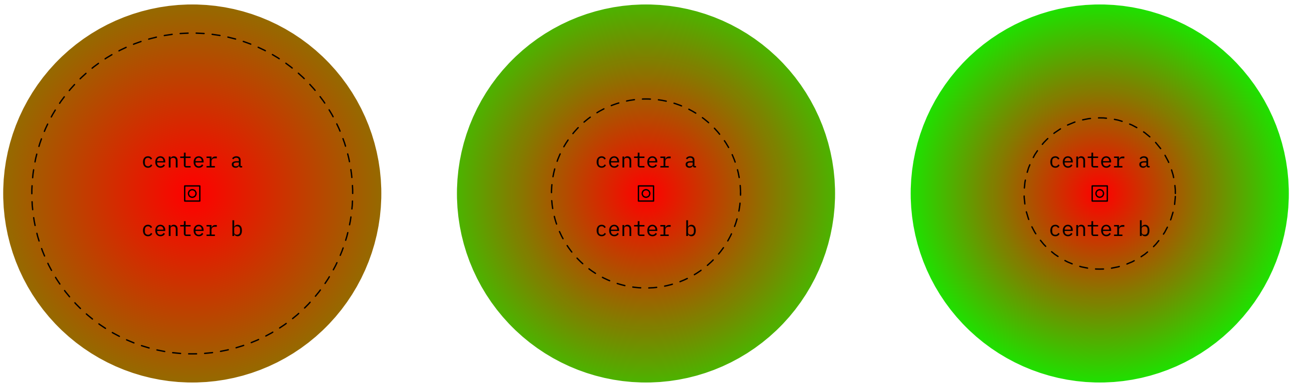


Figure 8.6 Just using the radius.

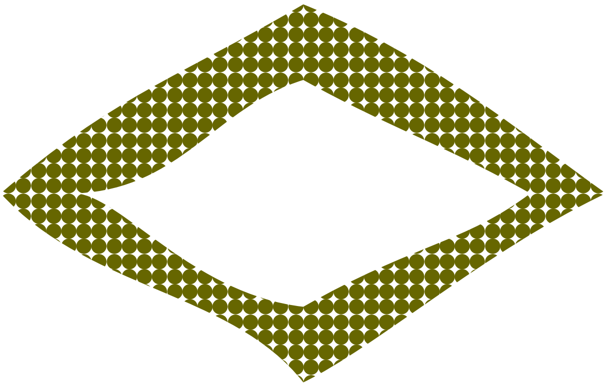
name	type	default	comment
alternative	string	circular	or linear
path	path		
trace	boolean	false	
domain	set of numerics		
radius	numeric		
	set of numerics		
factor	numeric		
origin	pair		
	set of pairs		
vector	set of numerics		
colors	set of strings		
center	numeric		
	set of numerics		
direction	string		up, down, left, right
	set of numerics		two points on the boundingbox

8.3 Patterns

Instead using a shade one can use a pattern which is basically a fill with a repeated image. Here are some examples:

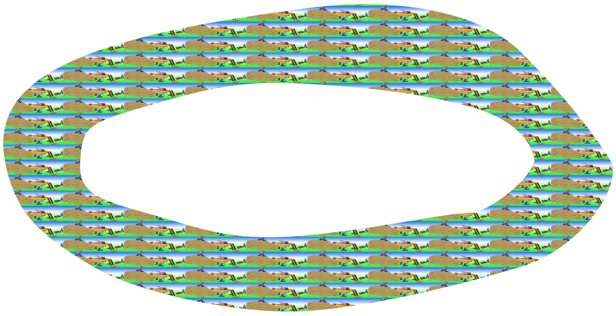
```
\startMPcode
draw
    (
        (fulldiamond xscaled 8cm yscaled 5cm randomizedcontrols 10mm) && reverse
        (fulldiamond xscaled 6cm yscaled 3cm randomizedcontrols 10mm) && cycle
    )
    withpattern image (fill fullcircle scaled 2mm withcolor "darkyellow" ;)
;
\stopMPcode
```

The `image` macro produces a picture that is then used for the filling:



That image can also be an (external) figure:

Of course one needs to find a suitable image for this, but here we just use one of the test figures:



```
\startMPcode
draw
    (
        (fullcircle xscaled 8cm yscaled 4cm randomizedcontrols 5mm) && reverse
        (fullcircle xscaled 6cm yscaled 2cm randomizedcontrols 5mm) && cycle
    )
    withpattern image (draw figure "hacker.jpg" ;)
    withpatternscale (1/10,1/20)
;
\stopMPcode
```

8.4 Luminance

Todo: groups and such.

9 Contour

This feature started out as experiment triggered by a request on the mailing list. In the end it was a nice exploration of what is possible with a bit of Lua. In a sense it is more subsystem than a simple MetaPost macro because quite some Lua code is involved and more might be used in the future. It's part of the fun.

A contour is a line through equivalent values z that result from applying a function to two variables x and y. There is quite a bit of analysis needed to get these lines. In MetaFun we currently support three methods for generating a colorful background and three for putting lines on top:

One solution is to use the the isolines and isobands methods are described on the marching squares page of wikipedia:

`https://en.wikipedia.org/wiki/Marching_squares`

This method is relative efficient as we don't do much optimization, simply because it takes time and the gain is not that much relevant. Because we support filling of multiple curves in one go, we get efficient paths anyway without side effects that normally can occur from many small paths alongside. In these days of multi megabyte movies and sound clips a request of making a pdf file small is kind of strange anyway. In practice the penalty is not that large.

As background we can use a bitmap. This method is also quite efficient because we use indexed colors which results in a very good compression. We use a simple mapping on a range of values.

A third method is derived from the one that is distributed as C source file at:

`https://physiology.arizona.edu/people/secomb/contours`
`https://github.com/secomb/GreensV4`

We can create a background image, which uses a sequence of closed curves[2]. It can also provide two variants of lines around the contours (we tag them shape and shade). It's all a matter of taste. In the meantime I managed to optimize the code a bit and I suppose that when I buy a new computer (the code was developed on an 8 year old machine) performance is probably acceptable.

In order of useability you can think of isoband (band) with isolines (cell), bitmap (bitmap) with isolines (cell) and finally shapes (shape) with edges (edge). But let's start with a couple of examples.

```
\startMPcode{doublefun}
    draw lmt_contour [
        xmin =  0, xmax = 4*pi, xstep = .05,
        ymin = -6, ymax = 6,    ystep = .05,

        levels     = 7,
        height     = 5cm,
        preamble   = "local sin, cos = math.sin, math.cos",
        function   = "cos(x) + sin(y)",
        background = "bitmap",
        foreground = "edge",
```

---

[2] I have to figure out how to improve it a bit so that multiple path don't get connected.

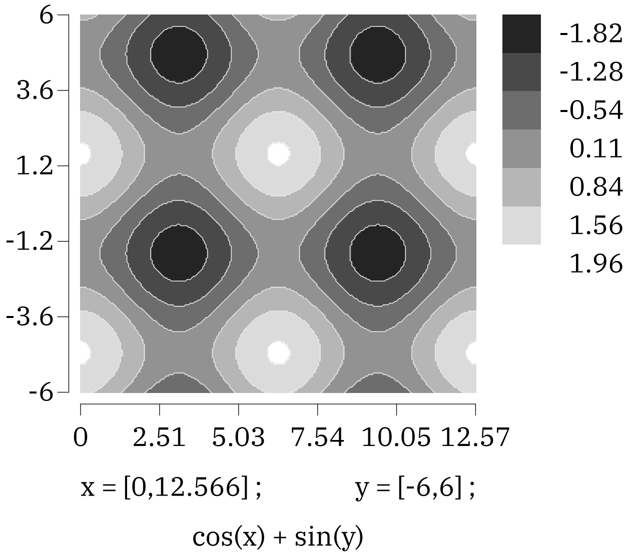


Figure 9.1

```
        linewidth  = 1/2,
        cache      = true,
    ] ;
\stopMPcode
```

In figure 9.1 we see the result. There is a in this case black and white image generated and on top of that we see lines. The step determines the resolution of the image. In practice using a bitmap is quite okay and also rather efficient: we use an indexed colorspace and, as already was mentioned, because the number of colors is limited such an image compresses well. A different rendering is seen in figure 9.2 where we use the shape method for the background. That method creates outlines but is much slower, and when you use a high resolution (small step) it can take quite a while to identify the shapes. This is why we set the cache flag.

```
\startMPcode{doublefun}
    draw lmt_contour [
        xmin =  0, xmax = 4*pi, xstep = .10,
        ymin = -6, ymax = 6,    ystep = .10,

        levels     = 7,
        preamble   = "local sin, cos = math.sin, math.cos",
        function   = "cos(x) - sin(y)",
        background = "shape",
        foreground = "shape",
        linewidth  = 1/2,
        cache      = true,
    ] ;
\stopMPcode
```

We mentioned colorspace but haven't seen any color yet, so let's set some in figure 9.3. Two variants are shown: a background shape with foreground shape and a background `bitmap` with a foreground edge. The bitmap renders quite fast, definitely when we compare with the shape, while the quality is as good at this size.

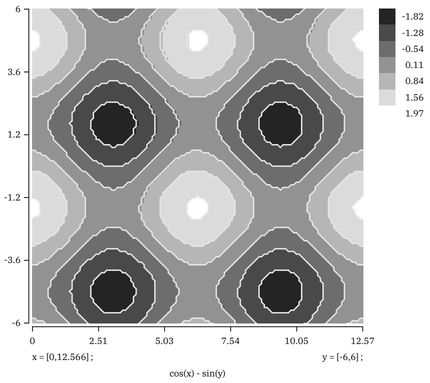
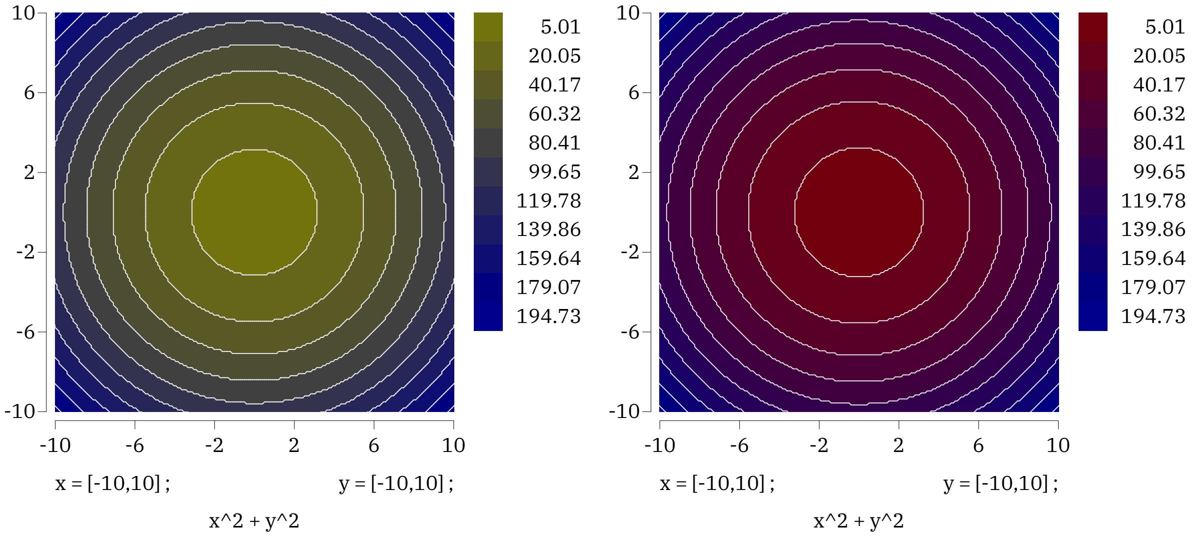


Figure 9.2

```
\startMPcode{doublefun}
    draw lmt_contour [
        xmin = -10, xmax = 10, xstep =  .1,
        ymin = -10, ymax = 10, ystep =  .1,

        levels     = 10,
        height     = 7cm,
        color      = "shade({1/2,1/2,0},{0,0,1/2})",
        function   = "x^2 + y^2",
        background = "shape",
        foreground = "shape",
        linewidth  = 1/2,
        cache      = true,
    ] xsized .45TextWidth ;
\stopMPcode
```



shape **bitmap**

Figure 9.3

We use the `doublefun` instance because we need to be sure that we don't run into issues with scaled numbers, the default model in MetaPost. The function that gets passed is *not* using MetaPost but Lua, so basically you can do very complex things. Here we directly pass code, but you can for instance also do this:

```
\startluacode
    function document.MyContourA(x,y)
        return x^2 + y^2
    end
\stopluacode
```

and then `function = "document.MyContourA(x,y)"`. As long as the function returns a valid number we're okay. When you pass code directly you can use the `preamble` key to set local shortcuts. In the previous examples we took `sin` and `cos` from the math library but you can also roll out your own functions and/or use the more elaborate `xmath` library. The color parameter is also a function, one that returns one or three arguments. In the next example we use `lin` to calculate a fraction of the current level and total number of levels.

```
\startMPcode{doublefun}
    draw lmt_contour [
        xmin = -3, xmax = 3, xstep =  .01,
        ymin = -1, ymax = 1, ystep =  .01,

        levels     = 10,
        default    = .5,
        height     = 5cm,
        function   = "x^2 + y^2 + x + y/2",
        color      = "lin(l), 0, 1/2",
        background = "bitmap"
        foreground = "none",
        cache      = true,
    ] xsized TextWidth ;
\stopMPcode
```

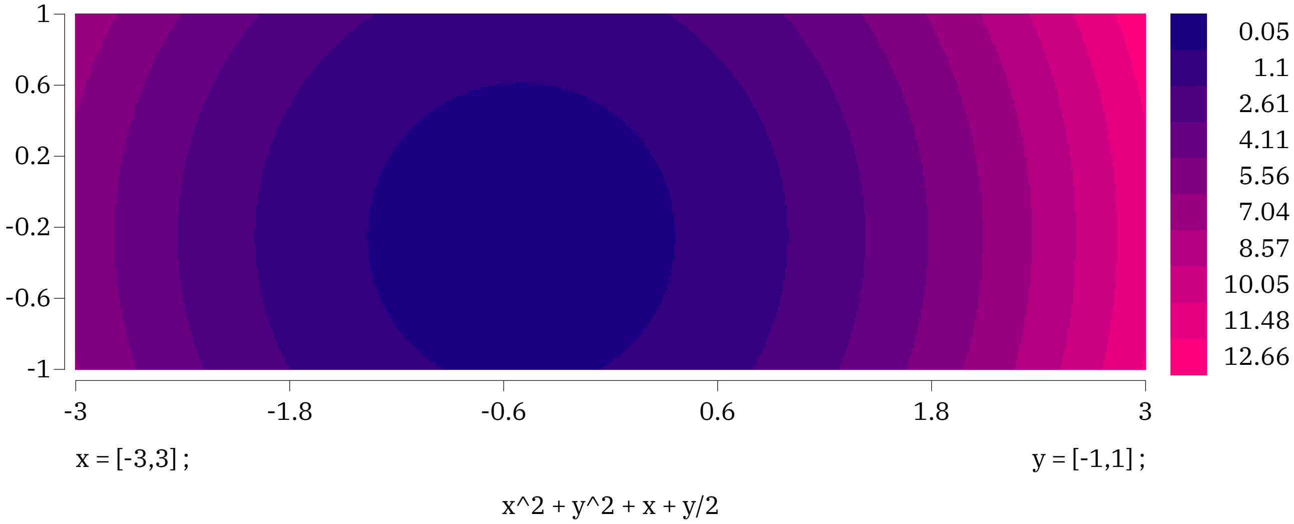


Figure 9.4

Instead of a bitmap we can use an isoband, which boils down to a set of tiny shapes that make up a bigger one. This is shown in figure 9.5.

```
\startMPcode{doublefun}
    draw lmt_contour [
```

```
        xmin = -3, xmax = 3, xstep =  .01,
        ymin = -1, ymax = 1, ystep =  .01,

        levels     = 10,
        default    = .5,
        height     = 5cm,
        function   = "x^2 + y^2 + x + y/2",
        color      = "lin(l), 1/2, 0",
        background = "band",
        foreground = "none",
        cache      = true,
    ] xsized TextWidth ;
\stopMPcode
```

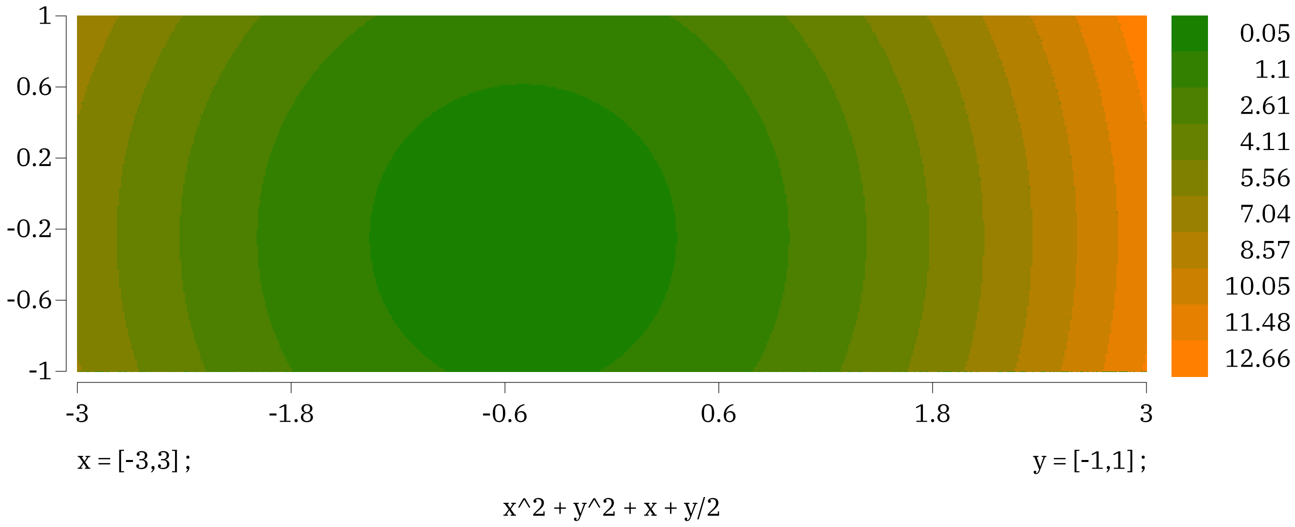


Figure 9.5

You can draw several functions and see where they overlap:

```
\startMPcode{doublefun}
    draw lmt_contour [
        xmin = -pi, xmax = 4*pi, xstep = .1,
        ymin = -3,  ymax = 3,    ystep = .1,

        range      = { -.1, .1 },
        preamble   = "local sin, cos = math.sin, math.cos",
        functions  = {
            "sin(x) + sin(y)", "sin(x) + cos(y)",
            "cos(x) + sin(y)", "cos(x) + cos(y)"
        },
        background = "bitmap",
        linecolor  = "black",
        linewidth  = 1/10,
        color      = "shade({1,1,0},{0,0,1})"
        cache      = true,
    ] xsized TextWidth ;
\stopMPcode
```

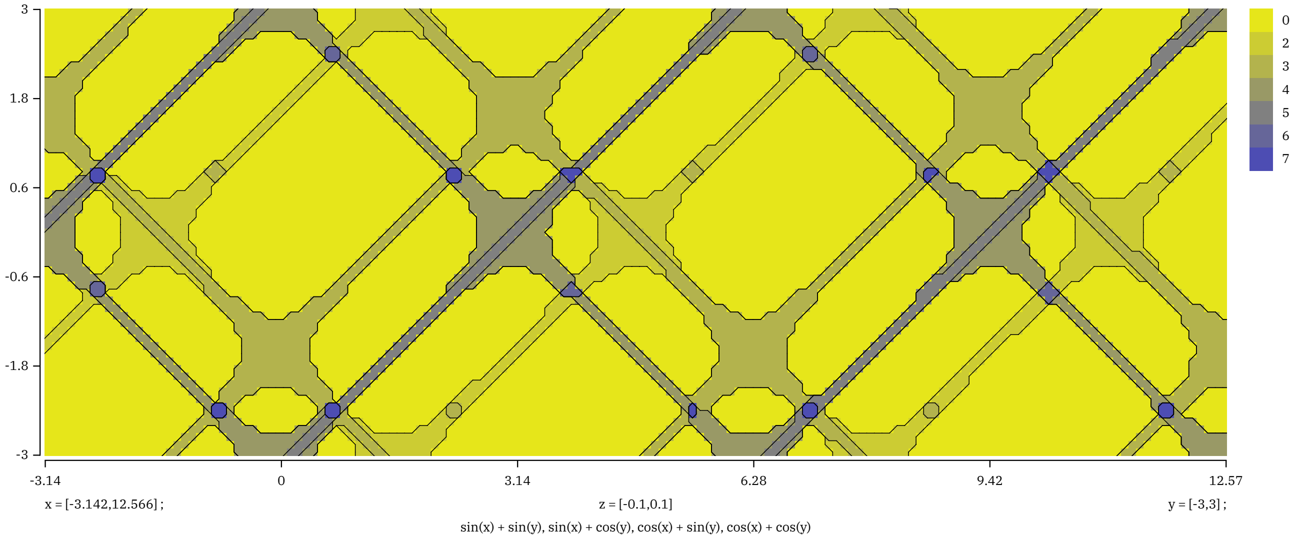


Figure 9.6

The range determines the *z* value(s) that we take into account. You can also pass a list of colors to be used. In figure 9.7 this is demonstrated. There we also show a variant foreground `cell`, which uses a bit different method for calculating the edges.[3]

```
\startMPcode{doublefun}
    draw lmt_contour [
        xmin = -2*pi, xmax =  2*pi, xstep = .01,
        ymin = -3,    ymax =  3,    ystep = .01,

        range      = { -.1, .1 },
        preamble   = "local sin, cos = math.sin, math.cos",
        functions  = { "sin(x) + sin(y)", "sin(x) + cos(y)" },
        background = "bitmap",
        foreground = "cell",
        linecolor  = "white",
        linewidth  = 1/10,
        colors     = { (1/2,1/2,1/2), red, green, blue }
        level      = 3,
        linewidth  = 6,
        cache      = true,
    ] xsized TextWidth ;
\stopMPcode
```

Here the number of levels depends on the number of functions as each can overlap with another; for instance the outcome of two functions can overlap or not which means 3 cases, and with a value not being seen that gives 4 different cases.

```
\startMPcode{doublefun}
    draw lmt_contour [
        xmin = -2*pi, xmax =  2*pi, xstep = .01,
        ymin = -3,    ymax =  3,    ystep = .01,
```

---

[3] This a bit of a playground: more variants might show up in due time.

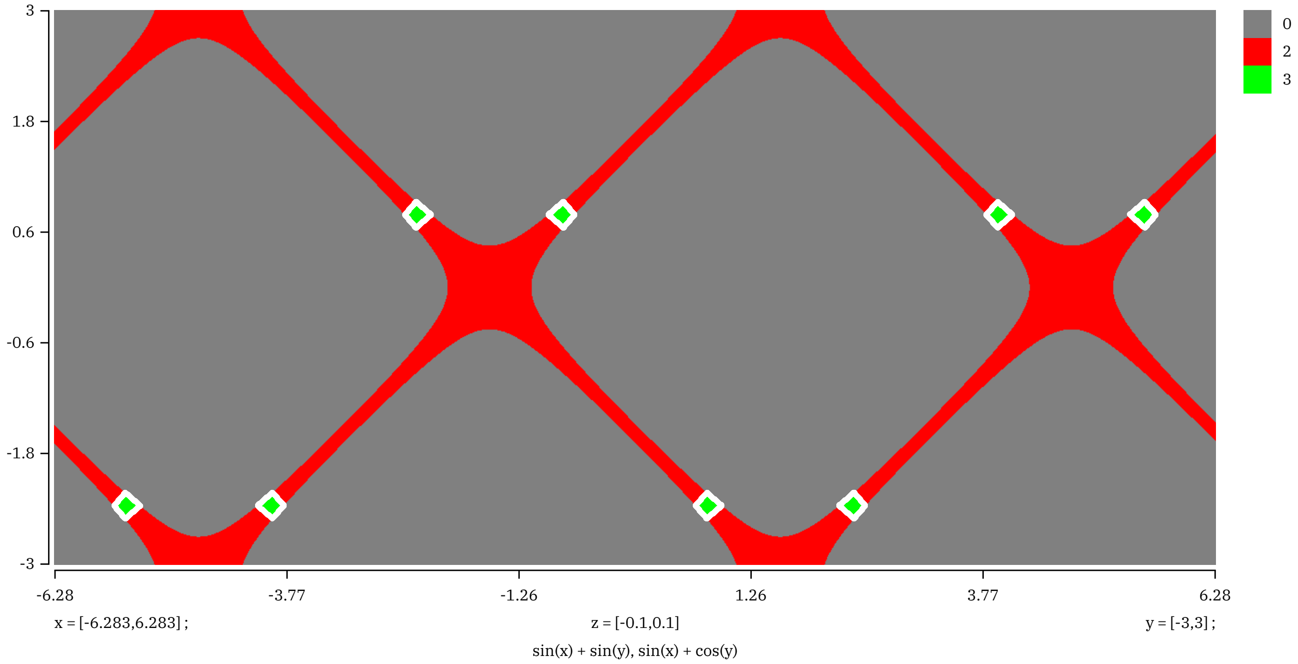


Figure 9.7

```
        range      = { -.1, .1 },
        preamble   = "local sin, cos = math.sin, math.cos",
        functions  = {
            "sin(x) + sin(y)",
            "sin(x) + cos(y)",
            "cos(x) + sin(y)",
            "cos(x) + cos(y)"
        },
        background = "bitmap",
        foreground = "none",
        level      = 3,
        color      = "shade({2/3,0,0},{2/3,1,2/3})"
        cache      = true,
    ] xsized TextWidth ;
\stopMPcode
```

Of course one can wonder how useful showing many functions but it can give nice pictures, as shown in figure 9.8.

```
\startMPcode{doublefun}
    draw lmt_contour [
        xmin = -2*pi, xmax =  2*pi, xstep = .01,
        ymin = -3,    ymax =  3,    ystep = .01,

        range      = { -.3, .3 },
        preamble   = "local sin, cos = math.sin, math.cos",
        functions  = {
            "sin(x) + sin(y)",
            "sin(x) + cos(y)",
            "cos(x) + sin(y)",
```

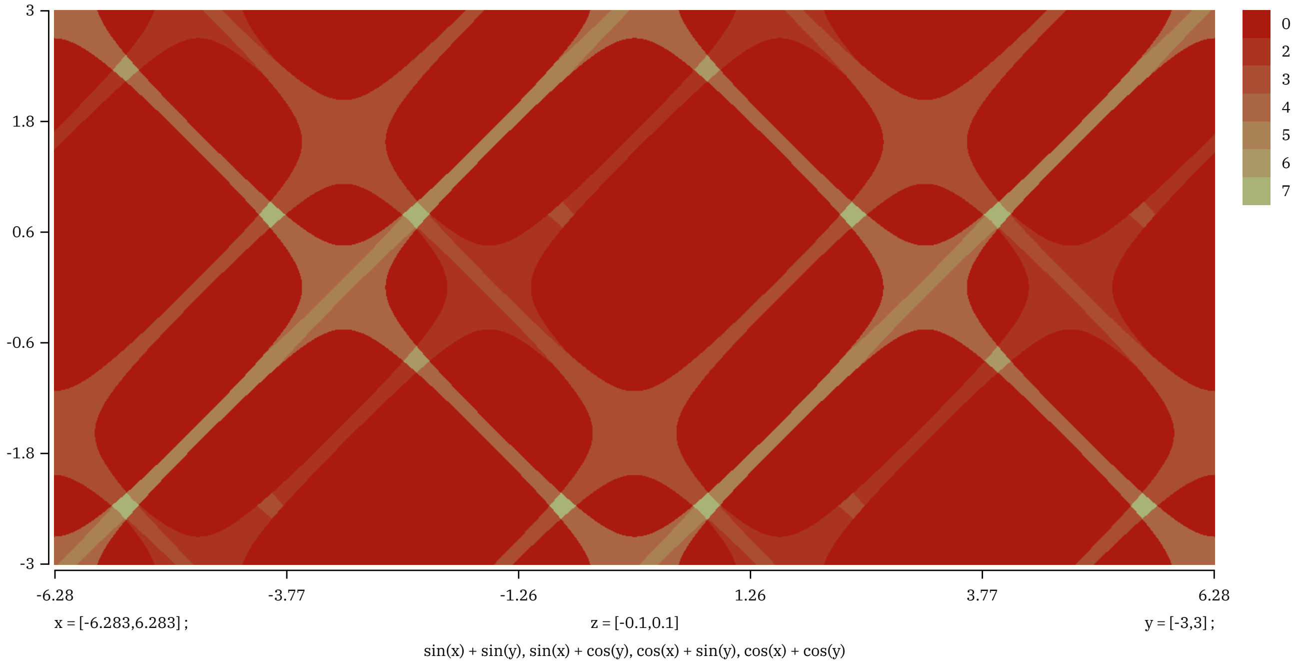


Figure 9.8

```
            "cos(x) + cos(y)"
        },
        background = "bitmap",
        foreground = "none",
        level      = 3,
        color      = "shade({1,0,0},{0,1,0})"
        cache      = true,
    ] xsized TextWidth ;
\stopMPcode
```

We can enlargen the window, which is demonstrated in figure 9.9. I suppose that such images only make sense in educational settings.

In figure 9.10 we see different combinations of backgrounds (in color) and foregrounds (edges) in action.

```
\startMPcode{doublefun}
    draw lmt_contour [
        xmin =  0, xmax = 4*pi, xstep = 0,
        ymin = -6, ymax = 6,    ystep = 0,

        levels = 5, legend = false, linewidth  = 1/2,

        preamble   = "local sin, cos = math.sin, math.cos",
        function   = "cos(x) - sin(y)",
        color      = "shade({1/2,0,0},{0,0,1/2})",

        background = "bitmap", foreground = "cell",
    ] xsized .3TextWidth ;
\stopMPcode
```

There are quite some settings. Some deal with the background, some with the foreground and quite some deal with the legend.

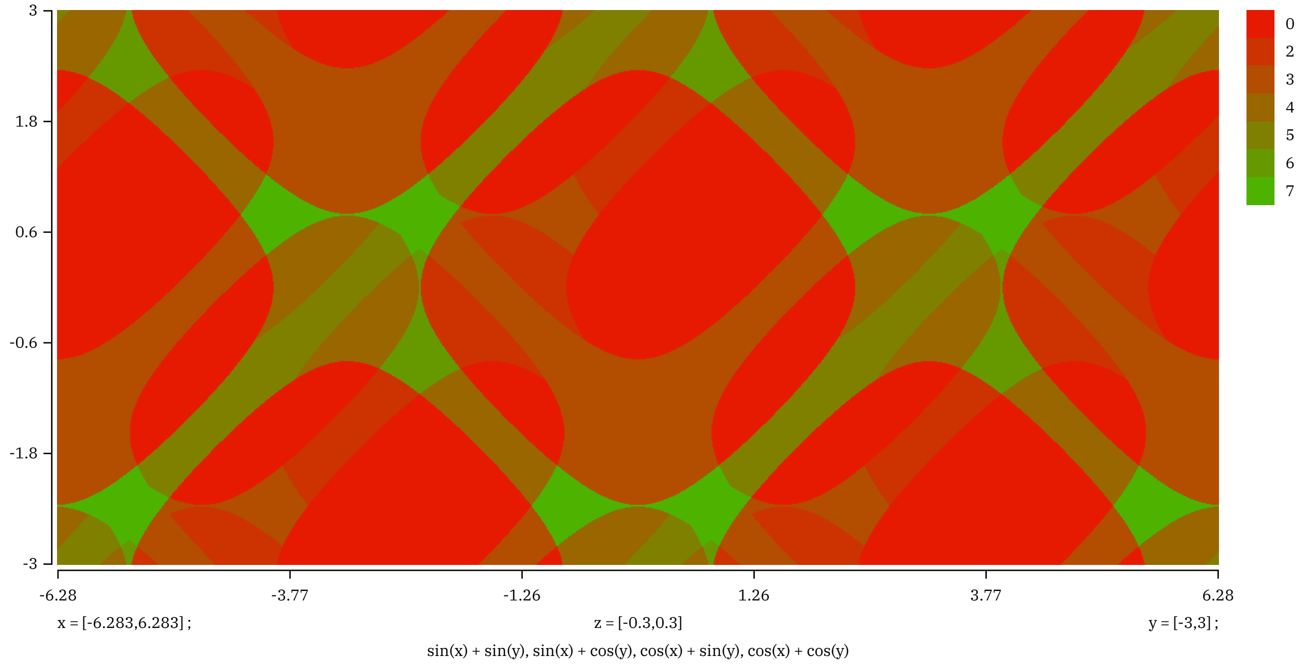
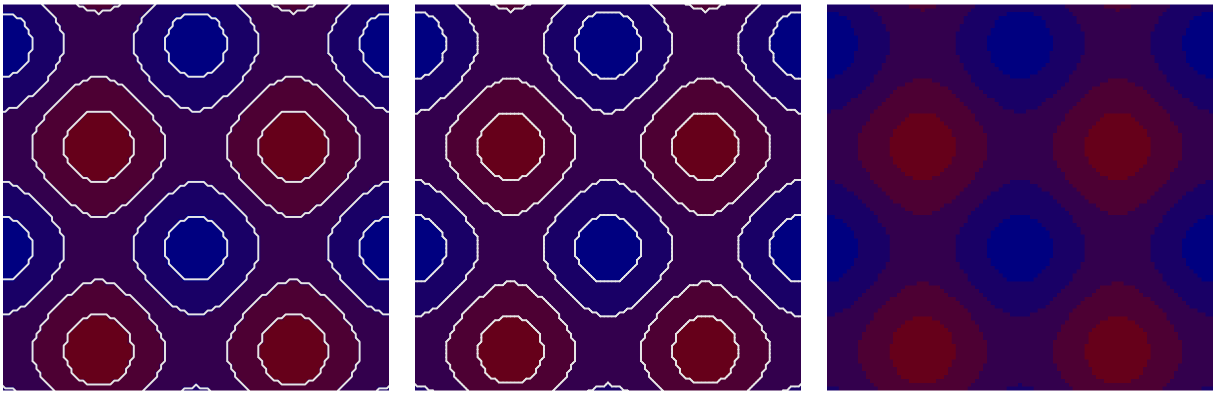


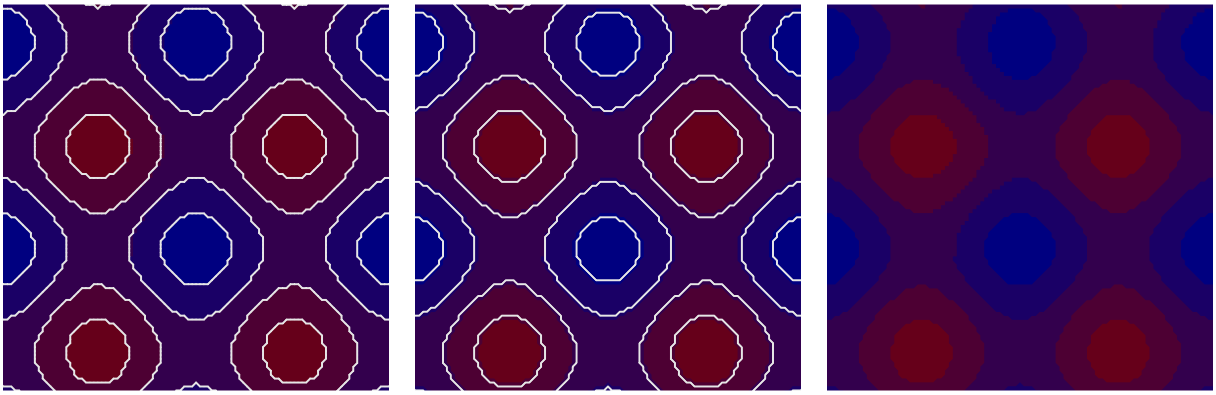
Figure 9.9

name	type	default	comment
`xmin`	`numeric`	`0`	needs to be set
`xmax`	`numeric`	`0`	needs to be set
`ymin`	`numeric`	`0`	needs to be set
`ymax`	`numeric`	`0`	needs to be set
`xstep`	`numeric`	`0`	auto 1/200 when zero
`ystep`	`numeric`	`0`	auto 1/200 when zero
`checkresult`	`boolean`	`false`	checks for overflow and NaN
`defaultnan`	`numeric`	`0`	the value to be used when NaN
`defaultinf`	`numeric`	`0`	the value to be used when overflow
`levels`	`numeric`	`10`	number of different levels to show
`level`	`numeric`		only show this level (foreground)
`preamble`	`string`		shortcuts
`function`	`string`	`x + y`	the result z value
`functions`	`list`		multiple functions (overlapping levels)
`color`	`string`	`lin(l)`	the result color value for level l (1 or 3 values)
`colors`	`numeric`		used when set
`background`	`string`	`bitmap`	band, bitmap, shape
`foreground`	`string`	`auto`	cell, edge, shape auto
`linewidth`	`numeric`	`.25`	
`linecolor`	`string`	`gray`	
`width`	`numeric`	`0`	automatic when zero
`height`	`numeric`	`0`	automatic when zero
`trace`	`boolean`	`false`	
`legend`	`string`	`all`	x y z function range all
`legendheight`	`numeric`	`LineHeight`	

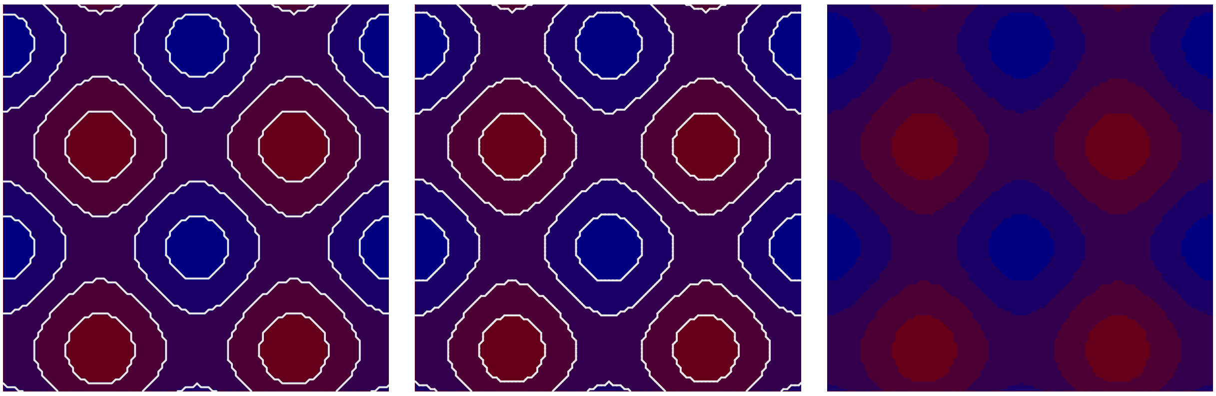
`legendwidth`	`numeric`	`LineHeight`	
`legendgap`	`numeric`	`0`	
`legenddistance`	`numeric`	`EmWidth`	
`textdistance`	`numeric`	`2EmWidth/3`	
`functiondistance`	`numeric`	`ExHeight`	
`functionstyle`	`string`		ConTEXt style name
`xformat`	`string`	`@0.2N`	number format template
`yformat`	`string`	`@0.2N`	number format template
`zformat`	`string`	`@0.2N`	number format template
`xstyle`	`string`		ConTEXt style name
`ystyle`	`string`		ConTEXt style name
`zstyle`	`string`		ConTEXt style name
`axisdistance`	`numeric`	`ExHeight`	
`axislinewidth`	`numeric`	`.25`	
`axisoffset`	`numeric`	`ExHeight/4`	
`axiscolor`	`string`	`black`	
`ticklength`	`numeric`	`ExHeight`	
`xtick`	`numeric`	`5`	
`ytick`	`numeric`	`5`	
`xlabel`	`numeric`	`5`	
`ylabel`	`numeric`	`5`	



bitmap edge bitmap cell bitmap none



shape shape shape edge shape none



band edge band cell band none

Figure 9.10

10 Surface

This is work in progress so only some examples are shown here. Yet to be decided is how we deal with axis and such.

In figure 10.1 we see an example of a plot with axis as well as lines drawn.

```
\startMPcode{doublefun}
    draw lmt_surface [
        preamble  = "local sin, cos = math.sin, math.cos",
        code      = "sin(x*x) - cos(y*y)"
        xmin      = -3,
        xmax      =  3,
        ymin      = -3,
        ymax      =  3,
        xvector   = { -0.3, -0.3 },
        height    = 5cm,
        axis      = { 40mm, 40mm, 30mm },
        clipaxis  = true,
        axiscolor = "gray",
    ] xsized .8TextWidth ;
\stopMPcode
```

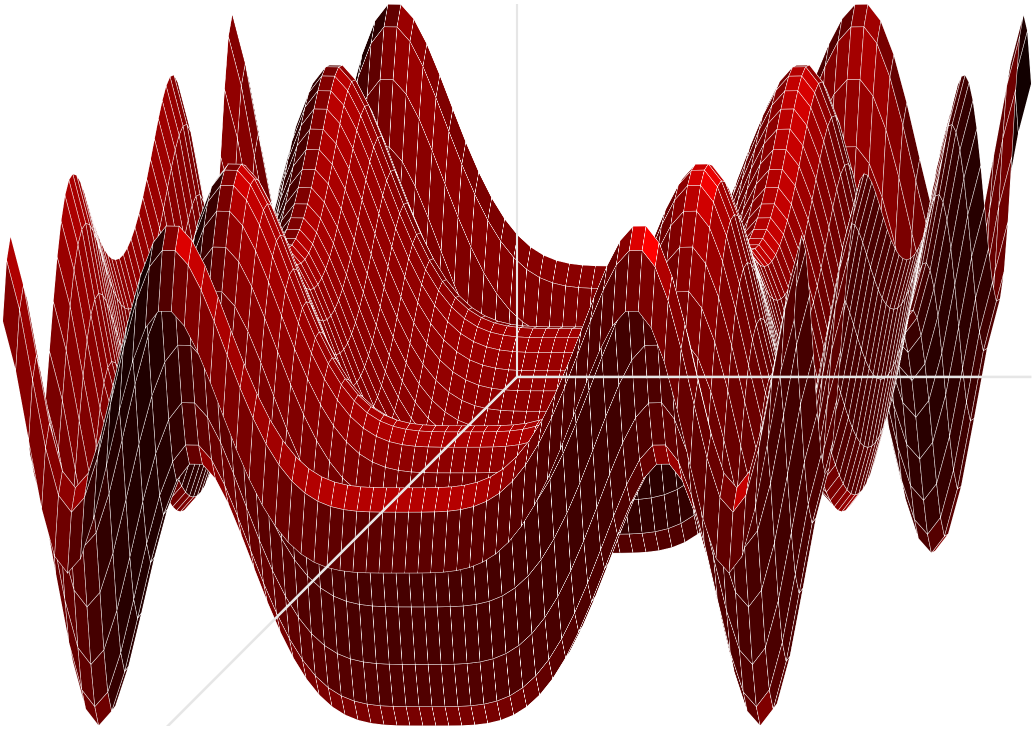


Figure 10.1

In figure 10.2 we don't draw the axis and lines. We also use a high resolution.

```
\startMPcode{doublefun}
    draw lmt_surface [
```

```
        preamble  = "local sin, cos = math.sin, math.cos",
        code      = "sin(x*x) - cos(y*y)"
        color     = "f, f/2, 1-f"
        color     = "f, f, 0"
        xstep     = .02,
        ystep     = .02,
        xvector   = { -0.4, -0.4 },
        height    = 5cm,
        lines     = false,
    ] xsized .8TextWidth ;
\stopMPcode
```

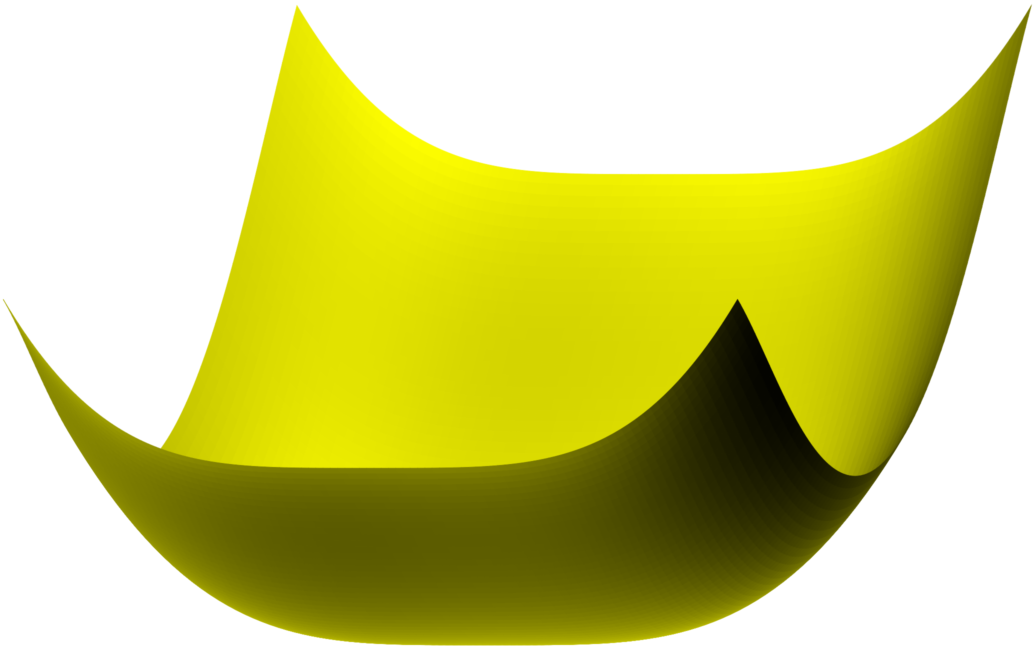


Figure 10.2

The preliminary set of parameters is:

name	**type**	**default**	**comment**	
`code`	`string`		color	string`"f, 0, 0"`
`linecolor`	`numeric`	`1`	gray scale	
`xmin`	`numeric`	`-1`		
`xmax`	`numeric`	`1`		
`ymin`	`numeric`	`-1`		
`ymax`	`numeric`	`1`		
`xstep`	`numeric`	`.1`		
`ystep`	`numeric`	`.1`		
`snap`	`numeric`	`.01`		
`xvector`	`list`	`{ -0.7, -0.7 }`		
`yvector`	`list`	`{ 1, 0 }`		
`zvector`	`list`	`{ 0, 1 }`		
`light`	`list`	`{ 3, 3, 10 }`		
`bright`	`numeric`	`100`		
`clip`	`boolean`	`false`		

lines	boolean	true
axis	list	{ }
clipaxis	boolean	false
axiscolor	string	"gray"
axislinewidth	numeric	1/2

11 Mesh

This is more a gimmick than of real practical use. A mesh is a set of paths that gets transformed into hyperlinks. So, as a start you need to enable these:

```
\setupinteraction
  [state=start,
   color=white,
   contrastcolor=white]
```

We just give a bunch of examples of meshes. A path is divided in smaller paths and each of them is part of the same hyperlink. An application is for instance clickable maps but (so far) only Acrobat supports such paths.

```
\startuseMPgraphic{MyPath1}
    fill OverlayBox withcolor "darkyellow" ;
    save p ; path p[] ;
    p1 := unitsquare xysized( OverlayWidth/4, OverlayHeight/4) ;
    p2 := unitsquare xysized(2OverlayWidth/4,3OverlayHeight/5) shifted (
      OverlayWidth/4,0) ;
    p3 := unitsquare xysized( OverlayWidth/4, OverlayHeight  ) shifted (3
      OverlayWidth/4,0) ;
    fill p1 withcolor "darkred" ;
    fill p2 withcolor "darkblue" ;
    fill p3 withcolor "darkgreen" ;
    draw lmt_mesh [ paths = { p1, p2, p3 } ] ;
    setbounds currentpicture to OverlayBox ;
\stopuseMPgraphic
```

Such a definition is used as follows. First we define the mesh as overlay:

```
\defineoverlay[MyPath1][\useMPgraphic{MyPath1}]
```

Then, later on, this overlay can be used as background for a button. Here we just jump to another page. The rendering is shown in figure 11.1.

```
\button
  [height=3cm,
   width=4cm,
   background=MyPath1,
   frame=off]
  {Example 1}
  [realpage(2)]
```

More interesting are non-rectangular shapes so we show a bunch of them. You can pass multiple paths, influence the accuracy by setting the number of steps and show the mesh with the tracing option.

```
\startuseMPgraphic{MyPath2}
    save q ; path q ; q := unitcircle xysized(OverlayWidth,OverlayHeight) ;
    save p ; path p ; p := for i=1 upto length(q) :
```

Figure 11.1

```
        (center q) -- (point (i-1) of q) -- (point i of q) -- (center q) --
    endfor cycle ;
    fill q withcolor "darkgray" ;
    draw lmt_mesh [
        trace = true,
        paths = { p }
    ] withcolor "darkred" ;

    setbounds currentpicture to OverlayBox ;
\stopuseMPgraphic

\startuseMPgraphic{MyPath3}
    save q ; path q ; q := unitcircle xysized(OverlayWidth,OverlayHeight)
      randomized 3mm ;
    fill q withcolor "darkgray" ;
    draw lmt_mesh [
        trace = true,
        paths = { meshed(q,OverlayBox,.05) }
    ] withcolor "darkgreen" ;
  % draw OverlayMesh(q,.025) withcolor "darkgreen" ;
    setbounds currentpicture to OverlayBox ;
\stopuseMPgraphic

\startuseMPgraphic{MyPath4}
    save q ; path q ; q := unitcircle xysized(OverlayWidth,OverlayHeight)
      randomized 3mm ;
    fill q withcolor "darkgray" ;
    draw lmt_mesh [
        trace = true,
        auto  = true,
        step  = 0.0125,
        paths = { q }
    ] withcolor "darkyellow" ;
    setbounds currentpicture to OverlayBox ;
\stopuseMPgraphic

\startuseMPgraphic{MyPath5}
    save q ; path q ; q := unitdiamond xysized(OverlayWidth,OverlayHeight)
      randomized 2mm ;
    q := q shifted - center q shifted center OverlayBox ;
    fill q withcolor "darkgray" ;
    draw lmt_mesh [
```

```
        trace = true,
        auto  = true,
        step  = 0.0125,
        paths = { q }
    ] withcolor "darkmagenta" ;
    setbounds currentpicture to OverlayBox ;
\stopuseMPgraphic

\startuseMPgraphic{MyPath6}
    save p ; path p[] ;
    p1 := p2 := fullcircle xysized(2OverlayWidth/5,2OverlayHeight/3) ;
    p1 := p1 shifted - center p1 shifted center OverlayBox shifted (-1
      OverlayWidth/4,0) ;
    p2 := p2 shifted - center p2 shifted center OverlayBox shifted ( 1
      OverlayWidth/4,0) ;
    fill p1 withcolor "middlegray" ;
    fill p2 withcolor "middlegray" ;
    draw lmt_mesh [
        trace = true,
        auto  = true,
        step  = 0.02,
        paths = { p1, p2 }
    ] withcolor "darkcyan" ;
    setbounds currentpicture to OverlayBox ;
\stopuseMPgraphic

\startuseMPgraphic{MyPath7}
    save p ; path p[] ;
    p1 := p2 := fullcircle xysized(2OverlayWidth/5,2OverlayHeight/3) rotated 45
      ;
    p1 := p1 shifted - center p1 shifted center OverlayBox shifted (-1
      OverlayWidth/4,0) ;
    p2 := p2 shifted - center p2 shifted center OverlayBox shifted ( 1
      OverlayWidth/4,0) ;
    fill p1 withcolor "middlegray" ;
    fill p2 withcolor "middlegray" ;
    draw lmt_mesh [
        trace = true,
        auto  = true,
        step  = 0.01,
        box   = OverlayBox enlarged -5mm,
        paths = { p1, p2 }
    ] withcolor "darkcyan" ;
    draw OverlayBox enlarged -5mm withcolor "darkgray" ;
    setbounds currentpicture to OverlayBox ;
\stopuseMPgraphic
```

This is typical a feature that, if used at all, needs some experimenting but at least the traced images look interesting enough. The six examples are shown in figure 11.2.

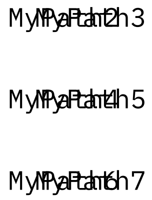


Figure 11.2

12 Function

It is tempting to make helpers that can do a lot. However, that also means that we need to explain a lot. Instead it makes more sense to have specific helpers and just make another one when needed. Rendering functions falls into this category. At some point users will come up with specific cases that other users can use. Therefore, the solution presented here is not the ultimate answer. We start with a simple example:

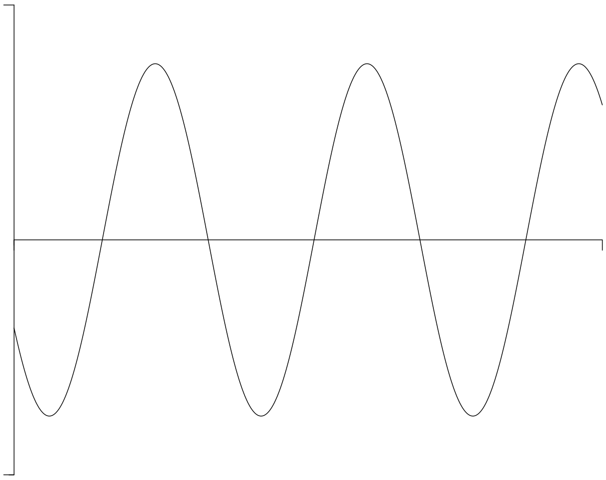


Figure 12.1

This image is defined as follows:

```
\startMPcode{doublefun}
    draw lmt_function [
        xmin =  0, xmax = 20, xstep = .1,
        ymin = -2, ymax =  2,

        sx = 1mm, xsmall = 80, xlarge = 20,
        sy = 4mm, ysmall = 40, ylarge =  4,

        linewidth = .025mm, offset = .1mm,

        code = "1.5 * math.sind (50 * x - 150)",
    ]
        xsized 8cm
    ;
\stopMPcode
```

We can draw multiple functions in one go. The next sample split the drawing over a few ranges and is defined as follows; in figure 12.2 we see the result.

```
\startMPcode{doublefun}
    draw lmt_function [
        xmin =  0, xmax = 20, xstep = .1,
        ymin = -2, ymax =  2,

        sx = 1mm, xsmall = 80, xlarge = 20,
        sy = 4mm, ysmall = 40, ylarge =  4,
```

```
        linewidth = .025mm, offset = .1mm,

        xticks    = "bottom",
        yticks    = "left",
        xlabels   = "nolimits",
        ylabels   = "yes",
        code      = "1.5 * math.sind (50 * x - 150)",
      % frame     = "ticks",
        frame     = "sticks",
        ycaption  = "\strut \rotate[rotation=90]{something vertical, using
          $\sin{x}$}",
        xcaption  = "\strut something horizontal",
        functions = {
            [ xmin =  1.0, xmax =  7.0, close = true, fillcolor = "darkred" ],
            [ xmin =  7.0, xmax = 12.0, close = true, fillcolor = "darkgreen" ],
            [ xmin = 12.0, xmax = 19.0, close = true, fillcolor = "darkblue" ],
            [
                drawcolor = "darkyellow",
                drawsize  = 2
            ]
        }
    ]
        xsized TextWidth
    ;
\stopMPcode
```

Instead of the same function, we can draw different ones and when we use transparency we get nice results too.

```
\definecolor[MyColorR][r=.5,t=.5,a=1]
\definecolor[MyColorG][g=.5,t=.5,a=1]
\definecolor[MyColorB][b=.5,t=.5,a=1]

\startMPcode{doublefun}
    draw lmt_function [
        xmin =  0, xmax = 20, xstep = .1,
        ymin = -1, ymax =  1,

        sx = 1mm, xsmall = 80, xlarge = 20,
        sy = 4mm, ysmall = 40, ylarge =  4,

        linewidth = .025mm, offset = .1mm,

        functions = {
            [
                code      = "math.sind (50 * x - 150)",
                close     = true,
                fillcolor = "MyColorR"
            ],
            [
                code      = "math.cosd (50 * x - 150)",
```

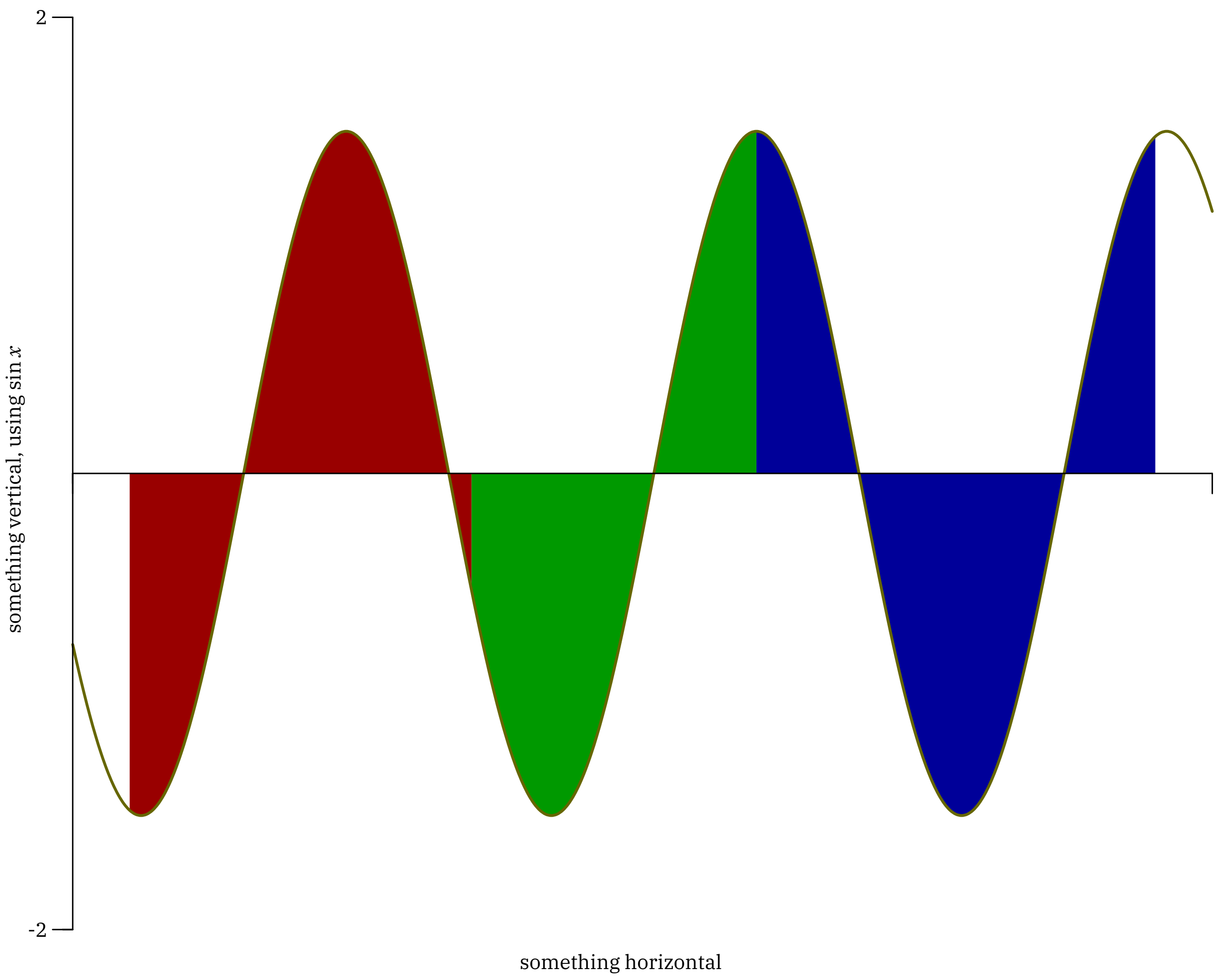


Figure 12.2

```
                    close     = true,
                    fillcolor = "MyColorB"
                ]
            },
        ]
            xsized TextWidth
        ;
\stopMPcode
```

It is important to choose a good step. In figure 12.4 we show 4 variants and it is clear that in this case using straight line segments is better (or at least more efficient with small steps).

```
\startMPcode{doublefun}
    draw lmt_function [
        xmin =  0, xmax = 10, xstep = .1,
        ymin = -1, ymax =  1,

        sx = 1mm, sy = 4mm,

        linewidth = .025mm, offset = .1mm,
```

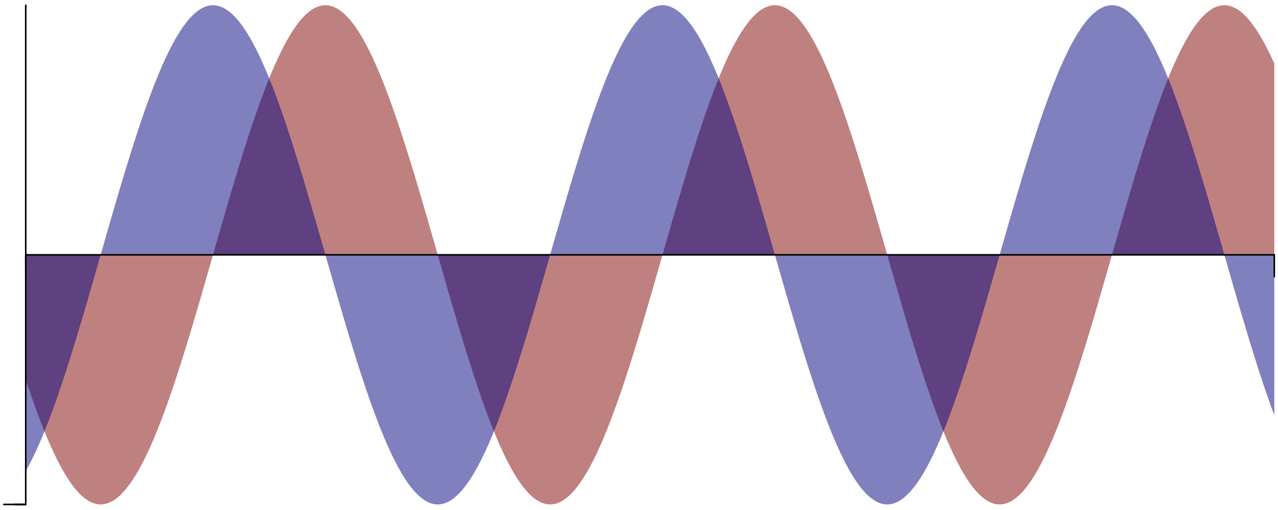


Figure 12.3

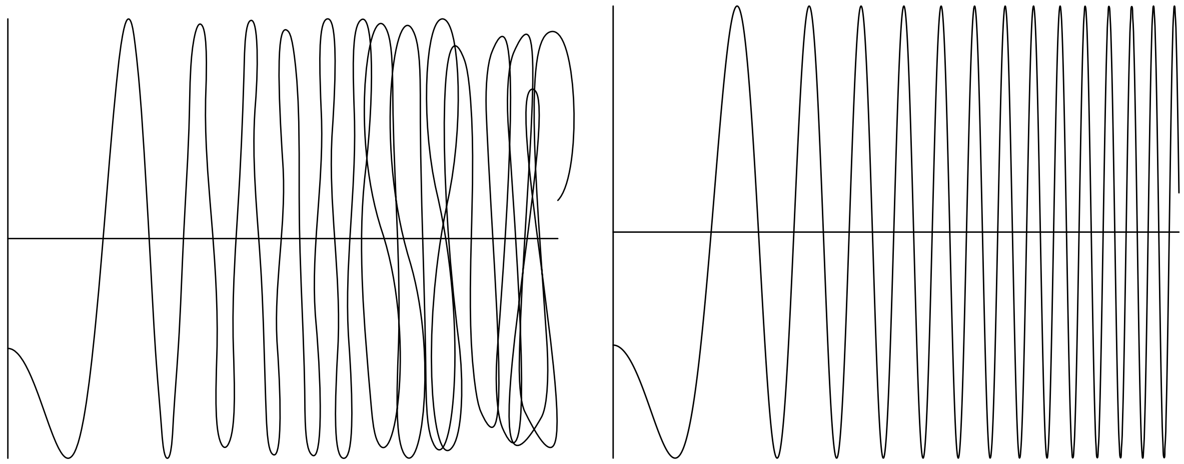
```
        code  = "math.sind (50 * x^2 - 150)",
        shape = "curve"
    ]
        xsized .45TextWidth
    ;
\stopMPcode
```

You can manipulate the axis (a bit) by tweaking the first and last ticks. In the case of figure 12.5 we also put the shape on top of the axis.

```
\startMPcode{doublefun}
    draw lmt_function [
        xfirst =  9, xlast = 21, ylarge = 2, ysmall = 1/5,
        yfirst = -1, ylast =  1, xlarge = 2, xsmall = 1/4,

        xmin = 10, xmax = 20, xstep = .25,
        ymin = -1, ymax =  1,

        drawcolor = "darkmagenta",
        shape     = "steps",
        code      = "0.5 * math.random(-2,2)",
        linewidth = .025mm,
        offset    = .1mm,
        reverse   = true,
    ]
        xsized TextWidth
    ;
\stopMPcode
```

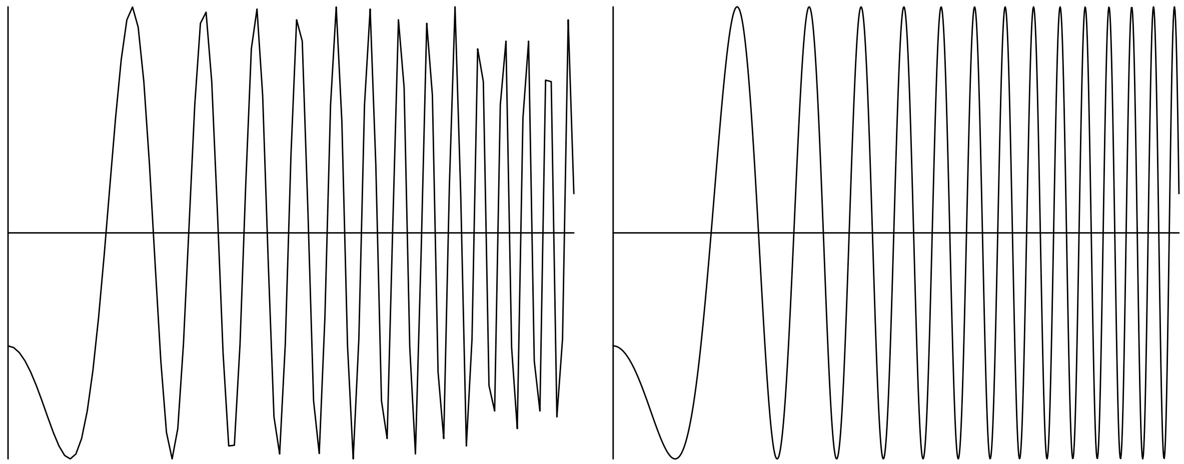
The whole repertoire of parameters (in case of doubt just check the source code as this kind of code is not that hard to follow) is:

name	type	default	comment
sx	numeric	1mm	horizontal scale factor



`xstep=.10` and `shape="curve"`

`xstep=.01` and `shape="curve"`



`xstep=.10` and `shape="line"`

`xstep=.01` and `shape="line"`

Figure 12.4

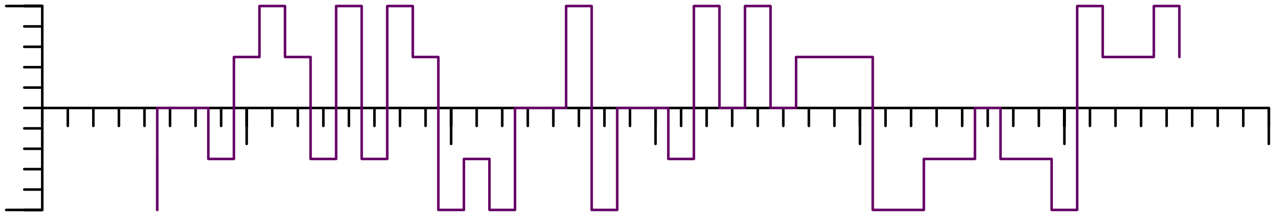


Figure 12.5

`sy`	`numeric`	`1mm`	vertical scale factor
`offset`	`numeric`	`0`	
`xmin`	`numeric`	`1`	
`xmax`	`numeric`	`1`	
`xstep`	`numeric`	`1`	
`xsmall`	`numeric`		optional step of small ticks
`xlarge`	`numeric`		optional step of large ticks
`xlabels`	`string`	`no`	yes, no or `nolimits`
`xticks`	`string`	`bottom`	possible locations are `top`, `middle` and `bottom`
`xcaption`	`string`		
`ymin`	`numeric`	`1`	

ymax	numeric	1	
ystep	numeric	1	
ysmall	numeric		optional step of small ticks
ylarge	numeric		optional step of large ticks
xfirst	numeric		left of xmin
xlast	numeric		right of xmax
yfirst	numeric		below ymin
ylast	numeric		above ymax
ylabels	string	no	yes, no or nolimits
yticks	string	left	possible locations are left, middle and right
ycaption	string		
code	string		
close	boolean	false	
shape	string	curve	or line
fillcolor	string		
drawsize	numeric	1	
drawcolor	string		
frame	string		options are yes, ticks and sticks
linewidth	numeric	.05mm	
pointsymbol	string		like type dots
pointsize	numeric	2	
pointcolor	string		
xarrow	string		
yarrow	string		
reverse	boolean	false	when true draw the function between axis and labels

In the beginning of 2025 we added support for sampled functions and parametric plots. Here are some examples but keep in mind that the interfaces might be extended.

```
\startMPcode
path p ; p := lmt_samplefunction [
    preamble  = "local tan = math.tan",
    code      = "return tan(x)",
    xmin      = -5*pi,
    xmax      =  5*pi,
    ymin      =  -10,
    ymax      =   10,
  % tolerance = 0.001,
] scaled 10 ;

draw    p withpen pencircle scaled 10   withcolor "darkred" ;
drawdot p withpen pencircle scaled  2.5 withcolor white ;
\stopMPcode
```

We draw with a thick line and show the points that make up the paths. As you can see in figure 12.6 the density is larger where more points are needed.

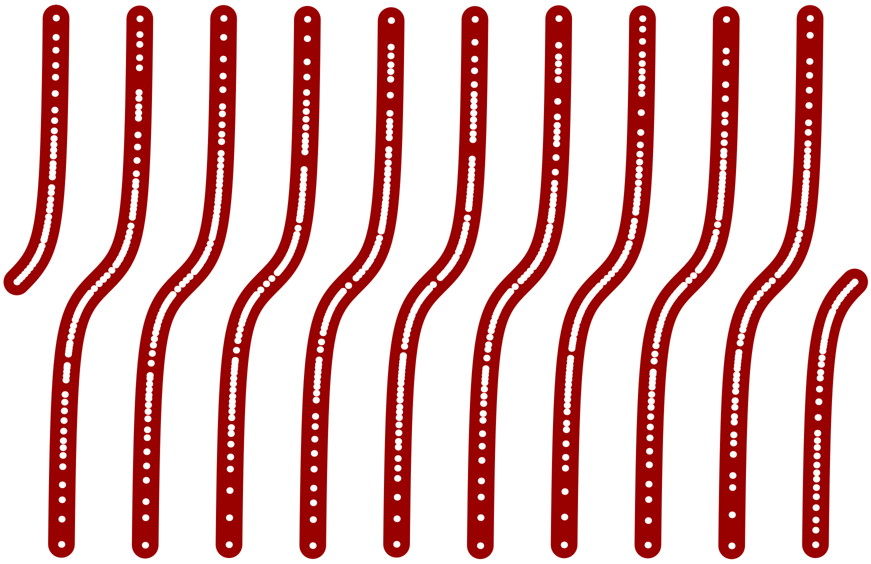


Figure 12.6

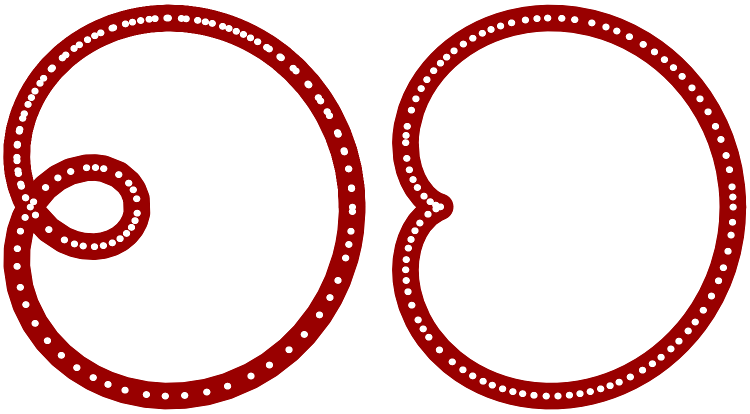
A parameteric plot runs over a range and gets two or one function passed:

```
\startMPcode
path p ; p := lmt_parametricplot [
    preamble  = "local sin, cos = math.sin, math.cos",
    xcode     = "2*cos(t/3)*cos(t)",
    ycode     = "2*cos(t/3)*sin(t)",
    tmin      =   0,
    tmax      =   4*pi,
    tolerance = 0.001,
] ysized 5cm ;

draw    p withpen pencircle scaled 10   withcolor "darkred" ;
drawdot p withpen pencircle scaled  2.5 withcolor white ;
\stopMPcode
```

and

```
\startMPcode
path p ; p := lmt_parametricplot [
    preamble  = "local sin, cos = math.sin, math.cos",
    rcode     = "1 + cos(t)",
    tmin      =   0,
    tmax      =   2*pi,
    tolerance = 0.00025,
] ysized 5cm ;

draw    p withpen pencircle scaled 10   withcolor "darkred" ;
drawdot p withpen pencircle scaled  2.5 withcolor white ;
\stopMPcode
```

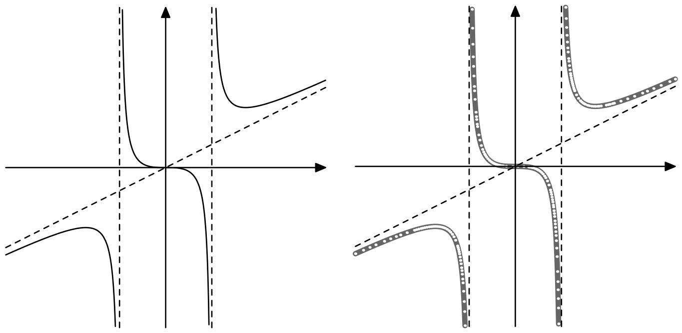
show this. The results are collected in figure ?? where again we also highlight the points that make the curve. Drawing these function is of course up to MetaPost but the calculations happen at the Lua end.



two functions one function

Figure 12.7

Figure 12.8 shows a bit more decorated example, taken from Mikaels lecture notes. The rightmost variant shows the density of the points.



untraced traced

Figure 12.8

```
% usage: \useMPgraphic{sample}{trace=0}

\startuseMPgraphic{sample}{trace}
path p ; p := lmt_samplefunction [
    code = "return x*x*x/(2*x*x - 6)", % no preamble needed
    xmin = -6,
    xmax =  6,
    ymin = -6,
    ymax =  6,
] scaled 10 ;

if \MPvar{trace} == 1 :
    draw    p withpen pencircle scaled 2 withcolor .4white ;
    drawdot p withpen pencircle scaled 1 withcolor   white ;
else :
    draw p ;
fi ;

drawarrow (-60,  0) -- (60, 0) ;
drawarrow (  0,-60) -- ( 0,60) ;
draw (( sqrt(3)*10,-60) -- ( sqrt(3)*10,60)) withdashes 2 ;
```

```
draw ((-sqrt(3)*10,-60) -- (-sqrt(3)*10,60)) withdashes 2 ;
draw ((-60,-30) -- (60,30))                  withdashes 2 ;
\stopuseMPgraphic
```

13 Chart

This is another example implementation but it might be handy for simple cases of presenting results. Of course one can debate the usefulness of certain ways of presenting but here we avoid that discussion. Let's start with a simple pie chart (figure 13.1).

```
\startMPcode
    draw lmt_chart_circle [
        samples    = { { 1, 4, 3, 2, 5, 7, 6 } },
        percentage = true,
        trace      = true,
    ] ;
\stopMPcode
```

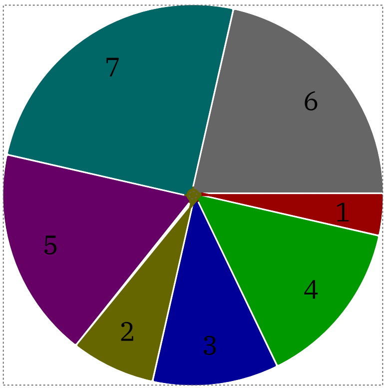


Figure 13.1

As with all these LMTX extensions, you're invited to play with the parameters. in figure 13.2 we see a variant that adds labels as well as one that has a legend.

The styling of labels and legends can be influenced independently.

```
\startMPcode
draw lmt_chart_circle [
    height      = 4cm,
    samples     = { { 1, 4, 3, 2, 5, 7, 6 } },
    percentage  = true,
    trace       = true,
    labelcolor  = "white",
    labelformat = "@0.1f",
    labelstyle  = "ttxx"
] ;
\stopMPcode
```

```
\startMPcode
draw lmt_chart_circle [
    height      = 4cm,
    samples     =  { { 1, 4, 3, 2, 5, 7, 6 } },
    percentage  = false,
    trace       = true,
```

```
    linewidth   = .125mm,
    originsize  = 0,
    labeloffset = 3cm,
    labelstyle  = "bfxx",
    legendstyle = "tfxx",
    legend      = {
        "first", "second", "third", "fourth",
        "fifth", "sixths", "sevenths"
    }
] ;
\stopMPcode
```

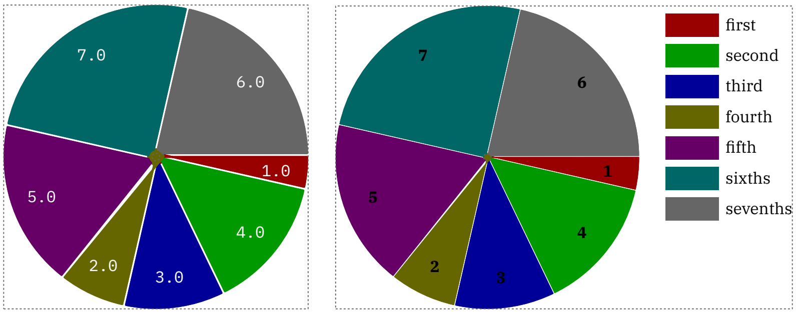


Figure 13.2

A second way of rendering are histograms, and the interface is mostly the same. In figure 13.3 we see two variants

```
\startMPcode
    draw lmt_chart_histogram [
        samples    = { { 1, 4, 3, 2, 5, 7, 6 } },
        percentage = true,
        cumulative = true,
        trace      = true,
    ] ;
\stopMPcode
```

and one with two datasets:

```
\startMPcode
    draw lmt_chart_histogram [
        samples    = {
            { 1, 4, 3, 2, 5, 7, 6 },
            { 1, 2, 3, 4, 5, 6, 7 }
        },
        background = "lightgray",
        trace      = true,
    ] ;
\stopMPcode
```

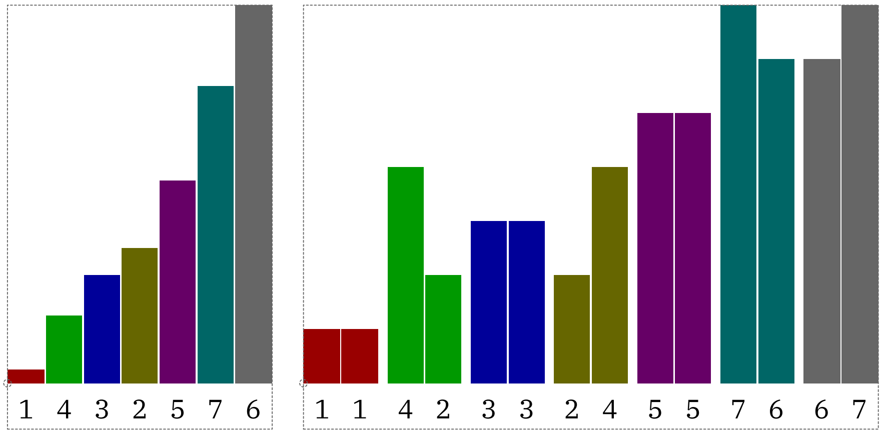


Figure 13.3

A cumulative variant is shown in figure 13.4 where we also add a background (color).

```
\startMPpage[offset=5mm]
    draw lmt_chart_histogram [
        samples         = {
            { 1, 4, 3, 2, 5, 7, 6 },
            { 1, 2, 3, 4, 5, 6, 7 }
        },
        percentage      = true,
        cumulative      = true,
        showlabels      = false,
        backgroundcolor = "lightgray",
    ] ;
\stopMPpage
```

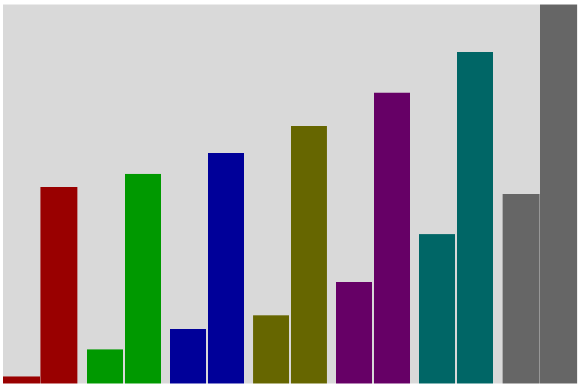


Figure 13.4

A different way of using colors is shown in figure 13.5 where each sample gets its own (same) color.

```
\startMPcode
    draw lmt_chart_histogram [
        samples    = {
            { 1, 4, 3, 2, 5, 7, 6 },
            { 1, 2, 3, 4, 5, 6, 7 }
```

```
        },
        percentage = true,
        cumulative = true,
        showlabels = false,
        background = "lightgray",
        colormode  = "local",
    ] ;
\stopMPcode
```

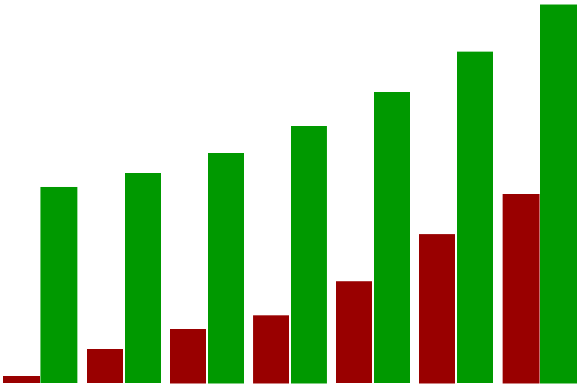


Figure 13.5

As with pie charts you can add labels and a legend:

```
\startMPcode
    draw lmt_chart_histogram [
        height          = 6cm,
        samples         = { { 1, 4, 3, 2, 5, 7, 6 } },
        percentage      = true,
        cumulative      = true,
        trace           = true,
        labelstyle      = "ttxx",
        labelanchor     = "top",
        labelcolor      = "white",
        backgroundcolor = "middlegray",
    ] ;
\stopMPcode
```

The previous and next examples are shown in figure 13.6. The height specified here concerns the graphic and excludes the labels,

```
\startMPcode
    draw lmt_chart_histogram [
        height      = 6cm,
        width       = 10mm,
        samples     = { { 1, 4, 3, 2, 5, 7, 6 } },
        trace       = true,
        maximum     = 7.5,
        linewidth   = 1mm,
        originsize  = 0,
        labelanchor = "bot",
```

```
        labelcolor  = "black"
        labelstyle  = "bfxx"
        legendstyle = "tfxx",
        labelstrut  = "yes",
        legend      = {
            "first", "second", "third", "fourth",
            "fifth", "sixths", "sevenths"
        }
    ] ;
\stopMPcode
```

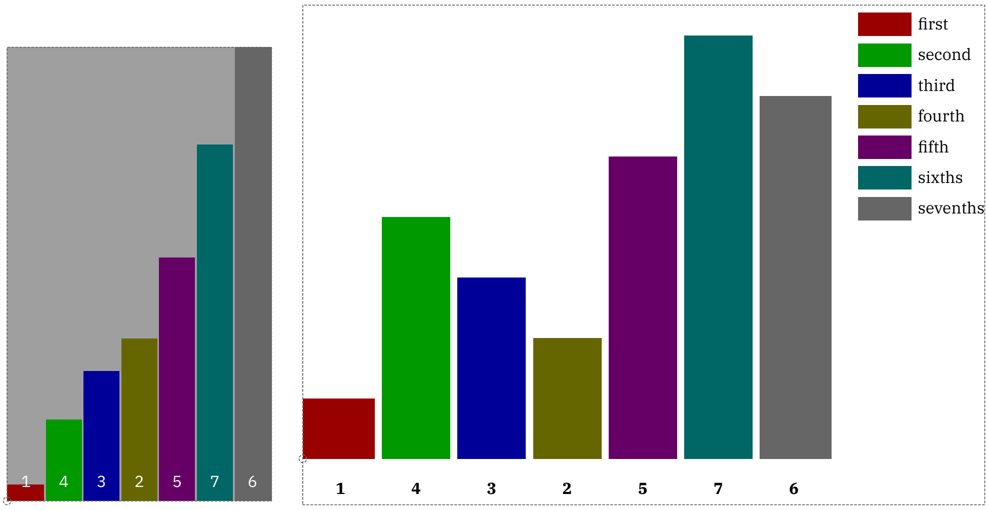


Figure 13.6

The third category concerns bar charts that run horizontal. Again we see similar options driving the rendering (figure 13.7).

```
\startMPcode
    draw lmt_chart_bar [
        samples     = { { 1, 4, 3, 2, 5, 7, 6 } },
        percentage = true,
        cumulative = true,
        trace      = true,
    ] ;
\stopMPcode
```

```
\startMPcode
    draw lmt_chart_bar [
        samples         = { { 1, 4, 3, 2, 5, 7, 6 } },
        percentage      = true,
        cumulative      = true,
        showlabels      = false,
        backgroundcolor = "lightgray",
    ] ;
\stopMPcode
```

Determining the offset of labels is manual work:

```
\startMPcode
draw lmt_chart_bar [
    width           = 4cm,
    height          = 5mm,
    samples         = { { 1, 4, 3, 2, 5, 7, 6 } },
    percentage      = true,
    cumulative      = true,
    trace           = true,
    labelcolor      = "white",
    labelstyle      = "ttxx",
    labelanchor     = "rt",
    labeloffset     = .25EmWidth,
    backgroundcolor = "middlegray",
] ;
\stopMPcode
```

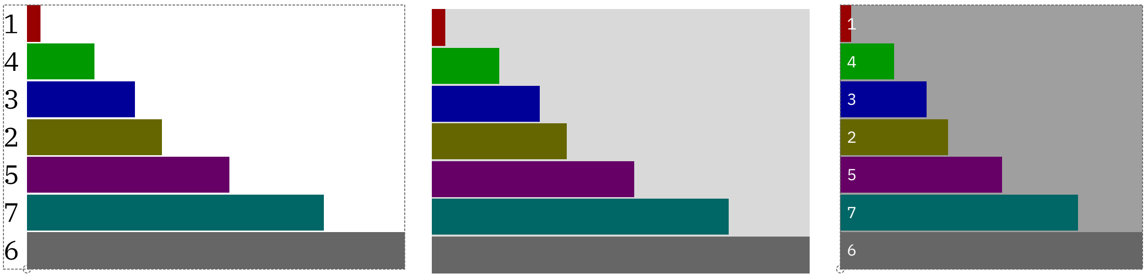


Figure 13.7

Here is one with a legend (rendered in figure 13.8):

```
\startMPcode
draw lmt_chart_bar [
    width       = 8cm,
    height      = 10mm,
    samples     = { { 1, 4, 3, 2, 5, 7, 6 } },
    trace       = true,
    maximum     = 7.5,
    linewidth   = 1mm,
    originsize  = 0,
    labelanchor = "lft",
    labelcolor  = "black"
    labelstyle  = "bfxx"
    legendstyle = "tfxx",
    labelstrut  = "yes",
    legend      = {
        "first", "second", "third", "fourth",
        "fifth", "sixths", "sevenths"
    }
] ;
\stopMPcode
```

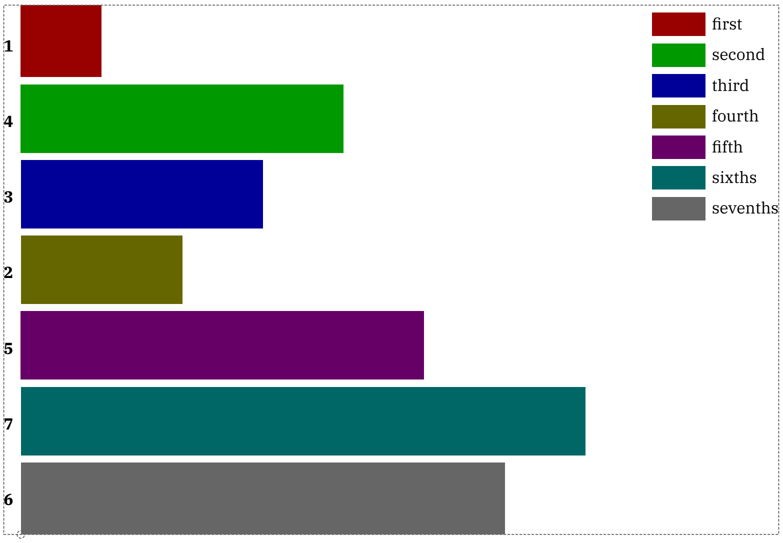


Figure 13.8

You can have labels per dataset as well as draw multiple datasets in one image, see figure 13.9:

```
\startMPcode
    draw lmt_chart_bar [
        samples = {
            { 1, 4, 3, 2, 5, 7, 6 },
            { 3, 2, 5, 7, 5, 6, 1 }
        },
        labels      = {
            { "a1", "b1", "c1", "d1", "e1", "f1", "g1" },
            { "a2", "b2", "c2", "d2", "e2", "f2", "g2" }
        },
        labeloffset = -EmWidth,
        labelanchor = "center",
        labelstyle  = "ttxx",
        trace       = true,
        center      = true,
    ] ;

    draw lmt_chart_bar [
        samples     = {
            { 1, 4, 3, 2, 5, 7, 6 }
        },
        labels      = {
            { "a", "b", "c", "d", "e", "f", "g" }
        },
        labeloffset = -EmWidth,
        labelanchor = "center",
        trace       = true,
        center      = true,
    ] shifted (10cm,0) ;
\stopMPcode
```

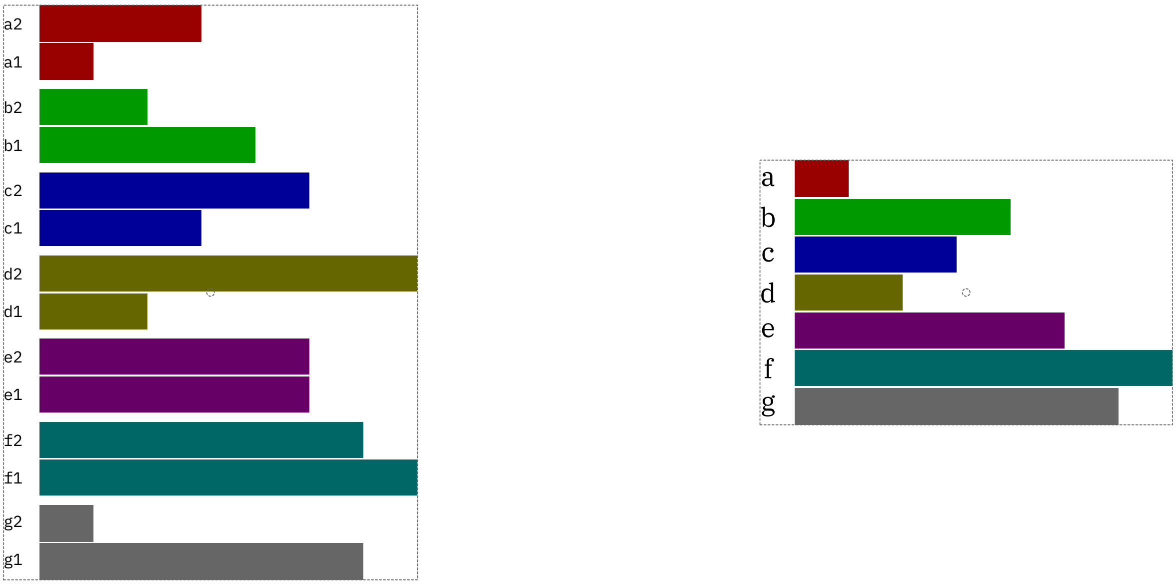


Figure 13.9

name	type	default	comment
`originsize`	`numeric`	`1mm`	
`trace`	`boolean`	`false`	
`showlabels`	`boolean`	`true`	
`center`	`boolean`	`false`	
`samples`	`list`		
	`cumulative`	`boolean`	false
`percentage`	`boolean`	`false`	
`maximum`	`numeric`	`0`	
`distance`	`numeric`	`1mm`	
`labels`	`list`		
`labelstyle`	`string`		
`labelformat`	`string`		
`labelstrut`	`string`	`auto`	
`labelanchor`	`string`		
`labeloffset`	`numeric`	`0`	
`labelfraction`	`numeric`	`0.8`	
`labelcolor`	`string`		
`backgroundcolor`	`string`		
`drawcolor`	`string`	`white`	
`fillcolors`	`list`		primary (dark) colors
`colormode`	`string`	`global`	or local
`linewidth`	`numeric`	`.25mm`	
`legendcolor`	`string`		
`legendstyle`	`string`		
`legend`	`list`		

Pie charts have:

name	default

`height`	`5cm`
`width`	`5mm`
`labelanchor`	
`labeloffset`	`0`
`labelstrut`	`no`

Histograms come with:

name	default
`height`	`5cm`
`width`	`5mm`
`labelanchor`	`bot`
`labeloffset`	`1mm`
`labelstrut`	`auto`

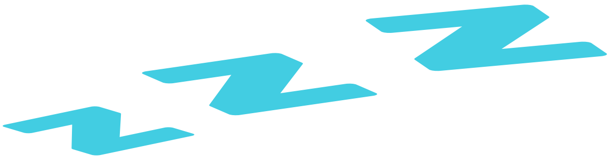
Bar charts use:

name	default
`height`	`5cm`
`width`	`5mm`
`labelanchor`	`lft`
`labeloffset`	`1mm`
`labelstrut`	`no`

14 SVG

There is not that much to tell about this command. It translates an svg image to MetaPost operators. We took a few images from a mozilla emoji font:

```
\startMPcode
    draw lmt_svg [
        filename = "mozilla-svg-002.svg",
        height   = 2cm,
        width    = 8cm,
    ] ;
\stopMPcode
```



Because we get pictures, you can mess around with them:

```
\startMPcode
    picture p ; p := lmt_svg [ filename = "mozilla-svg-001.svg" ] ;
    numeric w ; w := bbwidth(p) ;
    draw p ;
    draw p xscaled -1 shifted (2.5*w,0);
    draw p rotatedaround(center p,45) shifted (3.0*w,0) ;
    draw image (
        for i within p : if filled i :
            draw pathpart i withcolor green ;
        fi endfor ;
    ) shifted (4.5*w,0);
    draw image (
        for i within p : if filled i :
            fill pathpart i withcolor red withtransparency (1,.25) ;
        fi endfor ;
    ) shifted (6*w,0);
\stopMPcode
```

Of course. often you won't know in advance what is inside the image and how (well) it has been defined so the previous example is more about showing some MetaPost muscle.

The supported parameters are:

name	type	default	comment
`filename`	`path`		
`width`	`numeric`		
`height`	`numeric`		

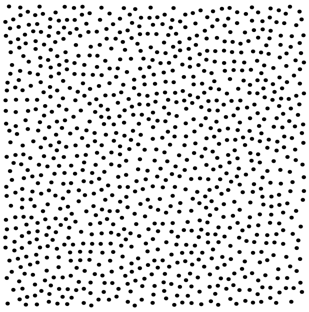
15 Poisson

When, after a post on the ConTEXt mailing list, Aditya pointed me to an article on mazes I ended up at poisson distributions which to me looks nicer than what I normally do, fill a grid and then randomize the resulting positions. With some hooks this can be used for interesting patterns too. The algorithm is based on the discussion at:

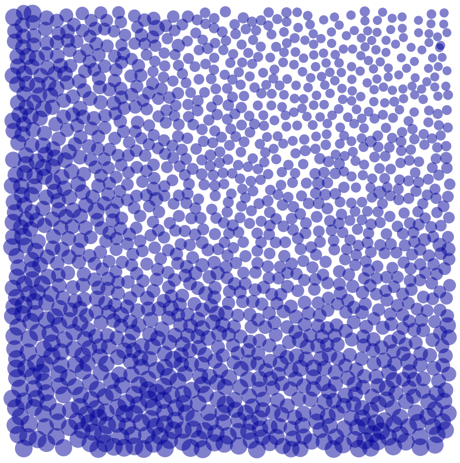
`http://devmag.org.za/2009/05/03/poisson-disk-sampling`

Other websites mention some variants on that but I saw no reason to look into those in detail. I can imagine more random related variants in this domain so consider this an appetizer. The user is rather simple because some macro is assumed to deal with the rendering of the distributed points. We just show some examples (because the interface might evolve).

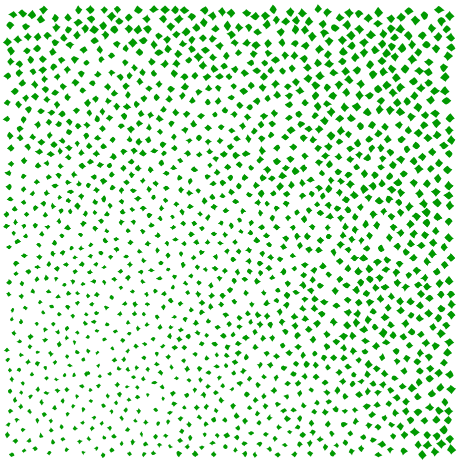
```
\startMPcode
    draw lmt_poisson [
        width    = 40,
        height   = 40,
        distance =  1,
        count    = 20,
        macro    = "draw"
    ] xsized 4cm ;
\stopMPcode
```



```
\startMPcode
    vardef tst (expr x, y, i, n) =
        fill fullcircle scaled (10+10*(i/n)) shifted (10x,10y)
            withcolor "darkblue" withtransparency (1,.5) ;
    enddef ;

    draw lmt_poisson [
        width     = 50,
        height    = 50,
        distance  =  1,
        count     = 20,
        macro     = "tst",
        arguments =  4
    ] xsized 6cm ;
\stopMPcode
```

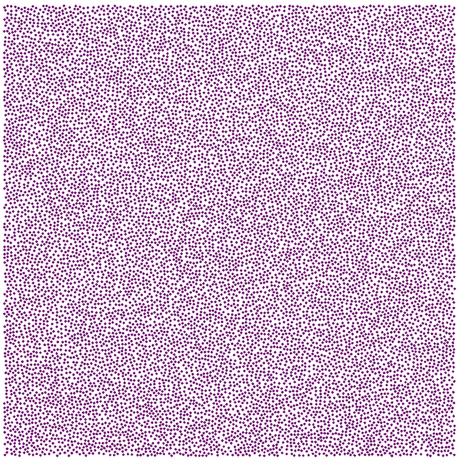


```
\startMPcode
    vardef tst (expr x, y, i, n) =
        fill fulldiamond scaled (5+5*(i/n)) randomized 2 shifted (10x,10y)
            withcolor "darkgreen" ;
    enddef ;

    draw lmt_poisson [
        width     = 50,
        height    = 50,
        distance  =  1,
        count     = 20,
        macro     = "tst",
        initialx  = 10,
        initialy  = 10,
        arguments =  4
    ] xsized 6cm ;
\stopMPcode
```

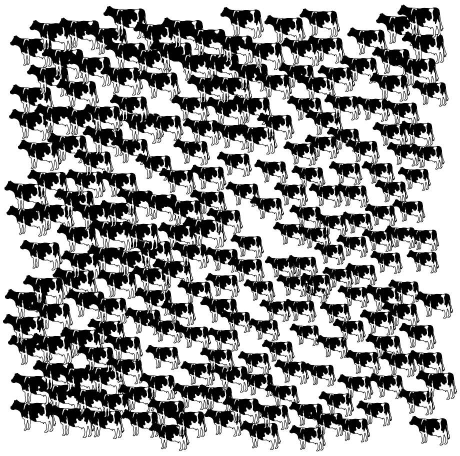


```
\startMPcode{doublefun}
    vardef tst (expr x, y, i, n) =
        fill fulldiamond randomized (.2*i/n) shifted (x,y);
    enddef ;

    draw lmt_poisson [
```

```
        width     = 150,
        height    = 150,
        distance  =   1,
        count     =  20,
        macro     = "tst",
        arguments =   4
    ] xsized 6cm withcolor "darkmagenta" ;
\stopMPcode
```



```
\startMPcode
    vardef tst (expr x, y, i, n) =
        draw externalfigure "cow.pdf" ysized (10+5*i/n) shifted (10x,10y);
    enddef ;
    draw lmt_poisson [
        width     = 20,
        height    = 20,
        distance  =  1,
        count     = 20,
        macro     = "tst"
        arguments = 4,
    ] xsized 6cm ;
\stopMPcode
```



The supported parameters are:

name	type	default	comment
`width`	`numeric`	`50`	
`height`	`numeric`	`50`	
`distance`	`numeric`	`1`	
`count`	`numeric`	`20`	
`macro`	`string`	`"draw"`	
`initialx`	`numeric`	`10`	
`initialy`	`numeric`	`10`	
`arguments`	`numeric`	`4`	

16 Fonts

Fonts are interesting phenomena but can also be quite hairy. Shapes can be missing or not to your liking. There can be bugs too. Control over fonts has always been on the agenda of TEX macro packages, and ConTEXt provides a lot of control, especially in MkIV. In LMTX we add some more to that: we bring back MetaFont's but now in the MetaPost way. A simple example shows how this is (maybe I should say: will be) done.

We define three simple shapes and do that (for now) in the `simplefun` instance because that's what is used when generating the glyphs.

```
\startMPcalculation{simplefun}
    vardef TestGlyphLB =
        image (
            fill (unitsquare xscaled 10 yscaled 16 shifted (0,-3))
                withcolor "darkred" withtransparency (1,.5)
            ;
        )
    enddef ;

    vardef TestGlyphRB =
        image (
            fill (unitcircle xscaled 15 yscaled 12 shifted (0,-2))
                withcolor "darkblue" withtransparency (1,.5)
            ;
        )
    enddef ;

    vardef TestGlyphFS =
        image (
            fill (unittriangle xscaled 15 yscaled 12 shifted (0,-2))
                withcolor "darkgreen" withtransparency (1,.5)
            ;
        )
    enddef ;
\stopMPcalculation
```

This is not that spectacular, not is the following:

```
\startMPcalculation{simplefun}
    lmt_registerglyphs [
        name  = "test",
        units = 10, % 1000
    ] ;

    lmt_registerglyph [
        category = "test",
        unicode  = 123,
        code     = "draw TestGlyphLB ;",
```

```
        width    = 10, % 1000
        height   = 13, % 1300
        depth    = 3   %  300
    ] ;

    lmt_registerglyph [
        category = "test",
        unicode  = 125,
        code     = "draw TestGlyphRB ;",
        width    = 15,
        height   = 10,
        depth    = 2
    ] ;

    lmt_registerglyph [
        category = "test",
        unicode  = "/",
        code     = "draw TestGlyphFS ;",
        width    = 15,
        height   = 10,
        depth    = 2
    ] ;

\stopMPcalculation
```

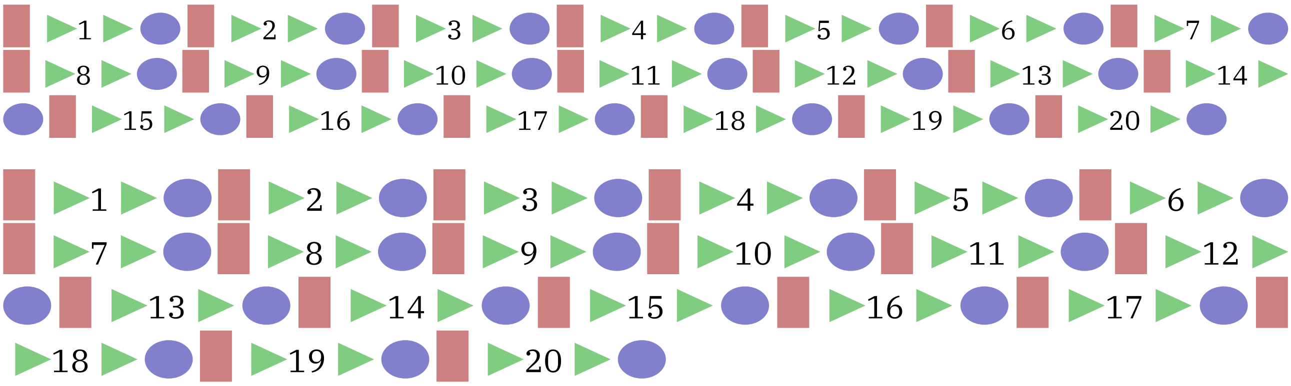
We now define a font. We always use a font as starting point which has the advantage that we always get something reasonable when we test. Of course you can use this mps font feature with other fonts too.

```
\definefontfeature[metapost][metapost=test] % or: mps={category=test}

\definefont[MyFontA][Serif*metapost @ 10bp]
\definefont[MyFontB][Serif*metapost @ 12bp]
```

These fonts can now be used:

```
\MyFontA \dorecurse{20}{\{ /#1/ \} }\par
\MyFontB \dorecurse{20}{\{ /#1/ \} }\par
```

We get some useless text but it demonstrates the idea:



If you know a bit more about ConTEXt you could think: so what, wasn't this already possible? Sure, there are various ways to achieve similar effects, but the method described here has a few advantages: it's relatively easy and we're talking about real fonts here. This means that using the shapes for characters is pretty efficient.

A more realistic example is given next. It is a subset of what is available in the ConTEXt core.

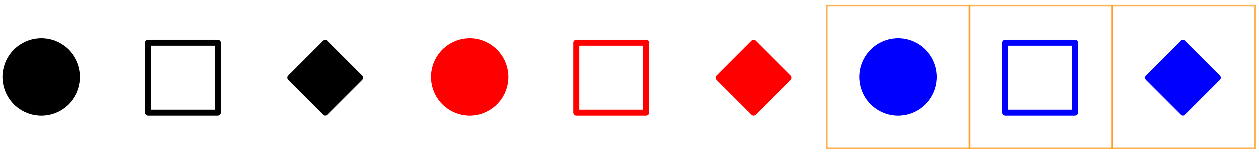
```
\startMPcalculation{simplefun}

    pen SymbolPen ; SymbolPen := pencircle scaled 1/4 ;

    vardef SymbolBullet =
        fill unitcircle scaled  3 shifted (1.5,1.5) withpen SymbolPen
    enddef ;
    vardef SymbolSquare =
        draw unitsquare scaled (3-1/16) shifted (1.5,1.5) withpen SymbolPen
    enddef ;
    vardef SymbolBlackDiamond =
        fillup unitdiamond scaled (3-1/16) shifted (1.5,1.5) withpen SymbolPen
    enddef ;
    vardef SymbolNotDef =
        draw center unitcircle
            scaled 3
            shifted (1.5,1.5)
            withpen SymbolPen scaled 4
    enddef ;

    lmt_registerglyphs [
        name     = "symbols",
        units    = 10,
        usecolor = true,
        width    = 6,
        height   = 6,
        depth    = 0,
        code     = "SymbolNotDef ;",
    ] ;

    lmt_registerglyph [ category = "symbols", unicode = "0x2022",
        code   = "SymbolBullet ;"
    ] ;
    lmt_registerglyph [ category = "symbols", unicode = "0x25A1",
        code   = "SymbolSquare ;"
    ] ;
    lmt_registerglyph [ category = "symbols", unicode = "0x25C6",
        code   = "SymbolBlackDiamond ;"
    ] ;
\stopMPcalculation
```

We could use these symbols in for instance itemize symbols. You might notice the potential difference in bullets:

```
\definefontfeature[metapost][metapost=symbols]

\definefont[MyFont] [Serif*metapost sa 1]

\startitemize[packed]
    \startitem {\MyFont                •  }\quad Regular rendering. \stopitem
    \startitem {\MyFont\red            •  }\quad Rendering with color.
      \stopitem
    \startitem {\MyFont\blue\showglyphs •  }\quad Idem but with boundingboxes
      shown. \stopitem
\stopitemize
```

- ● ▫ ◆ Regular rendering.
- ● ▫ ◆ Rendering with color.
- ● ▫ ◆ Idem but with boundingboxes shown.

When blown up, these symbols look as follows:



You can use these tricks with basically any font, so also with math fonts. However, at least for now, you need to define these before the font gets loaded.

```
\startMPcalculation{simplefun}

    pen KindergartenPen ; KindergartenPen := pencircle scaled 1 ;

    % 10 x 10 grid

    vardef KindergartenEqual =
        draw image
            (
                draw (2,6) -- (9,5) ;
                draw (2,4) -- (8,3) ;
            )
            shifted (0,-2)
            withpen KindergartenPen
            withcolor "KindergartenEqual"
    enddef ;
    vardef KindergartenPlus =
        draw image
            (
                draw (1,4) -- (9,5) ;
                draw (4,1) -- (5,8) ;
            )
            shifted (0,-2)
            withpen KindergartenPen
            withcolor "KindergartenPlus"
```

```
enddef ;
vardef KindergartenMinus =
    draw image
        (
            draw (1,5) -- (9,4) ;
        )
        shifted (0,-2)
        withpen KindergartenPen
        withcolor "KindergartenMinus"
enddef ;
vardef KindergartenTimes =
    draw image
        (
            draw (2,1) -- (9,8) ;
            draw (8,1) -- (2,8) ;
        )
        shifted (0,-2)
        withpen KindergartenPen
        withcolor "KindergartenTimes"
enddef ;
vardef KindergartenDivided =
    draw image
        (
            draw (2,1) -- (8,9) ;
        )
        shifted (0,-2)
        withpen KindergartenPen
        withcolor "KindergartenDivided"
enddef ;

lmt_registerglyphs [
    name     = "kindergarten",
    units    = 10,
  % usecolor = true,
    width    = 10,
    height   = 8,
    depth    = 2,
] ;

lmt_registerglyph [ category = "kindergarten", unicode = "0x003D",
    code = "KindergartenEqual"
] ;
lmt_registerglyph [ category = "kindergarten", unicode = "0x002B",
    code = "KindergartenPlus"
] ;
lmt_registerglyph [ category = "kindergarten", unicode = "0x2212",
    code = "KindergartenMinus"
] ;
lmt_registerglyph [ category = "kindergarten", unicode = "0x00D7",
```

```
        code = "KindergartenTimes"
    ] ;
    lmt_registerglyph [ category = "kindergarten", unicode = "0x002F",
        code = "KindergartenDivided"
    ] ;

\stopMPcalculation
```

We also define the colors. If we leave `usecolor` to true, the text colors will be taken.

```
\definecolor[KindergartenEqual]   [darkgreen]
\definecolor[KindergartenPlus]    [darkred]
\definecolor[KindergartenMinus]   [darkred]
\definecolor[KindergartenTimes]   [darkblue]
\definecolor[KindergartenDivided][darkblue]
```

```
\definefontfeature[mathextra][metapost=kindergarten]
```

Here is an example:

```
\switchtobodyfont[cambria]

$ y = 2 \times x + a - b / 3 $
```

Scaled up:

$$y = 2 \times x + a - b / 3$$

Of course this won't work out well (yet) with extensible yet, due to related definitions for which we don't have an interface yet. There is one thing that you need to keep in mind: the fonts are flushed when the document gets finalized so you have to make sure that colors are defined at the level that they are still valid at that time. So best put color definitions like the above in the document style.

This is an experimental interface anyway so we don't explain the parameters yet as there might be more of them.

Sometimes examples can be made from answers to questions on the mailing list, like the following:

```
\startMPcalculation{simplefun}
    vardef QuotationDash =
        draw image (
            interim linecap := squared ;
            save l ; l := 0.2 ;
            draw (l/2,2) -- (15-l/2,2) withpen pencircle scaled l ;
        )
    enddef ;

    lmt_registerglyphs [
        name     = "symbols",
        units    = 10,
```

```
        usecolor = true,
        width    = 15,
        height   = 2.1,
        depth    = 0,
    ] ;

    lmt_registerglyph [ category = "symbols", unicode = "0x2015", code  = "
      QuotationDash ;" ] ;
\stopMPcalculation

\definefontfeature[default][default][metapost=symbols]
```

Of course you need to figure out how to enter the equivalent of `\char "2015` and/or the font used in your editor should have that character too. Here the wide dash is about twice the `\emdash`.

17 Color

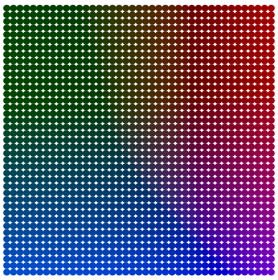
17.1 Lab colors

There are by now plenty of examples made by users that use color and MetaFun provides all kind of helpers. So do we need more? When I play around with things or when users come with questions that then result in a nice looking graphic, the result might en dup as example of coding. The following is an example of showing of colors. We have a helper that goes from a so called lab specification to rgb and it does that via xyz transformations. It makes no real sense to interface this beyond this converter. We use this opportunity to demonstrate how to make an interface.

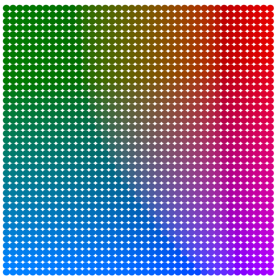
```
\startMPdefinitions
  vardef cielabmatrix(expr l, mina, maxa, minb, maxb, stp) =
    image (
      for a = mina step stp until maxa :
        for b = minb step stp until maxb :
          draw (a,b) withcolor labtorgb(l,a,b) ;
        endfor ;
      endfor ;
    )
  enddef ;
\stopMPdefinitions
```

Here we define a macro that makes a color matrix. It can be used as follows

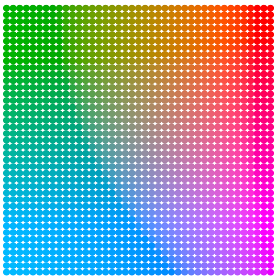
```
\startcombination[nx=4,ny=1]
  {\startMPcode draw cielabmatrix(20, -100, 100, -100, 100, 5) ysized 35mm
    withpen pencircle scaled 2.5 ; \stopMPcode} {\type {l = 20}}
  {\startMPcode draw cielabmatrix(40, -100, 100, -100, 100, 5) ysized 35mm
    withpen pencircle scaled 2.5 ; \stopMPcode} {\type {l = 40}}
  {\startMPcode draw cielabmatrix(60, -100, 100, -100, 100, 5) ysized 35mm
    withpen pencircle scaled 2.5 ; \stopMPcode} {\type {l = 60}}
  {\startMPcode draw cielabmatrix(80, -100, 100, -100, 100, 5) ysized 35mm
    withpen pencircle scaled 2.5 ; \stopMPcode} {\type {l = 80}}
\stopcombination
```



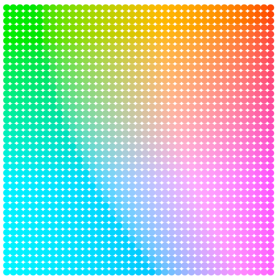
`l = 20`



`l = 40`



`l = 60`

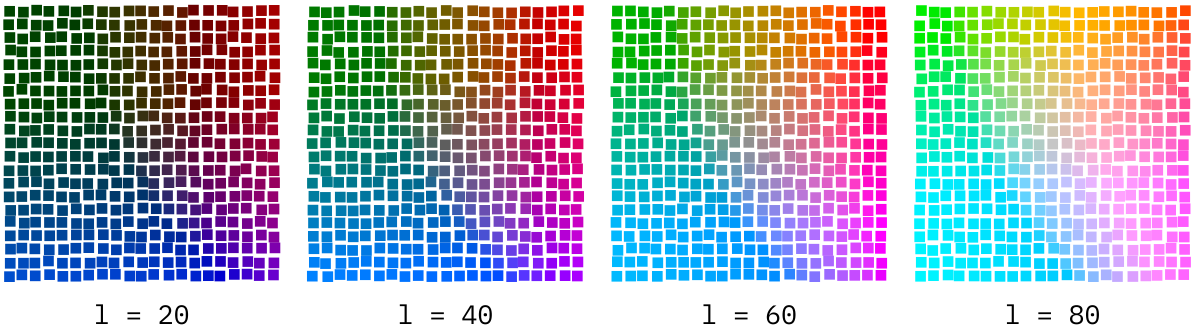


`l = 80`

One can of course mess around a bit:

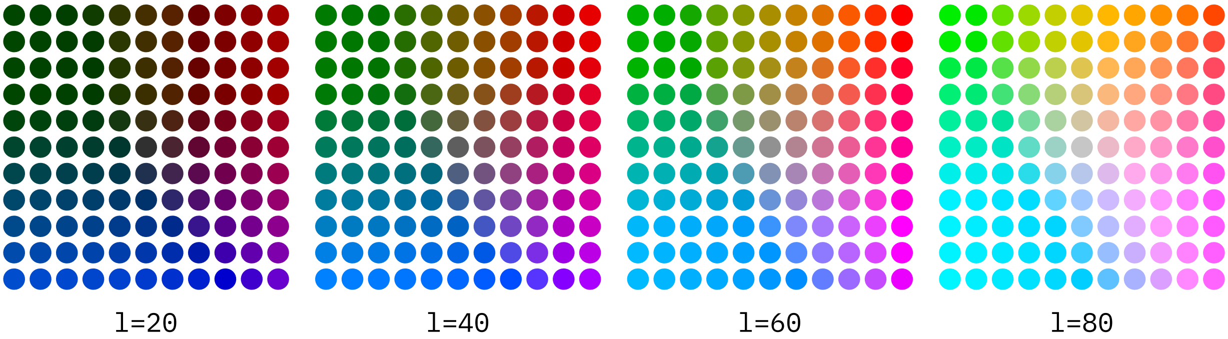
```
\startcombination[nx=4,ny=1]
```

```
  {\startMPcode draw cielabmatrix(20, -100, 100, -100, 100, 10) ysized 35mm
    randomized 1 withpen pensquare scaled 4 ; \stopMPcode} {\type {l = 20}}
  {\startMPcode draw cielabmatrix(40, -100, 100, -100, 100, 10) ysized 35mm
    randomized 1 withpen pensquare scaled 4 ; \stopMPcode} {\type {l = 40}}
  {\startMPcode draw cielabmatrix(60, -100, 100, -100, 100, 10) ysized 35mm
    randomized 1 withpen pensquare scaled 4 ; \stopMPcode} {\type {l = 60}}
  {\startMPcode draw cielabmatrix(80, -100, 100, -100, 100, 10) ysized 35mm
    randomized 1 withpen pensquare scaled 4 ; \stopMPcode} {\type {l = 80}}
\stopcombination
```



Normally, when you don't go beyond this kind of usage, a simple macro like the above will do. But when you want to make something that is upward compatible (which is one of the principles behind the ConTEXt user interface(s), you can do this:

```
\startcombination[nx=4,ny=1]
    {\startMPcode draw lmt_labtorgb [ l = 20, step = 20 ] ysized 35mm withpen
      pencircle scaled 8 ; \stopMPcode} {\type {l=20}}
    {\startMPcode draw lmt_labtorgb [ l = 40, step = 20 ] ysized 35mm withpen
      pencircle scaled 8 ; \stopMPcode} {\type {l=40}}
    {\startMPcode draw lmt_labtorgb [ l = 60, step = 20 ] ysized 35mm withpen
      pencircle scaled 8 ; \stopMPcode} {\type {l=60}}
    {\startMPcode draw lmt_labtorgb [ l = 80, step = 20 ] ysized 35mm withpen
      pencircle scaled 8 ; \stopMPcode} {\type {l=80}}
\stopcombination
```



This is a predefined macro in the reserved `lmt_` namespace (don't use that one yourself, create your own). First we preset the possible parameters:

```
presetparameters "labtorgb" [
  mina = -100,
```

```
  maxa =  100,
  minb = -100,
  maxb =  100,
  step =    5,
  l    =   50,
] ;
```

Next we define the main interface macro:

```
def lmt_labtorgb = applyparameters "labtorgb" "lmt_do_labtorgb" enddef ;
```

Last we do the actual implementation, which looks a lot like the one we started with:

```
vardef lmt_do_labtorgb =
  image (
    pushparameters "labtorgb" ;
      save l ; l := getparameter "l" ;
      for a = getparameter "mina" step getparameter "step"
            until getparameter "maxa" :
        for b = getparameter "minb" step getparameter "step"
            until getparameter "maxb" :
          draw (a,b) withcolor labtorgb(l,a,b) ;
        endfor ;
      endfor ;
    popparameters ;
  )
enddef ;
```

Of course we can now add all kind of extra features but this is what we currently have. Maybe this doesn't belong in the MetaFun core but it's not that much code and a nice demo. After all, there is much in there that is seldom used.

A perceptive color space that uses the lab model is lhc. Here is an example of how that can be used:

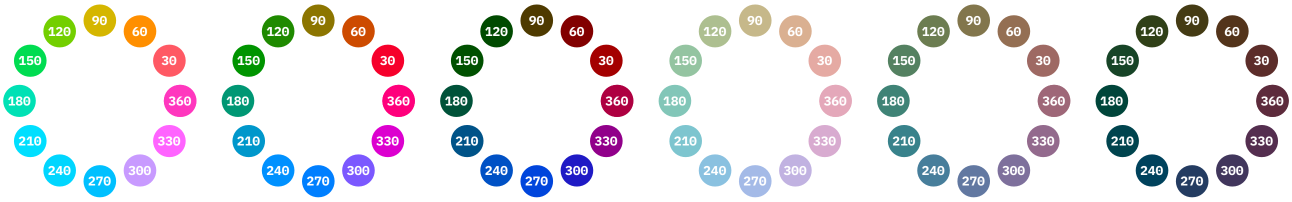
```
\startMPdefinitions
vardef lchcolorcircle(expr l, c, n) =
    image (
        save p, h ; path p ; numeric h ;
        p := arcpointlist n of fullcircle ;
        for i within p :
            h := i*360/n ;
            draw
                pathpoint scaled 50
                withpen pencircle scaled (120/n)
                withcolor lchtorgb(l,c,h) ;
            draw
                textext ("\tt\bf" & decimal h)
                scaled .4
                shifted (pathpoint scaled 50)
                withcolor white ;
        endfor ;
```

```
    )
enddef ;
\stopMPdefinitions
```

Of course you can come up with another representation than this but here is how it looks:

```
\startMPcode
draw image (
    draw lchcolorcircle(75,100,24) ;
    draw lchcolorcircle(50,100,24) scaled .75 ;
    draw lchcolorcircle(25,100,24) scaled .50 ;
) ysized 4cm ;
\stopMPcode
```

You can get rather nice color pallets by manipulating the axis without really knowing what color you get. The h value is in angles and shown inside the circles.



`l=75,c=100` `l=50,c=100` `l=25,c=100` `l=75,c=25` `l=50,c=25` `l=25,c=25`

Of course we can again wrap this into a parameter driven macro, this time `lmt_lchcircle` which accepts `l`, `c`, `steps` and a `labels` boolean.

17.2 Transparency

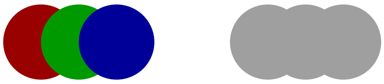
Although transparency is independent from color we discuss one aspect here. Where color is sort of native to MetaPost, especially when we talk rgb and cmyk, other color spaces are implemented using so called prescripts, think “information bound to paths and related wrappers”.

When you do this:

```
\startMPcode
path c ; c := fullcircle scaled 1cm ;
picture p ; p := image (
    fill c shifted ( 0mm,0) withcolor "darkred" ;
    fill c shifted ( 5mm,0) withcolor "darkgreen" ;
    fill c shifted (10mm,0) withcolor "darkblue" ;
) ;

draw p ; draw p shifted (3cm,0) withcolor "middlegray" ;
\stopMPcode
```

You will notice that the picture gets recolored so the color properties set on the picture are applied to separate elements that make it. A picture itself is actually just a list of objects and it has no properties of its own. A way around this is to wrap it in a group, bound or clip which basically means something:

begin, list of objects, end. By putting properties on the wrapper we can support features that apply to what gets wrapped without adapting the properties directly.



Because transparency is also implemented with prescripts we have a problem: should it apply to the wrapper or to everything? In the LuaTeX version of the MetaPost library the scripts get assigned to the first element that supports them and because there only paths can have these properties, you cannot simply change the transparency without looping over the picture and redraw it.

```
\startMPcode
picture q ; q := image (
    fill c shifted ( 0mm,0) withcolor "darkcyan"    withtransparency (1,.5);
    fill c shifted ( 5mm,0) withcolor "darkmagenta" withtransparency (1,.5);
    fill c shifted (10mm,0) withcolor "darkyellow"  withtransparency (1,.5);
) ;

draw q ; draw q shifted (3cm,0) withtransparency (1,.25) ;
\stopMPcode
```

In LuaMetaTeX we have a way to assign the properties to the elements so we get three less transparent circles:

In MkIV only the first circle becomes lighter.

```
\startMPcode
picture r ; r := image (
    draw p ;
    draw q shifted (7cm,0cm) ;
) ;

draw r ;
draw r shifted (3cm,0) withtransparency (1,.75) ;
\stopMPcode
```

This example shows that when we draw p and q we get the elements at the same level (flattened) so we can indeed apply the transparency to all of them.

So, keep in mind that this only works in MkXL and not in MkVI (unless we also upgrade LuaTeX to support this).

17.3 Surrounding color

Here is an example that shows how to make a graphic listen to the current color:

```
\startcolor[blue]
    blue before
    \startMPcode
        setbackendoption "noplugins" ;
        fill fullcircle xscaled 3EmWidth yscaled 1.5ExHeight ;
    \stopMPcode\space
    blue after
\stopcolor
```

This backend option disables *all* additional features so it will only work for for relative simple graphics. There might be more detailed control in the future.

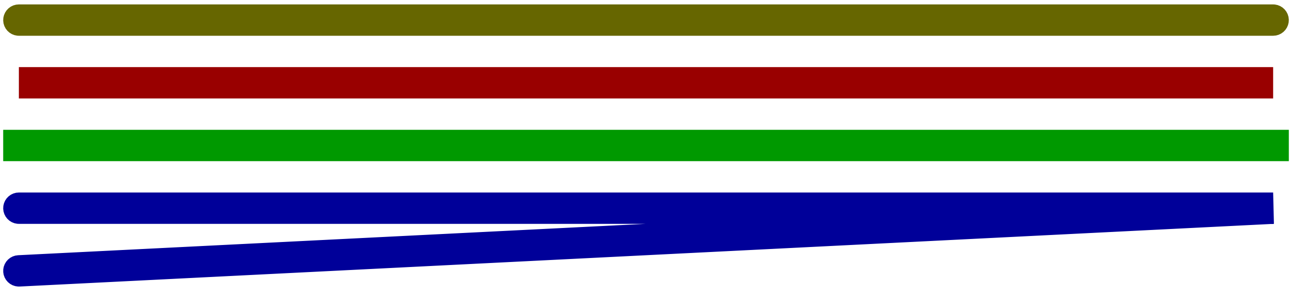
blue before  blue after

18 Lines

We assume that when you arrived here you already know how to deal with drawing lines including the way they get connected. When you draw a line some properties are controlled by variables which forces you to save existing values when you temporarily adapts them.

```
\startMPcode
draw (0, 0) -- (20, 0)                                withcolor "darkyellow" ;
draw (0,-1) -- (20,-1)           withlinecap  butt    withcolor "darkred" ;
draw (0,-2) -- (20,-2)           withlinecap  squared withcolor "darkgreen" ;
draw (0,-3) -- (20,-3) -- (0,-4) withlinejoin squared withcolor "darkblue" ;
\stopMPcode
```

These `with` features are a LuaMetaTEX extension:



19 Paths

19.1 Introduction

In the end MetaPost is all about creating (beautiful) paths. In this chapter we introduce some extensions to the engine that can be of help when constructing paths. Some relate to combining paths segments, others to generating the points.

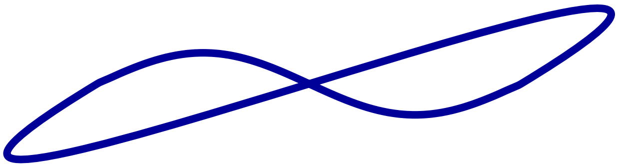
19.2 Cycles

The `cycle` commands closes a path: the end gets connected to the start. One way to construct a path stepwise is using a `for` loop, as in:

```
\startMPcode
draw (
    (0,sin(0)) for i=pi/20 step pi/20 until 2pi :
        .. (i,sin(i))
    endfor
) xysized(8cm,2cm)
withpen pencircle scaled 1mm
withcolor "darkred" ;
\stopMPcode
```

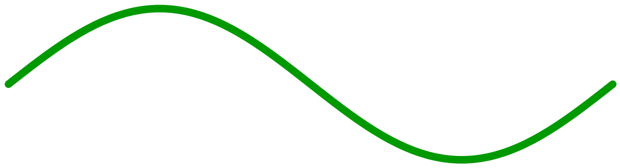
This looks kind of ugly because we need to make sure that we only put the `..` between points. If we have a closed path we can do this:

```
\startMPcode
draw (
    for i=0 step pi/20 until 2pi :
        (i,sin(i)) ..
    endfor cycle
) xysized(8cm,2cm)
withpen pencircle scaled 1mm
withcolor "darkblue" ;
\stopMPcode
```



But that is not what we want here. It is for this reason that we have a different operator, one that closes a path without cycling:

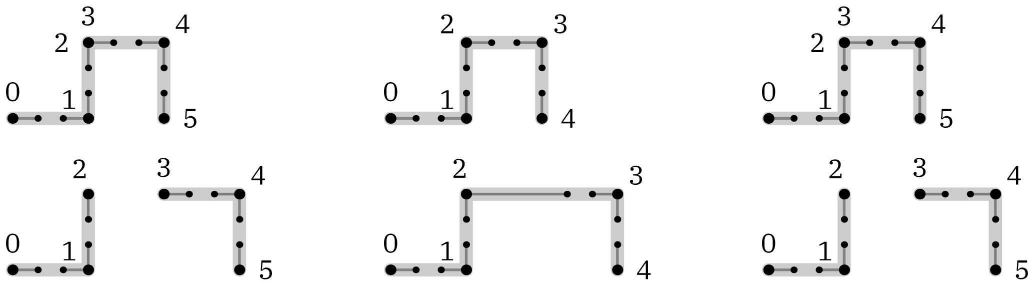
```
\startMPcode
draw (
    for i=0 step pi/20 until 2pi :
        (i,sin(i)) ..
    endfor nocycle
) xysized(8cm,2cm)
withpen pencircle scaled 1mm
withcolor "darkgreen" ;
\stopMPcode
```



19.3 Combining paths

The & concat operator requires the last point of the previous and the first point of the current path to be the same. This restriction is lifted with the &&, &&& and &&&& commands.

```
\startMPcode
def Example(expr p, q) =
    draw image (
        drawpathonly (p &&   q) shifted ( 0u,0) ;
        drawpathonly (p &&&  q) shifted ( 5u,0) ;
        drawpathonly (p &&&& q) shifted (10u,0) ;
    ) ;
enddef ;

path p[] ; numeric u ; u := 1cm ;
p[1] := (0u,0u) -- (1u,0u) -- (1u,1u) ;
p[2] := (1u,1u) -- (2u,1u) -- (2u,0u) ;

Example(p[1], p[2]) ;

Example(p[1] shifted (0u,-2u), p[2] shifted (1u,-2u)) ;
\stopMPcode
```



The precise working can be best be seen from what path we get. The single ampersand just does a concat but issues an error when the paths don't touch so we leave that one out.

```
\startMPdefinitions
path p, q, r ;
p := (0,0) -- (1,0) ;
q := (2,0) -- (3,0) ;
r := (1,0) -- (3,0) ;
vardef Example(expr p) =
  % show (p);
    drawpathonly p scaled 4cm ;
enddef ;
\stopMPdefinitions
```

This gives us:

```
(0,0)         .. controls (0.33,0) and (0.67,0) .. % p && q
(1,0) {end}   .. controls (2,    0) and (1,    0) ..
(2,0) {begin} .. controls (2.33,0) and (2.67,0) ..
(3,0)
```

```
(0,0)         .. controls (0.33,0) and (0.67,0) .. % p && r
(1,0) {end}   .. controls (1,    0) and (1,    0) ..
(1,0) {begin} .. controls (1.67,0) and (2.33,0) ..
(3,0)
```

```
(0,0)         .. controls (0.33,0) and (0.67,0) .. % p &&& q
(1,0)         .. controls (2.33,0) and (2.67,0) ..
(3,0)
```

```
(0,0)         .. controls (0.33,0) and (0.67,0) .. % p &&& r
(1,0)         .. controls (1.67,0) and (2.33,0) ..
(3,0)
```

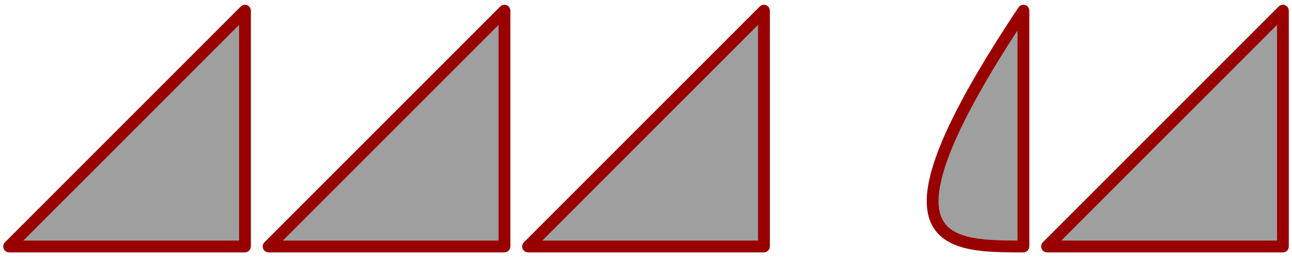
```
(0,0)         .. controls (0.33,0) and (0.67,0) .. % p &&&& q
(1,0) {end}   .. controls (2,    0) and (1,    0) ..
(2,0) {begin} .. controls (2.33,0) and (2.67,0) ..
(3,0)
```

```
(0,0)         .. controls (0.33,0) and (0.67,0) .. % p &&&& r
(1,0) {end}   .. controls (1,   0) and (1,   0) ..
(1,0) {begin} .. controls (1.67,0) and (2.33,0) ..
(3,0)
```

If we have one (concat) ampersand we check if the paths touch, error or move on. If we have three (tolerant concat) or four (tolerant append) ampersands we check if the end and begin are the same and if so, we remove one and set the controls points halfway, and then degrade to one (concat) or two (append) ampersands. Finally when (then) we have one ampersand (concat) we connect with some curl magic but when we have two (append) we connect without the curl magic: we let the left and right control points be the points.

Here is another example of usage. Watch how &&& doesn't influence an already closed curve.

```
\startMPcode
path p[] ;

p[1] := (0,0) -- (100,0) -- (100,100) ; for i=2 upto 5 : p[i] := p[1] ; endfor ;

p[1] := p[1]   -- cycle ; p[1] := p[1]   -- cycle ; p[1] := p[1]   -- cycle ;
p[2] := p[2]   -- cycle ; p[2] := p[2]  &&& cycle ; p[2] := p[2]  &&& cycle ;
p[3] := p[3]   -- cycle ; p[3] := p[3] &&&& cycle ; p[3] := p[3] &&&& cycle ;
p[4] := p[4]  &&& cycle ;
p[5] := p[5] &&&& cycle ;

for i=1 upto 5 :
  % show(p[i]) ;
    fill p[i] shifted (i*110,0) withcolor "middlegray" ;
    draw p[i] shifted (i*110,0) withcolor "darkred" withpen pencircle scaled 5  ;
endfor ;
currentpicture := currentpicture xsized TextWidth ;
\stopMPcode
```



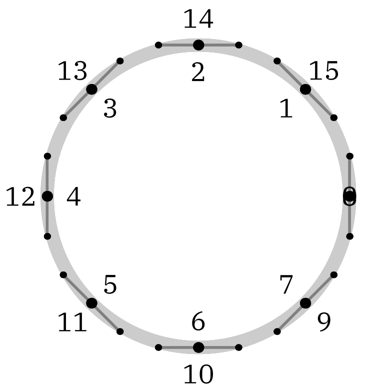
The paths are, here shown with less precision:

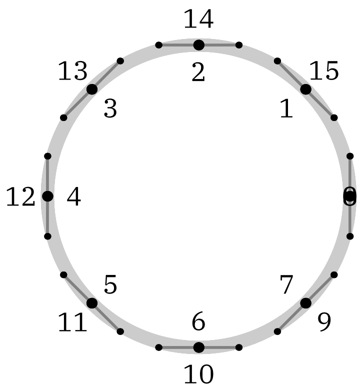
```
(0,0) .. controls (33.33,0) and (66.67,-0)
.. (100,0) .. controls (100,33.33) and (100,66.67)
.. (100,100) .. controls (66.67,66.67) and (33.33,33.33)
.. (0,0) .. controls (0,0) and (0,0)
.. (0,0) .. controls (0,0) and (0,0)
```

```
.. cycle

(0,0) .. controls (33.33,0) and (66.67,-0)
.. (100,0) .. controls (100,33.33) and (100,66.67)
.. (100,100) .. controls (66.67,66.67) and (33.33,33.33)
.. cycle

(0,0) {begin} .. controls (33.33,0) and (66.67,-0)
.. (100,0) .. controls (100,33.33) and (100,66.67)
.. (100,100) .. controls (66.67,66.67) and (33.33,33.33)
.. (0,0) {end} .. controls (0,0) and (0,0) % duplicate {end} is
.. (0,0) {end} .. controls (0,0) and (0,0) % sort of an error
.. cycle

(100,100) .. controls (33.33,0) and (66.67,-0)
.. (100,0) .. controls (100,33.33) and (100,66.67)
.. cycle

(0,0) {begin} .. controls (33.33,0) and (66.67,-0)
.. (100,0) .. controls (100,33.33) and (100,66.67)
.. (100,100) {end} .. controls (0,0) and (100,100)
.. cycle
```

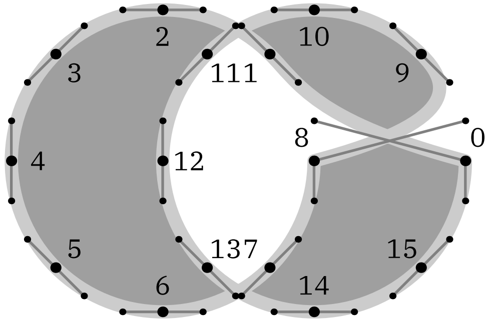
These somewhat complicated rules also relate to the intended application: the backend can apply `fill` or `eofill` in which case also cycles are involved as the following examples demonstrate:

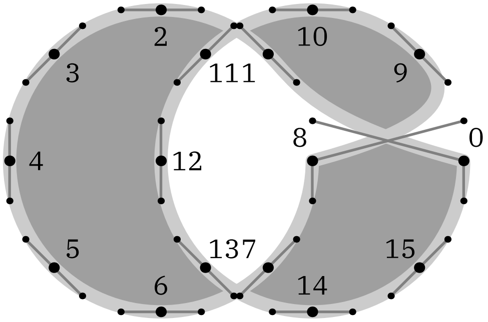
```
\startMPdefinitions
path p, q, r ;
p := fullcircle ;
q := reverse fullcircle ;
r := fullcircle shifted (1/2,0) ;
vardef Example(expr p) =
    image (
        eofill p scaled 4cm withcolor "middlegray" ;
        drawpathonly p scaled 4cm ;
    )
enddef ;
\stopMPdefinitions

\startMPcode
    draw Example(p &&&           q &&& cycle) ;
    draw Example(p &&& cycle &&& q &&& cycle) shifted (8cm,0) ;
\stopMPcode
```



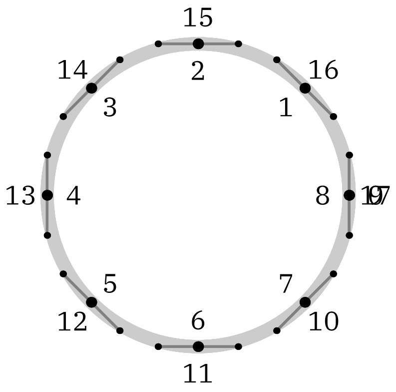


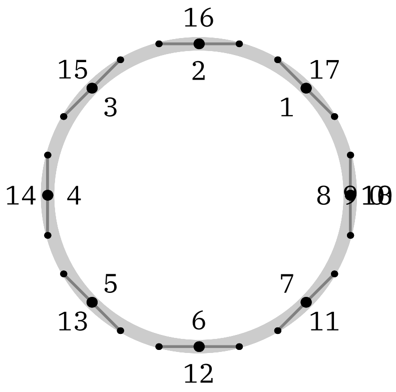
```
\startMPcode
    draw Example(p &&&             r &&& cycle) ;
    draw Example(p &&& cycle &&& r &&& cycle) shifted (8cm,0) ;
\stopMPcode
```



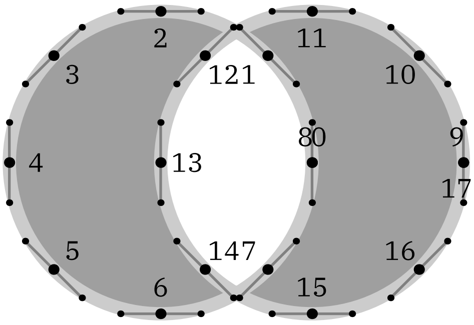


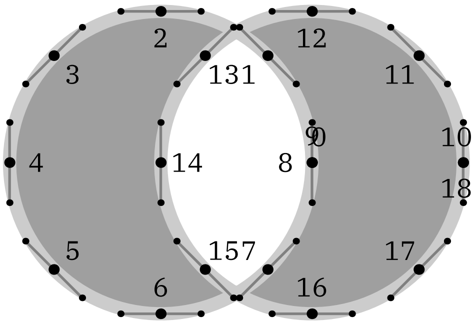
```
\startMPcode
    draw Example(p &&&&             q &&&& cycle) ;
    draw Example(p &&&& cycle &&&& q &&&& cycle) shifted (8cm,0) ;
\stopMPcode
```





```
\startMPcode
    draw Example(p &&&&             r &&&& cycle) ;
    draw Example(p &&&& cycle &&&& r &&&& cycle) shifted (8cm,0) ;
\stopMPcode
```





19.4 Implicit points

In the MetaPost library that comes with LuaMetaTEX we have a few extensions that relate to paths. You might wonder why we need these but some relate to the fact that paths can be generated programmatically. A prominent operator (or separator) is `..` and contrary to what one might expect the frequently used `--` is a macro:

```
def -- = { curl 1 } .. { curl 1 } enddef ;
```

This involves interpreting nine tokens as part of expanding the macro and in practice that is fast even for huge paths. Nevertheless we now have a `--` primitive that involves less interpreting and also avoids some intermediate memory allocation of numbers. Of course you can still define it as macro.

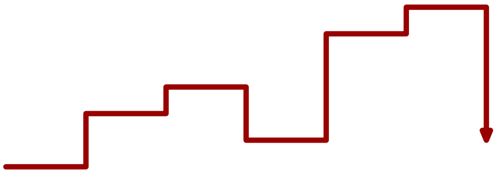
When you look at PostScript you'll notice that it has operators for relative and absolute positioning in the horizontal, vertical or combined direction. In LuaMetaTEX we now have similar operators that we will demonstrate with a few examples.

```
\startMPcode
drawarrow origin
    -- xrelative  300
    -- yrelative   20
    -- xrelative -300
    -- cycle
withpen pencircle scaled 2
withcolor "darkred" ;
\stopMPcode
```

In the next example we show a relative position combined with an absolute and we define them as macros. You basically gets what goes under the name 'turtle graphics':

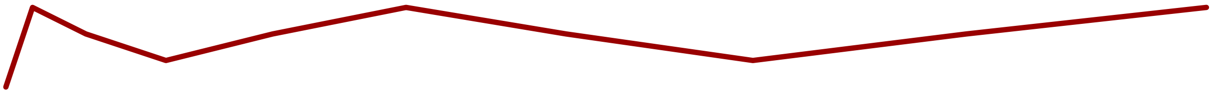
```
\startMPcode
save h ; def h = -- xrelative enddef ;
save v ; def v = -- yabsolute enddef ;

drawarrow origin
    h 30 v 20 h 30 v 30
    h 30 v 10 h 30 v 50
    h 30 v 60 h 30 v 10
```

```
withpen pencircle scaled 2
withcolor "darkred" ;
\stopMPcode
```



When you provide a pair to `xabsolute` or `yabsolute`, the xpart is the (relative) advance and the second the absolute coordinate.

```
\startMPcode
draw origin
    -- yabsolute(10,30)
    -- yabsolute(20,20)
    -- yabsolute(30,10)
    -- yabsolute(40,20)
    -- yabsolute(50,30)
    -- yabsolute(60,20)
    -- yabsolute(70,10)
    -- yabsolute(80,20)
    -- yabsolute(90,30)
withpen pencircle scaled 2
withcolor "darkred" ;
\stopMPcode
```



The `xyabsolute` is sort of redundant and is equivalent to just a pair, but maybe there is a use for it. When the two coordinates are the same you can use a numeric.

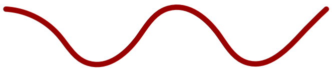
```
\startMPcode
draw origin
    -- xyabsolute(10, 10) % -- xyabsolute 10
    -- xyabsolute(20, 10)
    -- xyabsolute(30,-10)
    -- xyabsolute(40,-10)
    -- xyabsolute(50, 10)
    -- xyabsolute(60, 10)
    -- xyabsolute(70,-10)
    -- xyabsolute(80,-10)
withpen pencircle scaled 2
withcolor "darkred" ;
\stopMPcode
```

The relative variant also can take a pair and numeric, as in:

```
\startMPcode
draw origin
    -- xyrelative 10
    -- xyrelative 10
    -- xyrelative(10,-10)
    -- xyrelative(10,-10)
    -- xyrelative 10
    -- xyrelative 10
    -- xyrelative(10,-10)
    -- xyrelative(10,-10)
withpen pencircle scaled 2
withcolor "darkred" ;
\stopMPcode
```

In these examples we used `--` but you can mix in `..` and control point related operations, although the later is somewhat less intuitive here.

```
\startMPcode
draw    yabsolute(10,30)
    .. yabsolute(20,20)
    .. yabsolute(10,10)
    .. yabsolute(20,20)
    .. yabsolute(10,30)
    .. yabsolute(20,20)
    .. yabsolute(10,10)
    .. yabsolute(20,20)
    .. yabsolute(10,30)
withpen pencircle scaled 2
withcolor "darkred" ;
\stopMPcode
```



And with most features, users will likely find a use for it:

```
\startMPcode
draw for i=1 upto 5 :
    yabsolute(10,30) ---
    yabsolute(20,20) ...
    yabsolute(10,10) ---
    yabsolute(20,20) ...
endfor nocycle
withpen pencircle scaled 2
withcolor "darkred" ;
\stopMPcode
```

Here is a more impressive example, the result is shown in figure ??:

```
\startMPcode
for n=10 upto 40 :
    path p ; p := (
        for i = 0 step pi/n until pi :
            yabsolute(cos(i)^2-sin(i)^2,sin(i)^2-cos(i)^2) --
        endfor cycle
    ) ;
    draw p
        withpen pencircle scaled 1/20
        withcolor "darkred" withtransparency (1,.25) ;
endfor ;
currentpicture := currentpicture xysized (TextWidth,.25TextWidth) ;
\stopMPcode
```

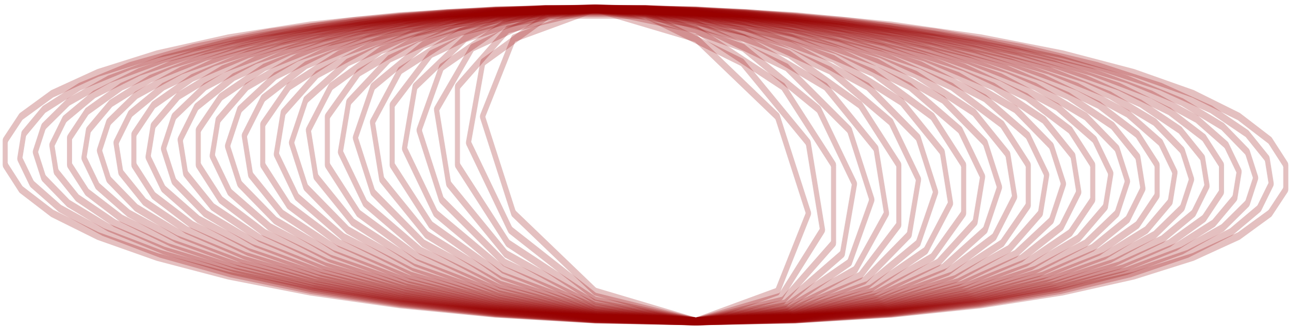


Figure 19.1 Combined relative x and absolute y positioning

19.5 Control points

Most users will create paths by using `..`, `...`, `--` and `---` and accept what they get by the looks. If your expectations are more strict you might use `tension` or `curl` with directions and vectors for the so called control points between connections. In figure 19.2 you see not only `controls` in action but also two operators that can be used to set the first and second control point. For the record: if you use `controls` without and the singular pair will be used for both control points.

```
\startMPcode
path p, q, r, s ;

p = origin {dir 25} .. (80,0) ..      controls ( 80, 0) and (100,40) .. (140,30)
  .. {dir 0} (180,0) ;
q = origin {dir 25} .. (80,0) ..      controls (100,40) and (140,30) .. (140,30)
  .. {dir 0} (180,0) ;
r = origin {dir 25} .. (80,0) .. secondcontrol              (100,40) .. (140,30)
  .. {dir 0} (180,0) ;
s = origin {dir 25} .. (80,0) ..  firstcontrol (100,40)              .. (140,30)
  .. {dir 0} (180,0) ;
```

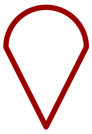
```
def Example(expr p, t, c) =
    draw p ;
    drawpoints p withcolor "middlegray" ;
    drawcontrollines p withpen pencircle scaled .3 withcolor c ;
    drawcontrolpoints p withpen pencircle scaled 2  withcolor c ;
    label.lft("\smallinfofont current", point 1 of p) ;
    label.top("\smallinfofont next", point 2 of p) ;
    draw thetextext.rt("\infofont path " & t, (point 3 of p) shifted (5,0)) ;
enddef ;

draw image (
    Example(p, "p", "darkred")  ; currentpicture := currentpicture yshifted 50 ;
    Example(q, "q", "darkblue") ; currentpicture := currentpicture yshifted 50 ;
    Example(r, "r", "darkred")  ; currentpicture := currentpicture yshifted 50 ;
    Example(s, "s", "darkblue") ; currentpicture := currentpicture yshifted 50 ;
) xsized TextWidth ;
\stopMPcode
```

19.6 Arcs

In PostScript and svg we have an arc command but not in MetaPost. In LMTX we provide a macro that does something similar:

```
\startMPcode
draw
    (0,0) --
    (arc(0,180) scaled 30 shifted (0,30)) --
    cycle
withpen pencircle scaled 2
withcolor "darkred" ;
\stopMPcode
```

The result is not spectacular:



Instead of a primitive with five arguments and the prescribed line drawn from the current point to the beginning of the arc we just use `..`, `scaled` for the radius, and `shifted` for the origin. It actually permits more advanced trickery.

```
\startMPcode
draw
    (0,0) ..
    (arc(30,240) xscaled 60 yscaled 30 shifted (0,30)) ..
    cycle
withpen pencircle scaled 2
withcolor "darkred" ;
```

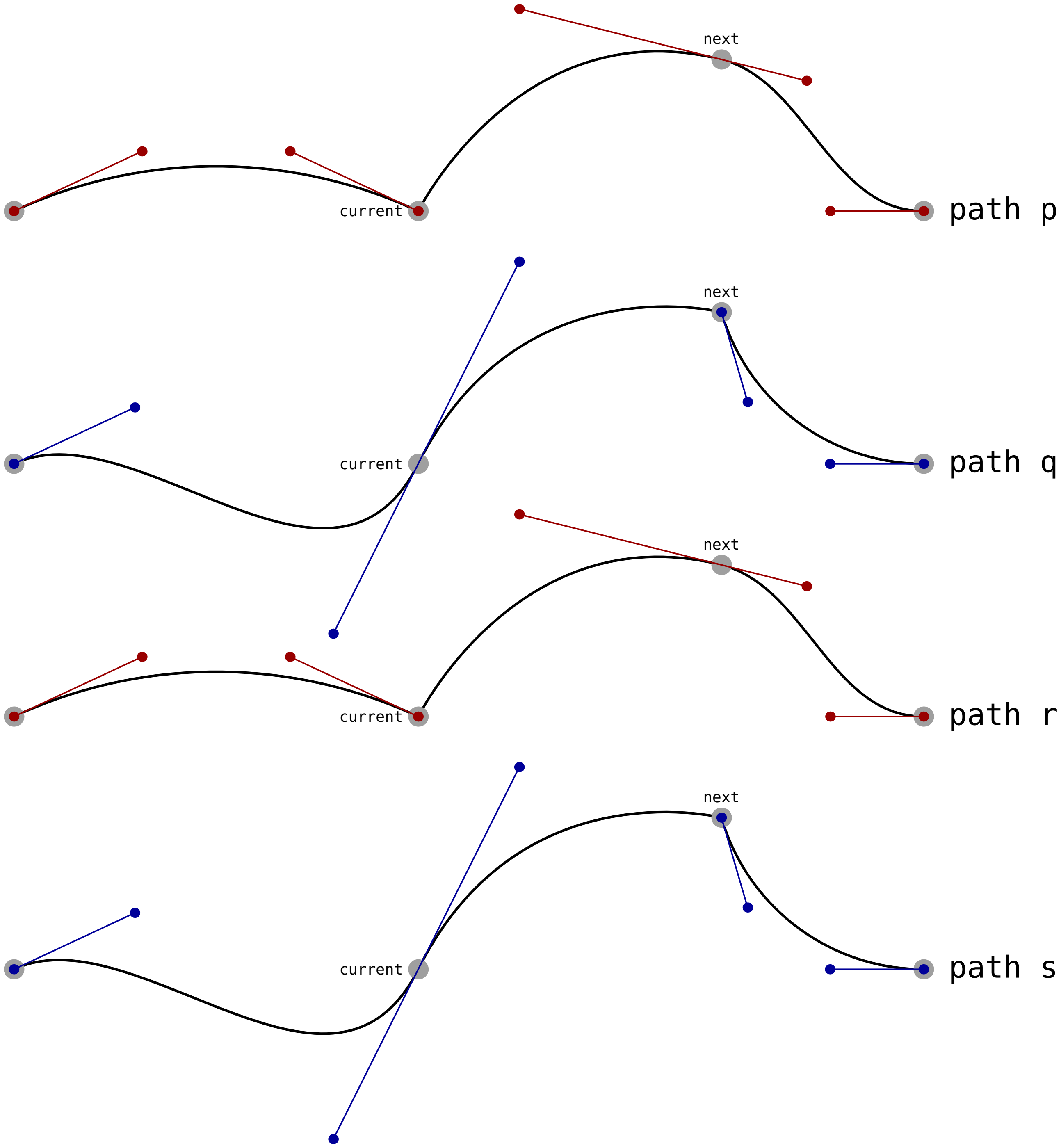
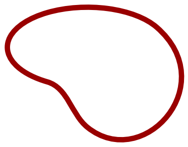


Figure 19.2 Three ways to set the control points.

```
\stopMPcode
```

Here time we get smooth connections:



but because we scale differently also a different kind of arc: it is no longer a circle segment, which is often the intended use of arc.

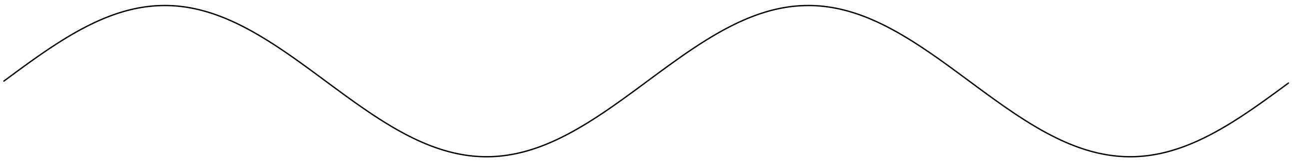
19.7 Loops

The MetaPost program is a follow up on MetaFont, which primary target was to design fonts. The paths that make op glyphs are often not that large and because in most cases we don't know in advance how large a path is they are implemented as linked lists. Now consider a large paths, with say 500 knots. The following assignment:

```
pair a ; a := point 359 of p ;
```

has to jump across 358 knots before it reaches the requested point. Let's take an example of drawing a function by (naively) stepping over values:

```
\startMPcode
path p ; p := for i=0 step 4pi/500 until 4pi: (i,sin(i)) -- endfor nocycle ;
p := p xysized(TextWidth,2cm) ;
draw p ;
\stopMPcode
```



Of course we can just calculate the point directly but here we just want to illustrate a problem.

```
\startMPcode
draw p ; for i=0 step 5 until length(p) :
    drawdot point i of p withpen pencircle scaled 2 ;
endfor ;
\stopMPcode
```

For 500 points, on a modern computer running over the list is rather fast but when we are talking 5000 points is gets noticeable, and given what MetaPost is used for, having many complex graphics calculated at runtime can have some impact on runtime.

Of course we can just calculate the point directly but here we just want to illustrate a problem. Where the previous loop takes 0.002 seconds, the second loop needs 0.001 seconds:

```
\startMPcode
pair p ; for i within p :
   if i mod 5 == 0 :
       drawdot pathpoint withpen pencircle scaled 2 ;
   fi ;
endfor ;
\stopMPcode
```

These numbers are for assigning the point to a pair variable so that we don't take into account the extra drawing (and backend) overhead. The difference in runtime can be neglected but what if we go to 5000 points? Not unsurprisingly we go down from 0.142 seconds to 0.004 seconds. There are plenty examples where runtime can be impacted, for instance when one first takes the `xpart point i` and then the `ypart point i`.

One motivation for adding a more efficient loop for paths is that in generative art one has such long parts and drawing that took tens of minutes or more now can be generated in seconds. Another motivation is in analyzing and manipulating paths. In that case we also need access to the control points and maybe even preceding or succeeding points. In figure 19.3 we show the output of the following code:

```
\startMPcode
path p ; p := fullcircle scaled 10cm ;
fill p withcolor "darkred" ;
draw p withpen pencircle scaled 1mm withcolor "middleblue" ;

for i within p :
    draw pathpoint       withpen pencircle scaled 4mm withcolor "middlegray" ;
    draw pathprecontrol  withpen pencircle scaled 2mm withcolor "middlegreen" ;
    draw pathpostcontrol withpen pencircle scaled 2mm withcolor "middlegreen" ;
    draw textext("\ttbf" & decimal i) shifted .6[deltapoint -2,origin] withcolor
      white ;
    draw textext("\ttbf" & decimal i) shifted .4[pathpoint    ,origin] withcolor
      white ;
    draw textext("\ttbf" & decimal i) shifted .2[deltapoint  2,origin] withcolor
      white ;
endfor ;
\stopMPcode
```

The MetaPost library in LuaMetaTEX uses double linked lists for paths so going back and forward is a rather cheap operation.

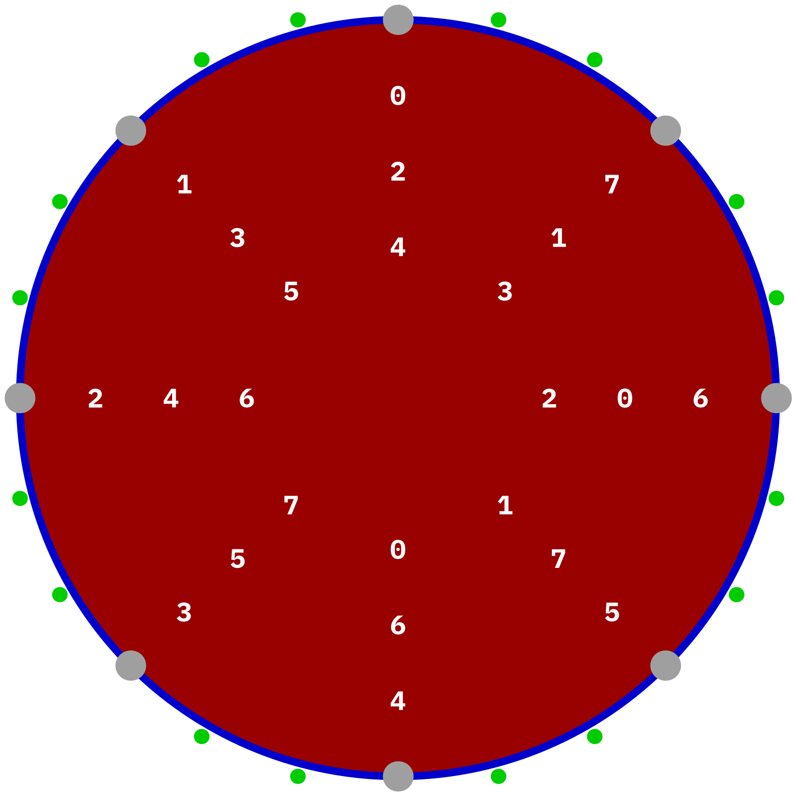


Figure 19.3 Fast looping over paths.

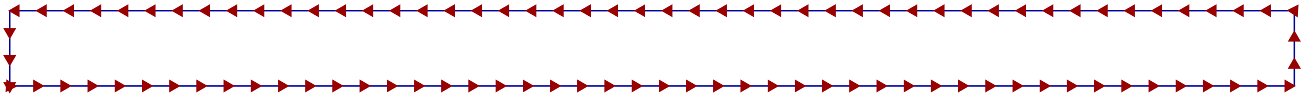
A nice application of this feature is the following, where we use yet another point property, `pathdirection`:

```
vardef dashing (expr pth, shp, stp) =
    for i within arcpointlist stp of pth :
        shp
            rotated angle(pathdirection)
            shifted pathpoint
        &&
    endfor nocycle
enddef ;
```

With:

```
\startMPcode
path p ; p := unitsquare xysized (TextWidth,1cm) ;
draw p withpen pencircle scaled .2mm withcolor darkblue ;
fill dashing (p, triangle scaled 1mm, 100) && cycle withcolor "darkred" ;
\stopMPcode
```

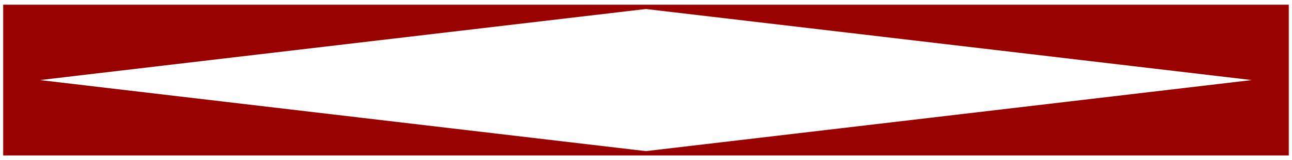
we get:



It is worth noticing that the path returned by dashing is actually a combined path where the pen gets lifted between the subpaths. This is what the && does. The `nocycle` is there to intercept the last ‘connector’ (which of course could also have been a `--` or `...` So we end up with an open path, which why in case of a fill we need to close it by `cycle`. In the next example we show all the accessors:

```
\startMPcode[instance=scaledfun]
path p; p := (fullsquare scaled 3 && fullsquare rotated 45 scaled 2 && cycle) ;

for i within p :
    message(
            "index "         & decimal  pathindex
        & ", lastindex "   & decimal  pathlastindex
        & ", length "      & decimal  pathlength
        & ", first "       & if       pathfirst : "true" else : "false" fi
        & ", last "        & if       pathlast  : "true" else : "false" fi
        & ", state "       & decimal  pathstate % end/begin subpath
        & ", point "       & ddecimal pathpoint
        & ", postcontrol " & ddecimal pathprecontrol
        & ", precontrol "  & ddecimal pathpostcontrol
        & ", direction "   & ddecimal pathdirection
        & ", delta "       & ddecimal deltapoint 1
    );
endfor ;

eofill p xysized (TextWidth, 2cm) withcolor "darkred" ;
\stopMPcode
```

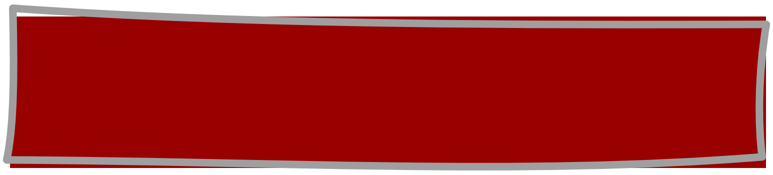
If you want to see the messages you need to process it yourself, but this is how the ten point shape looks like:



19.8 Randomized paths

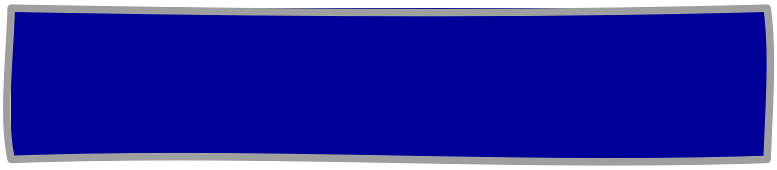
When randomizing a path the points move and when such a path has to bound a specific areas that can result in overlap which what is bounded.

```
\startMPcode
path p ; p := fullsquare xyscaled (10cm,2cm) ;
fill p withcolor "darkred" ;
draw p randomized 3mm withpen pencircle scaled 1mm withcolor "middlegray";
setbounds currentpicture to p ;
\stopMPcode
```

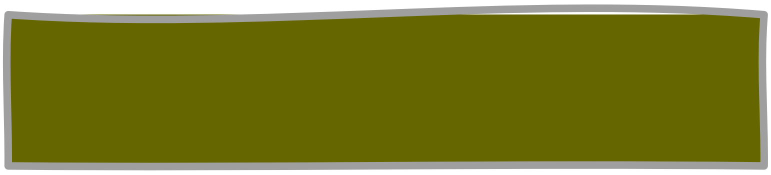


Here are two variants that randomize a path but keep the points where they are. They might be better suited for cases where there is text within the area.

```
\startMPcode
path p ; p := fullsquare xyscaled (10cm,2cm) ;
fill p withcolor "darkblue" ;
draw p randomizedcontrols 3mm withpen pencircle scaled 1mm withcolor "middlegray
  ";
setbounds currentpicture to p ;
\stopMPcode
```



```
\startMPcode
path p ; p := fullsquare xyscaled (10cm,2cm) ;
fill p withcolor "darkyellow" ;
draw p randomrotatedcontrols 15 withpen pencircle scaled 1mm withcolor "
  middlegray";
setbounds currentpicture to p ;
\stopMPcode
```

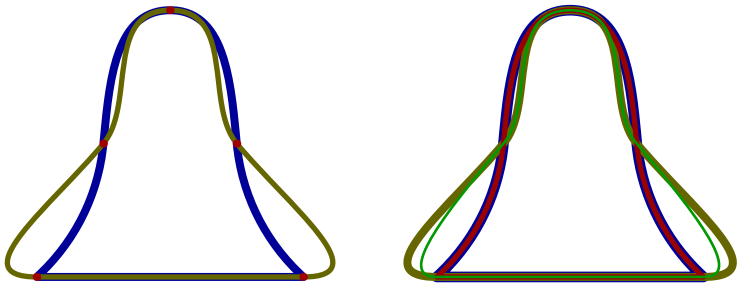


19.9 Connecting

In LuaMetaTEX the `--` operator is a primitive, like `..` and when exploring this we came up with this example that demonstrates the difference with (still a macro) `---`.

```
\startMPcode
path p[] ;
p[1] = origin --  (100, 0) ..  (75, 50) ..  (50, 100) ..  (25, 50) ..  cycle ;
p[2] = origin --- (100, 0) ..  (75, 50) ..  (50, 100) ..  (25, 50) ..  cycle ;
p[3] = origin --  (100, 0) ... (75, 50) ... (50, 100) ... (25, 50) ... cycle ;
p[4] = origin --- (100, 0) ... (75, 50) ... (50, 100) ... (25, 50) ... cycle ;

draw p[1] withpen pencircle scaled 3bp withcolor "darkblue" ;
draw p[2] withpen pencircle scaled 2bp withcolor "darkyellow" ;
drawpoints p[1] withpen pencircle scaled 3bp withcolor darkred ;

draw image (
    draw p[1] withpen pencircle scaled 4bp withcolor "darkblue" ;
    draw p[2] withpen pencircle scaled 3bp withcolor "darkyellow" ;
    draw p[3] withpen pencircle scaled 2bp withcolor "darkred" ;
    draw p[4] withpen pencircle scaled 1bp withcolor "darkgreen" ;
) shifted (150,0) ;
\stopMPcode
```

Where `...` makes a more tight curve, `---` has consequences for the way a curve gets connected to a straight line segment.



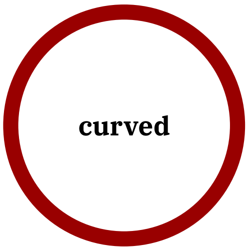
19.10 Curvature

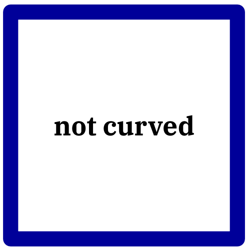
Internally MetaPost only has curves but when a path is output it makes sense to use lines when possible. The ConTEXt backend takes care of that (and further optimizations) but you can check yourself too.

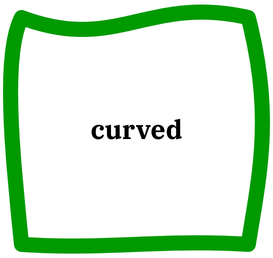
```
\startMPcode
def Test(expr p, c) =
    draw
        p
        withpen pencircle scaled 2mm
        withcolor c ;
    draw
        textext("\bf " & if not (subpath(2,3) of p hascurvature 0.02) : "not"
          else : "" fi & " curved" )
```

```
        shifted center p ;
enddef ;

Test(fullcircle scaled 3cm shifted (0cm,0),"darkred");
Test(fullsquare scaled 3cm shifted (4cm,0),"darkblue");
Test(fullsquare scaled 3cm shifted (8cm,0) randomizedcontrols 1cm,"darkgreen");
\stopMPcode
```

The `hascurvature` macro is a primary and applies a curvature criterium to a (sub)path. The default tolerance in the backend is 131/65536 or 0.002. The same default is used for eliminating points that ‘are the same’.







In the rare case that the backend decides for straight lines while actually there is a curve, you can use `withcurvature 1` to bypass the check.

19.11 Joining paths

Say that you have three paths:

```
path p[] ;
p[1] := (0,0) -- (100,0) ;
p[2] := (101,0) -- (100,100) ;
p[3] := (100,101) ;
```

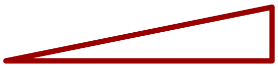
If you join these with:

```
draw p[1] & p[2] & p[3] -- cycle ;
```

You will get an error message telling that the paths don’t have common points so that they can’t be joined. This can be a problem when your snippets are the result of cutting up a path. In practice the difference between the to be joined coordinates is small, so we provide a way to get around this problem:

```
\startMPcode
    interim jointolerance := 5eps ;
    draw (0,0) -- (100,0) & (100+4eps,0) -- (100,20) & (100,20+2eps) -- cycle
        withpen pencircle scaled 2 withcolor "darkred" ;
\stopMPcode
```

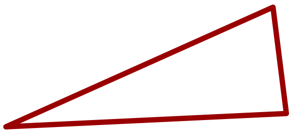
Up to the tolerance is accepted as difference in either direction, so indeed we get a valid result:



Larger values can give a more noticeable side effect:

```
\startMPcode
    interim jointolerance := 20 ;
    draw (0,0) -- (100,0) & (110,10) -- (100,40) & (100,50) -- cycle
        withpen pencircle scaled 2 withcolor "darkred" ;
\stopMPcode
```

It all depends on your need it this is considered okay:



As with everything TEX and MetaPost, once you see what is possible it can be abused:

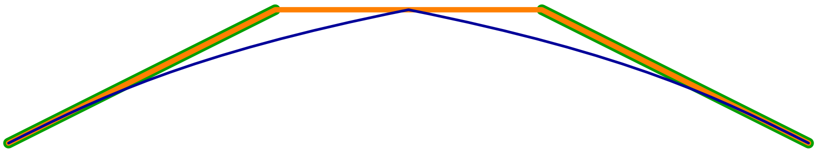
```
\startMPcode
    interim jointolerance := 20 ;
    randomseed := 10 ;
    draw for i=1 upto 200 :
       (i,50 randomized 10) --
    endfor nocycle
        withpen pencircle scaled .1 ;
    randomseed := 10 ;
    draw for i=1 upto 200 :
        (i,50 randomized 10) if odd i : & else : -- fi
    endfor nocycle
        withcolor "darkred" ;
\stopMPcode
```

We leave it up to the reader to decide how the red line can be interpreted.

Here is another nice example:

```
\startMPcode
    path p[] ;
    p[1] := origin -- (100,50) ;
    p[2] := (200,50) -- (300,0) ;
    draw p[1] && p[2] withpen pencircle scaled 4 withcolor darkgreen ;
    draw p[1] -- p[2] withpen pencircle scaled 2 withcolor "orange" ;
    interim jointolerance := 100 ;
    draw p[1] & p[2]  withpen pencircle scaled 1 withcolor "darkblue" ;
\stopMPcode
```

Watch how we get a curve:

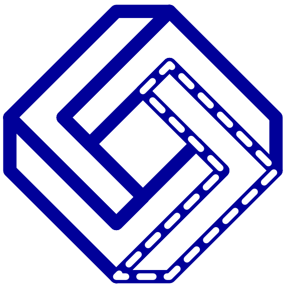


19.12 Dashing

In addition to `dashes` we provide `withdashes` that distributes the dashes along the path in such a way that the pieces are equivalent.

```
\startMPcode
numeric u ; u := 10pt ;

path piece, impossible ;

piece := (0,2u)
    -- xyrelative (  u,  u)
    -- xyrelative ( 4u,-4u)
    -- xyrelative (-4u,-4u)
    -- xyrelative (-2u,  0)
    -- xyrelative ( 4u, 4u)
    -- cycle ;

impossible :=
    piece              &&
    piece rotated  90 &&
    piece rotated 180 &&
    piece rotated 270 ;

draw impossible
    withpen pencircle scaled .5u
    withcolor "darkblue" ;

draw piece
    withdashes .5u
    withpen pencircle scaled .25u
    withcolor white ;
\stopMPcode
```

This turtle graphics example (by Milkael S) also demonstrates appending subpaths to a single path.



20 Envelopes

20.1 Introduction

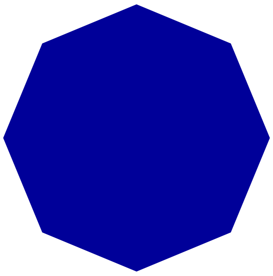
Envelopes are what MetaPost makes for a non circular path. A circular path is supported directly by PostScript and pdf. When such a oath is rotated, it is still somewhat easy because MetaPost outputs the shape twice, transformed differently, but in the end we have one curve, and filling the right space the two curves bound which is native behavior of path filling. When the pen is more complex, that is not a transformed basic pencircle, MetaPost will calculate a so called envelope. This chapter limits the explanation to what we can observe and better explanations about pens can be found in the MetaFont book.

20.2 Pens

The code involves is non trivial and can only work reliable for paths made from straight lines which which is why a pen is always reduced to a path with straight lines. Internally the term ‘convex hull’ is used. In LuaMetaTEX we have that operation as primitive.

```
\startMPcode
pen mypen ; mypen := makepen (fullcircle);
draw origin withpen mypen scaled 100 withcolor "darkblue" ;
\stopMPcode
```

By drawing just one point we see the pen:

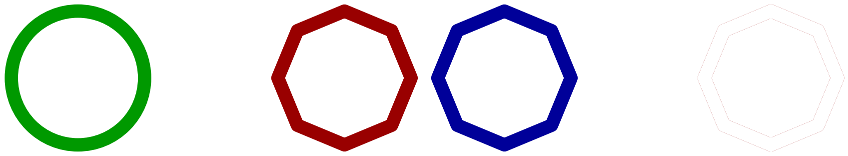


Indeed the circle has been simplified here.

```
\startMPcode
def ShowPaths(expr pth) =
    path p[] ;
    p[0] := pth scaled 50;
    p[1] := uncontrolled p[0] ; % show(p[1]);
    p[2] := convexed     p[0] ; % show(p[2]);
    draw p[0] shifted (  0,0) withpen pencircle scaled 5 withcolor "darkgreen" ;
    draw p[1] shifted (100,0) withpen pencircle scaled 5 withcolor "darkred"   ;
    draw p[2] shifted (160,0) withpen pencircle scaled 5 withcolor "darkblue"  ;
    draw p[1] shifted (260,0) withpen pencircle scaled 5 withcolor "darkred"  ;
    draw p[2] shifted (260,0) withpen pencircle scaled 5 withcolor "white" ;
enddef ;

ShowPaths(fullcircle) ;
```

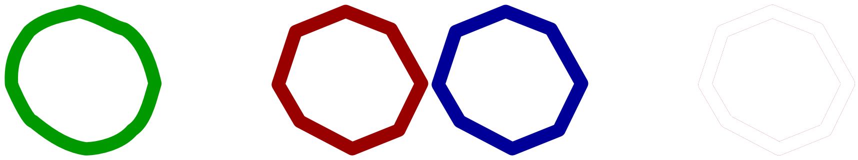
```
\stopMPcode
```

In this case the straightforward removal of control points gives the same result as first calculating the convex hull.



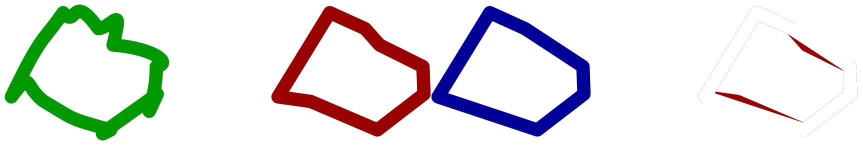
```
\startMPcode
ShowPaths(fullcircle randomized .1) ;
\stopMPcode
```

In this example we still seem to get what we expect:



```
\startMPcode
ShowPaths(fullcircle randomized .4) ;
\stopMPcode
```

But a bit of exaggeration shows that we don't get the same:



It all has to do with heuristics and nasty border cases when we turn corners. Here is what these (not randomized) paths look like, first the `uncontrolled`:

```
(25,0) .. controls (22.56,5.89) and (20.12,11.79)
.. (17,68,17,68) .. controls (11.79,20.12) and (5.89,22.56)
.. (0,25) .. controls (-5.89,22.56) and (-11.79,20.12)
.. (-17,68,17,68) .. controls (-20.12,11.79) and (-22.56,5.89)
.. (-25,0) .. controls (-22.56,-5.89) and (-20.12,-11.79)
.. (-17,68,-17,68) .. controls (-11.79,-20.12) and (-5.89,-22.56)
.. (0,-25) .. controls (5.89,-22.56) and (11.79,-20.12)
.. (17,68,-17,68) .. controls (20.12,-11.79) and (22.56,-5.89)
.. cycle
```

and here is the `unconvexed`:

```
(-25,0) .. controls (-22.56,-5.89) and (-20.12,-11.79)
.. (-17,68,-17,68) .. controls (-11.79,-20.12) and (-5.89,-22.56)
.. (0,-25) .. controls (5.89,-22.56) and (11.79,-20.12)
.. (17,68,-17,68) .. controls (20.12,-11.79) and (22.56,-5.89)
.. (25,0) .. controls (22.56,5.89) and (20.12,11.79)
```

```
.. (17,68,17,68) .. controls (11.79,20.12) and (5.89,22.56)
.. (0,25) .. controls (-5.89,22.56) and (-11.79,20.12)
.. (-17,68,17,68) .. controls (-20.12,11.79) and (-22.56,5.89)
.. cycle
```

Now, in order to see what convexing has to do with pens we also introduce a ‘nep’ which is a pen that doesn’t get its path convexed. We mainly have this variant available for experimenting and documentation purposes. Take these definitions:

```
\startMPdefinitions
path PthP ; PthP := (fullcircle scaled 100) randomized 80 ;
pen  PenP ; PenP := makepen      PthP ;
nep  NepP ; NepP := makenep      PthP ;
path ConP ; ConP := convexed     PthP ;
path UncP ; UncP := uncontrolled PthP ;
\stopMPdefinitions
```

That are used in:

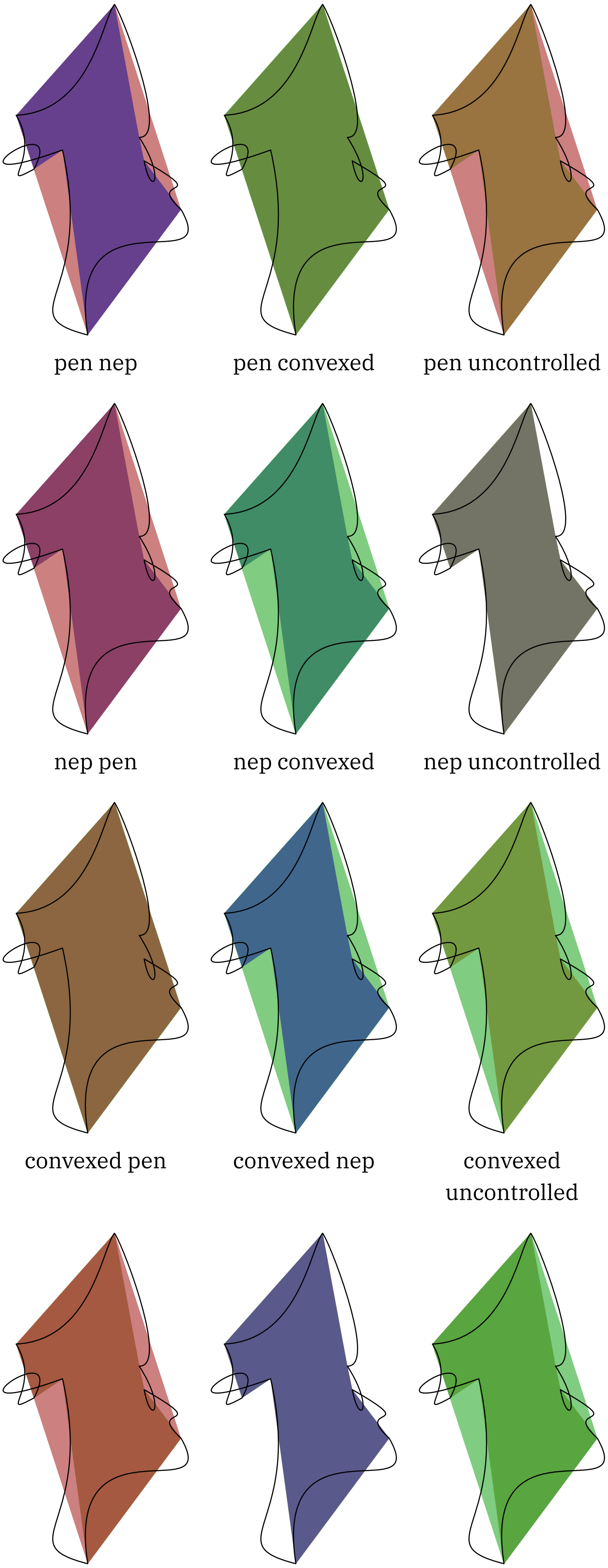
```
\startMPdefinitions
def Pth =
    draw PthP  ;
enddef ;
def Pen =
    draw origin withpen PenP withcolor "darkred" withtransparency (1,.5) ;
enddef ;
def Nep =
    draw origin withpen NepP withcolor "darkblue" withtransparency (1,.5);
enddef ;
def Con =
    fill ConP withpen pencircle scaled 0 withcolor "darkgreen" withtransparency
      (1,.5) ;
enddef ;
def Unc =
    fill UncP withpen pencircle scaled 0 withcolor "darkyellow" withtransparency
      (1,.5) ;
enddef ;
\stopMPdefinitions
```

The main reason for showing the differences in figure 20.1 is that one should be aware of possible side effects

In case you doubt if all this matters, if we use a not to weird path, we’re fine, as is demonstrated in figure 20.2; here we used

```
PthP := fullcircle yscaled 80 xscaled 140 rotated 45 ;
```

And when we use such rather normal (non extreme) paths for pens we’re ready for envelopes.



pen nep

pen convexed

pen uncontrolled

nep pen

nep convexed

nep uncontrolled

convexed pen

convexed nep

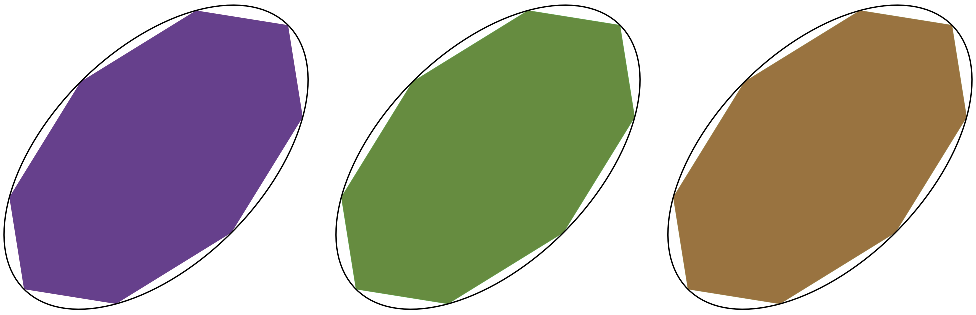
convexed uncontrolled

uncontrolled pen

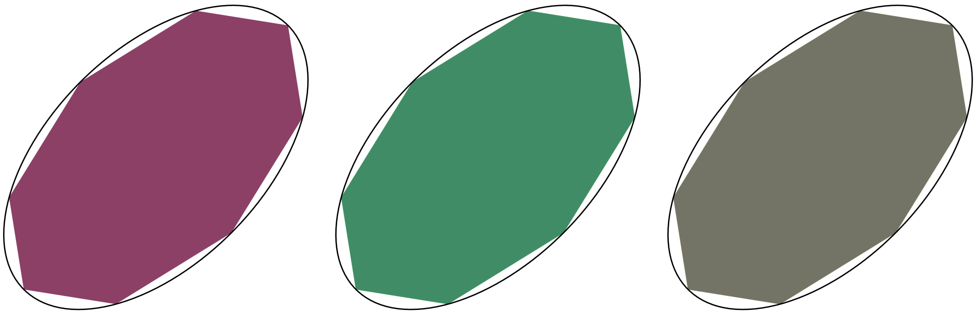
uncontrolled nep

uncontrolled convexed

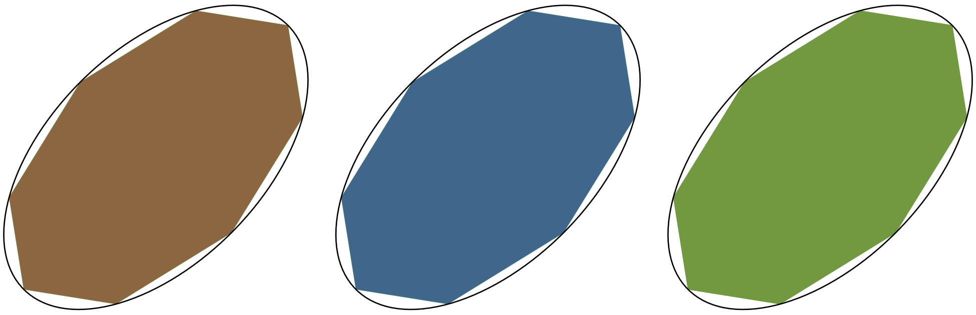
Figure 20.1 Pens are paths with straight lines.



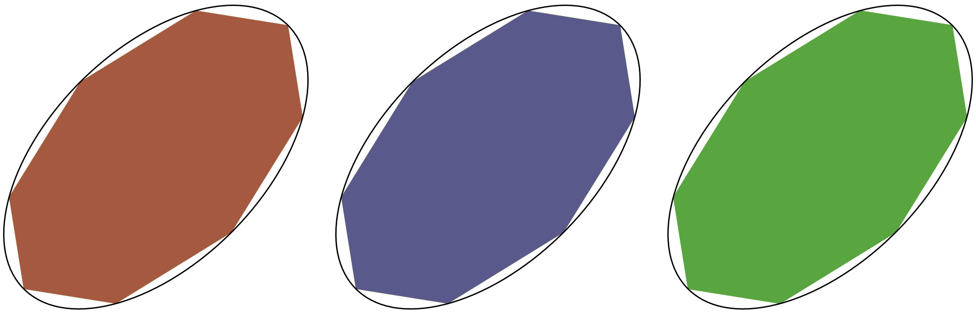
pen nep
pen convexed
pen uncontrolled



nep pen
nep convexed
nep uncontrolled



convexed pen
convexed nep
convexed uncontrolled



uncontrolled pen
uncontrolled nep
uncontrolled convexed

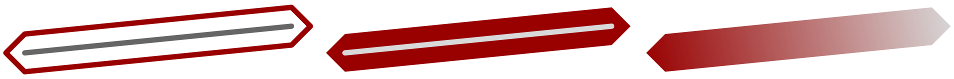
Figure 20.2 When using decent pens the results will be consistent.

20.3 Usage

An envelop is the outline that we get when we run a pen over a path. An envelop is (of course) a closed path. Here is a simple example:

```
\startMPcode
path p ; p := origin -- (100,10) -- cycle ;
path e ; e := envelope pensquare scaled 10 rotated 45 of p ;

draw e withpen pencircle scaled 2 withcolor "darkred" ;
draw p withpen pencircle scaled 2 withcolor "darkgray" ;

fill e shifted (120,0) withcolor "darkred" ;
draw p shifted (120,0) withcolor "lightgray" withpen pencircle scaled 2  ;

fill e shifted (240,0)
    withshademethod "linear"
    withshadecolors ("darkred","lightgray") ;
\stopMPcode
```

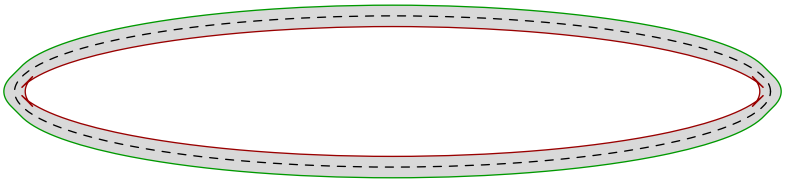
This also demonstrates that this way you can apply a shade to a path:



One problem with envelopes is that you can get unexpected results so let's try to explore some details. We start by defining a main path, a pen, a path from the pen, and two envelopes.

```
\startMPcode
path PthP ; PthP := fullcircle xysized(10cm,2cm) ;
pen  PenP ; PenP := pensquare scaled 2mm rotated 45 ;
path PthU ; PthU := fullsquare scaled 2mm rotated 45 ;
path PatP ; PatP := makepath PenP ;

path PthI ; PthI := envelope PenP of reverse PthP ;
path PthO ; PthO := envelope PenP of PthP ;

fill PthI && PthO && cycle withcolor "lightgray" ;

draw PthI withcolor "darkred" ;
draw PthO withcolor "darkgreen" ;
draw PthP dashed evenly ;
\stopMPcode
```

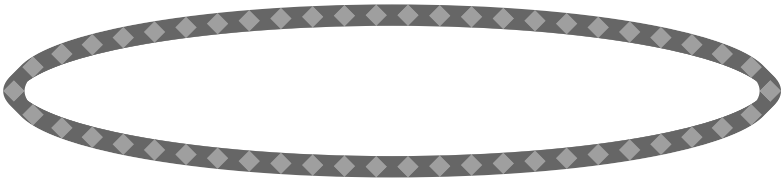
Watch the difference between the two envelopes: one is the result from traveling the pen clockwise and one from running anti-clockwise:



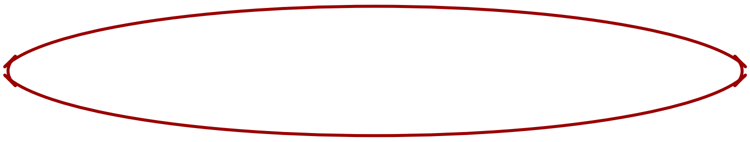
We can emulate running the pen over the path:

```
\startMPcode
fill PthI && PthO && cycle withcolor "darkgray" ;
fill
    for i within (arcpointlist 50 of PthP) :
        PatP shifted pathpoint &&
    endfor cycle
    withcolor "middlegray" ;
\stopMPcode
```

Instead of drawing 50 paths, we draw an efficient single one made from 50 segments and we get this:



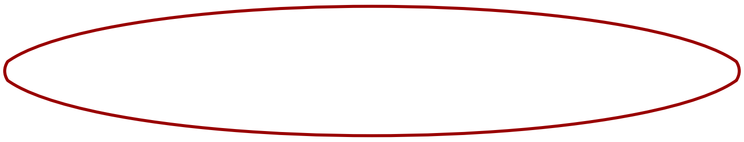
If you look closely at the first rendering you will notice an artifact in the inner envelope.



We can get rid of this with a helper macro:

```
\startMPcode
draw reducedenvelope(PthI) withpen pencircle scaled .4mm withcolor "darkred" ;
\stopMPcode
```

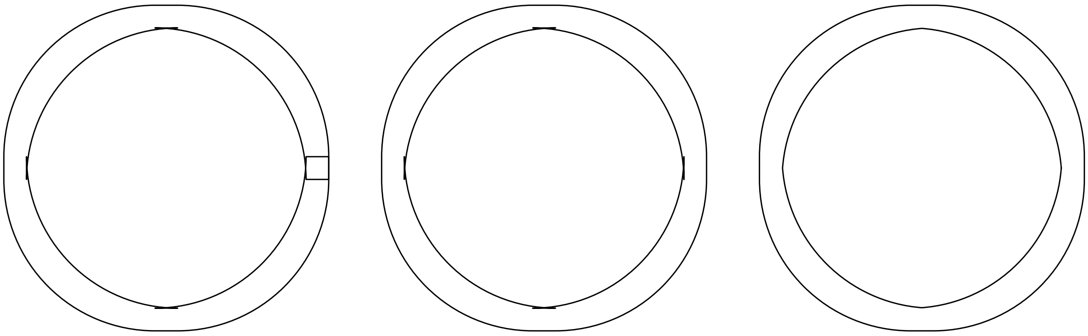
Of course you get no guarantees but here it works:



One reason why the helper is not in the core is that it doesn’t catch all cases:

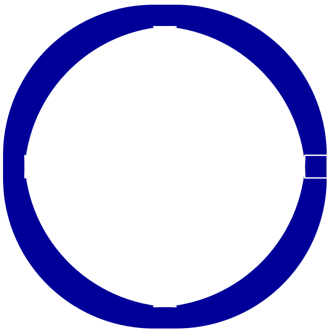
```
\startMPcode
path p ; p := fullcircle scaled 4cm ;
pen  e ; e := pensquare scaled 3mm ;
draw envelope e of p ;
draw envelope e of reverse p ;
p := p rotated eps shifted (5cm,0) ;
draw envelope e of p ;
draw envelope e of reverse p ;
p := p shifted (5cm,0) ;
draw          p  enveloped e ;
draw (reverse p) enveloped e ;
\stopMPcode
```

Watch how a tiny rotations rid us of the weird rectangle, and the helper makes three extra inflected points go away but we're still stuck with an imperfection.



When we only fill the envelope we don't suffer from this'because the artifacts stay within the bounds. Sometimes rotating the pen by eps also helps.

```
\startMPcode
path p ; p := fullcircle scaled 4cm ;
pen  e ; e := pensquare scaled 3mm ;
fill
    (envelope e of p) && (envelope e of reverse p) && cycle
    withcolor "darkblue" ;
draw % just show the artifacts:
    (envelope e of p) && (envelope e of reverse p) && cycle
    withcolor "white" ;
\stopMPcode
```



20.4 Details

For those who are interested in seeing what goes on behind the scenes, this section shows some examples that we made when writing an article about envelopes. We start with a couple of definitions

```
\startMPdefinitions
loadmodule("misc") ;

path mypaths[] ;
path mypens[] ;

mypens[ 1] := fullcircle     scaled 15mm ;
mypens[ 2] := fulldiamond    scaled 15mm ;
```

```
mypens[ 3] := fulltriangle   scaled 15mm ;
mypens[ 4] := fullsquare     scaled 15mm ; % randomized 4mm ;
mypens[ 5] := starring(-1/3) scaled 15mm ;
mypens[ 6] := starring(-1/2) scaled 15mm ;
mypens[ 7] := starring(-eps) scaled 15mm ;
mypens[ 8] := starring(1)    scaled 15mm ;
mypens[ 9] := starring(1/2)  scaled 15mm ;
mypens[10] := starring(eps)  scaled 15mm ;

mypaths[1] := fullcircle                                scaled 10cm ;
mypaths[2] := ((0,0) -- (1/2,1/2) -- (2/2,0))           scaled 10cm ;
mypaths[3] := ((0,0) -- (1/2,1/2) -- (2/2,0) -- cycle) scaled 10cm ;
\stopMPdefinitions
```

We are not going to use all these shapes and pens here but you might want to try out some yourself. We Figure 20.3 we apply a so called pensquare to the paths. In Figure 20.4 we use a star but MetaPost will turn this one into a rectangle. In Figure 20.5 we also use star but here the points are used.

```
\startMPcode
draw showenvelope(mypaths[1], mypens[4]) ;
draw showenvelope(mypaths[2], mypens[4]) shifted (10cm, 1cm) ;
draw showenvelope(mypaths[3], mypens[4]) shifted (10cm,-6cm) ;
\stopMPcode
```

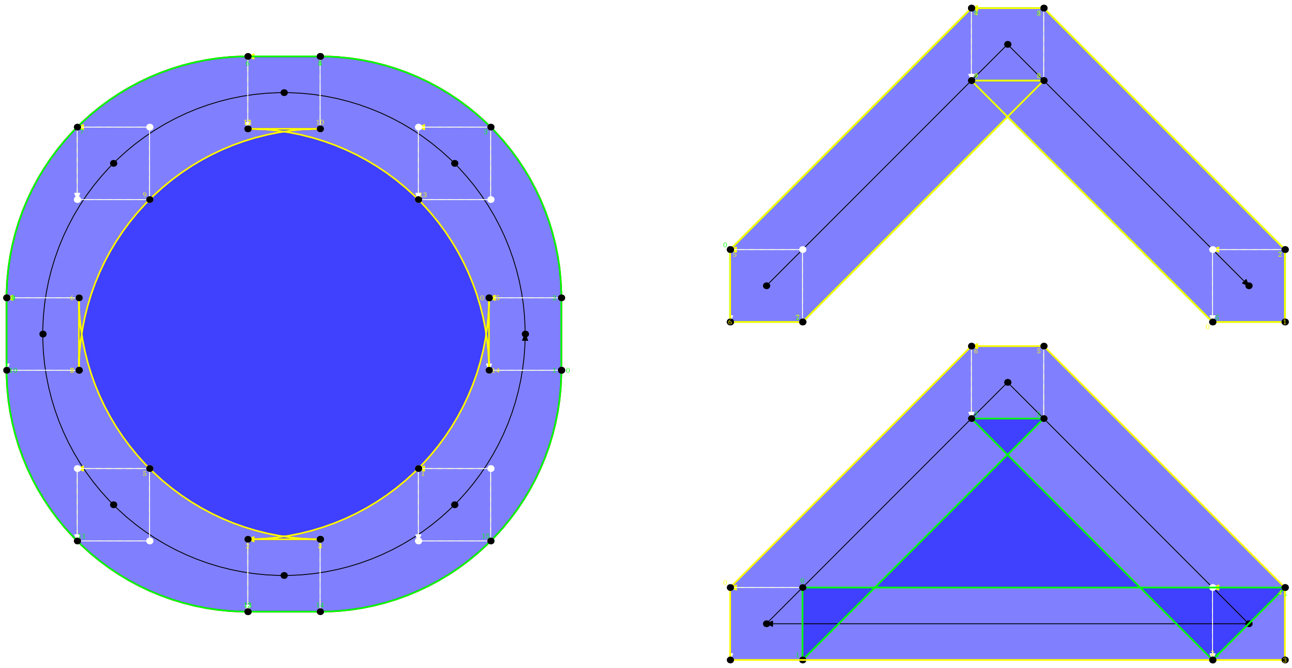


Figure 20.3 How pen 4 creates an envelope.

```
\startMPcode
draw showenvelope(mypaths[1], mypens[6]) ;
draw showenvelope(mypaths[2], mypens[6]) shifted (10cm, 1cm) ;
draw showenvelope(mypaths[3], mypens[6]) shifted (10cm,-6cm) ;
\stopMPcode
\stopMPcode
```

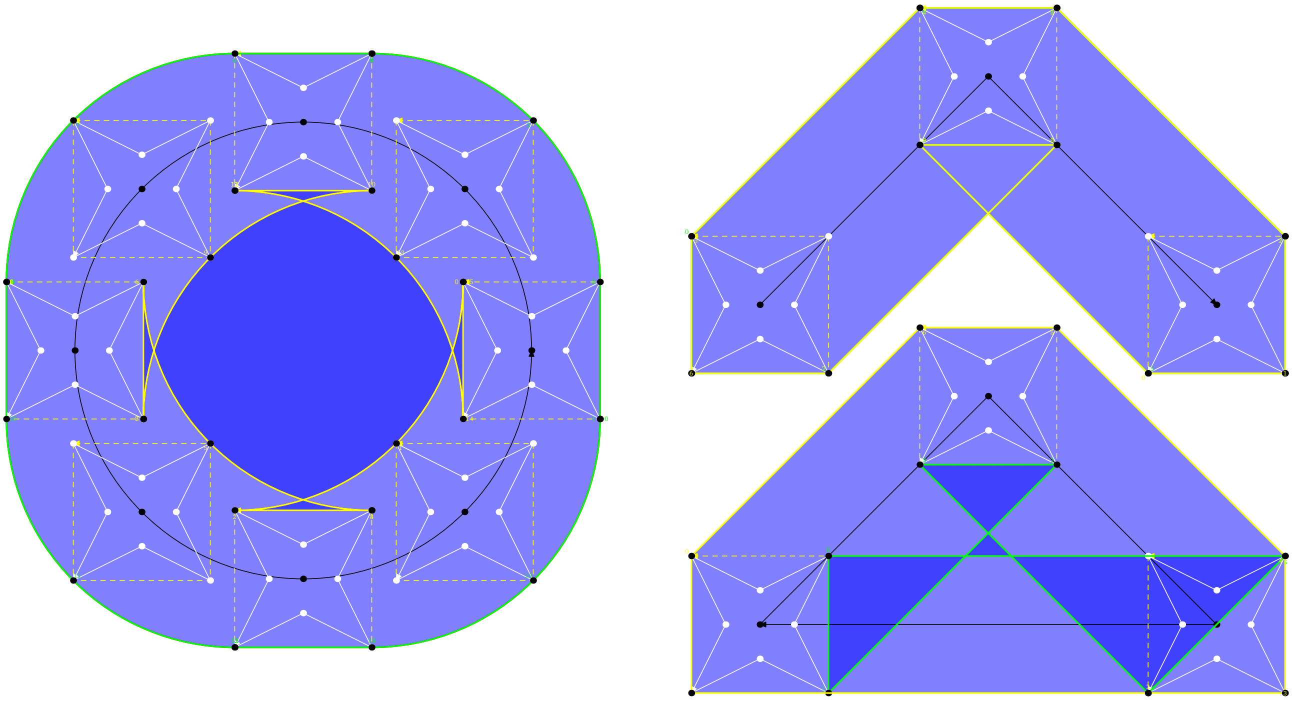


Figure 20.4 How pen 6 creates an envelope.

```
\startMPcode
draw showenvelope(mypaths[1], mypens[9]) ;
draw showenvelope(mypaths[2], mypens[9]) shifted (10cm, 1cm) ;
draw showenvelope(mypaths[3], mypens[9]) shifted (10cm,-6cm) ;
\stopMPcode
```

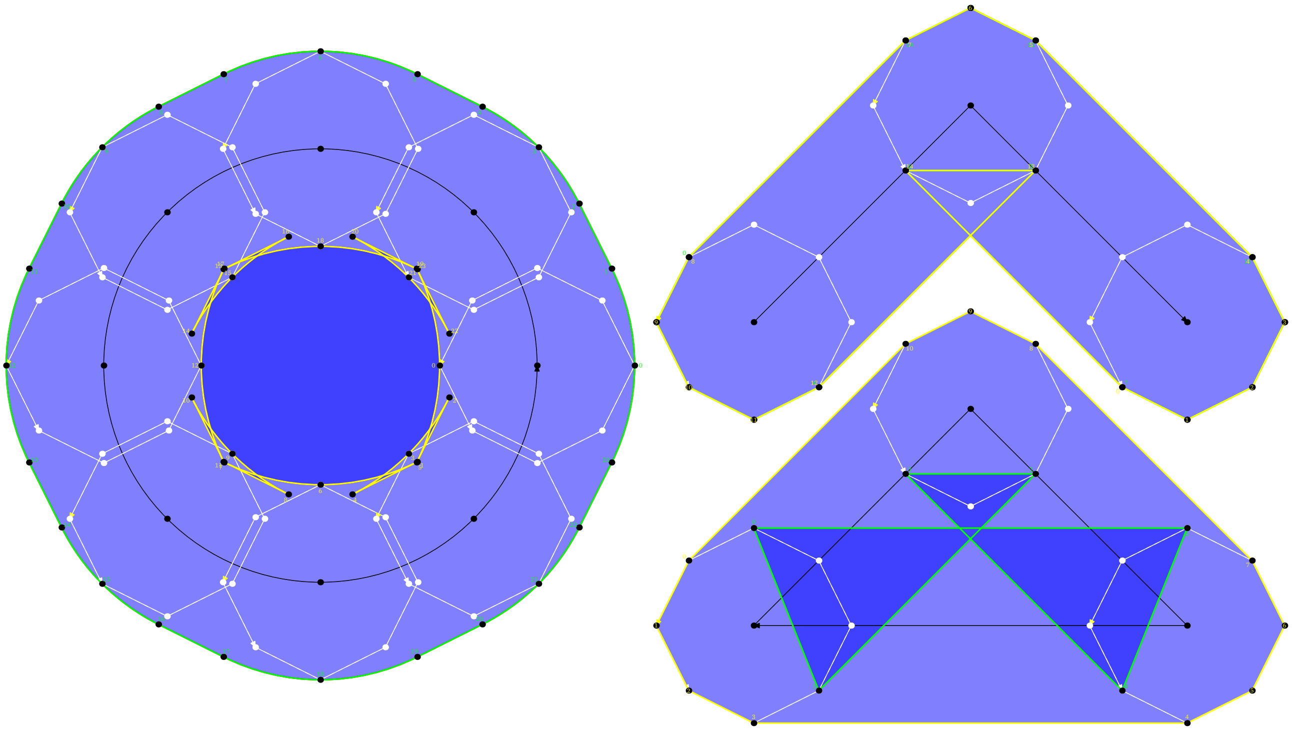


Figure 20.5 How pen 9 creates an envelope.

20.5 Reducing

If you watch the third shape in the previous examples, the last figure differs in that it has a symmetrical inner envelope. We can actually use this knowledge to define a pensquare that is better suited for envelopes. We take this example:

```
\startMPdefinitions
def ExamplePaths =
    path PthA ; PthA := fullcircle scaled 5cm ;
    path PthB ; PthB := triangle   scaled 5cm ;

    draw envelope pensquare scaled 10mm of reverse PthA
        withpen pencircle scaled 2mm
        withcolor "darkblue"
    ;
    draw envelope pensquare scaled 10mm of reverse PthB
        withpen pencircle scaled 2mm
        withcolor "darkblue"
    ;

    draw (reverse PthA) enveloped (pensquare scaled 10mm)
        withpen pencircle scaled 2mm
        withcolor "darkred"
    ;
    draw (reverse PthB) enveloped (pensquare scaled 10mm)
        withpen pencircle scaled 2mm
        withcolor "darkred"
    ;
 enddef ;
\stopMPdefinitions
```

We define two renderings, one with the normal pensquare definition:

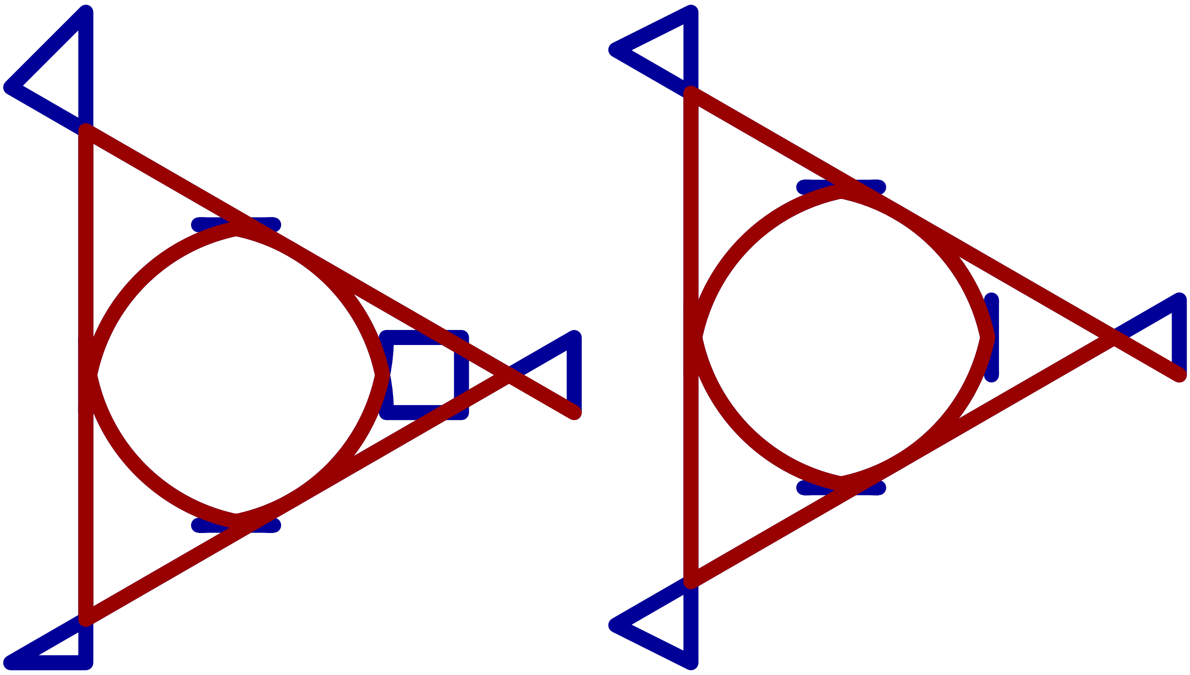
```
\startMPcode
pensquare := makepen(unitsquare shifted -(.5,.5)) ; ExamplePaths ;
\stopMPcode
```

and one with an alternative definition where we have middle points on the edges that stick out one eps:

```
\startMPcode
pensquare := makepen((starring(eps) scaled 1/2)) ; ExamplePaths ;
\stopMPcode
```

This gives figure 20.6. The blue extensions are what we get without clean up but at least the alternative has symmetrical ears.

When you have a somewhat weird envelope the `reducedenvelope` macro might be able to improve it. The `<pth> enveloped <pen>` primary macro has this built in.



default pensquare

alternative pensquare

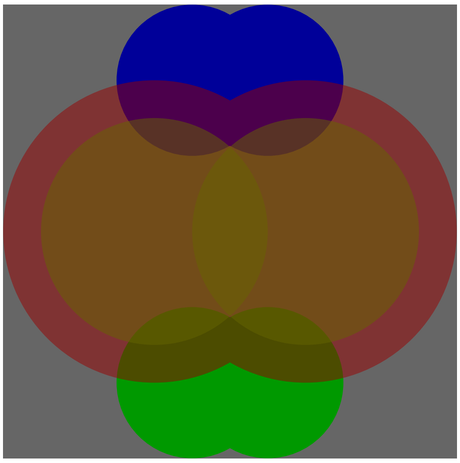
Figure 20.6 An alternative pensquare.

21 Groups

This is just a quick example of an experimental features.

```
\startMPcode
    fill fullcircle scaled 2cm shifted ( 5mm,2cm) withcolor "darkblue" ;
    fill fullcircle scaled 2cm shifted (15mm,2cm) withcolor "darkblue" ;

    fill fullcircle scaled 2cm shifted ( 5mm,-2cm) withcolor "darkgreen" ;
    fill fullcircle scaled 2cm shifted (15mm,-2cm) withcolor "darkgreen" ;

    draw image (
        fill fullcircle scaled 4cm                withcolor "darkred" ;
        fill fullcircle scaled 4cm shifted (2cm,0) withcolor "darkred" ;

        setgroup currentpicture to boundingbox currentpicture
            withtransparency (1,.5) ;
    ) ;

    draw image (
        fill fullcircle scaled 3cm                withcolor "darkyellow"
            withtransparency (1,.5) ;
        fill fullcircle scaled 3cm shifted (2cm,0) withcolor "darkyellow"
            withtransparency (1,.5) ;
    ) ;

    addbackground withcolor "darkgray" ;
\stopMPcode
```

A group create an object that when transparency is applied is treated as a group.

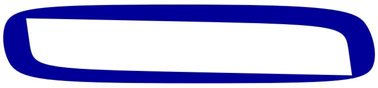


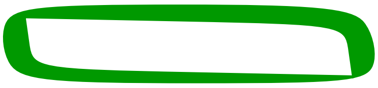
(Groups might become more powerful in the future, like reusable components but then some more juggling is needed.)

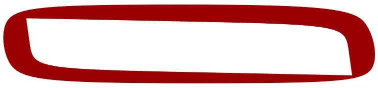
22 Potrace

22.1 Introduction

The potrace connection targets at bitmaps. You can think of logos that only exist as bitmaps while outlines are preferred, but in this case we actually think more of bitmaps that the user lays out. In order to give an impression what we are talking about I give three simple examples:


bytes


buffer


filename

Here we vectorize bitmaps with Peter Selingers potrace library, that we built in LuaMetaTEX. We can directly feed bytes in a MetaFun blob:

```
\startMPcode
    fill
        lmt_potraced [ bytes =
            "01111111111111111111111111111100
             11000000000000000000000000000110
             11000000000000000000000000000011
             11000000000000000000000000000011
             11000000000000000000000000000011
             01100000000000000000000000000011
             00111111111111111111111111111110",
        ] ysized 1cm
        withcolor "darkblue"
        withpen pencircle scaled 1 ;
\stopMPcode
```

But we can also go via a file that has the same data:

```
\startMPcode
    fill
        lmt_potraced [
            filename = "potraced.txt",
        ] ysized 1cm
        withcolor "darkgreen"
        withpen pencircle scaled 1 ;
\stopMPcode
```

Of course we can also use buffers:

```
\startMPcode
    fill
        lmt_potraced [
            buffer = "potraced",
        ] ysized 1cm
```

```
        withcolor "darkred"
        withpen pencircle scaled 1 ;
\stopMPcode
```

You feed a bitmap specification and get back a MetaPost path, likely multiple subpaths sewed together. You can of course draw and fill that path, or store it in a path variable and then do both.

In the following sections we will explore the various options and some tricks. The main message in this section is that you need to look at bitmaps with vectorized eyes because that is what you get in the end: a vector representation.

22.2 Functions

todo

22.3 Icons

When Mikael Sundqvist and I were playing with potrace in MetaFun his girls came up with this pattern.

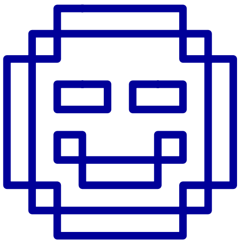
```
\startMPcode
fill
    lmt_potraced [ bytes =
        "001111100
         010000010
         100000001
         101101101
         100000001
         101000101
         100111001
         010000010
         001111100",
         size = 1,
    ] xysized (3cm,3cm)
    withcolor "middleorange" ;
\stopMPcode
```

This produces the following icon. The somewhat asymmetrical shape gives it a charm, and it is surprising how little code is needed. This picture inspired Willi Egger to make a ten by ten composition gadget for the attendants of the 2023 ConTEXt meeting that was used in a tutorial.

We use this to demonstrate a few more features of the interface:

```
\startMPcode
draw
    lmt_potraced [ bytes =
        "..11111..
         .1.....1.
         1.......1
         1.11.11.1
         1.......1
         1.1...1.1
         1..111..1
         .1.....1.
         ..11111..",
         polygon = true,
         size = 1,
    ] xysized (3cm,3cm)
    withcolor "darkblue"
    withpen pencircle scaled 1mm ;
\stopMPcode
```

This contour is actually accurate:

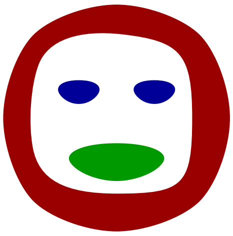


We can color some components:

```
\startMPcode
draw image (
    lmt_startpotraced [ bytes =
        "..11111..
         .1.....1.
         1.......1
         1.22.22.1
         1.......1
         1.3...3.1
         1..333..1
         .1.....1.
         ..11111.."
    ] ;
    fill lmt_potraced [ value = "1", size = 1 ]
        withcolor "darkred" ;
    fill lmt_potraced [ value = "3", size = 1 ]
        withcolor "darkgreen" ;
    fill lmt_potraced [ value = "2", size = 0 ]
        withcolor "darkblue" ;
```

```
    lmt_stoppotraced ;
) xysized (3cm,3cm) ;
\stopMPcode
```

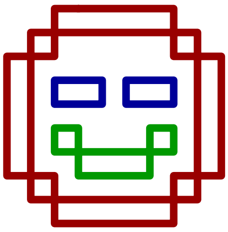
Of course there must be enough distinction (white space) between the shapes:



Again we show the polygons:

```
\startMPcode
draw image (
    lmt_startpotraced [ bytes =
        "..11111..
         .1.....1.
         1.......1
         1.22.22.1
         1.......1
         1.3...3.1
         1..333..1
         .1.....1.
         ..11111.."
    ] ;
    draw lmt_potraced [ value = "1", size = 1, polygon = true ]
        withcolor "darkred" ;
    draw lmt_potraced [ value = "3", size = 1, polygon = true ]
        withcolor "darkgreen" ;
    draw lmt_potraced [ value = "2", size = 0, polygon = true ]
        withcolor "darkblue" ;
    lmt_stoppotraced ;
)
    xysized (3cm,3cm)
    withpen pencircle scaled 1mm ;
\stopMPcode
```

Gives:



We can do the same with data defined in Lua:

```
\startluacode
io.savedata("temp.txt",[[
..11111..
.1.....1.
1.......1
1.22.22.1
1.......1
1.3...3.1
1..333..1
.1.....1.
..11111..
]])
\stopluacode
```

With:

```
\startMPcode
draw image (
    lmt_startpotraced [ filename = "temp.txt" ] ;
        fill lmt_potraced [ value = "1", size = 1 ]
            withcolor "darkcyan" ;
        fill lmt_potraced [ value = "3", size = 1 ]
            withcolor "darkmagenta" ;
        fill lmt_potraced [ value = "2", size = 0 ]
            withcolor "darkyellow" ;
    lmt_stoppotraced ;
) xysized (3cm,3cm) ;
\stopMPcode
```

Indeed we get:

22.4 Fonts

maybe

22.5 Contour type graphics

By combining some Lua with potrace we can do contour graphics. There is currently no interface for this; we give one example. Say that we want to plot the solutions to the equation

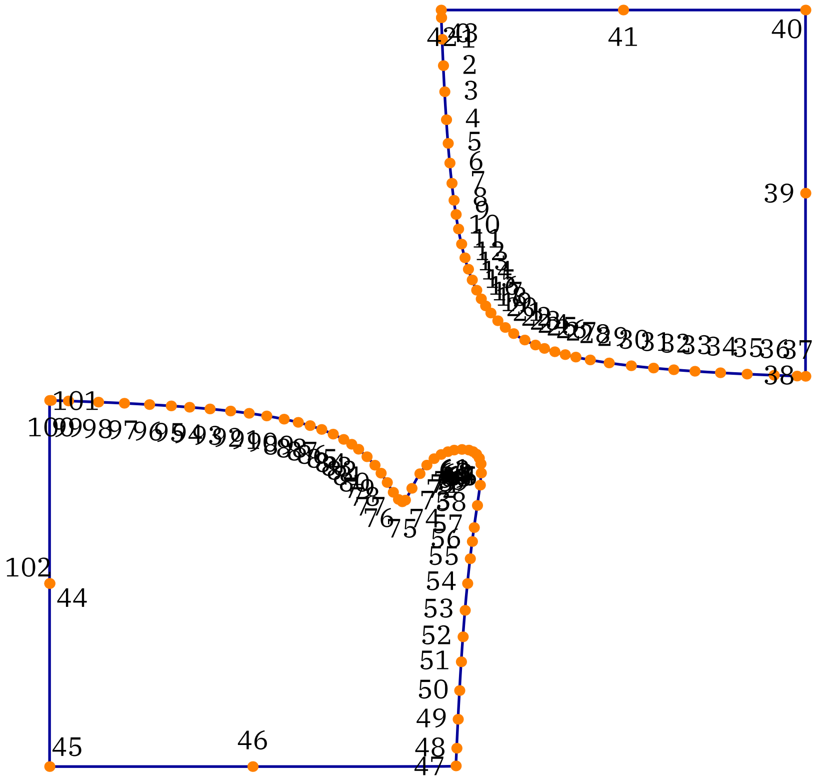
$$xy^2 + 2x^3y^3 - y - 1 = 0$$

for $-5 \le x \le 5$ and $-5 \le y \le 5$. We can then do

```
\startluacode
  local N    = 2000
  local xmin    = -5
  local ymin    = -5
  local xmax    = 5
  local ymax    = 5
  local xstep   = (xmax - xmin)/N
  local ystep   = (ymax - ymin)/N

  local function f(x,y)
    local x = xmin + xstep*x
    local y = ymin + ystep*y
    local z = x*y^2 + 2*x^3*y^3 - y - 1
    if z > 0 then
      return '1'
    else
      return '0'
    end
  end

  potrace.setbitmap("mybitmap", potrace.contourplot(N,N,f))
\stopluacode
```

Then we can make a graphic with

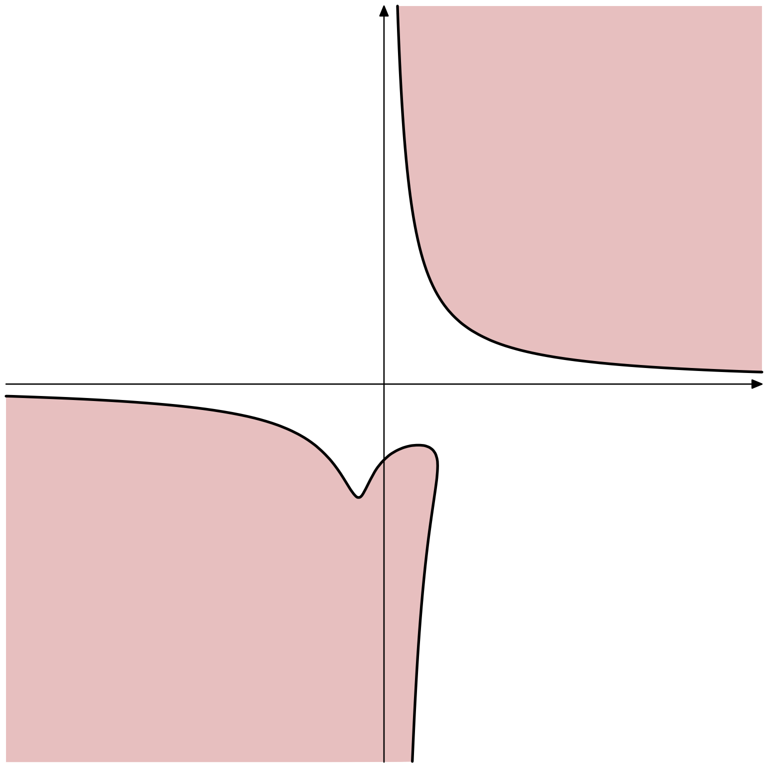
```
\startMPcode
  numeric N ; N := 2000 ;
  path p ; p := lmt_potraced [
    stringname = "mybitmap",
    value      = "1",
    tolerance  = 0.001,
    threshold  = 1,
    optimize   = true,
  ] ;
  p := p shifted (-N/2,-N/2) ;
  message(boundingbox p) ;
  p := p xsized 10cm ;
  draw p withpen pencircle scaled 1 withcolor "darkblue" ;
  drawpoints p withcolor "orange" ;
  drawpointlabels p ;
\stopMPcode
```



By looking at the result we see that we want the subpaths from point 42 to 43 (yes, it is sometimes a bit curious on where potrace starts and stops), 0 to 38, and 47 to 101. Now we can instead do

```
\startMPcode
  numeric N ; N := 2000 ;
  path p ; p := lmt_potraced [
    stringname = "mybitmap",
    value      = "1",
    tolerance  = 0.001,
    threshold  = 1,
    optimize   = true,
  ] ;
  p := p shifted (-N/2,-N/2) ;
  message(boundingbox p) ;
  p := p xsized 10cm ;
  fill p withcolor 0.75[darkred,white] ;
  p := subpath(42,43) of p && subpath(0,38) of p && subpath(47,101) of p  ;
  draw p withpen pencircle scaled 1 ;
  drawarrow (0,-5cm) -- (0,5cm) ;
  drawarrow (-5cm,0) -- (5cm,0) ;
\stopMPcode
```

to get



23 Extensions

23.1 Introduction

This is an uncorrected preliminary chapter.

The TeX and MetaPost macro languages each have their characteristics and as a result the Lua interfaces in both these subsystems are different. There are however some similarities in fetching from, scanning, and pushing back into these subsystems and by using wrappers the nasty details get hidden from users. Wrapping also permits these interfaces to evolve to a stable state.

In due time much will be documented but currently a lot is also a bit experimental because that is the way I can converge to what works best. You can assume that the solutions in the `mlib-*.lmt` files in some form stay (unless it looks too weird). Just stick to the abstractions and you will be fine.

The functionality described here is available in LMTX. Although some prototypes can be found in MkIV you should not expect the same behavior there.

23.2 The LUA interface (strings)

23.2.1 Strings

At some point the `runscript` primitive was added to mplib. Because officially the library is not bound to Lua this neutral name was chosen. In LuaMetaTeX we have a follow up on that library and although it's still neutral we just assume that Lua is used. The MetaFun follow up is therefore called LuaMetaFun, and it used the new interfaces to implement efficient going back and forth between TeX, MetaPost and Lua.

The `runscript` macro is used like this:

```
\startMPcode
draw
    textext("This will print \quotation{Hi} in the console!")
    xsized TextWidth
    withcolor "darkblue" ;
runscript("print('Hi')");
\stopMPcode
```

This will print “Hi” in the console!

The `runscript` primitive triggers a callback that gets the string passed. This callback then does some magic, normally compiling that string into byte code and execute it. The compiled function can return a string that is then fed back into the MetaPost `scantokens` primitive command. So, that return value has to be valid MetaPost!

```
\startMPcode
```

```
string s ;
s := runscript("mp.quoted('This will return a string!')") ;
draw textext(s)
    xsized TextWidth
    withcolor "darkgreen" ;
\stopMPcode
```

This will return a string!

The `mp.quoted` call is one of the build into ConTEXt ways to pipe back something to MetaPost. We will cover this later. If you don't want to use that feature, the call would have looked like this:

```
\startMPcode
string s ;
s := runscript("return " &
    "'" & ditto &
    "Ditto is a string that contains a double qoute!"
    & ditto & "'"
) ;
draw textext(s)
    xsized TextWidth
    withcolor "darkred" ;
\stopMPcode
```

Ditto is a string that contains a double qoute!

The `ditto` with ampersands trickery constructs a string with embedded quotes which is needed because you want to pass back a string and MetaPost only considers something a string when it sees double quotes.

23.2.2 Numerics

Instead of a string you can also pass a numeric:

```
\startMPcode
runscript 10000;
\stopMPcode
```

This time, on the console you will see something:

```
metapost > lua > 1: bad index: 10000
metapost > lua > 1: no result, invalid code: 10000
```

This I because at the Lua end this number should result in some action, in the case of ConTEXt calling a registered function. Because the given number is unknown nothing is done. These messages come from ConTEXt, and MetaPost will keep silent because we don't pass anything back.

This numeric interface only makes sense when the callback handles it and the way ConTEXt does that is probably unique to that macro package. You can of course create MetaPost instances yourself (in Lua) and handle callbacks your own way: you get a string, do this, you get a number, do that.

23.2.3 Helpers

In order to help users passing data to the Lua end there are some helper macros defined using the `lua` macro with suffixes:

```
draw lua.mp.foo(0,2,(3,4)) ;
fill lua.MP.foo(0,2,(3,4)) ;
```

The lowercase `mp` namespace is for ConTEXt itself so if you use that for your own extensions, there is no guarantee against future clashes. The uppercase `MP` namespace is for users. In any case you need to be aware of expansion, so `foo` should not expand to something weird (variable names and `vardef` macro names are okay).

At the Lua end these are mapped onto functions, like:

```
function mp.foo(n,m,p)
    -- do something
end
function MP.foo(n,m,p)
    -- do something
end
```

23.3 Printing back

In the previous chapter we saw `mp.quoted` being used to print back a string to MetaPost for processing by `scantokens`. Not all function in the `mp` namespace are meant for usage, so best stick to what is described here.

The most generic print is `mp.print` that takes multiple arguments. A numeric value is flushed as serialized number and a string is passed along (so no quotes are added). A boolean becomes `true` or `false`. A table with six elements is seen as a transform and otherwise passed as pair, color or cmyk color definition. The `print` command takes multiple arguments and the results are concatenated into one string with other prints so far.

Because this mechanism is already available in MkIV we remain compatible which means that the print functions are available in the `mp` namespace but also in the `mp.aux` namespace. In the meantime we moved to the `print` namespace. The main print command does a guess about what it is fed and will inject that as string. Thereby the next are all valid:

```
fill fullcircle scaled runscript("mp.print      ('3cm')") withcolor "darkred" ;
fill fullcircle scaled runscript("mp.print.print('2cm')") withcolor "darkgreen" ;
fill fullcircle scaled runscript("mp.aux.print  ('1cm')") withcolor "darkblue" ;
```

`string`	`string`	passed as it is but with percent, double quote and newline escaped
`boolean`	`boolean`	the true or false primitives

`integer`	`number`	an integer
`number`	`number`	a float
`numeric`	`number`	a float (same as previous)
`points`	`number`	a scaled numeric with `pt` unit
`pair`	`numbers or table`	a pair (x,y) or (x,x)
`pairpoints`	`numbers or table`	idem but with scaled numbers and a `pt` unit
`triplet`	`numbers or table`	a rgb triplet (r,g,b)
`tripletpoints`	`numbers or table`	idem but with scaled numbers and a `pt` unit
`quadruple`	`numbers or table`	a cmyk quadruple (c,m,y,k)
`quadruplepoints`	`numbers or table`	idem but with scaled numbers and a `pt` unit
`color`	`numbers or table`	a numeric, triplet or quadruple
`transform`	`numbers or table`	a six element transform
`print`	`whatever`	the normal semi-intelligent printer
`fprint`	`format, whatever`	the normal semi-intelligent printer using a format
`vprint`	`variable`	the normal semi-intelligent printer with escaped percents, quotes and newlines
`quoted`	`string`	a valid string surrounded by quotes with an optional first format specifier

A more complex printer is `path` that takes upto three arguments. The first argument is a table. Entries have two or six elements where the last two are control points. The second argument indicates the connector: `true` and `nil` indicate `..` while `false` will use `--`. When the last argument is true we have a closed path. Alternatively the table can have a boolean `cycle` field. So these are all valid:

```
local t1 = { {0,0}, {1,0}, {1,1}, {0,1} }
local t2 = { {0,0}, {1,0}, {1,1}, {0,1}, cycle = true }

mp.print.path(t1)
mp.print.path(t1,nil,true)
mp.print.path(t1,true,true)
mp.print.path(t1,false)
mp.print.path(t1,false,true)
mp.print.path(ts,false)
mp.print.path(t1,"...",true)
mp.print.path(t1,"..",true)
mp.print.path(t2,"..")
```

As with the already mentioned simple printers there is a variant that scales: `pathpoints` (an alternative is of course to scale the whole path by `pt`).

The result of what goes into the print functions is collected and flushed to MetaPost at the end of a call. You can directly push something in the buffer with `mp.direct` and condense the (so far) buffered content with `mp.flush`. Normally you will not need such low level handling.

23.4 Direct values

The print functions accumulate and flush at the end. Alternatively you can return a value. In that case the type determines what gets done:

`number`	native quantity
`boolean`	native quantity (*I need to check this!*)
`string`	feeds into `scantokens`
`table`	feeds concatenated into `scantokens`

Instead of return you can also call an injector. The repertoire is similar to the printers: `boolean`, `cmykcolor`, `color`, `integer`, `number`, `numeric`, `pair`, `path`, `quadruplet`, `string`, `transform`, `triplet` and `whatever` (kind of automatic):

```
function MP.MyFunction()
    mp.inject.string("This is just a string.")
end
```

The `whd`, `xy` and `pt` injectors inject triplets, pairs and numeric scaled from TEX scaled points to base points.

23.5 Registering

Quite some of the build in functionality uses a slightly different approach. It roughly works as follows:

```
% reserve an index and set its value:

newscriptindex user_me_foo ; user_me_foo := scriptindex "user_me_foo" ;

% wrap the call into a macro:

def me_foo = runscript user_me_foo enddef ;
```

A macro can of course be more complex, for instance take arguments and push those into the script call:

```
def me_foo(expr a, b) = runscript user_me_foo a b enddef ;
```

But before this is done at the MetaPost end, you need to define the Lua function:

```
local function user_me_foo()
    -- do something useful
end

metapost.registerscript("user_me_foo",user_me_foo)
```

In this case you use the print and inject functions, of course only when you want to push back some result.

Alternatively you can do:

```
metapost.registerdirect("user_me_foo",user_me_foo)
metapost.registertokens("user_me_foo",user_me_foo)
```

A direct script will treat return values as native, so string and tables are like quoted string and interpreted objects (boolean, numeric, tables). The tokens variant will feed the strings and concatenated tables into scantokens.

The script index can be fetched at the Lua end with:

```
local index = metapost.scriptindex(name)
```

23.6 Codes and such

Using the to be discussed scanners assumes that you know some of the internals (or at least concepts) of MetaPost. Taco has written some excellent tutorials on the way MetaPost handles input. Here we just mention what you can run into.

Each primitive, macro or variable falls into a category. The primitives are grouped in a way that permits handling them as category and the following table shows the grouping. Internally the subcategories are called modes. You should treat these numbers as abstractions because they can change over time, depending on how the library evolves. Modes can normally be ignored.

code	mode	name	code category
66	1	#@	macrospecial
53	129	&	ampersand
53	130	&&	ampersand
53	131	&&&	ampersand
53	132	&&&&	ampersand
60	112	*	secondarybinary
47	110	+	plusorminus
49	115	++	tertiarybinary
49	116	+-+	tertiarybinary
80	0	,	comma
47	111	-	plusorminus
51	0	..	pathjoin
59	113	/	slash
79	0	:	colon
78	0	:=	assignment
81	0	;	semicolon
55	123	<	primarybinary
55	124	<=	primarybinary
55	128	<>	primarybinary
56	127	=	equals
55	125	>	primarybinary
55	126	>=	primarybinary
66	2	@	macrospecial
66	3	@#	macrospecial
37	61	ASCII	unary
68	0	[	leftbracket
9	0	\	relax
69	0	]	rightbracket
60	114	^	secondarybinary
22	0	addto	addto
72	2	also	thingstoadd
57	122	and	and
37	96	angle	unary
37	95	arclength	unary

41	167	arcpoint	ofbinary
41	168	arcpointlist	ofbinary
41	166	arctime	ofbinary
64	0	atleast	atleast
27	1	batchmode	mode
35	0	begingroup	begingroup
37	17	blackpart	unary
37	13	bluepart	unary
33	21	boolean	typename
37	109	bounded	unary
41	172	boundingpath	ofbinary
1	0	btex	btex
41	174	bytefound	ofbinary
41	176	bytemapbounds	ofbinary
60	142	bytemapscaled	secondarybinary
41	175	bytepath	ofbinary
41	173	bytevalue	ofbinary
37	87	centerof	unary
37	88	centerofmass	unary
37	62	char	unary
44	22	charcode	internal
44	25	chardp	internal
44	24	charht	internal
44	26	charic	internal
44	23	charwd	internal
23	38	clip	setbounds
19	5	clipbytemap	newbytemap
37	107	clipped	unary
37	46	closefrom	unary
33	29	cmykcolor	typename
33	28	color	typename
37	67	colormodel	unary
72	1	contour	thingstoadd
62	0	controls	controls
37	57	convexed	unary
19	2	copybytemap	newbytemap
37	86	corners	unary
37	79	cosd	unary
65	0	curl	curl
37	14	cyanpart	unary
40	97	cycle	cycle
71	1	dashed	with
37	70	dashpart	unary
20	1	def	macrodef
44	36	defaultzeroangle	internal
31	0	delimiters	delimiters
37	94	deltadirection	unary
37	91	deltapoint	unary

37	93	deltapostcontrol	unary
37	92	deltaprecontrol	unary
41	154	direction	ofbinary
41	150	directiontime	ofbinary
72	0	doublepath	thingstoadd
4	3	else	fiorelse
4	4	elseif	fiorelse
20	0	enddef	macrodef
6	0	endfor	iteration
82	0	endgroup	endgroup
5	1	endinput	input
41	171	envelope	ofbinary
29	2	errhelp	message
29	1	errmessage	message
27	4	errorstopmode	mode
2	0	etex	etex
30	0	everyjob	everyjob
8	0	exitif	exittest
13	0	expandafter	expandafter
61	8	expr	parametertype
36	39	false	nullary
4	2	fi	fiorelse
37	105	filled	unary
62	1	firstcontrol	controls
37	80	floor	unary
6	2	for	iteration
6	1	forever	iteration
6	3	forsuffixes	iteration
37	12	greenpart	unary
37	18	greypart	unary
37	108	grouped	unary
37	60	hex	unary
3	1	if	if
58	0	infont	primarydef
24	0	inner	protection
5	0	input	input
16	0	interim	interim
44	41	intersectionprecision	internal
49	144	intersectiontimes	tertiarybinary
49	145	intersectiontimeslist	tertiarybinary
44	3	jobname	internal
44	42	jointolerance	internal
37	48	known	unary
49	118	knowncrossprod	tertiarybinary
49	119	knowndiv	tertiarybinary
49	117	knowndotprod	tertiarybinary
49	120	knownmod	tertiarybinary
37	75	knownnorm	unary

37	64	length	unary
44	40	lessdigits	internal
17	0	let	let
44	32	linecap	internal
44	31	linejoin	internal
37	82	llcorner	unary
37	83	lrcorner	unary
37	15	magentapart	unary
37	56	makenep	unary
37	54	makepath	unary
37	55	makepen	unary
12	0	maketext	maketext
28	1	maxknotpool	onlyset
37	76	mexp	unary
44	34	miterlimit	internal
37	77	mlog	unary
36	170	mpversion	nullary
33	24	nep	typename
19	3	newbytemap	newbytemap
18	0	newinternal	newinternal
40	98	nocycle	cycle
37	65	nolength	unary
27	2	nonstopmode	mode
36	44	normaldeviate	nullary
37	50	not	unary
36	41	nullpen	nullary
36	40	nullpicture	nullary
44	2	numberprecision	internal
44	1	numbersystem	internal
33	31	numeric	typename
37	59	oct	unary
37	47	odd	unary
73	0	of	of
49	121	or	tertiarybinary
24	1	outer	protection
44	30	overloadmode	internal
33	30	pair	typename
33	25	path	typename
36	158	pathdirection	nullary
36	163	pathfirst	nullary
36	160	pathindex	nullary
36	164	pathlast	nullary
36	161	pathlastindex	nullary
36	162	pathlength	nullary
37	68	pathpart	unary
36	155	pathpoint	nullary
36	157	pathpostcontrol	nullary
36	156	pathprecontrol	nullary

36	159	pathstate	nullary
44	27	pausing	internal
33	23	pen	typename
36	43	pencircle	nullary
41	165	penoffset	ofbinary
37	69	penpart	unary
33	26	picture	typename
41	151	point	ofbinary
41	153	postcontrol	ofbinary
37	72	postscriptpart	unary
41	152	precontrol	ofbinary
37	71	prescriptpart	unary
61	1	primary	parametertype
20	3	primarydef	macrodef
28	0	randomseed	onlyset
37	45	readfrom	unary
36	42	readstring	nullary
37	11	redpart	unary
19	6	reducebytemap	newbytemap
19	8	resetbytemap	newbytemap
19	9	resetbytemaps	newbytemap
44	39	restoreclipcolor	internal
37	52	reverse	unary
33	28	rgbcolor	typename
60	133	rotated	secondarybinary
11	0	runscript	runscript
15	0	save	save
60	135	scaled	secondarybinary
10	0	scantokens	scantokens
27	3	scrollmode	mode
61	2	secondary	parametertype
20	4	secondarydef	macrodef
62	2	secondcontrol	controls
41	149	segment	ofbinary
37	63	segments	unary
23	40	setbounds	setbounds
19	0	setbyte	newbytemap
19	4	setbytemap	newbytemap
19	1	setbytemapoffset	newbytemap
19	7	setbytemapoptions	newbytemap
23	39	setgroup	setbounds
25	1	setproperty	property
60	136	shifted	secondarybinary
21	0	shipout	shipout
26	2	show	show
26	4	showdependencies	show
26	1	showstats	show
44	28	showstopping	internal

26	0	showtoken	show
26	3	showvariable	show
27	5	silentmode	mode
37	78	sind	unary
60	134	slanted	secondarybinary
37	74	sqrt	unary
44	33	stacking	internal
37	73	stackingpart	unary
75	0	step	step
38	0	str	str
33	22	string	typename
37	106	stroked	unary
41	169	subarclength	ofbinary
41	148	subpath	ofbinary
41	147	substring	ofbinary
61	9	suffix	parametertype
63	0	tension	tension
61	3	tertiary	parametertype
20	5	tertiarydef	macrodef
44	29	texscriptmode	internal
61	10	text	parametertype
44	19	time	internal
74	0	to	to
44	6	tracingcapsules	internal
44	8	tracingchoices	internal
44	10	tracingcommands	internal
44	7	tracingdependencies	internal
44	5	tracingequations	internal
44	12	tracingmacros	internal
44	15	tracingonline	internal
44	13	tracingoutput	internal
44	11	tracingrestores	internal
44	9	tracingspecs	internal
44	14	tracingstats	internal
44	4	tracingtitles	internal
33	27	transform	typename
60	137	transformed	secondarybinary
36	38	true	nullary
44	37	truecorners	internal
37	66	turningnumber	unary
37	84	ulcorner	unary
37	58	uncontrolled	unary
37	53	uncycle	unary
37	81	uniformdeviate	unary
37	49	unknown	unary
76	0	until	until
37	85	urcorner	unary
20	2	vardef	macrodef

1	1	verbatimtex	btex
39	0	void	void
44	35	warningcheck	internal
71	17	withbytemap	with
71	11	withcmykcolor	with
71	15	withcurvature	with
71	8	withgreyscale	with
77	0	within	within
71	12	withlinecap	with
71	13	withlinejoin	with
71	16	withmesh	with
71	14	withmiterlimit	with
71	5	withnestedpostscript	with
71	4	withnestedprescript	with
71	18	withnothing	with
71	7	withoutcolor	with
71	0	withpen	with
71	3	withpostscript	with
71	2	withprescript	with
71	10	withrgbcolor	with
71	6	withstacking	with
32	0	write	write
40	102	xabsolute	cycle
37	5	xpart	unary
37	89	xrange	unary
40	99	xrelative	cycle
60	138	xscaled	secondarybinary
37	7	xxpart	unary
40	104	xyabsolute	cycle
37	8	xypart	unary
40	101	xyrelative	cycle
60	141	xyscaled	secondarybinary
40	103	yabsolute	cycle
37	16	yellowpart	unary
37	6	ypart	unary
37	90	yrange	unary
40	100	yrelative	cycle
60	139	yscaled	secondarybinary
37	9	yxpart	unary
37	10	yypart	unary
60	140	zscaled	secondarybinary
50	0	{	leftbrace
70	0	}	rightbrace

Variables are of a certain type. Possible variable types are available in `metapost.types` via numeric and verbose keys: 0: undefined, 1: vacuous, 2: boolean, 3: unknownboolean, 4: string, 5: unknownstring, 6: pen, 7: unknownpen, 8: nep, 9: unknownnep, 10: path, 11: unknownpath, 12: picture, 13:

unknownpicture, 14: transform, 15: color, 16: cmykcolor, 17: pair, 18: numeric, 19: known, 20: dependent, 21: protodependent, 22: independent, 23: tokenlist, 24: structured, 25: unsuffixedmacro, 26: suffixedmacro.

The possible command codes (as seen in the primitive table) are available in `metapost.codes` via numeric and verbose keys: 0: undefined, 1: btex, 2: etex, 3: if, 4: fiorelse, 5: input, 6: iteration, 7: repeatloop, 8: exittest, 9: relax, 10: scantokens, 11: runscript, 12: maketext, 13: expandafter, 14: definedmacro, 15: save, 16: interim, 17: let, 18: newinternal, 19: newbytemap, 20: macrodef, 21: shipout, 22: addto, 23: setbounds, 24: protection, 25: property, 26: show, 27: mode, 28: onlyset, 29: message, 30: everyjob, 31: delimiters, 32: write, 33: typename, 34: leftdelimiter, 35: begingroup, 36: nullary, 37: unary, 38: str, 39: void, 40: cycle, 41: ofbinary, 42: capsule, 43: string, 44: internal, 45: tag, 46: numeric, 47: plusorminus, 48: secondarydef, 49: tertiarybinary, 50: leftbrace, 51: pathjoin, 52: pathconnect, 53: ampersand, 54: tertiarydef, 55: primarybinary, 56: equals, 57: and, 58: primarydef, 59: slash, 60: secondarybinary, 61: parametertype, 62: controls, 63: tension, 64: atleast, 65: curl, 66: macrospecial, 67: rightdelimiter, 68: leftbracket, 69: rightbracket, 70: rightbrace, 71: with, 72: thingstoadd, 73: of, 74: to, 75: step, 76: until, 77: within, 78: assignment, 79: colon, 80: comma, 81: semicolon, 82: endgroup, 83: stop, 84: undefinedcs.

When you scan for input not all of these make sense, often you will stick to dealing with symbols like brackets, braces, equal signs and variables or expressions.

23.7 Scanners

The most low level scanners are `token` and `symbol`. Although we have them in the `mp.scan` namespace they are just library calls. You use them like:

```
if scan.symbol(true) == "[" then -- "]"
    scan.symbol()
else
    ...
end
```

Here we check if the upcoming token is a specific symbol. The `true` will push back the token. A second boolean argument will enforce expansion.

Scanning can be hairy because the engine is set up in a way that mix lookahead, expand, resolve and processing. So, you can run into a numeric constant, but also in a not yet resolved quantity (take = versus `:=`). When writing more complex scanners it helps to print codes and types.

The `scan.token` function returns a command, mode and expression type but in practice you only have to consider the first value. Other scanners are `boolean`, `cmykcolor`, `color`, `expression`, `integer`, `next`, `number`, `numeric`, `pair`, `path`, `pen`, `property`, `string`, `transform`, plus some implemented around these. Keep in mind that scanners are bound to an instance so the functions in the `scan` namespace are actually wrappers around the library calls.

Because some tokens trigger further scanning (e.g. expressions) we also have two dedicated sub tables with scanners: `tokenscanners` and `typescanners` where, when indexed with a token (command) or type you get the appropriate scanner to get a real result. When you look at what is built into ConTEXt you will notice that we often look ahead and then trigger the appropriate scanner. This approach permits to come up with syntaxes that are different than what MetaPost normally does, so for instance brackets and braces can be used to fence parameters and collections, while lists of comma separated numbers can be grabbed that are not part of pairs, triplets, quadruples etc.

23.8 Special helpers

23.8.1 Hashes

This is typically one of the examples that popped up when Alan Braslau and I were exploring the new possibilities. Due to the way MetaPost implements hashes using Lua might turn out to be more efficient. Here are some examples:

```
\startMPcode
%     newhash("foo") ;
    tohash("foo","bar","gnu") ;
    tohash("foo","rab","ung") ;
    fill fullcircle scaled 1cm withcolor "lightgray" ;
    draw textext(fromhash("foo","bar")) ;
    draw textext(fromhash("foo","rab")) rotated 90 ;
    disposehash("foo") ;
\stopMPcode
```

In this example we allocate a hash and afterwards get rid of it. When you don't allocate one it will be automatically allocated. Hashes are persistent, so if you want to be sure you start fresh you'd better create one explicitly. And if you use a large one, you'd better clean up afterwards.[4]

```
\startMPcode
    resethash("foo")
    tohash("foo",1,"gnu") ;
    tohash("foo",2,"ung") ;
    fill fullcircle scaled 1cm withcolor "lightgray" ;
    for i=1 upto 3 :
        if inhash("foo",i) :
            draw textext(fromhash("foo",i))
                rotated ((i-1) * 90) ;
        fi ;
    endfor ;
\stopMPcode
```

Here we check if something is present in a hash. This example also demonstrates that we can use numbers as key. And yes, you can also use boolean keys:

---

[4] In MkXL the newhash macro is no longer needed to get a unique index.

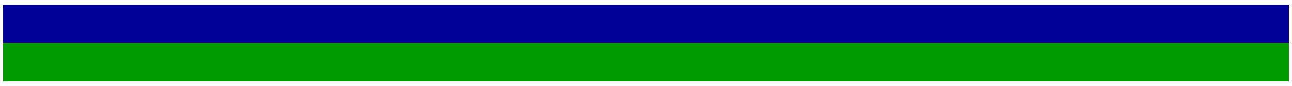
```
\startMPcode
    resethash("foo")
    tohash("foo",false,"gnu") ;
    tohash("foo",true,"ung") ;
    fill fullcircle scaled 1cm withcolor "lightgray" ;
    draw textext(fromhash("foo",false)) ;
    draw textext(fromhash("foo",true)) rotated 90 ;
\stopMPcode
```

Looking at the implementation of these macros (at the MetaPost end) and functions (at the Lua end) will give you an idea how all these interfaces work together.

23.8.2 Modes

You can query the modes set at the TeX end. You can also check the systemmode.

```
\enablemode[weird]
\startMPcode
    fill fullsquare xyscaled (TextWidth,5mm)
        withcolor if texmode("weird") : "darkblue" else : "darkgreen" fi ;
\stopMPcode
\disablemode[weird]
\startMPcode
    fill fullsquare xyscaled (TextWidth,5mm)
        withcolor if texmode("weird") : "darkblue" else : "darkgreen" fi ;
\stopMPcode
```

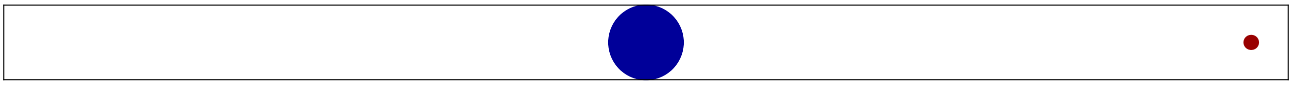


23.8.3 Positions

Keeping track of positions is a core feature and accessible in MetaPosttoo. Here is a somewhat weird example. Positions are always relative to a region, normally the page, but here we provide one via `\framed`.

```
\framed [region=MyRegion,offset=overlay,width=1tw] \bgroup \hpos {here} \bgroup
    \startMPcode
        fill fullcircle scaled 10mm
            withcolor "darkblue" ;
        draw positionxy("here")
            shifted - positionxy("MyRegion")
            withpen pencircle scaled 2mm
            withcolor "darkred" ;
        draw positionxy("here")
            shifted - positionxy("MyRegion")
```

```
            shifted (wdpart positionwhd("MyRegion"),0)
            withpen pencircle scaled 5mm
            withcolor "darkgreen" ;
        % otherwise we oscillate
        setbounds currentpicture to
            fullcircle scaled 10mm ;
    \stopMPcode
\egroup \egroup
```



positionanchor	string
positionbox	path using connector --
positioncolumn	numeric
positioncurve	path using connector ..
positiondepth	numeric
positionhangafter	numeric
positionhangindent	numeric
positionheight	numeric
positionhsize	numeric
positionleftskip	numeric
positionllx	numeric
positionlly	numeric
positionlowerleft	pair
positionlowerright	pair
positionpage	numeric
positionparagraph	numeric
positionparindent	numeric
positionpath	path using connector --
positionpx	numeric
positionpxy	pair
positionpy	numeric
positionregion	string
positionrightskip	numeric
positionupperleft	pair
positionupperright	pair
positionurx	numeric
positionury	numeric
positionwhd	(wd,ht,dp)
positionwidth	numeric

Positioning can be tricky. You really need to make sure that the bounding box of the result is right because when it changes, positions also change you get cyclic runs and quite possible graphics that get larger and larger.

23.8.4 TEX quantities

You can set and get some of TEX's internal quantities:

```
\scratchdimen=100pt \scratchcounter=250 \scratchtoks={okay} \def\Good{good}
\startMPcode
draw textext(getdimen("scratchdimen"))   shifted (0cm,0) withcolor "darkblue"  ;
draw textext(getcount("scratchcounter")) shifted (3cm,0) withcolor "darkred"   ;
draw textext(gettoks ("scratchtoks"))    shifted (6cm,0) withcolor "darkgreen" ;
draw textext(getmacro("Good"))           shifted (9cm,0) withcolor "darkyellow"
  ;
\stopMPcode
```

```
99.62640099626401    250          okay          good
```

Valid getters are `getmacro`, `getdimen`, `getcount` and `gettoks` and their counterparts are `set...` and `setglobal...`. Instead of names you can use numbers for registers, but don't mess up the system ones:

```
\startMPcode
setdimen(2,2*100pt) setcount(2,2*250) settoks(2,"OKAY") setmacro("Good","GOOD")
draw textext(getdimen(2))      shifted (0cm,0) withcolor "darkblue"   ;
draw textext(getcount(2))      shifted (3cm,0) withcolor "darkred"    ;
draw textext(gettoks (2))      shifted (6cm,0) withcolor "darkgreen"  ;
draw textext(getmacro("Good")) shifted (9cm,0) withcolor "darkyellow" ;
\stopMPcode
```

```
199.25199629806195    500          OKAY          GOOD
```

23.8.5 UTF8

Because we use an utf8 engine we also have MetaPost accepting that encoding. The normal string primitives are unchanged and operate on (ascii) bytes but we have some additional helpers (and more might show up if needed). Here is an example:

```
\startMPcode
string s ; s := "ÀÁÂÃÄÅàáâãäå" ;
draw textext(s)             shifted ( 0cm,0) withcolor "darkyellow" ;
draw textext(utfnum("Â"))   shifted ( 3cm,0) withcolor "darkmagenta" ;
draw textext(utflen(s))     shifted ( 6cm,0) withcolor "darkcyan" ;
draw textext(utfsub(s,3,4)) shifted ( 9cm,0) withcolor "darkblue" ;
draw textext(utfsub(s,6))   shifted (12cm,0) withcolor "darkred" ;
\stopMPcode
```

```
ÀÁÂÃÄÅàáâãäå      194          12          ÂÃ          Åàáâãäå
```

23.8.6 Checkers

There are a couple of checkers, mostly used in modules. Here's are a few that Alan needs for the node module:

```
\startMPcode
    draw image (
        draw textext(if isarray p[1][2] : "Y__" else : "N__" fi) ;
```

```
        draw textext(if isarray p[1]    : "_Y_" else : "_N_" fi) ;
        draw textext(if isarray p       : "__Y" else : "__N" fi) ;
    ) xsized 3cm withcolor "darkred" ;
\stopMPcode
```

YYN

```
\startMPcode
    draw image (
        draw textext(prefix p[1][2]) shifted (10,0) withcolor "darkred" ;
        draw textext(prefix p[1]   ) shifted (20,0) withcolor "darkgreen" ;
        draw textext(prefix p      ) shifted (30,0) withcolor "darkblue" ;
    ) ysized 12mm ;
\stopMPcode
```

p p p

```
\startMPcode
    draw image (
        draw textext(dimension p[1][2]) shifted (10,0) withcolor "darkred" ;
        draw textext(dimension p[1]   ) shifted (20,0) withcolor "darkgreen" ;
        draw textext(dimension p      ) shifted (30,0) withcolor "darkblue" ;
    ) ysized 12mm ;
\stopMPcode
```

2 1 0

```
\startMPcode
    picture p ; p := textext("some text") ;
    path    q ; q := fullcircle scaled 3cm ;
    draw textext(tostring(isobject(p)))                  withcolor "darkgreen" ;
    draw textext(tostring(isobject(q))) shifted (50,0) withcolor "darkblue" ;
\stopMPcode
```

```
true    false
```

23.8.7 Key-value interfaces

There are plenty of examples in the `mp-lmtx.mpxl` *file and more will be added. Just make sure you create your own unique namespace and don't use the ones that ConTEXt uses (like* `lmt_`*).*

24 Bytemaps

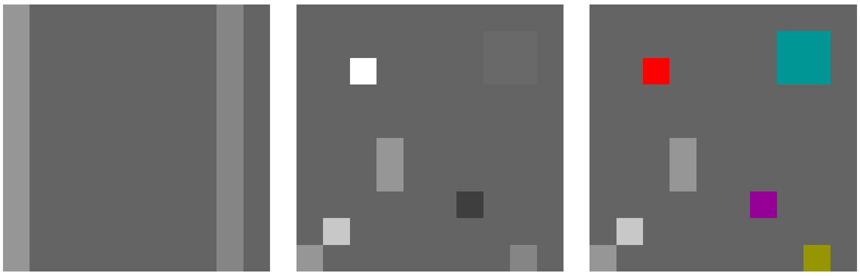
In this chapter we explore bytemaps, which essentially are arrays of bytes that can be used to store states. There are two variants: single bytes and triplets, or in MetaPost speak: numerics or colors. The reason why we added this to the engine is that we expect these to be used in situations where storing states efficiently makes sense. The results can of course be bitmaps but also become a path.

In the following example we create three bytemaps: one with a single row of bytes, one with ten rows and columns black and white, and a same size bytemap with three color components.

```
\startMPcode
newbytemap 1 of  10;
newbytemap 2 of (10,10);
newbytemap 3 of (10,10,3);

for i=1 upto 3 :

    setbytemap i to 100 ;

    setbyte (0,0) of i to 150 ;
    setbyte (1,1) of i to 200 ;

    setbyte (8,0) of i to (150,150,0) ;
    setbyte (6,2) of i to (150,0,150) ;

    setbyte (3,3,1,2) of i to 150 ;
    setbyte (7,7,2,2) of i to (0,150,150) ;

    setbyte (2,7,1) of i to 0 ;
    setbyte (2,7,2) of i to 0 ;
    setbyte (2,7,0) of i to 255 ;

    fill
        unitsquare scaled 100
        shifted ((i-1)*110,0)
        withbytemap i ;

endfor ;
\stopMPcode
```

When the `setbyte` command gets two arguments, they indicate the coordinates. When a third argument is passed, it specifies the (color) channel. When four arguments are passes, the last two indicate the width and height of the area to be set. When a color is assigned to a singly byte pane the color gets reduced to a gray scale.

A bytemap is drawn using a path and `withbytemap` specifier. A `fill` create a tight result, while a `draw` adds half the pen to the dimensions.

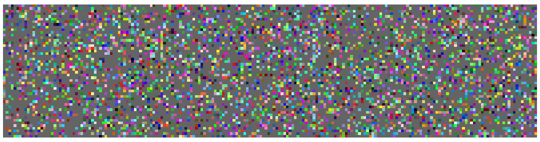


Before we show some more methods, we will do the same as above from the Lua end.

```
\startluacode
local random = math.random
local setbytemap = mp.setbytemap

function MP.MakeByteMap(i)
    mp.newbytemap(i,200,50,3)
    mp.fillbytemap(i,100,100,100)
    for j=1,5000 do
        setbytemap(i,
            random(0,199), random(0,49),
            random(0,255), random(0,255), random(0,255)
        )
    end
end
\stopluacode
```

Next we use this Lua function:

```
\startMPcode
lua.MP.MakeByteMap(4);

fill
    unitsquare xyscaled (200,50)
    withbytemap 4 ;
\stopMPcode
```



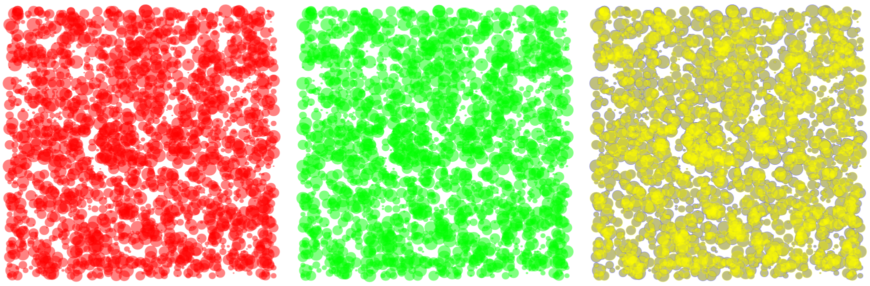
Instead of a bitmap you can also get a path:

```
\startluacode
local random = math.random
local setbytemap = mp.setbytemap

function MP.MakeBytePath(i)
    mp.newbytemap(i,100,100,1)
    mp.fillbytemap(i,100)
    for j=1,2500 do
```

```
        setbytemap(i,
            random(0,99),
            random(0,99),
            random(1,10)
        )
    end
end
\stopluacode
```

This function will fill a gray scale bytemap. We can filter the values and turn the result into a path:

```
\startMPcode
lua.MP.MakeBytePath(5);

picture p[] ;

p[1] := image (
    for i=1 upto 10 :
        if bytefound i of 5 :
            drawdot (bytepath i of 5) shifted (.5,.5)
            withpen pencircle scaled (i/2) ;
        fi ;
    endfor ;
) ;

p[2] := image (
    for i=1 step 2 until 10 :
        if bytefound (i,i+1) of 5 :
            drawdot (bytepath (i,i+1) of 5) shifted (.5,.5)
            withpen pencircle scaled ((i+.5)/2) ;
        fi ;
    endfor ;
) ;

draw p[1]                 withcolor red    withtransparency (1,.5) ;
draw p[2] shifted (110,0) withcolor green  withtransparency (1,.5) ;
draw p[1] shifted (220,0) withcolor blue   withtransparency (1,.5) ;
draw p[2] shifted (220,0) withcolor yellow withtransparency (1,.5) ;
\stopMPcode
```



You can of course fill a bytemap with more meaningful data, as in:

```
\startMPcode
```

```
newbytemap 1 of (100,102) ;
setbyte (0,0,100,102) of 1 to 150 ;

for i=0 upto 99 :
    setbyte(i,99*sin((i/99)*pi),2,4) of 1 to (2.55*i) ;
endfor ;

fill unitsquare xyscaled (400,100) withbytemap 1 ;
\stopMPcode
```

And just show the result as it is. Here we fill small areas:

These bytemaps are not implemented as datatype because it would not only complicate matters but also be inconsistent with for instance equations and scanning. The interface more looks like function calls. Currently we have these, but more (variants) might show up:

```
newbytemap         index of (nx,ny) ;
newbytemap         index of (nx,ny,nz) ;
copybytemap        index to m ;
resetbytemap       index ;
resetbytemaps      ;
setbytemap         index to value;
setbytemap         index to (r,g,b);

setbytemapoption   of index to n ;
setbytemapoffset   of index to (x,y) ;
reducebytemap      of index ;
setbyte            (x,y) of index to value ;
setbyte            (x,y) of index to (r,g,b) ;
setbyte            (x,y,z) of index to value ;
setbyte            (x,y,z) of index to (r,g,b) ;
setbyte            (x,y,dx,dy) of index to value ;
setbyte            (x,y,dx,dy) of index to (r,g,b) ;

withbytemap        index
withbytemask       value
bytevalue          (x,y) of index
bytevalue          (x,y,[z]) of index
bytefound          value of index
bytefound          (min,max) of index
bytepath           value of index
bytepath           (min,max) of index
```

The `withbytemask` directive is not really a primitive but a backend option instead.

Currently the following Lua functions are available:

```
mp.newbytemap     (nx,ny,nz)
mp.setbytemap     (x,y,value)
mp.setbytemap     (x,y,r,g,b)
mp.getbytemap     (x,y)
mp.getbytemap     (x,y,z)
mp.fillbytemap    (x,y,value)
mp.fillbytemap    (x,y,r,g,b)
```

```
mp.getbytemapdata ()     : return nx, ny, nz
mp.getbytemapdata (true) : return nx, ny, nz, data
```

These use the `mplib` library functions that expect an instance as argument but in the ConTEXt mp library that is taken care of automatically.

You can stack a bitmap result on top of another object without it covering that object by using the `withbytemask` directive.

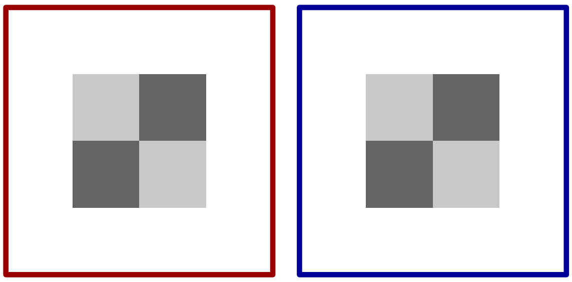
```
\startMPcode
newbytemap 2 of (10,10,3);

setbytemap 2 to (100,0,0) ;

fill unitcircle scaled 100                withcolor blue ;
fill unitsquare scaled 100 shifted (110,0) withbytemap 2 ;
fill unitcircle scaled 100 shifted (220,0) withcolor blue ;
fill unitsquare scaled 100 shifted (220,0) withbytemap 2 withbytemask 100 ;
\stopMPcode
```

You can clip a bytemap to what actually is used.

```
\startMPcode
newbytemap 4 of (4,4) ;
setbytemap 4 to 255 ;

setbyte (1,1) of 4 to 100 ;
setbyte (1,2) of 4 to 200 ;
setbyte (2,1) of 4 to 200 ;
setbyte (2,2) of 4 to 100 ;

pickup pencircle scaled 2;
```

```
fill unitsquare scaled 100 withbytemap 4 ;
draw unitsquare scaled 100 withcolor "darkred" ;

path b ; b := bytemapbounds 255 of 4;

message (llcorner b); % metapost > message 1 1
message (urcorner b); % metapost > message 2 2

copybytemap 4 to 5 ;
clipbytemap 5 to 255 ;

fill unitsquare scaled 100 shifted (110,0) withbytemap 5 ;
draw unitsquare scaled 100 shifted (110,0) withcolor "darkblue" ;
\stopMPcode
```

Here we copy the bytemap because we want to show the original and when we draw (fill) the bytemap we only add a reference so the bitmap that gets embedded is the last one assigned to that slot!

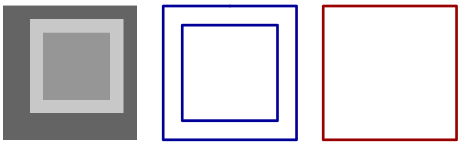


In LuaMetaFun you can use Potrace to turn a bitmap into an outline and bytemaps can be a source too:

```
\startMPcode
newbytemap 1 of (10,10) ;
setbytemap 1 to 100 ;

for i=2 upto 8 :
    for j=2 upto 8 :
        setbyte (i,j) of 1 to 200 ; % 49 "1"
    endfor ;
endfor ;
for i=3 upto 7 :
    for j=3 upto 7 :
        setbyte (i,j) of 1 to 150;
    endfor ;
endfor ;

picture p[] ;

p[1] := image (
    fill unitsquare ysized 50 withbytemap 1 ;
);

p[2] := image (
    draw lmt_potraced [
        explode = true,
```

```
        value   = (char 200), % "1",
        bytemap = 1
    ] ysized 50
    withcolor "darkblue"
    withpen pencircle scaled 1 ;
);

p[3] := image (
    draw lmt_potraced [
        explode = true,
        value   = (char 150),
        bytemap = 1
    ] ysized 50
    withcolor "darkred"
    withpen pencircle scaled 1 ;
);

for i=1 upto 3 :
    draw p[i]
        shifted - center p[i]
        shifted ((i-1) * 60,0)
    ;
endfor ;
\stopMPcode
```

You need to scald these images to your needs; here we just normalize them to the same size. Of course you might need to fine tune the tracing.



You can also load a bitmap from file, for instance:

```
\startMPcode
triplet bm ; bm := loadbytemapfromfile(3, "mill.png") ;
numeric nx ; nx := redpart bm;
numeric ny ; ny := greenpart bm;

newinternal v ;
for col=nx/3 upto 2nx/3 :
    for row=ny/3 upto 2ny/3 :
        v := bytevalue (col,row) of 3 ;
        if (v > 50) and (v < 150) :
            setbyte (col,row) of 3 to 255 ;
        fi
    endfor ;
endfor ;

draw image (
```

```
    fill unitsquare xyscaled (nx,ny)
        withbytemap 3 ;
    draw unitsquare xyscaled (nx/3,ny/3)
        shifted (nx/3,ny/3)
        withpen pencircle scaled 5
        withcolor "darkred" ;
) ysized 8cm ;
\stopMPcode
```

Of course manipulating bitmaps will add to the the runtime but in this case it can be neglected (see figure 24.1).

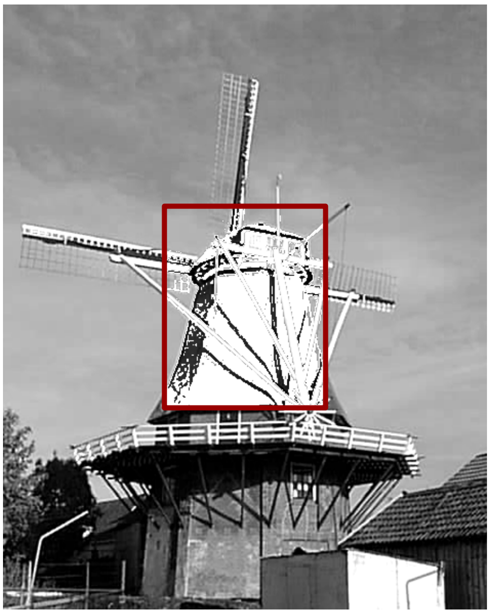
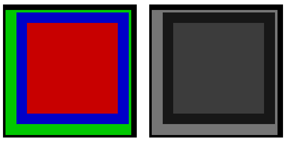


Figure 24.1 A bytemap loaded from file.

Here are a few more examples. We fill a few areas first. Of course this can be done with regular paths but imagine more complex patterns. We make a copy of that bytemap and then convert the pixels to gray scales:

```
\startMPcode
newbytemap 4 of (50,50,3) ;

setbyte (1,1,47,47) of 4 to (0,200,0) ;
setbyte (5,5,42,42) of 4 to (0,0,200) ;
setbyte (9,9,34,34) of 4 to (200,0,0) ;

fill unitsquare
  scaled 50
  withbytemap 4 ;

copybytemap 4 to 8;

reducebytemap 8;

fill unitsquare
```

```
  scaled 50 shifted (55,0)
  withbytemap 8 ;
\stopMPcode
```



This example shows how to use an offset (which can save some calculations) as well as the fact that we can rotate a bitmap (see figure 24.2).

```
\startMPcode
newbytemap 3 of (100,100,3) ;
setbytemap 3 to 50 ;
setbytemapoffset (1,1) of 3;
numeric n ;
for i=1 step 5 until 100 :
    n := 100 ;
    for j=1 step 5 until 100 :
        setbyte (i,j) of 3 to n ;
        n := n + 5 ;
    endfor ;
endfor ;

draw unitsquare
  scaled 200
  rotated 45
  withbytemap 3 ;
\stopMPcode
```

We store bytes which normally represent a small integer value, or in our case an unsigned cardinal (in Modula speak). However, we can also use that byte for storing a small float, in our case a posit. The number of possible value sis of course limited, as can be seen in table 24.1.

In order to get this working, we need to set option 2, as in the following example.[5] Because we use small values, we get a rather dark result but we can expand the bitmap.

```
\startMPcode
    newbytemap 1 of (10,10) ;
    setbytemapoptions 1 to 2 ;
  % setbytemap 1 to 255 ;
  % setbytemap 1 to 1.5 ;
    for i=0 upto 9 :
        for j=0 upto 9 :
            n := uniformdeviate(1) ;
            setbyte (i,j) of 1 to n ;
          % show((n,bytevalue (i,j) of 1)) ;
        endfor ;
```

---

[5] Options are a bitset. Option 1 makes a bytemap persistent across figures, and is handy when you want to access it later on.

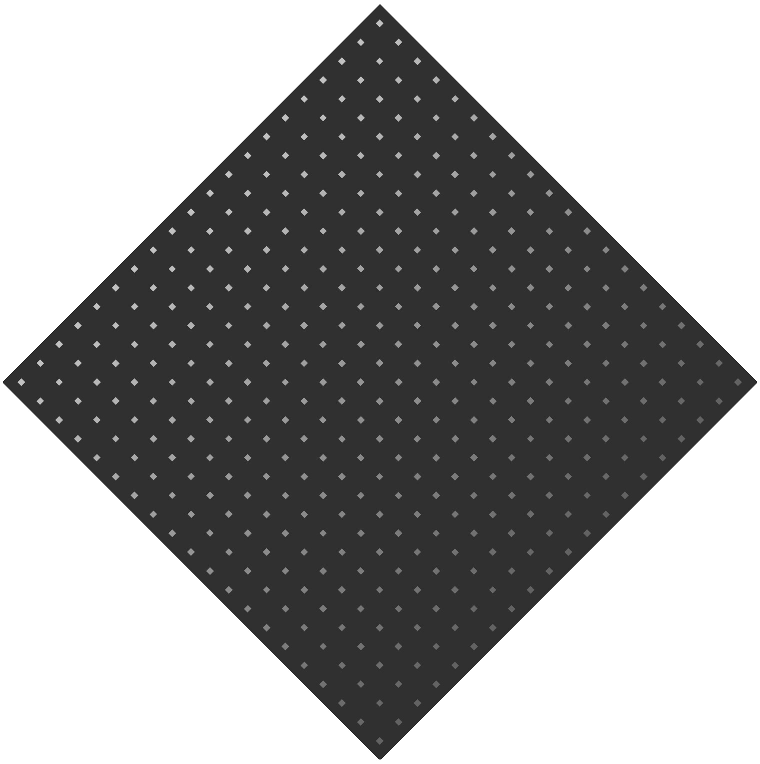
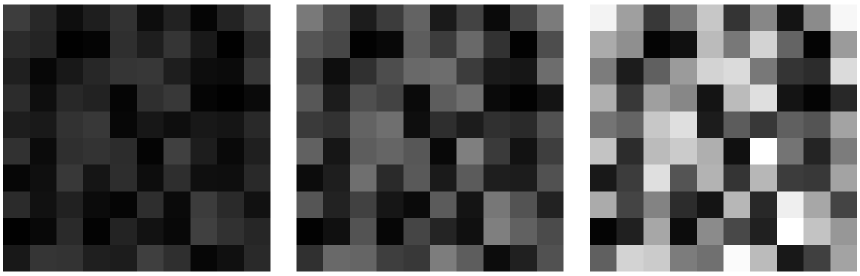


Figure 24.2 Using offset.

```
    endfor ;

    draw image (
        draw unitsquare                  withbytemap 1 ;
        draw unitsquare shifted (1.1,0) withbytemap 1 withbyteexpansion 127 ;
        draw unitsquare shifted (2.2,0) withbytemap 1 withbyteexpansion 255 ;
    ) scaled 100;
\stopMPcode
```

This expansion option is mostly there because of tracing (and showing off in the manual) but we can imagine users finding some application.



The next code produces figure 24.3 which demonstrates that we get back a float instead of an integer in some cases. Posits (for now) only make sense for numeric bytemaps.

```
\startMPcode
```

0.0	00	0.671875	2B	1.6875	56	-64.0	81	-1.625	AC	-0.640625	D7
0.015625	01	0.6875	2C	1.71875	57	-32.0	82	-1.59375	AD	-0.625	D8
0.03125	02	0.703125	2D	1.75	58	-24.0	83	-1.5625	AE	-0.609375	D9
0.046875	03	0.71875	2E	1.78125	59	-16.0	84	-1.53125	AF	-0.59375	DA
0.0625	04	0.734375	2F	1.8125	5A	-14.0	85	-1.5	B0	-0.578125	DB
0.078125	05	0.75	30	1.84375	5B	-12.0	86	-1.46875	B1	-0.5625	DC
0.09375	06	0.765625	31	1.875	5C	-10.0	87	-1.4375	B2	-0.546875	DD
0.109375	07	0.78125	32	1.90625	5D	-8.0	88	-1.40625	B3	-0.53125	DE
0.125	08	0.796875	33	1.9375	5E	-7.5	89	-1.375	B4	-0.515625	DF
0.140625	09	0.8125	34	1.96875	5F	-7.0	8A	-1.34375	B5	-0.5	E0
0.15625	0A	0.828125	35	2.0	60	-6.5	8B	-1.3125	B6	-0.484375	E1
0.171875	0B	0.84375	36	2.125	61	-6.0	8C	-1.28125	B7	-0.46875	E2
0.1875	0C	0.859375	37	2.25	62	-5.5	8D	-1.25	B8	-0.453125	E3
0.203125	0D	0.875	38	2.375	63	-5.0	8E	-1.21875	B9	-0.4375	E4
0.21875	0E	0.890625	39	2.5	64	-4.5	8F	-1.1875	BA	-0.421875	E5
0.234375	0F	0.90625	3A	2.625	65	-4.0	90	-1.15625	BB	-0.40625	E6
0.25	10	0.921875	3B	2.75	66	-3.875	91	-1.125	BC	-0.390625	E7
0.265625	11	0.9375	3C	2.875	67	-3.75	92	-1.09375	BD	-0.375	E8
0.28125	12	0.953125	3D	3.0	68	-3.625	93	-1.0625	BE	-0.359375	E9
0.296875	13	0.96875	3E	3.125	69	-3.5	94	-1.03125	BF	-0.34375	EA
0.3125	14	0.984375	3F	3.25	6A	-3.375	95	-1.0	C0	-0.328125	EB
0.328125	15	1.0	40	3.375	6B	-3.25	96	-0.984375	C1	-0.3125	EC
0.34375	16	1.03125	41	3.5	6C	-3.125	97	-0.96875	C2	-0.296875	ED
0.359375	17	1.0625	42	3.625	6D	-3.0	98	-0.953125	C3	-0.28125	EE
0.375	18	1.09375	43	3.75	6E	-2.875	99	-0.9375	C4	-0.265625	EF
0.390625	19	1.125	44	3.875	6F	-2.75	9A	-0.921875	C5	-0.25	F0
0.40625	1A	1.15625	45	4.0	70	-2.625	9B	-0.90625	C6	-0.234375	F1
0.421875	1B	1.1875	46	4.5	71	-2.5	9C	-0.890625	C7	-0.21875	F2
0.4375	1C	1.21875	47	5.0	72	-2.375	9D	-0.875	C8	-0.203125	F3
0.453125	1D	1.25	48	5.5	73	-2.25	9E	-0.859375	C9	-0.1875	F4
0.46875	1E	1.28125	49	6.0	74	-2.125	9F	-0.84375	CA	-0.171875	F5
0.484375	1F	1.3125	4A	6.5	75	-2.0	A0	-0.828125	CB	-0.15625	F6
0.5	20	1.34375	4B	7.0	76	-1.96875	A1	-0.8125	CC	-0.140625	F7
0.515625	21	1.375	4C	7.5	77	-1.9375	A2	-0.796875	CD	-0.125	F8
0.53125	22	1.40625	4D	8.0	78	-1.90625	A3	-0.78125	CE	-0.109375	F9
0.546875	23	1.4375	4E	10.0	79	-1.875	A4	-0.765625	CF	-0.09375	FA
0.5625	24	1.46875	4F	12.0	7A	-1.84375	A5	-0.75	D0	-0.078125	FB
0.578125	25	1.5	50	14.0	7B	-1.8125	A6	-0.734375	D1	-0.0625	FC
0.59375	26	1.53125	51	16.0	7C	-1.78125	A7	-0.71875	D2	-0.046875	FD
0.609375	27	1.5625	52	24.0	7D	-1.75	A8	-0.703125	D3	-0.03125	FE
0.625	28	1.59375	53	32.0	7E	-1.71875	A9	-0.6875	D4	-0.015625	FF
0.640625	29	1.625	54	64.0	7F	-1.6875	AA	-0.671875	D5		
0.65625	2A	1.65625	55	nan	80	-1.65625	AB	-0.65625	D6		

Table 24.1 Possible float values in a bytemap.

```
newbytemap 1 of (20,5) ;
setbytemapoptions 1 to 2 ;
setbytemap 1 to 0 ;
for i=0 upto 19 :
    for j=0 upto 4 :
        n := 0 randomized 10 ;
        setbyte (i,j) of 1 to n ;
```

```
            % show((n,bytevalue (i,j) of 1)) ;
        endfor ;
    endfor ;

    pickup pencircle scaled 2 ;
    drawdot (bytepath (1,2) of 1) scaled 10 withcolor "darkred" ;
    drawdot (bytepath (2,3) of 1) scaled 10 withcolor "darkgreen" ;
    drawdot (bytepath (3,4) of 1) scaled 10 withcolor "darkblue" ;

    for i=0 upto 19 :
        for j=0 upto 4 :
            n := bytevalue (i,j) of 1;
            draw
                textext (decimal n)
                scaled .2
                shifted (10i,10j-3)
            ;
        endfor ;
    endfor ;
\stopMPcode
```

A nice aspect of posits is that a comparison is very fast because we can directly compare the integer representation. Another advantage over for instance so called minifloats is that they waste less on `NaN` and `Infinity` as well as are most accurate in the smaller values.

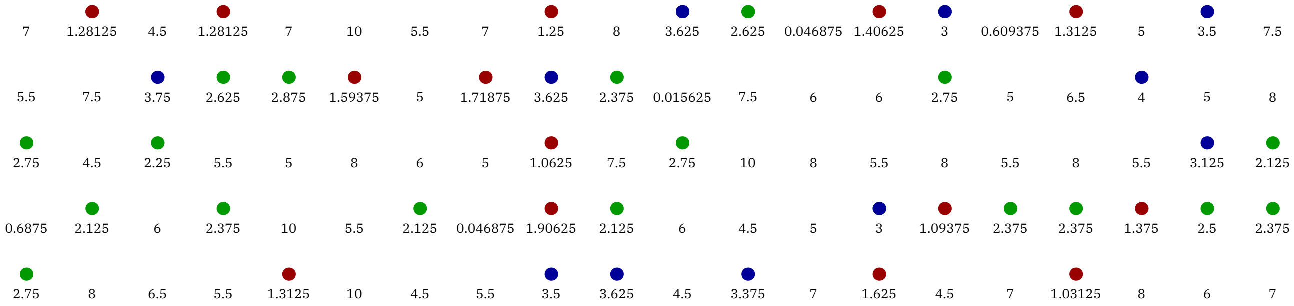


Figure 24.3 Some operations return posits.

Filing a bytemap with values can be done with a MetaPost loop or at the Lua end. A rather convenient variant is using a function and delate the loop to the engine. This is more efficient when we also need access to the currently set values. Here are some examples:

```
\startluacode
    local round  = math.round
    local random = math.random

    function MP.bw_inverse(x,y,s)
        return 255 - s
    end
    function MP.bw_darker(x,y,s)
        return 0.8 * s
    end
    function MP.bw_grain(x,y,s)
        return s - 25 + random(50)
```

```
    end
\stopluacode

\startMPcode
    triplet bm ; bm := loadbytemapfromfile(1, "mill.png") ;
    copybytemap 1 to 2;
    copybytemap 1 to 3;
    copybytemap 1 to 4;
    draw lmt_bytemap [
        bytemap       = 1,
    ] bytemapscaled 1 xsized .25TextWidth ;
    draw lmt_bytemap [
        bytemap       = 2,
        colorfunction = "bw_inverse",
    ] bytemapscaled 1 xsized .25TextWidth xshifted .25TextWidth;
    draw lmt_bytemap [
        bytemap       = 3,
        colorfunction = "bw_darker",
    ] bytemapscaled 1 xsized .25TextWidth xshifted .50TextWidth;
    draw lmt_bytemap [
        bytemap       = 4,
        colorfunction = "bw_grain",
    ] bytemapscaled 1 xsized .25TextWidth xshifted .75TextWidth;
\stopMPcode
```

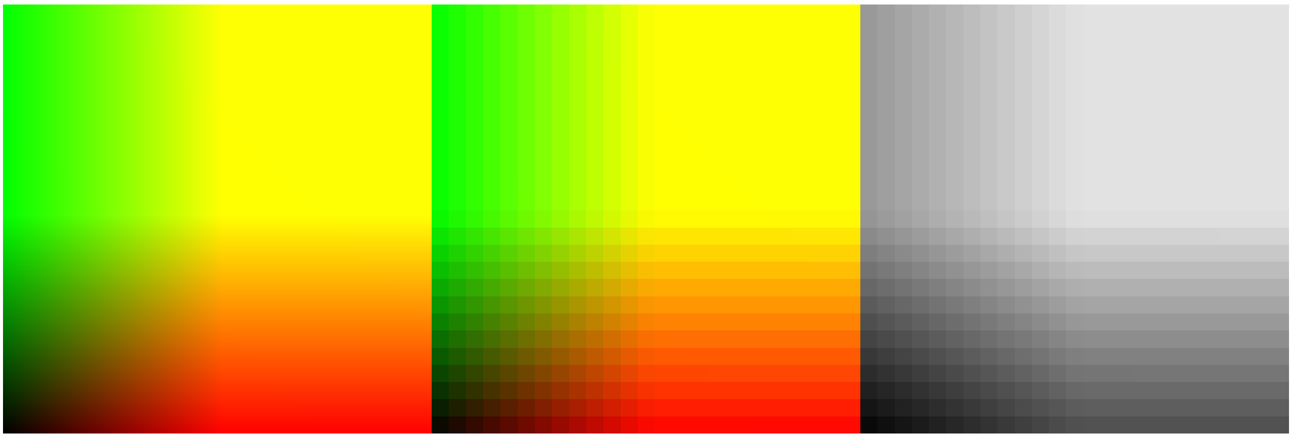
The replacement happens in-place which is why we make some copied to work with.

Here we manipulated the existing bytes. In the next example we don't do that but nevertheless use the function approach:

```
\startluacode
    function MP.bc_test(x,y,b1,b2,b3)
        return x/2, y/2, (x+y)/500
    end
\stopluacode
```

Here we demonstrate the downsample feature. Doing that in pure Lua is actually not that slow but of course this one is faster.

```
\startMPcode
    newbytemap 1 of (1000,1000,3) ;
    newbytemap 2 of (1,1,1);
    picture p ; p := image ( draw lmt_bytemap [
        bytemap       = 1,
        colorfunction = "bc_test",
    ] ; ) ;
    downsamplebytemap(1,2,40);
    copybytemap 2 to 3 ;
    reducebytemap 3;
    draw unitsquare bytemapscaled 1 xsized (TextWidth/3) withbytemap 1 ;
    draw unitsquare bytemapscaled 1 xsized (TextWidth/3) shifted ( TextWidth/3,0
      ) withbytemap 2 ;
    draw unitsquare bytemapscaled 1 xsized (TextWidth/3) shifted (2TextWidth/3,0
      ) withbytemap 3 ;
\stopMPcode
```



Here we apply downsampling to a ‘real’ image:

```
\startMPcode
    triplet bm ; bm := loadbytemapfromfile(1, "mill.png") ;
    newbytemap 2 of (1,1,1);
    downsamplebytemap(1,2,2);
    draw unitsquare
        bytemapscaled 1 xsized .5 TextWidth
        withbytemap 1
    ;
    draw unitsquare
        bytemapscaled 1 xsized .5TextWidth
        shifted (.5TextWidth,0)
        withbytemap 2
    ;
\stopMPcode
```

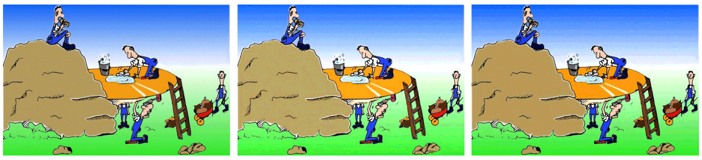
Sampling takes the average of the given cell width, in this case 2. The result in normally quite okay.

Let’s summarize some of these manipulations. Bytemaps are mostly there for special purposes and not so much for manipulating images that one normally would include with `\externalfigure`. One can fill a bytemap manually, from a lossless png file or a lossy jpg file. For runtime usage the bytemaps loaded from file can best not be too large. When the resolution is reasonable performance is quite ok. And if you’re worried about a resolution, just think of the days that a few megapixel images were quite acceptable.

There are various ways to make a large bytemap a bit smaller. For instance, you can reduce the color range in the output file. Here we can either clip or round and in both cases an 8 bit color component becomes a 4 bit one. So, for every six bytes (two pixels) we get three bytes back. In practice, because we now have less distinctive values, compression also works out better.

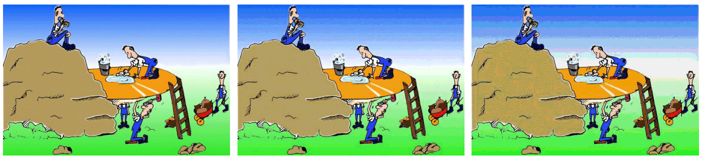
```
\startMPcode
    newbytemap 1 of (1,1,1) ;
    newbytemap 2 of (1,1,1) ;
    newbytemap 3 of (1,1,1) ;
    loadbytemap (1,"hacker.jpg") ;
    copybytemap 1 to 2 ;
    copybytemap 1 to 3 ;
    draw bytemap 1 xsized 3cm xshifted 0.0cm ;
    draw bytemap 2 xsized 3cm xshifted 3.1cm withreduction "round" ;
    draw bytemap 3 xsized 3cm xshifted 6.2cm withreduction "clip" ;
\stopMPcode
```

Reduction gives us:



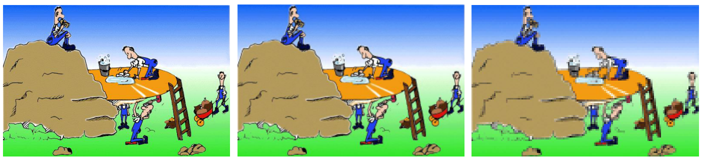
Another way to reduce the size is downgrading. Here we also reudce the number of distinct values, and you can go pretty extreme here:

```
\startMPcode
    newbytemap 1 of (1,1,1) ;
    newbytemap 2 of (1,1,1) ;
    newbytemap 3 of (1,1,1) ;
    loadbytemap (1,"hacker.jpg") ;
    downgradebytemap (1,2,20) ;
    downgradebytemap (1,3,40) ;
    draw bytemap 1 xsized 3cm xshifted 0.0cm ;
    draw bytemap 2 xsized 3cm xshifted 3.1cm ;
    draw bytemap 3 xsized 3cm xshifted 6.2cm ;
\stopMPcode
```



A probably nicer result is possible with downsampling:

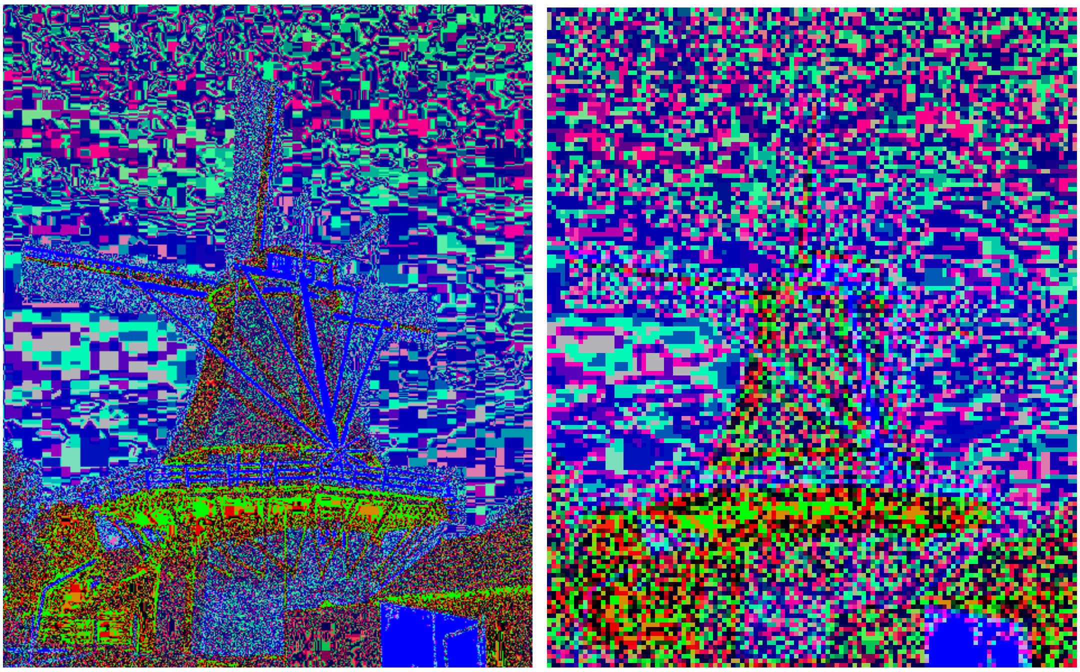
```
\startMPcode
    newbytemap 1 of (1,1,1) ;
    newbytemap 2 of (1,1,1) ;
    newbytemap 3 of (1,1,1) ;
    loadbytemap (1,"hacker.jpg") ;
    downsamplebytemap (1,2,2) ;
    downsamplebytemap (1,3,4) ;
    draw bytemap 1 xsized 3cm xshifted 0.0cm ;
    draw bytemap 2 xsized 3cm xshifted 3.1cm ;
    draw bytemap 3 xsized 3cm xshifted 6.2cm ;
\stopMPcode
```



In the next example we show a combination of effects: downsampling and using a palette:

```
\startluacode
    local sin = math.sin
    local cos = math.cos
    local p = { }
```

```
    for i=0,255 do
        p[i] = { 255*sin(i), 255*cos(i), i }
    end
    figures.palettes.foo = p
\stopluacode

\startMPcode
    newbytemap 1 of (1,1,1) ;
    newbytemap 2 of (1,1,1) ;
    loadbytemap (1,"mill.png") ;
    downsamplebytemap (1,2,4) ;
    draw bytemap 1 xsized 7cm xshifted 0.0cm withpalette "foo" ;
    draw bytemap 2 xsized 7cm xshifted 7.2cm withpalette "foo" ;
\stopMPcode
```



25 Noise

This feature is work in progress and probably will be for a long time because we have to figure out nice parameters to drive this mechanism. An introduction can be found in `beyond-noisy` which is published in the fall TugBoat 2025 by Keith McKay and Hans Hagen. Some of what we tell below comes from that article.

With noise we mean so called Perlin noise. Among the many implementations of Perlin noise those of Stefan Gustavson at Linköping University makes most sense so we started from those. He has has written an excellent introduction on procedural methods that can be used for creating pseudo-random noise see https://github.com/stegu/noiseisbeautiful/tree/main that points to https://liu.diva-portal.org/smash/get/diva2:1954979/FULLTEXT01.pdf.

Although we started our experiments with coding in pure MetaPost eventually we ended up with using the available bytemap mechanism combined with a mix of engine features and Lua. This performs well enough to make it a runtime feature.

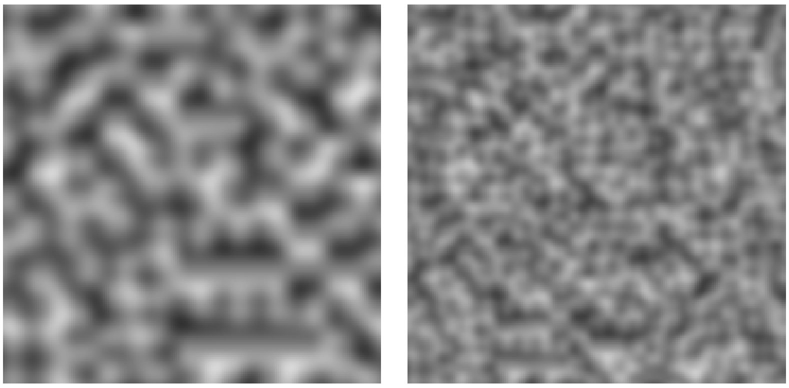
We show a few examples and assume that you read up on the matter, so we won't explain the parameters in detail. Just experiment with them and when you've found interesting patterns, share them on the mailing list; we might add some to a module.

Stefan improved the original Perlin a bit and called it Simplex. In addition he added features to feed back some details and apply angles. The later permit animations: for that you just generate a lot of MetaPost pages and feed those into a graphical editor to produce an animated gif or png.

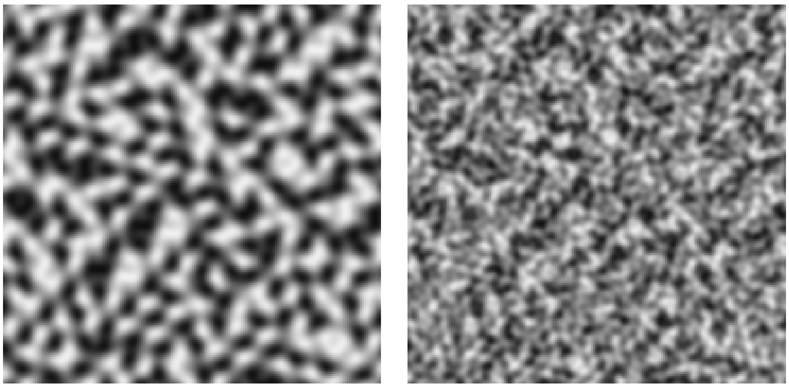
```
\startMPcode
draw lmt_noise [
    bytemap    = 1,
    nx         = 200,
    ny         = 200,
    nz         = 1,
    iterations = 1,
    frequency  = 0.05,
    minimum    = 0,
    maximum    = 255,
    method     = "perlin",
] xsized 5cm ;
\stopMPcode
```

Here we replaced `perlin` by `simplex` in the comparison. In figure 25.1 we compare these for one and two iterations. There are various parameters that we will shortly explain later:

`bytemap`	number	1
`nx`	number	10
`ny`	number	10
`nz`	number	1
`iterations`	number	1
`amplitude`	number	1.0
`frequency`	number	1.0



perlin 1 perlin 2



simplex 1 simplex 2

Figure 25.1 Perlin versus Simplex, one or two iterations.

`persistence`	number	0.5
`lacunarity`	number	2.0
`minimum`	number	0
`maximum`	number	255
`preamble`	string	
`colorcode`	string	
`colorfunction`	string	
`method`	string	perlin
`angle`	number	0
`initialize`	boolean	true
`filename`	string	
`trace`		false

A color function can be more advanced than just returning the normalized noise. Here we register one in the MP namespace. It works with one of the derivatives.

```
\startluacode
    local cos = math.cos
    local r = 256
    function MP.MyFunction(v,x,y,dx,dy)
        -- dx dy only available in last step
        return dy and cos(r-dy) * r
    end
```

```
\stopluacode
```

We will loop a few times to show the difference between iterations as demonstrated in figure 25.2.

```
\startMPcode
    draw lmt_noise [
        bytemap       = 9,
        nx            = 200,
        ny            = 200,
        nz            = 1,
        iterations    = \recurselevel,
        frequency     = .05,
        amplitude     = 1,
        lacunarity    = 2.0,
        minimum       = 0,
        maximum       = 255,
        colorfunction = "MyFunction",
        method        = "detail",
    ] xsized .2TextWidth ;
\stopMPcode
```

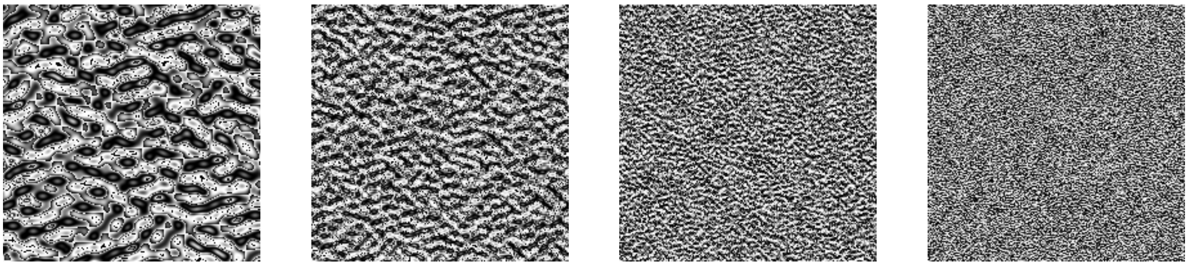


Figure 25.2 Detail noise generation with 2 upto 5 iterations.

The next (third) example is actually a nice one, as it combines several MetaFun features. For this we use a slightly adapted TeX logo because we want overlapping glyphs. A suitable font for this purpose is TeXGyre Bonum.

```
\startluacode
    function MP.MyFunction(v)
        return v/2, v, v/4
    end
\stopluacode
```

An angle is used to calculate a sin and cosine multiplier so we get rotating effects. This example was used to check how well an animation will look as well as performance. Keep in mind that TeX is not really an animation machine, but these graphics can be used in that context. One can of course imagine changing page backgrounds in a document.

```
\startMPcode
draw
    (
        lmt_outline [
            kind = "outline",
```

```
            text = "\strut \bf \MyTeX",
        ]
    ) xsized .25TextWidth
    withpattern image (
        draw lmt_noise [
            bytemap       = 4,
            nx            = 500,
            ny            = 500,
            nz            = 3,
            iterations    = 1,
            frequency     = 0.01,
            amplitude     = 1,
            persistence   = 0.5,
            lacunarity    = 2.0,
            minimum       = 0,
            maximum       = 255,
            colorfunction = "MyFunction",
            method        = "angle",
            angle         = \recurselevel,
          % z             = sqrt(\recurselevel),
          % trace         = true,
        ] xsized .25TextWidth ;
    ) ;
\stopMPcode
```

Figure 25.3 Angled noise generated logos, to be used in a movie.

Controlled noise

The bytemap resolution is determined by `nx` and `ny` and the color depth `nz`. We need to explicitly mention a bytemap number because later on one might want to mess with it. When `filename` is given that file is loaded. It had better be a png image! When `initialize` is false an existing bytemap is used.

Possible `methods` are perlin, simplex, detail and angle. Internally these will be combined with an optional `angle` and `z` parameter to choose the right generator. The result is normalized to the range `minumum` and `maximum`.

The `colorfunction` has been show above and alternatively one can specify one or three return values in `colorcode` in which the `preamble` can define Lua shortcuts. We leave it at this, since there is more in the manuals.

That leaves the parameters that really control the noise. The general term of the wrapping generator is called ‘octave’ because multiple steps determine the result. A single step over `x` and `y` is defined as follows, we use Lua speak:

```
maxamplitude = 0
for i = 1, iterations do
    result       = simplex_noise_2(x * frequency, y * frequency)
    noise        = noise + amplitude * result
    maxamplitude = maxamplitude + amplitude
    amplitude    = amplitude * persistence
    frequency    = frequency * lacunarity
}
noise = noise / maxamplitude
noise = noise * (maximum - minimum) + (maximum + minimum)
noise = noise / 2
if noise > maximum then
    noise = maximum
elseif noise < minimum then
    noise = minimum
end
```

An explanation of these terms and additional terms can be found on the internet and in the literature relating to Perlin noise. They can explain this way better than we can. Here we just want to show what is involved. Believe us, the `simplex_noise_2` and similar functions do quite some computations so it is surprising that we can be as fast as we are!

To give you a taste of future usage, we show what the the example library `\useMPlibrary[noise]` provides: backgrounds.

Here is an example of a noisy frame around some text.

Figure 25.4 A noisy frame.

Figure 25.5 More colorful noise behind a quote (Zelensky).

Figure 25.5 is defined as follows. We use a predefined MetaPost graphic that we hook into the framed command.

```
\startuseMPgraphic{perlinbackground}{min,max}
    PerlinOverlay(1, 50, 255, 50, 3, "1-v, 0, v") ; % updated
\stopuseMPgraphic

\framed
```

```
  [align=normal,
   frame=off,
   offset=4pt,
   foregroundstyle=bold,
   foregroundcolor=white,
   background=perlinbackground,
   rulethickness=4pt]
  {\samplefile{zelensky}}
```

We use the opportunity to point out that we can fill a bytemap with noise but then optionally wipe out part of that bytemap, which is what we do when we just want an outline. As you can see, we have access from the MetaPost end to properties like `OverlayWidth` that relate to the background (an overlay). These are actually `vardef` macros that does an efficient Lua call to get some property.

```
def PerlinOverlay(expr kind, min, max, med, n, code) = % updated
    numeric resolution ; resolution := 5 ;
    numeric variation ; variation := 100 ;
    numeric nx ; nx := resolution * round(OverlayWidth +2OverlayOffset);
    numeric ny ; ny := resolution * round(OverlayHeight+2OverlayOffset);
    numeric lw ; lw := resolution * round(OverlayLineWidth);
    picture p ; p := image ( draw lmt_noise [
        bytemap   = 3,
        nx        = nx,
        ny        = ny,
        nz        = n,
        method    = "detail",
        minimum   = min,
        maximum   = max,
        colorcode = code,
        z         = if variation <> 0 : uniformdeviate fi variation,
    ] ) ;
    if kind == 0 : % frame
        setbyte (lw,lw,nx-2lw,ny-2lw) of 3 to 255 ;
    elseif kind == 2 : % ligher inside
        setbyte (lw,lw,nx-2lw,ny-2lw) of 3 to (med,max) ;
    fi ;
    draw p xysized (OverlayWidth,OverlayHeight) ;
enddef ;
```

26 Interface

26.1 Macros

Because graphic solutions are always kind of personal or domain driven it makes not much sense to cook up very generic solutions. If you have a project where MetaPost can be of help, it also makes sense to spend some time on implementing the basics that you need. In that case you can just copy and tweak what is there. The easiest way to do that is to make a test file and use:

```
\startMPpage
    % your code
\stopMPpage
```

Often you don't need to write macros, and standard drawing commands will do the job, but when you find yourself repeating code, a wapper might make sense. And this is why we have this key/value interface: it's easier to abstract your settings than to pass them as (expression or text) arguments to a macro, especially when there are many.

You can find many examples of the key/value driven user interface in the source files and these are actually not that hard to understand when you know a bit of MetaPost and the additional macros that come with MetaFun. In case you wonder about overhead: the performance of this mechanism is pretty good.

Although the parameter handler runs on top of the Lua interface, you don't need to use Lua unless you find that MetaPost can't do the job. I won't give examples of coding because I think that the source of MetaFun provides enough clues, especially the file `mp-lmtx.mpxl`. As the name suggests this is part of the ConTEXt version LMTX, which runs on top of LuaMetaTEX. I leave it open if I will backport this functionality to LuaTEX and therefore MkIV.

An excellent explanation of this interface can be found at:

`https://adityam.github.io/context-blog/post/new-metafun-interface/`

So (at least for now) here I can stick to just mentioning the currently stable interface macros:

`presetparameters`	`name [...]`	Assign default values to a category of parameters. Sometimes it makes sense not to set a default, because then you can check if a parameter has been set at all.
`applyparameters`	`name macro`	This prepares the parameter handler for the given category and calls the given macro when that is done.
`getparameters`	`name [...]`	The parameters given after the category name are set.
`hasparameter`	`names`	Returns `true` when a parameter is set, and `false` otherwise.
`hasoption`	`names options`	Returns `true` when there is overlap in given options, and `false` otherwise.

`getparameter`	`names`	Resolves the parameter with the given name. because a parameter itself can have a parameter list you can pass additional names to reach the final destination.
`getparameterdefault`	`names`	Resolves the parameter with the given name. because a parameter itself can have a parameter list you can pass additional names to reach the final destination. The last value is used when no parameter is found.
`getparametercount`	`names`	Returns the size if a list (array).
`getmaxparametercount`	`names`	Returns the size if a list (array) but descends into lists to find the largest size of a sublist.
`getparameterpath`	`names string boolean`	Returns the parameter as path. The optional string is one of `--`, `..` or `...` and the also optional boolean will force a closed path.
`getparameterpen`	`names`	Returns the parameter as pen (path).
`getparametertext`	`names boolean`	Returns the parameter as string. The boolean can be used to force prepending a so called `\strut`.
`pushparameters`	`category`	Pushed the given (sub) category onto the stack so that we don’t need to give the category each time.
`popparameters`		Pops the current (sub) category from the stack.

Most commands accept a list of strings separated by one or more spaces, The resolved will then stepwise descend into the parameter tree. This means that a parameter itself can refer to a list. When a value is an array and the last name is a number, the value at the given index will be returned.

```
"category" "name" ... "name"
"category" "name" ... number
```

The `category` is not used when we have pushed a (sub) category which can save you some typing and also is more efficient. Of course than can mean that you need to store values at a higher level when you need them at a deeper level.

There are quite some extra helpers that relate to this mechanism, at the MetaPost end as well as at the Lua end. They aim for instance at efficiently dealing with paths and can be seen at work in the mentioned module.

There is one thing you should notice. While MetaPost has numeric, string, boolean and path variables that can be conveniently be passed to and from Lua, communicating colors is a bit of a hassle. This is because rgb and cmyk colors and gray scales use different types. For this reason it is strongly recommended to use strings that refer to predefined colors instead. This also enforces consistency with the TeX end. As convenience you can define colors at the MetaFun end.

```
\startMPcode
    definecolor [ name = "MyColor", r = .5, g = .25, b = .25 ]

    fill fullsquare xyscaled (TextWidth,5mm) withcolor "MyColor" ;
```

```
\stopMPcode
```

26.2 Units

Many dimensions used at the TEX end are also available in MetaFun. Examples are `TextWidth`, `EmHeight` and `StrutHeight`. In MkIV they are numeric variables that get set every graphic but in MkXL these are not numeric variables but (hidden) Lua calls so they can't be set at the MetaPost end; but they are injected as numeric quantities so you can efficiently them in calculations.

In MetaPost examples you often find u being used as unit, like:

```
u := 1cm ; draw (u,0) -- (u,u) -- (3u,0);
```

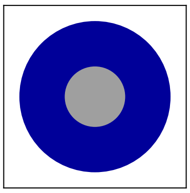
However, what if you want to set such a unit at the TEX end? For this purpose we have a dedicated variable, which is demonstrated in the following examples. First we set a variable:

```
\uunit=1cm

\startMPcode
    definecolor [ name = "MyColor", r = .5, g = .25, b = .25 ]

    fill fullsquare xyscaled (TextWidth,5mm) withcolor "MyColor" ;
\stopMPcode
```

and next we apply it:

```
\framed[offset=.2uu,strut=no]
    \bgroup
        \startMPcode
            fill fullcircle scaled (2uu) withcolor "darkblue" ;
            fill fullcircle scaled (8mm) withcolor "middlegray" ;
        \stopMPcode
    \egroup
```

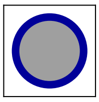
The `\uunit` dimension register is hooked into TEX's unit parser as type uu (user unit). At the MetaPost end uu is effectively a Lua call that fetches the of the dimension from the TEXend and presents it a a numeric.



When we set

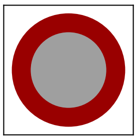
```
\uunit=5mm
```

The same code gives::



```
\framed[offset=.1uu,strut=no]
    \bgroup
        \startMPcode
            save uu ; numeric uu ; uu := 5mm ;
            fill fullcircle scaled (3uu) withcolor "darkred"  ;
            fill fullcircle scaled (2uu) withcolor "middlegray" ;
        \stopMPcode
    \egroup
```

This demonstrates that we can overload uu but make sure to save it first so that later it is available again.



26.3 Paths from LUA

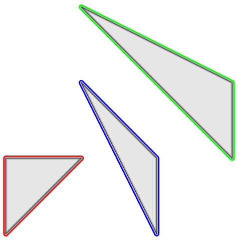
Passing paths to MetaPost using specific properties is sort of tricky because once the points are set, the solver will be applied. This translates curls, tensions and/or explicit control points into the final control points.

In the next example we show a few interfaces. Not all of that might be perfect yes but in most cases it works out.

```
\startluacode
    local shapes = { }
    shapes[1] = { {0,0}, {-1,-1}, {-1, 0}, {0,0}, "cycle" }
    shapes[2] = { {0,1}, { 1, 0}, { 1,-1}, {0,1}, "cycle" }
    shapes[3] = { {0,2}, { 2, 0}, { 2, 1}, {0,2}, "cycle" }
    shapes[4] = {
        {0,0}, {-1,-1}, {-1, 0}, {0,0}, "cycle", "append",
        {0,1}, { 1, 0}, { 1,-1}, {0,1}, "cycle", "append",
        {0,2}, { 2, 0}, { 2, 1}, {0,2}, "cycle", "append",
    }
    shapes[5] = {
        { path = shapes[1], append = true },
        { path = shapes[2], append = true },
        { path = shapes[3], append = true },
    }
    function mp.getshapepath(n)
        mp.inject.path(shapes[n])
    end
\stopluacode

\startMPcode
```

```
    path p ;
    p := lua.mp.getshapepath(1) scaled 1cm ;
    draw p withpen pencircle scaled 2pt withcolor red ;
    p := lua.mp.getshapepath(2) scaled 1cm ;
    draw p withpen pencircle scaled 2pt withcolor blue ;
    p := lua.mp.getshapepath(3) scaled 1cm ;
    draw p withpen pencircle scaled 2pt withcolor green ;
    p := lua.mp.getshapepath(4) scaled 1cm &&cycle ;
    fill p withcolor 0.9 ;
    draw p withpen pencircle scaled 1pt withcolor 0.7 ;
    p := lua.mp.getshapepath(5) scaled 1cm ;
    draw p withpen pencircle scaled .25pt withcolor 0.2 ;
\stopMPcode
```

Especially cycling and appending needs to be done precisely in order not to get redundant (or bad) points.



This combines the first three paths similar to the fourth and fifths. If you doubt what you get you can always show the path and look for {begin} and {end} indicators.

```
\startMPcode
    path p ;
    p := lua.mp.getshapepath(1) scaled 1cm &&
         lua.mp.getshapepath(2) scaled 1cm &&
         lua.mp.getshapepath(3) scaled 1cm ;
    draw p withpen pencircle scaled 1pt withcolor 0.7 ;
  % show(p);
\stopMPcode
```

We draw the result and see that they are decoupled indeed thanks to some && magic: