METAFUN

context mkiv

December 11, 2014

Hans Hagen

Introduction

This document is about METAPOST and TEX. The former is a graphic programming language, the latter a typographic programming language. However, in this document we will not focus on real programming, but more on how we can interface between those two languages. We will do so by using CONTEXT, a macro package written in TEX, in which support for METAPOST is integrated in the core. The TEX macros are integrated in CONTEXT, and the METAPOST macros are bundled in METAFUN.

When Donald Knuth wrote his typographical programming language TEX he was in need for fonts, especially mathematical fonts. So, as a side track, he started writing METAFONT, a graphical language. When you read between the lines in the METAFONT book and the source code, the name John Hobby is mentioned alongside complicated formulas. It will be no surprise then, that, since he was tightly involved in the development of METAFONT, after a few years his METAPOST showed up.

While its ancestor METAFONT was originally targeted at designing fonts, METAPOST is more oriented to drawing graphics as used in scientific publications. Since METAFONT produced bitmap output, some of its operators make use of this fact. METAPOST on the other hand produces POSTSCRIPT code, which means that it has some features not present in METAFONT and vice versa.

With METAFUN I will demonstrate that METAPOST can also be used, or misused, for less technical drawing purposes. We will see that METAPOST can fill in some gaps in TEX, especially its lack of graphic capabilities. We will demonstrate that graphics can make a document more attractive, even if it is processed in a batch processing system like TEX. Most of all, we will see that embedding METAPOST definitions in the TEX source enables a smooth communication between both programs.

The best starting point for using METAPOST is the manual written by its author John Hobby. You can find this manual at every main TEX repository. Also, a copy of the METAFONT book from Donald Knuth is worth every penny, if only because it will give you the feeling that many years of graphical fun lays ahead.

In this METAFUN manual we will demonstrate how you can embed graphics in a TEX document, but we will also introduce most of the features of METAPOST. For this reason you will see a lot of METAPOST code. For sure there are better methods to solve problems, but I have tried to demonstrate different methods and techniques as much as possible.

I started using METAPOST long after I started using TEX, and I never regret it. Although I like TEX very much, I must admit that sometimes using METAPOST is even more fun. Therefore, before we start exploring both in depth, I want to thank their creators, Donald Knuth and John Hobby, for providing me these fabulous tools. Of course I also need to thank Hàn Thế Thành, for giving the TEX community PDFTEX, as well as providing me the hooks I considered necessary for implementing some of the features presented here. In the meantime Taco Hoekwater has created the METAPOST library so that it can be an integral component of LUATEX. After that happened, the program was extended to deal with more than one number implementation: in addition to scaled integers we now can switch to floats and arbitrary precision decimal or binary calculations. I myself prototyped a simple but efficient LUA script interface. With Luigi Scarso who is now the maintainer of METAPOST, we keep improving the system, so who knows what will show up next.

I also want to thank David Arnold and Ton Otten for their fast proofreading, for providing me useful input, and for testing the examples. Without David's patience and help, this document would be far from perfect English and less complete. Without Ton's help, many small typos would have gone unnoticed.

In the second version of this manual the content was been adapted to CONTEXT MKIV that uses LUATEX and the built in METAPOST library. In the meantime some LUA has been brought into the game, not only to help construct graphics, but also as a communication channel. In the process some extra features have been added and some interfacing has been upgraded. This third version of this document deals with that too. It makes no sense to maintain compatibility with CONTEXT MKII, but many examples can be used there as well. In the meantime most CONTEXT users have switch to MKIV, so this is no real issue.

Hans Hagen,

Hasselt NL,
December 11, 2014

Content

Conventions

When reading this manual, you may be tempted to test the examples shown. This can be done in several ways. You can make a file and process that file by METAPOST. Such a file looks like:

```
beginfig(1) ;
  fill fullcircle scaled 5cm withcolor red ; % a graphic
endfig ;

end .
```

Don't forget the semi–colons that end the statements. If the file is saved as `yourfile.mp`, then the file can be processed. Before we process this file, we first need to load some basic METAPOST definitions, because the built in repertoire of commands is rather limited. Such a set is called a format. The standard format is called metapost but we will use a more extensive set of macros `metafun`. In the past such a set was converted into a `mem` file and running the above file was done with:

```
mpost --mem=metafun.mem yourfile
```

However, this is no longer the case and macros need to be loaded at startup as follows:

```
mpost --ini metafun.mpii yourfile.mp
```

Watch the suffix `mpii`: this refers to the stand alone, the one that doesn't rely on LUATEX.

After the run the results are available in `yourfile.1` and can be viewed with GHOSTSCRIPT. You don't need to close the file so reprocessing is very convenient.

Because we will go beyond standard METAPOST, we will use the `mpiv` files. These work with the library which in turn means that we will run from within CONTEXT. This has the advantage that we also have advanced font support at our hands. In that case, a simple file looks like:

```
\starttext
  \startMPpage
    fill fullcircle scaled 5cm withcolor red ;
  \stopMPpage
  \startMPpage
    fill unitsquare scaled 5cm withcolor red ;
  \stopMPpage
\stoptext
```

If the file is saved as `yourfile.tex`, then you can produce a PDF file with:[1]

```
context yourfile
```

The previous call will use LUATEX and CONTEXT MKIV to produce a file with two pages using the built in METAPOST library with METAFUN. When you use this route you will automatically get the integrated text support shown in this manual, including OPENTYPE support. If one page is enough, you can also say:

```
\startMPpage
fill fullcircle scaled 5cm withcolor red ;
\stopMPpage
```

[1] In fact, you could also process the METAPOST file directly because the `context` script will recognize it as such and wrap it into CONTEXT.

So when you have a running CONTEXT on your system you don't need to bother about installing METAPOST and running METAFUN.

We will use lots of color. Don't worry if your red is not our red, or your yellow does not match ours. We've made color definitions to match the overall design of this document, but you should feel free to use any color of choice in the upcoming examples.

By default, CONTEXT has turned its color mechanism on. If you don't want your graphics to have color, you should say:

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

but in todays documents color is so normal that you will probably never do that.

1 Welcome to MetaPost

In this chapter, we will introduce the most important METAPOST concepts as well as demonstrate some drawing primitives and operators. This chapter does not replace the METAFONT book or METAPOST manual, both of which provide a lot of explanations, examples, and (dirty) tricks.

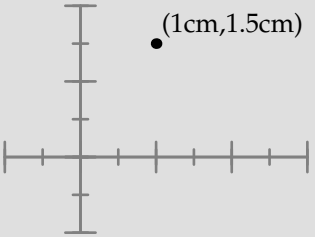
As its title says, the METAFONT book by Donald. E. Knuth is about fonts. Nevertheless, buying a copy is worth the money, because as a METAPOST user you can benefit from the excellent chapters about curves, algebraic expressions, and (linear) equations. The following sections are incomplete in many aspects. More details on how to define your own macros can be found in both the METAFONT book and METAPOST manual, but you will probably only appreciate the nasty details if you have written a few simple figures yourself. This chapter will give you a start.

A whole section is dedicated to the basic extensions to METAPOST as provided by METAFUN. Most of them are meant to make defining graphics like those shown in this document more convenient.

Many of the concepts introduced here will be discussed in more detail in later chapters. So, you may consider this chapter to be an appetizer for the following chapters. If you want to get started quickly, you can safely skip this chapter now.

1.1 Paths

Paths are the building blocks of METAPOST graphics. In its simplest form, a path is a single point.



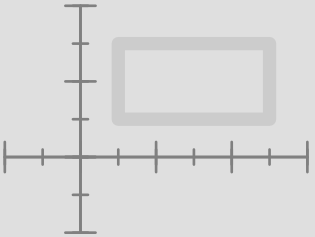
Such a point is identified by two numbers, which represent the horizontal and vertical position, often referred to as x and y, or (x,y). Because there are two numbers involved, in METAPOST this point is called a pair. Its related datatype is therefore `pair`. The following statements assigns the point we showed previously to a pair variable.

```
pair somepoint ; somepoint := (1cm,1.5cm) ;
```

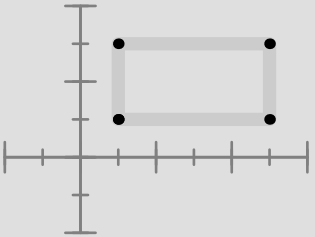
A pair can be used to identify a point in the two dimensional coordinate space, but it can also be used to denote a vector (being a direction or displacement). For instance, `(0,1)` means 'go up'. Looking through math glasses, you may consider them vectors, and if you know how to deal with them, METAPOST may be your friend, since it knows how to manipulate them.

You can connect points and the result is called a path. A path is a straight or bent line, and is not necessarily a smooth curve. An example of a simple rectangular path is:[2]

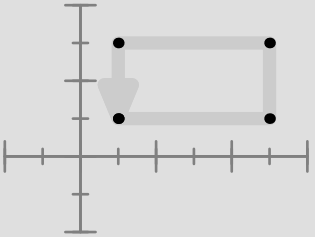
[2] In the next examples we use the debugging features discussed in **chapter 11** to visualize the points, paths and bounding boxes.



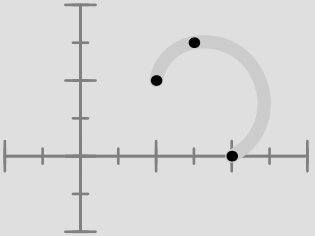
This path is constructed out of four points:



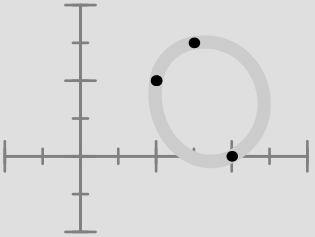
Such a path has both a beginning and end and runs in a certain direction:



A path can be open or closed. The previous path is an example of a closed path. An open path looks like this:



When we close this path —and in a moment we will see how to do this— the path looks like:



The open path is defined as:

```
(1cm,1cm)..(1.5cm,1.5cm)..(2cm,0cm)
```

The 'double period' connector `..` tells METAPOST that we want to connect the lines by a smooth curve. If you want to connect points with straight line segments, you should use `--`.

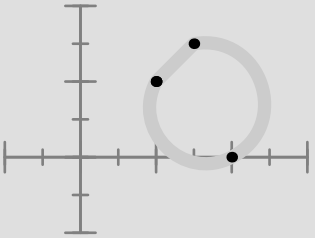
Closing the path is done by connecting the first and last point, using the `cycle` command.

```
(1cm,1cm)..(1.5cm,1.5cm)..(2cm,0cm)..cycle
```

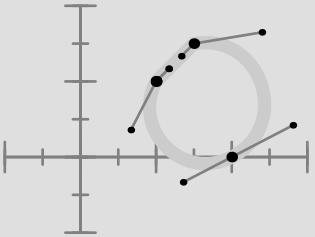
Feel free to use `..` or `--` at any point in your path.

```
(1cm,1cm)--(1.5cm,1.5cm)..(2cm,0cm)..cycle
```

This path, when drawn, looks like this:

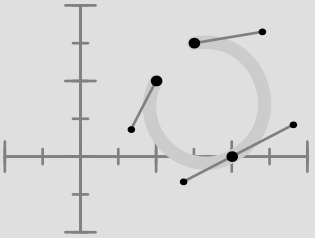


As you can see in some of the previous examples, METAPOST is capable of drawing a smooth curve through the three points that make up the path. We will now examine how this is done.



The six small points are the so called control points. These points pull their parent point in a certain direction. The further away such a point is, the stronger the pull.

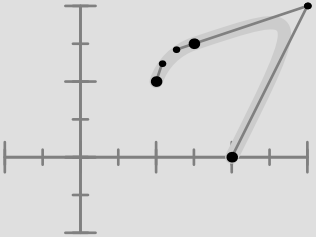
Each point has at most two control points. As you can see in the following graphic, the endpoints of a non closed curve have only one control point.



This time we used the path:

```
(1.5cm,1.5cm)..(2cm,0cm)..(1cm,1cm)
```

When you connect points by a smooth curve, METAPOST will calculate the control points itself, unless you specify one or more of them.



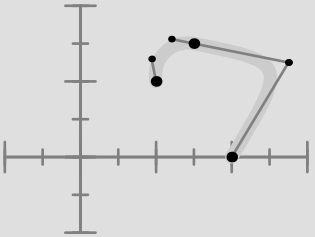
This path is specified as:

```
(1cm,1cm)..(1.5cm,1.5cm)..controls (3cm,2cm)..(2cm,0cm)
```

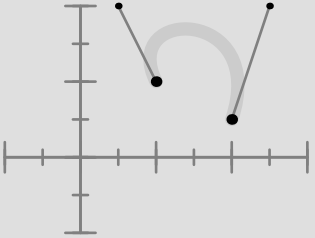
In this path, the second and third point share a control point. Watch how the curve is pulled in that direction. It is possible to pull a bit less by choosing a different control point:

```
(1cm,1cm)..(1.5cm,1.5cm)..controls (2.75cm,1.25cm)..(2cm,0cm)
```

Now we get:



We can also specify a different control point for each connecting segment.



This path is defined as:

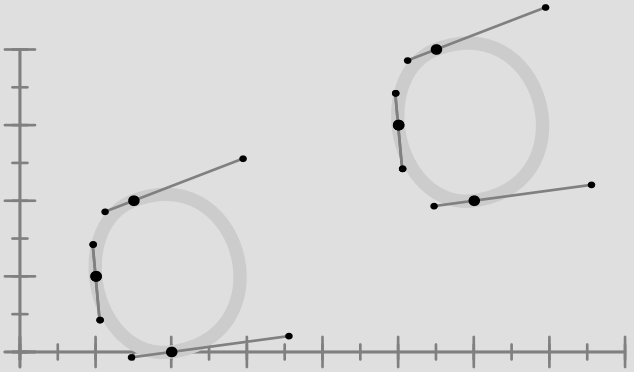
```
(1cm,1cm)..controls (.5cm,2cm) and (2.5cm,2cm)..(2cm,.5cm)
```

1.2 Transformations

We can store a path in a path variable. Before we can use such a variable, we have to allocate its memory slot with `path`.

```
path p ; p := (1cm,1cm)..(1.5cm,2cm)..(2cm,0cm) ;
```

Although we can manipulate any path in the same way, using a variable saves us the effort to key in a path more than once.

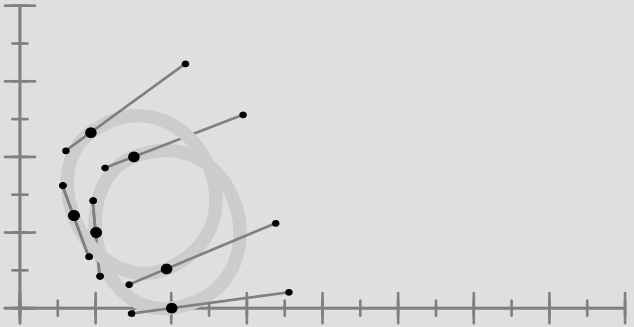


In this graphic, the path stored in `p` is drawn twice, once in its displaced form. The displacement is defined as:

```
p shifted (4cm,2cm)
```

In a similar fashion you can rotate a path. You can even combine shifts and rotations. First we rotate the path 15 degrees counter–clockwise around the origin.

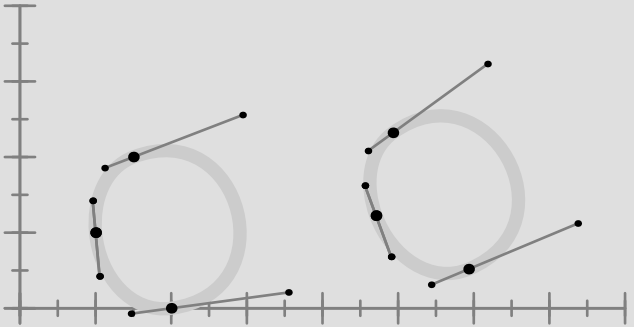
```
p rotated 15
```



This rotation becomes more visible when we also shift the path to the right by saying:

```
rotated 15 shifted (4cm,0cm)
```

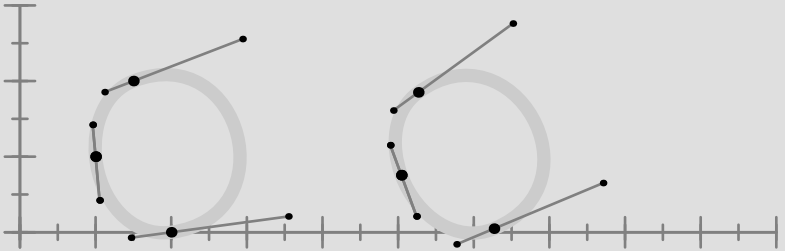
Now we get:



Note that `rotated 15` is equivalent to `p rotatedaround (origin, 15)`.

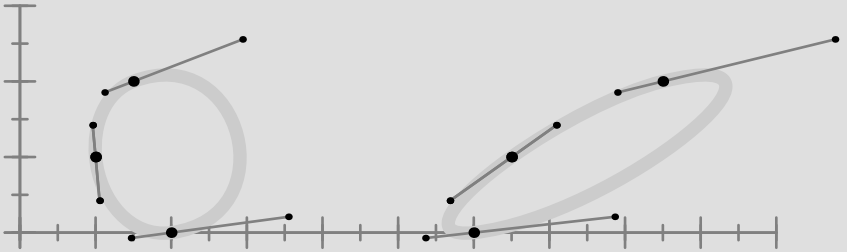
It may make more sense to rotate the shape around its center. This can easily be achieved with the `rotatedaround` command. Again, we move the path to the right afterwards.

```
p rotatedaround(center p, 15) shifted (4cm,0cm)
```



Yet another transformation is slanting. Just like characters can be upright or slanted, a graphic can be:

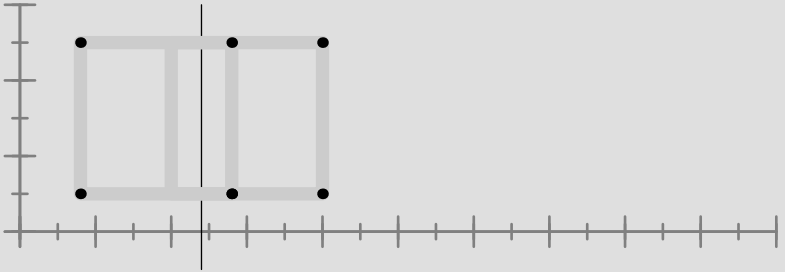
```
p slanted 1.5 shifted (4cm,0cm)
```



The slant operation's main application is in tilting fonts. The x–coodinates are increased by a percentage of their y–coordinate, so here every x becomes $x + 1.5y$. The y–coordinate is left untouched. The following table summarizes the most important primitive transformations that METAPOST supports.

METAPOST code	mathematical equivalent
`(x,y) shifted (a,b)`	$(x + a, y + b)$
`(x,y) scaled s`	(sx, sy)
`(x,y) xscaled s`	(sx, y)
`(x,y) yscaled s`	(x, sy)
`(x,y) zscaled (u,v)`	$(xu - yv, xv + yu)$
`(x,y) slanted s`	$(x + sy, y)$
`(x,y) rotated r`	$(x\cos(r) - y\sin(r), x\sin(r) + y\cos(r))$

The previously mentioned `rotatedaround` is not a primitive but a macro, defined in terms of shifts and rotations. Another transformation macro is mirroring, or in METAPOST terminology, `reflectedabout`.



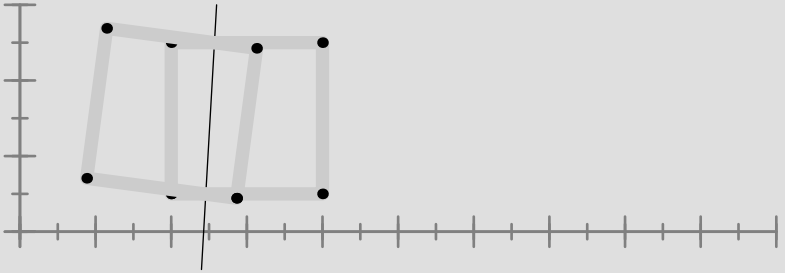
The reflection axis is specified by a pair of points. For example, in the graphic above, we used the following command to reflect the square about a line through the given points.

```
p reflectedabout((2.4cm,-.5),(2.4cm,3cm))
```

The line about which the path is mirrored. Mirroring does not have to be parallel to an axis.

```
p reflectedabout((2.4cm,-.5),(2.6cm,3cm))
```

The rectangle now becomes:



The table also mentions `zscaled`.

A `zscaled` specification takes a vector as argument:

```
p zscaled (2,.5)
```

The result looks like a combination of scaling and rotation, and conforms to the formula in the previous table.

Transformations can be defined in terms of a transform matrix. Such a matrix is stored in a transform variable. For example:

```
transform t ; t := identity scaled 2cm shifted (4cm,1cm) ;
```

We use the associated keyword `transformed` to apply this matrix to a path or picture.

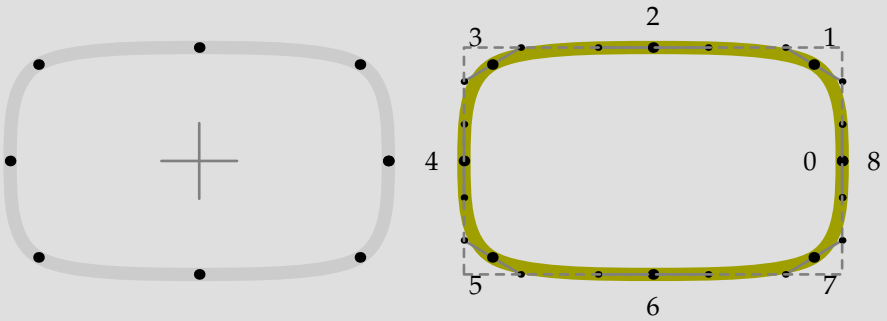
```
p transformed t
```

In this example we've taken the `identity` matrix as starting point but you can use any predefined transformation. The identity matrix is defined in such a way that it scales by a factor of one in both directions and shifts over the zero–vector.

Transform variables can save quite some typing and may help you to force consistency when many similar transformations are to be done. Instead of changing the scaling, shifting and other transformations you can then stick to just changing the one transform variable.

1.3 Constructing paths

In most cases, a path will have more points than the few shown here. Take for instance a so called *super ellipse*.



These graphics provide a lot of information. In this picture the crosshair in the center is the *origin* and the dashed rectangle is the *bounding box* of the super ellipse. The bounding box specifies the position of the graphic in relation to the origin as well as its width and height.

In the graphic on the right, you can see the points that make up the closed path as well as the control points. Each point has a number with the first point numbered zero. Because the path is closed, the first and last point coincide.

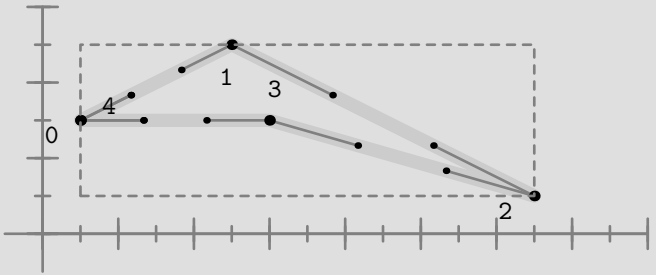
We've used the commands `..` and `--` as path connecting directives. In the next series of examples, we will demonstrate a few more. However, before doing that, we define a few points, using the predefined `z` variables.

```
z0 = (0.5cm,1.5cm) ; z1 = (2.5cm,2.5cm) ;
z2 = (6.5cm,0.5cm) ; z3 = (3.0cm,1.5cm) ;
```

Here `z1` is a short way of saying `(x1,y1)`. When a `z` variable is called, the corresponding `x` and `y` variables are available too. Later we will discuss METAPOST capability to deal with expressions, which are expressed using an `=` instead of `:=`. In this case the expression related to `z0` is expanded into:

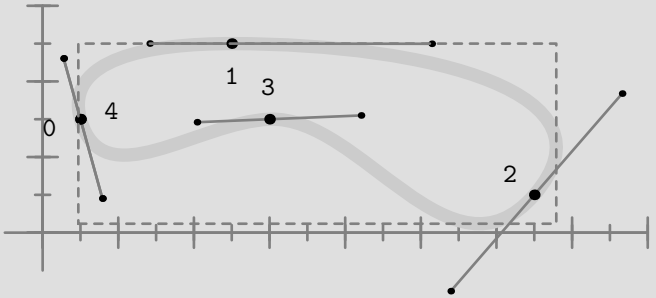
```
z0 = (x0,y0) = (0.5cm,1.5cm) ;
```

But for this moment let's forget about their expressive nature and simply see them as points which we will now connect by straight line segments.



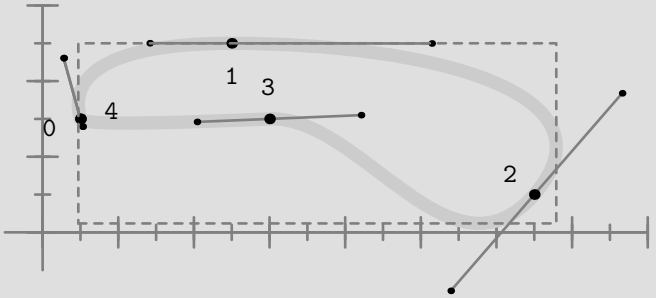

```
"z0--z1--z2--z3--cycle"
```

The smooth curved connection, using `..` looks like:



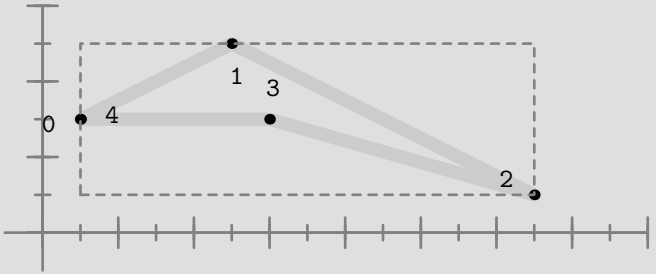

```
"z0..z1..z2..z3..cycle"
```

If we replace the .. by ..., we get a tighter path.



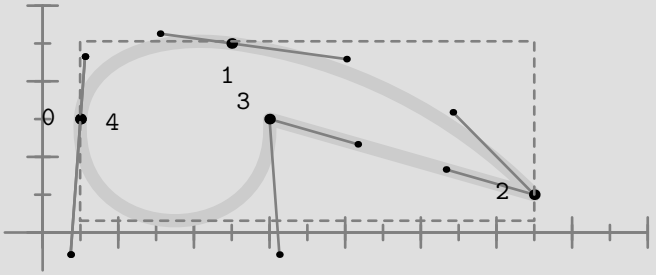

```
"z0...z1...z2...z3...cycle"
```

Since there are .., --, and ..., it will be no surprise that there is also ---.



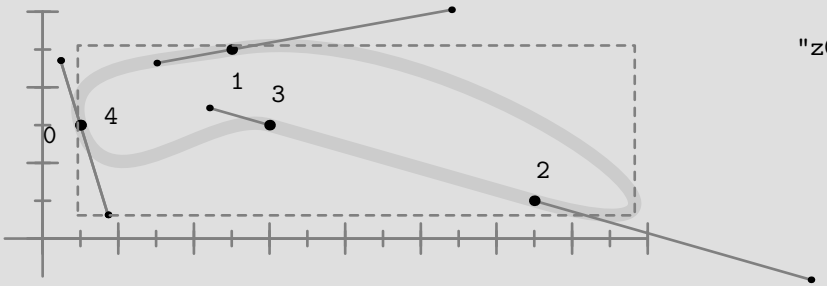

```
"z0---z1---z2---z3---cycle"
```

If you compare this graphic with the one using `--` the result is the same, but there is a clear difference in control points. As a result, combining `..` with `--` or `---` makes a big difference. Here we get a non–smooth connection between the curves and the straight line.



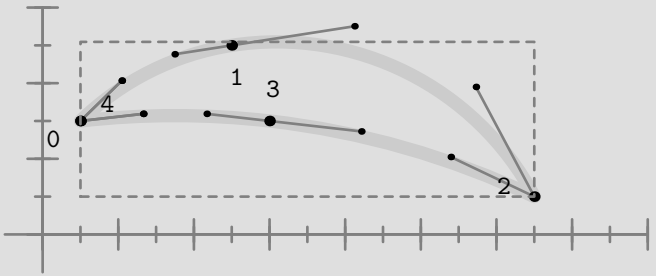

```
"z0..z1..z2--z3..cycle"
```

As you can see in the next graphic, when we use `---`, we get a smooth connection between the straight line and the rest of the curve.



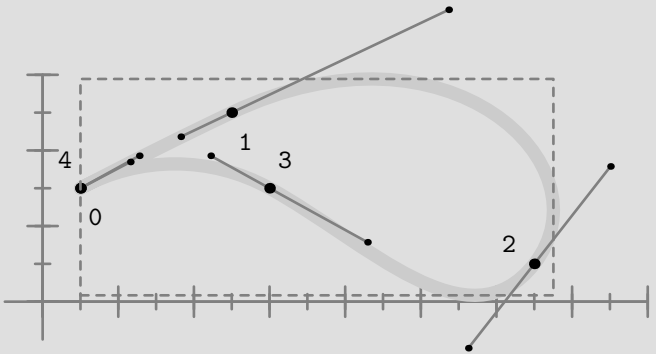

```
"z0..z1..z2---z3..cycle"
```

So far, we have joined the four points as one path. Alternatively, we can constrict subpaths and connect them using the ampersand symbol, `&`.



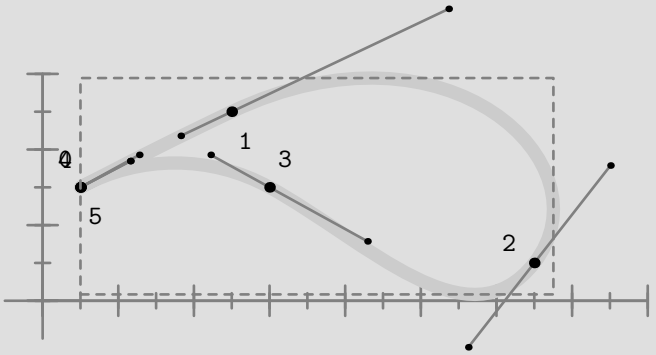

```
"z0..z1..z2 & z2..z3..z0 & cycle"
```

So far we have created a closed path. Closing is done by `cycle`. The following path may look closed but is in fact open.



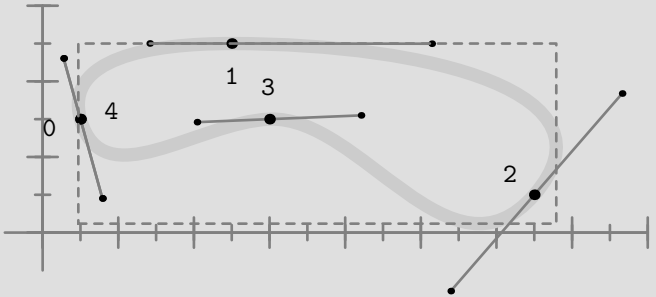

```
"z0..z1..z2..z3..z0"
```

Only a closed path can be filled. The closed alternative looks as follows. We will see many examples of filled closed paths later on.



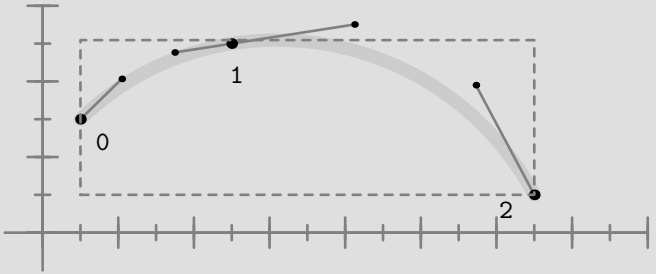

```
"z0..z1..z2..z3..z0..cycle"
```

Here the final .. will try to make a smooth connection, but because we already are at the starting point, this is not possible. However, the `cycle` command can automatically connect to the first point. Watch the difference between the previous and the next path.



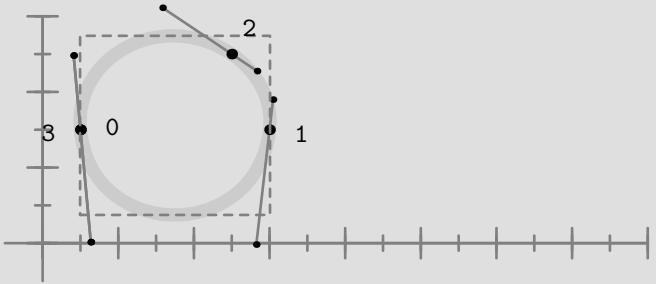

```
"z0..z1..z2..z3..cycle"
```

It is also possible to combine two paths into one that don't have common head and tails. First we define an open path:



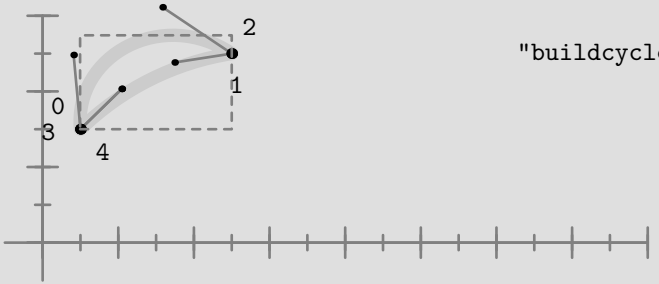

```
"z0..z1..z2"
```

The following path is a closed one, and crosses the previously shown path.




```
"z0..z3..z1..cycle"
```

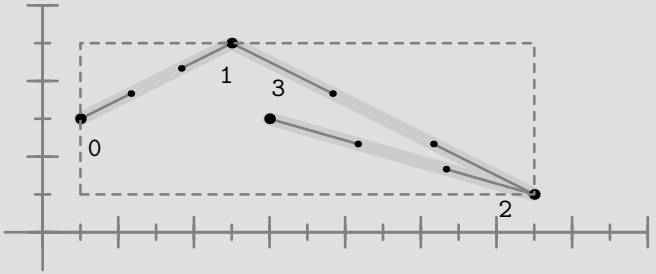
With `buildcycle` we can combine two paths into one.



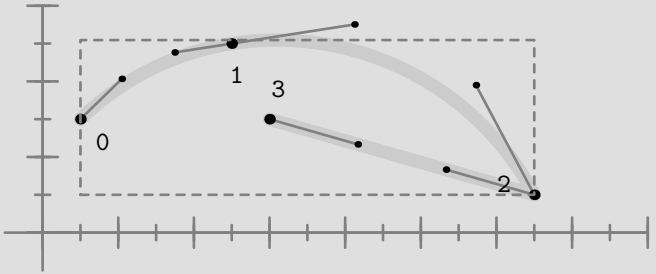

```
"buildcycle(z0..z1..z2 , z0..z3..z1..cycle)"
```

We would refer readers to the METAFONT book and the METAPOST manual for an explanation of the intricacies of the `buildcycle` command. It is an extremely complicated command, and there is just not enough room here to do it justice. We suffice with saying that the paths should cross at least once before the `buildcycle` command can craft a combined path from two given paths. We encourage readers to experiment with this command.

In order to demonstrate another technique of joining paths, we first draw a few strange paths. The last of these three graphics demonstrates the use of `softjoin`.

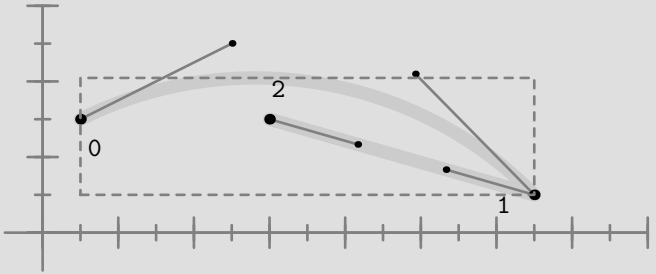



```
"z0--z1..z2--z3"
```



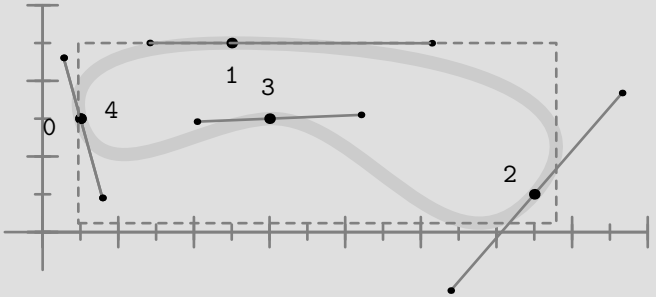

```
"z0..z1..z2--z3"
```

Watch how `softjoin` removes a point in the process of smoothing a connection. The smoothness is accomplished by adapting the control points of the neighbouring points in the appropriate way.




```
"z0--z1 softjoin z2--z3"
```

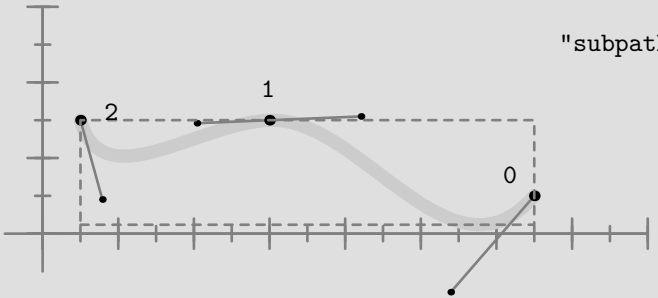
Once a path is known, you can cut off a slice of it. We will demonstrate a few alternative ways of doing so, but first we show one more time the path that we take as starting point.




```
"z0..z1..z2..z3..cycle"
```

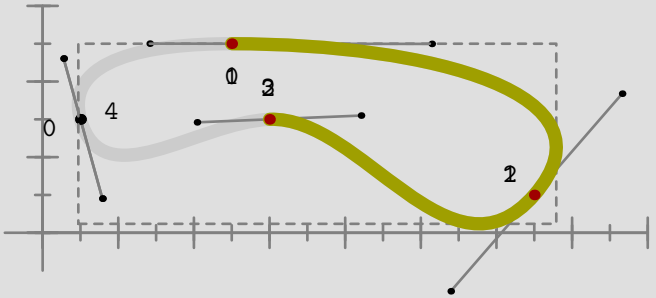
This path is made up out of five points, where the cycle duplicates the first point and connects the loose ends. The first point has number zero.

We can use these points in the `subpath` command, which takes two arguments, specifying the range of points to cut of the path specified after the keyword `of`.



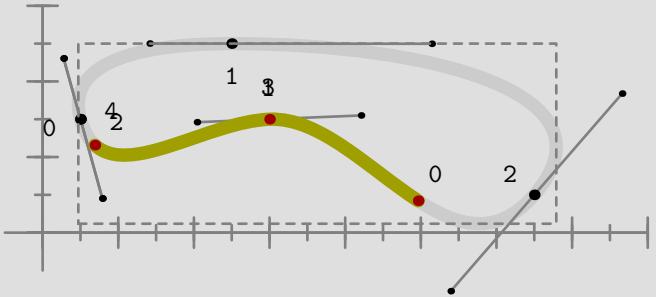

```
"subpath(2,4) of (z0..z1..z2..z3..cycle)"
```

The new (sub)path is a new path with its own points that start numbering at zero. The next graphic shows both the original and the subpath from point 1 upto 3.



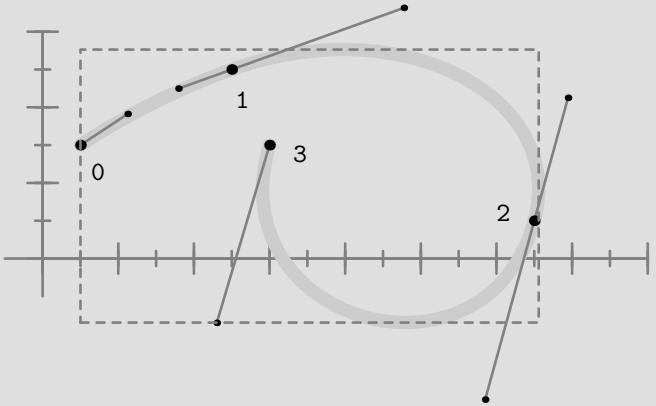

```
"(z0..z1..z2..z3..cycle)"
"subpath(1,3)"
```

In spite of what you may think, a point is not fixed. This is why in METAPOST a point along a path is officially called a time. The next example demonstrates that we can specify any time on the path.



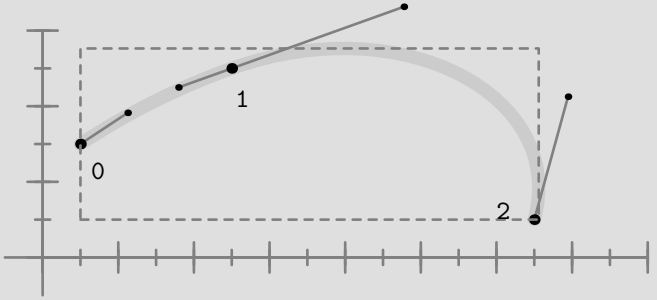

```
"(z0..z1..z2..z3..cycle)"
"subpath(2.45,3.85)"
```

Often we want to take a slice starting at a specific point. This is provided by `cutafter` and its companion `cutbefore`. Watch out, this time we use a non–cyclic path.

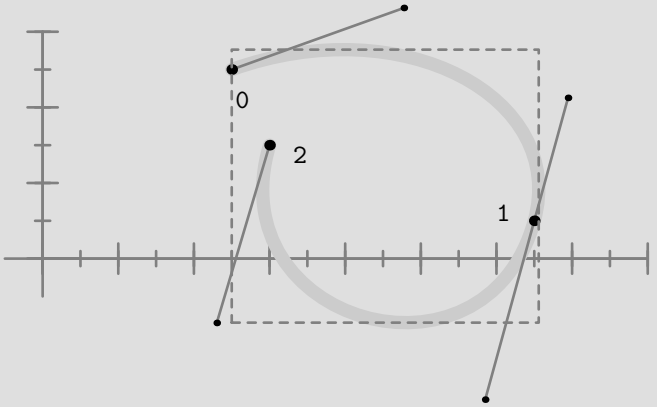



```
"(z0..z1..z2..z3)"
```

When you use `cutafter` and `cutbefore` it really helps if you know in what direction the path runs.

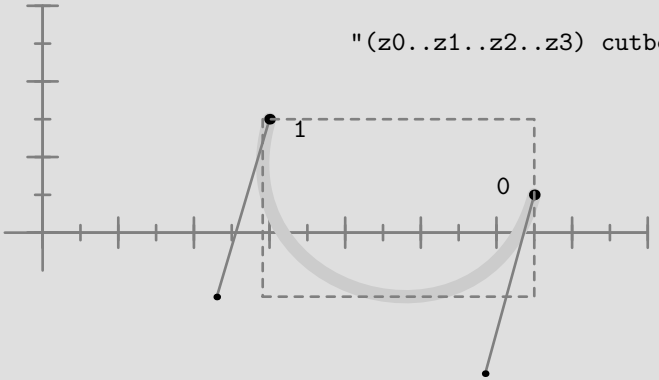



```
"(z0..z1..z2..z3) cutafter z2"
```

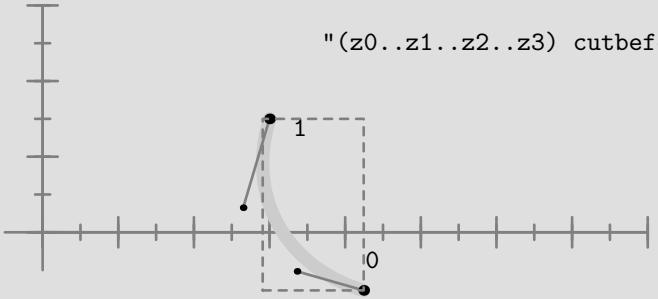



```
"(z0..z1..z2..z3) cutbefore z1"
```

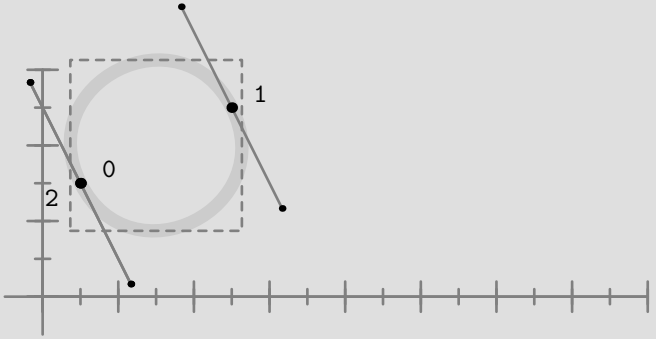
Here is a somewhat silly way of accomplishing the same thing, but it is a nice introduction to METAPOST's `point` operation. In order to use this command effectively, you need to know how many points make up the path.



As with `subpath`, you can use fractions to specify the time on the path, although the resulting point is not necessarily positioned linearly along the curve.



If you really want to know the details of where fraction points are positioned, you should read the METAFONT book and study the source of METAFONT and METAPOST, where you will find the complicated formulas that are used to calculate smooth curves.

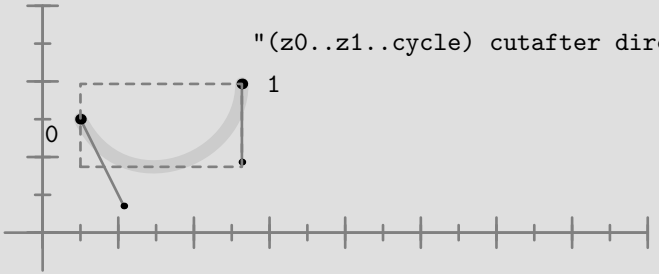



```
"z0..z1..cycle"
```

Like any closed path, this path has points where the tangent is horizontal or vertical. Early in this chapter we mentioned that a pair (or point) can specify a direction or vector. Although any angle is possible, we often use one of four predefined directions:

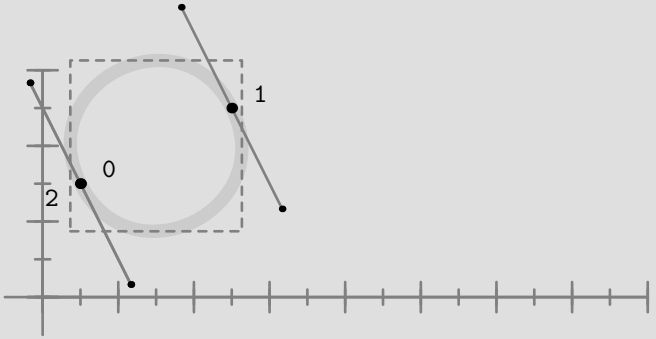
```
right   ( 1, 0)
up      ( 0, 1)
left    (-1, 0)
down    ( 0,-1)
```

We can use these predefined directions in combination with `directionpoint` and `cutafter`. The following command locates the first point on the path that has a tangent that points vertically upward, and then feeds this point to the `cutafter` command.



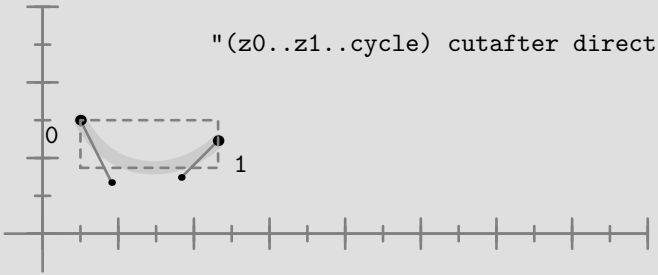

```
"(z0..z1..cycle) cutafter directionpoint up of (z0..z1..cycle)"
```

You are not limited to predefined direction vectors. You can provide a pair to indicate a direction. In the next example we use the following cyclic path:




```
"z0..z1..cycle"
```

Using ( ) is not mandatory but makes the expression look less complicated.



We will apply these commands in the next chapters, but first we will finish our introduction in METAPOST. We have seen how a path is constructed and what can be done with it. Now it is time to demonstrate how such a path is turned into a graphic.

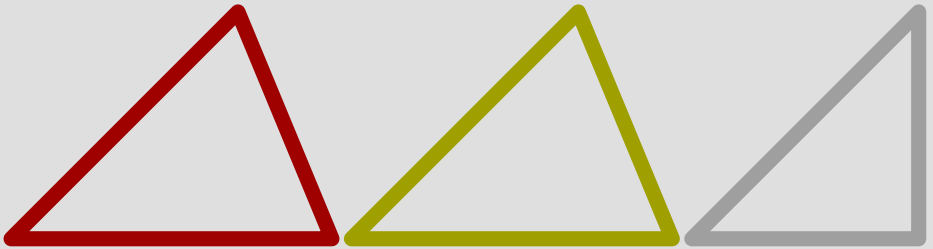
1.4 Angles

You can go from angles to vectors and vice versa using the `angle` and `dir` functions. The next example show both in action.

```
pickup pencircle scaled 2mm ;
draw (origin -- dir(45) -- dir(0) -- cycle)
  scaled 3cm                   withcolor .625red ;
draw (origin -- dir(angle(1,1)) -- dir(angle(1,0)) -- cycle)
  scaled 3cm shifted (3.5cm,0) withcolor .625yellow ;
draw (origin -- (1,1) -- (1,0) -- cycle)
  scaled 3cm shifted (7cm,0)   withcolor .625white ;
```

The `dir` command returns an unit vector, which is why the first two shapes look different and are smaller than the third one. We can compensate for that by an additional scaling:

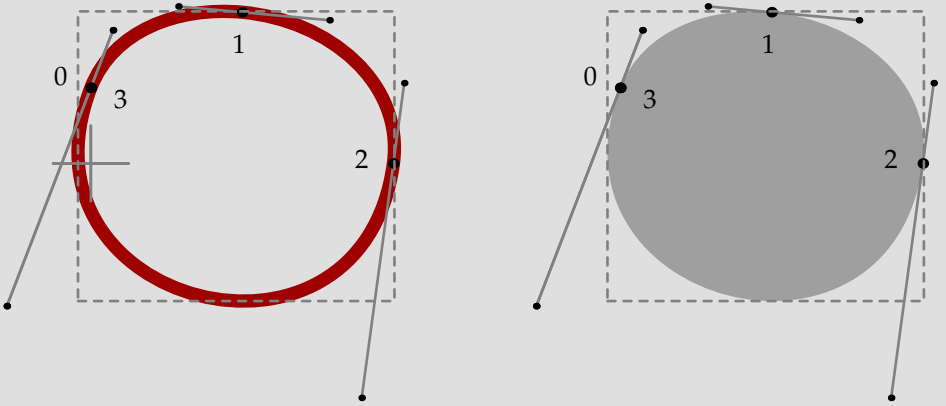
```
pickup pencircle scaled 2mm ;
draw (origin -- dir(45) -- dir(0) -- cycle)
  scaled sqrt(2) scaled 3cm                    withcolor .625red ;
draw (origin -- dir(angle(1,1)) -- dir(angle(1,0)) -- cycle)
  scaled sqrt(2) scaled 3cm shifted (4.5cm,0) withcolor .625yellow ;
draw (origin -- (1,1) -- (1,0) -- cycle)
                scaled 3cm shifted (9cm,0)   withcolor .625white ;
```



1.5

Drawing pictures

Once a path is defined, either directly or as a variable, you can turn it into a picture. You can draw a path, like we did in the previous examples, or you can fill it, but only if it is closed.



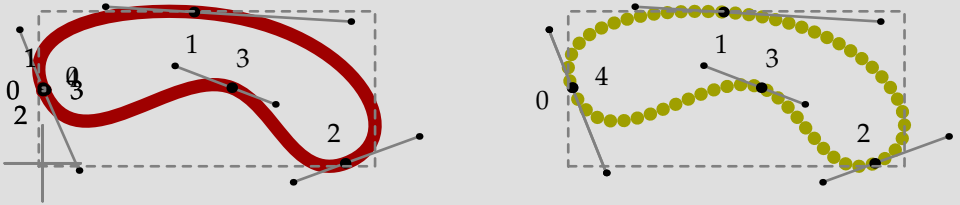
Drawing is done by applying the `draw` command to a path, as in:

```
draw (0cm,1cm)..(2cm,2cm)..(4cm,0cm)..cycle ;
```

The rightmost graphic was made with `fill`:

```
fill (0cm,1cm)..(2cm,2cm)..(4cm,0cm)..cycle ;
```

If you try to duplicate this drawing, you will notice that you will get black lines instead of red and a black fill instead of a gray one. When drawing or filling a path, you can give it a color, use all kinds of pens, and achieve special effects like dashes or arrows.



These two graphics were defined and drawn using the following commands. Later we will explain how you can set the line width (or penshape in terms of METAPOST).

```
path p ; p := (0cm,1cm)..(2cm,2cm)..(4cm,0cm)..(2.5cm,1cm)..cycle ;
drawarrow p withcolor .625red ;
draw p shifted (7cm,0) dashed withdots withcolor .625yellow ;
```

Once we have drawn one or more paths, we can store them in a picture variable. The straightforward way to store a picture is to copy it from the current picture:

```
picture pic ; pic := currentpicture ;
```

The following command effectively clears the picture memory and allows us to start anew.

```
currentpicture := nullpicture ;
```

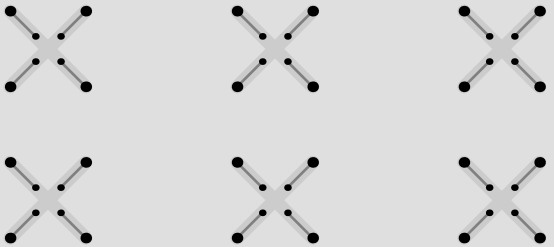
We can shift, rotate and slant the picture stored in `pic` as we did with paths. We can say:

```
draw pic rotated 45 withcolor red ;
```

A picture can hold multiple paths. You may compare a picture to grouping as provided by drawing applications.

```
draw (0cm,0cm)--(1cm,1cm) ; draw (1cm,0cm)--(0cm,1cm) ;
picture pic ; pic := currentpicture ;
draw pic shifted (3cm,0cm) ; draw pic shifted (6cm,0cm) ;
pic := currentpicture ; draw pic shifted (0cm,2cm) ;
```

We first draw two paths and store the resulting 'cross' in a picture variable. Then we draw this picture two times, so that we now have three copies of the cross. We store the accumulated drawing again, so that after duplication, we finally get six crosses.



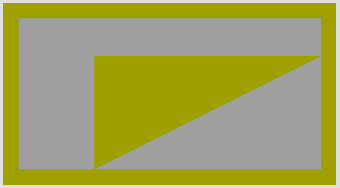
You can often follow several routes to reach the same solution. Consider for instance the following graphic.

```
fill (0,0)--(ww,0)--(ww,hh)--(w,hh)--(w,h)--(0,h)--cycle ;
fill (ww,0)--(w,0)--(w,hh)--cycle ;
```

The points that are used to construct the paths are defined using the constants `w`, `h`, `ww` and `hh`. These are defined as follows:

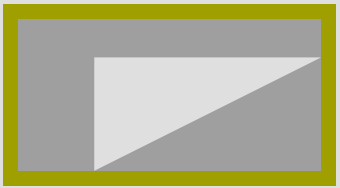
```
w := 4cm ; h := 2cm ; ww := 1cm ; hh := 1.5cm ;
```

In this case we draw two shapes that leave part of the rectangle uncovered. If you have a background, this technique allows the background to 'show through' the graphic.

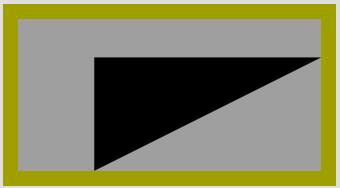


A not uncommon practice when making complicated graphics is to use unfill operations. Since METAPOST provides one, let us see what happens if we apply this command.

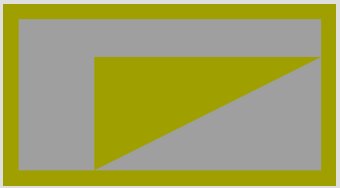
```
fill (0,0)--(w,0)--(w,h)--(0,h)--cycle ;
unfill (ww,0)--(w,hh)--(ww,hh)--cycle ;
```



This does not always give the desired effect, because METAPOST's `unfill` is not really an unfill, but a `fill` with color `background`. Since this color is white by default, we get what we just showed. So, if we set `background` to `black`, using `background := black`, we get:

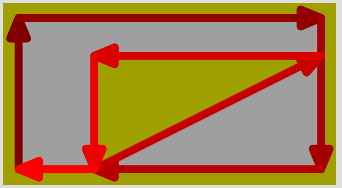


Of course, you can set the variable `background` to a different color, but this does not hide the fact that METAPOST lacks a real unfill operation.



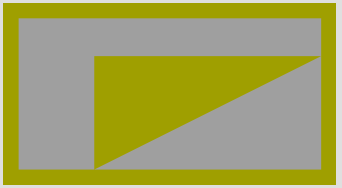
Since we don't consider this `unfill` a suitable operator, you may wonder how we achieved the above result.

```
fill (0,0)--(0,h)--(w,h)--(w,0)--(ww,0)--(w,hh)--(ww,hh)--
     (ww,0)--cycle ;
```



This feature depends on the POSTSCRIPT way of filling closed paths, which comes down to filling either the left or the right hand side of a curve. The following alternative works too.

```
fill (0,0)--(0,h)--(w,h)--(w,hh)--(ww,hh)--(ww,0)--(w,hh)--
     (w,0)--cycle ;
```



The next alternative will fail. This has to do with the change in direction at point `(0,0)` halfway through the path. Sometimes changing direction can give curious but desirable effects, but here it brings no good.

```
fill (0,0)--(0,h)--(w,h)--(w,0)--(0,0)--(ww,0)--(ww,hh)--
     (w,hh)--(ww,0)--cycle ;
```

This path fails because of the way POSTSCRIPT implements its fill operator. More details on how POSTSCRIPT defines fills can be found in the reference manuals.

Some of the operations we have seen are hard coded into METAPOST and are called primitives. Others are defined as macros, that is, a sequence of METAPOST commands. Since they are used often, you may expect `draw` and `fill` to be primitives, but they are not. They are macros defined in terms of primitives.

Given a path `pat`, you can consider a `draw` to be defined in terms of:

```
addto currentpicture doublepath pat
```

The `fill` command on the other hand is defined as:

```
addto currentpicture contour pat
```

Both macros are actually a bit more complicated but this is mainly due to the fact that they also have to deal with attributes like the pen and color they draw with.

You can use `doublepath` and `contour` directly, but we will use `draw` and `fill` whenever possible.

Given a picture `pic`, the following code is valid:

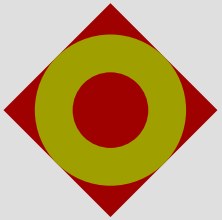
```
addto currentpicture also pic
```

You can add pictures to existing picture variables, where `currentpicture` is the picture that is flushed to the output file. Watch the subtle difference between adding a `doublepath`, `contour` or `picture`.

Here is another nice example of what happens when you fill a path that is partially reversed.

```
fill fullsquare rotated 45 scaled 2cm
    withcolor .625 red ;
fill fullcircle scaled 2cm -- reverse fullcircle scaled 1cm -- cycle
    withcolor .625 yellow;
```

The inner circle is indeed not filled:



1.6 Variables

At this point you may have noted that METAPOST is a programming language. Contrary to some of today's languages, METAPOST is a simple and clean language. Actually, it is a macro language. Although METAPOST and TEX are a couple, the languages differ in many aspects. If you are using both, you will sometimes wish that features present in one would be available in the other. When using both languages, in the end you will understand why the conceptual differences make sense.

Being written in PASCAL, it will be no surprise that METAPOST has some PASCAL–like features, although some may also recognize features from ALGOL68 in it.

First there is the concept of variables and assignments. There are several data types, some of which we already have seen.

numeric	real number in the range −4096... + 4096
boolean	a variable that takes one of two states: true or false
pair	point or vector in 2–dimensional space
path	a piecewise collection of curves and line segments
picture	collection of stroked or filled paths
string	sequence of characters, like `"metapost"`
color	vector of three (rgb) or four (cmyk) numbers

There are two additional types, `transform` and `pen`, but we will not discuss these in depth.

transform	transformation vector with six elements
pen	pen specification

You can achieve interesting effects by using pens with certain shapes. For the moment you may consider a pen to be a path itself that is applied to the path that is drawn.

The `numeric` data type is used so often that it is the default type of any non declared variable. This means that

```
n := 10 ;
```

is the same as

```
numeric n ; n := 10 ;
```

When writing collections of macros, it makes sense to use the second method, because you can never be sure if `n` isn't already declared as a picture variable, and assigning a numeric to a picture variable is not permitted.

Because we often deal with collections of objects, such as a series of points, all variables can be organized in arrays. For instance:

```
numeric n[] ; n[3] := 10 ; n[5] := 13 ;
```

An array is a collection of variables of the same type that are assigned and accessed by indexing the variable name, as in `n[3] := 5`. Multi–dimensional arrays are also supported. Since you need a bit of imagination to find an application for 5–dimensional arrays, we restrict ourselves to a two–dimensional example.

```
numeric n[][] ; n[2][3] := 10 ;
```

A nice feature is that the bounds of such an array needs not to be set beforehand. This also means that each cell that you access is reported as *unknown* unless you have assigned it a value.

Behind the screens there are not really arrays. It's just a matter of creating hash entries. It might not be obvious, but the following assignments are all equivalent:

```
i_111_222     := 1cm ;
i_[111]_[222] := 1cm ;
i_[111][222]  := 1cm ;
draw
    image (
        draw (0cm,i_111_222) ;
        draw (1cm,i_[111]_[222]) ;
        draw (2cm,i_[111][222]) ;
    )
```

```
withpen pencircle scaled 5mm
withcolor .625 red ;
```

Sometimes METAPOST ways are mysterious:

1.7 Conditions

The existence of boolean variables indicates the presence of conditionals. Indeed, the general form of METAPOST's conditional follows:

```
if n=10 : draw p ; else : draw q ; fi ;
```

Watch the colons after the if and else clause. They may not be omitted. The semi–colons on the other hand, are optional and depend on the context. You may say things like:

```
draw if n=10 : p ; else : q ; fi ;
```

Here we can omit a few semi–colons:

```
draw if n=10 : p else : q fi withcolor red ;
```

Adding semi–colons after `p` and `q` will definitely result in an error message, since the semi–colon ends the draw operation and `withcolor red` becomes an isolated piece of nonsense.

There is no case statement available, but for most purposes, the following extension is adequate:

```
draw p withcolor if n<10 : red elseif n=10 : green else : blue fi ;
```

There is a wide repertoire of boolean tests available.

```
if picture p :
if known   n :
if odd     i :
if cycle   q :
```

Of course, you can use `and`, `or`, `not`, and `( )` to construct very advanced boolean expressions. If you have a bit of programming experience, you will appreciate the extensive support of conditionals in METAPOST.

1.8 Loops

Yet another programming concept present in METAPOST is the loop statement, the familiar 'for loop' of all programming languages.

```
for i=0 step 2 until 20 :
  draw (0,i) ;
endfor ;
```

As explained convincingly in Niklaus Wirth's book on algorithms and datastructures, the for loop is the natural companion to an array. Given an array of length n, you can construct a path out of the points that make up the array.

```
draw for i=0 step 1 until n-1 : p[i] .. endfor p[n] ;
```

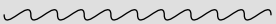
If the step increment is not explicitly stated, it has an assumed value of 1. We can shorten the previous loop construct as follows:

```
draw for i=0 upto n-1 : p[i] .. endfor p[n] ;
```

After seeing `if` in action, the following `for` loop will be no surprise:

```
draw origin for i=0 step 10 until 100 : ..{down}(i,0) endfor ;
```

This gives the zig–zag curve:



You can use a loop to iterate over a list of objects. A simple 3–step iteration is:

```
for i=p,q,r :
  fill i withcolor .8white ;
  draw i withcolor red ;
endfor ;
```

Using `for` in this manner can sometimes save a bit of typing. The list can contain any expression, and may be of different types.

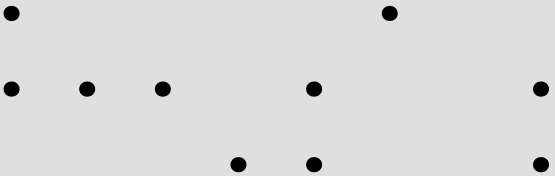
In the previous example the `i` is an independent variable, local to the for loop. If you want to change the loop variable itself, you need to use `forsuffixes`. In the next loop the paths `p`, `q` and `r` are all shifted.

```
forsuffixes i = p, q, r :
  i := i shifted (3cm,2cm) ;
endfor ;
```

Sometimes you may want to loop forever until a specific condition occurs. For this, METAPOST provides a special looping mechanism:

```
numeric done[][], i, j, n ; n := 0 ;
forever :
  i := round(uniformdeviate(10)) ; j := round(uniformdeviate(2)) ;
  if unknown done[i][j] :
    drawdot (i*cm,j*cm) ; n := n + 1 ; done[i][j] := n ;
  fi ;
  exitif n = 10 ;
endfor ;
```

Here we remain in the loop until we have 10 points placed. We use an array to keep track of placed points. The METAPOST macro `uniformdeviate(n)` returns a random number between 0 and n and the `round` command is used to move the result toward the nearest integer. The `unknown` primitive allows us to test if the array element already exists, otherwise we exit the conditional. This saves a bit of computational time as each point is drawn and indexed only once.



The loop terminator `exitif` and its companion `exitunless` can be used in `for`, `forsuffixes` and `forever`.

1.9 Macros

In the previous section we introduced `upto`. Actually this is not part of the built in syntax, but a sort of shortcut, defined by:

```
def upto = step 1 until enddef ;
```

You just saw a macro definition where `upto` is the name of the macro. The counterpart of `upto` is `downto`. Whenever you use `upto`, it is replaced by `step 1 until`. This replacement is called expansion.

There are several types of macros. A primary macro is used to define your own operators. For example:

```
primarydef p doublescaled s =
  p xscaled (s/2) yscaled (s*2)
enddef ;
```

Once defined, the `doublescaled` macro is implemented as in the following example:

```
draw somepath doublescaled 2cm withcolor red ;
```

When this command is executed, the macro is expanded. Thus, the actual content of this command becomes:

```
draw somepath xscaled 1cm yscaled 4cm withcolor red ;
```

If in the definition of `doublescaled` we had added a semi–colon after `(s*2)`, we could not have set the color, because the semicolon ends the statement. The `draw` expects a path, so the macro can best return one.

A macro can take one or more arguments, as in:

```
def drawrandomscaledpath (expr p, s) =
  draw p xscaled (s/2) yscaled (s*2) ;
enddef ;
```

When using this macro, it is expected that you will pass it two parameters, the first being a path, the second a numeric scale factor.

```
drawrandomscaledpath(fullsquare, 3cm) ;
```

Sometimes we want to return a value from a macro. In that case we must make sure that any calculations don't interfere with the expectations. Consider:

```
vardef randomscaledpath(expr p, s) =
  numeric r ; r := round(1 + uniformdeviate(4)) ;
  p xscaled (s/r) yscaled (s*r)
enddef ;
```

Because we want to use the same value of `r` twice, we have to use an intermediate variable. By using a `vardef` we hide everything but the last statement. It is important to distinguish `def` macros from those defined with `vardef`. In the latter case, `vardef` macros are not a simple expansion and replacement. Rather, `vardef` macros return the value of their last statement. In the case of the `randomscaledpath` macro, a path is returned. This macro is used in the following manner:

```
path mypath ; mypath := randomscaledpath(unitsquare,4cm) ;
```

Note that we send `randomscaledpath` a path (`unitsquare`) and a scaling factor (`4cm`). The macro returns a scaled path which is then stored in the path variable `mypath`.

The following argument types are accepted:

expr	something that can be assigned to a variable
text	arbitrary METAPOST code ending with a ;
suffix	a variable bound to another variable

An expression is passed by value. This means that in the body of the macro, a copy is used and the original is left untouched. On the other hand, any change to a variable passed as suffix is also applied to the original.

Local variables must be handled in a special manner, since they may conflict with variables used elsewhere. This is because all variables are global by default. The way out of this problem is using grouping in combination with saving variables. The use of grouping is not restricted to macros and may be used anywhere in your code. Variables saved and declared in a group are local to that group. Once the group is exited the variables cease to exist.

```
vardef randomscaledpath(expr p, s) =
  begingroup ; save r ; numeric r ;
    r := round(1 + uniformdeviate(4)) ;
    p xscaled (s/r) yscaled (s*r)
  endgroup
enddef ;
```

In this particular case, we could have omitted the grouping, since `vardef` macros are always grouped automatically. Therefore, we could have defined the macro as:

```
vardef randomscaledpath(expr p, s) =
  save r ; numeric r ; r := round(1 + uniformdeviate(4)) ;
  p xscaled (s/r) yscaled (s*r)
enddef ;
```

The command `save r` declares that the variable `r` is local to the macro. Thus, any changes to the (new) numeric variable `r` are local and will not interfere with a variable `r` defined outside the macro. This is important to understand, as variables outside the macro are global and accessible to the code within the body of the macro.

Macro definitions may be nested, but since most METAPOST code is relatively simple, it is seldom needed. Nesting is discouraged as it makes your code less readable.

Besides def and vardef, METAPOST also provides the classifiers primarydef, secondarydef and tertiarydef. You can use these classifiers to define macros like those provided by METAPOST itself:

```
primarydef   x mod               y = ... enddef ;
secondarydef p intersectionpoint q = ... enddef ;
tertiarydef  p softjoin          q = ... enddef ;
```

A primary macro acts like the binary operators * or scaled and shifted. Secondary macros are like +, - and logical or, and take less precedence. The tertiary operators like < or the path and string concatenation operator & have tertiary macros as companions. More details can be found in the METAFONT book. When it comes to taking precedence, METAPOST tries to be as natural as possible, in the sense that you need to provide as few ( )'s as possible. When in doubt, or when surprised by unexpected results, use parentheses.

1.10 Arguments

The METAPOST macro language is rather flexible in how you feed arguments to macros. If you have only one argument, the following definitions and calls are valid.

```
def test  expr a  = enddef ; test (a) ; test a ;
def test (expr a) = enddef ; test (a) ; test a ;
```

A more complex definition is the following. As you can see, you can call the test macro in your favorite way.

```
def test (expr a,b) (expr c,d) = enddef ;

test (a) (b) (c) (d) ;
test (a,b) (c,d) ;
```

```
test (a,b,c) (d) ;
test (a,b,c,d) ;
```

The type of the arguments is one of `expr`, `primary` or `suffix`. When fetching arguments, METAPOST uses the type to determine how and what to grab. A fourth type is `text`. When no parenthesis are used, a `text` argument grabs everything upto the next semicolon.

```
def test (expr a) text b = enddef ;

test (a) ; test (a) b ;
```

You can use a `text` to grab arguments like `withpen pencircle scaled 10 withcolor red`. Because `text` is so hungry, you may occasionally need a two stage definition:

```
def test    expr a         = dotext(a) enddef ;
def dotest (expr a) text b = ...       enddef ;

test a ; test a b ;
```

This definition permits arguments without parenthesis, which is something you want with commands like `draw`.

The `vardef` alternative behaves in a similar way. It always provides grouping. You need to generate a return value and as a result may not end with a semicolon.

You may consider the whole `vardef` to be encapsulated into parenthesis and thereby to be a (self contained) variable. Adding additional parenthesis often does more harm than good:

```
vardef test (expr a) =
  ( do tricky things with a ; manipulated_a )
```

```
enddef ;
```

Here the tricky things become part of the return value, which quite certainly is something that you don't want.

The three operator look–alike macro definitions are less flexible and have the definition scheme:

```
primarydef   x test y = enddef ;
secondarydef x test y = enddef ;
tertiarydef  x test y = enddef ;
```

When defining macros using this threesome you need to be aware of the associated priorities. When using these definitions, you also have to provide your own grouping.

In the plain METAPOST macro collection (`plain.mp`) you can find many examples of clever definitions. The following (simplified) version of `min` demonstrates how we use the argument handler to isolate the first argument from the provided list, simply by using two arguments.

```
vardef min (expr u) (text t) =
  save min_u ; min_u := u ;
  for uu = t : if uu<u : min_u := uu ; fi endfor
  min_u
enddef ;
```

The special sequence `@#` is used to pick up a so called delimited argument:

```
vardef TryMe@#(expr x) =
  % we can now use @#, which is just text
enddef ;
```

This feature is used in the definition of z as used in z1 or z234:

```
vardef z@# = (x@#,y@#) enddef ;
```

Other applications can be found in the label drawing macros where the anchor point is assigned to the obscure variable @#.

1.11 Pens

When drawing, three attributes can be applied to it: a dashpattern, a pen and/or a color. You may consider an arrowhead an attribute, but actually it is just an additional drawing, appended to the path.

The (predefined) pencircle attribute looks like:

```
withpen pencircle
```

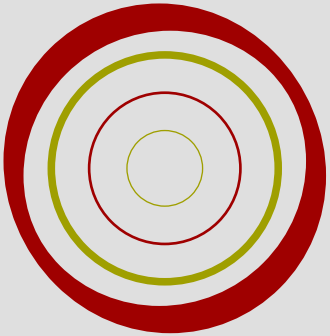
where pencircle is a special kind of path, stored in a pen variable. Like any path, you can transform it. You can scale it equally in all directions:

```
withpen pencircle scaled 1mm
```

You can also provide unequal scales, creating an elliptically shaped and rotated pen.

```
withpen pencircle xscaled 2mm yscaled 4mm rotated 30
```

In the following graphic, the circle in the center is drawn without any option, which means that the default pen is used, being a pencircle with a radius of half a base point. The other three circles are drawn with different pen specifications.



If you forget about the colors, the METAPOST code to achieve this is as follows.

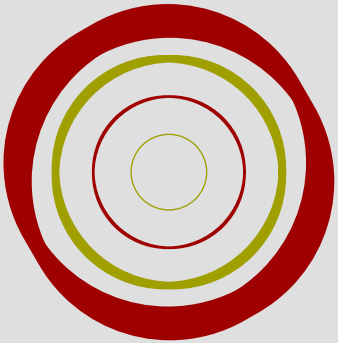
```
path p ; p := fullcircle scaled 1cm ;
draw p ;
draw p scaled 2 withpen pencircle ;
draw p scaled 3 withpen pencircle scaled 1mm ;
draw p scaled 4 withpen pencircle xscaled 2mm yscaled 4mm rotated 30 ;
```

If this were the only way of specifying a pen, we would be faced with a considerable amount of typing, particularly in situations where we use pens similar to the fourth specification above. For that reason, METAPOST supports the concept of a current pen. The best way to set this pen is to use the `pickup` macro.

```
pickup pencircle xscaled 2mm yscaled 4mm rotated 30 ;
```

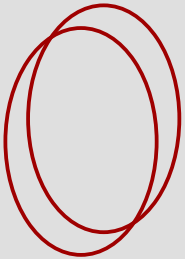
This macro also stores some characteristics of the pen in variables, so that they can be used in (the more complicated) calculations that are involved in situations like drawing font–like graphics.

If we substitute `pencircle` by `pensquare`, we get a different kind of shapes. In the non rotated pens, the top, bottom, left and right parts of the curve are thinner.

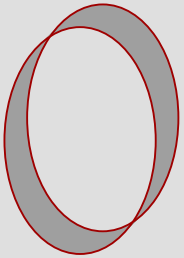


You should look at pens in the way an artist does. He follows a shape and in doing so he or she twists the pen (and thereby the nib) and puts more or less pressure on it.

The chance that you have an appropriate pen laying at your desk is not so large, but you can simulate the following METAPOST's pen by taking two pencils and holding them together in one hand. If you position them in a 45 degrees angle, and draw a circle, you will get something like:



If you take a calligraphic pen with a thin edge of .5cm, you will get:



You can define such a pen yourself:

```
path p ; p := fullcircle xscaled 2cm yscaled 3cm ;
pen doublepen ; doublepen := makepen ((0,0)--(.3cm,.3cm)) ;
pickup doublepen ; draw p ;
```

Here we define a new pen using the `pen` command. Then we define a path, and make a pen out of it using the `makepen` macro. The path should be a relatively simple one, otherwise METAPOST will complain.

You can use `makepen` with the previously introduced `withpen`:

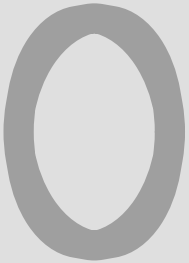
```
draw p withpen makepen ((0,0)--(.3cm,.3cm)) ;
```

and `pickup`:

```
pickup makepen ((0,0)--(.3cm,.3cm)) ; draw p ;
```

You can use `makepen` and `makepath` to convert paths into pens and vice versa.

Pens are very important when defining fonts, and METAFONT is meant to be a font creation tool. Since METAPOST has a slightly different audience, it lacks some features in this area, but offers a few others instead. Nevertheless, one can try to design a font using METAPOST. Of course, pens are among the designers best kept secrets. But even then, not every O is a nice looking one.



1.12 Joining lines

The way lines are joined or end is closely related to the way POSTSCRIPT handles this. By setting the variables `linejoin` and `linecap`, you can influence the drawing process. **Figure 1.1** demonstrates the alternatives. The gray curves are drawn with both variables set to `rounded`.

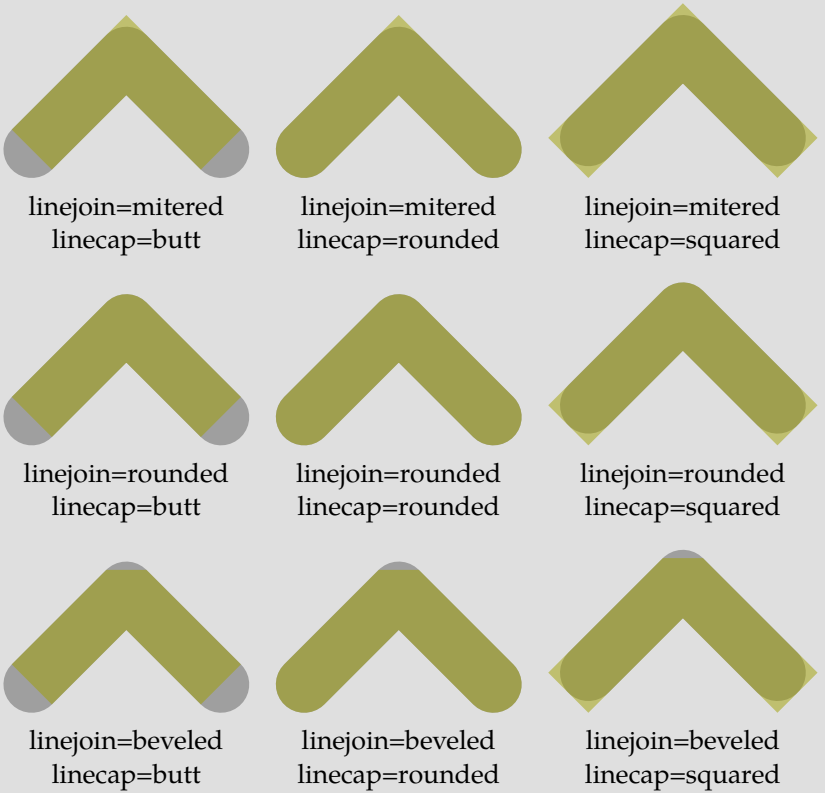


Figure 1.1 The nine ways to end and join lines.

By setting the variable `miterlimit`, you can influence the mitering of joints. The next example demonstrates that the value of this variable acts as a trigger.

```
interim linejoin := mitered ;
for i :=1 step 1 until 5 :
  interim miterlimit := i*pt ;
  draw ((0,0)--(.5,1)--(1,0)) shifted (1.5i,0) scaled 50pt
    withpen pencircle scaled 10pt withcolor .625red ;
endfor ;
```

The variables `linejoin`, `linecap` and `miterlimit` are so called *internal* variables. When we prefix their assignments by `interim`, the setting will be local within groups, like `beginfig ... endfig`).

1.13 Colors

So far, we have seen some colors in graphics. It must be said that METAPOST color model is not that advanced, although playing with colors in the METAPOST way can be fun. In later chapters we will discuss some extensions that provide shading.

Colors are defined as vectors with three components: a red, green and blue one. Like pens, colors have their with–command:

```
withcolor (.4,.5.,6)
```

You can define color variables, like:

```
color darkred ; darkred := (.625,0.0) ;
```

You can now use this color as:

```
withcolor darkred
```

Given that `red` is already defined, we also could have said:

```
withcolor .625red
```

Because for METAPOST colors are just vectors, you can do things similar to points. A color halfway red and green is therefore accomplished with:

```
withcolor .5[red,green]
```

Since only the RGB color space is supported, this is about all we can tell about colors for this moment. Later we will discuss some nasty details.

1.14 Dashes

A dash pattern is a simple picture that is build out of straight lines. Any slightly more complicated picture will be reduced to straight lines and a real complicated one is rejected, and in this respect METAPOST considers a circle to be a complicated path.

The next example demonstrates how to get a dashed line. First we built picture p, that we apply to a path. Here we use a straight path, but dashing can be applied to any path.

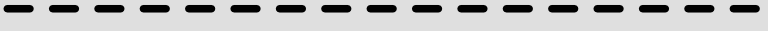
```
picture p ; p := nullpicture ;
addto p doublepath ((0,0)--(3mm,3mm)) shifted (6mm,6mm) ;
draw (0,0)--(10cm,0) dashed p withpen pencircle scaled 1mm ;
```

This way of defining a pattern is not that handy, especially if you start wondering why you need to supply a slanted path. Therefore, METAPOST provides a more convenient mechanism to define a pattern.

```
picture p ; p := dashpattern(on 3mm off 3mm) ;
draw (0,0)--(10cm,0) dashed p withpen pencircle scaled 1mm ;
```

Most dashpatterns can be defined in terms of on and off. This simple on–off dashpattern is predefined as picture `evenly`. Because this is a picture, you can (and often need to) scale it.

```
draw (0,0)--(10cm,0) dashed (evenly scaled 1mm)
  withpen pencircle scaled 1mm ;
```



Opposite to a defaultpen, there is no default color and default dash pattern set. The macro `drawoptions` provides you a way to set the default attributes.

```
drawoptions(dashed evenly withcolor red) ;
```

1.15 Text

Since METAFONT is meant for designing fonts, the only means for including text are those that permit you to add labels to positions for the sole purpose of documentation.

Because METAPOST is derived from METAFONT it provides labels too, but in order to let users add more sophisticated text, like a math formula, to a graphic, it also provides an interface to TEX.

Because we will spend a whole chapter on using text in METAPOST we limit the discussion here to a few fundamentals.

```
pair a ; a := (3cm,3cm) ;
label.top("top",a) ; label.bot("bot",a) ;
label.lft("lft",a) ; label.rt ("rt" ,a) ;
```

These four labels show up at the position stored in the pair variable a, anchored in the way specified after the period.

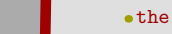
The command dotlabel also typesets the point as rather visible dot.

```
pair a ; a := (3cm,3cm) ;
dotlabel.top("top",a) ; dotlabel.bot("bot",a) ;
dotlabel.lft("lft",a) ; dotlabel.rt ("rt" ,a) ;
```

The command thelabel returns the typeset label as picture that you can manipulate or draw afterwards.

```
pair a ; a := (3cm,3cm) ; pickup pencircle scaled 1mm ;
drawdot a withcolor .625yellow ;
draw thelabel.rt("the right way",a) withcolor .625red ;
```

You can of course rotate, slant and manipulate such a label picture like any other picture.



The font can be specified in the string `defaultfont` and the scale in `defaultscale`. Labels are defined using the low level operator `infont`. The next statement returns a picture:

```
draw "this string will become a sequence of glyphs (MP)"
  infont defaultfont scaled defaultscale ;
```

By default the `infont` operator is not that clever and does not apply kerning. Also, typesetting math or accented characters are not supported. The way out of this problem is using `btex ... etex`.

```
draw btex this string will become a sequence of glyphs (\TeX) etex ;
```

The difference between those two methods is shown below. The outcome of `infont` depends on the current setting of the variable `defaultfont`.

this string will become a sequence of glyphs (MP)
this string will become a sequence of glyphs (TEX)

When you run inside CONTEXT (as we do here) there is no difference between `infont` and the TEX methods. This is because we overload the `infont` operator and also pass its content to TEX. Both `infont` and `btex` use the macro `textext` which is intercepted and redirects the task to TEX. This happens in the current run so there is no need to pass extra information about fonts.

Instead of passing strings to `infont`, you can also pass characters, using `char`, for example `char(73)`. When you use `infont` you normally expect the font to be ASCII conforming. If this is not the case, you must make sure that the encoding of the font that you use matches your expectations. However, as we overload this macro it does not really matter since the string is passed to TEX anyway. For instance, UTF encoded text should work fine as CONTEXT itself understands this encoding.

1.16 Linear equations

In the previous sections, we used the assignment operator `:=` to assign a value to a variable. Although for most of the graphics that we will present in later chapters, an assignment is appropriate, specifying a graphic in terms of expressions is not only more flexible, but also more in the spirit of the designers of METAFONT and METAPOST.

The METAFONT book and METAPOST manual provide lots of examples, some of which involve math that we don't consider to belong to everyones repertoire. But, even for non mathematicians using expressions can be a rewarding challenge.

The next introduction to linear equations is based on my first experiences with METAPOST and involves a mathematical challenge posed by a friend. I quickly ascertained that a graphical proof was far more easy than some proof with a lot of $\sin(this)$ and $\cos(that)$ and long forgotten formulas.

I was expected to prove that the lines connecting the centers of four squares drawn upon the four sides of a quadrilateral were perpendicular (see **figure 1.2**).

This graphic was generated with the following command:

```
\placefigure
  [here][fig:problem]
```

Figure 1.2 The problem.

```
{The problem.}
{\scale
   [width=\textwidth]
   {\useMPgraphic{solvers::one}{i=0.6,j=1.0,s=1}}}
```

We will use this example to introduce a few new concepts, one being instances. In a large document there can be many METAPOST graphics and they might fall in different categories. In this manual we have graphics that are generated as part of the style as wel as examples that show what METAFUN can do. As definitions and variables in METAPOST are global by default, there is a possibility that we end up with clashes. This can be avoided by grouping graphics in instances. Here we create an instance for the example that we're about to show.

```
\defineMPinstance
  [solvers]
  [format=metafun,
   extensions=yes,
   initializations=yes]
```

We can now limit the scope of definitions to this specific instance. Let's start with the macro that takes care of drawing the solution to our problem. The macro accepts four pairs of coordinates that determine the central quadrilateral. All of them are expressions.

```
\startMPdefinitions{solvers}
def draw_problem (expr p, q, r, s, show_labels) =
  begingroup ; save x, y, a, b, c, d, e, f, g, h ;

  z11 = z42 = p ; z21 = z12 = q ; z31 = z22 = r ; z41 = z32 = s ;

  a = x12 - x11 ; b = y12 - y11 ; c = x22 - x21 ; d = y22 - y21 ;
  e = x32 - x31 ; f = y32 - y31 ; g = x42 - x41 ; h = y42 - y41 ;

  z11 = (x11,   y11)   ; z12 = (x12,   y12)   ;
  z13 = (x12-b, y12+a) ; z14 = (x11-b, y11+a) ;
  z21 = (x21,   y21)   ; z22 = (x22,   y22)   ;
  z23 = (x22-d, y22+c) ; z24 = (x21-d, y21+c) ;
  z31 = (x31,   y31)   ; z32 = (x32,   y32)   ;
  z33 = (x32-f, y32+e) ; z34 = (x31-f, y31+e) ;
  z41 = (x41,   y41)   ; z42 = (x42,   y42)   ;
  z43 = (x42-h, y42+g) ; z44 = (x41-h, y41+g) ;

  pickup pencircle scaled .5pt ;

  draw z11--z12--z13--z14--cycle ; draw z11--z13 ; draw z12--z14 ;
  draw z21--z22--z23--z24--cycle ; draw z21--z23 ; draw z22--z24 ;
  draw z31--z32--z33--z34--cycle ; draw z31--z33 ; draw z32--z34 ;
  draw z41--z42--z43--z44--cycle ; draw z41--z43 ; draw z42--z44 ;
```

```
  z1 = 0.5[z11,z13] ; z2 = 0.5[z21,z23] ;
  z3 = 0.5[z31,z33] ; z4 = 0.5[z41,z43] ;

  draw z1--z3 dashed evenly ; draw z2--z4 dashed evenly ;

  z0 = whatever[z1,z3] = whatever[z2,z4] ;
  mark_rt_angle (z1, z0, z2) ; % z2 is not used at all

  if show_labels > 0 :
    draw_problem_labels ;
  fi ;

  endgroup ;
enddef ;
\stopMPdefinitions
```

Because we want to call this macro more than once, we first have to save the locally used values. Instead of declaring local variables, one can hide their use from the outside world. In most cases variables behave globally. If we don't save them, subsequent calls will lead to errors due to conflicting equations. We can omit the grouping commands, because we wrap the graphic in a figure, and figures are grouped already.

We will use the predefined z variable, or actually a macro that returns a variable. This variable has two components, an x and y coordinate. So, we don't save z, but the related variables x and y.

Next we draw four squares and instead of hard coding their corner points, we use METAPOST's equation solver. Watch the use of = which means that we just state dependencies. In languages like PERL, the equal sign is used in assignments, but in METAPOST it is used to express relations.

In a first version, we will just name a lot of simple relations, as we can read them from a sketch drawn on paper. So, we end up with quite some z related expressions.

For those interested in the mathematics behind this code, we add a short explanation. Absolutely key to the construction is the fact that you traverse the original quadrilateral in a clockwise orientation. What is really going on here is vector geometry. You calculate the vector from z_{11} to z_{12} (the first side of the original quadrilateral) with:

```
(a,b) = z12 - z11 ;
```

This gives a vector that points from z_{11} to z_{12}. Now, how about an image that shows that the vector $(-b,a)$ is a 90 degree rotation in the counterclockwise direction. Thus, the points z_{13} and z_{14} are easily calculated with vector addition.

```
z13 = z12 + (-b,a) ;
z14 = z11 + (-b,a) ;
```

This pattern continues as you move around the original quadrilateral in a clockwise manner.[3]

The code that calculates the pairs a through h, can be written in a more compact way.

```
(a,b) = z12 - z11 ; (c,d) = z22 - z21 ;
(e,f) = z32 - z31 ; (g,h) = z42 - z41 ;
```

The centers of each square can also be calculated by METAPOST. The next lines define that those points are positioned halfway the extremes.

```
z1 = 0.5[z11,z13] ; z2 = 0.5[z21,z23] ;
z3 = 0.5[z31,z33] ; z4 = 0.5[z41,z43] ;
```

[3] Thanks to David Arnold for this bonus explanation.

Once we have defined the relations we can let METAPOST solve the equations. This is triggered when a variable is needed, for instance when we draw the squares and their diagonals. We connect the centers of the squares using a dashed line style.

Just to be complete, we add a symbol that marks the right angle. First we determine the common point of the two lines, that lays at *whatever* point METAPOST finds suitable.

The definition of `mark_rt_angle` is copied from the METAPOST manual and shows how compact a definition can be (see **page 23** for an introduction to `zscaled`).

```
\startMPdefinitions{solvers}
angle_radius := 10pt ;

def mark_rt_angle (expr a, b, c) =
  draw ((1,0)--(1,1)--(0,1))
    zscaled (angle_radius*unitvector(a-b))
    shifted b
enddef ;
\stopMPdefinitions
```

So far, most equations are rather simple, and in order to solve them, METAPOST did not have to work real hard. The only boundary condition is that in order to find a solution, METAPOST must be able to solve all dependencies.

The actual value of the `whatever` variable is that it saves us from introducing a slew of variables that will never be used again. We could write:

```
z0 = A[z1,z3] = B[z2,z4] ;
```

and get the same result, but the `whatever` variable saves us the trouble of introducing intermediate variables for which we have no use once the calculation is finished.

The macro `mark_rt_angle` draws the angle symbol and later we will see how it is defined. First we draw the labels. Unfortunately we cannot package `btex ... etex` into a macro, because it is processed in a rather special way. Each `btex ... etex` occurance is filtered from the source and converted into a snippet of TEX code. When passed through TEX, each snippet becomes a page, and an auxiliary program converts each page into a METAPOST picture definition, which is loaded by METAPOST. The limitation lays in the fact that the filtering is done independent from the METAPOST run, which means that loops (and other code) are not seen at all. Later we will introduce the METAFUN way around this.

In order to get all the labels typeset, we have to put a lot of code here. The macro `dotlabel` draws a dot and places the typeset label.

```
\startMPdefinitions{solvers}
def draw_problem_labels =
    pickup pencircle scaled 5pt ;

    dotlabel.llft("$Z_{11}$", z11) ; dotlabel.ulft("$Z_{12}$", z12) ;
    dotlabel.ulft("$Z_{13}$", z13) ; dotlabel.llft("$Z_{14}$", z14) ;

    dotlabel.lrt ("$Z_{21}$", z21) ; dotlabel.llft("$Z_{22}$", z22) ;
    dotlabel.urt ("$Z_{23}$", z23) ; dotlabel.ulft("$Z_{24}$", z24) ;

    dotlabel.urt ("$Z_{31}$", z31) ; dotlabel.ulft("$Z_{32}$", z32) ;
    dotlabel.urt ("$Z_{33}$", z33) ; dotlabel.urt ("$Z_{34}$", z34) ;

    dotlabel.lrt ("$Z_{41}$", z41) ; dotlabel.urt ("$Z_{42}$", z42) ;
    dotlabel.llft("$Z_{43}$", z43) ; dotlabel.lrt ("$Z_{44}$", z44) ;
```

```
    dotlabel.urt ("$Z_{0}$", z0) ;
    dotlabel.lft ("$Z_{1}$", z1) ; dotlabel.top ("$Z_{2}$", z2) ;
    dotlabel.rt  ("$Z_{3}$", z3) ; dotlabel.bot ("$Z_{4}$", z4) ;
enddef ;
\stopMPdefinitions
```

Watch out: as we are in CONTEXT, we can pass regular TEX code to the label macro. In a standalone METAPOST run you'd have to use the `btex` variant.

We are going to draw a lot of pictures, so we define an extra macro. This time we hard–code some values. The fractions `i` and `j` are responsible for the visual iteration process, while `s` determines the labels. We pass these variables to the graphic using an extra argument. When you define the (useable) graphic you need to tell what variables it can expect.

```
\startuseMPgraphic{one}{i,j,s}
  draw_problem (
              (400pt,400pt), (300pt,600pt),
    \MPvar{i}[(300pt,600pt), (550pt,800pt)],
    \MPvar{j}[(400pt,400pt), (550pt,500pt)],
    \MPvar{s}
 ) ;
\stopuseMPgraphic
```

Of course we could have used a loop construct here, but defining auxiliary macros probably takes more time than simply calling the drawing macro directly. The results are shown on a separate page (**figure 1.3**).

We will use a helper macro (that saves us typing):

```
\def\MyTest#1#2%
```

```
{\scale
   [width=.25\textwidth]
   {\useMPgraphic{solvers::one}{i=#1,j=#2,s=0}}}
```

We now can say:

```
\startcombination[3*4]
  {\MyTest{1.0}{1.0}} {1.0 / 1.0} {\MyTest{0.8}{1.0}} {0.8 / 1.0}
  {\MyTest{0.6}{1.0}} {0.6 / 1.0} {\MyTest{0.4}{1.0}} {0.4 / 1.0}
  {\MyTest{0.2}{1.0}} {0.2 / 1.0} {\MyTest{0.0}{1.0}} {0.0 / 1.0}
  {\MyTest{0.0}{1.0}} {0.0 / 1.0} {\MyTest{0.0}{0.8}} {0.0 / 0.8}
  {\MyTest{0.0}{0.6}} {0.0 / 0.6} {\MyTest{0.0}{0.4}} {0.0 / 0.4}
  {\MyTest{0.0}{0.2}} {0.0 / 0.2} {\MyTest{0.0}{0.0}} {0.0 / 0.0}
\stopcombination
```

Watch how we pass the settings to the graphic definition using an extra argument. We force using the `solvers` instance by prefixing the name.

It does not need that much imagination to see the four sided problem converge to a three sided one, which itself converges to a two sided one. In the two sided alternative it's not that hard to prove that the angle is indeed 90 degrees.

As soon as you can see a clear pattern in some code, it's time to consider using loops. In the previous code, we used semi indexes, like `12` in `z12`. In this case `12` does reflect something related to square 1 and 2, but in reality the 12 is just twelve. This does not harm our expressions.

A different approach is to use a two dimensional array. In doing so, we can access the variables more easily using loops. If we omit the labels, and angle macro, the previously defined macro can be reduced considerably.

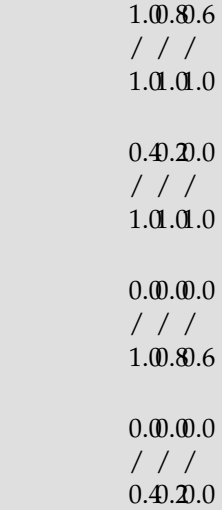


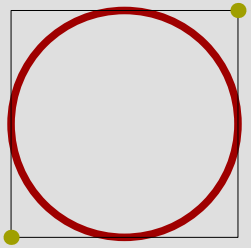
Figure 1.3 The solution.

```
def draw_problem (expr n, p, q, r, s) = % number and 4 positions
  begingroup ; save x, y ;

  z[1][1] = p ; z[2][1] = q ; z[3][1] = r ; z[4][1] = s ;

  for i=1 upto 4 :
    z[i][1] = (x[i][1],y[i][1]) = z[if i=1: 4 else: i-1 fi][2] ;
    z[i][2] = (x[i][2],y[i][2]) ;
    z[i][3] = (x[i][2]-y[i][2]+y[i][1], y[i][2]+x[i][2]-x[i][1]) ;
    z[i][4] = (x[i][1]-y[i][2]+y[i][1], y[i][1]+x[i][2]-x[i][1]) ;
    z[i] = 0.5[z[i][1],z[i][3]] ;
  endfor ;

  z[0] = whatever[z[1],z[3]] = whatever[z[2],z[4]] ;

  pickup pencircle scaled .5pt ;

  for i=1 upto 4 :
    draw z[i][1]--z[i][2]--z[i][3]--z[i][4]--cycle ;
    draw z[i][1]--z[i][3] ; draw z[i][2]--z[i][4] ;
    if i<3 : draw z[i]--z[i+2] dashed evenly fi ;
  endfor ;

  draw ((1,0)--(1,1)--(0,1))
    zscaled (unitvector(z[1]-z[0])*10pt)
    shifted z[0] ;

  endgroup ;
enddef ;
```

I think that we could argue quite some time about the readability of this code. If you start from a sketch, and the series of equations does a good job, there is hardly any need for such improvements to the code. On the other hand, there are situations where the simplified (reduced) case can be extended more easily, for instance to handle 10 points instead of 4. It all depends on how you want to spend your free hours.

1.17 Clipping

For applications that do something with a drawing, for instance TEX embedding a graphic in a text flow, it is important to know the dimensions of the graphic. The maximum dimensions of a graphic are specified by its bounding box.



A bounding box is defined by its lower left and upper right corners. If you open the POSTSCRIPT file produced by METAPOST, you may find lines like:

```
%%BoundingBox: -46 -46 46 46
```

or, when supported,

```
%%HiResBoundingBox: -45.35432 -45.35432 45.35432 45.35432
```

The first two numbers define the lower left corner and the last two numbers the upper right corner. From these values, you can calculate the width and height of the graphic.

A graphic may extend beyond its bounding box. It depends on the application that uses the graphic whether that part of the graphic is shown.

In METAPOST you can ask for all four points of the bounding box of a path or picture as well as the center.

`llcorner p`	lower left corner
`lrcorner p`	lower right corner
`urcorner p`	upper right corner
`ulcorner p`	upper left corner
`center p`	the center point

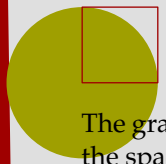
You can construct the bounding box of path `p` out of the four points mentioned:

```
llcorner p -- lrcorner p -- urcorner p -- ulcorner p -- cycle
```

You can set the bounding box of a picture, which can be handy if you want to build a picture in steps and show the intermediate results using the same dimensions as the final picture, or when you want to show only a small piece.

```
fill fullcircle scaled 2cm withcolor .625yellow ;
setbounds currentpicture to unitsquare scaled 1cm ;
draw unitsquare scaled 1cm withcolor .625red ;
```

Here, we set the bounding box with the command `setbounds`, which takes a path.



The graphic extends beyond the bounding box, but the bounding box determines the placement and therefore the spacing around the graphic. We can get rid of the artwork outside the bounding box by clipping it.

```
fill fullcircle scaled 2cm withcolor .625yellow ;
clip currentpicture to unitsquare scaled 1cm ;
```

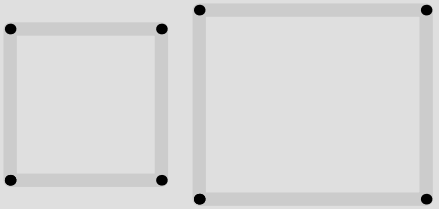
The resulting picture is just as large but shows less of the picture.

1.18 Some extensions

We will now encounter a couple of transformations that can make your life easy when you use METAPOST for making graphics like the ones demonstrated in this document. These transformations are not part of standard METAPOST, but come with METAFUN.

A very handy extension is `enlarged`. Although you can feed it with any path, it will return a rectangle larger or smaller than the boundingbox of that path. You can specify a pair or a numeric.

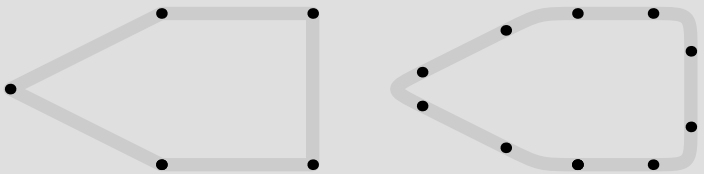
```
path p ; p := fullsquare scaled 2cm ;
drawpath p ; drawpoints p ;
p := (p shifted (3cm,0)) enlarged (.5cm,.25cm) ;
drawpath p ; drawpoints p ;
```



There are a few more alternatives, like `bottomenlarged`, `rightenlarged`, `topenlarged` and `leftenlarged`.

The `cornered` operator will replace sharp corners by rounded ones (we could not use `rounded` because this is already in use).

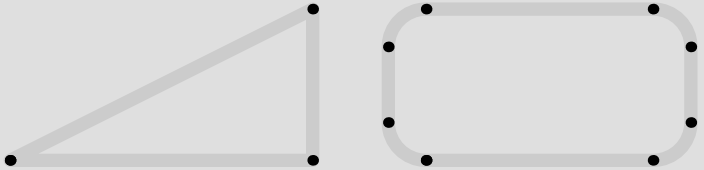
```
path p ; p := ((1,0)--(2,0)--(2,2)--(1,2)--(0,1)--cycle)
  xysized (4cm,2cm) ;
drawpath p ; drawpoints p ;
p := (p shifted (5cm,0)) cornered .5cm ;
drawpath p ; drawpoints p ;
```



The `smoothed` operation is a less subtle one, since it operates on the bounding box and thereby can result in a different shape.

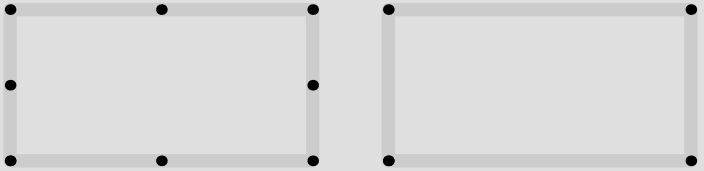
```
path p ; p := ((1,0)--(2,0)--(2,2)--cycle) xysized (4cm,2cm) ;
```

```
drawpath p ; drawpoints p ;
p := (p shifted (5cm,0)) smoothed .5cm ;
drawpath p ; drawpoints p ;
```



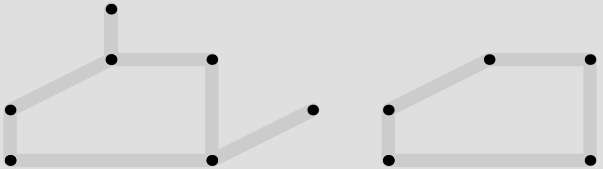
The next one, `simplified`, can be applied to paths that are constructed automatically. Instead of testing for duplicate points during construction, you can clean up the path afterwards.

```
path p ; p :=
  ((0,0)--(1,0)--(2,0)--(2,1)--(2,2)--(1,2)--(0,2)--(0,1)--cycle)
  xysized (4cm,2cm) ;
drawpath p ; drawpoints p ;
p := simplified (p shifted (5cm,0)) ;
drawpath p ; drawpoints p ;
```



A cousin of the previous operation is `unspiked`. This one removes ugly left overs. It works well for the average case.

```
path p ; p :=
  ((0,0)--(2,0)--(3,1)--(2,0)--(2,2)--(1,2)--(1,3)--(1,2)--(0,1)--cycle)
  xysized (4cm,2cm) ;
drawpath p ; drawpoints p ;
p := unspiked (p shifted (5cm,0)) ;
drawpath p ; drawpoints p ;
```

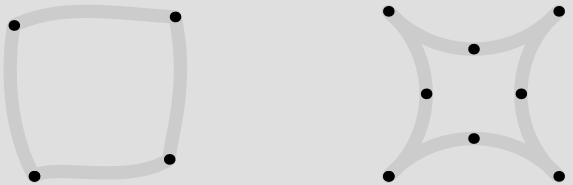


There are a couple of operations that manipulate the path in more drastic ways. Take `randomized`.

```
path p ; p := fullsquare scaled 2cm ;
drawpath p ; drawpoints p ;
p := (p shifted (5cm,0)) randomized .5cm ;
drawpath p ; drawpoints p ;
```

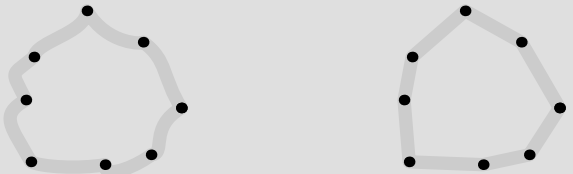
Or how about `squeezed`:

```
path p ; p := fullsquare scaled 2cm randomized .5cm ;
drawpath p ; drawpoints p ;
p := (p shifted (5cm,0)) squeezed .5cm ;
drawpath p ; drawpoints p ;
```



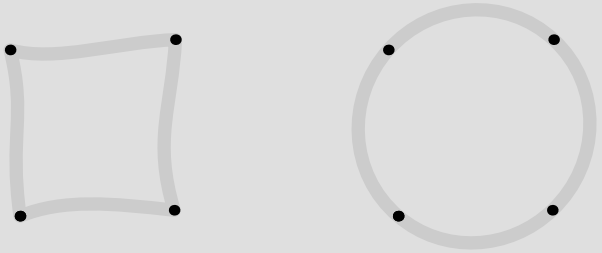
A punked path is, like a punked font, a font with less smooth curves (in our case, only straight lines).

```
path p ; p := fullcircle scaled 2cm randomized .5cm ;
drawpath p ; drawpoints p ;
p := punked (p shifted (5cm,0)) ;
drawpath p ; drawpoints p ;
```



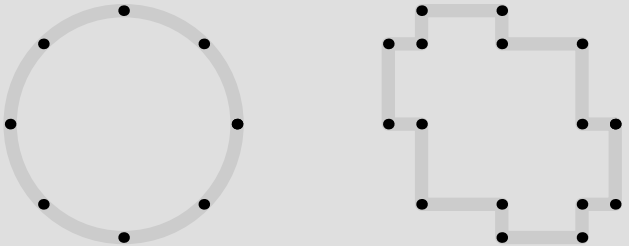
A curved path on the other hand has smooth connections. Where in many cases a punked path becomes smaller, a curved path will be larger.

```
path p ; p := fullsquare scaled 2cm randomized .5cm ;
drawpath p ; drawpoints p ;
p := curved (p shifted (5cm,0)) ;
drawpath p ; drawpoints p ;
```



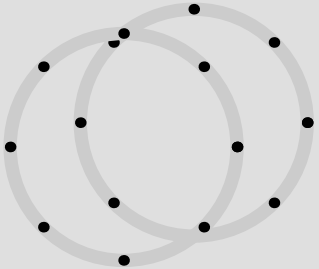
Probably less usefull (although we use it in one of the OPENTYPE visualizers) is `laddered`:

```
path p ; p := fullcircle scaled 3cm ;
drawpath p ; drawpoints p ;
p := laddered (p shifted (5cm,0)) ;
drawpath p ; drawpoints p ;
```



When writing PPCHTEX (that can be used to draw chemical structure formulas) I needed a parallelizing macro, so here it is:

```
path p ; p := fullcircle scaled 3cm ;
drawpath p ; drawpoints p ;
p := p paralleled 1cm ;
drawpath p ; drawpoints p ;
```



If you use a negative argument (like `-1cm`) the parallel line will be drawn at the other side.

The `blownup` operation scales the path but keeps the center in the same place.

```
path p ; p := fullsquare xyscaled (4cm,1cm) randomized .5cm ;
drawpath p ; drawpoints p ;
p := p blownup .5cm ;
drawpath p ; drawpoints p ;
```

The `shortened` operation also scales the path but only makes it longer or shorter. This macro only works on straight paths.

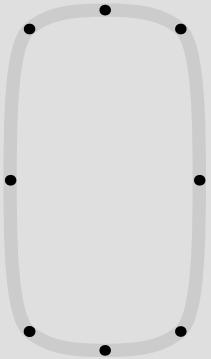
```
path p ; p := (0,0) -- (2cm,3cm) ;
drawpath p ; drawpoints p ;
p := p shortened 1cm ;
drawpath p ; drawpoints p ;
p := p shortened -1cm ;
drawpath p ; drawpoints p ;
```

Here are a few more drawing helpers. Even if you don't need them you might at some point take a look at their definitions to see what happens there. First we give a square round corners with `roundedsquare`:

```
path p ; p := roundedsquare(2cm,4cm,.25cm) ;
drawpath p ; drawpoints p ;
```

Next we draw a square-like circle (or circle-like square) using `tensecircle`:

```
path p ; p := tensecircle(2cm,4cm,.25cm) ;
drawpath p ; drawpoints p ;
```

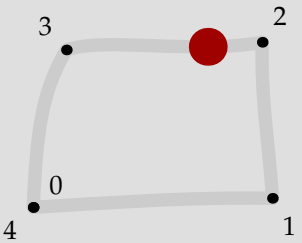


Often I make such helpers in the process of writing larger drawing systems. Take `crossed`:

```
path p ; p := origin crossed 1cm ;
drawpath p ; drawpoints p ;
p := (origin crossed fullcircle scaled 2cm crossed .5cm) shifted (3cm,0) ;
drawpath p ; drawpoints p ;
```

These examples demonstrate that a path is made up out of points (something that you probably already knew by now). The METAPOST operator `of` can be used to 'access' a certain point at a curve.

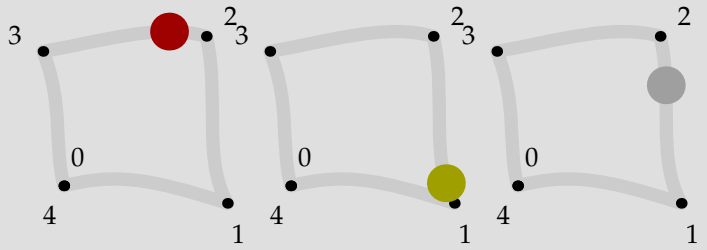
```
path p ; p := fullsquare xyscaled (3cm,2cm) randomized .5cm ;
drawpath p ; drawpoints p ; drawpointlabels p ;
draw point 2.25 of p withpen pencircle scaled 5mm withcolor .625red ;
```



To this we add two more operators: `on` and `along`. With `on` you get the point at the supplied distance from point 0; with `along` you get the point at the fraction of the length of the path.

```
path p, q, r ;
p := fullsquare xyscaled (2cm,2cm) randomized .5cm ;
q := p shifted (3cm,0) ; r := q shifted (3cm,0) ;
drawpath p ; drawpoints p ; drawpointlabels p ;
drawpath q ; drawpoints q ; drawpointlabels q ;
drawpath r ; drawpoints r ; drawpointlabels r ;
pickup pencircle scaled 5mm ;
draw point 2.25   of    p withcolor .625red ;
draw point 2.50cm on    q withcolor .625yellow ;
draw point  .45   along r withcolor .625white ;
```

Beware: the `length` of a path is the number of points minus one. The shapes below are constructed from 5 points and a length of 4. If you want the length as dimension, you should use `arclength`.



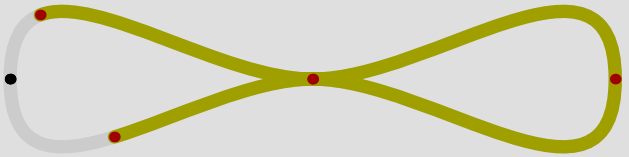
We will now play a bit with simple lines. With cutends, you can (indeed) cut off the ends of a curve. The specification is a dimension.

```
path p ; p := (0cm,0cm)        -- (4cm,1cm) ;
path q ; q := (5cm,0cm){right} .. (9cm,1cm) ;
drawpath p ; drawpoints p ; drawpath q ; drawpoints q ;
p := p cutends .5cm ; q := q cutends .5cm ;
drawpathoptions (withpen pencircle scaled 5pt withcolor .625yellow) ;
drawpointoptions(withpen pencircle scaled 4pt withcolor .625red) ;
drawpath p ; drawpoints p ; drawpath q ; drawpoints q ;
resetdrawoptions ;
```

As with more operators, cutends accepts a numeric or a pair. Watch the subtle difference between the next and the previous use of cutends.

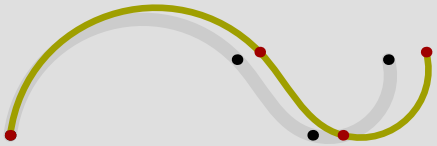
```
path p ; p := (0cm,0) .. (4cm,0) .. (8cm,0) .. (4cm,0) .. cycle ;
```

```
drawpath p ; drawpoints p ; p := p cutends (2cm,1cm) ;
drawpathoptions (withpen pencircle scaled 5pt withcolor .625yellow) ;
drawpointoptions(withpen pencircle scaled 4pt withcolor .625red) ;
drawpath p ; drawpoints p ;
resetdrawoptions ;
```



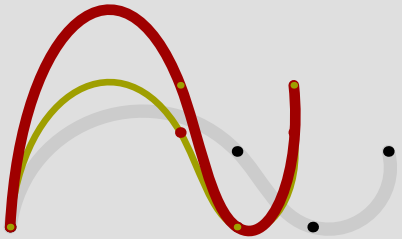
When `stretched` is applied to a path, it is scaled but the starting point (point 0) keeps its location. The specification is a scale.

```
path p ; p := (0cm,0) .. (3cm,1cm) .. (4cm,0) .. (5cm,1cm) ;
drawpath p ; drawpoints p ; p := p stretched 1.1 ;
drawpathoptions (withpen pencircle scaled 2.5pt withcolor .625yellow) ;
drawpointoptions(withpen pencircle scaled 4.0pt withcolor .625red) ;
drawpath p ; drawpoints p ; resetdrawoptions ;
```



We can scale in two directions independently or even in one direction by providing a zero value. In the next example we apply the stretch two times.

```
path p ; p := (0cm,0) .. (3cm,1cm) .. (4cm,0) .. (5cm,1cm) ;
drawpath p ; drawpoints p ; p := p stretched (.75,1.25) ;
drawpathoptions (withpen pencircle scaled 2.5pt withcolor .625yellow) ;
drawpointoptions(withpen pencircle scaled 4.0pt withcolor .625red) ;
drawpath p ; drawpoints p ; p := p stretched (0,1.5) ;
drawpathoptions (withpen pencircle scaled 4.0pt withcolor .625red) ;
drawpointoptions(withpen pencircle scaled 2.5pt withcolor .625yellow) ;
drawpath p ; drawpoints p ; resetdrawoptions ;
```



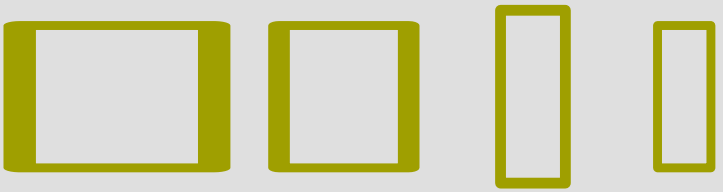
We already met the `randomize` operator. This one is the chameleon under the operators.

```
draw fullsquare xyscaled (4cm,2cm)
  randomized .25cm
  shifted origin randomized (1cm, 2cm)
  withcolor red randomized (.625, .850)
  withpen pencircle scaled (5pt randomized 1pt) ;
```

So, `randomized` can handle a numeric, pair, path and color, and its specification can be a numeric, pair or color, depending on what we're dealing with.

In the previous example we also see `xyscaled` in action. Opposite to `scaled`, `xscaled` and `yscaled`, this is not one of METAPOST build in features. The same is true for the `.sized` operators.

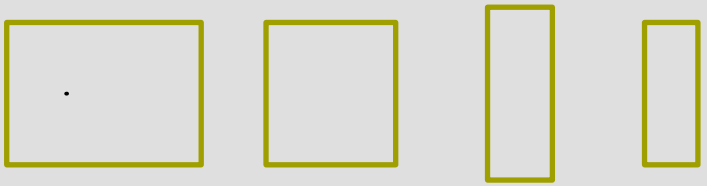
```
picture p ; p := image
  ( draw fullsquare
      xyscaled (300,800)
      withpen pencircle scaled 50
      withcolor .625 yellow ; ) ;
draw p xysized (3cm,2cm) shifted (bbwidth(currentpicture)+.5cm,0) ;
draw p xysized  2cm      shifted (bbwidth(currentpicture)+.5cm,0) ;
draw p xsized   1cm      shifted (bbwidth(currentpicture)+.5cm,0) ;
draw p ysized   2cm      shifted (bbwidth(currentpicture)+.5cm,0) ;
```



Here, the `image` macro creates an (actually rather large) picture. The last four lines actually draw this picture, but at the given dimensions. Watch how the line width scales accordingly. If you don't want this, you can add the following line:

```
redraw currentpicture withpen pencircle scaled 2pt ;
draw boundingbox currenpicture withpen pencircle scaled .5mm ;
```

Watch how the boundingbox is not affected:



In this example we also used `bbwidth` (which has a companion macro `bbheight`). You can apply this macro to a path or a picture.

In fact you don't always need to follow this complex route if you want to simply redraw a path with another pen or color.

```
draw fullcircle scaled 1cm
  withcolor .625red    withpen pencircle scaled 1mm ;
draw currentpicture
  withcolor .625yellow withpen pencircle scaled 3mm ;
draw boundingbox currentpicture
  withpen pencircle scaled .5mm ;
```

This is what you will get from this:

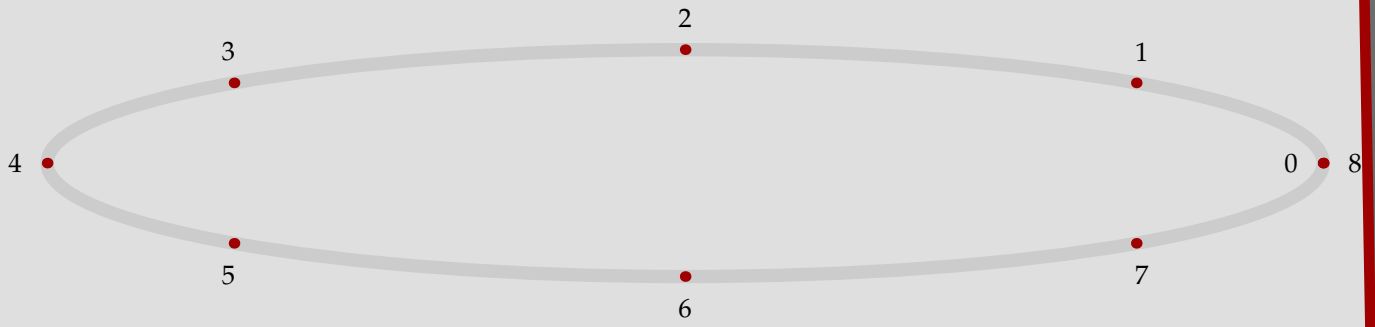
If you want to add a background color to a picture you can do that afterwards. This can be handy when you don't know in advance what size the picture will have.

```
fill fullcircle scaled 1cm withcolor .625red ;
addbackground withcolor .625 yellow ;
```

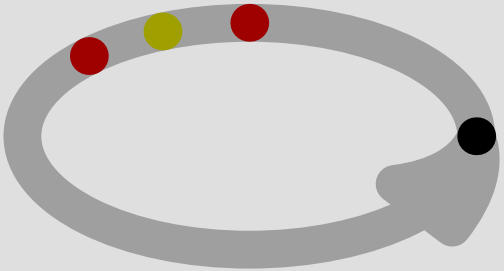
The background is just a filled rectangle that gets the same size as the current picture, that is put on top of it.

1.19 Cutting and pasting

When enhancing or building a graphic, often parts of already constructed paths are needed. The `subpath`, `cutbefore` and `cutafter` operators can be used to split paths in smaller pieces. In order to do so, we must know where we are on the path that is involved. For this we use points on the path. Unfortunately we can only use these points when we know where they are located. In this section we will combine some techniques discussed in previous sections. We will define a few macros, manipulate some paths and draw curves and points.



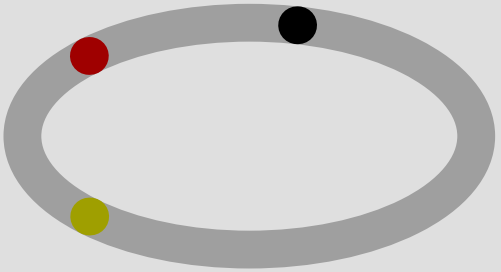
This circle is drawn by scaling the predefined path `fullcircle`. This path is constructed using 8 points. As you can see, these points are not distributed equally along the path. In the following graphic, the second and third point of the curve are colored red, and point 2.5 is colored yellow. Point 0 is marked in black. This point is positioned halfway between point 2 and 3.



It is clear that, unless you know exactly how the path is constructed, other methods should be available. A specific point on a path is accessed by `point ... of`, but the next example demonstrates two more alternatives.

```
path p ; p := fullcircle scaled 3cm xscaled 2 ;
pickup pencircle scaled 5mm ;
draw                 p withcolor .625white ;
draw point 3   of    p withcolor .625red ;
draw point .6  along p withcolor .625yellow ;
draw point 3cm on    p ;
```

So, in addition to `on` to specify a point by number (in METAPOST terminology called time), we have `along` to specify a point as fraction of the path, and `on` to specify the position in a dimension.



The `on` and `along` operators are macros and can be defined as:

```
primarydef len on pat =
  (arctime len of pat) of pat
enddef ;

primarydef pct along pat =
  (arctime (pct * (arclength pat)) of pat) of pat
enddef ;
```

These macros introduce two new primitives, `arctime` and `arclength`. While `arctime` returns a number denoting the time of the point on the path, `arclength` returns a dimension.

"When mathematicians draw parametric curves, they frequently need to indicate the direction of motion. I often have need of a little macro that will put an arrow of requested length, anchored at a point on the curve, and bending with the curve in the direction of motion."

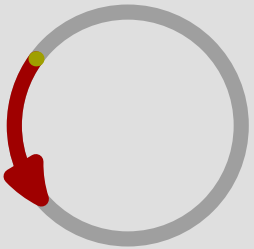
When David Arnold asked me how this could be achieved, the fact that a length was requested meant that the solution should be sought in using the primitives and macros we introduced a few paragraphs before. Say that we want to call for such an arrow as follows.

```
path p ; p := fullcircle scaled 3cm ;
pair q ; q := point .4 along p ;
pickup pencircle scaled 2mm ;
draw                p        withcolor .625white ;
drawarrow somearrow(p,q,2cm) withcolor .625red ;
draw                  q      withcolor .625yellow ;
```

Because we want to follow the path, we need to construct the arrow from this path. Therefore, we first reduce the path by cutting off the part before the given point. Next we cut off the end of the resulting path so that we keep a slice that has the length that was asked for. Since we can only cut at points, we determine this point using the `arctime` primitive.

```
vardef somearrow (expr pat, loc, len) =
  save p ; path p ; p := pat cutbefore loc ;
  (p cutafter point (arctime len of p) of p)
enddef ;
```

By using a `vardef` we hide the intermediate assignments. Such `vardef` is automatically surrounded by `begingroup` and `endgroup`, so the `save` is local to this macro. When processed, this code produces the following graphic:



This graphic shows that we need a bit more control over the exact position of the arrow. It would be nice if we could start the arrow at the point, or end there, or center the arrow around the point. Therefore, the real implementation is a bit more advanced.

```
vardef pointarrow (expr pat, loc, len, off) =
  save l, r, s, t ; path l, r ; numeric s ; pair t ;
  t := if pair loc : loc else : point loc along pat fi ;
  s := len/2 - off ; if s<=0 : s := 0 elseif s>len : s := len fi ;
  r := pat cutbefore t ;
  r := (r cutafter point (arctime s of r) of r) ;
  s := len/2 + off ; if s<=0 : s := 0 elseif s>len : s := len fi ;
  l := reverse (pat cutafter t) ;
  l := (reverse (l cutafter point (arctime s of l) of l)) ;
  (l..r)
enddef ;
```

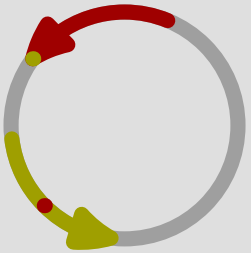
This code fragment also demonstrates how we can treat the `loc` argument as pair (coordinates) or fraction of the path. We calculate the piece of path before and after the given point separately and paste them afterwards as `(l..r)`. By adding braces we can manipulate the path in expressions without the danger of handling `r` alone.

We can now implement left, center and right arrows by providing this macro the right parameters. The offset (the fourth parameter), is responsible for a backward displacement. This may seem strange, but negative values would be even more confusing.

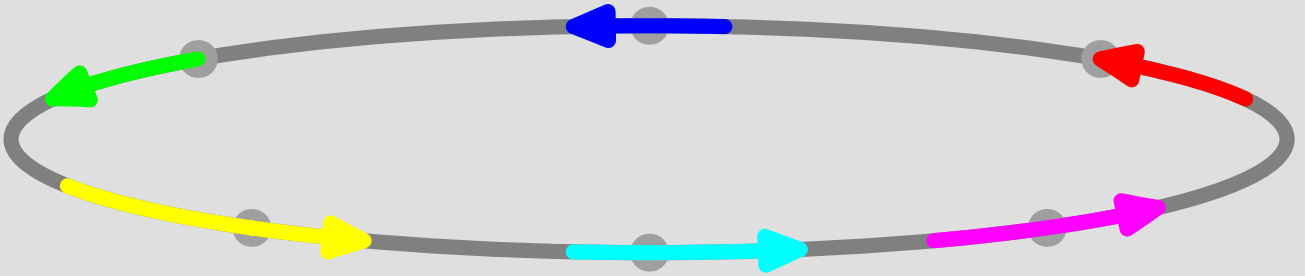
```
def rightarrow (expr p,t,l) = pointarrow(p,t,l,-l) enddef ;
def leftarrow  (expr p,t,l) = pointarrow(p,t,l,+l) enddef ;
def centerarrow(expr p,t,l) = pointarrow(p,t,l, 0) enddef ;
```

We can now apply this macro as follows:

```
path p ; p := fullcircle scaled 3cm ;
pickup pencircle scaled 2mm ;
draw p withcolor .625white ;
drawarrow leftarrow  (p,     .4     ,2cm) withcolor .625red ;
drawarrow centerarrow(p,point 5 of p,2cm) withcolor .625yellow ;
draw point .4 along p withcolor .625yellow ;
draw point  5 of    p withcolor .625red ;
```



Watch how we can pass a point (`point 5 of p`) as well as a fraction (`.4`). The following graphic demonstrates a few more alternatives.



The arrows are drawn using the previously defined macros. Watch the positive and negative offsets in call to `pointarrow`.

```
drawarrow leftarrow   (p,point 1 of p,2cm) withcolor red    ;
drawarrow centerarrow (p,point 2 of p,2cm) withcolor blue   ;
drawarrow rightarrow  (p,point 3 of p,2cm) withcolor green  ;
drawarrow pointarrow  (p,.60,4cm,+.5cm)    withcolor yellow ;
```

```
drawarrow pointarrow  (p,.75,3cm,-.5cm)    withcolor cyan    ;
drawarrow centerarrow (p,.90,3cm)          withcolor magenta ;
```

Current picture

When you draw paths, texts and/or pictures they are added to the so called current picture. You can manipulate this current picture as is demonstrated in this manual. Let's show a few current picture related tricks.

```
draw fullcircle scaled 1cm withpen pencircle scaled 1mm withcolor .625red ;
```

We can manipulate the picture as a whole:

```
draw fullcircle scaled 1cm withpen pencircle scaled 1mm withcolor .625red ;
currentpicture := currentpicture slanted .5 ;
```

Sometimes it's handy to temporarily set aside the current picture.

```
draw fullcircle scaled 1cm withpen pencircle scaled 1mm withcolor .625red ;
currentpicture := currentpicture slanted .5 ;
pushcurrentpicture ;
```

```
draw fullcircle scaled 1cm withpen pencircle scaled 1mm withcolor .625yellow ;
currentpicture := currentpicture slanted -.5 ;
popcurrentpicture ;
```

These are METAFUN commands but METAPOST itself comes with a variant, `image`, and you explicitly have to draw this picture (or otherwise add it to the currentpicture).

```
draw fullcircle scaled 1cm withpen pencircle scaled 1mm withcolor .625red ;
currentpicture := currentpicture slanted .5 ;
draw image (
  draw fullcircle scaled 1cm
      withpen pencircle scaled 1mm withcolor .625yellow ;
  currentpicture := currentpicture slanted -.5 ;
) ;
```

Each graphic starts fresh with an empty current picture. In METAFUN we make sure that we also reset some otherwise global variables, like color, pen and some line properties.

2 A few more details

In this chapter we will see how to define a METAPOST graphic, and how to include it in a document. Since the exact dimensions of graphics play an important role in the placement of a graphic, we will explore the way a bounding box is constructed.

We will also pay attention to the usage of units and the side effects of scaling and shifting, since they can contradict our expectations in unexpected ways. Furthermore we will explore a few obscure areas.

2.1 Making graphics

In this manual we will use METAPOST in a rather straightforward way, and we will try to avoid complicated math as much as possible. We will do a bit of drawing, clipping, and moving around. Occasionally we will see some more complicated manipulations.

When defined as stand–alone graphic, a METAPOST file looks like this:

```
% Let's draw a circle.

beginfig (7) ;
  draw fullcircle scaled 3cm withpen pencircle scaled 1cm ;
endfig ;

end .
```

The main structuring components in such a file are the `beginfig` and `endfig` macros. Like in a big story, the file has many sub–sentences, where each sub–sentence ends with a semi–colon. Although the `end` command

at the end of the file concludes the story, putting a period there is a finishing touch. Actually, after the `end` command you can put whatever text you wish, your comments, your grocery list, whatever. Comments in METAPOST, prefixed by a percent sign, as in `% Let's draw a circle`, are ignored by the interpreter, but useful reminders for the programmer.

If the file is saved as `yourfile.mp`, then the file is processed by METAPOST by issuing the following command:

```
mpost yourfile
```

after which you will have a graphic called `yourfile.7`, which contains a series of POSTSCRIPT commands. Because METAPOST does all the work, this file is efficient and compact. The number of distinct POSTSCRIPT operators used is limited, which has the advantage that we can postprocess this file rather easily. Alternatively METAPOST can generate SVG output. It does when you say

```
outputformat := "svg" ;
```

Here we will not go into details about this format. Even POSTSCRIPT is not covered in detail as we use METAPOST mostly in embedded form.

We can view this file in a POSTSCRIPT viewer like GHOSTVIEW or convert the graphic to PDF (using `mptopdf`) and view the result in a suitable PDF viewer like ACROBAT. Of course, you can embed such a file in a CONTEXT document, using a command like:

```
\externalfigure[yourfile.7]
```

We will go in more detail about embedding graphics in **chapter 3**.

If you have installed CONTEXT, somewhere on your system there resides a file `mp-tool.mp`. If you make a stand–alone graphic, it's best to put the following line at the top of your file:

```
input mp-tool ; % or input metafun ;
```

By loading this file, the resulting graphic will provide a high resolution bounding box, which enables more accurate placement. The file also sets the `prologues := 1` so that viewers like GHOSTVIEW can refresh the file when it is changed.

Next we will introduce some more METAPOST commands. From now on, we will omit the encapsulating `beginfig` and `endfig` macros. If you want to process these examples yourself, you should add those commands yourself, or if you use CONTEXT you don't need them at all.

```
pickup pencircle scaled .5cm ;
draw unitsquare xscaled 8cm yscaled 1cm withcolor .625white ;
draw origin withcolor .625yellow ;
pickup pencircle scaled 1pt ;
draw bbox currentpicture withcolor .625red ;
```

In this example we see a mixture of so called primitives as well as macros. A primitive is something hard coded, a built–in command, while a macro is a collection of such primitives, packaged in a way that they can be recalled easily. Where `scaled` is a primitive and `draw` a macro, `unitsquare` is a path variable, an abbreviation for:

```
unitsquare = (0,0) -- (1,0) -- (1,1) -- (0,1) -- cycle ;
```

The double dash (`--`) is also a macro, used to connect two points with a straight line segment. However, `cycle` is a primitive, which connects the last point of the unitsquare to the first on unitsquare's path. Path variables must first be declared, as in:

```
path unitsquare ;
```

A large collection of such macros is available when you launch METAPOST. Consult the METAPOST manual for details.

In the first line of our example, we set the drawing pen to `.5cm`. You can also specify such a dimension in other units, like points (`pt`). When no unit is provided, METAPOST will use a big point (`bp`) , the POSTSCRIPT approximation of a point.

The second line does just as it says: it draws a rectangle of certain dimensions in a certain color. In the third line we draw a colored dot at the origin of the coordinate system in which we are drawing. Finally, we set up a smaller pen and draw the bounding box of the current picture, using the variable `currentpicture`. Normally, all drawn shapes end up in this picture variable.

2.2 Bounding boxes

If you take a close look at the last picture in the previous section, you will notice that the bounding box is larger than the picture. This is one of the nasty side effects of METAPOST's `bbox` macro. This macro draws a box, but with a certain offset. The next example shows how we can manipulate this offset. Remember that in order to process the next examples, you should embed the code in `beginfig` and `endfig` macros. Also, in stand–alone graphics, don't forget to say `\input mp-tool` first.

```
pickup pencircle scaled .5cm ;
draw unitsquare xscaled 8cm yscaled 1cm withcolor .625white ;
path bb ; bboxmargin := 0pt ; bb := bbox currentpicture ;
```

```
draw bb withpen pencircle scaled 1pt withcolor .625red ;
draw origin withpen pencircle scaled 5pt withcolor .625yellow ;
```

In the third line we define a path variable. We assign the current bounding box to this variable, but first we set the offset to zero. The last line demonstrates how to draw such a path. Instead of setting the pen as we did in the first line, we pass the dimensions directly.

Where `draw` draws a path, the `fill` macro fills one. In order to be filled, a path should be closed, which is accomplished by the `cycle` primitive, as we saw in constructing the `unitsquare` path.

```
pickup pencircle scaled .5cm ;
fill unitsquare xscaled 8cm yscaled 1cm withcolor .625white ;
path bb ; bboxmargin := 0pt ; bb := bbox currentpicture ;
draw bb withpen pencircle scaled 1pt withcolor .625red ;
draw origin withpen pencircle scaled 5pt withcolor .625yellow ;
```

This example demonstrates that when we fill the path, the resulting graphic is smaller. Where `draw` follows the center of a path, `fill` stays inside the path.

A third alternative is the `filldraw` macro. From the previous examples, we would expect a bounding box that matches the one of the drawn path.

```
pickup pencircle scaled .5cm ;
filldraw unitsquare xscaled 8cm yscaled 1cm withcolor .625white ;
path bb ; bboxmargin := 0pt ; bb := bbox currentpicture ;
draw bb withpen pencircle scaled 1pt withcolor .625red ;
draw origin withpen pencircle scaled 5pt withcolor .625yellow ;
```

The resulting graphic has the bounding box of the fill. Note how the path, because it is stroked with a .5cm pen, extends beyond the border of the previous bounding box. The way this image shows up depends on the viewer (settings) you use to render the graphic. For example, in GHOSTVIEW, if you disable clipping to the bounding box, only the positive quadrant of the graphic is shown.[4]

From the previous examples, you can conclude that the following alternative results in a proper bounding box:

```
pickup pencircle scaled .5cm ;
path p ; p := unitsquare xscaled 8cm yscaled 1cm ;
fill p withcolor .625white ;
draw p withcolor .625white ;
path bb ; bboxmargin := 0pt ; bb := bbox currentpicture ;
```

[4] Old versions of METAPOST calculated the boundingbox differently for a `filldraw`: through the middle of the penpath.

```
draw bb withpen pencircle scaled 1pt withcolor .625red ;
draw origin withpen pencircle scaled 5pt withcolor .625yellow ;
```

The CONTEXT distribution comes with a set of METAPOST modules, one of which contains the `drawfill` macro, which provides the outer bounding box.[5] Next we demonstrate its use in another, more complicated example.

```
picture finalpicture ; finalpicture := nullpicture ;
numeric n ; n := 0 ; bboxmargin := 0pt ;
pickup pencircle scaled .5cm ;

def shape =
  unitsquare scaled 2cm withcolor .625white ;
  draw bbox currentpicture
    withpen pencircle scaled .5mm withcolor .625red ;
  addto finalpicture also currentpicture shifted(n*3cm,0) ;
  currentpicture := nullpicture ; n := n+1 ;
enddef ;

fill shape ; draw shape ; filldraw shape ; drawfill shape ;

currentpicture := finalpicture ;
```

[5] Starting from version 1.0 METAPOST calculates the boundingbox differently and the distinction between `drawfill` and `filldraw` is gone. We keep them around both for compatibility.

Here we introduce a macro definition, `shape`. In METAPOST, the start of a macro definition is indicated with the keyword `def`. Thereafter, you can insert other variables and commands, even other macro definitions. The keyword `enddef` signals the end of the macro definition. The result is shown in **figure 2.1**; watch the bounding boxes. Close reading of the macro will reveal that the `fill`, `draw`, `filldraw` and `drawfill` macros are applied to the first `unitsquare` path in the macro.

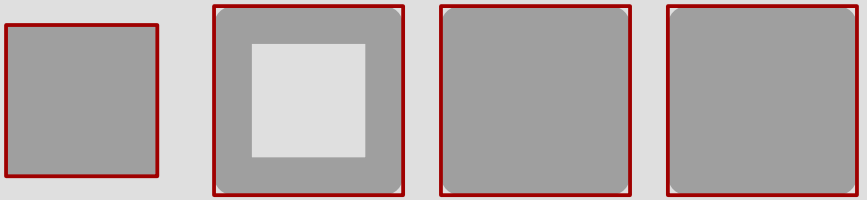


Figure 2.1 A `fill`, `draw`, `filldraw` and `drawfill` applied to the same square.

In this macro, `bbox` calls a macro that returns the enlarged bounding box of a path. By setting `bboxmargin` we can influence how much the bounding box is enlarged. Since this is an existing variable, we don't have to allocate it, like we do with `numeric n`. Unless you take special precautions, variables are global by nature and persistent outside macros.

```
picture finalpicture ; finalpicture := nullpicture ;
```

Just as `numeric` allocates an integer variable, the `picture` primitive allocates a picture data structure. We explicitly have to set this picture to nothing using the built–in primitive `nullpicture`.

Later on, we will add the drawn paths as accumulated in `currentpicture` to this `finalpicture` in the following manner.

```
addto finalpicture also currentpicture shifted(n*3cm,0) ;
```

Since we want to add a few more and don't want them to overlap, we shift them. Therefore we have to erase the current picture as well as increment the shift counter.

```
currentpicture := nullpicture ; n := n+1 ;
```

The `drawfill` macro is one of the METAFUN macros. Another handy macro is `boundingbox`. When used instead of `bbox`, you don't have to set the margin to zero.

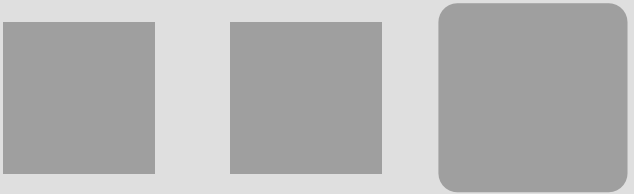


Figure 2.2 The influence of pens on `fill`.

There is a subtle point in filling a shape. In **figure 2.2** you see the influence of the pen on a `fill` operation. An indirect specification has no influence, and results in a filled rectangle with sharp corners. The third rectangle is drawn with a direct pen specification which results in a larger shape with rounds corners. However, the bounding box is the same in all three cases. The graphic is defined as follows. This time we don't use a (complicated) macro.

```
drawoptions (withcolor .625white) ;
path p ; p := unitsquare scaled 2cm ;
fill p shifted (3cm,0) ;
pickup pencircle scaled .5cm ; fill p shifted (6cm,0) ;
```

```
fill p shifted (9cm,0) withpen pencircle scaled .5cm ;
```

When a graphic is constructed, its components end up in an internal data structure in a more or less layered way. This means that as long as a graphic is not flushed, you may consider it to be a stack of paths and texts with the paths being drawn or filled shapes or acting as clipping paths or bounding boxes.

When you ask for the dimensions of a graphic the lower left and upper right corner are calculated using this stack. Because you can explicitly set bounding boxes, you can lie about the dimensions of a graphic. This is a very useful feature. In the rare case that you want to know the truth and nothing but the truth, you can tweak the `truecorners` numeric variable. We will demonstrate this with a few examples.

```
fill fullcircle scaled 1cm withcolor .625yellow ;
```

```
fill fullcircle scaled 1cm withcolor .625yellow ;
setbounds currentpicture to boundingbox currentpicture enlarged 2mm ;
```

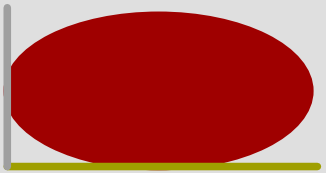
```
fill fullcircle scaled 1cm withcolor .625yellow ;
setbounds currentpicture to boundingbox currentpicture enlarged 2mm ;
interim truecorners := 1 ;
```

```
fill fullcircle scaled 1cm withcolor .625yellow ;
interim truecorners := 1 ;
setbounds currentpicture to boundingbox currentpicture enlarged 2mm ;
```

As you can see here, as soon as we set `truecorners` to 1, the bounding box settings are ignored.[6]

There are two related macros: `bbwidth` and `bbheight` that you can apply to a path.

```
fill unitcircle xscaled 4cm yscaled 2cm
    withpen pencircle scaled 1mm withcolor .625red ;
draw origin -- (bbwidth(currentpicture),0)
    withpen pencircle scaled 1mm withcolor .625yellow ;
draw origin -- (0,bbheight(currentpicture))
    withpen pencircle scaled 1mm withcolor .625white ;
```



[6] Normally you will use grouping to keep the interim local. In METAFUN each figure restores this variable at the beginning.

2.3

Units

Like TEX, METAPOST supports multiple units of length. In TEX, these units are hard coded and handled by the parser, where the internal unit of length is the scaled point (sp), something on the nanometer range. Because METAPOST is focused on POSTSCRIPT output, its internal unit is the big point (bp). All other units are derived from this unit and available as numeric instead of hard coded.

```
mm =  2.83464 ; pt =  0.99626 ; dd =  1.06601 ; bp :=  1 ;
cm = 28.34645 ; pc = 11.95517 ; cc = 12.79213 ; in := 72 ;
```

Careful reading reveals that only the bp and in are fixed, while the rest of the dimensions are scalar multiples of bp.

Since we are dealing with graphics, the most commonly used dimensions are pt, bp, mm, cm and in.

The text in the center of the leftmost graphic is typeset by METAPOST as a label.

```
fill fullsquare scaled 72.27pt withcolor .625yellow ;
fill fullcircle scaled 72.27pt withcolor white ;
label("72.27pt", center currentpicture) ;
```

In METAPOST the following lines are identical:

```
draw fullcircle scaled 100 ;
draw fullcircle scaled 100bp ;
```

You might be tempted to omit the unit, but this can be confusing, particularly if you also program in a language like METAFONT, where the `pt` is the base unit. This means that a circle scaled to 100 in METAPOST is not the same as a circle scaled to 100 in METAFONT. Consider the next definition:

```
pickup pencircle scaled 0 ;
fill unitsquare
  xscaled 400pt yscaled -.5cm withcolor .625red ;
fill unitsquare
  xscaled 400bp yscaled +.5cm withcolor .625yellow ;
drawoptions(withcolor white) ;
label.rt("400 pt", origin shifted (0, -.25cm)) ;
label.rt("400 bp", origin shifted (0, +.25cm)) ;
```

When processed, the difference between a `pt` and `bp` shows rather well. Watch how we use `.rt` to move the label to the right; you can compare this with TEX's macro `\rlap`. You might want to experiment with `.lft`, `.top`, `.bot`, `.ulft`, `.urt`, `.llft` and `.lrt`.

The difference between both bars is exactly 1.5pt (as calculated by TEX).

Where TEX is anchored in tradition, and therefore more or less uses the `pt` as the default unit, METAPOST, much like POSTSCRIPT, has its roots in the computer sciences. There, to simplify calculations, an inch is divided in 72 big points, and .72pt is sacrificed.

When you consider that POSTSCRIPT is a high end graphic programming language, you may wonder why this sacrifice was made. Although the difference between `1bp` and `1pt` is miniscule, this difference is the source of much (unknown) confusion. When TEX users talk about a `10pt` font, a desktop publisher hears `10bp`. In a similar vein, when we define a papersize having a width of `600pt` and a height of `450pt`, which is papersize `S6` in CONTEXT, a POSTSCRIPT or PDF viewer will report slightly smaller values as page dimensions. This is because those programs claim the `pt` to be a `bp`. [This confusion can lead to interesting discussions with desktop publishers when they have to use TEX. They often think that their demand of a baseline distance of `13.4` is met when we set it to `13.4pt`, while actually they were thinking of `13.4bp`, which of course in other programs is specified using a `pt` suffix.]

Therefore, when embedding graphics in CONTEXT, we strongly recommend that you use `pt` as the base unit instead. The main reason why we spend so many words on this issue is that, when neglected, large graphics may look inaccurate. Actually, when taken care of, it is one of the (many) reasons why TEX documents always look so accurate. Given that the eye is sensitive to distortions of far less than `1pt`, you can be puzzled by the fact that many drawing programs only provide a bounding box in rounded units. Thereby, they round to the next position, to prevent unwanted cropping. For some reason this low resolution has made it into the high end POSTSCRIPT standard.

In CONTEXT we try to deal with these issues as well as possible.

2.4 Scaling and shifting

When we draw a shape, METAPOST will adapt the bounding box accordingly. This means that a graphic has its natural dimensions, unless of course we adapt the bounding box manually. When you limit your graphic to a simple shape, say a rectangle, shifting it to some place can get obscured by this fact. Therefore, the following series of shapes appear to be the same.

```
draw
  unitsquare xscaled 6cm yscaled 1.5cm
  withpen pencircle scaled 2mm withcolor .625red ;
```

```
draw
  unitsquare shifted (.5,.5) xscaled 6cm yscaled 1.5cm
  withpen pencircle scaled 2mm withcolor .625red ;
```

```
draw
  unitsquare shifted (-.5,-.5) xscaled 6cm yscaled 1.5cm
  withpen pencircle scaled 2mm withcolor .625red ;
```

```
draw
```

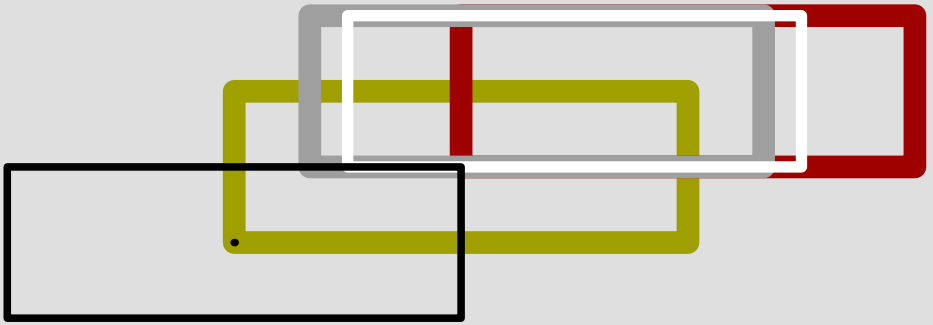
```
  unitsquare xscaled 6cm yscaled 1.5cm shifted (1cm,1cm)
  withpen pencircle scaled 2mm withcolor .625red ;
```

```
draw
  unitsquare xscaled 6cm yscaled 1.5cm shifted (1.5cm,1cm)
  withpen pencircle scaled 2mm withcolor .625red ;
```

However, when we combine such graphics into one, we will see in what respect the scaling and shifting actually takes place.

```
draw
  unitsquare xscaled 6cm yscaled 2cm
  withpen pencircle scaled 3.0mm withcolor .625yellow ;
draw
  unitsquare shifted (.5,.5) xscaled 6cm yscaled 2cm
  withpen pencircle scaled 3.0mm withcolor .625red ;
draw
  unitsquare xscaled 6cm yscaled 2cm shifted (1cm,1cm)
```

```
  withpen pencircle scaled 3.0mm withcolor .625white ;
draw
  unitsquare xscaled 6cm yscaled 2cm shifted (1.5cm,1cm)
  withpen pencircle scaled 1.5mm withcolor white ;
draw
  unitsquare shifted (-.5,-.5) xscaled 6cm yscaled 2cm
  withpen pencircle scaled 1mm withcolor black ;
draw origin withpen pencircle scaled 1mm ;
```



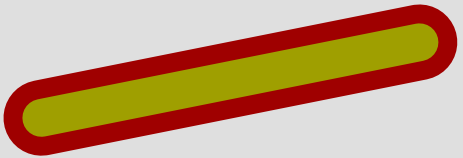
As you can see, the transformations are applied in series. Sometimes this is not what we want, in which case we can use parentheses to force the desired behaviour. The lesson learned is that *scaling and shifting* is not always the same as *shifting and scaling*.

```
draw
  origin -- origin shifted ((4cm,0cm) shifted (4cm,0cm))
  withpen pencircle scaled 1cm withcolor .625white ;
```

```
draw
  origin -- origin shifted (4cm,0cm) shifted (4cm,0cm)
  withpen pencircle scaled 8mm withcolor .625yellow ;
draw
  (origin -- origin shifted (4cm,0cm)) shifted (4cm,0cm)
  withpen pencircle scaled 6mm withcolor .625red ;
draw
  origin -- (origin shifted (4cm,0cm) shifted (4cm,0cm))
  withpen pencircle scaled 4mm withcolor white ;
```

Especially when a path results from a call to a macro, using parentheses around a path may help, as in the following example.

```
def unitslant = origin -- origin shifted (1,1) enddef ;
draw
  unitslant xscaled 5cm yscaled 1cm
  withpen pencircle scaled 1cm withcolor .625red ;
draw
  (unitslant) xscaled 5cm yscaled 1cm
  withpen pencircle scaled 5mm withcolor .625yellow ;
```



The next definition of `unitslant` is therefore better.

```
def unitslant = (origin -- origin shifted (1,1)) enddef ;
draw
  unitslant xscaled 5cm yscaled 1cm
  withpen pencircle scaled 5mm withcolor .625red ;
```

An even better alternative is:

```
path unitslant ; unitslant = origin -- origin shifted (1,1) ;
draw
  unitslant xscaled 5cm yscaled 1cm
  withpen pencircle scaled 5mm withcolor .625yellow ;
```

2.5

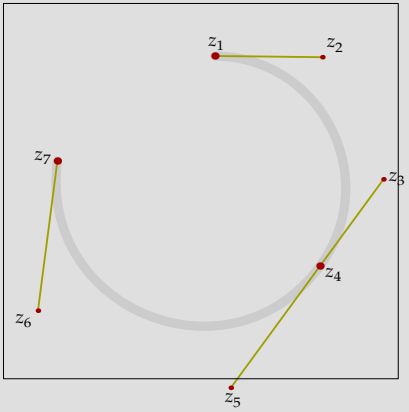
Curve construction

Chapter 3 of the METAFONT book explains the mathematics behind the construction of curves. Both METAFONT and METAPOST implement Bézier curves. The fact that these curves are named after Pierre Bézier obscures the fact that the math behind them originates with Sergeĭ Bernshteĭn.

The points on the curve are determined by the following formula:

$$z(t) = (1-t)^3 z_1 + 3(1-t)^2 t z_2 + 3(1-t)t^2 z_3 + t^3 z_4$$

Here, the parameter t runs from $[0,1]$. As you can see, we are dealing with four points. In practice this means that when we construct a curve from multiple points, we act on two points and the two control points in between. So, the segment that goes from z_1 to z_4 is calculated using these two points and the points that METAFONT/METAPOST calls post control point and pre control point.



The previous curve is constructed from the three points z_1, z_4 and z_7. The curve is drawn in METAPOST by `z1..z4..z7` and is made up out of two segments. The first segment is determined by the following points:

1. point z_1 of the curve
2. the postcontrol point z_2 of z_1
3. the precontrol point z_3 of z_4
4. point z_4 of the curve

On the next pages we will see how the whole curve is constructed from these quadruples of points. The process comes down to connecting the mid points of the straight lines to the points mentioned. We do this three times, which is why these curves are classified as third order approximations.

The first series of graphics demonstrates the process of determining the mid points. The third order midpoint is positioned on the final curve. The second series focuses on the results: new sets of four points that will be used in a next stage. The last series only shows the third order midpoints. As you can see, after some six iterations we have already reached a rather good fit of the final curve. The exact number of iterations depends on the resolution needed. You will notice that the construction speed (density) differs per segment.

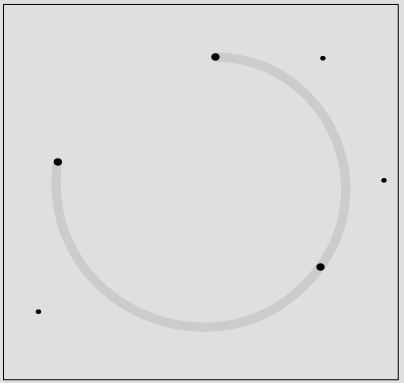
The path in these examples is defined as follows:

```
path p ; p := (4cm,4cm)..(6cm,0cm)..(1cm,2cm) ;
```

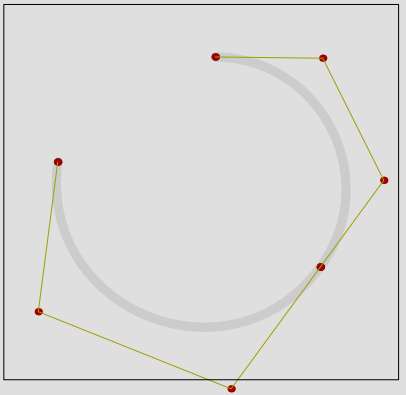
If you are playing with graphics like this, the METAFUN macro `randomize` may come in handy:

```
p := p randomized (1cm,.5cm) ;
```

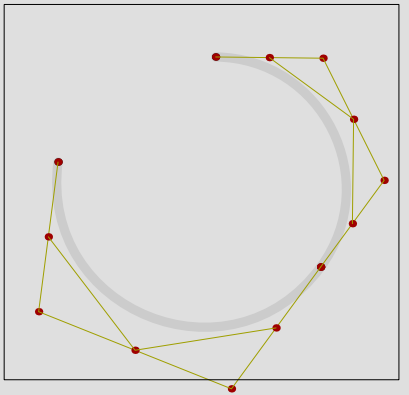
If we apply this operation a couple of times we can see how the control points vary. (Using the randomizer saves us the troubles of finding nice example values.) The angle between the tangent as well as the distance from the parent point determine the curve.



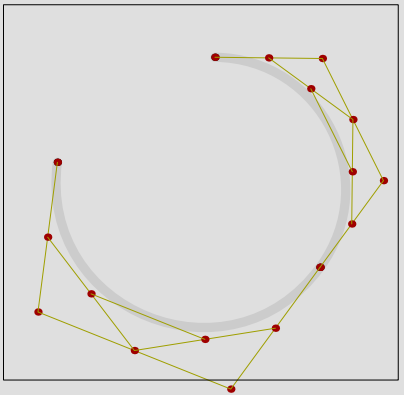
points



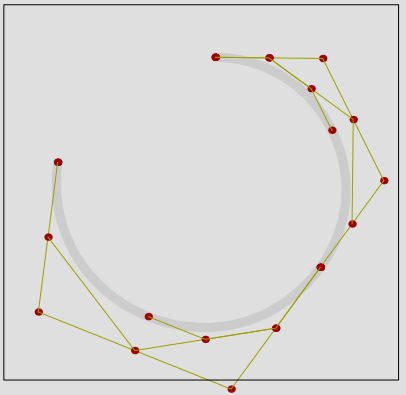
first order curve



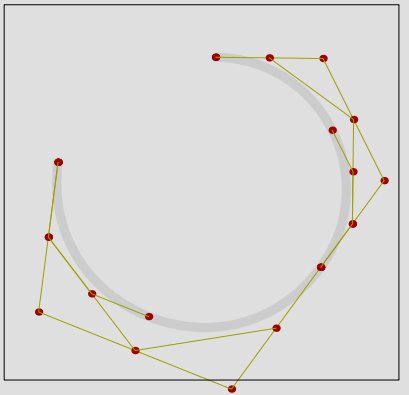
second order curve



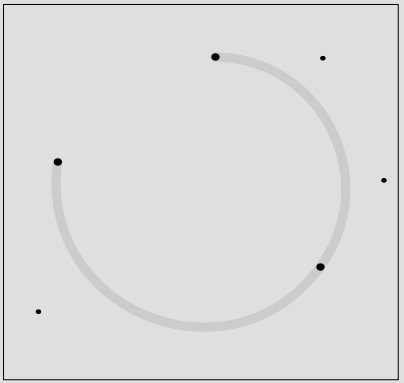
third order curve



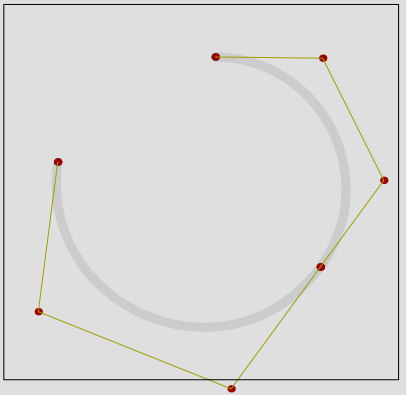
left side curves



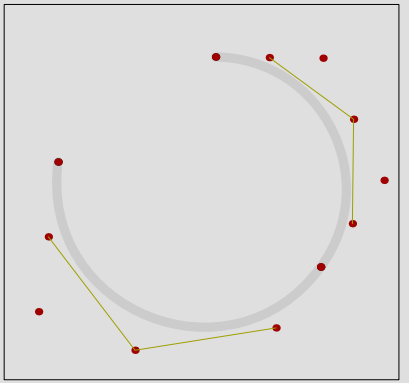
right side curves



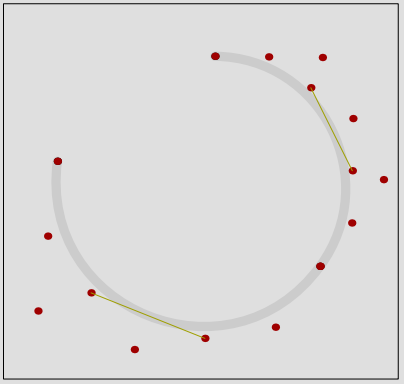
points



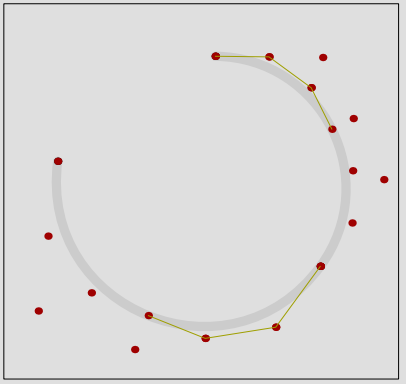
first order points



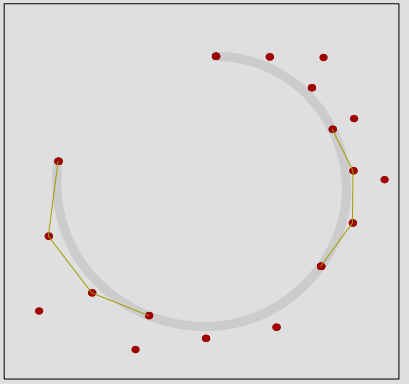
second order points



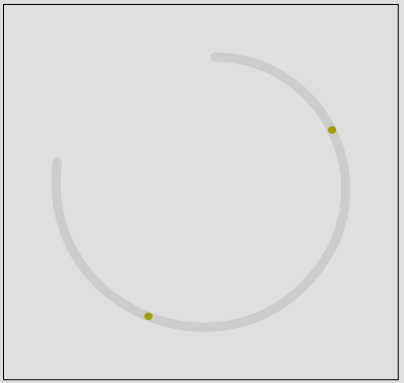
third order points



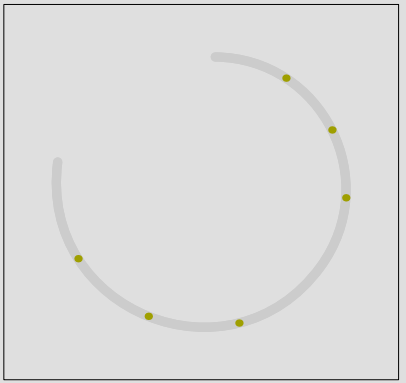
left side points



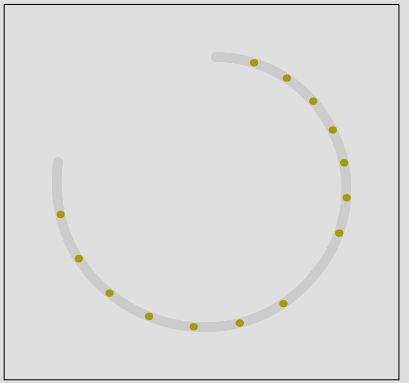
right side points



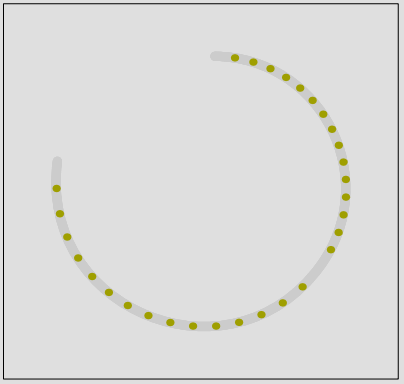
first iteration



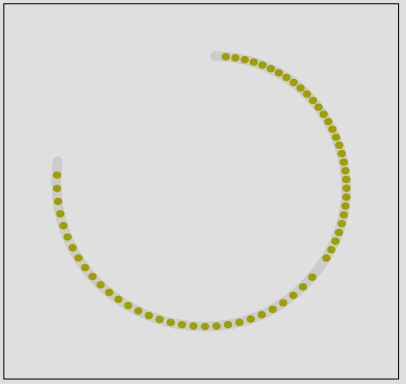
second iteration



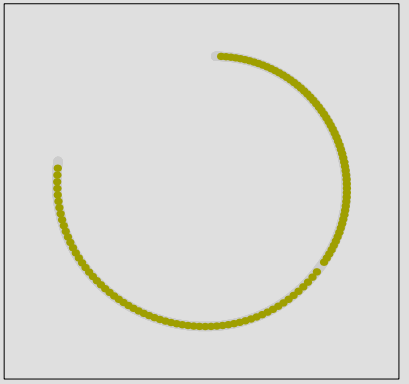
third iteration



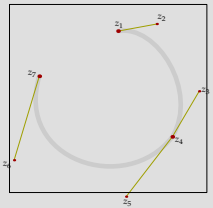
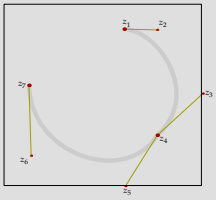
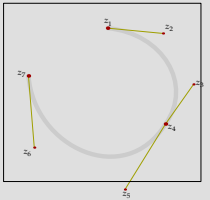
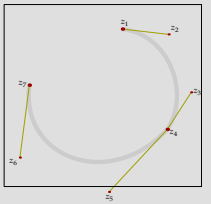
fourth iteration



fifth iteration



sixths iteration

Just in case you are interested in how such graphical simulations can be organized, we show simplified versions of the macros used here. (In the previous examples we minimized the complexity of the code by using buffers, but describing this mechanism is out of the scope of this section.)

We need to loop over all segments of a curve, where for each segment the left and right side sub curves are handled recursively, upto the requested depth (denoted as `n`). For this we define the following macros.

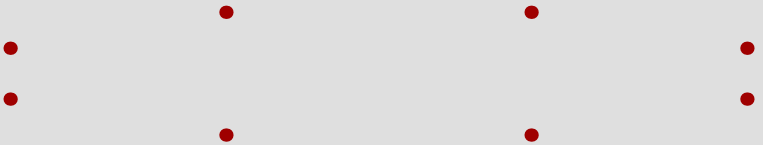
```
vardef dodrawmidpoints (expr a, b, c, d, n)  =
  save e, f, g, h, i, j ; pair e, f, g, h, i, j ;
  e := .5[a,b] ; f := .5[b,c] ; g := .5[c,d] ;
  h := .5[e,f] ; i := .5[f,g] ; j := .5[h,i] ;
  draw j ;
  if n>1 :
    dodrawmidpoints(a, e, h, j, n-1) ;
    dodrawmidpoints(j, i, g, d, n-1) ;
  fi ;
enddef ;

vardef drawmidpoints (expr p, n) =
  save a, b, c, d ; pair a, b, c, d ;
  for x=0 upto length(p)-1 :
```

```
      a := point x   of p ; b := postcontrol x   of p ;
      d := point x+1 of p ; c := precontrol  x+1 of p ;
      dodrawmidpoints(a, b, c, d, n) ;
    endfor ;
  enddef ;
```

We apply this macro to a simple shape:

```
drawmidpoints (fullcircle xscaled 300pt yscaled 50pt, 1) ;
```

When drawn, this results in the points that makes up the curve:



We now add an extra iteration (resulting in the yellow points):

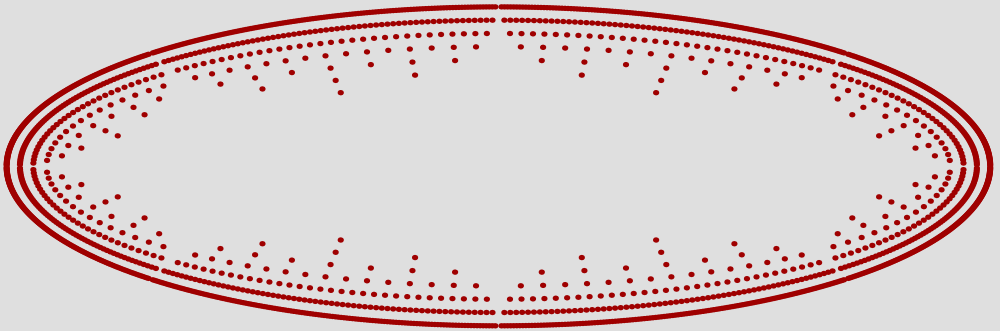
```
drawmidpoints (fullcircle xscaled 300pt yscaled 50pt, 2) ;
```

and get:

We don't even need that much iterations to get a good result. The depth needed to get a good result depends on the size of the pen and the resolution of the device on which the curve is visualized.

```
for i=1 upto 7 :
  drawmidpoints (fullcircle
    xscaled (300pt+i*10pt) yscaled (50pt+i*10pt), i) ;
endfor ;
```

Here we show 7 iterations in one graphic.



In practice it is not that trivial to determine the depth needed. The next example demonstrates how the resolution of the result depends on the length and nature of the segment.

```
drawmidpoints (fullsquare
  xscaled 300pt yscaled 50pt randomized (20pt,10pt), 5) ;
```

2.6 Inflection, tension and curl

The METAPOST manual describes the meaning of ... as "choose an inflection–free path between these points unless the endpoint directions make this impossible". To use the words of David Arnold: a point of inflection is where a path switches concavity, from concave up to concave down, for example.

It is surprisingly difficult to find nice examples that demonstrate the difference between .. and ..., as it is often 'impossible' to honour the request for less inflection. We will demonstrate this with a few graphics.

In the four figures on the next pages, you will see that ... is not really suited for taming wild curves. If you really want to make sure that a curve stays within certain bounds, you have to specify it as such using control or intermediate points. In the figures that follow, the gray curves draw the random path using .. on top of yellow curves that use the ... connection. As you can see, in only a few occasions do the yellow 'inflection' free curves show up.

For those who asked for the code that produces these pictures, we now include it here. We use a macro `sample` which we define as a usable graphic (nearly all examples in this manual are coded in the document source).

```
\startuseMPgraphic{sample}
def sample (expr rx, ry) =
  path p, q ; numeric n, m, r, a, b ;
  color c ; c := \MPcolor{lightgray} ;
```

```
  a := 3mm ; b := 2mm ; r := 2cm ; n := 7 ; m := 5 ;
  q := unitsquare scaled r xyscaled (n,m) shifted (.5r,.5r) ;
  draw q withpen pencircle scaled (b/4) withcolor .625yellow;
  for i=1 upto n : for j=1 upto m :
    p := (fullcircle scaled r randomized (r/rx,r/ry))
      shifted ((i,j) scaled r) ;
    pickup pencircle scaled a ;
    draw for k=0 upto length(p) :
      point k of p ..  endfor cycle withcolor c ;
    draw for k=0 upto length(p) :
      point k of p ... endfor cycle withcolor c ;
    pickup pencircle scaled b ;
    draw for k=0 upto length(p) :
      point k of p ..  endfor cycle withcolor .625yellow ;
    draw for k=0 upto length(p) :
      point k of p ... endfor cycle withcolor .625white ;
    for k=0 upto length(p) :
      draw point k of p withcolor .625red ;
    endfor ;
  endfor ; endfor ;
  setbounds currentpicture to q ;
enddef ;
\stopuseMPgraphic
```

As you see, not so much code is needed. The graphics themselves were produced with a couple of commands like:

```
\placefigure
  {Circles with minimized inflection and 25\% randomized points.}
  {\startMPcode
     \includeMPgraphic{sample} ; sample(4,4) ;
   \stopMPcode}
```

The tension specifier can be used to influence the curvature. To quote the METAPOST manual once more: "The tension parameter can be less than one, but it must be at least 3/4". The following paths are the same:

```
z1 .. z2
z1 .. tension 1 .. z2
z1 .. tension 1 and 1 .. z2
```

The triple dot command `...` is actually a macro that makes the following commands equivalent. Both commands will draw identical paths.

```
z1 ... z2
z1 .. tension atleast 1 .. z2
```

The `atleast` directive tells METAPOST to do some magic behind the screens. Both the 3/4 and the `atleast` lead directly to the question: "What, exactly, is the influence of the tension directive?" We will try to demystify the `tension` specifier through a sequence of graphics.

```
u := 1cm ; z1 = (0,0) ; z2 = (2u,4u) ; z3 = (4u,0) ;
def sample (expr p, c) =
  draw p withpen pencircle scaled 2.5mm withcolor white ;
  draw p withpen pencircle scaled 2.0mm withcolor c ;
enddef ;
```

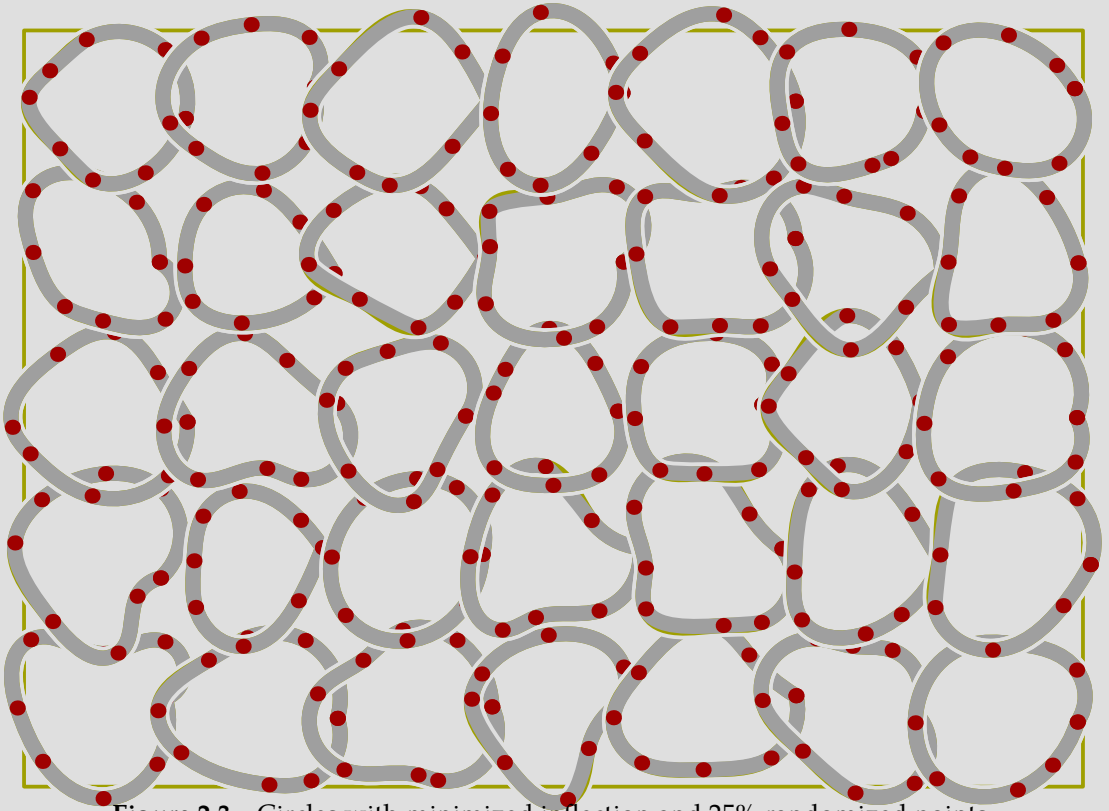


Figure 2.3 Circles with minimized inflection and 25% randomized points.

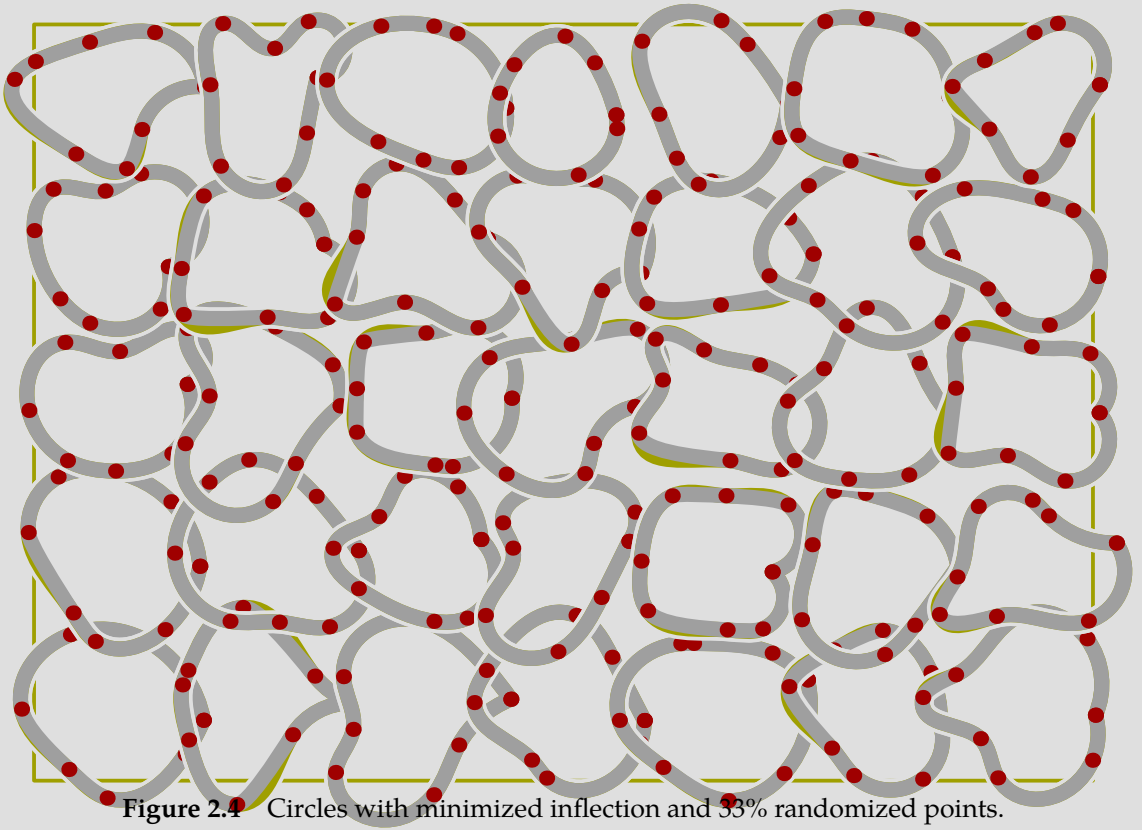


Figure 2.4 Circles with minimized inflection and 33% randomized points.

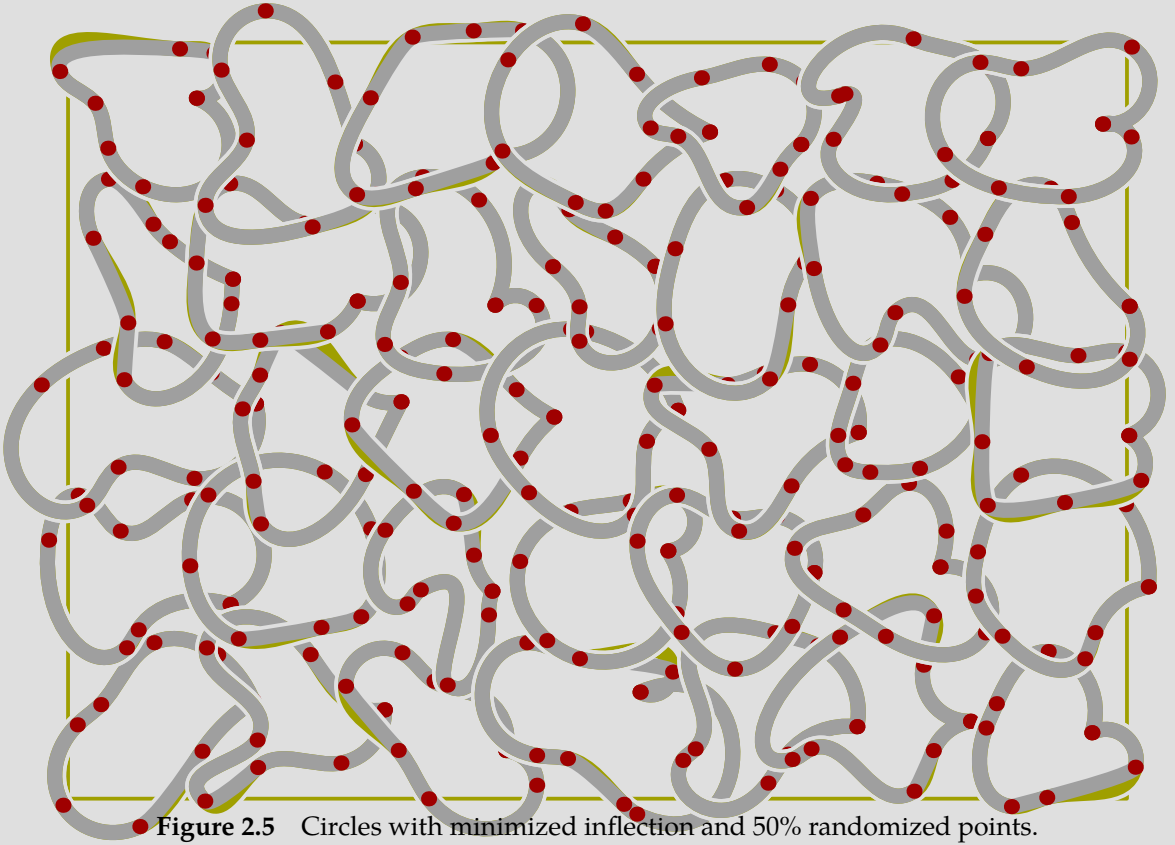


Figure 2.5 Circles with minimized inflection and 50% randomized points.

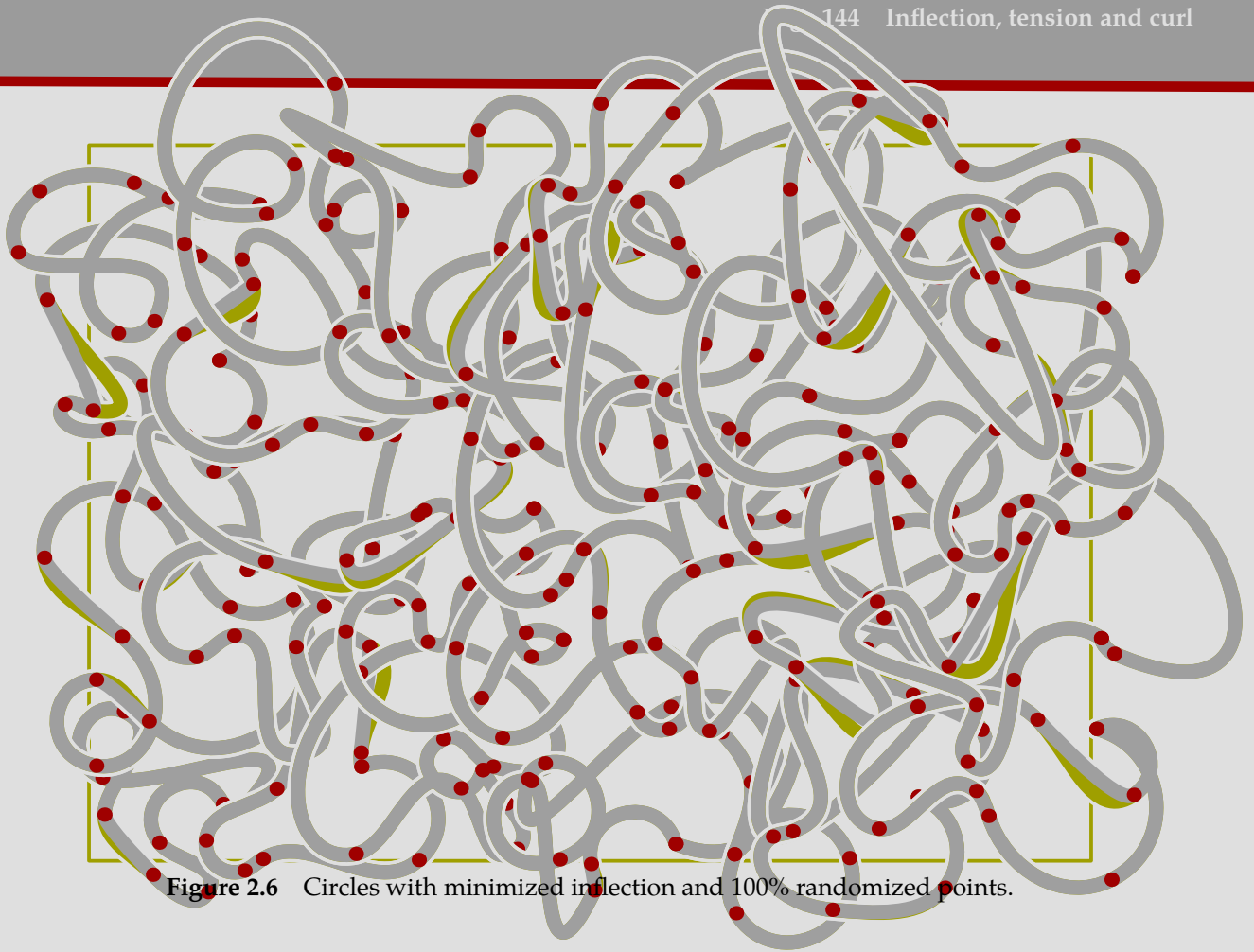
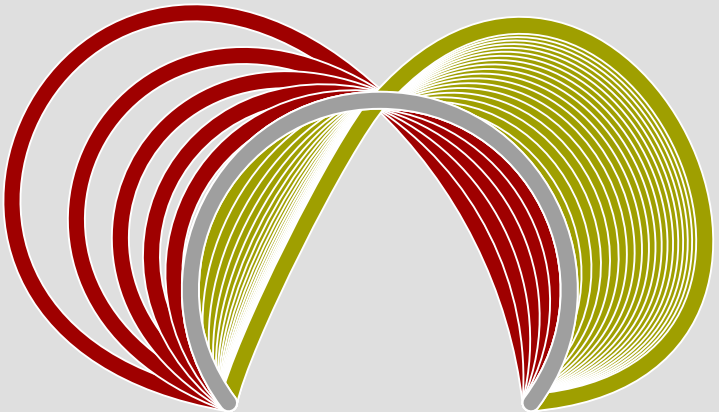


Figure 2.6 Circles with minimized inflection and 100% randomized points.

```
for i=.75 step .05 until 1 :
  sample (z1 .. tension i .. z2 .. z3, .625red) ;
endfor ;
for i=1 step .05 until 2 :
  sample (z1 .. tension i .. z2 .. z3, .625yellow) ;
endfor ;
sample (z1 .. z2 .. z3, .625white) ;
sample (z1 ... z2 ... z3, .625white) ;
```

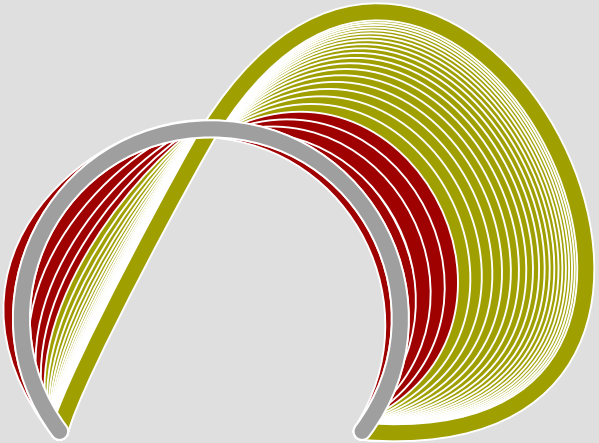
Indeed values less than .75 give an error message, but large values are okay. As you can see, the two gray curves are the same. Here, `atleast 1` means 1, even if larger values are useful.



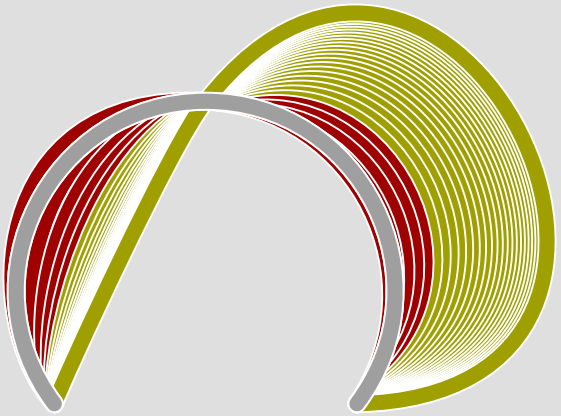
Curves finally are made up out of points, and each point has two control points. Since the `tension` specifier finally becomes a control point, it is not surprising that you may specify two tension values. If we replace the tension in the previous example by

```
.. tension i and 2i ..
```

we get the following graphic:



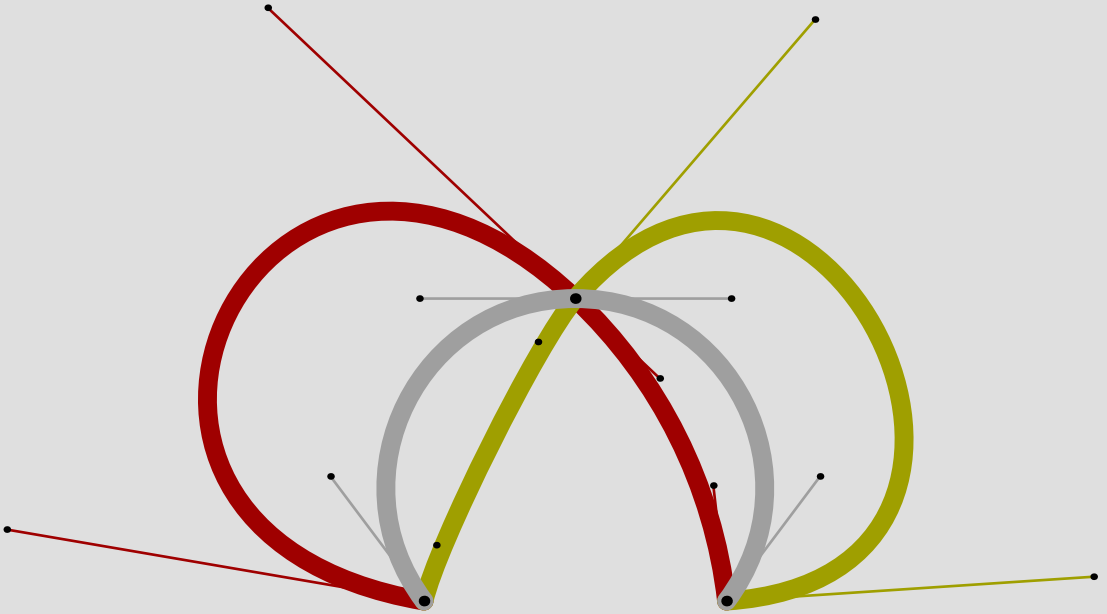
If we swap both values (`.. tension 2i and i ..`) we get:



We mentioned control points. We will now draw a few extreme tensions and show the control points as METAPOST calculates them.

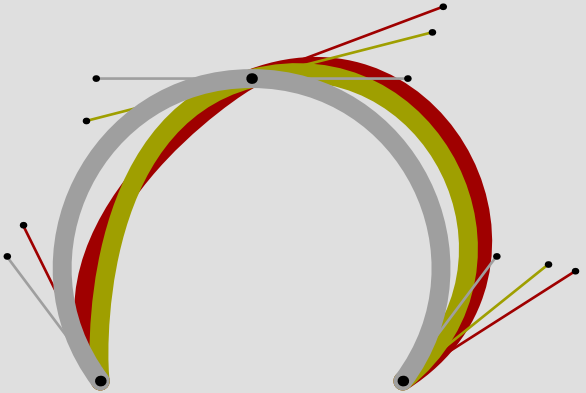
```
sample (z1 .. tension 0.75 .. z2 .. z3, .625red) ;
sample (z1 .. tension 2.00 .. z2 .. z3, .625yellow) ;
sample (z1 ..                 z2 .. z3, .625white) ;
```

First we will show the symmetrical tensions.



The asymetrical tensions are less prominent. We use the following values:

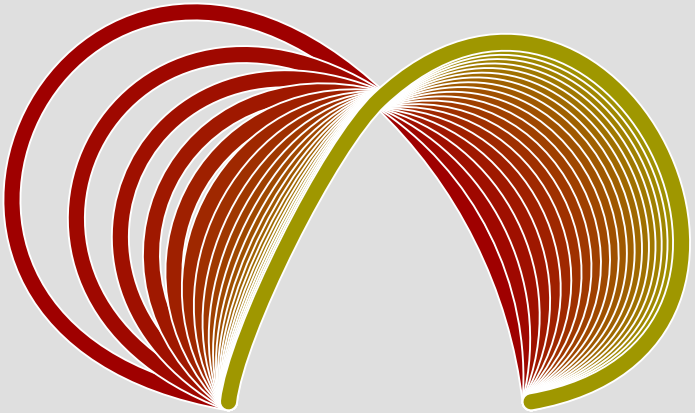
```
sample (z1 .. tension   .75 and 10    .. z2 .. z3, .625red) ;
sample (z1 .. tension 10    and   .75 .. z2 .. z3, .625yellow) ;
sample (z1 ..                         z2 .. z3, .625white) ;
```



What happens when you use the METAPOST maximum value of `infinity` instead of 10? Playing with this kind of graphic can be fun, especially when we apply a few tricks.

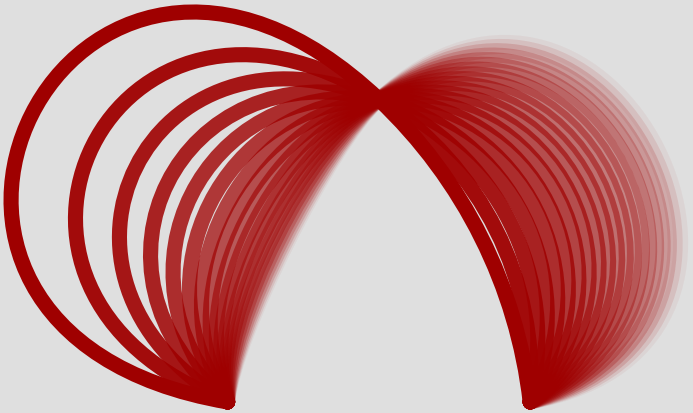
```
def sample (expr p, c) =
  draw p withpen pencircle scaled 2.5mm withcolor white ;
  draw p withpen pencircle scaled 2.0mm withcolor c ;
enddef;

u := 1cm ; z1 = (0,0) ; z2 = (2u,4u) ; z3 = (4u,0) ;

for i=0 step .05 until 1 :
  sample(z1 .. tension (.75+i) .. z2 .. z3, i[.625red,.625yellow]) ;
endfor;
```

Here we change the color along with the tension. This clearly demonstrates that we're dealing with a non linear phenomena.



We can (misuse) transparant colors to illustrate how the effect becomes less with growing tension.

```
def sample (expr p) (text c)=
  draw p withpen pencircle scaled 2.0mm withcolor c ;
enddef;

u := 1cm ; z1 = (0,0) ; z2 = (2u,4u) ; z3 = (4u,0) ;

for i=0 step .05 until 1 :
  sample(z1 .. tension (.75+i) .. z2 .. z3, transparent(1,1-i,.625red)) ;
endfor;
```



A third magic directive is `curl`. The curl is attached to a point between `{ }`, like `{curl 2}`. Anything between curly braces is a direction specifier, so instead of a `curl` you may specify a vector, like `{(2,3)}`, a pair of numbers, as in `{2,3}`, or a direction, like `{dir 30}`. Because vectors and angles are straightforward, we will focus a bit on `curl`.

```
z0 .. z1 .. z2
z0 {curl 1} .. z1 .. {curl 1} z2
```

So, a `curl` of 1 is the default. When set to 1, the begin and/or end points are approached. Given the following definitions:

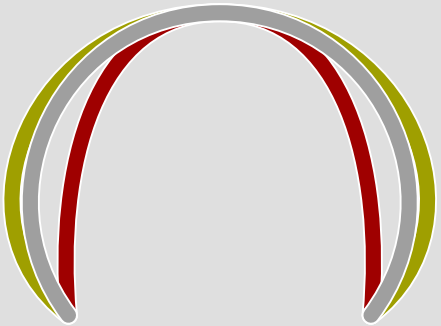
```
u := 1cm ; z1 = (0,0) ; z2 = (2u,4u) ; z3 = (4u,0) ;
def sample (expr p, c) =
  draw p withpen pencircle scaled 2.5mm withcolor white ;
```

```
  draw p withpen pencircle scaled 2.0mm withcolor c ;
enddef ;
```

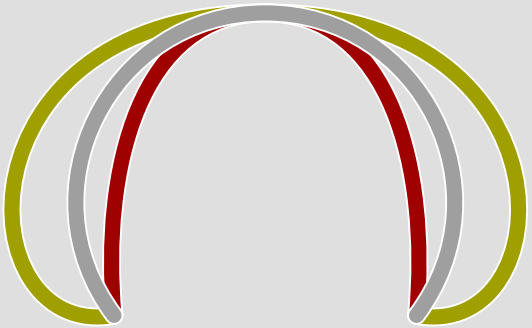
We can draw three curved paths.

```
sample (z1 {curl 0} .. z2 .. {curl 0} z3, .625red) ;
sample (z1 {curl 2} .. z2 .. {curl 2} z3, .625yellow) ;
sample (z1 {curl 1} .. z2 .. {curl 1} z3, .625white) ;
```

The third (gray) curve is the default situation, so we could have left the `curl` specifier out of the expression.



The curly specs have a lower bound of zero and no upper bound. When we use METAPOST maximum value of `infinity` instead of 2, we get:



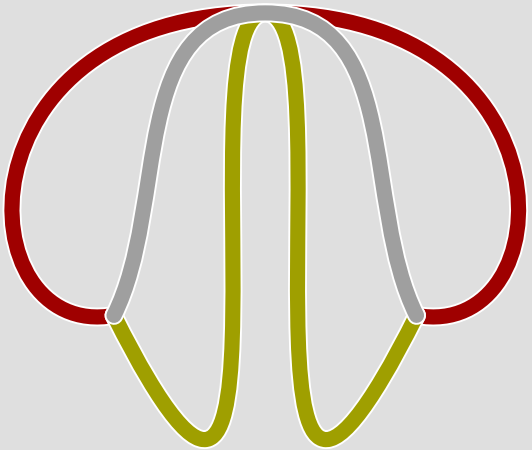
These curves were defined as:

```
sample (z1 {curl        0} .. z2 .. {curl        0} z3, .625red) ;
sample (z1 {curl infinity} .. z2 .. {curl infinity} z3, .625yellow) ;
sample (z1 {curl        1} .. z2 .. {curl        1} z3, .625white) ;
```

It may sound strange, but internally METAPOST can handle larger values than `infinity`.

```
sample (z1 {curl  infinity} .. z2 .. {curl  infinity} z3, .625red) ;
sample (z1 {curl 4infinity} .. z2 .. {curl 4infinity} z3, .625yellow) ;
sample (z1 {curl 8infinity} .. z2 .. {curl 8infinity} z3, .625white) ;
```

Although this is quite certainly undefined behaviour, interesting effects can be achieved. When you turn off METAPOST's first stage overflow catcher by setting `warningcheck` to zero, you can go upto 8 times `infinity`, which, being some 2^{15}, is still far from what today's infinity is supposed to be.



As the built–in METAPOST command `..` accepts the `curl` and `tension` directives as described in this section, you will now probably understand the following plain METAPOST definitions:

```
def --  = {curl 1} .. {curl 1}    enddef ;
def --- = .. tension infinity ..  enddef ;
def ... = .. tension atleast 1 .. enddef ;
```

These definitions also point out why you cannot add directives to the left or right side of `--`, `---` and `...`: they are directives themselves!

2.7 Transformations

A `transform` is a vector that is used in what is called an affine transformation. To quote the METAPOST manual:

"If $p = (p_x, p_y)$ is a pair and T is a transform, then `p transform T` is a pair of the form:

$$(t_x + t_{xx}p_x + t_{xy}p_y, t_y + t_{yx}p_x + t_{yy}p_y)$$

where the six numeric quantities $(t_x, t_y, t_{xx}, t_{xy}, t_{yx}, t_{yy})$ determine T."

In literature concerning POSTSCRIPT and PDF you will find many references to such transformation matrices. A matrix of $(s_x, 0, 0, s_y, 0, 0)$ is scaling by s_x in the horizontal direction and s_y in the vertical direction, while $(1, 0, t_x, 1, 0, t_y)$ is a shift over t_x, t_y. Of course combinations are also possible.

Although these descriptions seem in conflict with each other in the nature and order of the transform components in the vectors, the concepts are the same. You normally populate transformation matrices using `scaled`, `shifted`, `rotated`.

```
transform t ; t := identity shifted (a,b) rotated c scaled d ;
path p ; p := fullcircle transformed t ;
```

The previous lines of code are equivalent to:

```
path p ; p := fullcircle shifted (a,b) rotated c scaled d ;
```

You always need a starting point, in this case the identity matrix `identity`: $(0, 0, 1, 0, 0, 1)$. By the way, in POSTSCRIPT the zero vector is $(1, 0, 0, 1, 0, 0)$. So, unless you want to extract the components using `xpart`, `xypart`, `xxpart`, `ypart`, `yxpart` and/or `yypart`, you may as well forget about the internal representation.

You can invert a transformation using the `inverse` macro, which is defined as follows, using an equation:

```
vardef inverse primary T =
  transform T_ ; T_ transformed T = identity ; T_
enddef ;
```

Using transform matrices makes sense when similar transformations need to be applied on many paths, pictures, pens, or other transforms. However, in most cases you will use the predefined commands `scaled`, `shifted`, `rotated` and alike. We will now demonstrate the most common transformations in a text example.

```
draw btex \bfd MetaFun etex ;
draw boundingbox currentpicture withcolor .625yellow ;
```

Before a METAPOST run, the `btex ... etex`'s are filtered from the file and passed on to TEX. After that, the DVI file is converted to a list of pictures, which is consulted by METAPOST.[7] We can manipulate these pictures like any graphic as well as draw it with `draw`.

We show the transformations in relation to the origin and make the origin stand out a bit more by painting it a bit larger in white first.

```
draw origin withpen pencircle scaled 1.5mm withcolor white ;
draw origin withpen pencircle scaled 1mm withcolor .625red
```

The origin is in the lower left corner of the picture.

Because the transformation keywords are proper english, we let the pictures speak for themselves.

[7] This is no longer the case in LUATEX where we use MPLIB.

```
currentpicture := currentpicture shifted (0,-1cm) ;
```


```
currentpicture := currentpicture rotated 180 ;
```


```
currentpicture := currentpicture rotatedaround(origin,30) ;
```


```
currentpicture := currentpicture scaled 1.75 ;
```

```
currentpicture := currentpicture scaled -1 ;
```

```
currentpicture := currentpicture xscaled 3.50 ;
```

```
currentpicture := currentpicture xscaled -1 ;
```

```
currentpicture := currentpicture yscaled .5 ;
```

```
currentpicture := currentpicture yscaled -1 ;
```

```
currentpicture := currentpicture slanted .5 ;
```

```
currentpicture := currentpicture slanted -.5 ;
```


```
currentpicture := currentpicture zscaled (.75,.25) ;
```


```
currentpicture := currentpicture
  reflectedabout(llcorner currentpicture,urcorner currentpicture) ;
```

A path has a certain direction. When the `turningnumber` of a path is larger than zero, it runs in clockwise direction. The METAPOST primitive `reverse` changes the direction, while the macro `counterclockwise` can be used to get a path running in a well defined direction.

```
drawoptions(withpen pencircle scaled 2pt withcolor .625red) ;
path p ; p := fullcircle scaled 1cm ;
drawarrow                          p ;
drawarrow reverse                  p shifted (2cm,0) ;
```

```
drawarrow counterclockwise         p shifted (4cm,0) ;
drawarrow counterclockwise reverse p shifted (6cm,0) ;
drawarrow reverse counterclockwise p shifted (8cm,0) ;
```

2.8 Only this far

When you take a close look at the definitions of the Computer Modern Roman fonts, defined in the METAFONT book, you will notice a high level of abstraction. Instead of hard coded points you will find points defined in terms of 'being the same as this point' or 'touching that point'. In this section we will spend some time on this touchy aspect.

This rectangle is a scaled instance of the predefined METAFUN path `fullsquare` which is centered around the origin.

```
pickup pencircle scaled 2mm ;
path p ; p := fullsquare scaled 2cm ;
draw p withcolor .625white ;
```

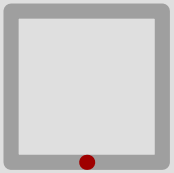
On this path, halfway between two of its corners, we define a point q:

```
pair q ; q := .5[llcorner p, lrcorner p] ;
```

We draw this point in red, using:

```
draw q withcolor .625red ;
```

As you can see, this point is drawn on top of the path.



There are four of those midpoints, and when we connect them, we get:

Because path p is centered around the origin, we can simply rotate point q a few times.

```
draw q -- q rotated 90 -- q rotated 180 --
  q rotated 270 -- cycle withcolor .625red ;
```

There are situations, where you don't want the red path to be drawn inside another path, or more general: where you want points to touch instead of being overlayed.

We can achieve this by defining point `q` to be located on top of the midpoint.

```
pair q ; q := top .5[llcorner p, lrcorner p] ;
```

The predefined macro `top` moves the point over the distance similar to the current pen width.

Because we are dealing with two drawing operations, and since the path inside is drawn through the center of points, we need to repeat this move in order to draw the red path really inside the other one.

```
pair q ; q := top top .5[llcorner p, lrcorner p] ;
```

Operations like `top` and its relatives `bot`, `lft` and `rt` can be applied sequentally.

We already showed that q was defined as a series of rotations.

```
draw q -- q rotated 90 -- q rotated 180 --
  q rotated 270 -- cycle withcolor .625red ;
```

As an intermezzo we will show an alternative definition of q. Because each point is rotated 90 degrees more, we can define a macro that expands into the point and rotates afterwards. Because each consecutive point on the path is rotated an additional 90 degrees, we use the METAPOST macro `hide` to isolate the assignment. The `hide` command executes the hidden command and afterwards continues as if it were never there. You must not confuse this with grouping, since the hidden commands are visible to its surroundings.

```
def qq = q hide(q := q rotated 90) enddef ;
draw qq -- qq -- qq -- qq -- cycle withcolor .625red ;
```

The macro `top` uses the characteristics of the current pen to determine the displacement. However, for the more complicated pen shapes we need a different trick to get an inside path. Let's start by defining an elliptical path.

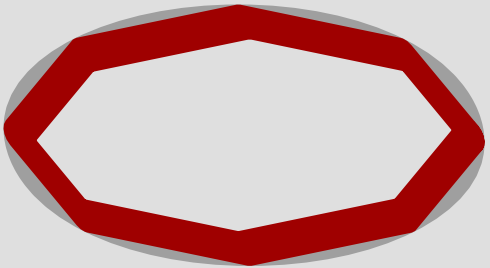
```
pickup pencircle xscaled 3mm yscaled 5mm rotated 30 ;
path p ; p := fullcircle xscaled 6cm yscaled 3cm ;
draw p withcolor .625white ;
```

We draw this path using a non standard pen. In the METAFONT manual you will find methods to draw shapes with similar pens, where the pen is also turning, as it does in real calligraphy. Here we stick to a more simple one.

We construct the inner path from the points that make up the curve. Watch how we use a for loop to compose the new path. When used this way, no semi colon may be used to end the loop, since it would isolate the color directive.

```
draw point 0 of p
  for i=1 upto length(p) : -- point (i) of p endfor
  withcolor .625red ;
```

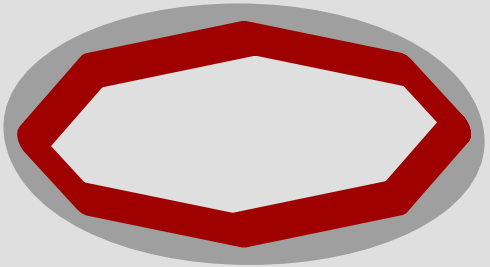
The points are still located on the original path.



We can move the points to the inside by shifting them over the penwidth in the direction perpendicular to the point. Because we use this transformation more than once, we wrap it into a macro. This also keeps the code readable.

```
vardef inside expr pnt of p =
  (point pnt of p shifted
     -(penoffset direction pnt of p of currentpen))
enddef ;
draw inside 0 of p
  for i=1 upto length(p) : -- inside i of p endfor
  withcolor .625red ;
```

Whenever you define a pen, METAPOST stores its characteristics in some private variables which are used in the `top` and alike directives. The `penoffset` is a built in primitive and is defined as the "point on the pen furthest to the right of the given direction". Deep down in METAPOST pens are actually simple paths and therefore METAPOST has a notion of a point on the penpath. In the METAFONT book and METAPOST manual you can find in depth discussions on pens.

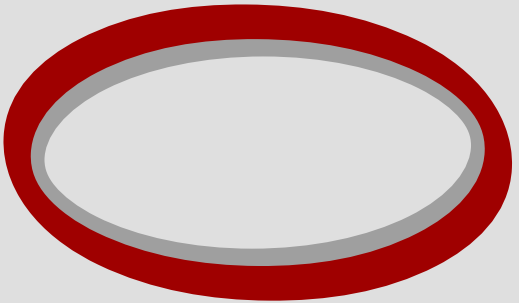


We're still not there. Like in a previous example, we need to shift over twice the pen width. To get good results, we should determine the width of the pen at that particular point, which is not trivial. The more general solution, which permits us to specify the amount of shifting, is as follows.

```
vardef penpoint expr pnt of p =
  save n, d ; numeric n, d ;
  (n,d) = if pair pnt : pnt else : (pnt,1) fi ;
  (point n of p shifted
     ((penoffset direction n of p of currentpen) scaled d))
enddef ;
```

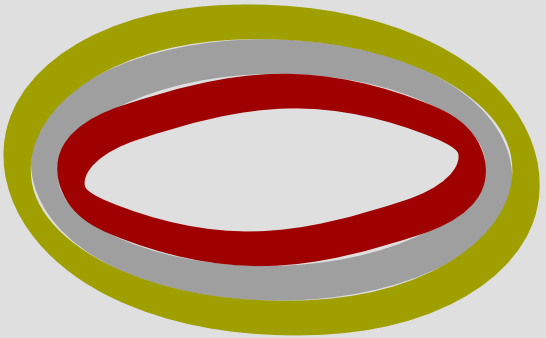
When the point specification is extended with a distance, in which case we have a pair expression, the point and distance are derived from this specification. First we demonstrate the simple case:

```
draw penpoint 0 of p
  for i=1 upto length(p)-1 : .. penpoint i of p endfor .. cycle
  withcolor .625red ;
```



In the next graphic, we draw both an inner and and outer path.

```
draw penpoint (0,-2) of p
  for i=1 upto length(p)-1 : .. penpoint (i,-2) of p endfor .. cycle
  withcolor .625red ;
draw penpoint (0,+2) of p
  for i=1 upto length(p)-1 : .. penpoint (i,+2) of p endfor .. cycle
  withcolor .625yellow ;
```



Another case when top and friends cannot be applied in a general way is the following. Consider the three paths:

```
path p, q, r ;

p := fullcircle scaled 3cm ;
q := p shifted (7cm,0cm) ;
r := center p -- center q ;
```

We draw these paths with:

```
draw p withpen pencircle scaled 10pt withcolor .625red ;
draw q withpen pencircle scaled 10pt withcolor .625yellow ;
draw r withpen pencircle scaled 20pt withcolor .625white ;
```

The line is drawn from center to center and since the line has a non zero width and a round line cap, it extends beyond this point.

If we want to line to stop at the circular paths, we can cut off the pieces that extend beyond those paths.

```
pair pr, qr ;

pr := p intersectionpoint r ;
qr := q intersectionpoint r ;

r := r cutbefore pr cutafter qr ;
```

This time we get:

Due to the thicker line width used when drawing the straight line, part of that line is still visible inside the circles. So, we need to clip off a bit more.

```
r := r cutbefore (point 5pt on r) ;
r := r cutafter (point -5pt on r) ;
```

The `point ... on` operation is a METAFUN macro that takes a dimension.

In order to save you some typing, METAFUN provides a macro `cutends` that does the same job:

```
r := r cutends 5pt ;
```

This time we draw the path in a different order:

```
draw r withpen pencircle scaled 20pt withcolor .625white ;
draw p withpen pencircle scaled 10pt withcolor .625red ;
draw q withpen pencircle scaled 10pt withcolor .625yellow ;
```

That way we hide the still remaining overlapping part of the line.

2.9 Directions

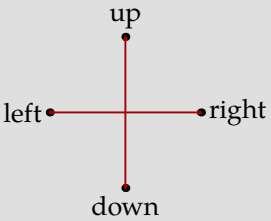
Quite often you have to tell METAPOST in what direction a line should be drawn. A direction is specified as a vector. There are four predefined vectors: `up`, `down`, `left`, `right`. These are defined as follows:

```
pair up, down, left, right ;
up = -down = (0,1) ; right = -left = (1,0) ;
```

We can use these predefined pairs as specifications and in calculations.

```
dotlabel.top("up"   , up    * 1cm) ;
dotlabel.bot("down" , down  * 1cm) ;
dotlabel.lft("left" , left  * 1cm) ;
dotlabel.rt ("right", right * 1cm) ;

drawoptions (withpen pencircle scaled .25mm withcolor .625 red) ;

drawarrow origin -- up    * 1cm ;
drawarrow origin -- down  * 1cm ;
```

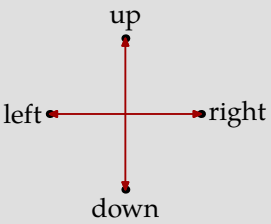
```
drawarrow origin -- left  * 1cm ;
drawarrow origin -- right * 1cm ;
```



This graphic can also be defined in a more efficient (but probably more cryptic) way. The next definition demonstrates a few nice tricks. Instead of looping over the four directions, we loop over their names. Inside the loop we convert these names, or strings, into a pair by scanning the string using `scantokens`. The `freedotlabel` macro is part of METAFUN and takes three arguments: a label string (or alternatively a picture), a point (location), and the 'center of gravity'. The label is positioned in the direction opposite to this center of gravity.

```
pair destination ;
for whereto = "up", "down", "left", "right" :
  destination := scantokens(whereto) * 1cm ;
  freedotlabel(whereto, destination, origin) ;
  drawarrow origin -- destination
    withpen pencircle scaled .25mm withcolor .625 red ;
endfor ;
```

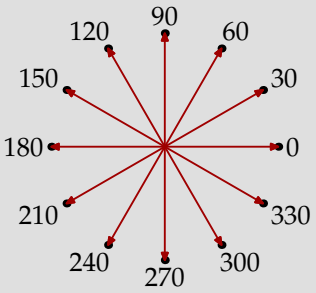
So, in this code fragment, we use the string as string and (by means of `scantokens`) as a point or vector.



The previous definition is a stepping stone to the next one. This time we don't use points, but the `dir` command. This command converts an angle into an unitvector.

```
pair destination ;
for whereto = 0 step 30 until 330 :
  destination := dir(whereto) * 1.5cm ;
  freedotlabel(decimal whereto, destination, origin) ;
  drawarrow origin -- destination
    withpen pencircle scaled .25mm withcolor .625 red ;
endfor ;
```

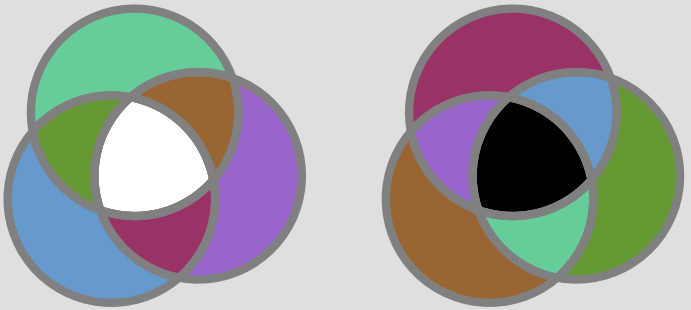
In METAPOST the angles go counter clockwise, which is not that illogical if you look at it from the point of view of vector algebra.



2.10 Analyzing pictures

Unless you really want to know all details, you can safely skip this section. The METAPOST features discussed here are mainly of importance when you write (advanced) macros.

We can decompose METAPOST pictures using a `within` loop. You may wonder if such a `within` loop construct has any real application, and as you can expect, it has. In **section 13.4** a macro is defined that draws a colored circle. If you want the inverted alternative, you can pass the inverted color specification, but wouldn't it be more convenient if there was an operator that did this for you automatically? Unfortunately there isn't one so we have to define one ourselves in a macro.



These circles were drawn using:

```
colorcircle(4cm,(.4,.6,.8),(.4,.8,.6),(.6,.4,.8)) ;
addto currentpicture also inverted currentpicture shifted (5cm,0) ;
```

When we `draw` a path, or stroke a path, as it is called officially, we actually perform an addition:

```
addto currentpicture doublepath somepath
```

The `fill` command is actually:

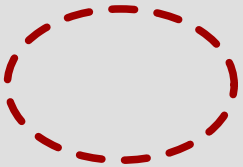
```
addto currentpicture contour somepath
```

We will need both `doublepath` and `contour` operations in the definition of `inverted`.

When METAPOST has digested a path into a picture, it keeps track of some characteristics. We can ask for them using `part...` operators. The following operators can be applied to a transform vector (one of METAPOST's data types), but also to a picture. Say that we have drawn a circle:

```
draw fullcircle
  xscaled 3cm yscaled 2cm
  dashed dashpattern(on 3mm off 3mm)
  withpen pencircle scaled 1mm
  withcolor .625red ;
picture p ; p := currentpicture ;
```

This circle looks like:

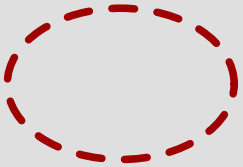


We can now ask for some of the characteristics of `currentpicture`, like its color. We could write the values to the log file, but it is more convenient to put them on paper.

```
label.rt("redpart: "   & decimal redpart   p, (4cm,+.5cm)) ;
label.rt("greenpart: " & decimal greenpart p, (4cm,  0cm)) ;
label.rt("bluepart: "  & decimal bluepart  p, (4cm,-.5cm)) ;
```

Here the `&` glues strings together, while the decimal operator converts a number into a string.

The result has no typographic beauty —keep in mind that here we use METAPOST to typeset the text—but the result serves its purpose.

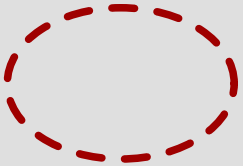


redpart: 0.625
greenpart: 0
bluepart: 0

We can also ask for the path itself (`pathpart`), the pen (`penpart`) and the dashpattern (`dashpart`), but these can only be assigned to variables of the corresponding type.

A path can be stroked or filled, in which case it is a cyclic path. It can have a non natural bounding box, be a clip path, consist of line segments or contain text. All these characteristics can be tested.

```
label.rt("filled: "  & condition filled         p, (4cm,+1.25cm)) ;
label.rt("stroked: " & condition stroked        p, (4cm,+0.75cm)) ;
label.rt("textual: " & condition textual        p, (4cm,+0.25cm)) ;
label.rt("clipped: " & condition clipped        p, (4cm,-0.25cm)) ;
label.rt("bounded: " & condition bounded        p, (4cm,-0.75cm)) ;
label.rt("cycle: "   & condition cycle pathpart p, (4cm,-1.25cm)) ;
```



filled: false
stroked: true
textual: false
clipped: false
bounded: false
cycle: true

In this code snippet, `condition` is a macro that takes care of translating a boolean value into a string (like `decimal` does with a numeric value).

```
def condition primary b =
  if b : "true" else : "false" fi
enddef ;
```

Clip paths and bounding boxes are kind of special in the sense that they can obscure components. The following examples demonstrate this. In case of a clip path or bounding box, the `pathpart` operator returns this path. In any case that asking for a value does not make sense —a clipping path for instance has no color— a zero (null) value is returned.

```
draw      fullcircle withpen pencircle scaled 3mm ;
clip      currentpicture to fullcircle ;
setbounds currentpicture to fullcircle ;
```

n: 1 / length: 1 / stroked: false / clipped: true / bounded: false

```
draw      fullcircle withpen pencircle scaled 3mm ;
setbounds currentpicture to fullcircle ;
clip      currentpicture to fullcircle ;
```

n: 1 / length: 1 / stroked: false / clipped: false / bounded: true

```
clip      currentpicture to fullcircle ;
draw      fullcircle withpen pencircle scaled 3mm ;
setbounds currentpicture to fullcircle ;
```

n: 1 / length: 0 / stroked: false / clipped: true / bounded: false
n: 2 / length: 1 / stroked: true / clipped: false / bounded: false

```
clip      currentpicture to fullcircle ;
setbounds currentpicture to fullcircle ;
draw      fullcircle withpen pencircle scaled 3mm ;
```

n: 1 / length: 1 / stroked: false / clipped: false / bounded: true
n: 2 / length: 1 / stroked: true / clipped: false / bounded: false

```
setbounds currentpicture to fullcircle ;
clip      currentpicture to fullcircle ;
draw      fullcircle withpen pencircle scaled 3mm ;
```

n: 1 / length: 1 / stroked: false / clipped: true / bounded: false
n: 2 / length: 1 / stroked: true / clipped: false / bounded: false

```
setbounds currentpicture to fullcircle ;
draw      fullcircle withpen pencircle scaled 3mm ;
clip      currentpicture to fullcircle ;
```

n: 1 / length: 0 / stroked: false / clipped: false / bounded: true
n: 2 / length: 1 / stroked: true / clipped: false / bounded: false

The description lines were generated by the following loop:

```
n := 0 ;
for i within currentpicture : n := n + 1 ;
  label("n: "       & decimal           n & " / " &
        "length: "  & decimal   length  i & " / " &
        "stroked: " & condition stroked i & " / " &
        "clipped: " & condition clipped i & " / " &
        "bounded: " & condition bounded i , (0,-n*.5cm)) ;
endfor ;
```

If we have a textual picture, we can also ask for the text and font. Take the following picture:

```
picture p ;
p := "MetaFun" normalinfont "rm-lmr10" scaled 2 rotated 30 slanted .5 ;
p := p shifted (0,-ypart center p) ;
currentpicture := p ;
```

Here we can ask for:

```
label.rt("textpart: " & textpart p, (4cm,+0.25cm)) ;
label.rt("fontpart: " & fontpart p, (4cm,-0.25cm)) ;
```

and get:

textpart: MetaFun
fontpart: rm-lmr10

We use `normalinfont` instead of `infont` because in METAFUN this operator is overloaded and follows another route for including text.

If we're dealing with a path, the transformations have ended up in the path specification. If we have a text picture, we can explicitly ask for the transform components.

```
label.rt("xpart: "  & decimal xpart  p, (4cm,+1.25cm)) ;
label.rt("ypart: "  & decimal ypart  p, (4cm,+0.75cm)) ;
label.rt("xxpart: " & decimal xxpart p, (4cm,+0.25cm)) ;
label.rt("xypart: " & decimal xypart p, (4cm,-0.25cm)) ;
label.rt("yxpart: " & decimal yxpart p, (4cm,-0.75cm)) ;
label.rt("yypart: " & decimal yypart p, (4cm,-1.25cm)) ;
```

xpart: 0
ypart: -25.93834
xxpart: 2.23206
xypart: -0.13397
yxpart: 1
yypart: 1.73206

We will now define the `inverted` macro using these primitives. Because we have to return a picture, we cannot use `draw` and `fill` but need to use the low level operators. Because a picture can consist of more than one path, we need a temporary picture `pp`.

```
vardef inverted expr p =
  save pp ; picture pp ; pp := nullpicture ;
  for i within p :
```

```
    addto pp
    if stroked i or filled i :
      if filled i : contour else : doublepath fi pathpart i
      dashed dashpart i withpen penpart i
    else :
      also i
    fi
    withcolor white-(redpart i, greenpart i, bluepart i) ;
  endfor ;
  pp
enddef ;
```

We probably need to handle a few more border cases, but for general purposes, this macro works as expected.

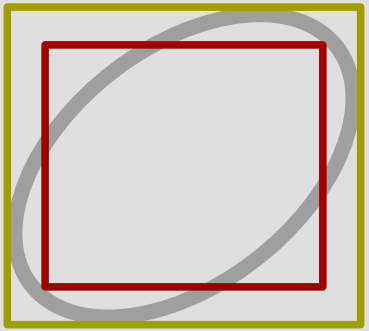
From the previous examples it may be clear that each picture has some associated data stored with it. From the `bounded` boolean test we can conclude that the bounding box is part of this data Internally METAPOST keeps track of two bounding boxes: the natural one, and the forced one. The forced one is actually a component of the picture which applies to all previously added graphics. You can calculate the bounding box from the `llcorner` and `urcorner` or if you like `ulcorner` and `lrcorner` and the METAFUN command `boundingbox` does so.

The four corners that make up the bounding box are either the natural ones, or the ones forced by `setbounds`. You can force METAPOST to report the natural ones by setting `truecorners` to 1. The next example demonstrates this feature.

```
pickup pencircle scaled 2mm ; path p, q ;
draw fullcircle
  scaled 4cm slanted .5 withcolor .625white ;
```

```
setbounds currentpicture to
  boundingbox currentpicture enlarged -5mm ;
interim truecorners := 0 ; p := boundingbox currentpicture ;
interim truecorners := 1 ; q := boundingbox currentpicture ;
pickup pencircle scaled 1mm ;
draw p withcolor .625red ;
draw q withcolor .625yellow ;
```

We use `interim` because `truecorners` is an internal METAPOST variable.

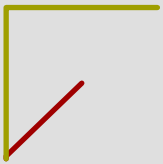


2.11 Pitfalls

When writing macros, you need to be careful in what operations apply to what object. There is for instance a difference between the following code:

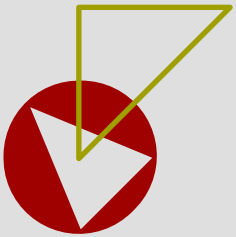
```
pickup pencircle scaled 2pt ;
```

```
draw  (0,0)--(0,1)--(1,1)  scaled 1cm withcolor .625 red ;
draw ((0,0)--(0,1)--(1,1)) scaled 2cm withcolor .625 yellow ;
```



The `scaled` operates on the previous expression which in the first case is the point `(1,1)` and in the second case the whole path.

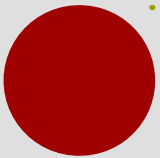
```
pickup pencircle scaled 2pt ;
draw  (0,0)--(0,1)--(1,1)--cycle  scaled 1cm withcolor .625 red ;
draw ((0,0)--(0,1)--(1,1)--cycle) scaled 2cm withcolor .625 yellow ;
```



Here the last element in the first case is not the cycle, and the next alternative does not help us much in discovering what is going on. (Well, at least something *is* going on, because the result seems to have some dimensions.)

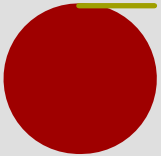
```
pickup pencircle scaled 2pt ;
```

```
draw  (1,1)--cycle  scaled 1cm withcolor .625 red ;
draw ((1,1)--cycle) scaled 1cm withcolor .625 yellow ;
```



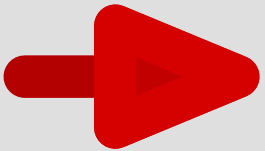
The next lines demonstrate that we're dealing with the dark sides of METAPOST, and from that we may conclude that in case of doubt it's best to add parenthesis when such fuzzy situations threaten to occur.

```
pickup pencircle scaled 2pt ;
draw  (0,1)--(1,1)--cycle  scaled 1cm withcolor .625 red ;
draw ((0,1)--(1,1)--cycle) scaled 1cm withcolor .625 yellow ;
```



There are more cases where the result may surprise you. Take the following code:

```
drawarrow ((0,0)--(10,0))
  withpen pencircle scaled 2pt
  withcolor red randomized (.4,.9) ;
currentpicture := currentpicture scaled 8 ;
```



The arrow is made up out of two pieces and each piece gets a different shade of red. This is because the attributes are collected and applied to each of the components that make up the arrow. Because for each component the attribute code is expanded again, we get two random colors. One way around this is to apply the color afterwards.

```
draw
  image (drawarrow ((0,0)--(10,0)) withpen pencircle scaled 2pt)
  scaled 8 withcolor red randomized (.4,.9) ;
```

Here the `image` macro creates a picture and as you can see, this provides a way to draw within a draw operation.

Once you see the benefits of `image`, you will use it frequently. Another handy (at first sight strange) macro is `hide`. You can use this in situations where you don't want code to interfere.

```
def mydraw text t =
  boolean error ; error := false ;
  def withpencil expr p = hide (error := true) enddef ;
```

```
    draw t ;
    if error : message "pencils are not supported here" fi ;
  enddef ;
  mydraw fullcircle scaled 10cm withpencil sharp ;
```

Here, setting the boolean normally interferes with the draw operation, but by hiding the assignment, this code becomes valid. This code will bring the message to your terminal and log file.

Once you start using expressions you have a good chance of encountering messages with regards to redundant expressions. The following code is for instance a recipe for problems:

```
z1 = (1,0) ; z1 = (2,0) ;
```

Changing the = into := helps, but this may not be what you want.

Because the z–variables are used frequently, they are reset each figure. You can also reset them yourself, using the `clearxy` macro. The METAFUN version clears all z–variables, unless you explictly specify what variables to reset.[8] If you want to play with this macro, see what happens when you run the following code:

```
show x0 ; z0 = (10,10) ;
show x0 ; x0 := whatever ; y0 := whatever ;
show x0 ; z0 = (20,20) ;
show x0 ; clearxy 0 ;
show x0 ; z0 = (30,30) ;
```

So, the following calls are all legal:

[8] This version resulted from a discussion on the METAFONT discussion list and is due to Bogusław Jackowski.

```
clearxy ; clearxy 1 ; clearxy 1, 8, 10 ;
```

Keep in mind that for each figure a full clear is done anyway. You should not confuse this command with `clearit`, which clears `currentpicture`.

2.12 TEX versus MetaPost

If you are defining your own TEX and METAPOST macros, you will notice that there are a couple of essential differences between the two macro languages. In TEX the following code is invalid.[9]

```
\def\fancyplied#1%
  {\ifnum#1=0
     \message{zero argument}%
   \fi
   \count0=#1 \multiply \count0 by \count0
   \count2=#1 \multiply \count2 by 2
   \count4=#1 \divide   \count4 by 2
   \advance \count0 by \count2
   \advance \count0 by \count4
   \count4 }
\hskip \fancyplied{3} pt
```

This is because TEX is very strict in what tokens it expects next. In METAPOST however, you can use `vardef`'d macros to hide nasty intermediate calculations.

[9] In ε-TEX the calculation can be done in less lines using a `\numexpr`.

```
vardef fancyplied expr x =
  if x=0 : message "x is zero" ; (x*x+2x+x/2)
enddef ;
a := a shifted (fancyplied 3pt,0) ;
```

Hiding intermediate calculations and manipulations is a very strong point of METAPOST.

Another important difference between both languages is the way grouping is implemented. Because TEX is dealing with a flow of information, strong grouping is a must and therefore part of the language. Occasionally you run into situations where you wished that you could reach over a group (for instance in order to pass a value).

In METAPOST grouping behaves quite different. First of all, it provides the mechanism that hides processing from the current flow. The previously mentioned `vardef` is implicitly grouped. Contrary to TEX, in METAPOST all assignments are global by default, even in a group. If you assign a variable inside a group it is persistent unless you first save the variable (or macro) using the `save` operator.

So, in the next code snippet, the value of `\value` inside the box is *no* but after the box is typeset, it will be *yes* again.

```
\def\value{yes} \hbox{\def\value{no}\value} \value
```

To make a value local in METAPOST, the following code is needed.

```
string value ; value := "yes" ;
def intermezzo
  begingroup ;
    save value ; string value ; value := "no" ;
  endgroup ;
```

```
enddef ;
```

Once you start writing your own METAPOST macros, you will appreciate this 'always global' behaviour. As with other differences between the two languages, they make sense if you look at what the programs are supposed to do.

2.13 Internals and Interims

Related to grouping is the internal numeric datatype. When numeric variables are defined as interim, you can quickly overload them inside a group.

```
newinternal mynumber ; mynumber := 1 ;

begingroup ; ... interim mynumber := 0 ; ... ; endgroup ;
```

You can only `interim` a variable if it is already defined using `newinternal`.

Among the METAPOST macros is one called `drawdot`. This macro is kind of redundant because, at least at first sight, you can use draw to achieve the same result. There is however a very subtle difference: a dot is slightly larger than a drawn point. We guess that it's about the device pixel, so you may not even notice it. It may even be due to differences in accuracy of the methods to render them.

```
pickup pencircle scaled 50pt ;
drawdot origin shifted (-120pt,0) ; draw origin shifted (-60pt,0) ;
drawdot origin ; draw origin withcolor white ;
setbounds currentpicture to boundingbox currentpicture enlarged 1pt ;
```

3 Embedded graphics

In addition to the `beginfig–endfig` method, there are other ways to define and include a METAPOST graphic. Each method has its advantages and disadvantages.

In the previous chapter we were still assuming that the graphic was defined in its own file. In this chapter we will introduce the interface between CONTEXT and METAPOST and demonstrate how the definitions of the graphics can be embedded in the document source.

3.1 Getting started

From now on, we will assume that you have CONTEXT running on your platform. Since PDF has full graphics support, we also assume that you use LUATEX in combination with CONTEXT MKIV, although most will also work with other engines and MKII. Since this document is not meant as a CONTEXT tutorial, we will limit this introduction to the basics needed to run the examples.

A simple document looks like:

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

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

```
context mytext
```

We will use color, and in MKIV color is enabled by default. If you don't want color you can tell CONTEXT, so

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

comes out in black and white.

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

3.2 External graphics

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

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

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

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

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

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

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

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

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

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

end.
```

This `star` macro will produce graphics like:

But, now that we have instant METAPOST available in LUATEX, there is no need for external images and we can collect them in libraries, as we will see later on.

3.3 Integrated graphics

An integrated graphic is defined in the document source or in a style definition file. The most primitive way of doing this is just inserting the code:

```
\startMPcode
  fill fullcircle scaled 200pt withcolor .625white ;
\stopMPcode
```

Such a graphic is used once at the spot where it is defined. In this document we also generate graphics while we finish a page, so there is a good chance that when we have constructed a graphic which will be called on the next page, the wrong graphic is placed.

For this reason there are are more convenient ways of defining and using graphics, which have the added advantage that you can predefine multiple graphics, thereby separating the definitions from the usage.

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

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

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

```
\useMPgraphic{name}
```

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

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

This definition is accompanied by:

```
\reuseMPgraphic{name}
```

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

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

Notice the use of `OverlayWidth` and `OverlayHeight`. These variables are set for each call to METAPOST.

After this we can say:

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

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

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

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

Now we can say:

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

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

```
\startuseMPgraphic{demo}
  draw unitsquare
    xscaled 5cm yscaled 1cm
```

```
    withpen pencircle scaled 2mm
    withcolor .625red ;
\stopuseMPgraphic
```

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


These graphics were placed with:

```
\hbox \bgroup
  \scale[width=5cm,height=1cm]{\useMPgraphic{demo}}\quad
  \scale[width=8cm,height=1cm]{\useMPgraphic{demo}}%
\egroup
```

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

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

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

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

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

The buttons will look like:

previous next index table of contents

Compare these with:

previous next index table of contents

Here the graphic was defined as:

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

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

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

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

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

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

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

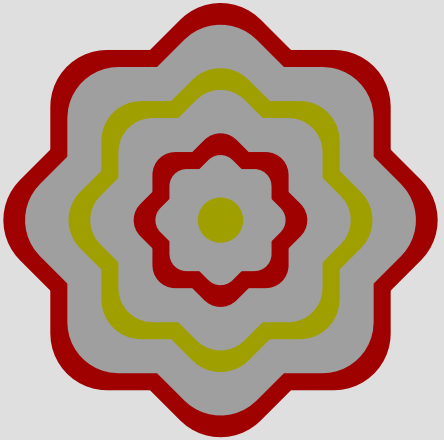


Figure 3.1 The logo is defined in the file `mylogos.mp` as figure 21 and processed by means of the `mprun` method.

Optionally you can specify a name and an instance. This has the advantage that the graphics don't interfere with the regular inline graphics. Here the instance used is `extrafun` and the name where the run is stored is `mydemo`.

```
\startMPrun{mydemo}
    input mfun-mrun-demo.mp ;
\stopMPrun

\placefigure
  {An external file can have multiple graphics. Here we show a few
```

```
 images that we used to use on the \PRAGMA\ \CONTEXT\ website.}
{\startcombination[2*2]
   {\externalfigure[mprun:extrafun::mydemo.1][height=6cm]} {downloads}
   {\externalfigure[mprun:extrafun::mydemo.2][height=6cm]} {links}
   {\externalfigure[mprun:extrafun::mydemo.3][height=6cm]} {mirrors}
   {\externalfigure[mprun:extrafun::mydemo.4][height=6cm]} {team}
 \stopcombination}
```

Keep in mind that the whole file will be processed (using the built in library) in order to get one graphic. Normally this is no big deal.

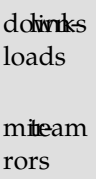


Figure 3.2 An external file can have multiple graphics. Here we show a few images that we used to use on the PRAGMA ADE CONTEXT website.

3.4 Using METAFUN but not CONTEXT

If you don't want to use CONTEXT but still want to use METAFUN, a rather convenient method is the following. Create a file that

```
\startMPpage
    % Your mp code goes here. You can use the textext
    % macro as discussed later to deal with typeset text.
\stopMPpage
```

When you process that file with the `context` command you will get a PDF file that you can include in any application that can embed a PDF image. In this case your exposure to CONTEXT is minimal.

3.5 Graphic buffers

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

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

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

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

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

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

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

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

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

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

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

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

```
\startlinecorrection[blank]
\processMPbuffer[red,graphic]
\stoplinecorrection
```

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

```
\startlinecorrection[blank]
\processMPbuffer[yellow,graphic]
```

```
\stoplinecorrection
```

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

3.6 Communicating color

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

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

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

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

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

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

```
\definecolor[darkyellow][y=.625] % a CMYK color
```

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

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

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

\useMPgraphic{color demo}
```

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

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

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

\definecolor[cmydarkyellow] [y=.625]
```

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

Figure 3.3 demonstrates what happens when we multiply colors by a factor. Since we are not dealing with real CMYK colors, multiplication gives different results for CMYK colors passed as `\MPcolor`.

Figure 3.3 All kinds of yellow.

So, `.625red` is the same as `[r=.5]`, but `.625yellow` is not the same as `[y=.5]`, but matches `[r=.5,g=.5]`. **Figure 3.4** shows the pure and half reds.

In order to prevent problems, we advise you to stick to RGB color specifications when possible. That way you not only prevent conversion problems, but also get more predictable results on printing and viewing devices. However, reality demands that sometimes CMYK colors are used, so how can we deal with that?

In the METAFUN macro collection there is a macro `cmyk` that takes four arguments, representing the cyan, magenta, yellow, and black component.

Figure 3.4 Some kinds of red.

```
fill fullsquare xyscaled (10cm,1cm) withcolor cmyk(1,0,.3,.3) ;
```

If you take a close look at the numbers, you will notice that the cyan component results in a 100% ink contribution. You will also notice that 30% black ink is added. This means that we cannot safely convert this color to RGB ($r = 1 - c - k < 0$) without losing information. Nevertheless the previous blue bar is presented all right. This is due to the fact that in METAFUN the CMYK colors are handled as they should, even when METAPOST does not support this color model.

If you use this feature independent of CONTEXT, you need to enable it by setting `cmykcolors` to `true`. You have to convert the resulting graphic to PDF by using for instance the `mptopdf` suite.

In CONTEXT you can influence this conversion by changing parameters related to color handling:

```
\setupcolors[cmyk=yes,mpcmyk=no]
```

Unless you know what you are doing, you don't have to change the default settings (both `yes`). In the CONTEXT reference manual you can also read how color reduction can be handled.

Special care should be paid to gray scales. Combining equal quantities of the three color inks will not lead to a gray scale, but to a muddy brown shade.

```
fill fullsquare xyscaled (10cm,  2cm) withcolor .5white ;
fill fullsquare xyscaled ( 6cm,1.5cm) withcolor cmyk(.5,.5,.5,0) ;
fill fullsquare xyscaled ( 2cm,  1cm) withcolor cmyk(0,0,0,.5) ;
```

In **figure 3.5 to 3.7** you can see some more colors defined in the CMYK color space. When you display the screen version of this document, you will notice that the way colors are displayed can differ per viewer. This is typical for CMYK colors and has to do with the fact that some assumptions are made with respect to the (print) medium.

c=1 y=.3 k=.3 c=.9 y=.15 c=.25 y=.8 c=.45 y=.1

Figure 3.5 CMYK support disabled, conversion to RGB.

Figure 3.6 CMYK support enabled, no support in METAPOST.

Figure 3.7 CMYK support enabled, no conversion to RGB, support in METAPOST

3.7 Common definitions

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

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

The following is only true for CONTEXT MKII! Users of MKIV can skip this section.

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

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

```
\runMPgraphicstrue
```

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

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

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

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

```
\recycleMPslotstrue
```

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

3.8 One page graphics

An advantage of using CONTEXT to make your METAPOST graphics is you don't have to bother about specials, font inclusion and all those nasty things that can spoil a good day. An example of such a graphic is the file mfun-800 that resides on the computer of the author.

```
% language-uk
%
% copyright=pragma-ade readme=readme.pdf licence=cc-by-nc-sa

\setupMPpage
  [offset=1pt,
   background=color,
   backgroundcolor=gray]

\definecolor [gray]   [s=.625]
\definecolor [red]    [r=.625]
\definecolor [yellow] [r=.625,g=.625]

\startuseMPgraphic{test}
    fill fullsquare rotated 45 scaled 4cm withcolor \MPcolor{yellow} ;
\stopuseMPgraphic

\starttext

\startMPpage
    \includeMPgraphic{test}
    fill fullcircle scaled 3cm withcolor \MPcolor{red} ;
\stopMPpage

\stoptext
```

Given that CONTEXT is present on your system, you can process this file with:

```
context mfun-800
```

You can define many graphics in one file. Later you can include individual pages from the resulting PDF file in your document:

```
\placefigure
  {A silly figure, demonstrating that stand||alone||graphics
   can be made.}
  {\externalfigure[mfun-800][page=1]}
```

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

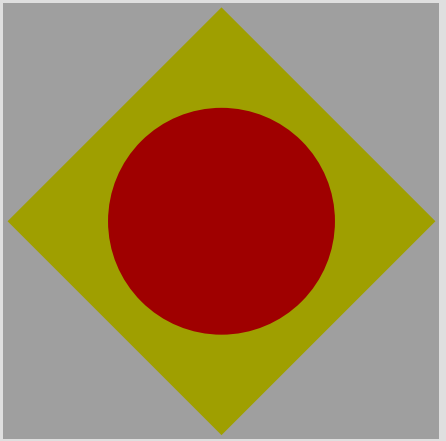


Figure 3.8 A silly figure, demonstrating that stand–alone–graphics can be made.

3.9

Managing resources

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

Special effects, like shading, are supported by dedicated METAPOST modules. These are included in the CONTEXT distribution and will be discussed later in **chapter 8**.

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

You may wonder what the advantage is of weighted gray conversion. **Figure 3.9** shows the difference between natural colors, weighted gray scales and straightforward, non–weighted, gray scales.

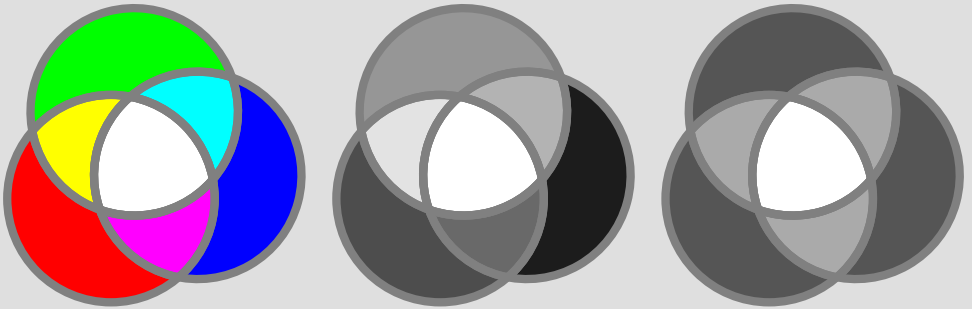


Figure 3.9 The advantage of weighted gray over linear gray.

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

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

Section 8.5 introduces the `grayed` operation that you can use to convert a colored picture into a gray one. This macro uses the same conversion method as mentioned here.

4 Enhancing the layout

One of the most powerful and flexible commands of CONTEXT is `\framed`*. We can use the background features of this command to invoke and position graphics that adapt themselves to the current situation. Once understood, overlays will become a natural part of the CONTEXT users toolkit.*

4.1 Overlays

Many CONTEXT commands support overlays. The term *overlay* is a bit confusing, since such an overlay in most cases will lay under the text. However, because there can be many layers on top of each other, the term suits its purpose.

When we want to put a METAPOST graphic under some text, we go through a three step process. First we define the graphic itself:

```
\startuniqueMPgraphic{demo circle}
  path p ;
  p := fullcircle xscaled \overlaywidth yscaled \overlayheight ;
  fill p withcolor .85white ;
  draw p withpen pencircle scaled 2pt withcolor .625red ;
\stopuniqueMPgraphic
```

This graphic will adapt itself to the width and height of the overlay. Both `\overlaywidth` and `\overlayheight` are macros that return a dimension followed by a space. The next step is to register this graphic as an overlay.

```
\defineoverlay[demo circle][\uniqueMPgraphic{demo circle}]
```

We can now use this overlay in any command that provides the `\framed` functionality. Since this graphic is defined as unique, CONTEXT will try to reuse already calculated and embedded graphics when possible.

```
\framed[background=demo circle]{This text is overlayed.}
```

The background can be set to `color`, `screen`, an overlay identifier, like `demo circle`, or a comma separated list of those.

This text is overlayed.

The `\framed` command automatically draws a ruled box, which can be quite useful when debugging a graphic. However, in this case we want to turn the frame off.

```
\framed
  [background=demo circle,frame=off]
  {This text is overlayed.}
```

This text is overlayed.

In this case, it would have made sense to either set the `offset` to a larger value, or to set `backgroundoffset`. In the latter case, the ellipse is positioned outside the frame.

The difference between the three offsets `offset`, `frameoffset` and `backgroundoffset` is demonstrated in **figure 4.1**. While the `offset` is added to the (natural or specified) dimensions of the content of the box, the other two are applied to the frame and background and don't add to the dimensions.

In the first row we only set the `offset`, while in the second row, the (text) offset is set to `3pt`. When not specified, the `offset` has a comfortable default value of `.25ex` (some 25% of the height of an x).

```
\setupframed
  [width=.3\textwidth,
   background=demo circle]
\startcombination[3*3]
  {\framed[offset=none]          {\TeX}} {\tt offset=none}
  {\framed[offset=overlay]       {\TeX}} {\tt offset=overlay}
  {\framed[offset=0pt]           {\TeX}} {\tt offset=0pt}
  {\framed[offset=1pt]           {\TeX}} {\tt offset=1pt}
  {\framed[offset=2pt]           {\TeX}} {\tt offset=2pt}
  {\framed[offset=4pt]           {\TeX}} {\tt offset=4pt}
  {\framed[offset=3pt]           {\TeX}} {\tt offset=3pt}
  {\framed[frameoffset=3pt]      {\TeX}} {\tt frameoffset=3pt}
  {\framed[backgroundoffset=3pt]{\TeX}} {\tt backgroundoffset=3pt}
\stopcombination
```

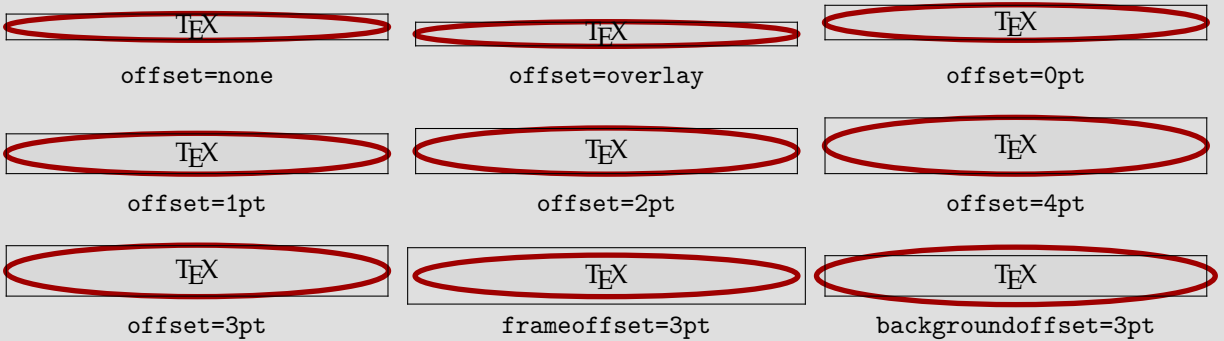


Figure 4.1 The three offsets.

As the first row in **figure 4.1** demonstrates, instead of a value, one can pass a keyword. The `overlay` keyword implies that there is no offset at all and that the lines cover the content. With `none` the frame is drawn tight around the content. When the offset is set to `0pt` or more, the text is automatically set to at least the height of a line. You can turn this feature of by saying `strut=off`. More details can be found in the CONTEXT manual.

In **figure 4.2** we have set offset to `3pt`, `frameoffset` to `6pt` and `backgroundoffset` to `9pt`. Both the frame and background offset are sort of imaginary, since they don't contribute to the size of the box.

```
\ruledhbox
  {\framed
     [offset=3pt,frameoffset=6pt,backgroundoffset=9pt,
      background=screen,backgroundscreen=.85]
     {Welcome in the hall of frame!}}
```

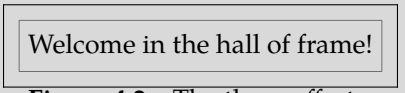


Figure 4.2 The three offsets.

4.2 Overlay variables

The communication between TEX and embedded METAPOST graphics takes place by means of some macros.

overlay status macro	meaning
`\overlaywidth`	the width of the graphic, as calculated from the actual width and background offset
`\overlayheight`	the height of the graphic, as calculated from the actual height, depth and background offset

`\overlaydepth`	the depth of the graphic, if available
`\overlaycolor`	the background color, if given
`\overlaylinecolor`	the color of the frame
`\overlaylinewidth`	the width of the frame

The dimensions of the overlay are determined by dimensions of the background, which normally is the natural size of a `\framed`. When a background offset is specified, it is added to `overlayheight` and `overlaywidth`.

Colors can be converted by `\MPcolor` and in addition to the macros mentioned, you can use all macros that expand into a dimension or dimen register prefixed by the TEX primitive `\the` (this and other primitives are explained in "The TEXbook", by Donald Knuth).

4.3 Stacking overlays

A background can be a gray scale (`screen`), a color (`color`), a previously defined overlay identifier, or any combination of these. The next assignments are therefore valid:

```
\framed[background=color,backgroundcolor=red]{...}
\framed[background=screen,backgroundscreen=.8]{...}
\framed[background=circle]{...}
\framed[background={color,cow},backgroundcolor=red]{...}
\framed[background={color,cow,grid},backgroundcolor=red]{...}
```

In the last three cases of course you have to define `circle`, `cow` and `grid` as overlay. These items are packed in a comma separated list, which is to be surrounded by `{}`.

4.4 Foregrounds

The overlay system is actually a system of layers. Sometimes we are confronted with a situation in which we want the text behind another layer. This can be achieved by explicitly placing the foreground layer, as in **figure 4.3**.

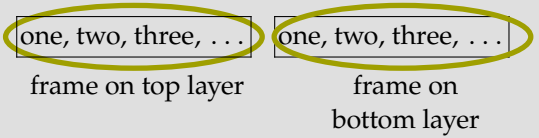


Figure 4.3 Foreground material moved backwards.

The graphic layer is defined as follows:

```
\startuniqueMPgraphic{backfore}
  draw fullcircle
    xscaled \overlaywidth yscaled \overlayheight
    withpen pencircle scaled 2pt
    withcolor .625yellow ;
\stopuniqueMPgraphic

\defineoverlay[backfore][\uniqueMPgraphic{backfore}]
```

The two framed texts have a slightly different definition. The leftmost graphic is defined as:

```
\framed
  [background=backfore,backgroundoffset=4pt]
```

```
  {one, two, three, \unknown}
```

The rightmost graphic is specified as:

```
\framed
  [background={foreground,backfore},backgroundoffset=4pt]
  {one, two, three, \unknown}
```

The current values of the frame color and frame width are passed to the overlay. It often makes more sense to use colors defined at the document level, if only to force consistency.

```
\startuniqueMPgraphic{super ellipse}
  path p ; p := unitsquare
    xscaled \overlaywidth yscaled \overlayheight
    superellipsed .85 ;
  pickup pencircle scaled \overlaylinewidth ;
  fill p withcolor \MPcolor{\overlaycolor} ;
  draw p withcolor \MPcolor{\overlaylinecolor} ;
\stopuniqueMPgraphic

\defineoverlay[super ellipse][\uniqueMPgraphic{super ellipse}]
```

This background demonstrates that a super ellipse is rather well suited as frame.

```
\framed
  [background=super ellipse,
   frame=off,
   width=3cm,
```

```
    align=middle,
    framecolor=darkyellow,
    rulethickness=2pt,
    backgroundcolor=darkgray]
  {\white This is a\\Super Ellipsed\\sentence.}
```

Such a super ellipse looks quite nice and is a good candidate for backgrounds, for which the superness should be at least .85.

This is a
Super Ellipsed
sentence.

4.5 Typesetting graphics

I have run into people who consider it kind of strange when you want to use TEX for non–mathematical typesetting. If you agree with them, you may skip this section with your eyes closed.

One of the CONTEXT presentation styles (number 15, tagged as balls) stepwise builds screens full of sentences, quotes or concepts, packages in balloons and typesets them as a paragraph. We will demonstrate that TEX can typeset graphics using the following statement.

“ know,mayyouASE METAFUN.enjoyitmayborn,isbabyaifandso,ifbutrelationship,ahavetheyifunclearstillisItlioness.abyrepresentedisMETAFONTwhilelion,aisambassadorTEX’s”

The low level CONTEXT macro `\processwords` provides a mechanism to treat the individual words of its argument. The macro is called as follows:

```
\processwords{As you may know, \TEX's ambassador is a lion,
while {\METAFONT} is represented by a lioness. It is still
unclear if they have a relationship, but if so, and if a
baby is born, may it enjoy \METAFUN.}
```

In order to perform a task, you should also define a macro `\processword`, which takes one argument. The previous quote was typeset with the following definition in place:

```
\def\processword#1{#1}
```

A slightly more complicated definition is the following:

```
\def\processword#1{\noindent\framed{#1}\space}
```

We now get:

As you may know, ETEX's ambassador is a lion, while METAFONT is represented by a lioness. It is still unclear if they have a relationship, but if so, and if a baby is born, may it enjoy METAFUN.

If we can use `\framed`, we can also use backgrounds.

```
\def\processword#1%
  {\noindent\framed[frame=off,background=lions]{#1} }
```

We can add a supperellipsed frame using the following definition:

```
\startuniqueMPgraphic{lions a}
```

```
  path p ; p := fullsquare
   xyscaled (\overlaywidth,\overlayheight) superellipsed .85 ;
  pickup pencircle scaled 1pt ;
  fill p withcolor .850white ; draw p withcolor .625yellow ;
\stopuniqueMPgraphic

\defineoverlay[lions][\uniqueMPgraphic{lions a}]
```

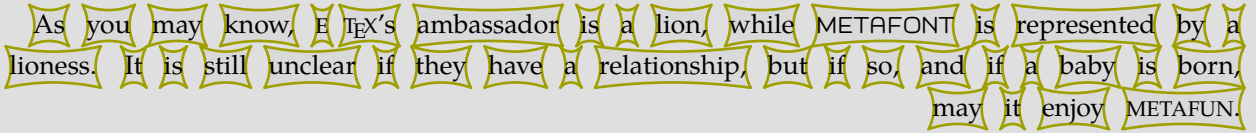
As you may know, ℰ TEX's ambassador is a lion, while METAFONT is represented by a lioness. It is still unclear if they have a relationship, but if so, and if a baby is born, may it enjoy METAFUN.

```
\startuseMPgraphic{lions b}
  path p ; p := fullsquare
    xyscaled (\overlaywidth,\overlayheight) randomized 5pt ;
  pickup pencircle scaled 1pt ;
  fill p withcolor .850white ; draw p withcolor .625yellow ;
\stopuseMPgraphic

\defineoverlay[lions][\uniqueMPgraphic{lions b}]
```

As you may know, ℰ TEX's ambassador is a lion, while METAFONT is represented by a lioness. It is still unclear if they have a relationship, but if so, and if a baby is born, may it enjoy METAFUN.

```
\startuniqueMPgraphic{lions c}
```

```
  path p ; p := fullsquare
    xyscaled (\overlaywidth,\overlayheight) squeezed 2pt ;
  pickup pencircle scaled 1pt ;
  fill p withcolor .850white ; draw p withcolor .625yellow ;
\stopuniqueMPgraphic

\defineoverlay[lions][\uniqueMPgraphic{lions c}]
```



These paragraphs were typeset with the following settings.

```
\setupalign[broad, right]  % == \veryraggedright
\setupalign[broad, middle] % == \veryraggedcenter
\setupalign[broad, left]   % == \veryraggedleft
```

The `broad` increases the raggedness. We defined three different graphics (a, b and c) because we want some to be unique, which saves some processing. Of course we don't reuse the random graphics. In the definition of `\processword` we have to use `\noindent` because otherwise TEX will put each graphic on a line of its own. Watch the space at the end of the macro.

4.6 Graphics and macros

Because TEX's typographic engine and METAPOST's graphic engine are separated, interfacing between them is not as natural as you may expect. In CONTEXT we have tried to integrate them as much as possible, but using

the interface is not always as convenient as it should be. What method you follow, depends on the problem at hand.

The official METAPOST way to embed TEX code into graphics is to use `btex ... etex`. As soon as CONTEXT writes the graphic data to the intermediate METAPOST file, it looks for these commands. When it has encountered an `etex`, CONTEXT will make sure that the text that is to be typeset by TEX is *not* expanded. This is what you may expect, because when you would embed those commands in a stand–alone graphic, they would also not be expanded, if only because METAPOST does not know TEX. With expansion we mean that TEX commands are replaced by their meaning (which can be quite extensive).

Users of CONTEXT MKIV can skip the next paragraph.

When METAPOST sees a `btex` command, it will consult a so called `mpx` file. This file holds the METAPOST representation of the text typeset by TEX. Before METAPOST processes a graphic definition file, it first calls another program that filters the `btex` commands from the source file, and generates a TEX file from them. This file is then processed by TEX, and after that converted to a `mpx` file. In CONTEXT we let TEXEXEC take care of this whole process.

Because the `btex ... etex` commands are filtered from the raw METAPOST source code, they cannot be part of macro definitions and loop constructs. When used that way, only one instance would be found, while in practice multiple instances may occur.

This drawback is overcome by METAFUN's `textext` command. This command still uses `btex ... etex` but writes these commands to a separate job related file each time it is used.[10] After the first METAPOST run, this file is merged with the original file, and METAPOST is called again. So, at the cost of an additional run, we can use text typeset by TEX in a more versatile way. Because METAPOST runs are much faster than TEX runs, the price to

[10] It took the author a while to find out that there is a METAPOST module called `tex.mp` that provides a similar feature, but with the disadvantage that each text results in a call to TEX. Each text goes into a temporary file, which is then included and results in METAPOST triggering TEX.

pay in terms of run time is acceptable. Unlike `btex ... etex`, the TEX code in `textext` command is expanded, but as long as CONTEXT is used this is seldom a problem, because most commands are somewhat protected.

If we define a graphic with text to be typeset by TEX, there is a good chance that this text is not frozen but passes as argument. A TEX–like solution for passing arbitrary content to such a graphic is the following:

```
\def\RotatedText#1#2%
  {\startuseMPgraphic{RotatedText}
     draw btex #2 etex rotated #1 ;
   \stopuseMPgraphic
   \useMPgraphic{RotatedText}}
```

This macro takes two arguments (the # identifies an argument):

```
\RotatedText{15}{Some Rotated Text}
```

The text is rotated over 15 degrees about the origin in a counterclockwise direction.

In CONTEXT we seldom pass settings like the angle of rotation in this manner. You can use `\setupMPvariables` to set up graphic–specific variables. Such a variable can be accessed with `\MPvar`.

```
\setupMPvariables[RotatedText][rotation=90]

\startuseMPgraphic{RotatedText}
  draw btex Some Text etex rotated \MPvar{rotation} ;
\stopuseMPgraphic
```

An example:

```
\RotatedText{-15}{Some Rotated Text}
```

In a similar fashion we can isolate the text. This permits us to use the same graphics with different settings.

```
\setupMPvariables[RotatedText][rotation=270]

\setMPtext{RotatedText}{Some Text}

\startuseMPgraphic{RotatedText}
  draw \MPbetex{RotatedText} rotated \MPvar{rotation} ;
\stopuseMPgraphic
```

This works as expected:

```
\RotatedText{165}{Some Rotated Text}
```

It is now a small step towards an encapsulating macro (we assume that you are familiar with TEX macro definitions).

```
\def\RotatedText[#1]#2%
  {\setupMPvariables[RotatedText][#1]%
```

```
    \setMPtext{RotatedText}{#2}%
    \useMPgraphic{RotatedText}}

\setupMPvariables[RotatedText][rotation=90]

\startuseMPgraphic{RotatedText}
  draw \MPbetex{RotatedText} rotated \MPvar{rotation} ;
\stopuseMPgraphic
```

Again, we default to a 90 degrees rotation, and pass both the settings and text in an indirect way. This method permits you to build complicated graphics and still keep macros readable.

```
\RotatedText[rotation=240]{Some Rotated Text}
```

You may wonder why we don't use the variable mechanism to pass the text. The main reason is that the text mechanism offers a few more features, one of which is that it passes the text straight on, without the danger of unwanted expansion of embedded macros. Using `\setMPtext` also permits you to separate TEX and METAPOST code and reuse it multiple times (imagine using the same graphic in a section head command).

There are three ways to access a text defined with `\setMPtext`. Imagine that we have the following definitions:

```
\setMPtext {1} {Now is this \TeX\ or not?}
```

```
\setMPtext {2} {See what happens here.}
\setMPtext {3} {Text streams become pictures.}
```

The \MPbetex macro returns a btex ... etex construct. The \MPstring returns the text as a METAPOST string, between quotes. The raw text can be fetched with \MPtext.

```
\startMPcode
  picture p ; p :=           \MPbetex {1}          ;
  picture q ; q :=  textext( \MPstring{2}        ) ;
  picture r ; r := thelabel("\MPtext  {3}",origin) ;

  for i=p, boundingbox p : draw i withcolor .625red    ; endfor ;
  for i=q, boundingbox q : draw i withcolor .625yellow ; endfor ;
  for i=r, boundingbox r : draw i withcolor .625white  ; endfor ;

  currentpicture := currentpicture scaled 2 ;
  draw origin withpen pencircle scaled 5.0mm withcolor white ;
  draw origin withpen pencircle scaled 2.5mm withcolor black ;
  draw boundingbox currentpicture withpen pencircle scaled .1mm dashed evenly ;
\stopMPcode
```

The first two lines return text typeset by TEX, while the last line leaves this to METAPOST.

If you watch closely, you will notice that the first (red) alternative is positioned with the baseline on the origin.

```
\startMPcode
```

```
    picture p ; p :=                  \MPbetex {1}          ;
    picture q ; q :=  textext.origin( \MPstring{2}        ) ;
    picture r ; r := thelabel.origin("\MPtext  {3}",origin) ;

    for i=p, boundingbox p : draw i withcolor .625red    ; endfor ;
    for i=q, boundingbox q : draw i withcolor .625yellow ; endfor ;
    for i=r, boundingbox r : draw i withcolor .625white  ; endfor ;

    currentpicture := currentpicture scaled 2 ;
    draw origin withpen pencircle scaled 5.0mm withcolor white ;
    draw origin withpen pencircle scaled 2.5mm withcolor black ;
    draw boundingbox currentpicture withpen pencircle scaled .1mm dashed evenly ;
  \stopMPcode
```

This draws:

This picture demonstrates that we can also position `textext`'s and `label`'s on the baseline. For this purpose the repertoire of positioning directives (`top`, `lft`, etc.) is extended with an `origin` directive. Another extra positioning directive is `raw`. This one does not do any positioning at all.

```
  picture q ; q :=  textext.origin( \MPstring{2}        ) ;
  picture r ; r := thelabel.origin("\MPtext  {3}",origin) ;
```

We will now apply this knowledge of text inclusion in graphics to a more advanced example. The next definitions are the answer to a question on the CONTEXT mailinglist with regards to framed texts with titles.

Zapf (1)

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.

In this example, the title is positioned on top of the frame. Title and text are entered as:

```
\FrameTitle{Zapf (1)}

\StartFrame
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.
\StopFrame
```

The implementation is not that complicated and uses the frame commands that are built in CONTEXT. Instead of letting TEX draw the frame, we use METAPOST, which we also use for handling the title. The graphic is defined as follows:

```
\startuseMPgraphic{FunnyFrame}
  picture p ; numeric w, h, o ;
  p := textext.rt(\MPstring{FunnyFrame}) ;
  w := OverlayWidth ; h := OverlayHeight ; o := BodyFontSize ;
  p := p shifted (2o,h-ypart center p) ; draw p ;
```

```
    drawoptions (withpen pencircle scaled 1pt withcolor .625red) ;
    draw (2o,h)--(0,h)--(0,0)--(w,0)--(w,h)--(xpart urcorner p,h) ;
    draw boundingbox p ;
    setbounds currentpicture to unitsquare xyscaled(w,h) ;
  \stopuseMPgraphic
```

Because the framed title is partly positioned outside the main frame, and because the main frame will be combined with the text, we need to set the boundingbox explicitly. This is a way to create so called free figures, where part of the figure lays beyond the area that is taken into account when positioning the graphic. The shift

```
... shifted (2o,h-ypart center p)
```

ensures that the title is vertically centered over the top line of the main box.

The macros that use this graphic combine some techniques of defining macros, using predefined CONTEXT classes, and passing information to graphics.

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

\defineframedtext[FunnyText][frame=off,background=FunnyFrame]

\def\StartFrame{\startFunnyText}
\def\StopFrame {\stopFunnyText }

\def\FrameTitle#1%
  {\setMPtext{FunnyFrame}{\hbox spread 1em{\hss\strut#1\hss}}}

\setMPtext{FunnyFrame}{} % initialize the text variable
```

There is a little bit of low level TEX code involved, like a horizontal box (\hbox) that stretches one em–space beyond its natural size (spread 1em) with a centered text (two times \hss). Instead of applying this spread, we could have enlarged the frame on both sides.

In the previous graphic we calculated the big rectangle taking the small one into account. This was needed because we don't use a background fill. The next definition does, so there we can use a more straightforward approach by just drawing (and filling) the small rectangle on top of the big one.

```
\startuseMPgraphic{FunnyFrame}
  picture p ; numeric o ; path a, b ; pair c ;
  p := textext.rt(\MPstring{FunnyFrame}) ;
  a := unitsquare xyscaled(OverlayWidth,OverlayHeight) ;
  o := BodyFontSize ;
  p := p shifted (2o,OverlayHeight-ypart center p) ;
  drawoptions (withpen pencircle scaled 1pt withcolor .625red) ;
  b := a randomized (o/2) ;
  fill b withcolor .85white ; draw b ;
  b := (boundingbox p) randomized (o/8) ;
  fill b withcolor .85white ; draw b ;
  draw p withcolor black;
  setbounds currentpicture to a ;
 \stopuseMPgraphic
```

Zapf (2)

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.

Because we use a random graphic, we cannot guarantee beforehand that the left and right edges of the small shape touch the horizontal lines in a nice way. The next alternative displaces the small shape plus text so that its center lays on the line. On the average, this looks better.

```
\startuseMPgraphic{FunnyFrame}
  picture p ; numeric o ; path a, b ; pair c ;
  p := textext.rt(\MPstring{FunnyFrame}) ;
  a := unitsquare xyscaled(OverlayWidth,OverlayHeight) ;
  o := BodyFontSize ;
  p := p shifted (2o,OverlayHeight-ypart center p) ;
  drawoptions (withpen pencircle scaled 1pt withcolor .625red) ;
  b := a randomized (o/2) ;
  fill b withcolor .85white ; draw b ;
  c := center p ;
  c := b intersectionpoint (c shifted (0,-o)--c shifted(0,o)) ;
  p := p shifted (c-center p) ;
  b := (boundingbox p) randomized (o/8) ;
  fill b withcolor .85white ; draw b ;
  draw p withcolor black;
  setbounds currentpicture to a ;
 \stopuseMPgraphic
```

Zapf (2)

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.

Yet another definition uses super ellipsed shapes instead of random ones. We need a high degree of superness (.95) in order to make sure that the curves don't touch the texts.

```
\startuseMPgraphic{FunnyFrame}
  picture p ; numeric o ; path a, b ; pair c ;
  p := textext.rt(\MPstring{FunnyFrame}) ;
  o := BodyFontSize ;
  a := unitsquare xyscaled(OverlayWidth,OverlayHeight) ;
  p := p shifted (2o,OverlayHeight-ypart center p) ;
  drawoptions (withpen pencircle scaled 1pt withcolor .625red) ;
  b := a superellipsed .95 ;
  fill b withcolor .85white ; draw b ;
  b := (boundingbox p) superellipsed .95 ;
  fill b withcolor .85white ; draw b ;
  draw p withcolor black ;
  setbounds currentpicture to a ;
\stopuseMPgraphic
```

Zapf (3)

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.

There are quite some hard coded values in these graphics, like the linewidths, offsets and colors. Some of these can be fetched from the `\framed` environment either by using TEX macros or dimensions, or by using

their METAFUN counterparts. In the following table we summarize both the available METAPOST variables and their TEX counterparts. They may be used interchangeably.

METAPOST variable	TEX command	meaning
OverlayWidth	`\overlaywidth`	current width
OverlayHeight	`\overlayheight`	current height
OverlayDepth	`\overlayheight`	current depth (often zero)
OverlayColor	`\MPcolor{\overlaycolor}`	background color
OverlayLineWidth	`\overlaylinewidth`	width of the frame
OverlayLineColor	`\MPcolor{}`	color of the frame
BaseLineSkip	`\the\baselineskip`	main line distance
LineHeight	`\the\baselineskip`	idem
BodyFontSize	`\the\bodyfontsize`	font size of the running text
StrutHeight	`\strutheight`	space above the baseline
StrutDepth	`\strutdepth`	space below the baseline
ExHeight	`1ex`	height of an x
EmWidth	`1em`	width of an m-dash

```
\startuseMPgraphic{FunnyFrame}
  picture p ; numeric o ; path a, b ; pair c ;
  p := textext.rt(\MPstring{FunnyFrame}) ;
  o := BodyFontSize ;
  a := unitsquare xyscaled(OverlayWidth,OverlayHeight) ;
  p := p shifted (2o,OverlayHeight-ypart center p) ;
  pickup pencircle scaled OverlayLineWidth ;
  b := a superellipsed .95 ;
```

```
  fill b withcolor OverlayColor ;
  draw b withcolor OverlayLineColor ;
  b := (boundingbox p) superellipsed .95 ;
  fill b withcolor OverlayColor ;
  draw b withcolor OverlayLineColor ;
  draw p withcolor black ;
  setbounds currentpicture to a ;
\stopuseMPgraphic
```

Zapf (3)

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.

We used the following command to pass the settings:

```
\setupframedtexts
  [FunnyText]
  [backgroundcolor=lightgray,
   framecolor=darkred,
   rulethickness=2pt,
   offset=\bodyfontsize,
   before={\blank[big,medium]},
   after={\blank[big]},
   width=\textwidth]
```

In a real implementation, we should also take care of some additional spacing before the text, which is why we have added more space before than after the framed text.

We demonstrated that when defining graphics that are part of the layout, you need to have access to information known to the typesetting engine. Take **figure 4.4**. The line height needs to match the font and the two thin horizontal lines should match the *x*–height. We also need to position the baseline, being the lowest one of a pair of lines, in such a way that it suits the proportions of the line as specified by the strut. A strut is an imaginary large character with no width. You should be aware of the fact that while TEX works its way top–down, in METAPOST the origin is in the lower left corner.

```
\usetypescript[serif][chorus]

\definefont[SomeHandwriting][TeXGyreChorus-MediumItalic*default at 12pt]

\start \SomeHandwriting\setstrut

\startMPpage
  StartPage ;
    path p ; numeric l, n ; l := 1.5LineHeight ; n := 0 ;
    p := origin shifted (l,0) -- origin shifted (PaperWidth-l,0) ;
    for i=PaperHeight-l step -l until l :
      n := n + 1 ;
      fill         p shifted (0,i+StrutHeight) --
           reverse p shifted (0,i-StrutDepth ) -- cycle
           withcolor .85white ;
      draw p shifted (0,i) withpen pencircle scaled .25pt withcolor .5white ;
      draw p shifted (0,i+ExHeight) withpen pencircle scaled .25pt withcolor .5white ;
      draw textext.origin("\strut How are those penalty lines called in english?
```

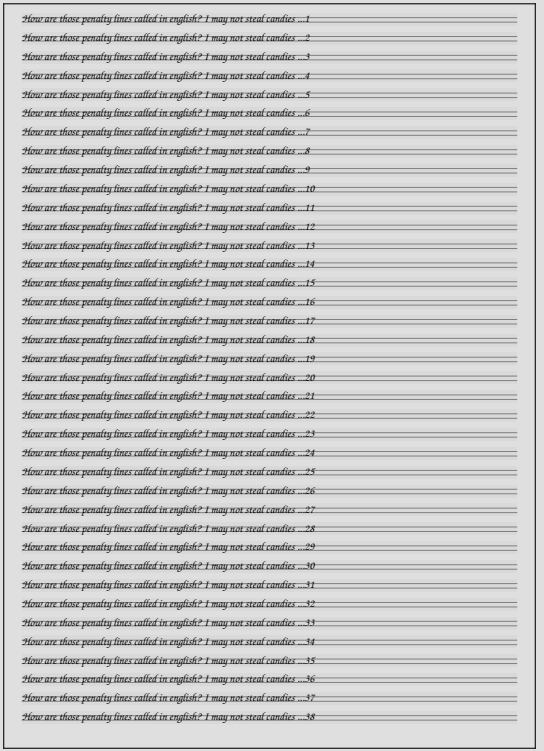


Figure 4.4 Penalty lines.

```
          I may not steal candies ..." & decimal n) shifted (1,i) shifted (0,-StrutDepth) ;
      endfor ;
    StopPage ;
  \stopMPpage
  \stop
```

This code demonstrates the use of `LineHeight`, `ExHeight`, `StrutHeight` and `StrutDepth`. We set the interline spacing to 1.5 so that we get a bit more loose layout. The variables mentioned are set each time a graphic is processed and thereby match the current font settings.

5 Positional graphics

In this chapter, we will explore some of the more advanced, but also conceptually more difficult, graphic capabilities of CONTEXT. *It took quite a few experiments to find the* right *way to support these kind of graphics, and you can be sure that in due time extensions will show up. You can skip this chapter if you are no* CONTEXT *user.*

5.1 The concept

After TEX has read a paragraph of text, it will try to break this paragraph into lines. When this is done, the result is flushed and after that, TEX will check if a page should be split off. As a result, we can hardly predict how a document will come out. Therefore, when we want graphics to adapt themselves to this text, we have to deal with this asynchronous feature of TEX in a rather advanced way. Before we present one way of dealing with this complexity, we will elaborate on the nature of such graphics.

When TEX entered the world of typesetting, desktop printers were not that common, let alone color desktop printers. But times have changed and nowadays we want color and graphics, and, if possible, we want them integrated in the text. To accomplish this several options are open:

1. Use a backend that acts on the typeset text. This is the traditional way, using specials to embed directives in the DVI output file.
2. Use the power of a second language and pass snippets of code to the backend which takes care of proper handling of those snippets of text. Impressive results are booked by passing POSTSCRIPT to the DVI file.
3. Extend TEX in such a way that TEX itself takes care of these issues. This is the way PDFTEX and its decendants work.

The first method is rather limited, although for business graphics the results are acceptable. The second method is very powerful but hardly portable, since it depends on the DVI to POSTSCRIPT postprocessor. But what about the third method?

There has been some reluctance to divert from traditional TEX and DVI, but with the development of PDFTEX, the third option has become a viable option. Much of what I will discuss here can be realized in DVI, using a dedicated postprocessor to extract the information needed. Although we believe that the PDFTEX way is the natural way to go, CONTEXT MKII also supports the same mechanism in DVI. In MKIV we use the built in mechanism.

As said, a decent portion of TEX's attention is focused on breaking paragraphs into lines and determining the optimal point to split off the page. Trying to locate the optimal points to break lines is a dynamic process. The space between words is flexible and we don't know in advance when a word or piece of a word —maybe it's best to talk of typographic globs instead— will end up on the page. It might even cross the page boundary.

In the previous paragraph word and globs are encircled and connected by an arrow. This graphic can be drawn only when the position and dimensions are known. Unfortunately, this information is only available after the paragraph is typeset and the best breakpoints are chosen. Because the text must be laid on top of the graphic, the graphic must precede the first word in the typeset stream or it must be positioned on a separate layer. In the latter case it can be calculated directly after the paragraph is typeset, but in the former case a second pass is needed. Because such graphics are not bound to one paragraph, the multi–pass option suits better because it gives us more control: the more we know about he final state, the better we can act upon it. Think of graphics on the first page that depend on the content of the last page or, as in this paragraph, backgrounds that depend on the typeset text.

It may be clear now that we need some positional information in order to provide features like the ones shown here. The fact that we will act upon in a second pass simplifies the task, although it forces us to store the positional information between runs in some place. This may look uncomfortable at first sight, but it also enables us to store some additional information. Now why is that needed?

A position has no dimensions, it's just a place somewhere on the page. In order to do tricks like those shown here, we also need to know the height and depth of lines at a specific point as well as the width of the box(es) we're dealing with. In the encircled examples, the dimensions of the box following the positional node are stored along with the position. In the background example, we store the current height and depth of the strut (an imaginary character | with maximum height and depth but no width) along with the current text width.

In order to process the graphics, we tag each point with a name, so that we can attach actions to those points. In fact they become trigger points. As we will demonstrate, we also need to store the current page number. This brings the data stored with a point to:

```
<identifier><pagenumber><x><y><width><height><depth>
```

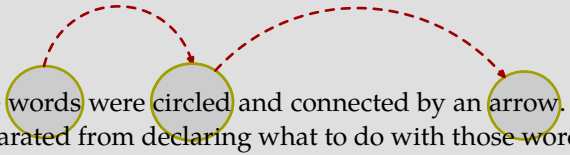
The page number is needed in order to let the graphics engine determine boundary conditions. Backgrounds like those shown here can span multiple pages. In order to calculate the right backgrounds, some additional information must be available, like the top and bottom of the current text area. In fact, these are just normal points that can be saved while processing the split off page. So, apart from positioning anchors in the text we need anchors on crucial points of the layout. This means that this kind of support cannot be fully integrated into the TEX kernel, unless we also add extensive support for layout definitions, and that is probably not what we want.

As soon as something like (x, y) shows up, a logical question is where $(0, 0)$ is located. Although this is a valid question, the answer is less important than you may expect. Even if we know that $(0, 0)$ is 'officially' located in the bottom left corner of the page, the simple fact that in CONTEXT we are dealing with a mixed page concept, like paper size and print paper size, or left and right pages, forces us to think in relative positions instead of absolute ones. Therefore, graphics, even those that involve multiple positions, are anchored to a position on the layer on which they are located. The `\MPanchor` macro takes care of this.

Users who simply want to use these features may wonder why we go into so much detail. The main reason is that in the end many users will want to go beyond the simple cases, and when dealing with these issues, you

must be aware not only of height, depth and width, but also of the crossing of a page boundary, and the height and depth of lines. In some cases your graphics may have to respond to layout characteristics, like differences between odd and even pages. Given that unique identifiers are used for anchor points, in CONTEXT you can have access to all the information needed.

5.2 Anchors and layers



In a previous section we saw that some words were circled and connected by an arrow. As with most things in CONTEXT, marking these words is separated from declaring what to do with those words. This paragraph is keyed in as:

```
In a previous section we saw that some \hpos {X-1} {words} were
\hpos {X-2} {circled} and connected by an \hpos {X-3} {arrow}.
As with most things in \CONTEXT, marking these words is separated
from declaring what to do with those words. This paragraph is keyed
in as:
```

We see three position anchors, each marked by an identifier: `X-1`, `X-2` and `X-3`. Each of these anchors can be associated with a (series) of graphic operations. Here we defined:

```
\setMPpositiongraphic{X-1}{mypos:arrow}{to=X-2}
\setMPpositiongraphic{X-2}{mypos:arrow}{to=X-3}
```

These examples clearly demonstrate that we cannot control to what extent graphics will cover text and vice versa. A solution to this problem is using position overlays. We can define such an overlay as follows:

```
\startpositionoverlay{backgraphics}
```

```
  \setMPpositiongraphic{G-1}{mypos:circle}
  \setMPpositiongraphic{G-2}{mypos:circle}
  \setMPpositiongraphic{G-3}{mypos:circle}
  \setMPpositiongraphic{G-4}{mypos:circle}
\stoppositionoverlay

\startpositionoverlay{foregraphics}
  \setMPpositiongraphic{G-1}{mypos:line}{to=G-2}
  \setMPpositiongraphic{G-2}{mypos:line}{to=G-3}
  \setMPpositiongraphic{G-3}{mypos:line}{to=G-4}
\stoppositionoverlay
```

First we have defined an overlay. This overlay can be attached to some overlay layer, like, in our case, the page. We define four small circles. These are drawn as soon as the page overlay is typeset. Because they are located in the background, they don't cover the text, while the lines do. The previous paragraph was typeset by saying:

```
First we have defined an \hpos {G-1} {overlay}. This
overlay can be attached to some overlay layer, like, in our
case, the \hpos {G-2} {page}. We define four small \hpos
{G-3} {circles}. These are drawn as soon as the page
overlay is typeset. Because they are located in the
background, they don't cover the \hpos {G-4} {text}, while
the lines do. The previous paragraph was typeset by saying:
```

As said, the circles are on the background layer, but the lines are not! They are positioned on top of the text. This is a direct result of the definition of the page background:

```
\defineoverlay [foregraphics] [\positionoverlay{foregraphics}]
```

```
\defineoverlay [backgraphics] [\positionoverlay{backgraphics}]

\setupbackgrounds
  [page]
  [background={backgraphics,foreground,foregraphics}]
```

In this definition, the predefined overlay `foreground` inserts the page data itself, so the foreground graphics end up on top. This example also demonstrates that you should be well aware of the way CONTEXT builds a page. There are six main layers, in some cases with sublayers. The body text goes into the main text layer, which, unless forced otherwise, lays on top.

1. paper background
2. area backgrounds
3. page backgrounds
4. text areas
5. logo areas
6. main text

The paper background is used for special (sometimes internal) purposes. There are three page backgrounds: left, right and both. The text areas, logo areas and backgrounds form a 5×5 matrix with columns containing the leftedge, leftmargin, text, rightmargin, and rightedge. The rows of the matrix contain the top, header, text, footer, and bottom. The main text is what you are reading now.

Since the page background is applied last, the previous layers can considered to be the foreground to the page background layer. And, indeed, it is available as an overlay under the name `foreground`, as we already saw in the example. Foregrounds are available in most cases, but (for the moment) not when we are dealing with the text area. Since anchoring the graphics is implemented rather independent of the position of the graphics themselves, this is no real problem, we can put them all on the page layer, if needed in separate overlays.

How is such a graphic defined? In fact these graphics are a special case of the already present mechanism of including METAPOST graphics. The circles are defined as follows:

```
\startMPpositiongraphic{mypos:circle}
  initialize_box(\MPpos{\MPvar{self}}) ;
  path p ; p := llxy..lrxy..urxy..ulxy..cycle ;
  pickup pencircle scaled 1pt ;
  fill p withcolor .800white ;
  draw p withcolor .625yellow ;
  anchor_box(\MPanchor{\MPvar{self}}) ;
\stopMPpositiongraphic
```

Drawing the lines is handled in a similar fashion.

```
\startMPpositiongraphic{mypos:line}
  path pa, pb, pab ; numeric na, nb ;
  initialize_box(\MPpos{\MPvar{from}}) ;
  na := nxy ; pa := llxy..lrxy..urxy..ulxy..cycle ;
  initialize_box(\MPpos{\MPvar{to}}) ;
  nb := nxy ; pb := llxy..lrxy..urxy..ulxy..cycle ;
  if na=nb :
    pab := center pa -- center pb ;
    pab := pab cutbefore (pab intersectionpoint pa) ;
    pab := pab cutafter  (pab intersectionpoint pb) ;
    pickup pencircle scaled 1pt ;
    draw pab withcolor .625yellow ;
    anchor_box(\MPanchor{\MPvar{from}}) ;
  fi ;
\stopMPpositiongraphic
```

The command \startMPpositiongraphic defines a graphic, in this example we have called it mypos:circle.

The METAPOST macro initialize_box returns the characteristics of the box as identified by \MPpos. After this call, the corners are available in llxy, lrxy, urxy and ulxy. The center is defined by cxy and the path stored in pxy. When we are finished drawing the graphic, we can anchor the result with anchor_box. This macro automatically handles positioning on specific layers.

The position macro \MPpos returns the current characteristics of a position. The previously defined G positions return:

position	page	x	y	width	height	depth
G-1	248	168.33380pt	240.03143pt	31.97002pt	7.25999pt	2.83000pt
G-2	248	516.34001pt	240.03143pt	21.15999pt	4.69000pt	2.83000pt
G-3	248	154.09756pt	227.45941pt	27.68001pt	7.25999pt	0.20000pt
G-4	248	232.15891pt	214.88739pt	16.46999pt	6.21001pt	0.20000pt

The numbers represent the real pagenumber p, the current position (x, y), and the dimensions of the box (w, h, d) if known. These values are fed directly into METAPOST graphics but the individual components can be asked for by \MPp, \MPx, \MPy, \MPw, \MPh and \MPd.

In the previous definition of the graphic, we saw another macro, \MPvar. When we invoke a graphic or attach a graphic to a layer, we can pass variables. We can also set specific variables in other ways, as we will see later.

```
\setMPpositiongraphic{G-1}{mypos:circle}
\setMPpositiongraphic{G-1}{mypos:line}{to=G-2}
```

In the second definition, we let the variable to point to another position. When needed, we can ask for the value of to by \MPvar{to}. For reasons of convenience, the current position is assigned automatically to from and self. This means that in the line we saw in the graphic:

```
initialize_box(\MPpos{\MPvar{self}}) ;
```

\MPvar{self} will return the current position, which, fed to \MPpos will return the list of positional numbers. We already warned the reader: this is not an easy chapter.

5.3 More layers

Overlays are one of the nicer features of CONTEXT and even more nice things can be build on top of them. Overlays are defined first and then assigned to framed boxes using the background variable.

You can stack overlays, which is why they are called as such. You can use the special overlay called foreground to move the topmost (often text) layer down in the stack.

background overlay	a text, graphic, hyperlink or widget
position overlay	a series of macros triggered by positions
background layer	a box that can hold boxes with offsets

The last kind of layer can be used in other situations as well, but in most cases it will be hooked into a background overlay.

```
\definelayer[MyLayer][option=test]

\setupbackgrounds[text][leftmargin][background=MyLayer]

\setlayer[MyLayer][x=.5cm,y=5cm]
  {\rotate{\framed{This goes to the background}}}
```

This goes to the background

In this case the framed text will be placed in the background of the (current) page with the given offset to the topleft corner. Instead of a fixed position, you can inherit the current position using the `position` directive. Say that we have a layer called `YourLayer` which we put in the background of the text area.

```
\definelayer[YourLayer]
\setupbackgrounds[text][text][background=YourLayer]
```

We can now move some framed text to this layer using `\setlayer` with the directive `position` set to `yes`.

```
here: \setlayer[YourLayer][position=yes]{\inframed{Here}}
```

here: Here

You can influence the placement by explicitly providing an offset (`hoffset` and/or `voffset`), a position (`x` and/or `y`) or a location directive (`location`). Normally you will use the offsets for the layer as a whole and the positions for individual contributions. The next example demonstrates the use of a location directive.

```
here: \setlayer[YourLayer][position=yes,location=c]{\inframed{Here}}
```

here: Here

Many layers can be in use at the same time. In the next example we put something in the page layer. By default, we turn on position tracking, which visualizes the bounding box of the content and shows the reference point.

```
\definelayer[BackLayer][position=yes]
\setupbackgrounds[page][background=BackLayer]
```

Next we define an overlay that we can put behind for instance framed texts. We use METAPOST to draw `Shape`.

```
\defineoverlay[Shape] [BackLayer] [\uniqueMPgraphic{Shape}]
```

```
\startuniqueMPgraphic{Shape}
  path p ; p := fullcircle xyscaled(OverlayWidth,OverlayHeight) ;
  fill p withcolor \MPcolor{lightgray} ;
  draw p withpen pencircle scaled 1pt withcolor \MPcolor{darkred} ;
\stopuniqueMPgraphic
```

We can now put this background shape behind the running text, for instance with:

```
.... some \inframed[background=Shape]{text} with a frame ...
.... some \Shaped{text} with a frame ...
```

.... some text with a frame ...
.... some text with a frame ...

The \Shaped macro was defined as:

```
\defineframed[Shaped][background=Shape,frame=off,location=low]
```

Watch how the graphics are moved to the background while the frame of the first text stays on top, since it remains part of the text flow.

.... some text with a frame ...
.... some text with a frame ...

In the previous instance of the example we have reversed the stacking. Reversal can be done with the direction directive.

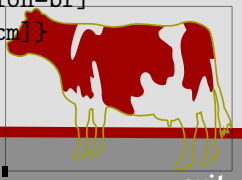
```
\setuplayer[BackLayer][direction=reverse]
```

You can influence the placement of a background component by using a different anchor point.

```
\setuplayer
  [BackLayer]
  [position=no,corner=bottom,height=\paperheight]

\setlayer[BackLayer][x=1cm,y=10cm,location=bl]
  {\externalfigure[somecow.pdf][width=1cm]}

\setlayer[BackLayer][x=.5cm,y=8cm,location=br]
  {\externalfigure[somecow.pdf][width=1cm]}

\setlayer[BackLayer][x=1cm,y=4cm,location=tl]
  {\externalfigure[somecow.pdf][width=1cm]}

\setlayer[BackLayer][x=10cm,y=.5cm,location=tr]
  {\externalfigure[somecow.pdf][width=3cm]}
```

Instead of using relative positions, you can also use absolute ones. Of course you need to know how your coordinates relate to the rest of the layout definition.

```
\setuplayer
  [BackLayer]
  [position=no,corner=bottom,height=\paperheight]

\setlayer[BackLayer][x=15cm,y=5cm,location=bl]
  {\externalfigure[somecow.pdf][width=3cm]}

\setlayer[BackLayer][x=15cm,y=5cm,location=br]
  {\externalfigure[somecow.pdf][width=3cm]}
```



```
\setlayer[BackLayer][x=15cm,y=5cm,location=tl]
  {\externalfigure[somecow.pdf][width=1cm]}

\setlayer[BackLayer][x=15cm,y=5cm,location=tr]
  {\externalfigure[somecow.pdf][width=2cm]}

\setlayer[BackLayer][x=15cm,y=5cm,location=c]
  {\externalfigure[somecow.pdf][width=3cm]}
```

These examples again demonstrate how we can influence the placement by assigning an anchor point to `position`. Here we also put the reference point in the lower left corner (`bottom`). This mechanism only works when we also use `height`.

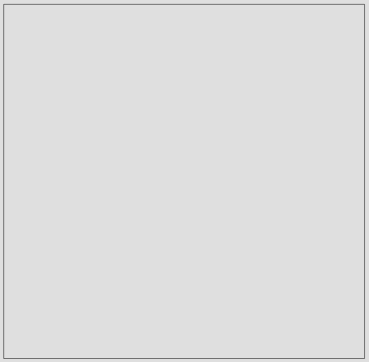
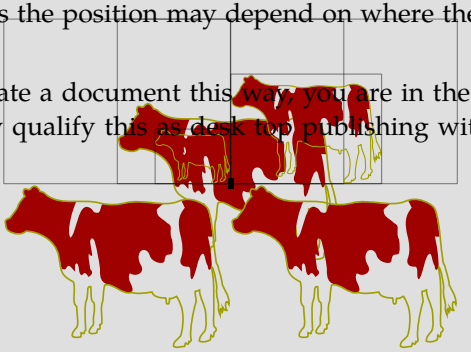


Figure 5.1

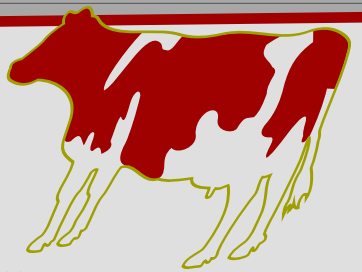
One of the reasons for developing the layer mechanism was that we needed to slightly change the position of figures in the final stage of typesetting. The previous pages demonstrate how one can position anything anywhere on the page, but in the case of figures the position may depend on where the text ends up on the page.

Normally, when you manipulate a document this way, you are in the final stage of typesetting. You may qualify this as desk top publishing without actually using a desktop.

```
\setlayer [BackLayer]
  [position=yes,
   location=c,
   voffset=-.5cm,
   width=50pt,
   height=50pt]
```



```
{\startMPcode
   draw externalfigure "somecow.pdf"
     xscaled 100bp yscaled 100bp rotated 25 ;
 \stopMPcode}
```



The previous example also demonstrated the use of METAPOST for rotating the picture. The `\MPfigure` macro encapsulates the code in a shortcut. You can achieve special effects by using the layers behind floating bodies and alike, but always keep in mind that the readability of the text should not be violated too much.

```
\startbuffer
\setlayer [BackLayer]
  [position=yes,voffset=-1.5cm,width=3cm,height=2cm]
  {\MPfigure{somecow.pdf}{scaled .5 slanted .5}}
\stopbuffer

\placefigure[right]{}{\ruledhbox{\getbuffer}}
```

Figure 5.2

In these examples we added a `\ruledhbox` around the pseudo graphics so that you can see what the size is of those graphics.

We have already seen a lot of parameters that can be used to control the content of a layer. There are a few more. General housekeeping takes place with:

`state`	`start`	enable the layer
	`stop`	disable the layer
`position`	`no`	use absolute positions

	`yes`	use relative positions
	`overlay`	idem, but ignore the size
`direction`	`normal`	put new data on top
	`reverse`	put new data below old data

Sometimes all data needs to be offset in a similar way. You can use both offset parameters for that.

`hoffset`	an additional horizontal displacement
`voffset`	an additional vertical displacement

You can position data anywhere in the layer. When positioning is turned on, the current position will get a placeholder. You can change the dimensions of that placeholder (when `position` is set to `overlay`), zero dimensions are used.

`x`	the horizontal displacement
`y`	the vertical displacement
`width`	the (non natural) width
`height`	the (non natural) height
`location`	`l r t b c lt lb rt rb`

The `location` directive determines what point of the data is used as reference point. You can keep track of this point and the placement when you enable test mode. This is how the rectangles in the previous examples where drawn.

`option`	`test`	show positioning information

When you are enhancing the final version of a document, you can explicitly specify on what page the data will go. Use this option with care.

`page`	the page where the data will go

Because layers can migrate to other pages, they may disappear due to the background not being recalculated. In case of doubt, you can force repetitive background calculation by:

```
\setupbackgrounds[state=repeat]
```

5.4 Complex text in graphics

If you like to embed METAPOST snippets in CONTEXT, you may want to combine text and graphics and let METAPOST provide the position and the dimensions of the text to be typeset outside by TEX. For most applications using the METAFUN `textext` macro works well enough, but when the typeset text needs to communicate with the typesetting engine, for instance because it contains hyperlinks or references, you can use the following method:

- define a layer
- define a (reusable) graphic
- put your text into the layer
- combine the graphic with the text

You must be aware of the fact that when the layer is flushed, its content is gone. You can take advantage of this by using the same graphic with multiple texts.

```
\definelayer[test]
```

You don't need to pass the width and height explicitly, but when you do so, you have access to them later.

```
\startuseMPgraphic{oeps}
  path p ; p := fullcircle scaled 6cm ;
  fill p withcolor .8white ;
  draw p withpen pencircle scaled 1mm withcolor .625red ;
  register ("somepos-1",0cm,0cm,center currentpicture) ;
  register ("somepos-2",3cm,1cm,(-1cm,-1cm)) ;
  register ("somepos-3",2cm,0cm,(-2cm,2cm)) ;
\stopuseMPgraphic
```

The METAFUN `register` macro takes the following arguments:

```
register ("tag",width,height,(x offset,y offset)) ;
```

The width and height are available in the macros `\MPlayerwidth` and `\MPlayerheight` and are equivalent to `\MPw{tag}` and `\MPh{tag}`,

```
\setMPlayer [test] [somepos-1] [location=c]
  {Does it work al right?}

\setMPlayer [test] [somepos-2]
  {\framed
     [width=\MPlayerwidth,height=\MPlayerheight,
      background=color,backgroundcolor=white]
     {It Works!}}

\setMPlayer [test] [somepos-3]
  {\externalfigure[cow.mps][width=2cm]}
```

Combining the graphic and the text is handled by the macro `\getMPlayer`.

```
\getMPlayer [test] {\useMPgraphic{oeps}}
```

The macro `\getMPlayer` is built on top of `\framed`. The settings passed in the (optional) second argument are the same as those to `\framed`.

```
\getMPlayer
  [test]
  [frame=on,offset=5pt]
  {\useMPgraphic{oeps}}
```

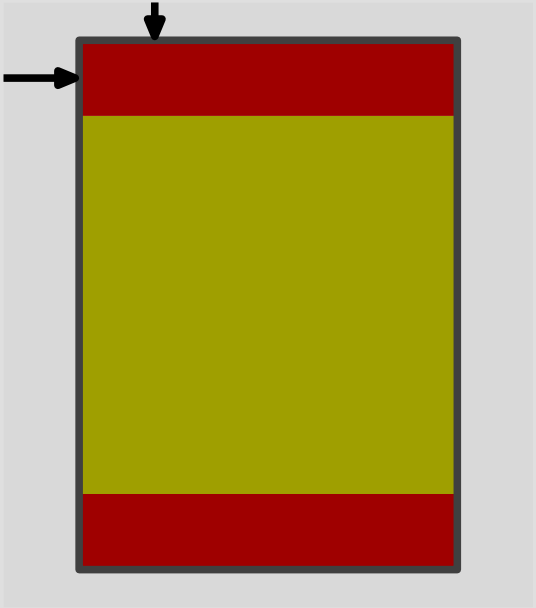
As you see, you need a bit of a twisted mind to handle graphics this way, but at least the functionality is there to create complex graphics in a declarative way.

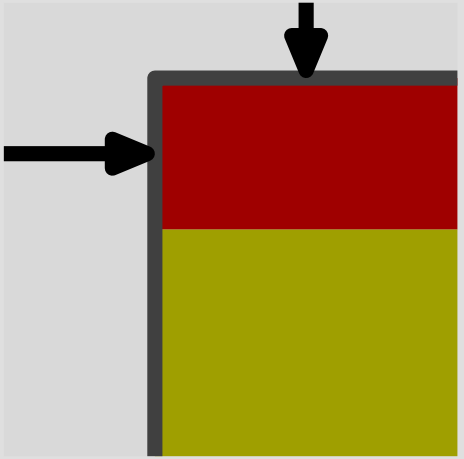
6 Page backgrounds

Especially in interactive documents, adding backgrounds to the page and text areas not only enhances readability, but also makes it more convenient to identify header, footers and navigational areas. In this chapter we will demonstrate that with METAPOST we can go beyond the TEX based features present in CONTEXT. One section is dedicated to graphics and printing, especially bleeding.

6.1 The basic layout

In the CONTEXT manual you can find many details on the composition of the page. When TEX typesets text, crossing the page boundary triggers TEX's output routine. This routine is responsible for pasting the body text that goes onto a page in the correct area. A simple representation of such a page is:





The red areas are the header and footer, while the yellow areas contains the text flow. We can turn headers on and off and/or hide them. For this reason, the header, text and footer areas together make up the height of the text.

A close look at the left picture will reveal that the two arrows point to the center of the lines. This is achieved by the `top` and `lft` directives. If we would not have clipped the picture, the arrow would have stuck half a line width outside the gray area that represents the page. When constructing such pictures, one should really pay attention to such details, since it pays off in the overall look and feel of the document.

The vertical arrow represents the top space, while the horizontal arrow denotes the distance to the back of the cover (back space). By changing their values, you can shift the main body text on the page. In a double sided layout scheme, the back space is automatically mirrored on even pages.

Since we want to teach a bit of METAPOST now and then, we will also show how these graphics were drawn. An advanced METAPOST user may wonder why we hard code the dimensions, and avoid METAPOST's powerful mechanisms for defining relations. Our experience has taught us that in pictures like this, providing a general solution seldom pays large dividents or savings in time.

```
\startuseMPgraphic{layout 1}
  pickup pencircle scaled 1mm ;
  fill unitsquare xyscaled (7cm,8cm)
    withcolor .85white ;
  fill unitsquare xyscaled (5cm,5cm) shifted (1cm,1.5cm)
    withcolor .625yellow ;
  fill unitsquare xyscaled (5cm,1cm) shifted (1cm,.5cm)
    withcolor .625red ;
  fill unitsquare xyscaled (5cm,1cm) shifted (1cm,6.5cm)
    withcolor .625red ;
  draw unitsquare xyscaled (5cm,7cm) shifted (1cm,.5cm)
    withcolor .25white ;
  drawarrow (2cm,8cm) -- top (2cm,7.5cm) ;
  drawarrow (0cm,7cm) -- lft (1cm,7cm) ;
  clip currentpicture to unitsquare xyscaled (7cm,8cm) ;
\stopuseMPgraphic
```

As you can see, the left graphic is defined as a series of rectangles. The `xyscaled` macro is part of the CONTEXT files, and saves some typing and space. It is defined as a primary, requiring both left and right operands.

```
primarydef p xyscaled q =
  p xscaled (xpart q) yscaled (ypart q)
enddef ;
```

Zooming in on the top left corner only takes a few lines. First we clip the correct part, next we scale it up, and finally we let the bounding box suit the left picture.

```
\startuseMPgraphic{layout 2}
  \includeMPgraphic{layout 1}
  clip currentpicture to unitsquare scaled 3cm shifted (0,5cm) ;
  currentpicture := currentpicture scaled 2 shifted (0,-8cm) ;
  setbounds currentpicture to unitsquare xyscaled (6cm,8cm) ;
\stopuseMPgraphic
```

This code demonstrates how you can reuse a graphic inside another one. This strategy can easily be used to stepwise build (or extend) graphics. The two graphics were put side by side with the following command. Watch the use of line correction commands. They optimize the white space around the graphic.

```
\startlinecorrection[blank]
\hbox
  {\useMPgraphic{layout 1}\hskip1cm
   \useMPgraphic{layout 2}}
\stoplinecorrection
```

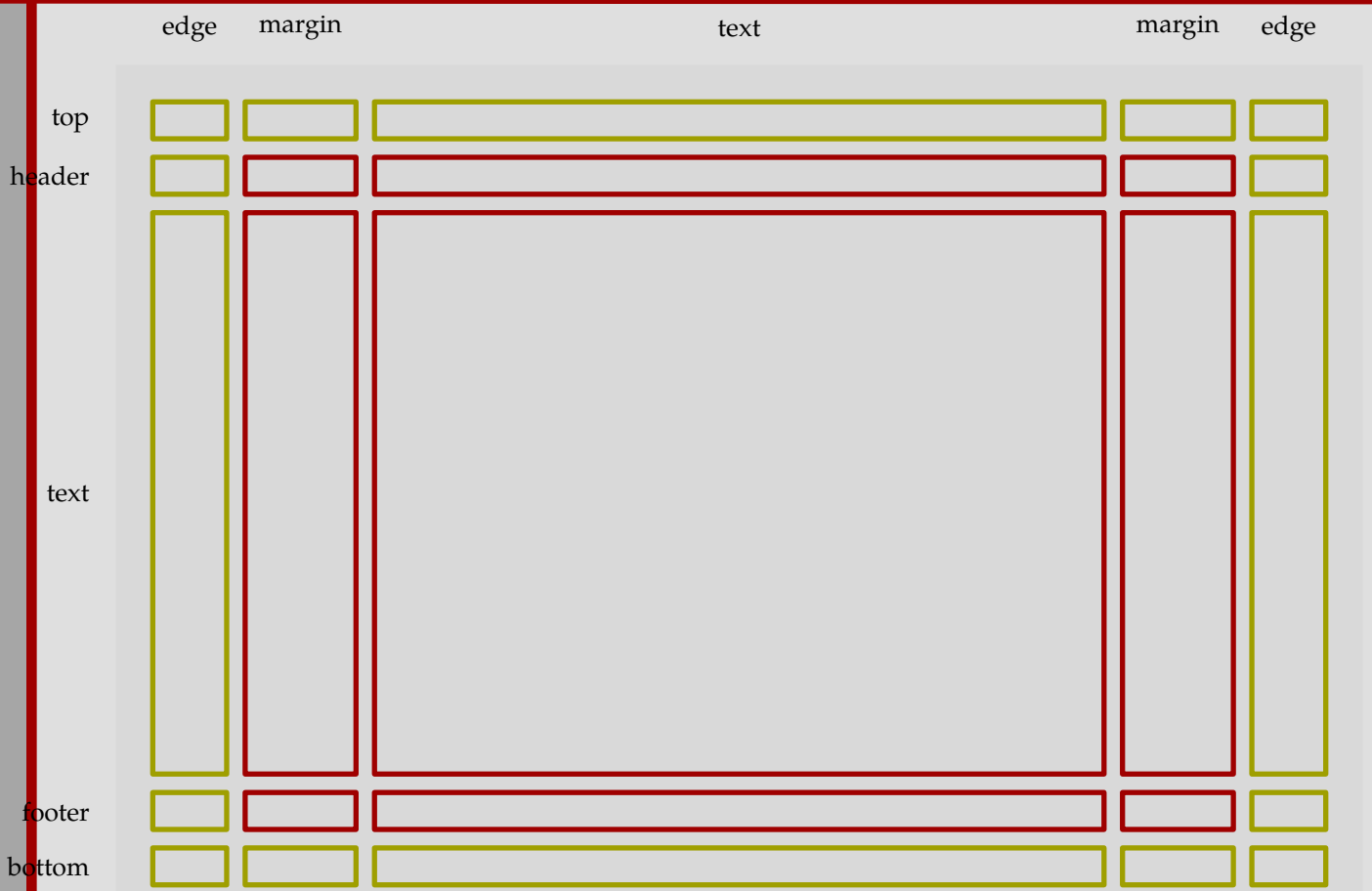
As soon as you want to make an electronic document, you will want to use different areas of the screen for different purposes: text, menus, buttons, etc. For this reason, CONTEXT provides not only left and right margins, but also additional left and right edge areas and top and bottom margins. These areas are shown in the figure on the next page.

When defining this graphic, all areas have related dimensions. Here it makes sense to let METAPOST calculate these dimensions as much as possible. First we define the five by five matrix of areas. We pass the width and height of the main text area. Because they are stored in TEX dimension registers, we have to prefix them by `\the`.

```
pickup pencircle scaled 2pt ;

numeric w[], h[], x[], y[], u ; u := .5cm ;

numeric width  ; width  := \the\textwidth  ;
numeric height ; height := \the\textheight ;
```

We now specify the lower left corners using = instead of the :=, which means that METAPOST will calculate `w[3]` and `h[3]` for us.

```
w[1] = 2u ; w[2] = 3u ; w[4] = 3u ; w[5] = 2u ;
h[1] = 1u ; h[2] = 1u ; h[4] = 1u ; h[5] = 1u ;

w[1]+w[2]+w[3]+w[4]+w[5]+4u = width  ;
h[1]+h[2]+h[3]+h[4]+h[5]+4u = height ;

x[1] = 1u ;                 y[1] = 1u ;
x[2] = x[1] + w[1] + .5u ; y[2] = y[1] + h[1] + .5u ;
x[3] = x[2] + w[2] + .5u ; y[3] = y[2] + h[2] + .5u ;
x[4] = x[3] + w[3] + .5u ; y[4] = y[3] + h[3] + .5u ;
```


edge
margin
text
margin
edge
top
header
text
footer
bottom

```
x[5] = x[4] + w[4] + .5u ; y[5] = y[4] + h[4] + .5u ;
```

Because we are going to repeat ourselves, we draw the areas using a macro. Depending on its importance, we color it red or yellow.

```
def do_it (expr xx, yy, cc) =
  draw unitsquare
    xyscaled (w[xx],h[yy]) shifted (x[xx],y[yy])
    withcolor if cc : .625red else : .625yellow fi ;
enddef ;

fill unitsquare xyscaled (width,height) withcolor .85white;

do_it (1,1,false) ; do_it (5,1,false) ;
do_it (2,1,false) ; do_it (3,1,false) ; do_it (4,1,false) ;

do_it (1,2,false) ; do_it (5,2,false) ;
do_it (2,2,true ) ; do_it (3,2,true ) ; do_it (4,2,true ) ;

do_it (1,3,false) ; do_it (5,3,false) ;
do_it (2,3,true ) ; do_it (3,3,true ) ; do_it (4,3,true ) ;

do_it (1,4,false) ; do_it (5,4,false) ;
do_it (2,4,true ) ; do_it (3,4,true ) ; do_it (4,4,true ) ;

do_it (1,5,false) ; do_it (5,5,false) ;
do_it (2,5,false) ; do_it (3,5,false) ; do_it (4,5,false) ;
```

This picture in itself is not yet explanatory, so we add some labels. Again, we use a macro, which we feed with a picture generated by TEX. Since these pictures are filtered from the source and pre–processed, we cannot embed the `btex–etex` in the macro `do_it` and pass a string. It has to be done this way.[11]

```
def do_it (expr yy, tt) =
  path p ;
  p := unitsquare xyscaled (w[1],h[yy]) shifted (x[1],y[yy]) ;
  label.lft(tt, center p shifted (-w[1]/2-u-.25cm,0)) ;
enddef ;

do_it (1,btex bottom etex) ;
do_it (2,btex footer etex) ;
do_it (3,btex text   etex) ;
do_it (4,btex header etex) ;
do_it (5,btex top    etex) ;
```

In the horizontal direction we have edges, margins and text. There are left and right edges and margins, which are swapped on even pages when you typeset a double sided document.

```
def do_it (expr xx, tt) =
  path p ;
  p := unitsquare xyscaled (w[xx],h[1]) shifted (x[xx],y[1]) ;
  label(tt, center p shifted (0,height-h[1]/2)) ;
enddef ;

do_it (1,btex edge   etex) ;
```

[11] This is true only in a regular METAPOST run. In CONTEXT MKIV we follow a different route.

```
do_it (2,btex margin etex) ;
do_it (3,btex text   etex) ;
do_it (4,btex margin etex) ;
do_it (5,btex edge   etex) ;
```

Since we want the graphic to match the dimensions of the text area of the current page, we have to make sure that the bounding box is adapted accordingly. By this action, the labels will fall outside the bounding box. When we directly embed a graphic, this works ok, but when we start scaling and reusing, due to the object reuse mechanism the graphic will be clipped to the bounding box.

```
setbounds currentpicture to
  unitsquare xyscaled (width,height) ;
```

In the following sections we will demonstrate how you can put graphics behind these 25 areas, as well as behind the (left and right) page.

6.2 Setting up backgrounds

One way of protecting a document for unwanted usage is to put an annoying word in the background. If you like this, you may try the following. The macro `ysized` is part of the macros that come with CONTEXT and scales a picture to a specific size.

```
\startuniqueMPgraphic{concept}
  draw btex \colored[s=.8]{\bf CONCEPT} etex rotated 60 ;
  currentpicture := currentpicture
    ysized (\overlayheight-.5cm) ;
```

```
\stopuniqueMPgraphic

\defineoverlay[concept][\uniqueMPgraphic{concept}]
```

You can now put this graphic in the page background by saying:

```
\setupbackgrounds[page][background=concept]
```

You may consider the next alternative a bit better, but still it renders the text unreadable. Like `xysized`, the macro `enlarged` is not part of standard METAPOST, but comes with CONTEXT.

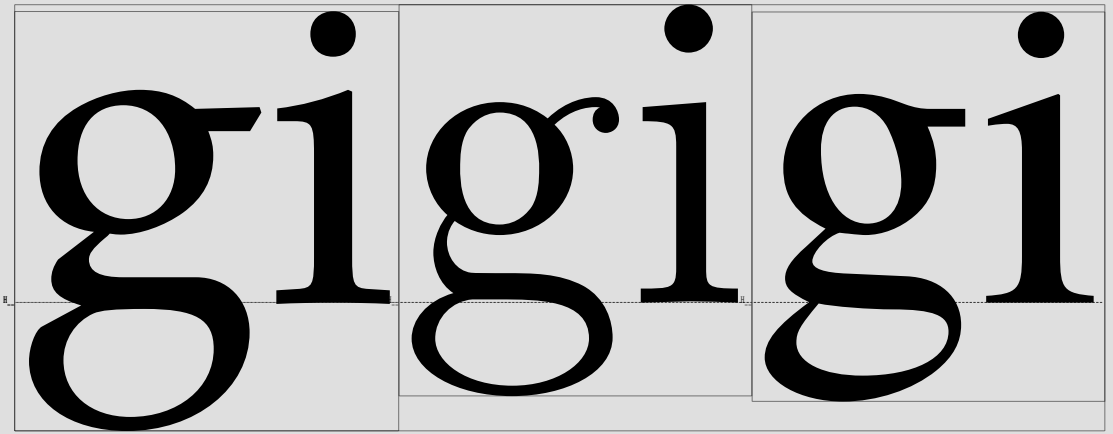
```
\startuniqueMPgraphic{copyright}
  picture p ; p := btex \colored[s=.8]{COPYRIGHT} etex
    rotated 90 ;
  setbounds p to boundingbox p enlarged 1pt ;
  draw p ;
  currentpicture := currentpicture
    xysized (\overlaywidth,\overlayheight) ;
\stopuniqueMPgraphic

\defineoverlay[copyright][\uniqueMPgraphic{copyright}]
```

Again, we put this graphic in the background. By using a unique graphic, we make sure that it's rendered only once and reused when possible.

```
\setupbackgrounds[text][rightmargin][background=copyright]
```

In both cases, we slightly scale down the graphic. We do so because otherwise a small portion of the text if clipped off. This is unrelated to TEX or METAPOST, but a characteristic of the font. Compare the following Pagella, Latin Modern and Termes gi's (the Pagella is the body font of this text).

```
\showboxes
\hbox \bgroup
    \hbox{\definedfont[file:texgyrepagella-regular at 6cm]gi}%
    \hbox{\definedfont[file:lmroman10-regular      at 6cm]gi}%
    \hbox{\definedfont[file:texgyretermes-regular  at 6cm]gi}%
\egroup
```



Watch how the bounding boxes differ and sometimes cross the shape. So, in order not to loose part of a glyph when clipping, you need to add a bit of space. **Figure 6.1** shows the two backgrounds in action.

Figure 6.1 Two examples of annoying backgrounds.

If you really want to add such texts to a document, in CONTEXT we don't have to use the page background, but can use one of the layout areas instead (like [text][text] or [text][leftmargin])

```
\setupframedtexts
  [FunnyText]
  [backgroundcolor=lightgray,
   framecolor=darkred,
   rulethickness=2pt,
   offset=\bodyfontsize,
   before={\blank[big,medium]},
   after={\blank[big]},
   width=\textwidth]
```

There is one drawback: when your left and right margin have different dimensions, the text will be scaled differently on odd and even pages. Normally this is no problem for a draft.

As an alternative you can use the \setuptexts command and wrap the graphic in a box with the right dimensions, using code like:

```
\startuniqueMPgraphic{copyright}
  picture p ; p := btex COPYRIGHT etex rotated 90 ;
  setbounds p to boundingbox p enlarged 1pt ;
  draw p withcolor .8white ;
  xyscale_currentpicture(\the\leftmarginwidth,\the\textheight) ;
\stopuniqueMPgraphic

\setuptexttexts [margin] [] [\uniqueMPgraphic{copyright}]
```

The graphic goes into the outer margin. The second argument can be used to put something in the inner margin.

6.3 Multiple overlays

You can stack overlays. Consider the next case, where we assume that you have enabled interaction support using `\setupinteraction[state=start]`:

```
\setupbackgrounds
  [page]
  [background={color,nextpage},
   backgroundcolor=darkyellow]
```

Here, the page gets a colored background and a hyperlink to the next page, previously defined by:

```
\defineoverlay[nextpage][\overlaybutton{nextpage}]
```

An `\overlaybutton` is just a button, with all attributes (color, frame, etc) set to nothing, having the dimensions of the overlay. The argument is one of the permitted destinations, like `nextpage`, `firstpage`, `SearchDocument` and alike.

For efficiency reasons, the background areas (like `[text][text]`) are calculated only when their definition has changed. When a background changes per page, we have to recalculate it on each page. In the next example, the macro `\overlaybutton` generates a different button on each page. But, since we don't explicitly set the background at each page, there is no way the background drawing mechanism can know that this button has changed. Therefore, we must force recalculation with:

```
\setupbackgrounds[state=repeat]
```

You can test this concept yourself with the following code. Here we assume that you have a file called `tufte.tex` on your system, which is the case if you have CONTEXT installed. However, you can just as easily use any file having a paragraph of two of text.

```
\starttext
\setupinteraction[state=start]
\setupbackgrounds[state=repeat]
\defineoverlay[nextpage][\overlaybutton{nextpage}]
\setupbackgrounds[text][text][background=nextpage]
\dorecurse{20}{\input tufte \par}
\stoptext
```

Note that you can move forward from page to page in the resulting PDF file by clicking on each page with the mouse. Now compile this file without setting the background state to `repeat` and note the difference as you click pages with the mouse.

Setting the state was not needed when we used the page background:

```
\setupbackgrounds[page][background=nextpage]
```

The `\dorecurse` macro is handy for testing since it saves us typing. One can nest this macro as in:

```
\dorecurse{20}{\dorecurse{10}{Hello World! }\par}
```

The current step is available in `\recurselevel` and the depth (nesting level) in `\recursedepth`.

6.4 Crossing borders

In many cases, the previously mentioned background areas will suffice, but in the case of more complicated backgrounds, you may wish to use METAPOST to draw graphics that combine or span these areas.

At runtime CONTEXT saves information on the layout that can be picked up by METAPOST. The framework for a page graphic is:

```
StartPage;
  % all kind of commands
StopPage ;
```

Between the `StartPage` and `StopPage` command you have access to a wide range of variables:

page	`PaperHeight PaperWidth`
	`PrintPaperHeight PrintPaperWidth`
	`PageOffset PageDepth`
margins	`TopSpace BackSpace`
text	`MakeupHeight MakeupWidth`
vertical	`TopHeight TopDistance`
	`HeaderHeight HeaderDistance`
	`TextHeight`
	`FooterDistance FooterHeight`
	`BottomDistance BottomHeight`
horizontal	`LeftEdgeWidth LeftEdgeDistance`
	`LeftMarginWidth LeftMarginDistance`
	`TextWidth`

```
RightMarginDistance RightMarginWidth
RightEdgeDistance RightEdgeWidth
```

Since using these variables to construct paths is not that handy the areas are available as predefined paths, which we will demonstrate here.

In **figure 6.2** you see two pages (odd and even) with a background spanning the outer margin and the text area. You can access an area in two ways. The area itself is available as `Area`.

```
StartPage ;
  fill Area[Text][Text] withcolor .85white ;
StopPage ;
```

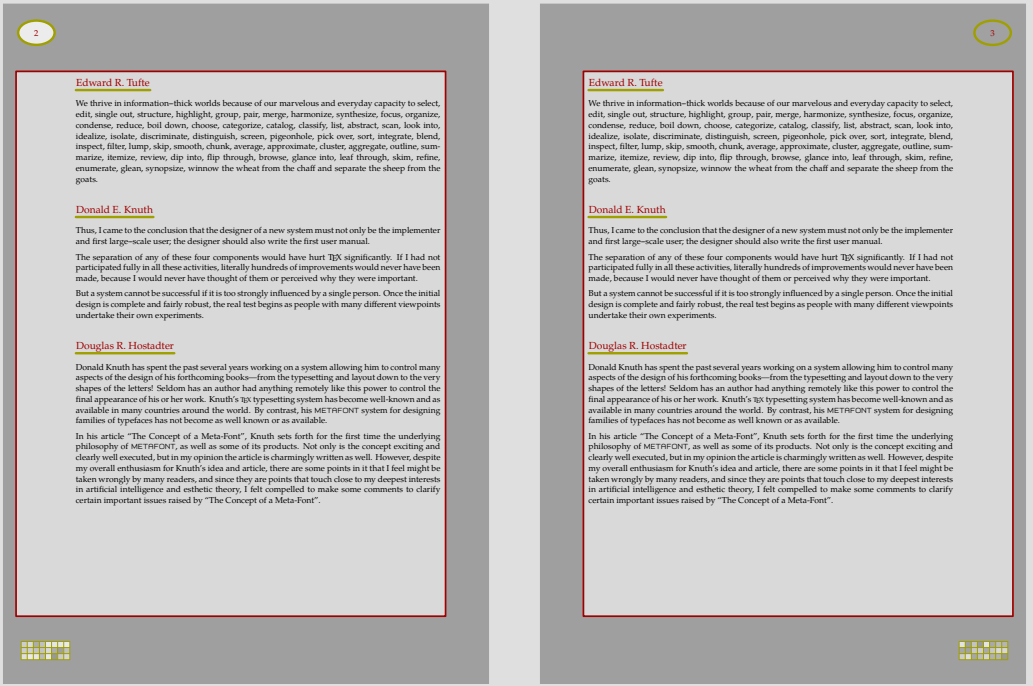
If you use an area this way, you will notice that it is not positioned at the right place. An `Area` is just a rectangle. If you want a positioned area, you should use the `Field` array:

```
StartPage ;
  fill Field[Text][Text] withcolor .85white ;
StopPage ;
```

The location of an area is available in `Location`, so the previous definition is the same as:

```
StartPage ;
  fill Area[Text][Text] shifted Location[Text][Text]
    withcolor .85white ;
StopPage ;
```

The following definition fills and draws the margin and text areas.



even odd

Figure 6.2 A background with combined areas.

```
\startuseMPgraphic{page}
```

```
  StartPage ;
    pickup pencircle scaled 2pt ;
    fill Page                      withcolor .625white ;
    fill Field[OuterMargin][Text] withcolor .850white ;
    fill Field[Text]       [Text] withcolor .850white ;
    draw Field[OuterMargin][Text] withcolor .625red ;
    draw Field[Text]       [Text] withcolor .625red ;
  StopPage ;
\stopuseMPgraphic
```

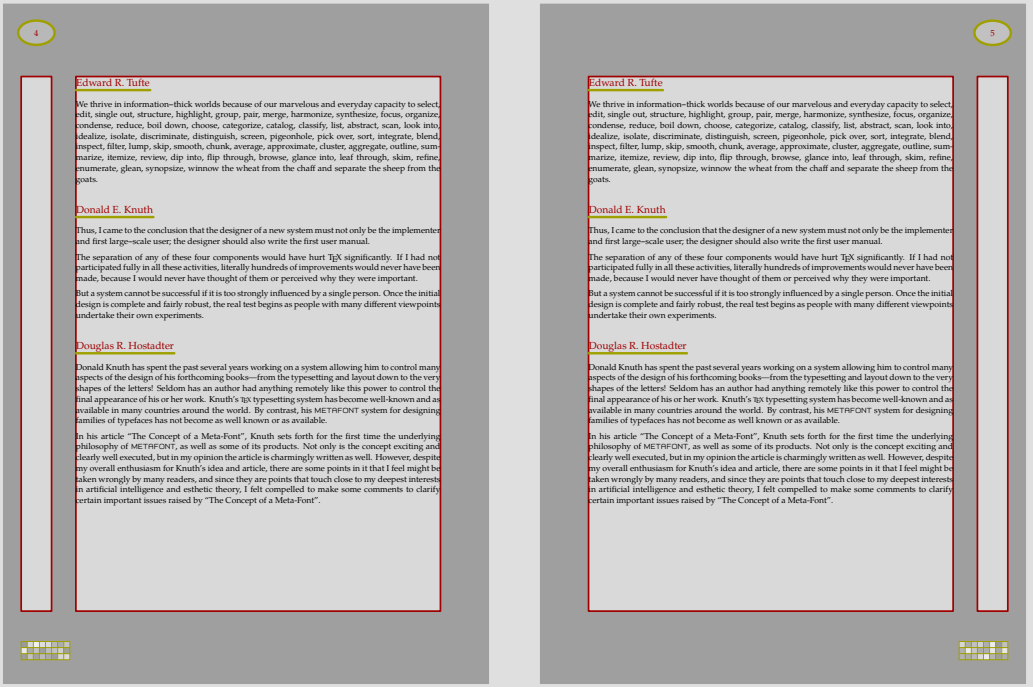
This background is assigned to the page layer by saying:

```
\defineoverlay[page][\useMPgraphic{page}]
\setupbackgrounds[page][background=page]
```

As you can see in **figure 6.3**, the text is typeset rather tightly between the left and right margins.

This can easily be solved by enlarging the areas a bit. The next example demonstrates this on the text area, which is shown in **figure 6.4**.

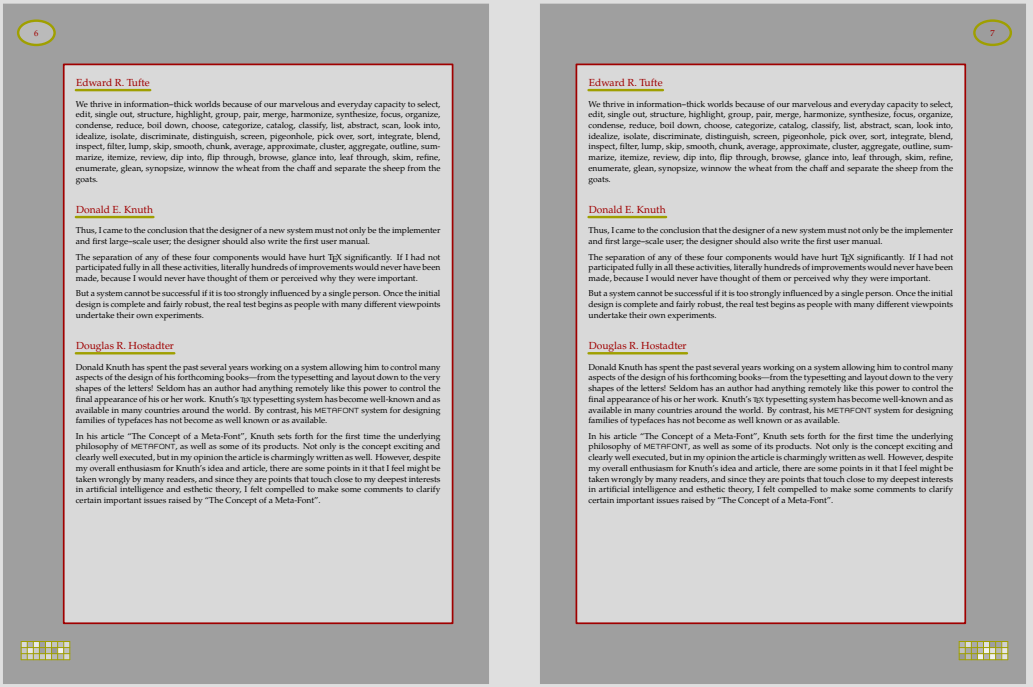
```
\startuseMPgraphic{page}
  StartPage ;
    pickup pencircle scaled 2pt ;
    fill Page                            withcolor .625white ;
    fill Field[Text][Text] enlarged .5cm withcolor .850white ;
    draw Field[Text][Text] enlarged .5cm withcolor .625red ;
  StopPage ;
\stopuseMPgraphic
```



even odd

Figure 6.3 A background with split areas.

The `enlarged` macro can be used like `shifted` and accepts either a numeric or a pair.



even odd

Figure 6.4 A background with enlarged text area.

How do we define a background as in **figure 6.2**? Because `Field` provides us the positioned areas, we can use

the corners of those.

```
\startuseMPgraphic{page}
  StartPage ;
    path Main ;
    if OnRightPage :
      Main := lrcorner Field[OuterMargin][Text] --
              llcorner Field[Text]       [Text] --
              ulcorner Field[Text]       [Text] --
              urcorner Field[OuterMargin][Text] -- cycle ;
    else :
      Main := llcorner Field[OuterMargin][Text] --
              lrcorner Field[Text]       [Text] --
              urcorner Field[Text]       [Text] --
              ulcorner Field[OuterMargin][Text] -- cycle ;
    fi ;
    Main := Main enlarged 6pt ;
    pickup pencircle scaled 2pt ;
    fill Page withcolor .625white ;
    fill Main withcolor .850white ;
    draw Main withcolor .625red ;
  StopPage ;
\stopuseMPgraphic
```

In this definition we calculate a different path for odd and even pages. When done, we enlarge the path a bit. If you want to use different offsets in all directions, you can use moved corner points.

```
\startuseMPgraphic{page}
  StartPage ;
    def somewhere =
      (uniformdeviate 1cm,uniformdeviate 1cm)
    enddef ;
    path Main ;
    Main := Field[Text][Text] lrmoved somewhere --
            Field[Text][Text] llmoved somewhere --
            Field[Text][Text] ulmoved somewhere --
            Field[Text][Text] urmoved somewhere -- cycle ;
    pickup pencircle scaled 2pt ;
    fill Page withcolor .625white ;
    fill Main withcolor .850white ;
    draw Main withcolor .625red ;
  StopPage ;
\stopuseMPgraphic
```

Here we displace the corners randomly which leads to backgrounds like **figure 6.5**. The following definition would have worked as well:

```
\startuseMPgraphic{page}
  StartPage ;
    path Main ; Main := Field[Text][Text] randomized 1cm ;
    pickup pencircle scaled 2pt ;
    fill Page withcolor .625white ;
    fill Main withcolor .850white ;
    draw Main withcolor .625red ;
```

```
  StopPage ;
\stopuseMPgraphic
```

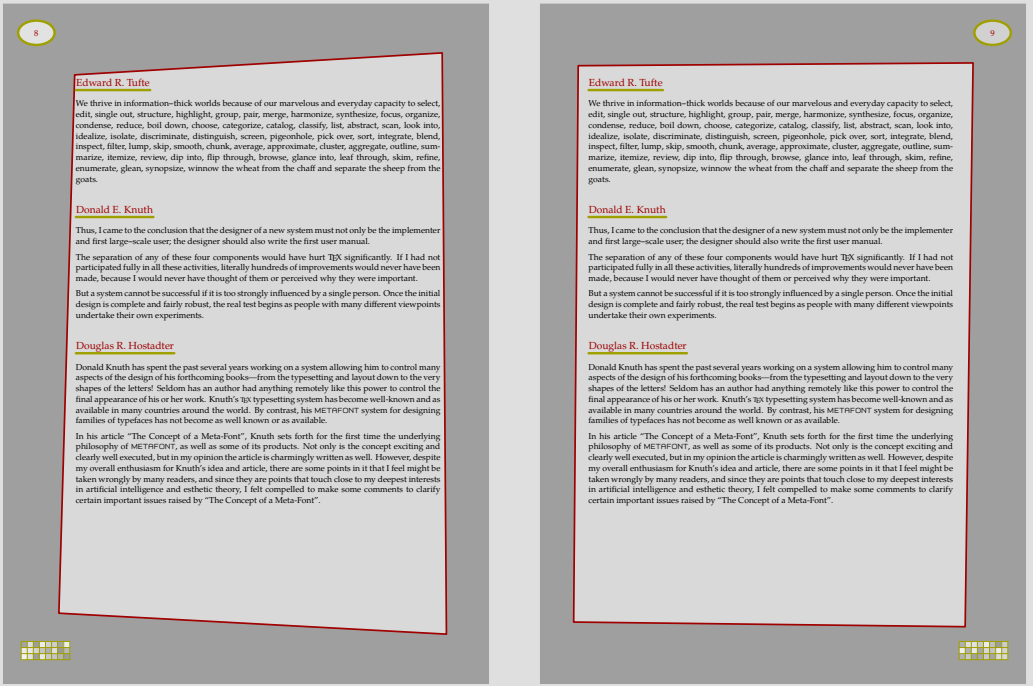
The previous graphics are defined as usable ones, which means that they will be recalculated each page. This is rather inefficient when the shapes don't change. But, using a reusable graphic instead, would result in only one graphic for both pages. Since the layout for the left and right page differs, another method is needed.

Instead of putting the same graphic on the page layer, we put two different ones on the left and right page layer.

```
\defineoverlay[left page] [\useMPgraphic{left page}]
\defineoverlay[right page][\useMPgraphic{right page}]

\setupbackgrounds[leftpage] [background=left page]
\setupbackgrounds[rightpage][background=right page]
```

Now we only have to split the previously defined graphic into two parts. In order to force consistency, we isolate the code that fills and draws. The left page code looks like:

```
\startreusableMPgraphic{left page}
  StartPage ;
    path Main ; Main :=
      llcorner Field[OuterMargin][Text] --
      lrcorner Field[Text]       [Text] --
      urcorner Field[Text]       [Text] --
      ulcorner Field[OuterMargin][Text] -- cycle ;
    \includeMPgraphic{draw page}
  StopPage ;
```



even odd

Figure 6.5 A random text area.

```
\stopreusableMPgraphic
```

The right page text looks similar:

```
\startreusableMPgraphic{right page}
  StartPage ;
    path Main ; Main :=
      lrcorner Field[OuterMargin][Text] --
      llcorner Field[Text]       [Text] --
      ulcorner Field[Text]       [Text] --
      urcorner Field[OuterMargin][Text] -- cycle ;
    \includeMPgraphic{draw page}
  StopPage ;
\stopreusableMPgraphic
```

Watch how we used a reusable graphic first and a simple usable one next. Actually, the next graphic is not a stand alone graphic.

```
\startuseMPgraphic{draw page}
  Main := Main enlarged 6pt ;
  pickup pencircle scaled 2pt ;
  fill Page withcolor .625white ;
  fill Main withcolor .850white ;
  draw Main withcolor .625red ;
\stopuseMPgraphic
```

We have seen some predefined paths and locations. Apart from the `Page` path, they take two arguments that specify their position on the layout grid.

`path Area [][]`	an area similar to a CONTEXT one
`pair Location [][]`	the position of this area
`path Field [][]`	the area positioned at the right place
`path Page`	the page itself

Some less used and more obscure variables are the following.

`numeric Hstep []`	the horizontal distance to the previous area
`numeric Vstep []`	the vertical distance to the previous area
`numeric Hsize []`	the width of an area
`numeric Vsize []`	the height of an area

The array variables are accessed by using constants:

horizontal	vertical
LeftEdge	Top
LeftEdgeSeparator	TopSeparator
LeftMargin	Header
LeftMarginSeparator	HeaderSeparator
Text	Text
RightMarginSeparator	FooterSeparator
RightMargin	Footer
RightEdgeSeparator	BottomSeparator
RightEdge	Bottom

In addition to these, there are `Margin`, `InnerMargin` and `OuterMargin` which adapt themselves to the current odd or even page. The same is true for `Edge`, `InnerEdge` and `OuterEdge`, although these will seldom be used, since interactive documents are always single sided.

We started this chapter with spending a lot of code to simulate the page areas. It will be clear now that in practice this is much easier using the mechanism described here.

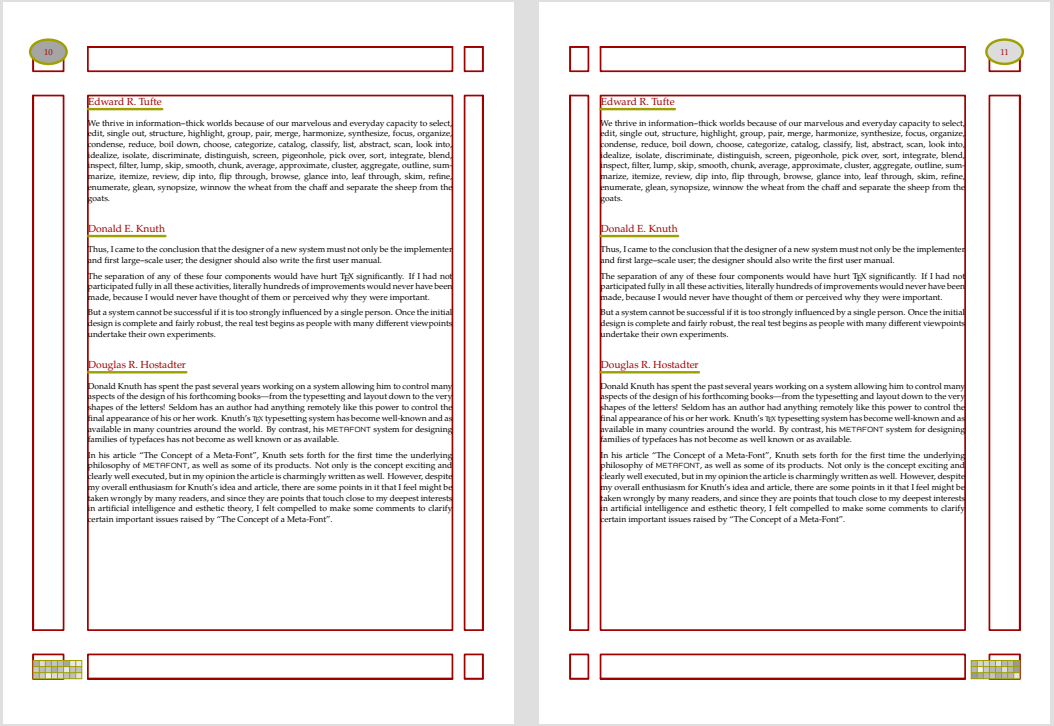
In **figure 6.6** we see all used areas. Areas that are not used are not drawn (which saves some testing). This background was defined as:

```
\startuseMPgraphic{page}
  StartPage
    for i=Top,Header,Text,Footer,Bottom :
      for j=LeftEdge,LeftMargin,Text,RightMargin,RightEdge :
        draw Field[i][j] withpen pencircle scaled 2pt withcolor .625red ;
      endfor ;
    endfor ;
  StopPage
\stopuseMPgraphic
```

We use two nested `for` loops to step over the areas. A `for` loop with a step of 1 will fail, because the indices are defined in a rather special way. On the other hand, the mechanism is rather tolerant, in the sense that `[i][j]` and `[j][i]` are both accepted.

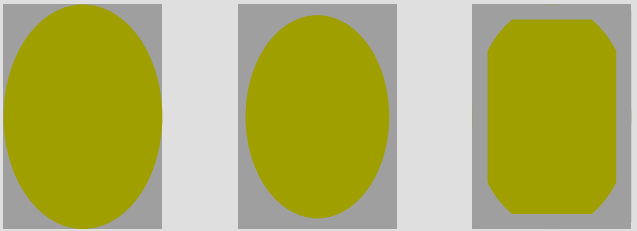
6.5 Bleeding

If you want to share your document all over the world, it makes sense to use a paper format like *letter* or *A4*. In that case, the layout often matches the paper size.



even odd

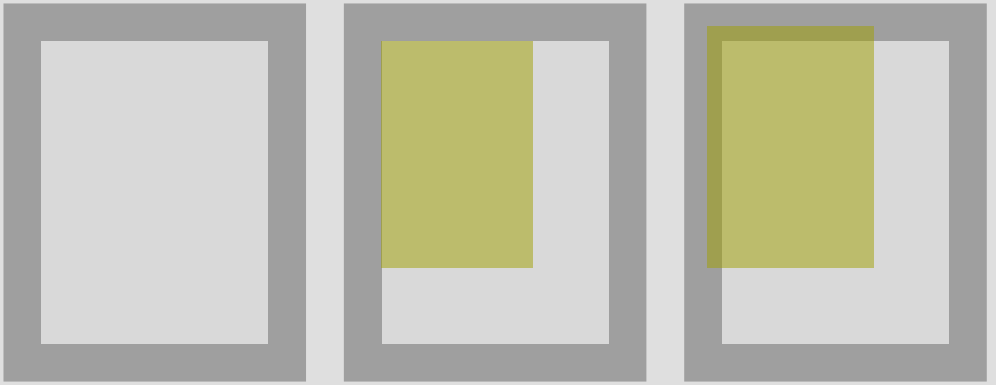
Figure 6.6 A quick way to draw all used areas.



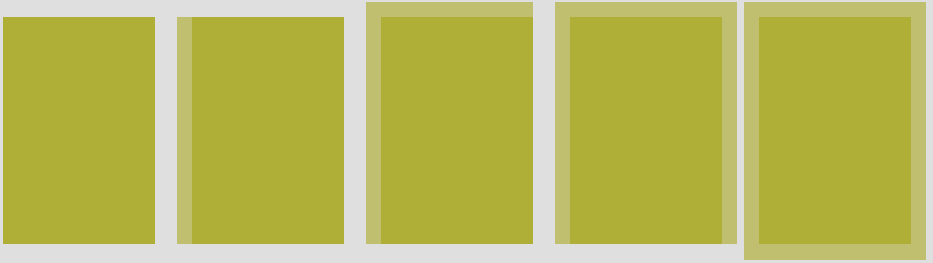
The left picture demonstrates what happens when you have a printer that is capable of printing from edge to edge. If you have such a printer, you're lucky. The middle picture demonstrates what happens if you have a properly set up printing program and/or printer: the page is scaled down so that the content fits into the non printable area of the printer. One reason why printers don't print from edge to edge is that the engine is not that happy when toner or ink ends up next to the page. The third picture shows what happens when a printer simply ignores content that runs over the non printable area. In many cases it's best to make sure that the content leaves a margin of 5mm from the edges.

Books and magazines seldom use the popular desk–top paper sizes. Here the designer determined the paper size and layout more or less independent from the size of the sheet on which the result is printed. Instead of one page per sheet, arrangements of 2 upto 32 or more pages per sheet are made. The process of arranging pages in such a way that these sheets can be folded and combined into books is called page imposition. CONTEXT supports a wide range of page imposition schemes. More information on this can be found in the CONTEXT manuals.

The fact that the sheet on which a page is printed is larger than the page itself opens the possibility to use the full page for content. In that case, especially when you use background graphics, you need to make sure that indeed the page is covered completely. Where in desk top printing you can get away with imperfection simply because the printing engines have their limitations, in professional output you need to be more considerate.



Slightly enlarging a graphic so that it exceeds the natural page limits is called bleeding. Because quite often layout elements have a rectangular nature, METAFUN provides a couple of operations that can save you some work in defining bleeding boxes.

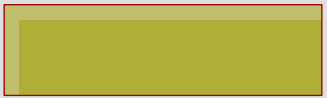


This graphic is generated as follows:

```
path p, q ;
def ShowPath =
  fill p withcolor transparent(1,.5,.625yellow) ;
  fill q withcolor transparent(1,.5,.625yellow) ;
  currentpicture := currentpicture shifted (-25mm,0) ;
enddef ;
p := q := fullsquare xyscaled (2cm,3cm) ; ShowPath ;
p := p leftenlarged   2mm ; ShowPath ;
p := p topenlarged    2mm ; ShowPath ;
p := p rightenlarged  2mm ; ShowPath ;
p := p bottomenlarged 2mm ; ShowPath ;
```

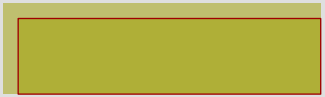
The trick is in the last couple of lines. In addition to the general `enlarged` operator, we have 4 operators that enlarge a rectangle in a certain direction. This means that we can define the original path using dimensions related to the layout, and add bleed strips independently.

```
path p ; p := fullsquare xyscaled (4cm,1cm) ;
path q ; q := p leftenlarged 2mm topenlarged 2mm ;
fill p withcolor transparent(1,.5,.625yellow) ;
fill q withcolor transparent(1,.5,.625yellow) ;
draw boundingbox currentpicture withcolor .625red ;
```



This example demonstrates that when we enlarge a graphic, the bounding box also gets larger. Because this can interfere with the placement of such a graphic, we need to make sure that the bleeding is there but not seen.

```
path p ; p := fullsquare xyscaled (4cm,1cm) ;
path q ; q := p leftenlarged 2mm topenlarged 2mm ;
fill p withcolor transparent(1,.5,.625yellow) ;
fill q withcolor transparent(1,.5,.625yellow) ;
setbounds currentpicture to p ;
draw boundingbox currentpicture withcolor .625red ;
```



There are two more operators: `innerenlarged` and `outerenlarged`. These expand to either `leftenlarged` or `rightenlarged`, depending on the page being left or right hand.

7 Shapes, symbols and buttons

One can use METAPOST to define symbols and enhance buttons. Here we introduce some of the gadgets that come with CONTEXT, as well as explain how to integrate such gadgets yourself.

7.1 Interfacing to TEX

In the early days of METAPOST support in CONTEXT, Tobias Burnus asked me if it was possible to define English rules. What exactly does an english rule look like? Here is one:

As you can see, such a rule has to adapt itself to the current text width, normally `\hsize` in TEX, or on request `\localhsize` in CONTEXT. We need to set the height to a reasonable size, related to the font size, and we also need to take care of proper spacing. Of course we want to run METAPOST as less times as possible, so we need to use unique graphics. Let's start with the graphic.

```
\setupMPvariables
  [EnglishRule]
  [height=1ex,
   width=\localhsize,
   color=darkgray]

\startuniqueMPgraphic{EnglishRule}{height,width,color}
  numeric height ; height = \MPvar{height} ;
  x1 = 0 ; x3 = \MPvar{width} ; x2 = x4 = .5x3 ;
  y1 = y3 = 0 ; y2 := -y4 = height/2 ;
```

```
    fill z1..z2..z3 & z3..z4..z1 & cycle withcolor \MPvar{color} ;
\stopuniqueMPgraphic
```

As you can see, we pass two arguments to the graphic definition. The first argument is the name, the second argument is a comma separated list of variables. This list serves two purposes. First this list is used to create a unique profile for the graphic. This means that when we change the value of a variable, a new graphic is generated that reflects the change. A second purpose of the list is to convert TEX data structures into METAPOST ones, especially dimensions and colors. The graphic itself is not that spectacular. We use & because we don't want smooth connections.

```
\defineblank
  [EnglishRule]
  [medium]

\def\EnglishRule%
  {\startlinecorrection[EnglishRule]
   \setlocalhsize \noindent \reuseMPgraphic{EnglishRule}
   \stoplinecorrection}
```

When setting the variables, we used `\localhsize`. This variable is set by `\setlocalhsize`. We need to use `\noindent`, a rather familiar TEX primitive, that we use here to start a non indented paragraph, being the graphic. The line correction is needed to get the spacing around the rule (graphic) right. We pass a blank skip identifier that is mapped to a convenient medium skip.

```
Why is this called an English line?

\startnarrower
  \EnglishRule
```

```
    Is it because they cannot draw a straight one? This
    could be true after a few strong beers, but then, how
    do Germans draw a line?
  \EnglishRule
\stopnarrower
```

As expected, the rule adapts itself to the current width of the text. The height of the rule in the middle matches the height of a character with no ascenders and descenders.

Why is this called an English line?

Is it because they cannot draw a straight one? This could be true after a few strong beers, but then, how do Germans draw a line?

7.2 Random graphics

Given enough time and paper, we can probably give you some

reasons why METAPOST is fun. To mention a few: you can enhance the layout with graphic ornaments, you can tune your graphics at runtime, and simple high quality graphics can be very effective.

The previous graphics draws exactly 1001 lines in a scratch–numbers–in–a–wall fashion. In 1998, the NTG did a survey among its members, and in the report, we used this fuzzy counter to enhance the rather dull tables.

system	%	# users
Atari	10.4	𝍸 𝍸 |
MSDOS	49.1	𝍸 𝍸 𝍸 𝍸 𝍸 𝍸 𝍸 𝍸 𝍸 𝍸 ||
OS/2	9.4	𝍸 𝍸
MacOS	5.7	𝍸 |
UNIX	51.9	𝍸 𝍸 𝍸 𝍸 𝍸 𝍸 𝍸 𝍸 𝍸 𝍸 𝍸
WINDOWS	64.2	𝍸 𝍸 𝍸 𝍸 𝍸 𝍸 𝍸 𝍸 𝍸 𝍸 𝍸 𝍸 |||

Table 7.1 Operating system (n=106).

Table 7.1 demonstrates how scratch numbers can be used. An interesting side effect is that when you look long enough to these kind of graphics, it looks like the lines are no longer horizontal. This table is defined as follows:

```
\starttabulate[|l|c|l|]
\HL
\NC system  \NC \%   \NC \# users                          \NC\NR
\HL
\NC Atari   \NC 10.4 \NC \useMPgraphic{fuzzycount}{n=11} \NC\NR
\NC MSDOS   \NC 49.1 \NC \useMPgraphic{fuzzycount}{n=52} \NC\NR
\NC OS/2    \NC ~9.4 \NC \useMPgraphic{fuzzycount}{n=10} \NC\NR
\NC MacOS   \NC ~5.7 \NC \useMPgraphic{fuzzycount}{n= 6} \NC\NR
\NC UNIX    \NC 51.9 \NC \useMPgraphic{fuzzycount}{n=55} \NC\NR
\NC WINDOWS \NC 64.2 \NC \useMPgraphic{fuzzycount}{n=68} \NC\NR
```

```
\HL
\stoptabulate
```

You will notice that we pass a variable to the graphic using a second argument. We can access this variable with `\MPvar`. The graphic is defined as usable graphic, because we want to generate a unique random one each time.

```
\startuseMPgraphic{fuzzycount}
   begingroup
   save height, span, drift, d, cp ;
   height := 3/ 5 * \baselinedistance ;
   span   := 1/ 3 * height ;
   drift  := 1/10 * height ;
   pickup pencircle scaled (1/12 * height) ;
   def d = (uniformdeviate drift) enddef ;
   for i := 1 upto \MPvar{n} :
     draw
       if (i mod 5)=0 : ((-d-4.5span,d)--(+d-0.5span,height-d))
       else           : ((-d,+d)--(+d,height-d)) fi
       shifted (span*i,d-drift) ;
   endfor;
   picture cp ; cp := currentpicture ; % for readability
   setbounds currentpicture to
     (llcorner cp shifted (0,-ypart llcorner cp) --
      lrcorner cp shifted (0,-ypart lrcorner cp) --
      urcorner cp -- ulcorner cp -- cycle) ;
   endgroup ;
```

```
\stopuseMPgraphic
```

The core of the macro is the `for` loop. Within this loop, we draw groups of four plus one lines. The draw path's look a bit complicated, but this has to do with the fact that a `mod` returns $0-4$ while we like to deal with $1-5$.

The height adapts itself to the height of the line. The bounding box correction at the end ensures that the baseline is consistent and that the random vertical offsets fall below the baseline.

Because we want to be sure that `n` has a value, we preset it by saying:

```
\setupMPvariables[fuzzycount][n=10]
```

In the table, it makes sense to adapt the drawing to the lineheight, but a more general solution is to adapt the height to the fontsize.

```
height := 3/ 4 * \the \bodyfontsize * \currentfontscale ;
```

In the table we called the graphic directly, but how about making it available as a conversion macro? In CONTEXT this is not that hard:

```
\def\fuzzycount#1{{\tx\useMPgraphic{fuzzycount}{n=#1}}}
\defineconversion[fuzzy][\fuzzycount]
```

Because such a counter should not be that large, we use `\tx` to switch to a smaller font. This also demonstrates how the graphic adapts itself to the font size.

We can now use this conversion for instance in an itemize. To save space we use three columns and no white space between the lines. The `2*broad` directive makes sure that we have enough room for the number.

I. one
II. two
III. three

IIII. four

𝍸 I. six

𝍸 III. eight

𝍸. five

𝍸 II. seven

𝍸 IIII.nine

```
\startitemize[fuzzy,pack,2*broad,columns,three]
\item one   \item two   \item three
\item four  \item five  \item six
\item seven \item eight \item nine
\stopitemize
```

A careful reader will have noticed that the previous definition of the fuzzy counter drawing is not suited to generate the graphics we started with.

```
\useMPgraphic{fuzzycount}{n=1001}
```

This time we want to limit the width to the current \hsize. We only need to add a few lines of code. Watch how we don't recalculate the bounding box when more lines are used.

```
\startuseMPgraphic{fuzzycount}
   begingroup
   save height, vstep, hsize, span, drift, d, cp ;
   height := 3/ 4 * \the \bodyfontsize * \currentfontscale ;
   span   := 1/ 3 * height ;
   drift  := 1/10 * height ;
   hsize  := \the\hsize ;
   vstep  := \the\lineheight ;
   xmax   := hsize div 5span ;
```

```
    pickup pencircle scaled (1/12 * height) ;
    def d = (uniformdeviate drift) enddef ;
    for i := 1 upto \MPvar{n} :
      xpos := ((i-1) mod (5*xmax))*span ;
      ypos := ((i-1) div (5*xmax))*vstep ;
      draw
        if (i mod 5)=0 : ((-d-4.5span,d)--(+d-0.5span,height-d))
        else           : ((-d,+d)--(+d,height-d)) fi
        shifted (xpos,-ypos+d-drift) ;
    endfor;
    picture cp ; cp := currentpicture ;
    if (ypart ulcorner cp - ypart llcorner cp) <= vstep :
      setbounds currentpicture to
        (llcorner cp shifted (0,-ypart llcorner cp) --
         lrcorner cp shifted (0,-ypart lrcorner cp) --
         urcorner cp -- ulcorner cp -- cycle) ;
    fi
    endgroup ;
  \stopuseMPgraphic
```

7.3 Graphic variables

In the previous sections we have seen that we can pass information to the graphic by means of variables. How exactly does this mechanism work?

A nice application of setting up variables for a specific graphic (or class of graphics) is the following. In an email message the author can express his own or the readers expected emotions with so called smilies like: ☺. If you want them in print, you can revert to combinations of characters in a font, but as a TEX user you may want to include nicer graphics.

A convenient way to implement these is to make them into symbols, one reason being that in that case they will adapt themselves to the current font size.

```
Say it with a \symbol [smile]\ or maybe even a \symbol
[smilemore], although seeing too many \dorecurse {10}
{\symbol [smile]\ } \unskip in one text may make you cry.
```

Say it with a ☺ or maybe even a ☺, although seeing too many ☺☺☺☺☺☺☺☺☺☺ in one text may make you cry.

Because we want an efficient implementation, we will use unique graphics, because these will only be generated when the circumstances change.

```
\definesymbol[smile]     [\uniqueMPgraphic{smile}{type=1}]
\definesymbol[smilemore][\uniqueMPgraphic{smile}{type=2}]
```

The definition itself then becomes:

```
\setupMPvariables[smile][type=1,height=1.25ex,color=darkred]

\startuniqueMPgraphic{smile}{type,height,color}
  numeric size ; size := \MPvar{height} ;
  drawoptions(withcolor \MPvar{color}) ;
  pickup pencircle xscaled (size/6) yscaled (size/12) ;
  draw halfcircle rotated -90 scaled size ;
```

```
    pickup pencircle scaled (size/4) ;
    if     \MPvar{type}=1 :
      for i=-1,+1 : draw origin shifted (0,i*size/4) ; endfor ;
    elseif \MPvar{type}=2 :
      for i=-1,+1 : draw origin shifted (-size/2,i*size/4) ; endfor ;
      pickup pencircle scaled (size/6) ;
      draw (size/4,0) -- (-size/4,0) ;
    fi ;
  \stopuniqueMPgraphic
```

We can now change some characteristics of the smilies without the need to redefine the graphic.

```
\setupMPvariables[smile][height=1ex,color=darkred]
```

Say it with a ☺ or maybe even a ☻, although seeing too many ☺☺☺☺☺☺☺☺☺ in one text may make you cry.

In order to keep the smilies unique there is some magic involved, watch the second argument in the next line:

```
\startuniqueMPgraphic{smile}{type,height,color}
```

Because unique graphics often are used in backgrounds, its uniqueness is determined by the overlay characteristics. In our case however the uniqueness is determined by the smilies type, height and color. By explicitly specifying these, we make sure that they count in the creation of the uniqueness stamp.

```
\midaligned{\switchtobodyfont[60pt]\symbol[smile]}
```

Because we use the ex–height, the previous call works as expected.

7.4 Shape libraries

Unfortunately it takes some effort to define graphics, attach them to an overlay, and invoke the background. However, the good news is that since in many cases we want a consistent layout, we only have to do it once. The next table has some hashed backgrounds. (More information about how to define tables can be found in the CONTEXT documentation and Up–To–Date documents.)

right	left	horizontal	vertical

Table 7.2 A hashed table.

This table is defined as:

```
\bTABLE[frame=off,meta:hash:linecolor=darkyellow,offset=3ex]
  \bTR
    \bTD[background=meta:hash:right]      right      \eTD
    \bTD[background=meta:hash:left]       left       \eTD
    \bTD[background=meta:hash:horizontal] horizontal \eTD
    \bTD[background=meta:hash:vertical]   vertical   \eTD
  \eTR
\eTABLE
```

The graphics themselves are defined in a METAPOST module. In this particular example, the macro `some_hash` is defined in the file `mp-back.mp`. This macro takes six arguments:

```
some_hash (width, height, linewidth, linecolor, angle, gap) ;
```

Because we don't want to define a specific overlay for each color and linewidth, we will use variables in the definition of the unique graphic.

```
\startuniqueMPgraphic{meta:hash}{linewidth,linecolor,angle,gap}
  if unknown context_back : input mp-back ; fi ;
  some_hash ( \overlaywidth, \overlayheight ,
              \MPvar{linewidth}, \MPvar{linecolor} ,
              \MPvar{angle}, \MPvar{gap} ) ;
\stopuniqueMPgraphic
```

These variables are preset using `\setupMPvariables`:

```
\setupMPvariables
  [meta:hash]
  [gap=.25\bodyfontsize,
   angle=45,
   linewidth=\overlaylinewidth,
   linecolor=\overlaylinecolor]
```

The last step in this process is to define the different alternatives as overlays:

```
\def\metahashoverlay#1{\uniqueMPgraphic{meta:hash}{angle=#1}}

\defineoverlay[meta:hash:right]     [\metahashoverlay{ +45}]
```

```
\defineoverlay[meta:hash:left]       [\metahashoverlay{ -45}]
\defineoverlay[meta:hash:horizontal][\metahashoverlay{+180}]
\defineoverlay[meta:hash:vertical]   [\metahashoverlay{ -90}]
```

As we can see in the definition of the table, we can pass settings to the \bTABLE command. Actually, we can pass such settings to each command that supports backgrounds, or more precisely \framed. **Table 7.3** is for instance defined as:

```
\bTABLE[frame=off,meta:hash:linewidth=.4pt,align=middle,offset=2ex]
  \bTR
    \bTD[background={meta:hash:left,meta:hash:right},
         meta:hash:linecolor=darkyellow]
        left        \par \& \par right    \eTD
    \bTD[background={meta:hash:horizontal,meta:hash:vertical},
         meta:hash:linecolor=darkred]
         horizontal \par \& \par vertical \eTD
  \eTR
\eTABLE
```

The long names are somewhat cumbersome, but in that way we can prevent name clashes. Also, since the METAPOST interface is english, the variables are also english.

7.5 Symbol collections

In CONTEXT, a symbol can be defined without much coding. The advantage of using symbols is that you can redefine them depending on the situation. So,

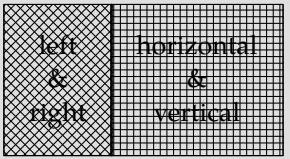


Table 7.3 A double hashed table.

```
\definesymbol [yes] [\em Yes!]
```

creates a symbol, that lets `\symbol[yes]` expand into *Yes!* Since nearly anything can be a symbol, we can also say:

```
\definesymbol [yes] [\mathematics{\star}]
```

or even the already defined symbol $\star$, by saying:

```
\definesymbol [yes] [{\symbol[star]}]
```

It may be clear that we can use a graphic as well:

```
\def\metabuttonsymbol#1{\uniqueMPgraphic{meta:button}{type=#1}}

\definesymbol[menu:left]   [\metabuttonsymbol{101}]
\definesymbol[menu:right]  [\metabuttonsymbol{102}]
\definesymbol[menu:list]   [\metabuttonsymbol{103}]
\definesymbol[menu:index]  [\metabuttonsymbol{104}]
\definesymbol[menu:person] [\metabuttonsymbol{105}]
```

```
\definesymbol[menu:stop]  [\metabuttonsymbol{106}]
\definesymbol[menu:info]  [\metabuttonsymbol{107}]
\definesymbol[menu:down]  [\metabuttonsymbol{108}]
\definesymbol[menu:up]    [\metabuttonsymbol{109}]
\definesymbol[menu:print] [\metabuttonsymbol{110}]
```

Since we have collected some nice buttons in a METAPOST file already, we can use a rather simple definition:

```
\startuniqueMPgraphic{meta:button}{type,size,linecolor,fillcolor}
  if unknown context_butt : input mp-butt ; fi ;
  some_button ( \MPvar{type},
                \MPvar{size},
                \MPvar{linecolor},
                \MPvar{fillcolor} ) ;
\stopuniqueMPgraphic
```

This leaves a few settings:

```
\setupMPvariables
  [meta:button]
  [type=1,
   size=2\bodyfontsize,
   fillcolor=gray,
   linecolor=darkred]
```

These symbols are collected in **table 7.4**, and are called up with the CONTEXT commands like `\symbol[menu:left]`. If needed, we can collect these button symbols in a so called symbol set, which permits us to instantly switch between sets with similar symbols.

Table 7.4 A collection of button symbols.

8 Special effects

Sometimes we want to go beyond METAPOST's native features. Examples of such an extension are CMYK colors, shading and transparency. Although features like this should be used with care, sometimes the documents look and feel can profit from it.

If you don't want the whole graphic, but only a part of it, clipping comes into play. In addition to the standard clipping features, we can use METAPOST to provide a decent clipping path. In this chapter we will uncover the details.

We will also introduce ways to include externally defined graphics and outline fonts. We will demonstrate that within reasonable bounds you can manipulate such graphics.

8.1 Shading

In this section we introduce different kinds of shading. Since METAPOST does not support this feature directly, we have to fall back on a few tricks. For the moment shading is only supported in PDF. In the following examples, we will use the next three colors:

```
\definecolor[a][darkyellow]
\definecolor[b][s=.8]
\definecolor[c][darkred]
```

A shade is a fill with a stepwise change in color. In POSTSCRIPT (level 2), the way this color changes can be circular, linear, or according to a user defined function. Circular and linear shades look like this:

Hi there, I'm Circular!

Whow, this is Linear!

As you can see, the shade lays behind the text, as a background overlay. These overlays are unique METAPOST graphics, so they will adapt themselves to the dimensions of the foreground.

```
\defineoverlay[circular shade][\uniqueMPgraphic{CircularShade}]
\defineoverlay[linear shade]  [\uniqueMPgraphic{LinearShade}]
```

The two framed texts are defined as:

```
\framed
  [background=circular shade,frame=off]
  {\bf \white Hi there, I'm Circular!}
```

and:

```
\framed
  [background=linear shade,frame=off]
  {\bf \white Whow, this is Linear!}
```

We still have to define the graphics. Here we use a macro that takes four arguments: a path, a number identifying the center of shading, and the colors to start and end with.

```
\startuniqueMPgraphic{CircularShade}
  path  p ;
  p := unitsquare xscaled \overlaywidth yscaled \overlayheight ;
  circular_shade(p,0,\MPcolor{a},\MPcolor{b}) ;
\stopuniqueMPgraphic
```

```
\startuniqueMPgraphic{LinearShade}
  path  p ;
  p := unitsquare xscaled \overlaywidth yscaled \overlayheight ;
  linear_shade(p,0,\MPcolor{a},\MPcolor{b});
\stopuniqueMPgraphic
```

The METAPOST macros, `circular_shade` and `linear_shade`, add information to the METAPOST output file, which is interpreted by the converter built in CONTEXT. Shading comes down to interpolation between two or more points or user supplied ranges. A poor mans way of doing this, is to build the graphics piecewise with slightly changing colors. But, instead of 'manually' stepping through the color values, we can use the more efficient and generalized POSTSCRIPT level 2 and PDF level 1.3 shading feature.

circular 0 circular 1 circular 2 circular 3 circular 4

linear 1 linear 2 linear 3 linear 4

linear 5 linear 6 linear 7 linear 8

Shading is not a METAPOST feature, which means that it has to be implemented using so called specials, directives that end up in the output file. Unfortunately these are not coupled to the specific path, which means that we have to do a significant amount of internal bookkeeping. Also, in PDF we have to make sure that the graphics and their resources (being the shading functions) are packaged together.

Because of this implementation, shading may behave somewhat unexpected at times. A rather normal case is the next one, where we place 5 shaded circles in a row.

```
path p ; p := fullcircle scaled 1cm ;
for i=0 step 2cm until 8cm :
  circular_shade(p shifted (i,0),0,\MPcolor{a},\MPcolor{b}) ;
endfor ;
```

At first sight, in the next situation, we would expect something similar, because we simply copy the same circle 5 times. However, due to the way we have implemented shading in CONTEXT, we do indeed copy the circles, but the shade definition is frozen and the same one is used for all 5 circles. This means that the center of the shading stays at the first circle.

```
circular_shade(fullcircle scaled 1cm,0,\MPcolor{a},\MPcolor{b}) ;
picture s ; s := currentpicture ; currentpicture := nullpicture ;
for i=0 step 2cm until 8cm :
  addto currentpicture also s shifted (i,0) ;
endfor ;
```

Unlike TEX, METAPOST does not keep its specials attached to the current path, and flushes them before the graphic data. Since we use these specials to register shading information, it is rather hard to tightly connect a specific shade with a certain fill, especially if an already performed fill is not accessible, which is the case when we copy a picture.

This may seem a disadvantage, but fortunately it also has its positive side. In the next example we don't copy, but reuse an already defined shade. By storing the reference to this shade, and referring to it by using `withshade`, we can use a shade that operates on multiple shapes.

```
sh := define_circular_shade
  (origin,origin,0,8cm,\MPcolor{a},\MPcolor{b}) ;
```

```
for i=0 step 2cm until 8cm :
  fill fullcircle scaled 1cm shifted (i,0) withshade sh ;
endfor ;
```

The low level macro `define_circular_shade` is fed with two pairs (points), two radius, and two colors. The shade is distributed between the colors according to the radius.

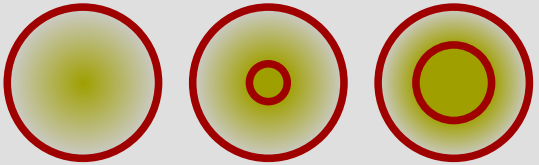
Shading can hardly be called an easy issue. The macros that we provide here are in fact simplifications, which means that at a lower level, one can do more advanced things. Here we limit ourselves to the more common cases. In the previous examples, we used an arrow to indicate the direction and magnitude of the shade. The next macro demonstrates the principles in a different way.

```
def test_shade (expr a, b, ra, rb) =
  pickup pencircle scaled 1mm ;

  color ca ; ca := \MPcolor{a} ;
  color cb ; cb := \MPcolor{b} ;
  color cc ; cc := \MPcolor{c} ;

  path pa ; pa := fullcircle scaled 2ra shifted a ;
  path pb ; pb := fullcircle scaled 2rb shifted b ;

  sh := define_circular_shade(a,b,ra,rb,ca,cb) ;

  fill pb withshade sh ;
  draw pb withcolor cc ;
```

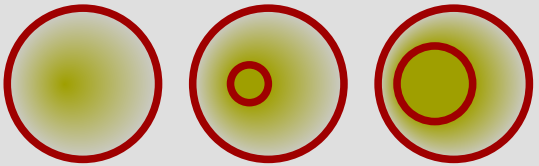
```
  draw pa withcolor cc ;
enddef ;
```

The shade is distributed between two circles, each with a radius and center point. All four can be set, but as the next calls demonstrate, we can normally do with less, which is why we provided the macro with less parameters.

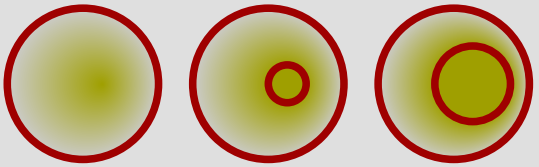
```
test_shade(origin, origin,   0cm, 1cm) ;

test_shade(origin, origin, .25cm, 1cm) ;

test_shade(origin, origin, .50cm, 1cm) ;
```



```
test_shade(origin, origin shifted (.25cm,0),   0cm, 1cm) ;

test_shade(origin, origin shifted (.25cm,0), .25cm, 1cm) ;

test_shade(origin, origin shifted (.25cm,0), .50cm, 1cm) ;
```



```
test_shade(origin shifted (.25cm,0), origin,   0cm, 1cm) ;

test_shade(origin shifted (.25cm,0), origin, .25cm, 1cm) ;

test_shade(origin shifted (.25cm,0), origin, .50cm, 1cm) ;
```

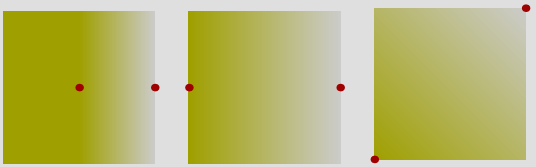


In a similar fashion, we can define a linear shade. This time we only pass two points and two colors.

```
def test_shade (expr a, b) =
  pickup pencircle scaled 1mm ;

  color ca ; ca := \MPcolor{a} ;
  color cb ; cb := \MPcolor{b} ;
  color cc ; cc := \MPcolor{c} ;

  sh := define_linear_shade(a,b,ca,cb) ;

  fill fullsquare scaled 2cm withshade sh ;
  draw a withcolor cc ;
  draw b withcolor cc ;
enddef ;
```

Although one can control shading to a large extend, in practice only a few cases really make sense.

```
test_shade(origin, origin shifted (1cm,0)) ;

test_shade(origin shifted (-1cm,0), origin shifted (1cm,0)) ;

test_shade(origin shifted (-1cm,-1cm), origin shifted (1cm,1cm)) ;
```



8.2 Transparency

In the screen version we use a light gray background color. As a result, some of the transparency methods demonstrated here give unexpected results. The A4 version of this document demonstrates the real effects.

Although transparent colors have been around for some time already, it was only around 2000 that they made it as a high level feature into document format languages like PDF. Supporting such a feature at a higher abstraction level is not only more portable, but also less sensitive for misinterpretation.

```
vardef ColorCircle (expr method, factor, ca, cb, cc) =
  save u, p ; path p ; p := fullcircle shifted (1/4,0) ;
  image
    ( fill p rotated  90 withcolor ca withtransparency (method,factor) ;
      fill p rotated 210 withcolor cb withtransparency (method,factor) ;
      fill p rotated 330 withcolor cc withtransparency (method,factor) ; )
```

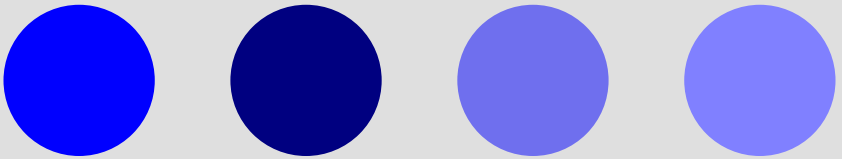
```
enddef ;

draw ColorCircle ("normal", .5, red, green, blue) xsized 3cm ;
currentpicture := currentpicture shifted (-4cm,0) ;
draw ColorCircle ("exclusion", .5, red, green, blue) xsized 3cm ;
currentpicture := currentpicture shifted (-4cm,0) ;
draw ColorCircle ("exclusion", 1, red, green, blue) xsized 3cm ;
```

```
cmykcolor xcyan    ; xcyan    := (1,0,0,0) ;
cmykcolor xmagenta ; xmagenta := (0,1,0,0) ;
cmykcolor xyellow  ; xyellow  := (0,0,1,0) ;

draw ColorCircle ("exclusion", .5, xcyan, xmagenta, xyellow) xsized 3cm ;
```

You can be tempted to use transparency as a convenient way to achieve soft colors. In that case you should be aware of the fact that rendering transparent colors takes more time than normal colors[12]

Fortunatey, METAPOST provides a similar mechanism. The last circle in the following row demonstrates how we can trigger colors proportionally to other colors. Normally `background` is white, but you can set predefined color variables to another value.

```
path p ; p := fullcircle scaled 2cm ;
fill p shifted (0cm,0) withcolor blue  ;
fill p shifted (3cm,0) withcolor .5blue  ;
fill p shifted (6cm,0) withcolor transparent (1,0.5,blue) ;
fill p shifted (9cm,0) withcolor .5[blue,white] ;
```

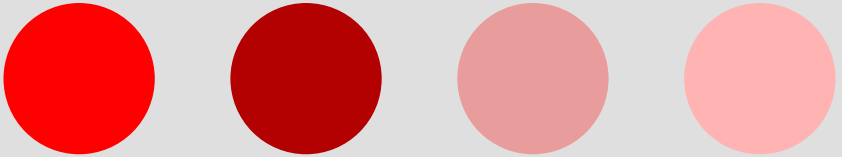


The next series demonstrates that we use the complementary factor `.7` in the METAPOST soft color to achieve the same softness as the `.3` transparency.

```
path p ; p := fullcircle scaled 2cm ;
fill p shifted (0cm,0) withcolor red  ;
fill p shifted (3cm,0) withcolor .7red ;
fill p shifted (6cm,0) withcolor transparent (1,0.3,red) ;
```

[12] When your printer does not support this feature natively, the intermediate (POSTSCRIPT) file send to the printing engine is also larger.

```
fill p shifted (9cm,0) withcolor .7[red,white] ;
```



So far we have applied transparent colors to shapes but text can also be the target. The following works okay in MKII.

```
vardef SampleText (expr t, c) =
  save p ; picture p ;
  p := image (draw t infont "\truefontname{Regular}") ;
  draw (p shifted (- xpart center p,0)) scaled 5 withcolor c;
enddef ;

SampleText ("Much Of This"   , transparent(1, .5, red  )) ;
SampleText ("Functionality"  , transparent(1, .5, green)) ;
SampleText ("Was Written"    , transparent(1, .5, blue )) ;
SampleText ("While Listening", transparent(1, .5, cmyk(1,0,0,0))) ;
SampleText ("To the CD's Of" , transparent(1, .5, cmyk(0,1,0,0))) ;
SampleText ("Tori Amos"      , transparent(1, .5, cmyk(0,0,1,0))) ;
```

The source code of this example illustrates that the CMYK color space is also supported. The `\truefontname` macro communicates the running font from TEX to METAPOST. Instead of such low level code one can of course also use the `textext` macro.

However, as we do the typesetting in TEX in MKIV this is the way to go:

```
vardef SampleText (expr t) =
  draw textext(t) scaled 5 ;
enddef ;

SampleText ("\colored[a=1,t=.5,r=1]{Much Of This}") ;
SampleText ("\colored[a=1,t=.5,g=1]{Functionality}") ;
SampleText ("\colored[a=1,t=.5,b=1]{Was Written}") ;
SampleText ("\colored[a=1,t=.5,c=1]{While Listening}") ;
SampleText ("\colored[a=1,t=.5,m=1]{To the CD's Of}") ;
SampleText ("\colored[a=1,t=.5,y=1]{Tori Amos}") ;
```

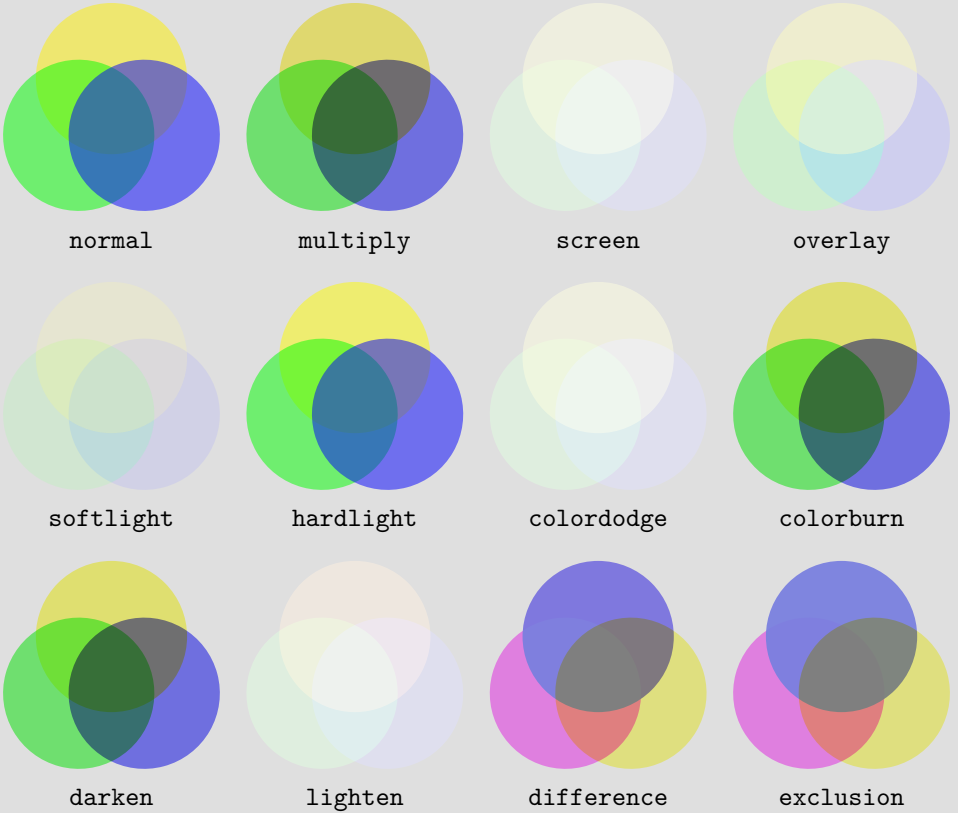
As expected we get:

Currently the 12 in PDF available transparency methods are supported.[13] You can use both numbers and names. As you may expect, both CONTEXT and METAFUN support transparency in the same way. **Figure 8.1** shows how the method affects the result.

In CONTEXT a transparent color is defined in a similar way as 'normal' colors. The transparency method is specified with the `a` key (either by number or by name) and the factor `t`.

```
\definecolor [tred]   [r=1,t=.5,a=exclusion]
```

[13] In the future we may also support more control over the individual methods.


normal
multiply
screen
overlay
softlight
hardlight
colordodge
colorburn
darken
lighten
difference
exclusion

```
\definecolor [tgreen] [g=1,t=.5,a=exclusion]
\definecolor [tblue]  [b=1,t=.5,a=exclusion]
```

Both keys are needed. You can define your own symbolic names using:

```
\definetransparency [myowndefault] [1]
```

The `\MPcolor` macro passes a color from CONTEXT to METAPOST, including the transparency specification.

```
\definecolor [tred]   [r=1,t=.5,a=exclusion]
\definecolor [tgreen] [g=1,t=.5,a=exclusion]
\definecolor [tblue]  [b=1,t=.5,a=exclusion]
```

Of course this also works well for CMYK colors.

```
\definecolor[tred]  [c=1,k=.2,t=.5,a=1]
\definecolor[tgreen][m=1,k=.2,t=.5,a=1]
\definecolor[tblue] [y=1,k=.2,t=.5,a=1]
```

Gray scales work as well:

```
\definecolor[ta][s=.9,t=.7,a=11]
\definecolor[tb][s=.7,t=.7,a=11]
\definecolor[tc][s=.5,t=.7,a=11]
```

We apply this to some text. By using an overlay we can conveniently explore the difference in fonts.

```
draw textext("\color[ta]{\tf Hello}") scaled 5 ;
draw textext("\color[tb]{\bf Hello}") scaled 5 ;
draw textext("\color[tc]{\sl Hello}") scaled 5 ;
```

8.3 Clipping

In this section we will use the graphic representation (although simplified) of a Dutch cow to demonstrate clipping.

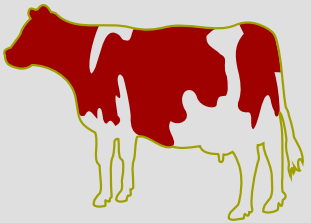


Figure 8.2 A cow.

Since this cow is defined as a METAPOST graphic, we use the suffix `mps` instead of `eps` or a number, although CONTEXT will recognize each as being METAPOST output. The placement of the cow is defined as:

```
\placefigure
  {A cow.}
  {\externalfigure[cow.mps][width=4cm]}
```

Clipping is combined with a matrix, as in **figure 8.3**. The content to be clipped is divided in `nx` by `ny` rectangles. For instance, `nx=5` and `ny=8` will produce a 40 cell grid with 5 columns of 8 rows.

Figure 8.3 A clipped cow.

Here we have divided the cow in six cells, so that we can clip its head and tail. This kind of clipping enables you to zoom in or focus on a specific part of a graphic.

```
\setupclipping[nx=3,ny=2]
\startcombination
  {\clip[x=1,y=1]{\externalfigure[cow.mps][width=4cm]}} {1,1}
  {\clip[x=3,y=1]{\externalfigure[cow.mps][width=4cm]}} {3,1}
\stopcombination
```

Alternatively, we can specify a `width`, `height`, `hoffset` and `voffset`, as demonstrated in **figure 8.4**.

Figure 8.4 Another clipped cow.

```
\placefigure
  [here][fig:clipped cow 2]
  {Another clipped cow.}
  {\clip
     [width=2cm,height=2cm,hoffset=0cm,voffset=0cm]
     {\externalfigure[cow.mps][width=4cm]}}
```

Because METAPOST supports clipping, it will be no surprise that both techniques can be combined. In the next example we will zoom in on the head of the cow. We also use this opportunity to demonstrate how you can package a clip in a figure definition.

```
\startMPclip{head clip}
 w := \width ; h := \height ;
```

```
 clip currentpicture to
  ((0,h)--(w,h){down}..{left}(0,0)--cycle) ;
\stopMPclip

\placefigure
  [here][fig:circular clipped cowhead]
  {A quarter circle applied to a cows head.}
  {\ruledhbox
     {\clip
        [nx=2,ny=2,x=1,y=1,mp=head clip]
        {\externalfigure[cow.mps][width=4cm]}}}
```

A more advanced clip is demonstrated in **figure 8.5**. We added `\ruledhbox` to demonstrate the dimensions of the resulting graphic. Putting something in such a ruled box is often a quick way to test spacing.

Figure 8.5 A quarter circle applied to a cows head.

Although a clip path definition can contain any METAPOST command, even graphics, it must contain at least one clipping path. The first one encountered in the resulting graphic is used. In the example we used a path that is built out of three subpaths.

```
(0,h)--(w,h){down}..{left}(0,0)--cycle
```

We start in the top left corner and draw a straight line. Next we draw a curve to the origin. Directives like `down` and `right` force the curve in a certain direction. With `cycle` we close the path. Because we use this path as a clipping path, we use `clip` instead of `draw` or `fill`.

Clipping as such is not limited to graphics. Take for instance the text buffer:

```
\startbuffer[sample]
\framed
  [align=middle,width=4cm,background=screen,frame=off]
  {A \METAPOST\ clip is not the same as a video clip,
   although we can use \METAPOST\ to produce a video clip.}
\stopbuffer
```

We can call up such a buffer as if it were an external figure. **Figure 8.6** shows the result. This time we use a different clip path:

```
\startMPclip{text clip}
  clip currentpicture to fullcircle shifted (.5,.5)
    xscaled \width yscaled \height ;
\stopMPclip
```

To load a buffer, we have to specify its name and type, as in:

```
\placefigure
  [here][fig:clipped text 1]
  {A clipped buffer (text).}
  {\clip
     [nx=1,ny=1,mp=text clip]
     {\externalfigure[sample][type=buffer,width=4cm]}}
```

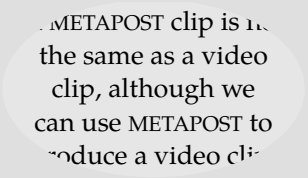


Figure 8.6 A clipped buffer (text).

The next few lines demonstrate that we can combine techniques like backgrounds and clipping.

```
\startuseMPgraphic{clip outline}
 draw fullcircle
   xscaled \overlaywidth yscaled \overlayheight
   withpen pencircle scaled 4mm
   withcolor .625red ;
\stopuseMPgraphic

\defineoverlay[clip outline][\useMPgraphic{clip outline}]

\placefigure
```

```
[here][fig:clipped text 2]
{A clipped buffer (text).}
{\framed
   [background=clip outline,offset=overlay,frame=off]
   {\clip
      [nx=1,ny=1,mp=text clip]
      {\externalfigure[sample][type=buffer,width=4cm]}}}
```

We could have avoided the `\framed` here, by using the `clip outline` overlay as a background of the sample. In that case, the resulting linewidth would have been 2.5 mm instead of 5 mm, since the clipping path goes through the center of the line.

Figure 8.7 A clipped buffer (text).

In most cases, the clip path will be a rather simple path and defining such a path every time you need it, can be annoying. **Figure 8.8** shows a collection of predefined clipping paths. These are available after loading the METAPOST clipping library.

```
\useMPlibrary[clp]
```

We already saw how the circular clipping path was defined. The diamond is defined in a similar way, using the predefined path `diamond`:

```
\startMPclip{diamond}
  clip currentpicture to unitdiamond
    xscaled \width yscaled \height ;
\stopMPclip
```

The definition of the negated ellipse (`negellipse`) uses the primary `peepholed`. This primary is defined in one of the METAPOST modules that come with CONTEXT.

```
\startMPclip{negellipse}
  clip currentpicture to (unitcircle peepholed unitsquare)
    xscaled \width yscaled \height ;
\stopMPclip
```

The definition of `peepholed` is rather dirty and using `peepholed` is restricted to well defined situations (like here). It's called a primary because it acts as an operator at the same level as `*` and `scaled`.

8.4 Including graphics

This document demonstrates that it is no big problem to include METAPOST graphics in a TEX document. But how about including graphics in a METAPOST picture? In this section we will explore a couple of macros that provide you this feature.

Before we go into details, we introduce a very impressive program called PSTOEDIT by Wolfgang Glunz. This program runs on top of GHOSTSCRIPT and is able to convert POSTSCRIPT code into other formats, among them

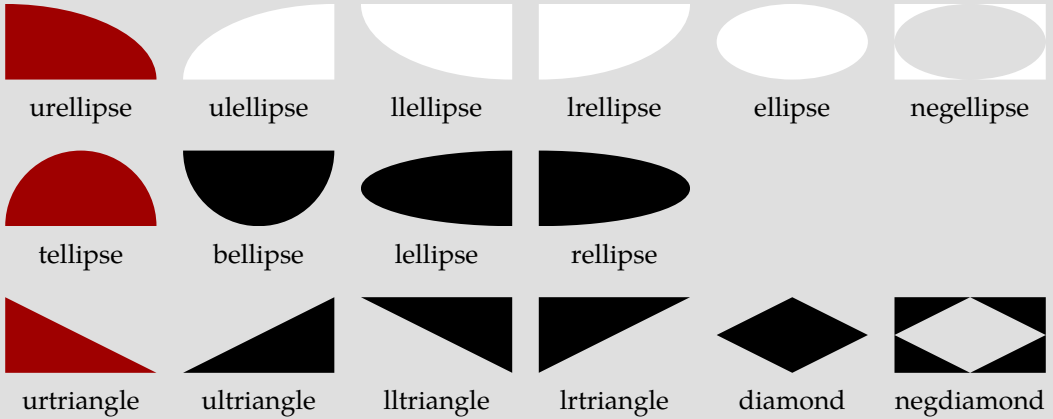


Figure 8.8 A collection of predefined clipping paths.

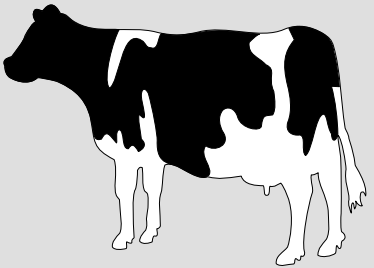
METAPOST (that part of the PSTOEDIT code is due to Scott Pakin). Some of the graphics that we use in this section are produced that way. For us, the next call works well, but the exact call may differ per version or platform.

```
pstoedit -ssp -dt -f mpost yourfile.ps newfile.mp
```

We have converted the Dutch cow that shows up in many CONTEXT documents into METAPOST using this program. The resulting METAPOST file encapsulates the cow in METAPOST figure 1: `beginfig(1)`. Of course you can process this file like any other, but more interesting is to use this code in an indirect way.

```
loadfigure "mycow.mp" number 1 scaled .5 ;
```

This call will load figure 1 from the specified METAPOST file, in such a way that there is no interference with the current (encapsulating) figure.

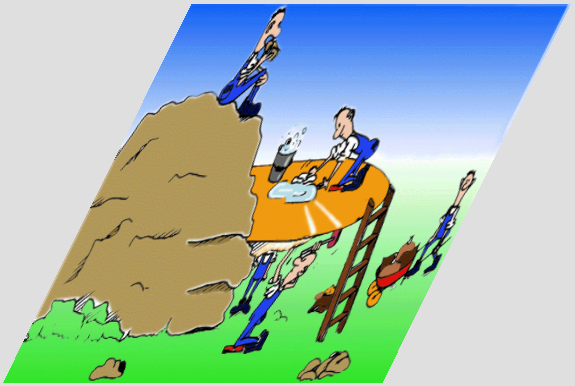


Because this graphic is the result from a conversion, there are only paths. If you want to import a more complex graphic, you need to make sure that the variables used in there do not conflict with the one currently in use.

METAPOST is good in drawing vector graphics, but lacks natural support for bitmaps, but the next macro offers a way out. This macro permits you to include graphics in PNG, PDF, and JPG format, or more precise: those formats supported by PDFTEX.

```
draw externalfigure "hacker.png" scaled 5cm shifted (-6cm,0) ;
draw externalfigure "hacker.png" scaled 5cm slanted .5 ;
```

You can apply the usual transformations, but only those applied directly will be taken into account. This means that you (currently) cannot store external figures in picture variables in order to transform them afterwards.



Although you are limited in what you can do with such graphics, you can include them multiple times with a minimum of overhead. Graphics are stored in objects and embedded only once.

```
numeric s ; pair d, c ;
for i := 1 upto 5 :
  s := 3cm randomized 1cm ;           % size of picture
  c := .5(s,s) ;                      % center of picture
  d := (2cm*i,.5cm) randomized .5cm ; % displacement
  draw externalfigure "hacker.png"
    scaled s rotatedaround (c,0 randomized 30) shifted d ;
endfor ;
```

Because we cannot store the graphic in a picture and scale afterwards, we calculate the scale in advance, so that we can rotate around the center.

As long as you don't mess around with a stored external figure, you're safe. The following example demonstrates how we can combine two special driven features: figure inclusion and shading.

```
picture p ;
p := externalfigure "hacker.png" scaled 150pt ;
clip p to unitcircle scaled 150pt ;
circular_shade(boundingbox p enlarged 10pt, 0, .2red, .9red) ;
addto currentpicture also p ;
```

We end this section with a few more words to METAPOST inclusion. It may seem that in order to use the features discussed here, you need to use CONTEXT as typesetting engine. This is not true. First of all, you can use the small TEX package MPTOPDF (described in another manual) or you can make small CONTEXT files with one page graphics. The advantage of the last method is that you can manipulate graphics a bit.

```
\setupcolors[cmyk=yes,rgb=no,state=start]

\starttext

\startMPpage[offset=6pt]
  loadfigure "niceone.mp" number 10 ;
\stopMPpage

\stoptext
```

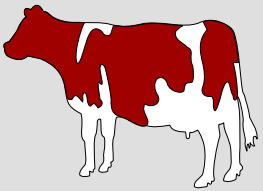
The resulting PDF file can be included as any other graphic and has the advantage that it is self contained.

8.5 Changing colors

One of the advantages of METAPOST graphics is that it is rather easy to force consistency in colors and line widths. You seldom can influence third party graphics that way, but we can use some METAFUN trickery to get around this limitation.

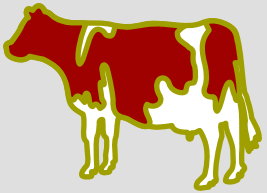
Say that we want a red cow instead of a black one. The following code does the trick:

```
loadfigure "mycow.mp" number 1 scaled .35 ;
refill currentpicture withcolor .625red ;
```



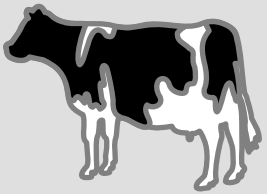
In a similar way we can influence the width and colors of the lines.

```
loadfigure "mycow.mp" number 1 scaled .35 ;
refill currentpicture withcolor .625red ;
redraw currentpicture withpen pencircle scaled 2pt withcolor .625yellow ;
```



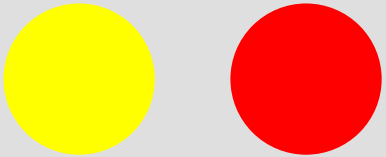
Of course we can also use the more fancy features of METAFUN, like transparency and shading.

```
loadfigure "mycow.mp" number 1 scaled .35 ;
numeric sh ; sh := define_linear_shade
  (llcorner currentpicture,urcorner currentpicture,.625red, .625yellow) ;
refill currentpicture withshade sh ;
redraw currentpicture withpen pencircle scaled 2pt withcolor .5white;
```



Before we show a next trick, we draw a few circles.

```
fill fullcircle scaled 2cm                 withcolor yellow ;
fill fullcircle scaled 2cm shifted (3cm,0) withcolor red ;
```



The yellow and red color do not match the main document colors, but this is no problem: we can remap them, without spoiling the original definition.

```
fill fullcircle scaled 2cm                withcolor yellow ;
fill fullcircle scaled 2cm shifted (3cm,0) withcolor red ;

remapcolor(yellow,.625yellow) ;
remapcolor(red   ,.625red) ;
recolor currentpicture ;
resetcolormap ;
```

We can combine the inclusion technique with remapping colors. This time using an artist impression of one of Hasselts Canals (gracht in Dutch).

```
loadfigure "gracht.mp" number 1 scaled .5 ;
```

If you think that the sky is too bright in this picture, and given that you also know which color is used, you can fool the reader by remapping a few colors.

```
loadfigure "gracht.mp" number 1 scaled .5 ;

color skycolor   ; skycolor   := (0.8,0.90,1.0) ;
color watercolor ; watercolor := (0.9,0.95,1.0) ;
remapcolor(skycolor  ,.8skycolor  ) ;
remapcolor(watercolor,.8watercolor) ;
recolor currentpicture ;
resetcolormap ;
```

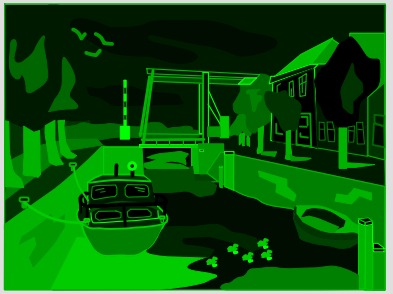
Including another METAPOST graphic, refilling, redrawing, and recoloring are all relatively simple features that use no real tricks. Opposite to the next feature, which is implemented using the METAPOST special driver that comes with CONTEXT.

METAPOST is not really meant for manipulating graphics, but the previous examples demonstrated that we have some control over individual colors. In the next series of examples we will treat the picture as a whole. First we invert the colors using `inverted`.

```
loadfigure "gracht.mp" number 1 scaled .5 ;
addto currentpicture also
  inverted currentpicture
  shifted (bbwidth(currentpicture)+.5cm,0) ;
```


This is a special case of `uncolored`. In the next example we explicitly specify the color.

```
loadfigure "gracht.mp" number 1 scaled .5 ;
addto currentpicture also
  (currentpicture uncolored green)
  shifted (bbwidth(currentpicture)+.5cm,0) ;
```

You can also multiply each color using `softened`. In the next sample, the colors have 80% of their value.

```
loadfigure "gracht.mp" number 1 scaled .5 ;
addto currentpicture also
  (currentpicture softened .8)
  shifted (bbwidth(currentpicture)+.5cm,0) ;
```


You can also use this operator to harden colors, simply by providing a value larger than 1. Keep in mind that colors are clipped at 1 anyway.

```
loadfigure "gracht.mp" number 1 scaled .5 ;
addto currentpicture also
  (currentpicture softened 1.2)
  shifted (bbwidth(currentpicture)+.5cm,0) ;
```


By providing a triplet, you can treat each color component independently.

```
loadfigure "gracht.mp" number 1 scaled .5 ;
addto currentpicture also
  (currentpicture softened (.7,.8,.9))
  shifted (bbwidth(currentpicture)+.5cm,0) ;
```


After these examples your are probably sick of seeing this picture in color, so let's turn the colors into a weigthed grayscales (in a way similar to the way black and white television treated color).

```
loadfigure "gracht.mp" number 1 scaled .5 ;
addto currentpicture also
  grayed currentpicture
  shifted (bbwidth(currentpicture)+.5cm,0) ;
```


8.6 Outline fonts

Outline fonts don't belong to METAPOST's repertoire of features. Nevertheless we can simulate this in a reasonable way. We will not discuss all details here, because most details are covered in the MAKEMPY manual.

The macro responsible for outline fonts is `graphictext`. The first argument should be a string. This string is processed by TEX. Additionally you can provide transformation directives and color specifications. The next example demonstrates this.

```
graphictext "\bf Fun" scaled 4 zscaled (1,1.5)
  withdrawcolor blue
  withfillcolor .5white
  withpen pencircle scaled 5pt
```

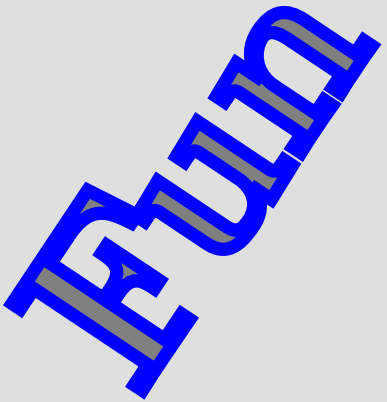
Once the text is typeset by TEX, it is converted to POSTSCRIPT and converted into METAPOST by the PSTOEDIT program. The resulting graphic is imported, analyzed, and processed conforming the specifications of `graphictext`.

By default the shapes are filled after they are drawn. This has the advantage that in characters built out of pieces, disturbing lines fragments are covered. The drawback is that you get only half the linewidth. You can reverse the drawing order by adding the `reversefill` directive. The previous graphic then comes out as:

```
graphictext "\bf Fun" scaled 4 zscaled (1,1.5)
  reversefill
```

```
withdrawcolor blue
withfillcolor .5white
withpen pencircle scaled 5pt
```

The `reversefill` directive can be countered by `outlinefill`.



The next example is taken from the MAKEMPY manual. It demonstrates that you can combine TEX's powerful line breaking with METAPOST's graphic capabilities.

```
\startuseMPgraphic{quotation}
  picture one ; one := image ( graphictext
    \MPstring{text}
    scaled 1.5
    withdrawcolor .625blue
```

```
    withfillcolor .625white
    withpen pencircle scaled 1pt ; ) ;
  picture two ; two := image ( graphictext
    \MPstring{author}
    scaled 2
    withdrawcolor .625red
    withfillcolor .625white
    withpen pencircle scaled 2pt ; ) ;
  currentpicture := one ;
  addto currentpicture also two
    shifted lrcorner one
    shifted - 1.125 lrcorner two
    shifted (0, - 1.250 * ypart urcorner two)  ;
  setbounds currentpicture to boundingbox currentpicture enlarged 3pt ;
\stopuseMPgraphic
```

In this graphic, we have two text fragments, the first one is a text, the second one the name of the author. We combine the quotation and author into this graphic using the following definitions:

```
\setMPtext{text}  {\vbox{\hsize 8.5cm \input zapf }}
\setMPtext{author}{\hbox{\sl Hermann Zapf}}
```

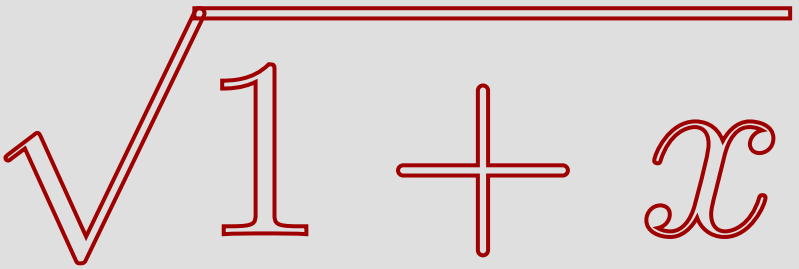
These definitions assume that the file `zapf.tex` is present on the system (which is the case when you have installed CONTEXT). The graphic can now be typeset using the following call:

```
\placefigure
  {A text does not need to be an outline in order to be
```

```
    typeset in an outline font.}
  {\useMPgraphic{quotation}}
```

The quality of the output depends on how the glyphs are constructed. For instance, in TEX, math symbols are sometimes composed of glyph fragments and rules.

```
graphictext
  "$$\sqrt{1+x}$$"
  scaled 8
  withdrawcolor .625red
  withpen pencircle scaled 1.5pt
```



This is not really a problem because we can also fill the shapes. It is the reason why the fill is applied after the draw and in such case the effective line width is half the size specified.

```
graphictext
  "$$\left({{\sqrt{1+x}}\over{\sqrt{2+x^2}}}\right)$$"
  scaled 4
```

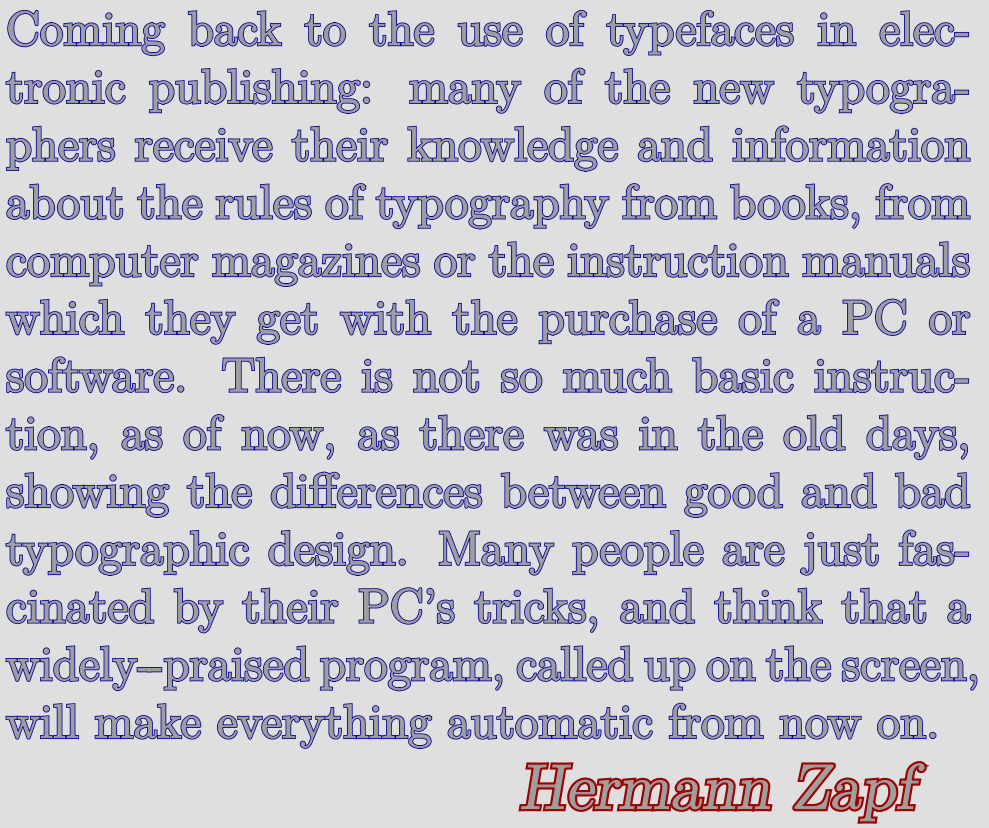
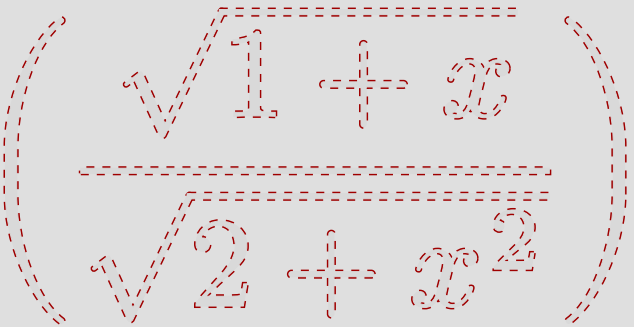


Figure 8.9 A text does not need to be an outline in order to be typeset in an outline font.

```
dashed evenly
withdrawcolor .625red
withfillcolor .850white
withpen pencircle scaled 1.5pt
```

In this example we also use a dashed line. Instead of normal colors, we could have used shades or transparent colors.



Instead of supplying the text directly, you can use the indirect method. This permits you to process rather complex data without messing up your METAPOST code.

```
\setMPtext {some math}%
  {\usemodule[mathml]
   \xmlprocessdata
     {main}
     {<math xmlns='http://www.w3c.org/mathml' version='2.0'>
        <apply> <log/>
```

```
          <logbase> <cn> 2 </cn> </logbase>
          <apply> <plus/>
            <ci> x </ci>
            <cn> 1 </cn>
          </apply>
        </apply>
      </math>}
    {}}
```

Here we feed some MATHML into TEX, which in turn shows up as a METAPOST graphic.

```
graphictext
  \MPstring{some math}
  scaled 4
  withdrawcolor .625red
  withfillcolor .625white
  withpen pencircle scaled 1.5pt
```

9 Functions

METAPOST provides a wide range of functions, like `sind` *and* `floor`*. We will discuss most of them here and define a few more. We will also introduce a few methods for drawing grids and functions.*

9.1 Overview

What follows is a short overview of the functions that can be applied to numeric expressions and strings. Functions that operate on pairs, colors, paths and pictures are discussed in other chapters.

First of all we have `+`, `-`, `/` and `*`. For as far as is reasonable, you can apply these to numerics, pairs and colors. Strings can be concatenated by `&`.

Pythagorean addition is accomplished by `++`, while Pythagorean subtraction is handled by `+-+`. The `**` operator gives you exponentiation. The nature of the METAPOST language is such that you can easily define interesting functions using such symbols.

The logarithmic functions are based on bytes. This makes them quite accurate but forces you to think like a computer.

`mexp(x)`	expential function with base 256
`mlog(x)`	logarithm with base 256

The basic goniometric functions operate on degrees, which is why they have a 'd' in their name.

`cosd(x)`	cosine of x with x in degrees
`sind(x)`	sine of x with x in degrees

There are three ways to truncate numbers. The `round` function can also handle pairs and colors.

`ceiling(x)`	the least integer greater than or equal to x
`floor(x)`	the greatest integer less than or equal to x
`round(x)`	round each component of x to the nearest integer

Of course we have:

`x mod y`	the remainder of x/y
`x div y`	the integer part of x/y
`abs(x)`	the absolute value of x
`sqrt(x)`	the square root of x
`x dotprod y`	the dot product of two vectors

What would life be without a certain randomness and uncertainty:

`normaldeviate`	a number with mean 0 and standard deviation 1
`uniformdeviate(x)`	a number between zero and x

The following functions are actually macros:

`decr(x,n)`	decrement x by n
`incr(x,n)`	increment x by n

`max(a,b,..)`	return the maximum value in the list
`min(a,b,..)`	return the minimum value in the list

The `min` and `max` funtions can be applied to numerics as well as strings.

The following functions are related to strings:

`oct s`	string representation of an octal number
`hex s`	string representation of a hexadecimal number
`str s`	string representation for a suffix
`ASCII s`	ASCII value of the first character
`char x`	character of the given ASCII code
`decimal x`	decimal representation of a numeric

With `substring (i,j) of s` you can filter the substring bounded by the given indices from the given string.

In METAFUN we provide a few more functions (you can take a look in `mp-tool.mp` to see how they are defined. You need to be aware of very subtle rounding errors. Normally these only show up when you reverse an operation. This is a result from mapping to and from internal quantities.

`sqr(x)`	x^2
`log(x)`	$\log(x)$
`ln(x)`	$\ln(x)$
`exp(x)`	e^x
`pow(x, p)`	x^p
`inv(x)`	$1/x$

The following sine and cosine functions take radians instead of angles in degrees.

```
sin(x)  asin(x)  invsin(x)
cos(x)  acos(x)  invcos(x)
```

There are no tangent functions, so we provide both the radian and degrees versions:

```
tan(x)  tand(x)
cot(x)  cotd(x)
```

Here are a couple of hyperbolic functions.

```
sinh(x)  asinh(x)
cosh(x)  acosh(x)
```

We end with a few additional string converters.

```
ddecimal x   decimal representation of a pair
dddecimal x  decimal representation of a color
condition x  string representation of a boolean
```

9.2 Grids

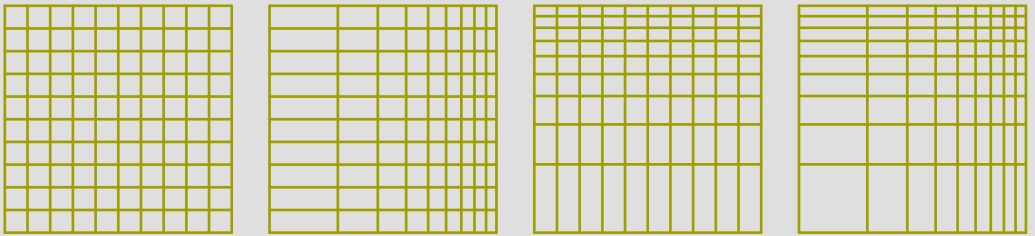
Some day you may want to use METAPOST to draw a function like graphic. In the regular TEX distributions you will find a module `graph.mp` that provides many ways to accomplish this. For the moment, METAFUN does not provide advanced features with respect to drawing functions, so this section will be relatively short.

When drawing a functions (for educational purposes) we need to draw a couple of axis or a grid as well as a shape. Along the axis we can put labels. For this we can use the METAPOST package `format.mp`, but this does not integrate that well into the way METAFUN deals with text typeset by TEX.

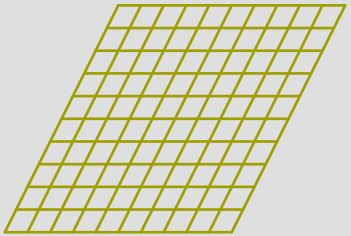
For those who love dirty tricks and clever macros, close reading of the code in `format.mp` may be worthwhile. The format macros in there use TEX to typeset the core components of a number, and use the dimensions of those components to compose combinations of signs, numbers and superscripts.

In METAFUN we have the module `mp-form.mp` which contains most of the code in `format.mp` but in a form that is a bit more suited for fine tuning. This permits us to use either the composition method, or to fall back on the `textext` method that is part of METAFUN. That way we can also handle fonts that have digits with different dimensions. Another 'change' concerns the pattern separator. Instead of a `%` we use `@`; you can choose to set another separator, but for embedded definitions `%` is definitely a bad choice because TEX sees it as a comment and ignores everything following it.

```
drawoptions(withpen pencircle scaled 1pt withcolor .625yellow) ;

draw hlingrid(0, 10, 1, 3cm, 3cm) ;
draw vlingrid(0, 10, 1, 3cm, 3cm) ;

draw hlingrid(0, 10, 1, 3cm, 3cm) shifted ( 3.5cm,0) ;
draw vloggrid(0, 10, 1, 3cm, 3cm) shifted ( 3.5cm,0) ;

draw hloggrid(0, 10, 1, 3cm, 3cm) shifted ( 7.0cm,0) ;
draw vlingrid(0, 10, 1, 3cm, 3cm) shifted ( 7.0cm,0) ;

draw hloggrid(0, 10, 1, 3cm, 3cm) shifted (10.5cm,0) ;
draw vloggrid(0, 10, 1, 3cm, 3cm) shifted (10.5cm,0) ;
```



```
drawoptions(withpen pencircle scaled 1pt withcolor .625yellow) ;

draw hlingrid(0, 10, 1, 3cm, 3cm) slanted .5 ;
draw vlingrid(0, 10, 1, 3cm, 3cm) slanted .5 ;
```



Using macros like these often involves a bit of trial and error. The arguments to these macros are as follows:

```
hlingrid (Min, Max, Step, Length, Width)
vlingrid (Min, Max, Step, Length, Height)
hloggrid (Min, Max, Step, Length, Width)
vloggrid (Min, Max, Step, Length, Height)
```

The macros take the following text upto the semi–colon into account and return a picture. We will now apply this knowledge to a more meaningful example. First we draw a grid.

You can use the grid drawing macros to produce your own paper, for instance using the following mixed TEX–METAFUN code:

```
\startMPpage
  StartPage ;
    width  := PaperWidth  ; height := PaperHeight ; unit := cm ;
    drawoptions(withpen pencircle scaled .2pt withcolor .8white) ;
    draw vlingrid(0, width /unit, 1/10, width,  height) ;
    draw hlingrid(0, height/unit, 1/10, height, width ) ;
    drawoptions(withpen pencircle scaled .5pt withcolor .4white) ;
    draw vlingrid(0, width /unit, 1,    width,  height) ;
    draw hlingrid(0, height/unit, 1,    height, width ) ;
  StopPage ;
\stopMPpage
```

This produces a page (as in **figure 9.1**) with a metric grid. If you're hooked to the inch, you can set `unit := 1in`. If you want to process this code, you need to wrap it into the normal document producing commands:

```
\setupcolors[state=start]

\starttext
   ... definitions ...
\stoptext
```

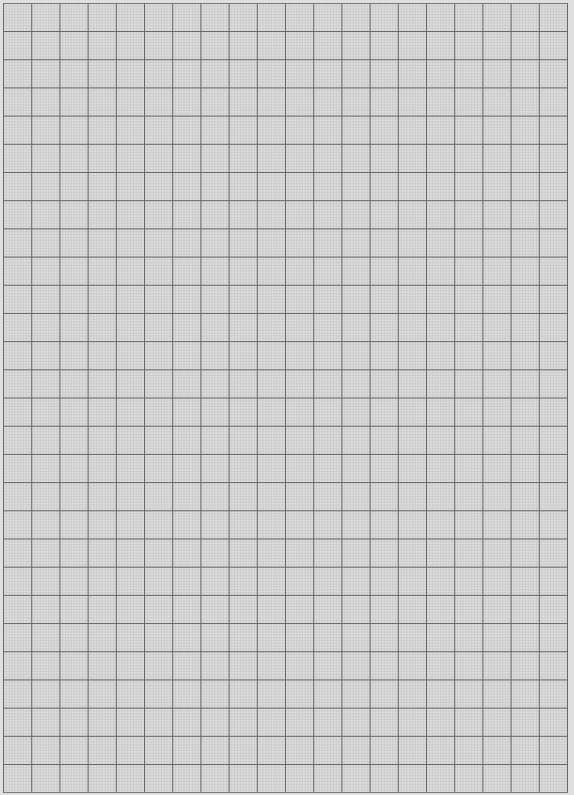


Figure 9.1 Quick and dirty grid paper.

9.3 Drawing functions

Today there are powerful tools to draw functions on grids, but for simple functions you can comfortably use METAPOST. Let's first draw a simple log–linear grid.

```
drawoptions(withpen pencircle scaled .25pt withcolor .5white) ;

draw hlingrid    (0, 20, .2, 20cm, 10cm) ;
draw vloggrid    (0, 10, .5, 10cm, 20cm) ;

drawoptions(withpen pencircle scaled .50pt) ;

draw hlingrid    (0, 20,  1, 20cm, 10cm) ;
draw vloggrid    (0, 10,  1, 10cm, 20cm) ;
```

To this grid we add some labels:

```
fmt_pictures  := false ;    % use TeX as formatting engine
textextoffset := ExHeight ; % a variable set by ConTeXt

draw hlintext.lft(0, 20,  5, 20cm, "@3e") ;
draw vlogtext.bot(0, 10,  9, 10cm, "@3e") ;
```

The arguments to the text placement macros are similar to the ones for drawing the axes. Here we provide a format string.

```
hlintext (Min, Max, Step, Length, Format)
vlintext (Min, Max, Step, Length, Format)
hlogtext (Min, Max, Step, Length, Format)
```

```
vlogtext (Min, Max, Step, Length, Format)
```

When drawing a smooth function related curve, you need to provide enough sample points. The `function` macro will generate them for you, but you need to make sure that for instance the maximum and minimum values are part of the generated series of points. Also, a smooth curve is not always the right curve. Therefore we provide three drawing modes:

method	**result**
1	a punked curve, drawn using --
2	a smooth curve, drawn using . .
3	a tight curve, drawn using . . .

If method 2 or 3 do not give the desired outcome, you can try a smaller step combined with method 1.

```
draw
  function(1,"log(x)","x",1,10,1) xyscaled (10cm,2cm)
  withpen pencircle scaled 5mm withcolor transparent(1,.5,yellow) ;

draw
  function(2,".5log(x)","x",1,10,1) xyscaled (10cm,2cm)
  withpen pencircle scaled 5mm withcolor transparent(1,.5,blue) ;
```

The first argument to the `function` macro specifies the drawing method. The last three arguments are the start value, end value and step. The second and third argument specify the function to be drawn. In this case the pairs `(x,x)` and `(.5log(x),x)` are calculated.

```
textextoffset := ExHeight ;
```

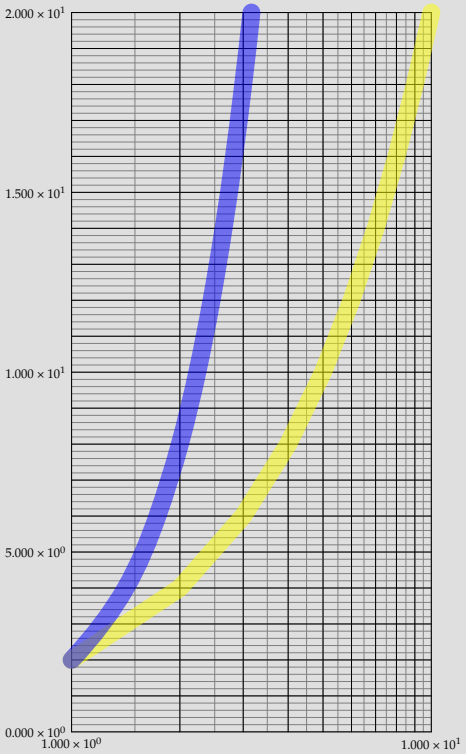


Figure 9.2 An example of a graphic with labels along the axes.

```
drawoptions(withpen pencircle scaled .50pt) ;

draw hlingrid(-10, 10, 1, 10cm, 10cm) ;
draw vlingrid(  0, 20, 1, 10cm, 20cm) shifted (0,-10cm) ;

drawoptions() ;

draw
  function(2,"x","sind(x)",0,360,10) xyscaled (1cm/36,10cm)
  withpen pencircle scaled 5mm withcolor transparent(1,.5,blue) ;

draw
  function(2,"x","sin(x*pi)",0,epsed(2),.1) xyscaled (10cm/2,5cm)
  withpen pencircle scaled 5mm withcolor transparent(1,.5,yellow) ;

draw
  function(2,"x","cosd(x)",0,360,10) xyscaled (1cm/36,10cm)
  withpen pencircle scaled 5mm withcolor transparent(1,.5,red) ;

draw
  function(2,"x","cos(x*pi)",0,epsed(2),.1) xyscaled (10cm/2,5cm)
  withpen pencircle scaled 5mm withcolor transparent(1,.5,green) ;
```

In this example we draw sinus and cosine functions using degrees and radians. In the case of radians the end points are not calculated due to rounding errors. In such case you can use the epsed value, which gives slightly more playroom.

```
draw function (1, "x", "sin(2x)"    , 1, 10, .01) scaled 1.5cm
  withpen pencircle scaled 1mm withcolor transparent(1,.5,red) ;
draw function (1, "x", "sin(2x*x)"  , 1, 10, .01) scaled 1.5cm
```

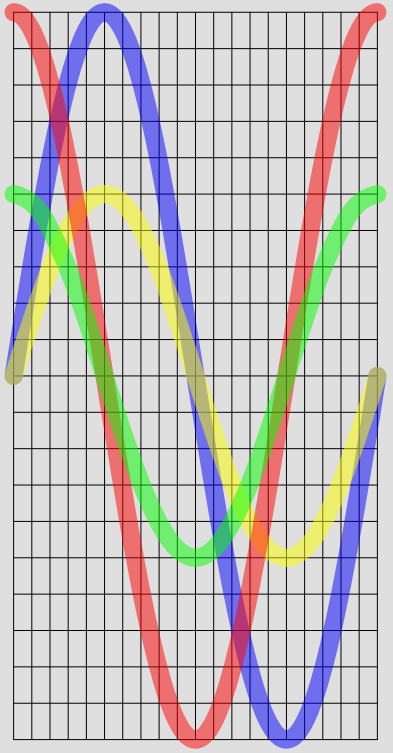
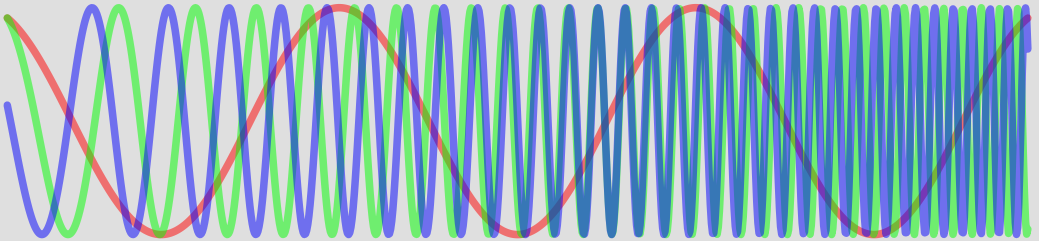


Figure 9.3 By using transparent colors, we don't have to calculate and mark the common points: they already stand out.

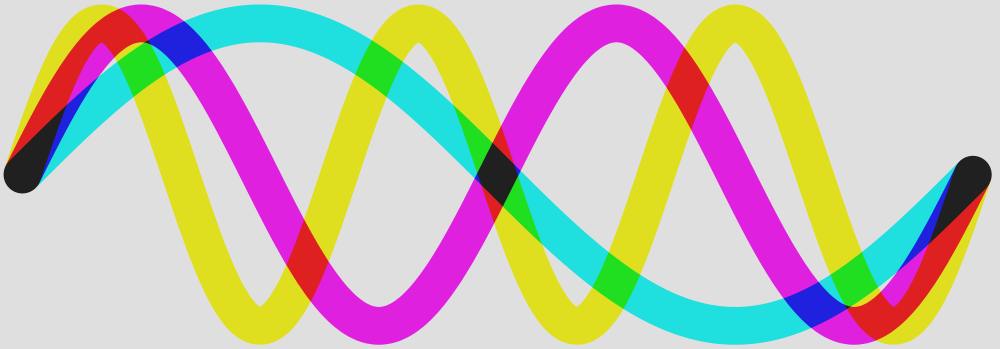
```
  withpen pencircle scaled 1mm withcolor transparent(1,.5,green) ;
draw function (1, "x", "sin(2x*x+x)", 1, 10, .01) scaled 1.5cm
  withpen pencircle scaled 1mm withcolor transparent(1,.5,blue) ;
```

Of course you can do without a grid. The next example demonstrates a nice application of transparencies.



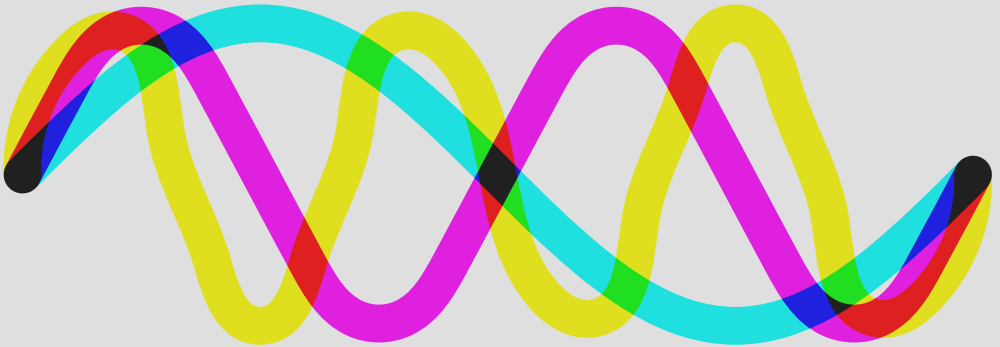
If we use the `exclusion` method for the transparencies, combined with no transparency, we get the following alternative.

```
draw function (2, "x", "sin(x)" , 0, 2pi, pi/40) scaled 2cm
  withpen pencircle scaled 5mm withcolor transparent("exclusion",1,red) ;
draw function (2, "x", "sin(2x)", 0, 2pi, pi/40) scaled 2cm
  withpen pencircle scaled 5mm withcolor transparent("exclusion",1,green) ;
draw function (2, "x", "sin(3x)", 0, 2pi, pi/40) scaled 2cm
  withpen pencircle scaled 5mm withcolor transparent("exclusion",1,blue) ;
```



The next alternative uses a larger step, and as a result (in drawing mode 2) gives worse results. (Without the epsed it would have looked even worse in the end points.

```
draw function (2, "x", "sin(x)" , 0, epsed(2pi), pi/10) scaled 2cm
  withpen pencircle scaled 5mm withcolor transparent("exclusion",1,red) ;
draw function (2, "x", "sin(2x)", 0, epsed(2pi), pi/10) scaled 2cm
  withpen pencircle scaled 5mm withcolor transparent("exclusion",1,green) ;
draw function (2, "x", "sin(3x)", 0, epsed(2pi), pi/10) scaled 2cm
  withpen pencircle scaled 5mm withcolor transparent("exclusion",1,blue) ;
```



There are enough applications out there to draw nice functions, like gnuplot for which Mojca Miklavec made a backend that works well with CONTEXT. Nevertheless it can be illustrative to explore the possibilities of the CONTEXT, LUATEX, METAPOST combination using functions.

First of all you can use LUA to make paths and this is used in some of the debugging and tracing options that come with CONTEXT. For instance, if you process a document with

```
context --timing yourdoc.tex
```

then you can afterwards process a file that is generated while processing this document:

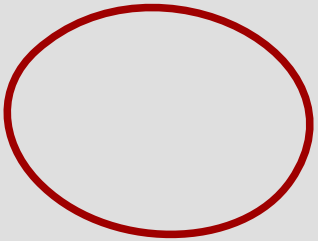
```
context --extras timing yourdoc
```

This will give you a document with graphics that show where LUATEX spent its time on. Of course these graphics are generated with METAPOST.

There are a few helpers built in (and more might follow). For example:

```
draw
  \ctxlua{metapost.metafun.topath({ {x=1,y=1}, {x=1,y=3}, {4,1}, "cycle" })}
  xysized(4cm,3cm)
  withpen pencircle scaled 1mm
  withcolor .625 red ;
```

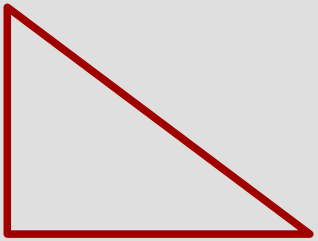
The `topath` function takes a table of points or strings.



You can pass a connector so

```
draw
  \ctxlua{metapost.metafun.topath({ {x=1,y=1}, {x=1,y=3}, {4,1}, "cycle" }, "--")}
  xysized(4cm,3cm)
  withpen pencircle scaled 1mm
  withcolor .625 red ;
```

gives:



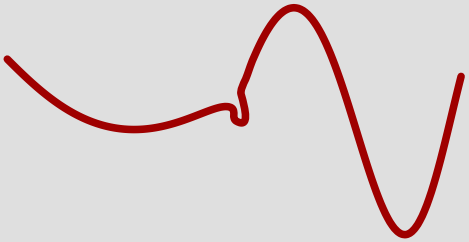
Writing such LUA functions is no big deal. For instance we have available:

```
function metapost.metafun.interpolate(f,b,e,s,c)
    tex.write("(")
    for i=b,e,(e-b)/s do
        local d = loadstring(string.formatters["return function(x) return %s end"](f))
        if d then
            d = d()
            if i > b then
                tex.write(c or "...")
            end
            tex.write(string.formatters["(%F,%F)"](i,d(i)))
        end
    end
    tex.write(")")
end
```

An example of usage is:

```
draw
  \ctxlua{metapost.metafun.interpolate(
    "math.sin(x^2+2*x+math.sqrt(math.abs(x)))",
    -math.pi/2,math.pi/2,100
  )}
  xysized(6cm,3cm)
  withpen pencircle scaled 1mm
  withcolor .625 red ;
```

And indeed we get some drawing:



Let's see what happens when we use less accuracy and a different connector:

```
draw
  \ctxlua{metapost.metafun.interpolate(
    "math.sin(x^2+2*x+math.sqrt(math.abs(x)))",
    -math.pi/2,math.pi/2,10,"--"
  )}
  xysized(6cm,3cm)
```

```
withpen pencircle scaled 1mm
withcolor .625 red ;
```

Now we get:

Of course we could extend this LUA function with all kind of options and we might even do that when we need it.

10 Typesetting in METAPOST

It is said that a picture tells more than a thousand words. So you might expect that text in graphics becomes superfluous. Out of experience we can tell you that this is not the case. In this chapter we explore the ways to add text to METAPOST graphics, and let you choose whether or not to have it typeset by TEX.

10.1 The process

You can let METAPOST process text that is typeset by TEX. Such text is first embedded in the METAPOST file in the following way:

```
btex Some text to be typeset by \TEX etex
```

This returns a picture, but only after METAPOST has made sure that TEX has converted it into something useful. This process of conversion is slightly system dependent and even a bit obscure. Traditional METAPOST calls a program that filters the `btex–etex` commands, next it calls TEX by passing the output routine, in order to make sure that each piece of text ends up on its own page, and afterwards it again calls a program that converts the DVI pages into METAPOST pictures.

In CONTEXT MKII, when using WEB2C, you can generate the graphics at run–time. This takes more time than processing the graphics afterwards, but has the advantage that TEX knows immediately what graphic it is dealing with. When enabled, CONTEXT will call either METAPOST, or, when the graphic contains `btex–etex` commands, call TEXEXEC, which in turn makes sure that the right auxiliary programs are executed.

In CONTEXT MKIV you won't notice this at all as there everything is tightly integrated with LUATEX's MPLIB.

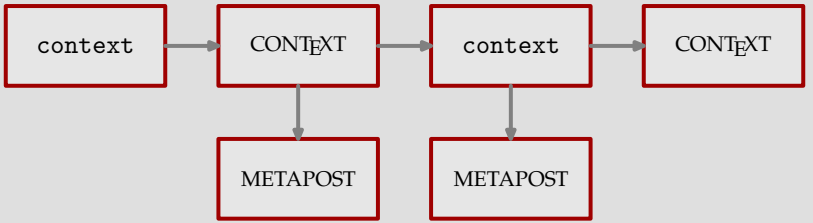


Figure 10.1 How TEX and METAPOST work together.

10.2 Environments

In case you want to pass code that is shared by all `btex–etex` pictures, METAPOST provides:

```
verbatimtex \DefineSomeCommands etex ;
```

However, in CONTEXT one has a better mechanism available. In CONTEXT MKII the advised method is passing environments. The best way to pass them is the following. As an example we switch to the 15 basic POSTSCRIPT fonts.

```
\startMPenvironment
  \usetypescript[palatino][texnansi] % mkii has encodings
  \setupbodyfont[palatino]
\stopMPenvironment
```

This means that in code like the following, a Palatino font will be used.

```
\startMPcode
draw btex Meta is a female lion! etex
  xysized (\the\textwidth,\the\textheight) ;
\stopMPcode
```

However, in CONTEXT MKIV this method is no longer recomended as all processing happens in the same run anyway.

```
\startMPcode
numeric w, h ; w := \the\textwidth ; h := w/2 ;

picture p ; p := btex \colored[r=.375,g=.375]{Meta is a female lion!} etex xysized (w,h) ;
picture q ; q := btex \colored[r=.625]       {Meta is a female lion!} etex xysized (w,h) ;

path b ; b := boundingbox p ; draw p ;

for i=(.28w,.90h),(.85w,.90h),(w,.05h) :
  picture r ; r := q ;
  path s ; s := (fullsquare xscaled .05w yscaled .4h) shifted i ;
  clip r to s ; draw r ; % draw s ;
endfor ;

setbounds currentpicture to b ;
\stopMPcode
```

Figure 10.2 shows the previous sentence in a slightly different look. You may consider coloring the dots to be an exercise in clipping.

Figure 10.2 An example of clipping.

An environment can be reset with `\resetMPenvironment` or by passing `reset` as key to `\startMPenvironment`.

```
\startMPenvironment[reset]
  \usetypescript[postscript][texnansi] % mkii
  \setupbodyfont[postscript]
```

```
\stopMPenvironment
```

So, to summarize: if you're using CONTEXT MKIV you might as well forgot what you just read.

10.3 Labels

In METAPOST you can use the `label` macro to position text at certain points.

```
label("x", origin) ;
```

The font and scale are determined by two variables, `defaultfont` and `defaultscale`, the former expecting the name of a font in the form of a string, the latter expecting a numeric to be used in the scaling of the font. Should you choose not to set these yourself, they default to `"cmr10"` and `1.0`, respectively. However, you can change the defaults as follows:

```
defaultfont  := "tir" ;
defaultscale := 1.2 ;
```

These settings select Adobe Times at about 12pt. You can also set these variables to CONTEXT related values. For CONTEXT graphics they are set to:

```
defaultfont  := "\truefontname{Regular}" ;
defaultscale := \the\bodyfontsize/10 ;
```

This means that they will adapt themselves to the current body font (in this document file:texgyrepagella-regular) and the current size of the bodyfont (here 10.0pt/10).

10.4 TEX text

In the next example we will use a special mechanism for building graphics step by step. The advantage of this method is that we can do intermediate calculations in TEX. Our objective is to write a macro that draws text along a circular path. While doing so we want to achieve the following:

- the text should be properly kerned, i.e. the spacing between characters should be optimal,
- the position on the circle should vary, and
- the radius of the circle should vary.

This implementation is not the most straightforward one, but by doing it step by step, at least we see what is involved. Later, we will see a better method. If you run these examples yourself, you must make sure that the TEX environment of your document matches the one used by METAPOST.

We let the bodyfont match the font used in this document, and define `RotFont` to be the regular typeface, the one you are reading right now, but bold.

```
\definefont[RotFont][RegularBold*default]
```

Since METAPOST is unaware of kerning, we have to use TEX to keep track of the positions. We will split the text into tokens (often characters) and store the result in an array of pictures (`pic`). We will also store the accumulated width in an array (`len`). The number of characters is stored in `n`. In a few paragraphs we will see why the other arrays are needed.

While defining the graphic, we need TEX to do some calculations. Therefore, we will use `\startMPdrawing` to stepwise construct the definition. The basic pattern we will follow is:

```
\resetMPdrawing
\startMPdrawing
```

```
    metapost code
  \stopMPdrawing
  tex code
  \startMPdrawing
    metapost code
  \stopMPdrawing
  \MPdrawingdonetrue
  \getMPdrawing
```

In the process, we will use a few variables. We will store the individual characters of the text in the variable `pic`, its width in `wid` and the length of the string so far in `len`. Later we will use the `pos` array to store the position where a character ends up. The variable `n` holds the number of tokens.

```
\resetMPdrawing
\startMPdrawing
  picture pic[] ;
  numeric wid[], len[], pos[], n ;
  wid[0] := len[0] := pos[0] := n := 0 ;
\stopMPdrawing
```

We also started fresh by resetting the drawing. From now on, each start command will add some more to this graphic. The next macro is responsible for collecting the data. Each element is passed on to TEX, using the `btex` construct. So, METAPOST itself will call TEX!

```
\def\whatever#1%
  {\appendtoks#1\to\MPtoks
   \setbox\MPbox=\hbox{\bfd\the\MPtoks}%
```

```
    \startMPdrawing
      n := n + 1 ; len[n] := \the\wd\MPbox ;
    \stopMPdrawing
    \startMPdrawing[-]
      pic[n] := textext("\bfd\setstrut\strut#1") ;
      pic[n] := pic[n] shifted - llcorner pic[n] ;
    \stopMPdrawing}

\handletokens MetaPost is Fun!\with\whatever
```

We use the low level CONTEXT macro `\appendtoks` to extend the token list `\MPtoks`. The `\handletokens` macro passes each token (character) of `MetaPost is Fun!` to the macro `\whatever`. The tokens are appended to the token register `\MPtoks` (already defined). Then we typeset the content of `\MPtoks` in `\MPbox` (also already defined). The width of the box is passed to METAPOST and stored in `len`.

By default the content of the drawing is expanded, which means that the macro is replaced by its current meaning, so the current width ends up in the METAPOST file. The next part of the drawing, starting with `btex`, puts the token in a picture. This time we don't expand the drawing, since we want to pass font information. Here, the `[-]` suppresses expansion of `btex \bfd #1 etex`. The process is iterated by `\handletokens` for each character of the text `MetaPost is Fun!`.

Before we typeset the text, now available in pieces in `pic`, in a circle, we will first demonstrate what they look like. You may like to take a look at the file `mpgraph.mp` to see what is passed to METAPOST.

```
\startMPdrawing
  pair len ; len := origin ;
  for i=1 upto n :
    draw pic[i] shifted len ;
```

```
    draw boundingbox pic[i] shifted len
      withpen pencircle scaled .25pt withcolor red ;
    len := len+(xpart urcorner pic[i]-xpart llcorner pic[i],0) ;
  endfor ;
\stopMPdrawing
```

We can call up this drawing with `\getMPdrawing`, but first we inform the compiler that our METAPOST drawing is completed.

```
\MPdrawingdonetrue\getMPdrawing
```

This results in:

Compare this text with the text as typeset by TEX:

MetaPost is Fun!

and you will see that the text produced by METAPOST is not properly kerned. When putting characters after each other, TEX uses the information available in the font, to optimize the spacing between characters, while METAPOST looks at characters as separate entities. But, since we have stored the optimal spacing in `len`, we can let METAPOST do a better job. Let's first calculate the correction needed.

```
\startMPdrawing
  for i=1 upto n :
    wid[i] := abs(xpart urcorner pic[i] - xpart llcorner pic[i]) ;
```

```
    pos[i] := len[i]-wid[i] ;
  endfor ;
\stopMPdrawing
```

This compares well to the text as typeset by TEX:

MetaPost is Fun!

We can now use the values in pos to position the pictures according to what TEX considered to be the best (relative) position.

```
\startMPdrawing
  for i=1 upto n :
    draw pic[i] shifted (pos[i],0) ;
    draw boundingbox pic[i] shifted (pos[i],0)
      withpen pencircle scaled .25pt withcolor red ;
  endfor ;
\stopMPdrawing
```

That this correction is adequate, is demonstrated in the next graphic. If you look closely, you will see that for instance the 'o' is moved to the left, under the capital 'P'.

When we want to position the pictures along a circle, we need to apply some rotations, especially because we want to go clockwise. Since we don't want to use 'complicated' math or more advanced METAPOST code yet, we will do it in steps.

```
\startMPdrawing
  for i=1 upto n:
    pic[i] := pic[i] rotatedaround(origin,-270) ;
  endfor ;
\stopMPdrawing
```

We will now center the pictures around the baseline. Centering comes down to shifting over half the height of the picture. This can be expressed by:

```
ypart -.5[ulcorner pic[i],llcorner pic[i]]
```

but different ways of calculating the distance are possible too.

```
\startMPdrawing
  for i=1 upto n :
    pic[i] := pic[i]
      shifted (0,ypart -.5[ulcorner pic[i],llcorner pic[i]]) ;
  endfor ;
\stopMPdrawing
```

So, now we have:

When we typeset on a (half) circle, we should map the actual length onto a partial circle. We denote the radius with an r and shift the pictures to the left.

```
\startMPdrawing
  numeric r ; r := len[n]/pi ;
  for i=1 upto n :
    pic[i] := pic[i] shifted (-r,0) ;
  endfor ;
\stopMPdrawing
```

You can now use the following code to test the current state of the pictures. Of course this code should not end up in the final definitions.

```
\startMPdrawing
  draw origin
    withpen pencircle scaled 5pt withcolor red ;
  for i=1 upto n :
    draw pic[i] ;
    draw boundingbox pic[i]
      withpen pencircle scaled .25pt withcolor red ;
  endfor ;
\stopMPdrawing
```

Later we will write a compact, efficient macro to take care of rotation. However, for the moment, so as not to overwhelm you with complicated code, we will rotate each individual picture with the following code fragment.

```
\startMPdrawing
  numeric delta, extra, radius, rot[] ;

  delta  := extra := radius := 0 ;

  for i=1 upto n :
    rot[i] := extra+delta-((pos[i]+.5wid[i])/len[n])*(180+2delta) ;
  endfor ;
\stopMPdrawing
```

Here we introduce a few variables that we can use later to tune the result a bit. With `delta`, the space between the characters can be increased, while `extra` rotates the whole string around the origin. The `radius` variable can be used to increase the distance to the origin. Without these variables, the assignment would have been:

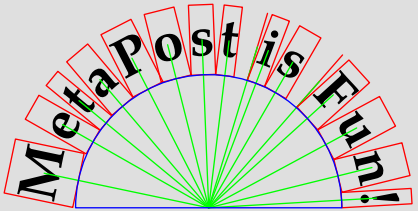
```
rot[i] := ((pos[i]+.5wid[i])/len[n])*180 ;
```

Placing the pictures is now rather easy:

```
\startMPdrawing
  for i=1 upto n :
    draw pic[i] shifted (-radius,0) rotatedaround(origin,rot[i]) ;
  endfor ;
\stopMPdrawing
```

The pictures are now positioned on half a circle, properly kerned.

A bit more insight is given in the next picture:



This was defined as follows. The path variable `tcycle` is predefined to the top half of a fullcircle.

```
\startMPdrawing
  def moved(expr i) =
    shifted (-radius,0) rotatedaround(origin,rot[i])
  enddef ;
  pickup pencircle scaled .5pt ;
  for i=1 upto n :
    draw pic[i] moved(i) ;
    draw boundingbox pic[i] moved(i) withcolor red ;
    draw origin -- center pic[i] moved(i) withcolor green ;
```

```
    endfor ;
    draw tcircle scaled 2r withcolor blue ;
  \stopMPdrawing
```

We will now package all of this into a nice, efficient macro, using, of course, the predefined scratch registers \MPtoks and \MPbox. First we define the token processor. Note again the expansion inhibition switch [-].

```
\def\processrotationtoken#1%
  {\appendtoks#1\to\MPtoks
   \setbox\MPbox=\hbox{\RotFont\the\MPtoks}%
   \startMPdrawing
     n := n + 1 ; len[n] := \the\wd\MPbox ;
   \stopMPdrawing
   \startMPdrawing[-]
     pic[n] := textext("\RotFont\setstrut\strut#1") ;
     pic[n] := pic[n] shifted - llcorner pic[n] ;
   \stopMPdrawing}
```

The main macro is a bit more complicated but by using a few scratch numerics, we can keep it readable.

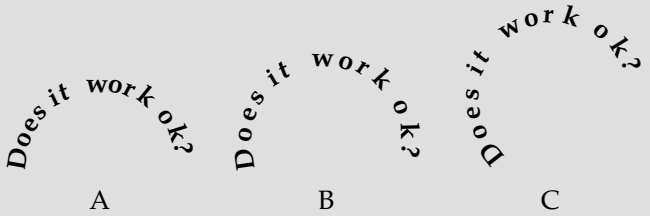
```
\def\rotatetokens#1#2#3#4% delta extra radius tokens
  {\vbox\bgroup
   \MPtoks\emptytoks
   \resetMPdrawing
   \startMPdrawing
     picture pic[] ;
     numeric wid, len[], rot ;
```

```
    numeric delta, extra, radius, n, r ;
    len[0] := n := 0 ;
    delta  := #1 ; extra := #2 ; radius := #3 ;
  \stopMPdrawing
  \handletokens#4\with\processrotationtoken
  \startMPdrawing
    r := len[n]/pi ;
    for i=1 upto n :
      wid := abs(xpart lrcorner pic[i] -
                 xpart llcorner pic[i]) ;
      rot := extra + delta -
             ((len[i]-.5wid)/len[n]) * (180+2delta) ;
      draw pic[i]
        rotatedaround (origin,-270) shifted (-r-radius,
        ypart -.5[ulcorner pic[i], llcorner pic[i]])
        rotatedaround (origin,rot) ;
    endfor ;
  \stopMPdrawing
  \MPdrawingdonetrue
  \getMPdrawing
  \resetMPdrawing
  \egroup}
```

We can use this macro as follows:

```
\startcombination[3*1]
  {\rotatetokens {0} {0}{0}{Does it work ok?}} {A}
```

```
  {\rotatetokens{20} {0}{0}{Does it work ok?}} {B}
  {\rotatetokens{20}{30}{0}{Does it work ok?}} {C}
\stopcombination
```



The previous macro is not really an example of generalization, but we used it for demonstrating how to build graphics in a stepwise way. If you put the steps in buffers, you can even combine steps and replace them at will. This is how we made the previous step by step examples: We put each sub–graphic in a buffer and then called the ones we wanted.

We now present a more general approach to typesetting along a given path. This method is not only more robust and general, it is also a more compact definition, especially if we omit the tracing and testing code. We use a familiar auxiliary definition. The `\setstrut` and `\strut` commands ensure that the lines have the proper depth and height.

```
\def\processfollowingtoken#1%
  {\appendtoks#1\to\MPtoks
   \setbox\MPbox=\hbox{\RotFont\setstrut\strut\the\MPtoks}%
   \startMPdrawing
     n := n + 1 ; len[n] := \the\wd\MPbox ;
   \stopMPdrawing
```

```
\startMPdrawing[-]
  pic[n] := btex \RotFont\setstrut\strut#1 etex ;
  pic[n] := pic[n] shifted -llcorner pic[n] ;
\stopMPdrawing}
```

The definition of `\followtokens` is as follows. Keep in mind that `\RotFont` is defined in the METAPOST environment. You may notice that we have added a directive to include the METAPOST graphic called `followtokens`. Storing the path in a graphic container instead of using `\startMPdrawing` is less sensitive for interference with other drawing processes.

This definition is not the right one!

```
\def\dofollowtokens#1#2%
  {\vbox\bgroup
   \forgetall
   \dontcomplain
   \doifundefined{RotFont}{\definefont[RotFont][RegularBold*default]}%
   \MPtoks\emptytoks
   \resetMPdrawing
   \startMPdrawing
     \includeMPgraphic{followtokens} ;
     picture pic[] ; numeric len[], n ; n := 0 ;
   \stopMPdrawing
   \handletokens#2\with\processfollowingtoken
   \startMPdrawing
     if unknown RotPath   : path    RotPath   ; RotPath   := origin ; fi ;
     if unknown RotColor : color   RotColor ; RotColor := black  ; fi ;
```

```
if unknown ExtraRot : numeric ExtraRot ; ExtraRot := 0      ; fi ;
numeric al, at, pl, pc, wid, pos ; pair ap, ad ;
al := arclength RotPath ;
if al=0 :
  al := len[n] + ExtraRot ;
  RotPath := origin -- (al,0) ;
fi ;
if al<len[n]:
  RotPath := RotPath scaled ((len[n]+ExtraRot)/al) ;
  al := arclength RotPath ;
fi ;
if \number#1 = 1 :
  pl := (al-len[n])/(if n>1 : (n-1) else : 1 fi) ;
  pc := 0 ;
else : % centered / MP
  pl := 0 ;
  pc := arclength RotPath/2 - len[n]/2 ;
fi ;
for i=1 upto n :
  wid := abs(xpart urcorner pic[i] - xpart llcorner pic[i]) ;
  pos := len[i]-wid/2 + (i-1)*pl + pc ;
  at := arctime   pos of RotPath ;
  ap := point     at  of RotPath ;
  ad := direction at  of RotPath ;
  draw pic[i] shifted (-wid/2,0) rotated(angle(ad)) shifted ap
    withcolor RotColor ;
```

```
    endfor ;
  \stopMPdrawing
  \MPdrawingdonetrue
  \getMPdrawing
  \resetMPdrawing
  \egroup}
```

So, how does this compare to earlier results? The original, full text as typeset by TEX, looks like:

We now follow some arbitrary path ...

In the examples, the text is typeset along the path with:

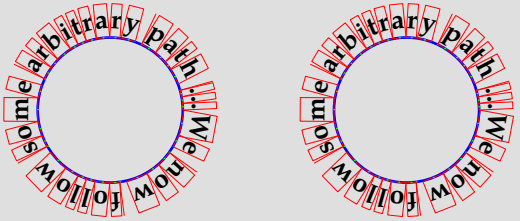
```
\followtokens{We now follow some arbitrary path ...}
```

We now follow some arbitrary path ...

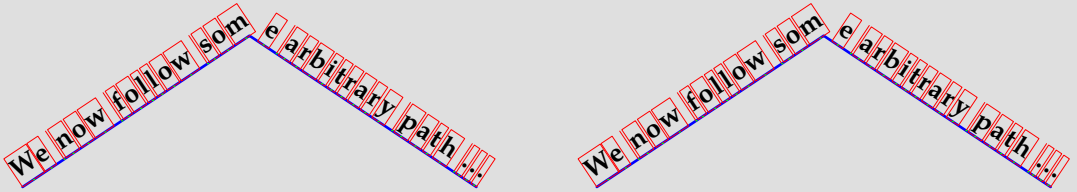
Since we did not set a path, a dummy path is used. We can provide a path by (re)defining the graphic `followtokens`.

```
\startuseMPgraphic{followtokens}
  path RotPath ; RotPath := fullcircle ;
\stopuseMPgraphic
```

```
\startuseMPgraphic{followtokens}
  path RotPath ; RotPath := reverse fullcircle ;
\stopuseMPgraphic
```

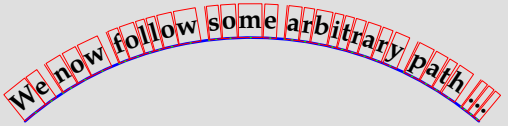


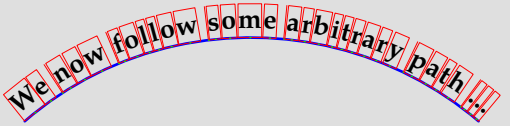

```
\startuseMPgraphic{followtokens}
  path RotPath ; RotPath := (-3cm,-1cm)--(0,1cm)--(3cm,-1cm) ;
\stopuseMPgraphic
```



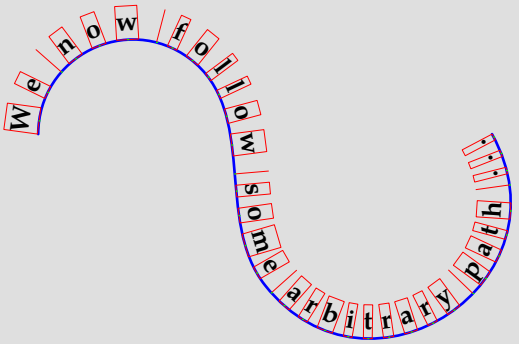

```
\startuseMPgraphic{followtokens}
  path RotPath ; RotPath := (-3cm,0)--(3cm,1cm) ;
\stopuseMPgraphic
```

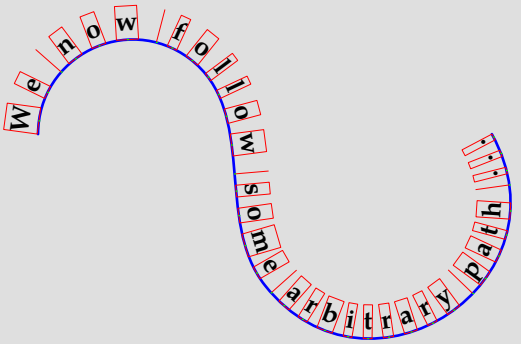

```
\startuseMPgraphic{followtokens}
  path RotPath ; RotPath := (-3cm,0)..(-1cm,1cm)..(3cm,0) ;
\stopuseMPgraphic
```






```
\startuseMPgraphic{followtokens}
  path RotPath ; RotPath := (-3cm,0)..(-1cm,1cm)..(0cm,-2cm)..(3cm,0) ;
\stopuseMPgraphic
```





When turned on, tracing will produce bounding boxes as well as draw the path. Tracing can be turned on by saying:

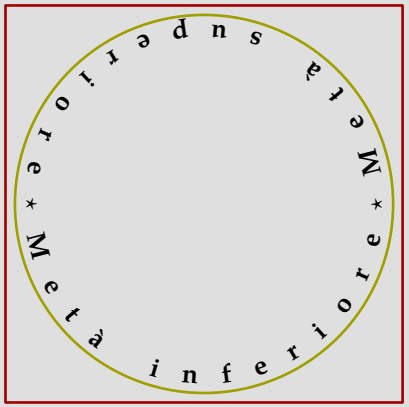
```
\startMPinclusions
  boolean TraceRot ; TraceRot := true ;
\stopMPinclusions
```

The next example is dedicated to Giuseppe Bilotta who wants to handle multiple strings and uses a patched version of `\followtokens`. To avoid a complicated explanation, we will present an alternative here that uses overlays. This method also avoids complicated path definitions.

```
\startoverlay
  {\startuseMPgraphic{followtokens}
     draw fullcircle scaled 5cm .
       withpen pencircle scaled 1pt withcolor .625yellow ;
     draw fullsquare scaled 5.25cm
       withpen pencircle scaled 1pt withcolor .625red ;
     drawoptions (withcolor .625red) ;
     path RotPath ; RotPath := halfcircle scaled 5cm ;
     setbounds currentpicture to boundingbox fullcircle scaled 5.25cm ;
   \stopuseMPgraphic
   \followtokens { Met{\`a} superiore }}
  {\startuseMPgraphic{followtokens}
     drawoptions (withcolor .625red) ;
     path RotPath ; RotPath := halfcircle rotated 90 scaled 5cm ;
     setbounds currentpicture to boundingbox fullcircle scaled 5.25cm ;
   \stopuseMPgraphic
```

```
    \followtokens { {$\star$} }}
   {\startuseMPgraphic{followtokens}
      drawoptions (withcolor .625red) ;
      path RotPath ; RotPath := halfcircle rotated 180 scaled 5cm ;
      setbounds currentpicture to boundingbox fullcircle scaled 5.25cm ;
    \stopuseMPgraphic
    \followtokens { Met{\`a} inferiore }}
   {\startuseMPgraphic{followtokens}
      drawoptions (withcolor .625red) ;
      path RotPath ; RotPath := halfcircle rotated 270 scaled 5cm ;
      setbounds currentpicture to boundingbox fullcircle scaled 5.25cm ;
    \stopuseMPgraphic
    \followtokens { {$\star$} }}
\stopoverlay
```

In order to fool the overlay macro that each graphic has the same size, we force a bounding box.



10.5 Talking to TEX

Sometimes, others may say oftentimes, we are in need for some fancy typesetting. If we want to typeset a paragraph of text in a non standard shape, like a circle, we have to fall back on `\parshape`. Unfortunately, TEX is not that strong in providing the specifications of more complicated shapes, unless you are willing to do some complicated arithmetic TEX. Given that METAPOST knows how to deal with shapes, the question is: "Can METAPOST be of help?"

In the process of finding out how to deal with this, we first define a simple path. Because we are going to replace pieces of code, we will compose the graphic from components. First, we create the path.

```
\startuseMPgraphic{text path}
  path p ; p := ((0,1)..(-1,0)..(1,0)--cycle) scaled 65pt ;
```

```
\stopuseMPgraphic
```

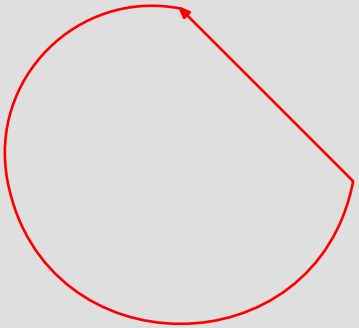
This shape is not that beautiful, but it has a few characteristics that will help us to identify bordercases.

```
\startuseMPgraphic{text draw}
  drawarrow p withpen pencircle scaled 1pt withcolor red ;
\stopuseMPgraphic
```

Now we use CONTEXT's `\includeMPgraphic` command to build our graphic from the previously defined components.

```
\startuseMPgraphic{text}
  \includeMPgraphic{text path}
  \includeMPgraphic{text draw}
\stopuseMPgraphic
```

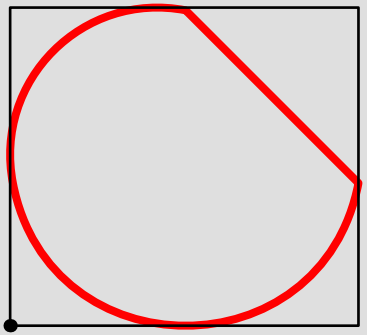
When called with `\useMPgraphic{text}`, we get:



For the moment we start the path at $(x = 0, y > 0)$, but later using more complicated macros, we will see that we can use arbitrary paths.

We are going to split the path in two, and will use the points that make up the bounding box as calcutated by METAPOST. The next graphic shows one of these points, the lower left corner, available as point `llcorner p`.

```
\startuseMPgraphic{text draw}
  draw             p withpen pencircle scaled 3pt withcolor red ;
  draw boundingbox p withpen pencircle scaled 1pt ;
  draw llcorner    p withpen pencircle scaled 5pt ;
\stopuseMPgraphic
```



The five points that METAPOST can report for each path or picture are:

`llcorner`	lower left corner
`lrcorner`	lower right corner
`urcorner`	upper right corner

`ulcorner`	upper left corner
`center`	intersection of the diagonals

If we want to typeset text inside this circle, we need to know where a line starts and ends. Given that lines are horizontal and straight, we therefore need to calculate the intersection points of the lines and the path. As a first step, we calculate the top and bottom of the path and after that we split off the left and right path.

```
\startuseMPgraphic{text split}
  pair t, b ; path l, r ;

  t := (ulcorner p -- urcorner p) intersectionpoint p ;
  b := (llcorner p -- lrcorner p) intersectionpoint p ;

  l := p cutbefore t ; l := l cutafter b ;
  r := p cutbefore b ; r := r cutafter t ;
\stopuseMPgraphic
```

The `intersectionpoint` macro returns the point where two paths cross. If the paths don't cross, an error is reported, when the paths cross more times, just one point is returned. The `cutafter` and `cutbefore` commands do as their names say and return a path.

In the `text split` code fragment, `t` and `b` are the top points of the main path, while `l` and `r` become the left and right half of path `p`.

We now draw the original path using a thick pen and both halves with a thinner pen on top of the original. The arrows show the direction.

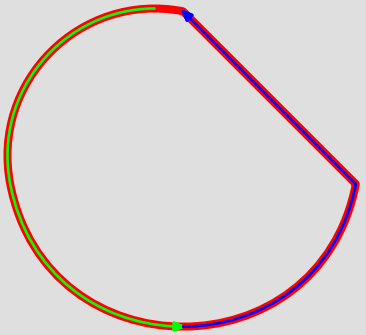
```
\startuseMPgraphic{text draw}
  draw      p withpen pencircle scaled 3pt withcolor red ;
```

```
    drawarrow l withpen pencircle scaled 1pt withcolor green ;
    drawarrow r withpen pencircle scaled 1pt withcolor blue ;
  \stopuseMPgraphic
```

We use \includeMPgraphic to assemble the components:

```
  \startuseMPgraphic{text}
    \includeMPgraphic{text path}
    \includeMPgraphic{text split}
    \includeMPgraphic{text draw}
  \stopuseMPgraphic
```

This graphic is typeset with \useMPgraphic{text}:



Before we are going to use them, we define some variables that specify the text. We use a baseline distance of 8 points. The part of the line above the baseline is 7.2 points, while the (maximum) depth is 2.8 points. These

ratios are the ones we use in CONTEXT. Because we don't want the text to touch the circle so we define an offset too also.

```
\startuseMPgraphic{text vars}
  baselineskip := 8pt ;
  strutheight  := (7.2/10) * baselineskip ;
  strutdepth   := (2.8/10) * baselineskip ;
  offset       := baselineskip/2 ;
  topskip      := strutheight ;
\stopuseMPgraphic
```

We more of less achieve the offset by scaling the path. In doing so, we use the width and height, which we call `hsize` and `vsize`, thereby conforming to the TEX naming scheme.

First we calculate both dimensions from the bounding box of the path. Next we down scale the path to compensate for the offset. When done, we recalculate the dimensions.

```
\startuseMPgraphic{text move}
  pair t, b ; path q, l, r ;

  hsize := xpart lrcorner p - xpart llcorner p ;
  vsize := ypart urcorner p - ypart lrcorner p ;

  q := p xscaled ((hsize-2offset)/hsize)
         yscaled ((vsize-2offset)/vsize) ;

  hsize := xpart lrcorner q - xpart llcorner q ;
  vsize := ypart urcorner q - ypart lrcorner q ;
\stopuseMPgraphic
```

We adapt the `text split` code to use the reduced path instead of the original.

```
\startuseMPgraphic{text split}
  t := (ulcorner q -- urcorner q) intersectionpoint q ;
  b := (llcorner q -- lrcorner q) intersectionpoint q ;

  l := q cutbefore t ; l := l cutafter b ;
  r := q cutbefore b ; r := r cutafter t ;
\stopuseMPgraphic
```

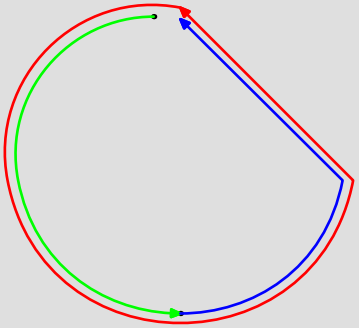
In order to test what we have reached so far, we draw the original path, the left and right part of the reduced path, and both the top and bottom point.

```
\startuseMPgraphic{text draw}
  drawarrow p withpen pencircle scaled 1pt withcolor red ;
  draw      t withpen pencircle scaled 2pt ;
  draw      b withpen pencircle scaled 2pt ;
  drawarrow l withpen pencircle scaled 1pt withcolor green ;
  drawarrow r withpen pencircle scaled 1pt withcolor blue ;
\stopuseMPgraphic
```

Again we use `\includeMPgraphic` to combine the components into a graphic.

```
\startuseMPgraphic{text}
  \includeMPgraphic{text path} \includeMPgraphic{text vars}
  \includeMPgraphic{text move} \includeMPgraphic{text split}
  \includeMPgraphic{text draw}
\stopuseMPgraphic
```

Then we use `\useMPgraphic{text}` to call up the picture.



The offset is not optimal. Note the funny gap at the top. We could try to fix this, but there is a better way to optimize both paths.

We lower the top edge of `q`'s bounding box by `topskip`, then cut any part of the left and right pieces of `q` that lie above it. Similarly, we raise the bottom edge and cut off the pieces that fall below this line.

```
\startuseMPgraphic{text cutoff}
  path tt, bb ;

  tt := (ulcorner q -- urcorner q) shifted (0,-topskip) ;
  bb := (llcorner q -- lrcorner q) shifted (0,strutdepth) ;

  l := l cutbefore (l intersectionpoint tt) ;
  l := l cutafter  (l intersectionpoint bb) ;
  r := r cutbefore (r intersectionpoint bb) ;
  r := r cutafter  (r intersectionpoint tt) ;
```

```
\stopuseMPgraphic
```

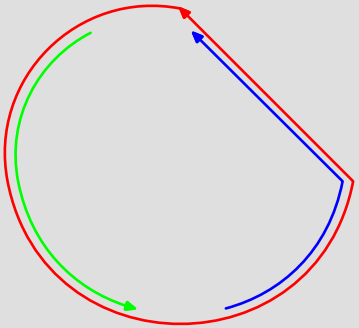
Because we use `\includeMPgraphic` to construct the graphic, we can redefine `text draw` to show the result of this effort.

```
\startuseMPgraphic{text draw}
  drawarrow p withpen pencircle scaled 1pt withcolor red ;
  drawarrow l withpen pencircle scaled 1pt withcolor green ;
  drawarrow r withpen pencircle scaled 1pt withcolor blue ;
\stopuseMPgraphic
```

The `text` graphic now becomes:

```
\startuseMPgraphic{text}
  \includeMPgraphic{text path}   \includeMPgraphic{text vars}
  \includeMPgraphic{text move}   \includeMPgraphic{text split}
  \includeMPgraphic{text cutoff} \includeMPgraphic{text draw}
\stopuseMPgraphic
```

Or, as graphic:



We are now ready for an attempt to calculate the shape of the text. For each line, we have to calculate the left and right intersection points, and since a line has a height and depth, we have to determine which part touches first.

```
\startuseMPgraphic{text calc}
  vardef found_point (expr lin, pat, sig) =
    pair a, b ;
    a := pat intersection_point (lin shifted (0,strutheight)) ;
    if intersection_found :
      a := a shifted (0,-strutheight) ;
    else :
      a := pat intersection_point lin ;
    fi ;
    b := pat intersection_point (lin shifted (0,-strutdepth)) ;
    if intersection_found :
      if sig :
```

```
        if xpart b > xpart a : a := b shifted (0,strutdepth) fi ;
      else :
        if xpart b < xpart a : a := b shifted (0,strutdepth) fi ;
      fi ;
    fi ;
    a
  enddef ;
\stopuseMPgraphic
```

Instead of using METAPOST's `intersectionpoint` macro, we use one that comes with CONTEXT. That way we don't get an error message when no point is found, and can use a boolean flag to take further action. Since we use a `vardef`, all calculations are hidden and the `a` at the end is returned, so that we can use this macro in an assignment. The `sig` variable is used to distinguish between the beginning and end of a line (the left and right subpath).

```
\startuseMPgraphic{text step}
  path line; pair lll, rrr ;

  for i=topskip step baselineskip until vsize :

    line := (ulcorner q -- urcorner q) shifted (0,-i) ;

    lll := found_point(line,l,true ) ;
    rrr := found_point(line,r,false) ;
\stopuseMPgraphic
```

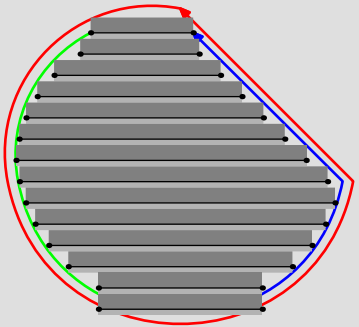
Here we divide the available space in lines. The first line starts at `strutheight` from the top.

We can now finish our graphic by visualizing the lines. Both the height and depth of the lines are shown.

```
\startuseMPgraphic{text line}
    fill (lll--rrr--rrr shifted (0,strutheight)--lll
      shifted (0,strutheight)--cycle) withcolor .5white ;
    fill (lll--rrr--rrr shifted (0,-strutdepth)--lll
      shifted (0,-strutdepth)--cycle) withcolor .7white ;
    draw lll withpen pencircle scaled 2pt ;
    draw rrr withpen pencircle scaled 2pt ;
    draw (lll--rrr) withpen pencircle scaled .5pt ;
\stopuseMPgraphic

\startuseMPgraphic{text done}
  endfor ;
\stopuseMPgraphic
```

The result is still a bit disappointing.

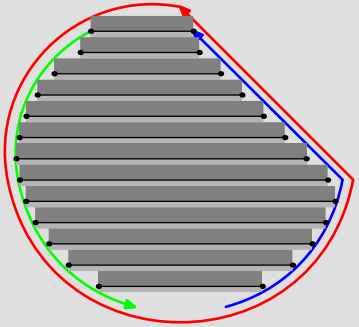
```
\startuseMPgraphic{text}
  \includeMPgraphic{text path}   \includeMPgraphic{text vars}
  \includeMPgraphic{text move}   \includeMPgraphic{text split}
  \includeMPgraphic{text cutoff} \includeMPgraphic{text draw}
  \includeMPgraphic{text calc}   \includeMPgraphic{text step}
  \includeMPgraphic{text line}   \includeMPgraphic{text done}
\stopuseMPgraphic
```



In order to catch the overflow at the bottom, we need to change the `for`–loop a bit, so that the number of lines does not exceed the available space. The test that surrounds the assignment of `vvsize` makes sure that we get better results when we (on purpose) take a smaller height.

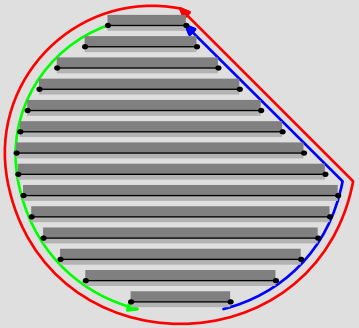
```
\startuseMPgraphic{text step}
  path line; pair lll, rrr ; numeric vvsize ;

  if (strutheight+strutdepth<baselineskip) :
    vvsize := vsize ;
  else :
    vvsize := (vsize div baselineskip) * baselineskip ;
  fi ;

  for i=topskip step baselineskip until vvsize :

    line := (ulcorner q -- urcorner q) shifted (0,-i) ;

    lll := found_point(line,l,true ) ;
```

```
    rrr := found_point(line,r,false) ;
\stopuseMPgraphic
```



We can manipulate the heigth and depth of the lines to give different (and maybe better) results.

```
\startuseMPgraphic{text vars}
baselineskip := 8pt ;
strutheight  := 4pt ;
strutdepth   := 2pt ;
offset       := 4pt ;
topskip      := 3pt ;
\stopuseMPgraphic
```



This kind of graphic trickery in itself is not enough to get TEX into typesetting within the bounds of a closed curve. Since METAPOST can write information to a file, and TEX can read such a file, a natural way to handle this is to let METAPOST write a \parshape specification.

```
\startuseMPgraphic{text macro}
  def provide_parshape (expr p, offset, baselineskip,
    strutheight, strutdepth, topskip) =

    \includeMPgraphic{text move}
    \includeMPgraphic{text split}
    \includeMPgraphic{text cutoff}
    \includeMPgraphic{text draw}
    \includeMPgraphic{text calc}
    \includeMPgraphic{text loop}
    \includeMPgraphic{text save}

  enddef ;
```

```
\stopuseMPgraphic
```

We have to adapt the for–loop to register the information about the lines. After the loop we write those values to a file using another loop.

```
\startuseMPgraphic{text loop}
  path line; pair lll, rrr ; numeric vvsize, n ; n := 0 ;

  if (strutheight+strutdepth<baselineskip) :
    vvsize := vsize ;
  else :
    vvsize := (vsize div baselineskip) * baselineskip ;
  fi ;

  for i=topskip step baselineskip until vvsize :

    line := (ulcorner q -- urcorner q) shifted (0,-i) ;

    lll := found_point(line,l,true ) ;
    rrr := found_point(line,r,false) ;

    n := n + 1 ;

    indent[n] := abs(xpart lll - xpart llcorner q) ;
    width[n]  := abs(xpart rrr - xpart lll) ;

  endfor ;
\stopuseMPgraphic

\startuseMPgraphic{text save}
  write "\parshape " & decimal n to "mfun-mp-data.txt" ;
```

```
  for i=1 upto n:
    write decimal indent[i]&"bp " &
          decimal  width[i]&"bp " to "mfun-mp-data.txt" ;
  endfor ;
  write EOF to "mfun-mp-data.txt" ;
\stopuseMPgraphic
```

We can call this macro using the part we used in the previous examples.

```
\startuseMPgraphic{text}
  \includeMPgraphic{text macro}

  path p ; p := ((0,1)..(-1,0)..(1,0)--cycle) scaled 65pt ;

  provide_parshape
   (p,                      % shape path
    .5*\baselinedistance,   % offset
    \baselinedistance,      % distance between lines
    \strutheight,           % height of a line
    \strutdepth,            % depth of a line
    \strutheight) ;         % height of first line
\stopuseMPgraphic
```

After we called `\useMPgraphic{text}`, the resulting file looks as follows. You can call up this file by its anonymous name `\MPdatafile`, since this macro gets the value of the graphic at hand.

```
\parshape 8
21.25648bp 46.39665bp
```

```
16.79666bp 54.3987bp
6.52979bp 77.31822bp
1.44792bp 95.05238bp
1.77025bp 107.38231bp
6.4069bp 106.13466bp
14.96152bp 89.02438bp
31.50317bp 55.9419bp
```

So, reading in this file at the start of a paragraph will setup TEX to follow this shape.

The final implementation is a bit more complicated since it takes care of paths that are not centered around the origin and don't start at the top point. We achieve this by moving the path to the center:

```
cp := center p ; q := p shifted - cp ;
```

The arbitrary starting point is taken care of by a slightly more complicated path cutter. First we make sure that the path runs counterclockwise.

```
if xpart directionpoint t of q < 0 : q := reverse q fi ;
```

Knowing this, we can split the path in two, using a slightly different splitter:

```
l := q cutbefore t ;
l := l if xpart point 0 of q < 0 : & q fi cutafter b ;
r := q cutbefore b ;
r := r if xpart point 0 of q > 0 : & q fi cutafter t ;
```

As always, when implementing a feature like this, some effort goes into a proper user interface. In doing so, we need some TEX trickery that goes beyond this text, like collecting text and splitting of the part needed. Also, we want to be able to handle multiple shapes at once, like the next example demonstrates.

10.6 Libraries

The macro discussed in the previous section is included in one of the METAPOST libraries, so we first have to say:

```
\useMPlibrary[txt]
```

We define four shapes. They are not really beautiful, but they demonstrate what happens in border cases. For instance, too small first lines are ignored. First we define a circle. Watch how the dimensions are set in the graphic. The arguments passed to `build_parshape` are: path, an offset, an additional horizontal and vertical displacement, the baseline distance, the height and depth of the line, and the height of the first line (topskip in TEX terminology). The height and depth of a line are often called strut height and depth, with a strut being an invisible character with maximum dimensions.

```
\startuseMPgraphic{test 1}
  path p ; p := fullcircle scaled 6cm ;

  build_parshape(p,6pt,0,0,\baselinedistance,
    \strutheight,\strutdepth,\strutheight) ;

  draw p withpen pencircle scaled 1pt ;
\stopuseMPgraphic
```

The second shape is a diamond. This is a rather useless shape, unless the text suits the small lines at the top and bottom.

```
\startuseMPgraphic{test 2}
  path p ; p := fullsquare rotated 45 scaled 5cm ;
  build_parshape(p,6pt,0,0,\baselinedistance,
    \strutheight,\strutdepth,\strutheight) ;
  draw p withpen pencircle scaled 1pt ;
\stopuseMPgraphic
```

The third and fourth shape demonstrate that providing a suitable offset is not always trivial.

```
\startuseMPgraphic{test 3}
  numeric w, h ; w := h := 6cm ;
  path p ; p := (.5w,h) -- (0,h) -- (0,0) -- (w,0) &
    (w,0) .. (.75w,.5h) .. (w,h) & (w,h) -- cycle ;
  build_parshape(p,6pt,0,0,\baselinedistance,
    \strutheight,\strutdepth,\strutheight) ;
  draw p withpen pencircle scaled 1pt ;
\stopuseMPgraphic
```

Contrary to the first three shapes, here we use a different path for the calculations and the drawing. Watch carefully! If, instead of an offset, we pass a path, METAPOST is able to calculate the right dimensions and offsets. This is needed, since we need these later on.

```
\startuseMPgraphic{test 4}
```

```
    numeric w, h, o ;

    def shape = (o,o) -- (w-o,o) & (w-o,o) .. (.75w-o,.5h) ..
      (w-2o,h-o) & (w-2o,h-o) -- (o,h-o) -- cycle
    enddef ;

    w := h := 6cm ; o := 6pt ; path p ; p := shape ;
    w := h := 6cm ; o := 0pt ; path q ; q := shape ;

    build_parshape(p,q,6pt,6pt,\baselinedistance,
      \strutheight,\strutdepth,\strutheight) ;

    draw q withpen pencircle scaled 1pt ;
  \stopuseMPgraphic
```

Since we also want these graphics as backgrounds, we define them as overlays. If you don't want to show the graphic, you may omit this step.

```
\defineoverlay[test 1][\useMPgraphic{test 1}]
\defineoverlay[test 2][\useMPgraphic{test 2}]
\defineoverlay[test 3][\useMPgraphic{test 3}]
\defineoverlay[test 4][\useMPgraphic{test 4}]
```

As text, we use a quote from Douglas R. Hofstadter's book "Metamagical Themas, Questing for the Essence of Mind and Pattern". Watch how we pass a list of shapes.

```
\startshapetext[test 1,test 2,test 3,test 4]
  \forgetall % as it says
  \setupalign[verytolerant,stretch,normal]%
```

```
  \input douglas % Douglas R. Hofstadter
\stopshapetext
```

Finally we combine text and shapes. Since we also want a background, we use \framed. The macros \parwidth and \parheight are automatically set to the current shape dimensions. The normal result is shown in **figure 10.3**.

```
\startbuffer
\setupframed
  [offset=overlay,align=normal,frame=off,
   width=\parwidth,height=\parheight]
\startcombination[2*2]
  {\framed[background=test 1]{\getshapetext}} {test 1}
  {\framed[background=test 2]{\getshapetext}} {test 2}
  {\framed[background=test 3]{\getshapetext}} {test 3}
  {\framed[background=test 4]{\getshapetext}} {test 4}
\stopcombination
\stopbuffer
```

By using a buffer we keep \placefigure readable.

```
\placefigure
  [here][fig:shapes]
  {A continuous text, typeset in a non||standard shape,
   spread over four areas, and right alligned.}
  {\getbuffer}
```

Figure 10.3 A continuous text, typeset in a non–standard shape, spread over four areas.

The traced alternative is shown in **figure 10.4**. This one is defined as:

```
\placefigure
  [here][fig:traced shapes]
  {A continuous text, typeset in a non||standard shape,
   spread over four areas (tracing on).}
  {\startMPinclusions
     boolean trace_parshape ; trace_parshape := true ;
   \stopMPinclusions
   \getbuffer}
```

We can combine all those tricks, although the input is somewhat fuzzy. First we define a quote typeset in a circular paragraph shape.

```
\startuseMPgraphic{center}
  build_parshape(fullcircle scaled 8cm,0,0,0,\baselinedistance,
    \strutheight,\strutdepth,\strutheight) ;
\stopuseMPgraphic

\startshapetext[center]
  \input douglas
\stopshapetext

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

We will surround this text with a circular line, that we define as follows. By using a buffer we keep things organized.

```
\startbuffer[circle]
```

Figure 10.4 A continuous text, typeset in a non–standard shape, spread over four areas (tracing on).

```
\startuseMPgraphic{followtokens}
  path RotPath ; RotPath := reverse fullcircle
    rotatedaround(origin,90)
    xscaled \overlaywidth yscaled \overlayheight ;
  drawoptions (withcolor .625red) ;
\stopuseMPgraphic

\followtokens
  {This is just a dummy text, kerned by T{\kern
   -.1667em\lower .5ex\hbox {E}}{\kern -.125emX} and typeset
   in a circle using {\setMFPfont M}{\setMFPfont
   E}{\setMFPfont T}{\setMFPfont A}{\setMFPfont
   P}{\setMFPfont O}{\setMFPfont S}{\setMFPfont T}.\quad}
\stopbuffer

\defineoverlay[edge][{\getbuffer[circle]}]
```

The text and graphics come together in a framed text:

```
\startbuffer[quote]
\framed
  [offset=6pt,background=edge,frame=off]
  {\getshapetext}
\stopbuffer

\placefigure
  {One more time Hofstadter's quotation.}
  {\getbuffer[quote]}
```

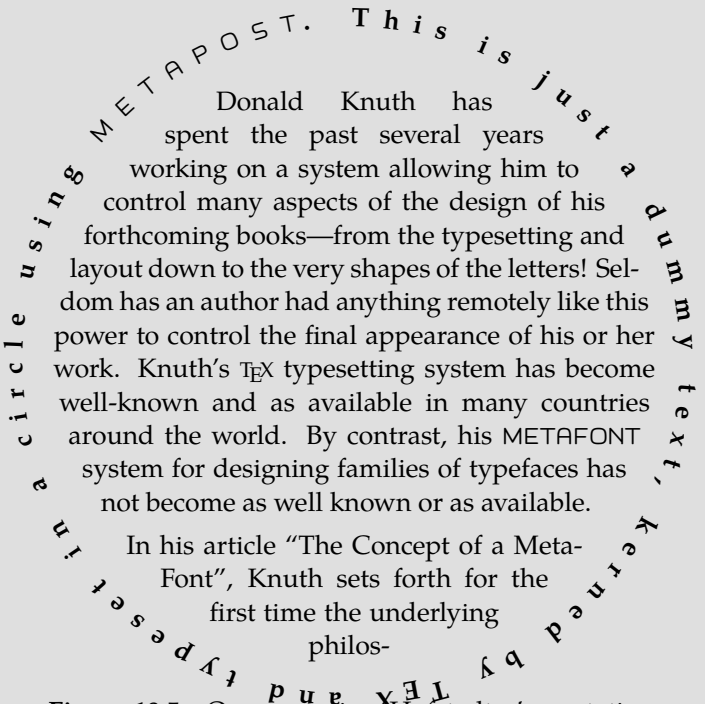


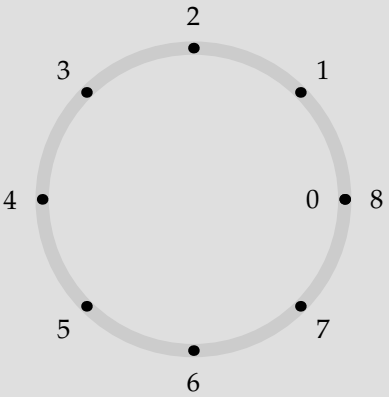
Figure 10.5 One more time Hofstadter's quotation.

Here also, I will rewrite things a bit so that we can avoid `\startMPdrawing` *outside the macro, and thereby avoid problems. I can also add the maps cdrom cover as example.*

11 Debugging

Those familiar with CONTEXT will know that it has quite some visual debugging features build in. So, what may you expect of the METAPOST macros that come with CONTEXT? In this chapter we will introduce a few commands that show some insight in what METAPOST is doing to your paths.

Since the outcome of METAPOST code is in many respects more predictable than that of TEX code, we don't need that advanced visual debugging features. Nevertheless we provide a few, that are all based on visualizing paths.



This visualization is achieved by using dedicated drawing commands:

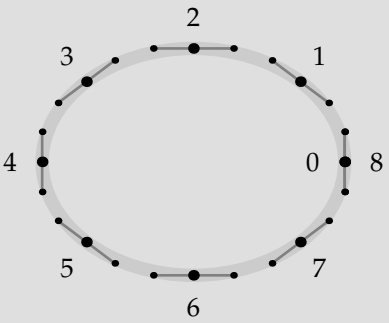
```
path p ; p := fullcircle scaled 4cm ;
```

```
drawpath p ; drawpoints p ; drawpointlabels p ;
```

Since control points play an important role in defining the shape, visualizing them may shed some insight in what METAPOST is doing.

```
path p ; p := fullcircle xscaled 4cm yscaled 3cm ;
drawpath p ; drawcontrollines p ;
drawpoints p ; drawcontrolpoints p ; drawpointlabels p ;
```

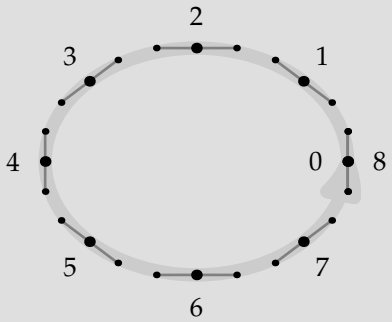
The pre and post control points show up as small dots and are connected to their parent point with thin lines.



You can deduce the direction of a path from the way the points are numbered, but using an arrow to indicate the direction is more clear.

```
path p ; p := fullcircle xscaled 4cm yscaled 3cm ;
drawarrowpath p ; drawcontrollines p ;
drawpoints p ; drawcontrolpoints p ; drawpointlabels p ;
```

The `drawarrowpath` is responsible for the arrow. Especially when you are in the process of defining macros that have to calculate intersections or take subpaths, knowing the direction may be of help.



The next table summarizes the special drawing commands:

`drawpath`	the path
`drawarrowpath`	the direction of the path
`drawcontrollines`	the lines to the control points
`drawpoints`	the points that make up the path
`drawcontrolpoints`	the control points of the points
`drawpointlabels`	the numbers of the points

You can set the characteristics of these like you set `drawoptions`. The default settings are as follows:

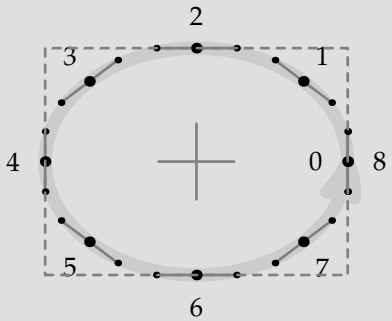
```
drawpathoptions    (withpen pencircle scaled 5   withcolor .8white) ;
drawpointoptions   (withpen pencircle scaled 4   withcolor   black) ;
drawcontroloptions(withpen pencircle scaled 2.5 withcolor   black) ;
```

```
drawlineoptions   (withpen pencircle scaled 1   withcolor .5white) ;
drawlabeloptions  () ;
```

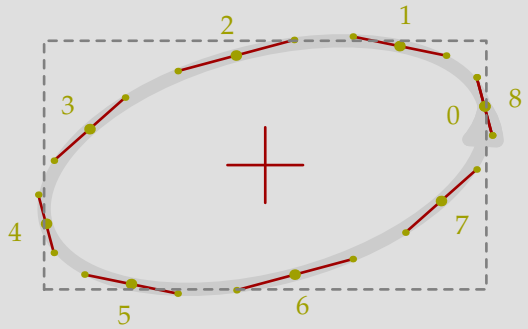
Two more options are `draworiginoptions` and `drawboundoptions` which are used when visualizing the bounding box and origin.

```
swappointlabels  := true ;
path p ; p := fullcircle xscaled 4cm yscaled 3cm ;
drawarrowpath p ; drawcontrollines p ;
drawpoints p ; drawcontrolpoints p ; drawpointlabels p ;
drawboundingbox p ; draworigin ;
```

In this example we have set `swappointlabels` to change the place of the labels. You can set the variable `originlength` to tune the appearance of the origin.



You can pass options directly, like you do with `draw` and `fill`. Those options override the defaults.



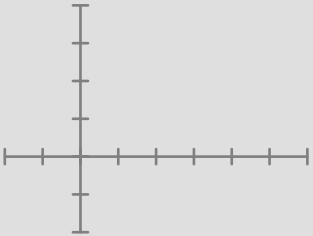
Here we used the options:

```
path p ; p := fullcircle xscaled 6cm yscaled 3cm rotated 15 ;
drawarrowpath     p ;
drawcontrollines  p withcolor .625red ;
drawpoints        p withcolor .625yellow ;
drawcontrolpoints p withcolor .625yellow ;
drawpointlabels   p withcolor .625yellow ;
drawboundingbox   p ;
draworigin          withcolor .625red ;
```

Sometimes it makes sense to draw a simple coordinate system, and for that purpose we have three more macros. They draw axis and tickmarks.

```
drawticks unitsquare xscaled 4cm yscaled 3cm shifted (-1cm,-1cm) ;
```

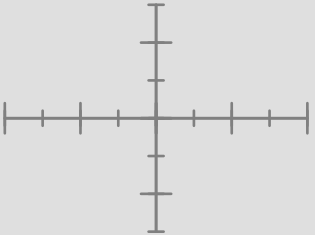
The system drawn is based on the bounding box specification of the path passed to the macro. You can also draw one axis, using `drawxticks` or `drawyticks`. Here we show the previous command.



By default, the ticks are placed at .5cm distance, but you can change this by setting `tickstep` to a different value.

```
tickstep := 1cm ; ticklength := 2mm ;
drawticks fullsquare xscaled 4cm yscaled 3cm ;
tickstep := tickstep/2 ; ticklength := ticklength/2 ;
drawticks fullsquare xscaled 4cm yscaled 3cm ;
```

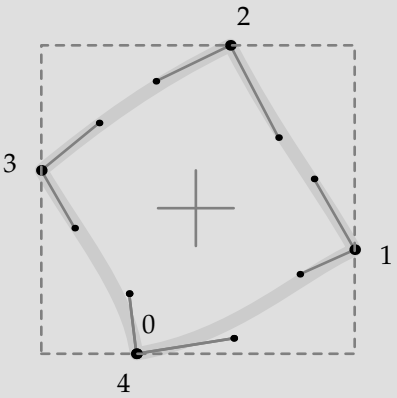
The `ticklength` variable specifies the length of a tick. Here we manipulated both the variables to get a more advanced system.



If visualizing a path would mean that we would have to key in al those draw–commands, you could hardly call it a comfortable tool. Therefore, we can say:

```
drawwholepath fullsquare scaled 3cm rotated 30 randomized 5mm ;
```

The `drawwholepath` command shows everything except the axis.



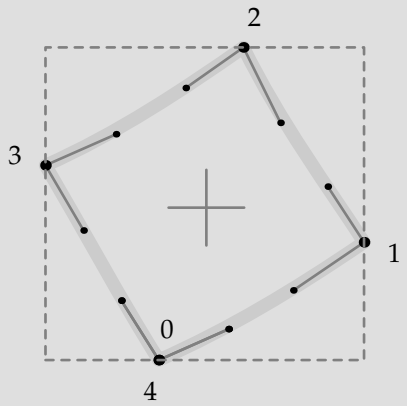
If even this is too much labour, you may say:

```
visualizepaths ;
```

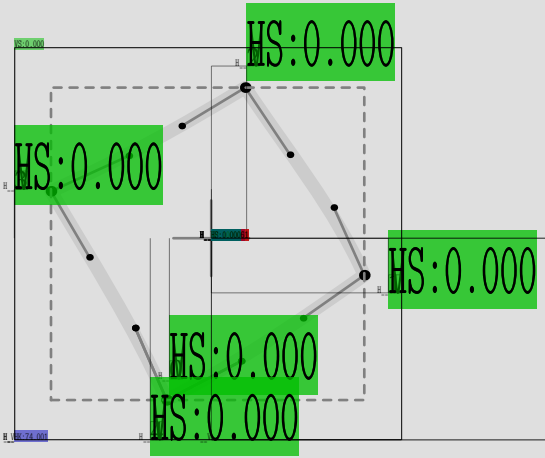
This redefines the `draw` and `fill` command in such a way that they also show all the information.

```
visualizepaths ;
draw fullsquare scaled 3cm rotated 30 randomized 2mm ;
```

You may compare this feature to the `\showmakeup` command available in CONTEXT, that redefines the TEX primitives that deal with boxes, glues, penalties, and alike.



Of course you may want to take a look at the METAPOST manual for its built in (more verbose) tracing options. One command that may prove to be useful is `show`, that you can apply to any variable. This command reports the current value (if known) to the terminal and log file.



The previous picture shows what is typeset when we also say `\showmakeup`. This command visualizes TEX's boxes, skips, kerns and penalties. As you can see, there are some boxes involved, which is due to the conversion of METAPOST output to PDF.

```
\startlinecorrection[blank]
... the graphic ...
\stoplinecorrection
```

The small bar is a kern and the small rectangles are penalties. More details on this debugger can be found in the CONTEXT manuals and the documentation of the modules involved.

12 Defining styles

Since the integration of METAPOST into CONTEXT, a complete new range of layout fetaures became available. In this document we have introduced several ways to include graphics in your document definition. In this chapter we go one step further and make dynamic graphics part of a document style.

12.1 Adaptive buttons

So far we have seen a lot of graphic ingredients that you can use to make your documents more attractive. In this chapter we will define a simple document style. This style was written for the PDFTEX presentations at the TUG 2000 conference in Oxford (UK).

This style exploits a few tricks, like graphics calculated using positional information. It also demonstrates how you can make menu buttons that dynamically adapt their shapes to the rest of the page layout.

page 1

page 2

page 3

page 4

page 5

Later we will see an instance with some more randomness in the graphics. While writing this style, the random alternative made me think of those organic buildings with non equal windows —we have a few of those in

The Netherlands—, so I decided to label this style as `pre-organic`. If you use CONTEXT, you can load this style with:

```
\usemodule[pre-organic]
```

At the end of this file, there is a small test file, so when you process the file `s-pre-19.tex`[14] with the options `--mode=demo` and `--pdf`, you will get a demo document.

We use one of the standard screen 'paper' sizes, and map it onto the same size, so that we get a nicely cropped page. Other screen sizes are `S4` and `S5`.

```
\setuppapersize[S6][S6]
```

Like in this METAFUN manual, we use the Palatino as main bodyfont. This font is quite readable on even low resolution screens, although I admit that this style is developed using an 1400 × 1050 pixel LCD screen, so the author may be a little biased.

```
\setupbodyfont[ppl]
```

The layout specification sets up a text area and a right edge area where the menus will go (see chapter ?? for a more in depth discussion on the layout areas). Watch how we use a rather large edge distance. By setting the header and footer dimensions to zero, we automatically get rid of page body ornaments, like the page number.

```
\setuplayout
  [topspace=48pt,
   backspace=48pt,
```

[14] This style is the 19th presentation style. Those numbered styles are internally mapped onto more meaningful names like in this case `pre-organic`.

```
    cutspace=12pt,
    width=400pt,
    margin=0cm,
    rightedge=88pt,
    rightedgedistance=48pt,
    header=0cm,
    footer=0cm,
    height=middle]
```

We use a moderate, about a line height, inter–paragraph white space.

```
\setupwhitespace[big]
```

Of course we use colors, since on computer displays they come for free.

```
\setupcolors[state=start]

\definecolor [red]    [r=.75]
\definecolor [yellow] [r=.75,g=.75]
\definecolor [gray]   [s=.50]
\definecolor [white]  [s=.85]
```

Because it is an interactive document, we have to enable hyperlinks and alike. However, in this style, we disable the viewer's 'highlight a hyperlink when it's clicked on' feature. We will use a menu, so we enable menus. Later we will see the contras`t color —hyperlinks gets that color when we are already on the location— in action.

```
\setupinteraction
  [state=start,
```

```
    click=off,
    color=red,
    contrastcolor=gray,
    menu=on]
```

The menu itself is set up as follows. Because we will calculate menu buttons based on their position on the page, we have to keep track of the positions. Therefore, we set the `position` variable to `yes`.

```
\setupinteractionmenu
  [right]
  [frame=off,
   position=yes,
   align=middle,
   topoffset=-.75cm,
   bottomoffset=-.75cm,
   color=gray,
   contrastcolor=gray,
   style=bold,
   before=,
   after=]
```

The menu content is rather sober: just a list of topics and a close button. Later we will define the command that generates topic entries. The alternative `right` lets the topic list inherit its characteristics from the menu.

```
\startinteractionmenu[right]
  \placelist[Topic][alternative=right]
  \vfill
```

```
    \but [CloseDocument] close \\
  \stopinteractionmenu
```

We have now arrived at the more interesting part of the style definition: the graphic that goes in the page background. Because this graphic will change, we define a useable METAPOST graphic. Page backgrounds are recalculated each page, opposite to the other backgrounds that are calculated when a new background is defined, or when repetitive calculation is turned on.

```
\setupbackgrounds [page] [background=page]
\defineoverlay    [page] [\useMPgraphic{page}]
\setupMPvariables [page] [alternative=3]
```

We will implement three alternative backgrounds. First we demonstrate the relatively simple super ellipsed one. The main complication is that we want the button shapes to follow the right edge of the curve that surrounds the text. We don't know in advance how many lines of text there will be in a button, and we also don't know at what height it will end up. Therefore, we need to calculate each button shape independently and for that purpose we need to know its position (see **chapter 5**). In **figure 12.1** you can see what lines we need in order to be calculate the button shapes.

We separate the calculation of the button shape from the rest by embedding it in its own usable graphic container. The `StartPage–StopPage` pair takes care of proper placement of the whole graphic.

```
\startuseMPgraphic{page}

  \includeMPgraphic{rightsuperbutton}

  StartPage ;

    path p, q ; pickup pencircle scaled 3pt ;
```

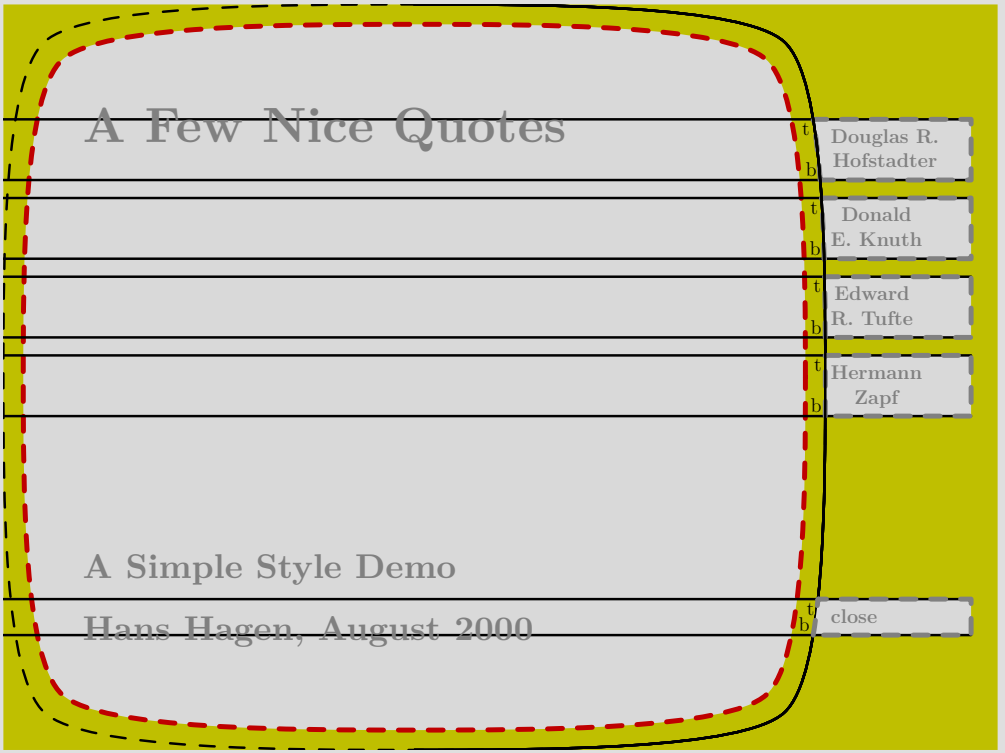


Figure 12.1 The lines used to calculate the button shapes.

```
    p := Field[Text][Text] enlarged 36pt superellipsed .90 ;

    fill Page withcolor \MPcolor{yellow} ;
    fill p    withcolor \MPcolor{white} ;
    draw p    withcolor \MPcolor{red} ;

    p := Field[Text][Text] enlarged 48pt superellipsed .90 ;

    def right_menu_button (expr nn, rr, pp, xx, yy, ww, hh, dd) =
      if (pp>0) and (rr>0) :
        q := rightsuperbutton(p,xx,yy,RightEdgeWidth,hh) ;
        fill q withcolor \MPcolor{white} ;
        draw q withcolor if rr=2 : \MPcolor{gray}
                         else    : \MPcolor{red}  fi ;
      fi ;
    enddef ;

    \MPmenubuttons{right}

  StopPage ;
\stopuseMPgraphic
```

The TEX macro `\MPmenubuttons` expands into a list of (in this case four) calls to the METAPOST macro `right_menu_button`. This list is generated by CONTEXT when it generates the menu. Because the page background is applied last, this list is available at that moment.

```
... (expr nn, rr, pp, xx, yy, ww, hh, dd) ...
```

This rather long list of arguments represents the following variables: number, referred page, current page, x coordinate, y coordinate, width, height and depth. The last six variables originate from the positioning

mechanism. Because the variables are only available after a second TEX pass, we only draw a button shape when the test for the page numbers succeeds.

```
\startuseMPgraphic{rightsuperbutton}
  vardef rightsuperbutton (expr pat, xpos, ypos, wid, hei) =

    save p, ptop, pbot, t, b, edge, shift, width, height ;
    path p, ptop, pbot ; pair t, b ;
    numeric edge, shift, width, height ;

    edge  := xpos + wid ; shift := ypos + hei ;

    p := rightpath pat ;

    ptop := ((-infinity,shift)--(edge,shift)) ;
    pbot := ((-infinity,shift-hei)--(edge,shift-hei)) ;

    t := p intersectionpoint ptop ;
    b := p intersectionpoint pbot ;

    p := subpath(0,xpart (p intersectiontimes ptop)) of p ;
    p := subpath(xpart (p intersectiontimes pbot),length(p)) of p ;

    (p --               t -- point 1 of ptop &
          point 1 of ptop -- point 1 of pbot &
          point 1 of pbot -- b
       -- cycle)

  enddef ;
\stopuseMPgraphic
```

The calculation of the button itself comes down to combining segments of the main shape and auxiliary lines. The `rightpath` macro returns the right half of the path provided. This half is shown as a non dashed line.

Topics are identified with `\Topic`, which is an instance of chapter headings. The number is made invisible. Since it still is a numbered section header, CONTEXT will write the header to the table of contents.

```
\definehead [Topic] [chapter]
\setuphead  [Topic] [number=no]
```

We will use a bold font in the table of contents. We also force a complete list.

```
\setuplist
  [Topic]
  [criterium=all,
   style=bold,
   before=,
   after=]
```

The `\TitlePage` macro looks horrible, because we want to keep the interface simple: a list of small sentences, separated by `\\`.

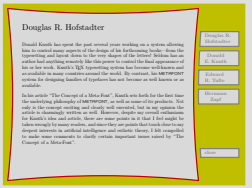
```
\def\TitlePage#1%
  {\startstandardmakeup
     \switchtobodyfont[big]
     \def\\{\vfill\bfb\let\\=\par}
     \bfd\setupinterlinespace\gray
     \vskip.5cm#1\\\vskip.5cm  % \\ is really needed -)
   \stopstandardmakeup}
```

A presentation that uses this style, may look like the one below. You can choose among three alternatives.

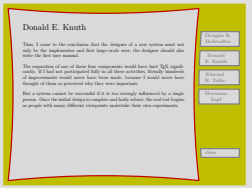
```
\useenvironment[pre-organic] \setupoutput[pdftex]

\setupMPvariables[page][alternative=1]

\starttext

\TitlePage
  {A Few Nice Quotes\\
   A Simple Style Demo\\
   Hans Hagen, August 2000}

\Topic {Douglas R. Hofstadter} \input douglas \page
\Topic {Donald E. Knuth}       \input knuth   \page
\Topic {Edward R. Tufte}       \input tufte   \page
\Topic {Hermann Zapf}          \input zapf    \page

\stoptext
```

We will not implement the two other alternative shapes: squeezed and randomized.

page 1



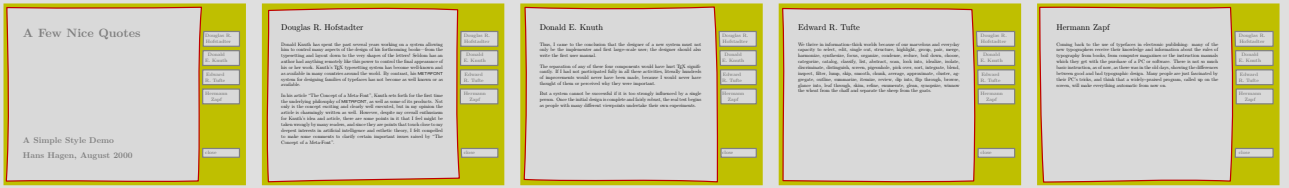
page 2



page 3

page 4

page 5



page 1 page 2 page 3 page 4 page 5

We combine all alternatives into one page graphic. The alternative is chosen by setting the `alternative` variable, as we demonstrated in the example.

```
\startuseMPgraphic{page}

  \includeMPgraphic{rightsuperbutton}

  StartPage ;

    numeric alternative, seed, superness, squeezeness, randomness ;
    path p, q ; transform t ;
```

This is one of those cases where a transform variable is useful. We need to store the random seed value because we want the larger path that is used in the calculations to have the same shape.

```
    alternative := \MPvar{alternative} ;
    seed        := uniformdeviate 100 ;

    if alternative > 10 :
      superness   := .85  + ((\realfolio-1)/\lastpage) * .25 ;
      squeezeness := 12pt - ((\realfolio-1)/\lastpage) * 10pt ;
```

```
else :
  superness   := .90 ;
  squeezeness := 12pt ;
fi ;

randomness := squeezeness ;

alternative := alternative mod 10 ;
```

If you read closely, you will notice that when we add 10 to the alternative, we get a page dependant graphic. So, in fact we have five alternatives. We use CONTEXT macros to fetch the (real) page number and the number of the last page. In further calculations we use the lower alternative numbers, which is why we apply a `mod`.

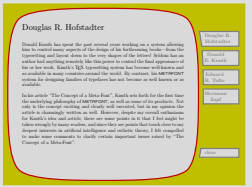
The rest of the code is not so much different from the previous definition. The hard coded point sizes match the page dimensions (600pt by 450pt) quite well.

```
t := identity if alternative=3: shifted (9pt,-9pt) fi ;

randomseed := seed ;

p := Field[Text][Text] enlarged if
  alternative = 1 : 36pt superellipsed superness   elseif
  alternative = 2 : 36pt squeezed      squeezeness elseif
  alternative = 3 : 36pt randomized    randomness  else
                  : 36pt                           fi ;

pickup pencircle scaled 3pt ;

fill Page withcolor \MPcolor{yellow} ;
fill p    withcolor \MPcolor{white} ;
```

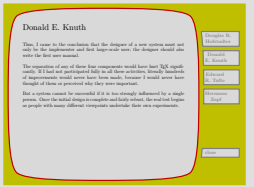
```
    draw p    withcolor \MPcolor{red} ;

    randomseed := seed ;

    p := ( Field[Text][Text] enlarged if
      alternative = 1 : 48pt superellipsed superness   elseif
      alternative = 2 : 48pt squeezed      squeezeness elseif
      alternative = 3 : 36pt randomized    randomness  else
                      : 48pt fi ) transformed t ;

    def right_menu_button (expr nn, rr, pp, xx, yy, ww, hh, dd) =
      if (pp>0) and (rr>0) :
        q := rightsuperbutton(p,xx,yy,RightEdgeWidth,hh) ;
        fill q withcolor \MPcolor{white} ;
        draw q withcolor if rr=2 : \MPcolor{gray}
                         else    : \MPcolor{red}  fi ;
      fi ;
    enddef ;

    \MPmenubuttons{right}

  StopPage ;
\stopuseMPgraphic
```

When we choose the alternatives 21 and 22 we get this result:

page 1



page 2

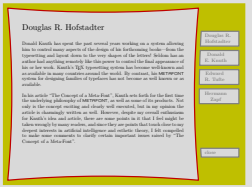


page 3

page 4

page 5

page 1



page 2

page 3

page 4

page 5

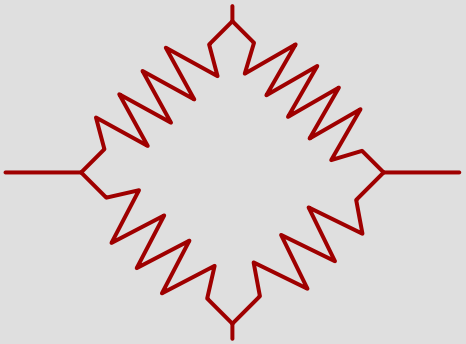
13 A few applications

For those who need to be inspired, we will demonstrate how METAPOST can be used to enhance your document with simple graphics. In these examples we will try to be not too clever, simply because we lack the experience to be that clever. The real tricks can be found in the files that come with METAPOST.

13.1 Simple drawings

In the words of John Hobby, the creator of METAPOST, "METAPOST is particularly well–suited for generating figures for technical documents where some aspects of a picture may be controlled by mathematical or geometrical constraints that are best expressed symbolically. In other words, METAPOST is not meant to take the place of a freehand drawing tool or even an interactive graphics editor".

An example of such a picture is the following one, which is dedicated to David Arnold, who asked me once how to draw a spring. So, imagine that we want to draw a schematic view of a system of four springs.



A rather natural way to define such a system is:

```
z1 = (+2cm,0) ; z2 = (0,+2cm) ;
z3 = (-2cm,0) ; z4 = (0,-2cm) ;

pickup pencircle scaled 1.5pt ;

drawoptions (withcolor .625red) ;

draw spring (z1, z2, .75cm, 2, 10) ; draw z1 -- 1.5 z1 ;
draw spring (z2, z3, .75cm, 2,  9) ; draw z2 -- 1.1 z2 ;
draw spring (z3, z4, .75cm, 2,  8) ; draw z3 -- 1.5 z3 ;
draw spring (z4, z1, .75cm, 2,  7) ; draw z4 -- 1.1 z4 ;
```

Here, the macro `spring` takes 5 arguments: two points, the width of the winding, the length of the connecting pieces, and the number of elements (half windings). The definition of `spring` is less complicated than readable.

```
def spring (expr a, b, w, h, n) =
  ( ( (0,0) -- (0,h) --
      for i=1 upto n-1: (if odd(i) : - fi w/2,i+h) -- endfor
      (0,n+h) -- (0,n+2h) )
    yscaled ((xpart (b-a) ++ ypart (b-a))/(n+2h))
    rotatedaround(origin,-90+angle(b-a))
    shifted a )
enddef ;
```

First we build a path starting in the origin, going left or right depending on the counter being an odd number.

```
pat := (0,0) ;
```

```
for i=1 upto n-1:
  if odd(i) :
    pat := pat -- (-w/2,i) ;
  else :
    pat := pat -- (+w/2,i) ;
  fi ;
endfor ;
pat := pat -- (0,n) ;
```

Once you are accustomed to the way METAPOST interprets (specialists may say expand) the source code, you will start using `if` and `for` statements in assignments. The previous code can be converted in a one liner, using the pattern:

```
pat := for i=1 upto n-1: (x,y)-- endfor (0,n) ;
```

The loop splits out a series of `(x,y)--` but the last point is added outside the loop. Otherwise `pat` would have ended with a dangling `--`. Of course we need to replace `(x,y)` by something meaningful, so we get:

```
pat := for i=1 upto n-1: (if odd(i):-fi w/2,i)--endfor (0,n) ;
```

We scale this path to the length needed. The expression $b - a$ calculates a vector, starting at a and ending at b. In METAPOST, the expression `a++b` is identical to $\sqrt{a^2 + b^2}$. Thus, the expression `xpart (b-a) ++ ypart (b-a)` calculates the length of the vector $b - a$. Because the unscaled spring has length $n + 2h$, scaling by the expression `((xpart (b-a) ++ ypart (b-a)) / (n+2h))` gives the spring the same length as the vector $b - a$.

Because we have drawn our spring in the vertical position, we first rotate it 90 degrees clockwise to a horizontal position, and then rotate it through an angle equal to the angle in which the vector $b - a$ is pointing. After that, we shift it to the first point. The main complications are that we also want to draw connecting lines at

the beginning and end, as well as support springs that connect arbitrary points. Since no check is done on the parameters, you should be careful in using this macro.

When we want to improve the readability, we have to use intermediate variables. Since the macro is expected to return a path, we must make sure that the content matches this expectation.

```
vardef spring (expr a, b, w, h, n) =
  pair vec ; path pat ; numeric len ; numeric ang ;
  vec := (b-a) ;
  pat := for i=1 upto n-1: (if odd(i):-fi w/2,i)--endfor (0,n) ;
  pat := (0,0)--(0,h)-- pat shifted (0,h)--(0,n+h)--(0,n+2h) ;
  len := (xpart vec ++ ypart vec)/(n+2h) ;
  ang := -90+angle(vec) ;
  ( pat yscaled len rotatedaround(origin,ang) shifted a )
enddef ;
```

If you use `vardef`, then the last statement is the return value. Here, when `p := spring (z1, z2, .75cm, 2, 10)` is being parsed, the macro is expanded, the variables are kept invisible for the assignment, and the path at the end is considered to be the return value. In a `def` the whole body of the macro is 'pasted' in the text, while in a `vardef` only the last line is visible. We will demonstrate this with a simple example.

```
def one = (n,n) ; n := n+1 ; enddef ;
def two = n := n + 1 ; (n,n) enddef ;
```

Now, when we say:

```
pair a, b ; numeric n ; n= 10 ; a := one ; b := two ;
```

we definitely get an error message. This is because, when macro `two` is expanded, METAPOST sees something:

```
b := n := n + 1 ;
```

By changing the second definition in

```
vardef two = n := n + 1 ; (n,n) enddef ;
```

the increment is expanded out of sight for `b :=` and the pair `(n,n)` is returned.

We can draw a slightly better looking spring by drawing twice with a different pen. The following commands use the spring macro implemented by the `vardef`.

```
path p ; p :=
  (0,0)--spring((.5cm,0),(2.5cm,0),.5cm,0,10)--(3cm,0) ;

draw p withpen pencircle scaled 2pt ;
draw p withpen pencircle scaled 1pt withcolor .8white;
```

This time we get:

Since the `spring` macro returns a path, you can do whatever is possible with a path, like drawing an arrow:

Or even (watch how we use the neutral unit `u` to specify the dimensions):

This was keyed in as:

```
drawarrow
  (0,0)--spring((.5cm,0),(2.5cm,0),.5cm,0,10)--(3cm,0)
  withpen pencircle scaled 2pt withcolor .625red ;
```

and:

```
numeric u ; u := 1mm ; pickup pencircle scaled (u/2) ;
drawoptions (withcolor .625red) ;
draw (0,0)--spring((5u,0),(25u,0),5u,0,10)--(30u,0) ;
drawoptions (dashed evenly withcolor .5white) ;
draw (0,0)--spring((5u,0),(35u,0),(25/35)*5u,0,10)--(40u,0) ;
```

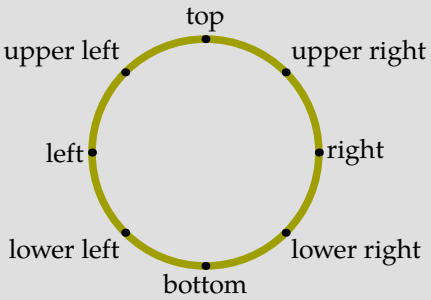
13.2 Free labels

The METAPOST label macro enables you to position text at certain points. This macro is kind of special, because it also enables you to influence the positioning. For that purpose it uses a special kind of syntax which we will not discuss here in detail.

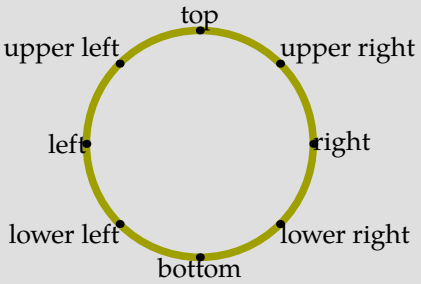
```
pickup pencircle scaled 1mm ;
path p ; p := fullcircle scaled 3cm ;
draw p withcolor .625yellow ;
dotlabel.rt   ("right"       , point 0 of p) ;
dotlabel.urt  ("upper right" , point 1 of p) ;
dotlabel.top  ("top"         , point 2 of p) ;
```

```
dotlabel.ulft ("upper left"  , point 3 of p) ;
dotlabel.lft  ("left"        , point 4 of p) ;
dotlabel.llft ("lower left"  , point 5 of p) ;
dotlabel.bot  ("bottom"      , point 6 of p) ;
dotlabel.lrt  ("lower right" , point 7 of p) ;
```

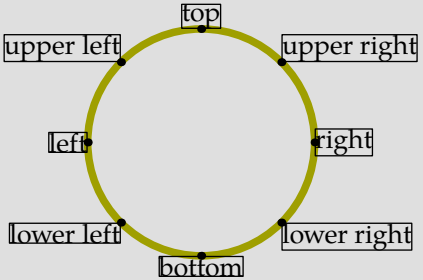
The `label` command just typesets a text, while `dotlabel` also draws a dot at the position of the label. The `thelabel` command returns a picture.



There is a numeric constant `labeloffset` that can be set to influence the distance between the point given and the content of the label. When we set the offset to zero, we get the following output.



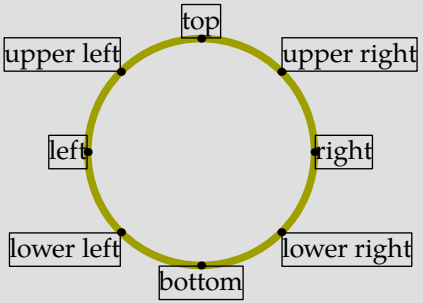
This kind of positioning works well as long as we know where we want the label to be placed. However, when we place labels automatically, for instance in a macro, we have to apply a few clever tricks. There are fore sure many ways to accomplish this goal, but here we will follow the mathless method.



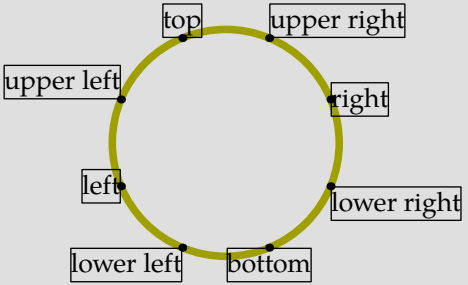
The previous graphic visualizes the bounding box of the labels. This bounding box is rather tight and therefore the placement of labels will always be suboptimal. Compare the alignment of the left- and rightmost labels. The `btex–etex` method is better, since then we can add struts, like:

```
btex \strut right etex
```

to force labels with uniform depths and heights. The next graphic demonstrates that this looks better indeed. Also, as TEX does the typesetting we get the current text font instead of the label font and the content will be properly typeset; for instance kerning will be applied when applicable. Spending some time on such details pays back in better graphics.



Now, what happens when we want to place labels in other positions? In the worst case, given that we place the labels manually, we end up in vague arguments in favour for one or the other placement.



Although any automatic mechanism will be sub–optimal, we can give it a try to write a macro that deals with arbitrary locations. This macro will accept three arguments and return a picture.

```
thefreelabel("some string or picture",a position,the origin)
```

Our testcase is just a simple `for` loop that places a series of labels. The `freedotlabel` macro is derived from `thefreelabel`.

```
pickup pencircle scaled 1mm ;
path p ; p := fullcircle scaled 3cm ;
draw p withcolor .625yellow ;
for i=0 step .5 until 7.5 :
  freedotlabel ("text" , point i of p, center p) ;
endfor ;
```

As a first step we will simply place the labels without any correction. We also visualize the bounding box.

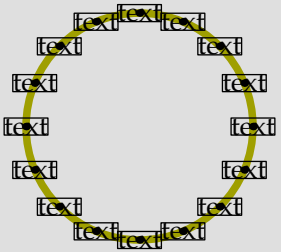
```
vardef thefreelabel (expr str, loc, ori) =
  save s ; picture s ; s := thelabel(str,loc) ;
  draw boundingbox s withpen pencircle scaled .5pt ;
  s
enddef ;
```

To make our lives more easy, we also define a macro that draws the dot as well as a macro that draws the label.

```
vardef freedotlabel (expr str, loc, ori) =
  drawdot loc ; draw thefreelabel(str,loc,ori) ;
enddef ;
```

```
vardef freelabel (expr str, loc, ori) =
  draw thefreelabel(str,loc,ori) ;
enddef ;
```

Now we get:



The original label macros permits us to align the label at positions, 4 corners and 4 points halfway the sides. It happens that circles are also composed of 8 points. Because in most cases the label is to be positioned in the direction of the center of a curve and the point at hand, it makes sense to take circles as the starting points for positioning the labels.

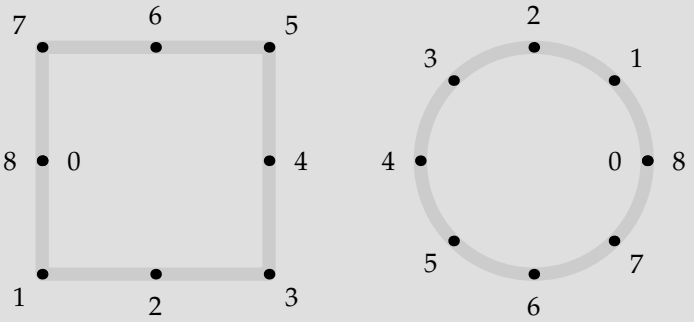
To help us in positioning, we define a special square path, `freesquare`. This path is constructed out of 8 points that match the positions that are used to align labels.

```
path freesquare ;

freesquare := ((-1,0)--(-1,-1)--(0,-1)--(+1,-1)--
               (+1,0)--(+1,+1)--(0,+1)--(-1,+1)--cycle) scaled .5 ;
```

We now show this free path together with a circle, using the following definitions:

```
drawpath        fullcircle scaled 3cm ;
drawpoints      fullcircle scaled 3cm ;
drawpointlabels fullcircle scaled 3cm ;
currentpicture := currentpicture shifted (5cm,0) ;
drawpath        freesquare scaled 3cm ;
drawpoints      freesquare scaled 3cm ;
drawpointlabels freesquare scaled 3cm ;
```

We use two drawing macros that are part of the suite of visual debugging macros.

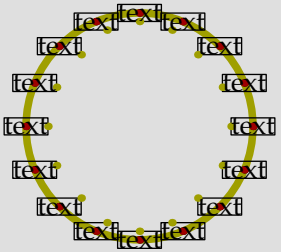


As you can see, point 1 is the corner point that suits best for alignment when a label is put at point 1 of the circle. We will now rewrite `thefreelabel` in such a way that the appropriate point of the associated `freesquare` is found.

```
vardef thefreelabel (expr str, loc, ori) =
  save s, p, q, l ; picture s ; path p, q ; pair l ;
  s := thelabel(str,loc) ;
```

```
    p := fullcircle scaled (2*length(loc-ori)) shifted ori ;
    q := freesquare xyscaled (urcorner s - llcorner s) ;
    l := point (xpart (p intersectiontimes (ori--loc))) of q ;
    draw q shifted loc withpen pencircle scaled .5pt ;
    draw l shifted loc withcolor .625yellow ;
    draw loc withcolor .625red ;
    s
  enddef ;
```

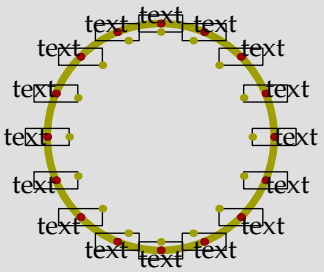
The macro xyscaled is part of METAFUN and scales in two directions at once. The METAPOST primitive `intersectiontimes` returns a pair of time values of the point where two paths intersect. The first part of the pair concerns the first path.



We are now a small step from the exact placement. If we change the last line of the macro into:

```
(s shifted -l)
```

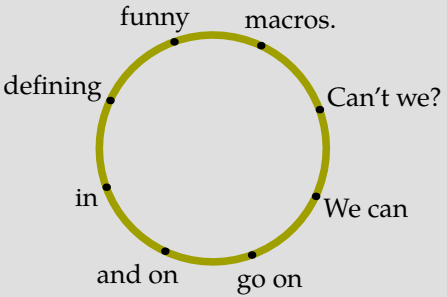
we get the displacement we want. Although the final look and feel is also determined by the text itself, the average result is quite acceptable.



Because we also want to pass pictures, and add a bit of offset too, the final implementation is slightly more complicated. The picture is handled with an additional condition, and the offset with the METAFUN macro `enlarged`.

```
newinternal freelabeloffset ; freelabeloffset := 3pt ;

vardef thefreelabel (expr str, loc, ori) =
  save s, p, q, l ; picture s ; path p, q ; pair l ;
  interim labeloffset := freelabeloffset ;
  s := if string str : thelabel(str,loc)
       else          : str shifted -center str shifted loc fi ;
  setbounds s to boundingbox s enlarged freelabeloffset ;
  p := fullcircle scaled (2*length(loc-ori)) shifted ori ;
  q := freesquare xyscaled (urcorner s - llcorner s) ;
  l := point (xpart (p intersectiontimes (ori--loc))) of q ;
  setbounds s to boundingbox s enlarged -freelabeloffset ;
  (s shifted -l)
enddef ;
```

Watch how we temporarily enlarge the bounding box of the typeset label text. We will now test this macro on a slightly rotated circle, using labels typeset by TEX. The `reverse` is there purely for cosmetic reasons, to suit the label texts.




```
pickup pencircle scaled 1mm ;
path p ; p := reverse fullcircle rotated -25 scaled 3cm ;
draw p withcolor .625yellow ; pair cp ; cp := center p ;
freedotlabel (btex \strut We can    etex, point 0 of p, cp) ;
freedotlabel (btex \strut go on     etex, point 1 of p, cp) ;
freedotlabel (btex \strut and on    etex, point 2 of p, cp) ;
freedotlabel (btex \strut in        etex, point 3 of p, cp) ;
freedotlabel (btex \strut defining  etex, point 4 of p, cp) ;
freedotlabel (btex \strut funny     etex, point 5 of p, cp) ;
freedotlabel (btex \strut macros.   etex, point 6 of p, cp) ;
freedotlabel (btex \strut Can't we? etex, point 7 of p, cp) ;
```

Unfortunately we can run into problems due to rounding errors. Therefore we use a less readable but more safe expression for calculating the intersection points. Instead of using point `loc` as endpoint we use `loc` shifted over a very small distance into the direction `loc` from `ori`. In the assignment to `l` we replace `loc` by:

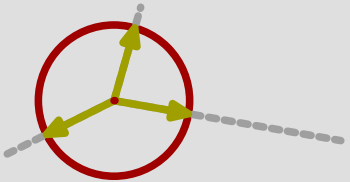
```
( (1+eps) * arclength(ori--loc) * unitvector(loc-ori) )
```

13.3 Marking angles

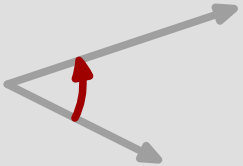
A convenient METAPOST macro is `unitvector`. When we draw a line segment from the origin to the point returned by this macro, the segment has a length of 1 base point. This macro has a wide range of applications, but some basic knowlegde of vector algebra is handy. The following lines of METAPOST code demonstrate the basics behind unitvectors.

```
pair uv ; pickup pencircle scaled 1mm ; autoarrows := true ;
draw fullcircle scaled 2cm withcolor .625red    ;
for i=(10,35), (-40,-20), (85,-15) :
  draw origin--i dashed evenly withcolor .625white ;
  drawarrow origin--unitvector(i) scaled 1cm withcolor .625yellow ;
endfor ;
draw origin withcolor .625red ;
```

The circle has a radius of 1cm, and the three line segments are drawn from the origin in the direction of the points that are passed as arguments. Because the vector has length of 1, we scale it to the radius to let it touch the circle. By setting `autoarrows` we make sure that the arrowheads are scaled proportionally to the linewidth of 1 mm.



An application of this macro is drawing the angle between two lines. In the METAPOST manual you can find two macros for drawing angles: `mark_angle` and `mark_rt_angle`. You may want to take a look at their definitions before we start developing our own alternatives.



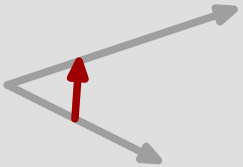
The previous graphic demonstrates what we want to accomplish: a circular curve indicating the angle between two straight lines. The lines and curve are drawn with the code:

```
pair a, b ; a := (2cm,-1cm) ; b := (3cm,1cm) ;
drawarrow origin--a ; drawarrow origin--b ;
drawarrow anglebetween(a,b) scaled 1cm withcolor .625red ;
```

where `anglebetween` is defined as:

```
def anglebetween (expr a, b) =
  (unitvector(a){a rotated 90} .. unitvector(b))
enddef ;
```

Both unitvectors return just a point on the line positioned 1 unit (later scaled to 1cm) from the origin. We connect these points by a curve that starts in the direction at the first point. If we omit the `a rotated 90` direction specifier, we get:



These definitions of `anglebetween` are far from perfect. If we don't start in the origin, we get the curve in the wrong place and when we swap both points, we get the wrong curve.

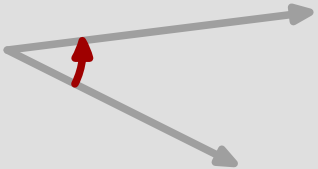
The solution for the displacement is given in the METAPOST manual and looks like this (we package the macro a bit different):

```
def anglebetween (expr endofa, endofb, common, length) =
  (unitvector (endofa-common){(endofa-common) rotated 90} ..
   unitvector (endofb-common)) scaled length shifted common
enddef ;
```

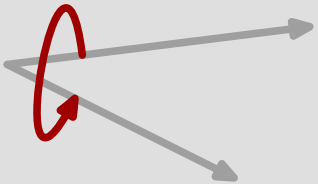
As you can see, we compensate for the origin of both vectors. This macro is called with a few more parameters. We need to pass the length, since we want to add the shift to the macro and the shift takes place after the scaling.

```
pair a, b, c ; a := (2cm,-1cm) ; b := (3cm,1cm) ; c := (-1cm,.5cm) ;
drawarrow c--a ; drawarrow c--b ;
drawarrow anglebetween(a,b,c,1cm) withcolor .625red ;
```

That the results are indeed correct, is demonstrated by the output of following example:



However, when we swap the points, we get:



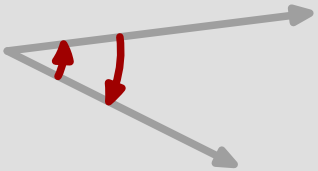
This means that instead of rotating over 90 degrees, we have to rotate over -90 or 270 degrees. That way the arrow will also point in the other direction. There are undoubtedly more ways to determine the direction, but the following method also demonstrates the use of `turningnumber`, which reports the direction of a path. For this purpose we compose a dummy cyclic path.

```
vardef anglebetween (expr endofa, endofb, common, length) =
  save tn ; tn := turningnumber(common--endofa--endofb--cycle) ;
show tn ;
  (unitvector(endofa-common){(endofa-common) rotated (tn*90)} ..
   unitvector(endofb-common)) scaled length shifted common
enddef ;
```

Because we use an intermediate variable, just to keep things readable, we have to use `vardef` to hide the assignment for the outside world. We demonstrate this macro using the following code:

```
pair a, b, c ; a := (2cm,-1cm) ; b := (3cm,1cm) ; c := (-1cm,.5cm) ;
drawarrow c--a ; drawarrow c--b ;
drawarrow anglebetween(a,b,c,0.75cm) withcolor .625red ;
drawarrow anglebetween(b,a,c,1.50cm) withcolor .625red ;
```

Watch how both arrows point in the direction of the line that is determined by the second point.



We now have the framework of an angle drawing macro ready and can start working placing the label.

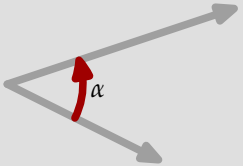
```
vardef anglebetween (expr endofa, endofb, common, length, str) =
  save curve, where ; path curve ; numeric where ;
  where := turningnumber (common--endofa--endofb--cycle) ;
  curve := (unitvector(endofa-common){(endofa-common) rotated (where*90)}
         .. unitvector(endofb-common)) scaled length shifted common ;
  draw thefreelabel(str,point .5 of curve,common) withcolor black ;
  curve
enddef ;
```

The macro `thefreelabel` is part of METAFUN and is explained in detail in **section 13.2**. This macro tries to place the label as good as possible without user interference.

```
pair a ; a := (2cm,-1cm) ; drawarrow origin--a ;
```

```
pair b ; b := (3cm, 1cm) ; drawarrow origin--b ;
drawarrow
  anglebetween(a,b,origin,1cm,btex $\alpha$ etex)
  withcolor .625red ;
```

Instead of a picture we may also pass a string, but using TEX by means of `btex–etex` often leads to better results.



Because in most cases we want the length to be consistent between figures and because passing two paths is more convenient than passing three points, the final definition looks slightly different.

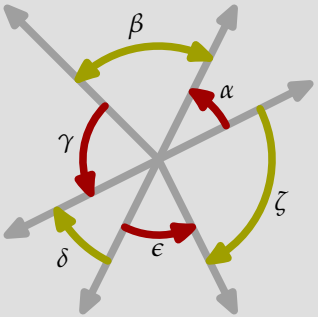
```
numeric anglelength ; anglelength := 20pt ;

vardef anglebetween (expr a, b, str) = % path path string
  save endofa, endofb, common, curve, where ;
  pair endofa, endofb, common ; path curve ; numeric where ;
  endofa := point length(a) of a ;
  endofb := point length(b) of b ;
  if round point 0 of a = round point 0 of b :
    common := point 0 of a ;
  else :
    common := a intersectionpoint b ;
  fi ;
```

```
  where := turningnumber (common--endofa--endofb--cycle) ;
  curve := (unitvector (endofa-common){(endofa-common) rotated (where*90)} ..
           unitvector (endofb-common)) scaled anglelength shifted common ;
  draw thefreelabel(str,point .5 of curve,common) withcolor black ;
  curve
enddef ;
```

This macro has a few more `if`'s than its predecessor. First we test if the label is a string, and if so, we calculate the picture ourselves, otherwise we leave this to the user.

```
path a, b, c, d, e, f ;
a := origin--( 2cm, 1cm) ; b := origin--( 1cm, 2cm) ;
c := origin--(-2cm, 2cm) ; d := origin--(-2cm,-1cm) ;
e := origin--(-1cm,-2cm) ; f := origin--( 1cm,-2cm) ;
for i=a, b, c, d, e, f : drawarrow i ; endfor ;
anglelength := 1.0cm ; drawoptions(withcolor .625red) ;
drawarrow         anglebetween(a,b,btex $\alpha  $ etex) ;
drawarrow         anglebetween(c,d,btex $\gamma  $ etex) ;
drawarrow         anglebetween(e,f,btex $\epsilon$ etex) ;
anglelength := 1.5cm ; drawoptions(withcolor .625yellow) ;
drawdblarrow      anglebetween(b,c,btex $\beta   $ etex) ;
drawarrow reverse anglebetween(d,e,btex $\delta  $ etex) ;
drawarrow         anglebetween(a,f,btex $\zeta   $ etex) ;
```

Because `anglebetween` returns a path, you can apply transformations to it, like reversing. Close reading of the previous code learns that the macro handles both directions.



Multiples of 90 degrees are often identified by a rectangular symbol. We will now extend the previously defined macro in such a way that more types can be drawn.

```
numeric anglelength ; anglelength := 20pt ;
numeric anglemethod ; anglemethod :=    1 ;

vardef anglebetween (expr a, b, str) = % path path string
  save pointa, pointb, common, middle, offset ;
  pair pointa, pointb, common, middle, offset ;
  save curve ; path curve ;
  save where ; numeric where ;
  if round point 0 of a = round point 0 of b :
    common := point 0 of a ;
  else :
    common := a intersectionpoint b ;
  fi ;
  pointa := point anglelength on a ;
```

```
  pointb := point anglelength on b ;
  where  := turningnumber (common--pointa--pointb--cycle) ;
  middle := ((common--pointa) rotatedaround (pointa,-where*90))
                          intersectionpoint
            ((common--pointb) rotatedaround (pointb, where*90)) ;
  if     anglemethod = 1 :
    curve  := pointa{unitvector(middle-pointa)}.. pointb;
    middle := point .5 along curve ;
  elseif anglemethod = 2 :
    middle := common rotatedaround(.5[pointa,pointb],180) ;
    curve  := pointa--middle--pointb ;
  elseif anglemethod = 3 :
    curve  := pointa--middle--pointb ;
  elseif anglemethod = 4 :
    curve  := pointa..controls middle..pointb ;
    middle := point .5 along curve ;
  fi ;
  draw thefreelabel(str, middle, common) withcolor black ;
  curve
enddef ;
```

Figure 13.1 shows the first three alternative methods implemented here. Instead of using `unitvectors`, we now calculate the points using the `arctime` and `arclength` primitives. Instead of complicated expressions, we use the METAFUN operators `along` and `on`. The following expressions are equivalent.

```
pointa := point anglelength on a ;
middle := point .5 along curve ;
```

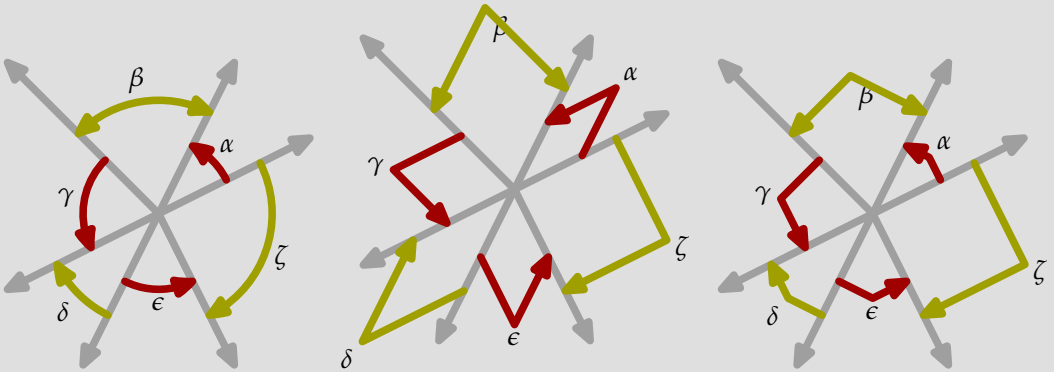


Figure 13.1 Three ways of marking angles.

```
pointa := point (arctime anglelength of a) of a ;
middle := arctime (.5(arclength curve)) of curve) of curve ;
```

The third method can be implemented in different, more math intensive ways, but the current implementation suits rather well and is understood by the author.

13.4 Color circles

In **chapter 3** we showed a few color circles. Drawing such a graphic can be done in several ways, and here we will show a few methods. First we will demonstrate how you can apply `cutafter` and `cutbefore`, next we

will show how the METAPOST macro `buildpath` can be used, and finally we will present a clean solution using `subpath`. We will assume that the circle is called with the macro:

```
colorcircle (4cm, red, green, blue) ;
```

We need to calculate seven paths. The first implementation does all the handywork itself and thereby is rather long, complicated and unreadable. It does not really use the strenghth of METAPOST yet.

```
vardef colorcircle (expr size, red, green, blue) =
  save r, g, b, rr, gg, bb, cc, mm, yy ;
  save b_r, b_g, g_r, g_b ;
  save radius ;

  path r, g, b, rr, bb, gg, cc, mm, yy ;
  pair b_r, b_g, g_r, g_b ;

  numeric radius ; radius := 3cm ;

  pickup pencircle scaled (radius/20) ;

  r := g := b := fullcircle scaled radius shifted (0,radius/4);

  r := r rotatedaround(origin, 15) ; % drawarrow r withcolor red   ;
  g := g rotatedaround(origin,135) ; % drawarrow g withcolor green ;
  b := b rotatedaround(origin,255) ; % drawarrow b withcolor blue  ;

  b_r :=         b intersectionpoint r ; % draw b_r ;
  b_g :=         b intersectionpoint g ; % draw b_g ;
  g_r := reverse g intersectionpoint r ; % draw g_r ;
  g_b := reverse g intersectionpoint b ; % draw g_b ;
```

```
  bb := b cutafter  b_r ; bb := bb cutbefore b_g ; % drawarrow bb ;
  gg := g cutbefore b_g ; gg := gg cutafter  g_r ; % drawarrow gg ;
  rr := r cutbefore g_r &        r cutafter  b_r ; % drawarrow rr ;

  cc := b cutbefore b_r ; cc := cc cutafter  g_b ; % drawarrow br ;
  yy := g cutbefore g_r ; yy := yy cutafter  g_b ; % drawarrow rg ;
  mm := r cutbefore g_r &        r cutafter  b_r ; % drawarrow gb ;

  bb := gg -- rr -- reverse bb -- cycle ;
  gg := bb rotatedaround(origin,120) ;
  rr := bb rotatedaround(origin,240) ;

  cc := mm -- cc -- reverse yy -- cycle ;
  yy := cc rotatedaround(origin,120) ;
  mm := cc rotatedaround(origin,240) ;

  fill fullcircle scaled radius withcolor white ;

  fill rr withcolor red   ; fill cc withcolor white-red   ;
  fill gg withcolor green ; fill mm withcolor white-green ;
  fill bb withcolor blue  ; fill yy withcolor white-blue  ;

  for i = rr,gg,bb,cc,mm,yy : draw i withcolor .5white ; endfor ;

  currentpicture := currentpicture xsized size ;
enddef ;
```

In determining the right intersection points, you need to know where the path starts and in what direction it moves. In case of doubt, drawing the path as an arrow helps. If you want to see the small paths used, you need to comment the lines with the `fill`'s and uncomment the lines with `draw`'s. Due to the symmetry and

the fact that we keep the figure centered around the origin, we only need to calculate two paths since we can rotate them.

There are for sure more (efficient) ways to draw such a figure, but this one demonstrates a few new tricks, like grouping. We use grouping here because we want to use `mm` to indicate the magenta path, and `mm` normally means millimeter. Within a group, you can save variables. These get their old values when the group is left.

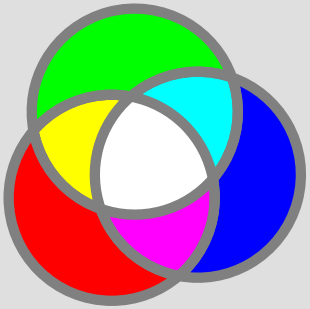
With `for` we process multiple paths after each other. In this case it hardly saves tokens, but it looks more clever.

One of the more efficient methods is using the `buildcycle` macro. This macro takes two or more paths and calculates the combined path. Although this is a rather clever macro, you should be prepared to help it a bit when paths have multiple intersection points. Again, we could follow a more secure mathematical method, but the next one took only a few minutes of trial and error. To save some memory, we redefine the `colors` graphic.

When we call this macro as:

```
colorcircle(4cm, red, green, blue) ;
```

we get:



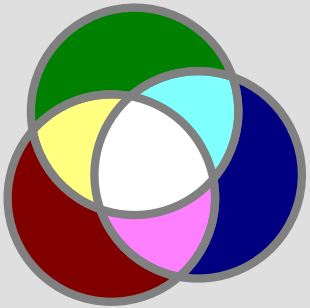
```
vardef colorcircle (expr size, red, green, blue) =
  save r, g, b, rr, gg, bb, cc, mm, yy ; save radius ;
  path r, g, b, rr, bb, gg, cc, mm, yy ; numeric radius ;

  radius := 5cm ; pickup pencircle scaled (radius/25) ;

  r := g := b := fullcircle scaled radius shifted (0,radius/4) ;

  r := r rotatedaround (origin, 15) ;
  g := g rotatedaround (origin,135) ;
  b := b rotatedaround (origin,255) ;

  r := r rotatedaround(center r,-90) ;
  g := g rotatedaround(center g, 90) ;

  gg := buildcycle(buildcycle(reverse r,b),g) ;
  cc := buildcycle(buildcycle(b,reverse g),r) ;

  rr := gg rotatedaround(origin,120) ;
```

```
  bb := gg rotatedaround(origin,240) ;

  yy := cc rotatedaround(origin,120) ;
  mm := cc rotatedaround(origin,240) ;

  fill fullcircle scaled radius withcolor white ;

  fill rr withcolor red   ; fill cc withcolor white-red   ;
  fill gg withcolor green ; fill mm withcolor white-green ;
  fill bb withcolor blue  ; fill yy withcolor white-blue  ;

  for i = rr,gg,bb,cc,mm,yy : draw i withcolor .5white ; endfor ;

  currentpicture := currentpicture xsized size ;
enddef ;
```

Since we don't want to duplicate a graphic, this time we show the dark alternatives.

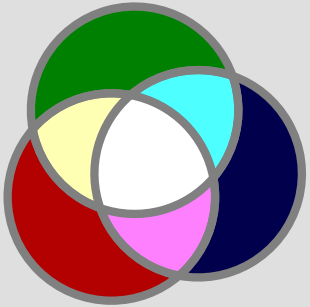
```
colorcircle(4cm, .5red, .5green, .5blue) ;
```

This kind of unsafe path calculations are very sensitive to breaking. Changing the `radius/4` into something else demonstrates this but we will not challenge this macro that much. Therefore, the 50% color circle shows up as:



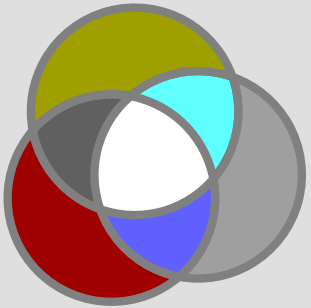
This command is part of METAFUN and can be used to determine nice color combinations by also looking at their complementary colors.

```
colorcircle (4cm, .7red, .5green, .3blue) ;
```



The next circle that we draw shows the three main colors used in this document. This circle is not that beautiful.

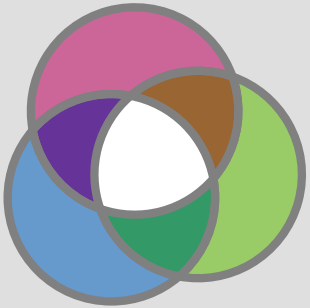
```
colorcircle(4cm,.625red,.625yellow,.625white) ;
```



This definition can be cleaned up a bit by using `transform`, but the fuzzy `buildcycle`'s remain.

```
vardef colorcircle (expr size, red, green, blue) =
  save r, g, b, rr, gg, bb, cc, mm, yy ; save radius ;
  path r, g, b, rr, bb, gg, cc, mm, yy ; numeric radius ;

  radius := 5cm ; pickup pencircle scaled (radius/25) ;

  transform t ; t := identity rotatedaround(origin,120) ;

  r := fullcircle scaled radius
   shifted (0,radius/4) rotatedaround(origin,15) ;

  g := r transformed t ; b := g transformed t ;

  r := r rotatedaround(center r,-90) ;
  g := g rotatedaround(center g, 90) ;
```

```
  gg := buildcycle(buildcycle(reverse r,b),g) ;
  cc := buildcycle(buildcycle(b,reverse g),r) ;

  rr := gg transformed t ; bb := rr transformed t ;
  yy := cc transformed t ; mm := yy transformed t ;

  fill fullcircle scaled radius withcolor white ;

  fill rr withcolor red   ; fill cc withcolor white-red   ;
  fill gg withcolor green ; fill mm withcolor white-green ;
  fill bb withcolor blue  ; fill yy withcolor white-blue  ;

  for i = rr,gg,bb,cc,mm,yy : draw i withcolor .5white ; endfor ;

  currentpicture := currentpicture xsized size ;
enddef ;
```



This rather nice circle is defined as:

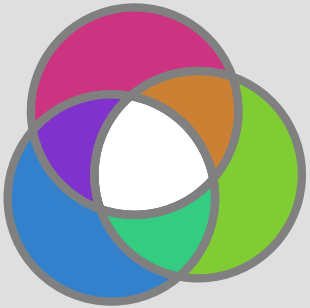
```
colorcircle(4cm,(.4,.6,.8),(.8,.4,.6),(.6,.8,.4));
```

The final implementation, which is part of METAFUN, is slightly more efficient.

```
vardef colorcircle (expr size, red, green, blue) =
  save r, g, b, c, m, y, w ; save radius ;
  path r, g, b, c, m, y, w ; numeric radius ;

  radius := 5cm ; pickup pencircle scaled (radius/25) ;

  transform t ; t := identity rotatedaround(origin,120) ;

  r := fullcircle rotated 90 scaled radius
         shifted (0,radius/4) rotatedaround(origin,135) ;

  b := r transformed t ; g := b transformed t ;

  c := buildcycle(subpath(1,7) of g, subpath(1,7) of b) ;
  y := c transformed t ; m := y transformed t ;

  w := buildcycle(subpath(3,5) of r,
         subpath(3,5) of g, subpath(3,5) of b) ;

  pushcurrentpicture ;

  fill r withcolor red   ;
  fill g withcolor green ;
  fill b withcolor blue  ;
  fill c withcolor white-red   ;
  fill m withcolor white-green ;
  fill y withcolor white-blue  ;
```

```
  fill w withcolor white ;
  for i = r,g,b,c,m,y : draw i withcolor .5white ; endfor ;
  currentpicture := currentpicture xsized size ;
  popcurrentpicture ;
enddef ;
```

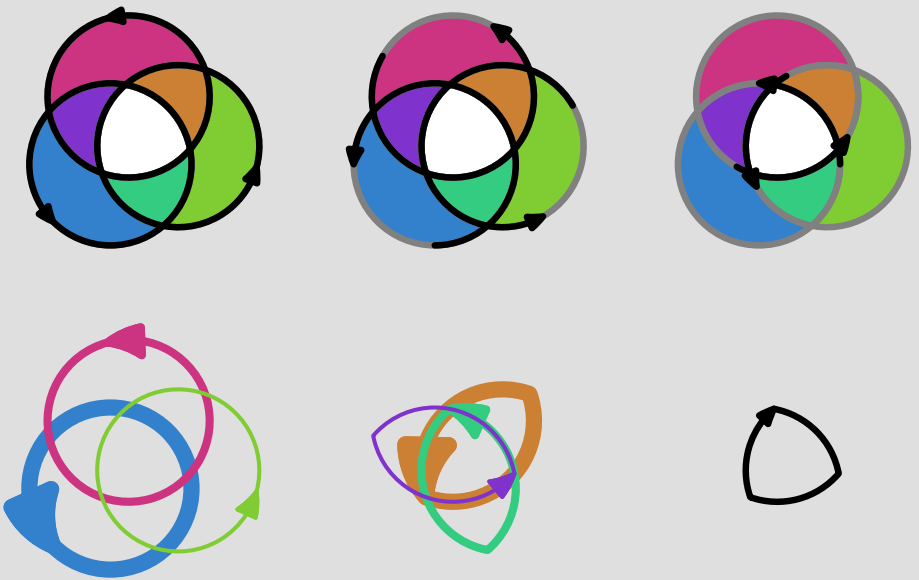
Here, we first fill the primary circles, next we fill the secondary ones. These also cover the center, which is why finally we fill the center with white.



The circle uses the following colors:

```
colorcircle(4cm,(.2,.5,.8),(.8,.2,.5),(.5,.8,.2));
```

The next graphic demonstrates how the subpaths look that build the shapes.



We did not mention what the push and pop commands are responsible for. Scaling the current picture is well defined as long as we deal with one graphic. However, if the current picture already has some content, this content is also scaled. The push and pop commands let us add content to the current picture as well as manipulating the picture as a whole without any side effects. The final result is put on top of the already drawn content. Instead of the sequence:

```
pushcurrentpicture ;
  ...
  currentpicture := currentpicture ... transformations ... ;
```

```
popcurrentpicture ;
```

you can say:

```
pushcurrentpicture ;
  ...
popcurrentpicture ... transformations ... ;
```

Both are equivalent to:

```
draw image ( ... ) ... transformations ... ;
```

For larger sequences of commands, the push–pop alternative gives a bit more more readable code.

13.5 Fool yourself

When doing a literature search on the human perception of black–white edges, I ran into several articles with graphics that I remember having seen before in books on psychology, physiology and/or ergonomics. One of the articles was by Edward H. Adelson of MIT and we will use a few of his example graphics in our exploration to what extend METAPOST can be of help in those disciplines. Since such graphics normally occur in typeset documents, we will define them in the document source.

Unless you belong to the happy few whose visual capabilities are not distorted by neural optimizations, in **figure 13.2** the gray rectangles at the left look lighter than those on the right. Alas, you can fool yourself, but METAPOST does not cheat. This graphic, referred to as White's illusion, is defined as follows.

```
\startbuffer
interim linecap := butt ; numeric u ; u := 1cm ;
```

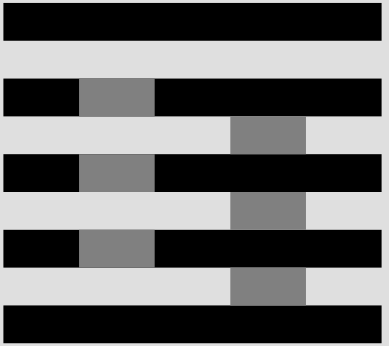


Figure 13.2 White's illusion.

```
pickup pencircle scaled .5u ;
for i=1u step u until 5u :
  draw (0,i) -- (5u,i) ;
endfor ;
for i=2u step u until 4u :
  draw   (u,i) -- (2u,i)                    withcolor .5white ;
  draw ((3u,i) -- (4u,i)) shifted (0,-.5u) withcolor .5white ;
endfor ;
\stopbuffer
```

Watch how we include the code directly. We have packaged this graphic in a buffer which we include as a floating figure.

```
\placefigure
```

```
[here][fig:tricked 1]
{White's illusion.}
{\processMPbuffer}
```

When passed to METAPOST, this code is encapsulated in its `beginfig` and `endfig` macros and thereby grouped. But any change to a variable that is not explicitly saved, migrates to the outer level. In order to prevent all successive graphics to have butt'd linecaps, we have to change this line characteristic locally. Because `linecap` is defined as an internal variable, we have to use `interim` to overload its value. Because `u` is a rather commonly used scratch variable, we don't save its value.

Watch how we use `u` as the loop step. In spite of what your eyes tell you, this graphic only has two explicit color directives, both being 50% black. In the next example we will use some real colors.

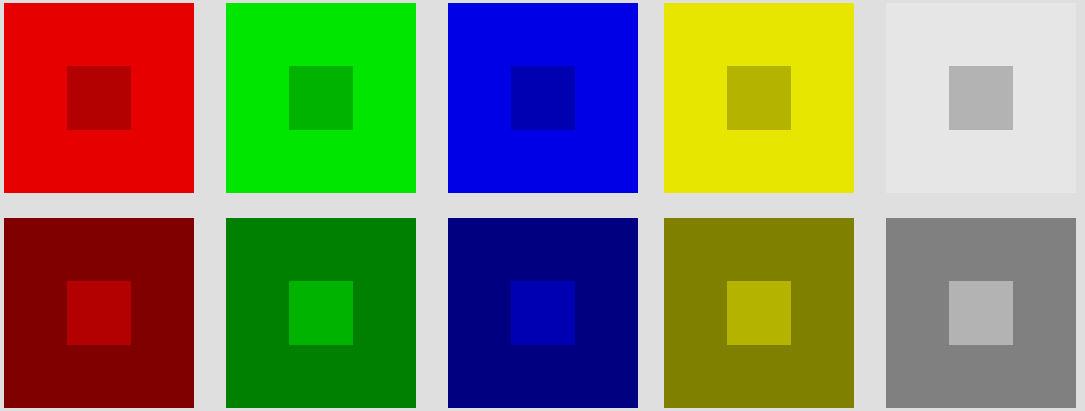


Figure 13.3 The simultaneous contrast effect.

In **figure 13.3** the small squares in the center of each colored pair of big squares have the same shade, but the way we perceive them are influenced by their surroundings. Both sets of squares are defined using usable graphics. The top squares are defined as:

```
\startuseMPgraphic{second}
  \includeMPgraphic{first}
  fill fullsquare scaled size  withcolor topshade ;
  fill fullsquare scaled delta withcolor centershade ;
\stopuseMPgraphic
```

and the bottom squares are coded as:

```
\startuseMPgraphic{third}
  \includeMPgraphic{first}
  fill fullsquare scaled size  withcolor bottomshade ;
  fill fullsquare scaled delta withcolor centershade ;
\stopuseMPgraphic
```

Because both graphics share code, we have defined that code as a separate graphic, that we include. The only point of interest in this definition is the fact that we let METAPOST interpolate between the two colors using `.5[ ]`.

```
\startuseMPgraphic{first}
  numeric size, delta ;
  size := 2.5cm ; delta := size/3 ;
  color mainshade, topshade, bottomshade, centershade ;
  mainshade := \MPcolor{funcolor} ;
```

```
    topshade := .9mainshade ; bottomshade := .5mainshade ;
    centershade := .5[topshade,bottomshade] ;
  \stopuseMPgraphic
```

The color `funcolor` is provided by CONTEXT, and since we want to use this graphic with different colors, this kind of mapping is quite convenient. The bunch of graphics is packaged in a combination with empty captions. Note how we set the color before we include the graphic.

```
\startcombination[5*2]
  {\definecolor[funcolor][red]   \useMPgraphic{second}} {}
  {\definecolor[funcolor][green] \useMPgraphic{second}} {}
  {\definecolor[funcolor][blue]  \useMPgraphic{second}} {}
  {\definecolor[funcolor][yellow]\useMPgraphic{second}} {}
  {\definecolor[funcolor][white] \useMPgraphic{second}} {}
  {\definecolor[funcolor][red]   \useMPgraphic{third}}  {}
  {\definecolor[funcolor][green] \useMPgraphic{third}}  {}
  {\definecolor[funcolor][blue]  \useMPgraphic{third}}  {}
  {\definecolor[funcolor][yellow]\useMPgraphic{third}}  {}
  {\definecolor[funcolor][white] \useMPgraphic{third}}  {}
\stopcombination
```

We use a similar arrangement for the following graphic, where we have replaced the definitions of `first`, `second` and `third` by new definitions.

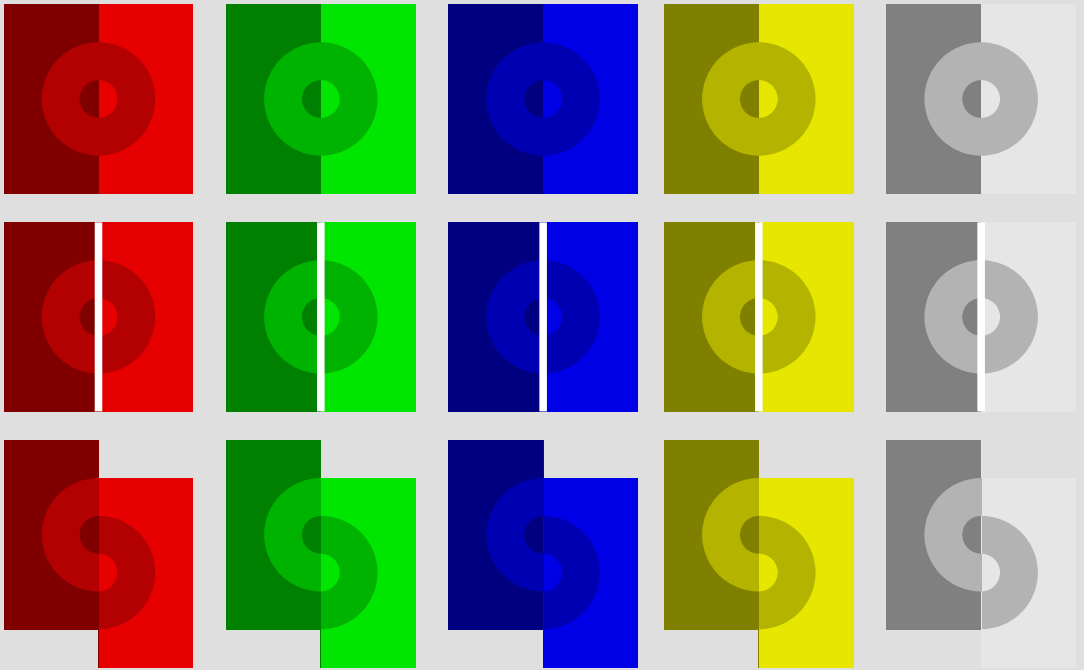


Figure 13.4 Koffka's examples of manipulating contrast by changing the spatial configuration.

The definition of the first row of **figure 13.4** is used in the second and third and therefore is the most complicated. We use quite some scratch variables to reach a high level of abstraction. The `xyscaled` operator is a METAFUN macro.

```
\startuseMPgraphic{first}
  numeric height, width, radius, gap ; gap := 1mm ;
  height = 2.5cm ; width := height/2 ; radius := height/2.5 ;
  color mainshade, leftshade, rightshade, centershade ;
  mainshade := \MPcolor{funcolor} ;
  leftshade := .9mainshade ; rightshade := .5mainshade ;
  centershade := .5[leftshade,rightshade] ;
  fill unitsquare xyscaled ( width,height) withcolor leftshade ;
  fill unitsquare xyscaled (-width,height) withcolor rightshade ;
  draw (fullcircle scaled radius) shifted (0,height/2)
    withpen pencircle scaled (radius/2) withcolor centershade ;
\stopuseMPgraphic
```

The graphics of the second row extend those of the first by drawing a white line through the middle. In this example setting the linecap is not really needed, because rounded top and bottoms in white are invisible and the part that extends beyond the points does not count in calculating the bounding box.

```
\startuseMPgraphic{second}
  \includeMPgraphic{first}
  interim linecap := butt ; pickup pencircle scaled gap ;
  draw (0,0) -- (0,height) withcolor white ;
\stopuseMPgraphic
```

The third row graphics again extend the first graphic. First we copy the picture constructed so far. Watch the double assignment. Next we clip the pictures in half, and shift the right half down over the width of the circle.

```
\startuseMPgraphic{third}
```

```
  \includeMPgraphic{first}
  picture p, q ; p := q := currentpicture ;
  clip p to unitsquare xscaled  width yscaled height ;
  clip q to unitsquare xscaled -width yscaled height ;
  currentpicture := p ;
  addto currentpicture also q shifted (0,radius/2) ;
\stopuseMPgraphic
```

13.6 Growing graphics

Although METAPOST is not really suited as a simulation engine, it is possible to build graphics that are built and displayed incrementally with a sequence of mouse clicks. The following example is the result of an email discussion David Arnold and the author had while METAFUN evolved.

Instead of defining the graphics in a separate METAPOST file, we will incorporate them in the document source in which they are used. We can use several methods.

1. Define macros and figures in a separate file and include the graphics as external graphics.
2. Define everything in the document source as usable graphics and include the graphics using `\useMPgraphic`.
3. Package the graphic components in buffers and paste those together as graphics that can be processed at run time.

The first method is the most independent one, which has its advantages if we want to use the graphics in other applications too. The second method works well in graphics where parts of the definitions change between invocations of the graphic. This method follows the template:

```
\startuseMPgraphic{whatever}
 ...
\stopuseMPgraphic

\startuseMPgraphic{result}
  ...
  \includeMPgraphic{whatever}
  ...
\stopuseMPgraphic

\useMPgraphic{result}
```

The disadvantage of this method is that it cannot be combined with `btex–etex` since it is nearly impossible to determine when, how, and to what extent the content of a graphic should be expanded before writing it to the temporary METAPOST file.

Therefore, we will demonstrate how buffers can be used. This third method closely parallels the first way of defining graphics. A nice side effect is that we can easily typeset these buffers verbatim, which we did to typeset this document.

We are going to do a classic compass and straightedge construction, the bisection of a line segment joining two arbitrary points. We will construct five graphics, where each one displays one step of the construction. We will embed each graphic in a start–stop command. Later we will see the advantage of this strategy.

```
\startbuffer[a]
def start_everything = enddef ;
def stop_everything  = enddef ;
\stopbuffer
```

We are going to draw a few dots, and to force consistency we first define a macro `draw_dot`. The current step will be highlighted in red using `stand_out`.

```
\startbuffer[b]
numeric u, w ; u := .5cm ; w := 1pt ;

pickup pencircle scaled w ;

def draw_dot expr p =
  draw p withpen pencircle scaled 3w ;
enddef ;

def stand_out =
  drawoptions(withcolor .625red) ;
enddef ;
\stopbuffer
```

First, we construct the macro that will plot two points A and B and connect them with a line segment.

```
\startbuffer[c]
def draw_basics =
  pair pointA, pointB ; path lineAB ;
  pointA := origin ; pointB := pointA shifted (5u,0) ;
  lineAB := pointA -- pointB ;
  draw lineAB ;
  draw_dot pointA ; label.lft(btex A etex, pointA) ;
  draw_dot pointB ; label.rt (btex B etex, pointB) ;
enddef ;
```

```
\stopbuffer
```

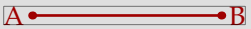
The code in this buffer executes the preceding macros. The `..._everything` commands are still undefined, but later we can use these hooks for special purposes.

```
\startbuffer[1]
start_everything ;
  stand_out ; draw_basics ;
stop_everything ;
\stopbuffer
```

This graphic can now be embedded by the CONTEXT command `\processMPbuffer`. This command, like the ordinary buffer inclusion commands, accepts a list of buffers.

```
\startlinecorrection[blank]
\ruledhbox{\processMPbuffer[a,b,c,1]}
\stoplinecorrection
```

We use `\ruledhbox` to show the tight bounding box of the graphic. The line correction takes care of proper spacing around non textual content, like graphics.[15] This is only needed when the graphic is part of the text flow!



Next, we draw two circles of equal radius, one centered at point A, the other at point B.

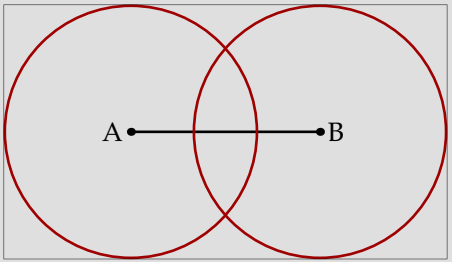
[15] These spacing commands try to get the spacing around the content visually compatible, and take the height and depth of the preceding and following text into account.

```
\startbuffer[d]
def draw_circles =
  path circleA, circleB ; numeric radius, distance ;
  distance := (xpart pointB) - (xpart pointA) ;
  radius := 2/3 * distance ;
  circleA := fullcircle scaled (2*radius) ;
  circleB := circleA shifted pointB ;
  draw circleA ;
  draw circleB ;
enddef ;
\stopbuffer
```

As you can see, we move down the `stand_out` macro so that only the additions are colored red.

```
\startbuffer[2]
start_everything ;
  draw_basics ; stand_out ; draw_circles ;
stop_everything ;
\stopbuffer
```

We now use `\processMPbuffer[a,b,c,d,2]` to include the latest step.



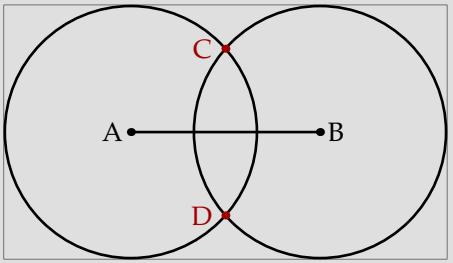
The next step in the construction of the perpendicular bisector requires that we find and label the points of intersection of the two circles centered at points A and B. The intersection points are calculated as follows. Watch the `reverse` operation, which makes sure that we get the second intersection point.

```
\startbuffer[e]
def draw_intersection =
  pair pointC, pointD ;
  pointC := circleA intersectionpoint circleB ;
  pointD := (reverse circleA) intersectionpoint (reverse circleB) ;
  draw_dot pointC ; label.lft(btex C etex, pointC shifted (-2w,0)) ;
  draw_dot pointD ; label.lft(btex D etex, pointD shifted (-2w,0)) ;
enddef ;
\stopbuffer
```

In placing the label, we must make sure that the text runs free of the lines and curves. Again, move the `stand_out` macro just prior to `draw_intersection` macro, so that this step is highlighted in the drawing color, while prior steps are drawn in the default color (in this case black).

```
\startbuffer[3]
```

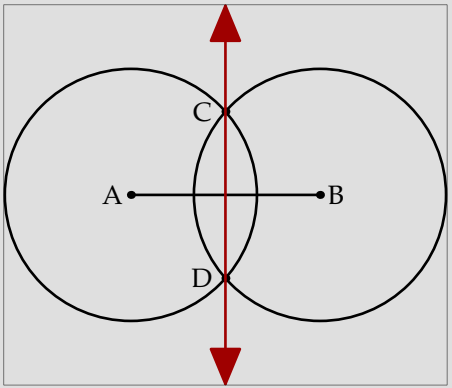
```
start_everything ;
  draw_basics ; draw_circles ; stand_out ; draw_intersection ;
stop_everything ;
\stopbuffer
```



The line drawn through points C and D will be the perpendicular bisector of the line segment connecting points A and B. In the next step we will draw a line using the plain METAPOST `drawdblarrow` macro that draws arrowheads at each end of a path.

```
\startbuffer[f]
def draw_bisector =
  path lineCD ;
  lineCD := origin -- origin shifted (2*distance,0) ;
  lineCD := lineCD rotated 90 shifted 0.5[pointA,pointB] ;
  lineCD := lineCD shifted (0,-distance) ;
  drawdblarrow lineCD ;
enddef ;
\stopbuffer
```

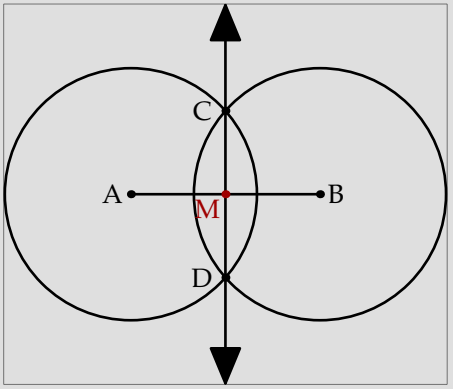
```
\startbuffer[4]
start_everything ;
  draw_basics ; draw_circles ; draw_intersection ; stand_out ;
  draw_bisector ;
stop_everything ;
\stopbuffer
```



The following code draws the intersection of line $C - D$ and line segment $A - B$, which can be shown to be the midpoint of segment $A - B$.

```
\startbuffer[g]
def draw_midpoint =
  pair pointM ;
  pointM := lineCD intersectionpoint lineAB ;
```

```
  draw_dot pointM ; label.llft(btex M etex, pointM) ;
enddef ;
\stopbuffer

\startbuffer[5]
start_everything ;
  draw_basics ; draw_circles ; draw_intersection ; draw_bisector ;
  stand_out ; draw_midpoint ;
stop_everything ;
\stopbuffer
```



As long as we place the graphics as individual insertions in our document, everything is fine. However, if we wish to place them all at once, or as we shall see later, place them on top of one another in a fieldstack, it makes sense to give them all the same bounding box. We can do this by completing the `start_everything` and `stop_everything` commands.

```
\startbuffer[a]
def start_everything =
  path bb ;
  draw_basics ;
  draw_circles ;
  draw_intersection ;
  draw_bisector ;
  draw_midpoint ;
  bb := boundingbox currentpicture ;
  currentpicture := nullpicture ;
enddef ;

def stop_everything =
  setbounds currentpicture to bb ;
enddef ;
\stopbuffer
```

In **figure 13.5** we demonstrate the effect of this redefinition. For this purpose we scale down the graphic to a comfortable 40%, of course by using an additional buffer. We also visualize the bounding box.

```
\startbuffer[h]
def stop_everything =
  setbounds currentpicture to bb ;
  draw bb withpen pencircle scaled .5pt withcolor .625yellow ;
  currentpicture := currentpicture scaled .4 ;
enddef ;
\stopbuffer
```

The graphic itself is defined as follows. Watch how we use the default buffer to keep the definitions readable.

```
\startbuffer
\startcombination[5*1]
  {\processMPbuffer[a,b,c,h,d,e,f,g,1]} {step 1}
  {\processMPbuffer[a,b,c,h,d,e,f,g,2]} {step 2}
  {\processMPbuffer[a,b,c,h,d,e,f,g,3]} {step 3}
  {\processMPbuffer[a,b,c,h,d,e,f,g,4]} {step 4}
  {\processMPbuffer[a,b,c,h,d,e,f,g,5]} {step 5}
\stopcombination
\stopbuffer

\placefigure
  [here][fig:1 till 5]
  {The five graphics, each with the same bounding box.}
  {\getbuffer}
```

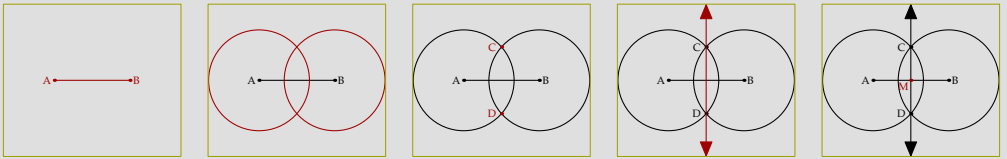


Figure 13.5 The five graphics, each with the same bounding box.

As the original purpose of these graphics was not to show them side by side, but to present them as field stack in a document to be viewed at the computer screen. For this purpose we have to define the graphics as symbols.

```
\definesymbol[step 1][{\processMPbuffer[a,b,c,d,e,f,g,1]}]
\definesymbol[step 2][{\processMPbuffer[a,b,c,d,e,f,g,2]}]
\definesymbol[step 3][{\processMPbuffer[a,b,c,d,e,f,g,3]}]
\definesymbol[step 4][{\processMPbuffer[a,b,c,d,e,f,g,4]}]
\definesymbol[step 5][{\processMPbuffer[a,b,c,d,e,f,g,5]}]
```

A field stack is a sequence of overlayed graphics. We will arrange these to cycle manually, with clicks of the mouse, through the sequence of graphs depicting the construction of the midpoint of segment $A - B$. So, in fact we are dealing with a manual simulation. The definition of such a stack is as follows:

```
\definefieldstack
  [midpoint construction]
  [step 1, step 2, step 3, step 4, step 5]
  [frame=on,offset=3pt,framecolor=darkyellow,rulethickness=1pt]
```

The first argument is to be a unique identifier, the second argument takes a list of symbols, while the third argument accepts settings. More on this command can be found in the CONTEXT manuals.

The stack is shown as **figure 13.6**. Its caption provides a button, which enables the reader to cycle through the stack. We call this a stack because the graphics are positioned on top of each other. Only one of them is visible at any time.

```
\placefigure
  [here][fig:steps]
  {Bisecting a line segment with compass and straightedge? Just
   click \goto {here} [JS(Walk_Field{midpoint construction})] to
   walk through the construction! (This stack is only visible
   in a \PDF\ viewer that supports widgets.)}
```

```
{\fieldstack[midpoint construction]}
```

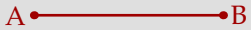


Figure 13.6 Bisecting a line segment with compass and straightedge? Just click **here** to walk through the construction! (This stack is only visible in a PDF viewer that supports widgets.)

At the start of this section, we mentioned three methods. When we use the first method of putting all the graphics in an external METAPOST file, the following framework suits. We assume that the file is called `step.mp` and that it is kept by the user along with his document source. We start with the definitions of the graphic steps. These are the same as the ones shown previously.

```
def draw_basics       = ... enddef ;
def draw_circles      = ... enddef ;
def draw_intersection = ... enddef ;
def draw_bisector     = ... enddef ;
```

```
def draw_midpoint     = ... enddef ;
def stand_out         = ... enddef ;
```

We can safe some code by letting the `..._everything` take care of the `beginfig` and `endfig` macros.

```
def start_everything (expr n) = beginfig(n) ; ... enddef ;
def stop_everything           =    ... ; endfig ; enddef ;
```

The five graphics now become:

```
start_everything (1) ;
  stand_out ; draw_basics ;
stop_everything ;

start_everything (2) ;
  draw_basics ; stand_out ; draw_circles ;
stop_everything ;

start_everything (3) ;
  draw_basics ; draw_circles ; stand_out ; draw_intersection ;
stop_everything ;

start_everything (4) ;
  draw_basics ; draw_circles ; draw_intersection ; stand_out ;
  draw_bisector ;
stop_everything ;

start_everything (5) ;
  draw_basics ; draw_circles ; draw_intersection ; draw_bisector ;
```

```
  stand_out ; draw_midpoint ;
stop_everything ;
```

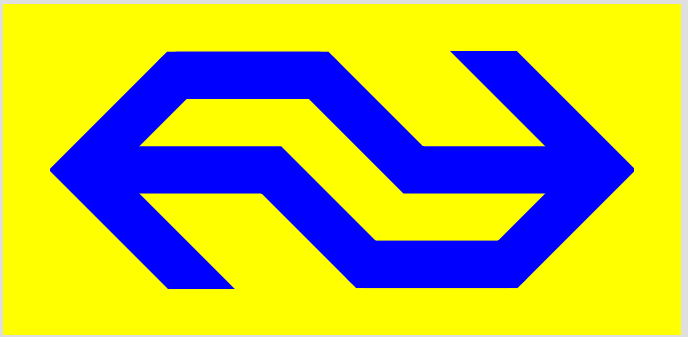
The definitions of the symbols now refer to an external figure.

```
\definesymbol[step 1][{\externalfigure[step.1]}]
\definesymbol[step 2][{\externalfigure[step.2]}]
\definesymbol[step 3][{\externalfigure[step.3]}]
\definesymbol[step 4][{\externalfigure[step.4]}]
\definesymbol[step 5][{\externalfigure[step.5]}]
```

Which method is used, depends on the way the graphics are used. In this example we wanted to change the definition of `...everything`, so here the third method was quite useful.

13.7 Simple Logos

Many company logos earn their beauty from their simplicity. One of the logos that most Dutch people have imprinted in their mind is that of the Dutch Railway Company (NS). An interesting feature of this logo is that, although it is widely known, drawing it on a piece of paper from mind is a task that many people fail.



This logo makes a good candidate for demonstrating a few fine points of drawing graphics, like using linear equations, setting line drawing characteristics, clipping and manipulating bounding boxes.

The implementation below is quite certainly not according to the official specifications, but it can nevertheless serve as an example of defining such logos.

As always, we need to determine the dimensions first. Here, both the height and line width depend on the width of the graphic.

Instead of calculating the blue shape such that it will be a filled outline, we will draw the logo shape using line segments. This is why we need the `line` parameter.

```
numeric width  ; width  = 3cm ;
numeric height ; height = width/2 ;
numeric line   ; line   = height/4 ;
```

We want sharp corners which can be achieved by setting `linejoin` to `mitered`.

```
linejoin := mitered ; pickup pencircle scaled line ;
```

The colors are rather primary blue and yellow. At the time of writing this manual, Dutch trains are still painted yellow, so we will use that shade as background color.

```
color nsblue   ; nsblue   := (0,0,1) ;
color nsyellow ; nsyellow := (1,1,0) ;
```

We will now describe the main curves. Although these expressions are not that advanced, they demonstrate that we can express relationships instead of using assignments.

```
z1 = (0, height/2) ;
z2 = (width/2-height/4, y1) ;
z3 = (width/2+height/4, y4) ;
z4 = (width, 0) ;

path p ; p := z1--z2--z3--z4 ;
```

Although it is accepted to consider `z` to be a variable, it is in fact a `vardef` macro, that expands into a pair `(x,y)`. This means that the previous definitions internally become:

```
(x1,y1) = (0, height/2) ;
(x2,y2) = (width/2-height/4, y1) ;
(x3,y3) = (width/2+height/4, y4) ;
(x4,y4) = (width, 0) ;
```

These 8 relations can be solved by METAPOST, since all dependencies are known.

```
x1 = 0                ; y1 = height/2 ;
```

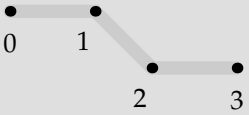
```
x2 = width/2-height/4 ; y2 = y1       ;
x3 = width/2+height/4 ; y3 = y4       ;
x4 = width            ; y4 = 0        ;
```

Since we express the variables x and y in terms of relations, we cannot reuse them, because that would mean that inconsistent relations occur. So, the following lines will lead to an error message:

```
z1 = (10,20) ; z1 = (30,50) ;
```

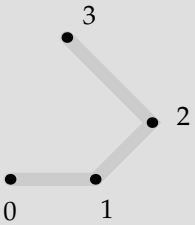
For similar reasons, we may not assign a value (using :=) to such a z variable. Within a METAPOST figure, z variables are automatically saved, which means that they can be reused for each figure.

So far, we have defined the following segment of the logo.

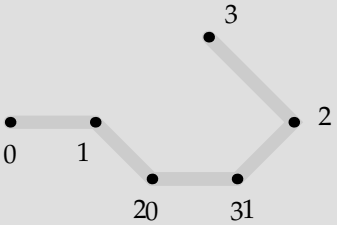


The next expressions are used to define the second segment. The third expression determines z7 to be positioned on the line z5--z6, where we extend this line by 50%.

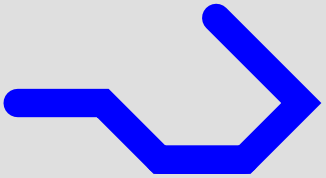
```
z5 = (x4+height/2, y1) ;
z6 = (x4, 2y1) ;
z7 = 1.5[z5,z6] ;

path q ; q := z3--z4--z5--z7 ;
```



If we combine these two segments, we get:



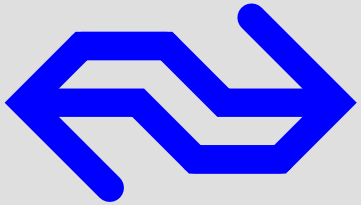
However, when we draw them using the right linewidth and color, you will notice that we're not yet done:



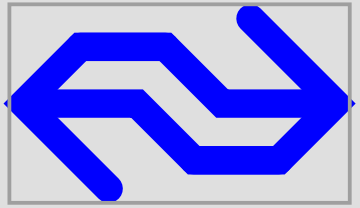
The second curve is similar to the first one, but rotated over 180 degrees.

```
addto currentpicture also currentpicture
```

```
rotatedaround (.5[z2,z3],180) shifted (height/4,height/2) ;
```



In order to get the sharp edges, we need to clip off part of the curves and at first sight, we may consider using a scaled bounding box. However, when we show the natural bounding box, you will notice that a more complicated bit of calculations is needed.



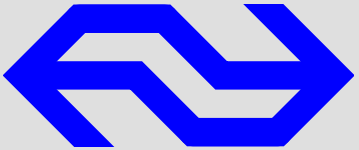
The right clip path is calculated using the following expressions. Watch how we use `rt` and `top` to correct for the linewidth.

```
numeric d, lx, ly, ux, uy ; d = line/2 ;
lx = -3d - d/3  ; ly = -d ; ux = rt x5 + d/3 ; uy = top y6 ;
path r ; r := (lx,ly)--(ux,ly)--(ux,uy)--(lx,uy)--cycle;
```

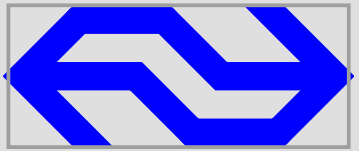
The clipping path is applied by saying:

```
clip currentpicture to r ;
```

The result is quite acceptable:

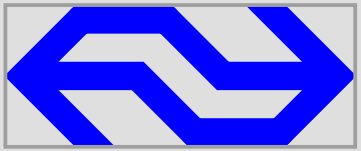


But, if you watch closely to how this graphic extends into to left margin of this document, you will see that the bounding box is not yet right.

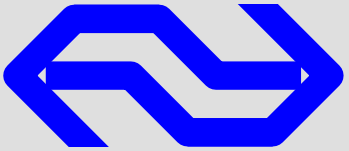


```
setbounds currentpicture to r ;
```

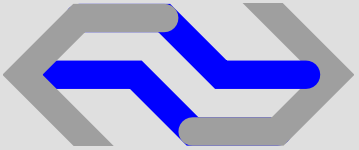
We use the same path `r` to correct the bounding box.



There are a few subtle points involved, like setting the `linejoin` variable. If we had not set it to `mitered`, we would have got round corners. We don't set the `linecap`, because a flat cap would not extend far enough into the touching curve and would have left a small hole. The next example shows what happens if we set these variables to the wrong values:



In fact we misuse the fact that both curves overlay each other.



The complete logo definition is a bit more extensive because we also want to add a background. Because we need to clip the blue foreground graphic, we must temporarily store it when we fill the background.

```
numeric width, height, line, delta ;
width = 5cm ; height = width/2 ; line = height/4 ; delta = line ;

linejoin := mitered ; pickup pencircle scaled line ;

color nsblue   ; nsblue   := (0,0,1) ;
color nsyellow ; nsyellow := (1,1,0) ;

z1 = (0, height/2) ;
```

```
z2 = (width/2-height/4, y1) ;
z3 = (width/2+height/4, y4) ;
z4 = (width, 0) ;

z5 = (x4+height/2, y1) ;
z6 = (x4, 2y1) ;
z7 = 1.5[z5,z6] ;

path p ; p := z1--z2--z3--z4 ; path q ; q := z3--z4--z5--z7 ;

numeric d, lx, ly, ux, uy ; d = line/2 ;

lx = -3d - d/3  ; ly = -d ; ux = rt x5 + d/3 ; uy = top y6 ;

path r ; r := (lx,ly)--(ux,ly)--(ux,uy)--(lx,uy)--cycle;

lx := lx-delta ; ly := ly-delta ; ux := ux+delta ; uy := uy+delta ;

path s ; s := (lx,ly)--(ux,ly)--(ux,uy)--(lx,uy)--cycle;

draw p withcolor nsblue ; draw q withcolor nsblue ;

addto currentpicture also currentpicture
  rotatedaround (.5[z2,z3],180) shifted (height/4,height/2) ;

picture savedpicture ; savedpicture := currentpicture ;

clip currentpicture to r ;
setbounds currentpicture to r ;

savedpicture := currentpicture ; currentpicture := nullpicture ;

fill s withcolor nsyellow ;
```

```
addto currentpicture also savedpicture ;
```

For practical use it makes sense to package this definition in a macro to which we pass the dimensions.

13.8 Music sheets

The next example demonstrates quite some features. Imagine that we want to make us a couple of sheets so that we can write a musical masterpiece. Let's also forget that TEX can draw lines, which means that somehow we need to use METAPOST.

Drawing a bar is not that complicated as the following code demonstrates.

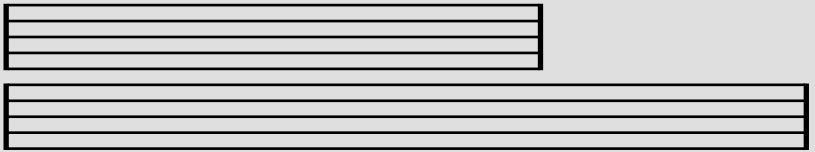
```
\startusableMPgraphic{bar}
  vardef MusicBar (expr width, gap, linewidth, barwidth) =
    image
      ( interim linecap := butt ;
        for i=1 upto 5 :
          draw ((0,0)--(width,0)) shifted (0,(i-1)*gap)
            withpen pencircle scaled linewidth ;
        endfor ;
        for i=llcorner currentpicture -- ulcorner currentpicture ,
              lrcorner currentpicture -- urcorner currentpicture :
          draw i withpen pencircle scaled barwidth ;
        endfor ; )
  enddef ;
\stopusableMPgraphic
```

We can define the sidebars a bit more efficient using two predefined subpaths:

```
for i=leftboundary currentpicture, rightboundary currentpicture :
```

We define a macro `MusicBar` that takes four arguments. The first two determine the dimensions, the last two concern the line widths. Now watch how we can use this macro:

```
\includeMPgraphic{bar} ;
draw MusicBar (200pt, 6pt, 1pt, 2pt) ;
draw MusicBar (300pt, 6pt, 1pt, 2pt) shifted (0,-30pt) ;
```



As you can see in this example, the bar is a picture that can be transformed (shifted in our case). However, a close look at the macro teaches us that it does a couple of draws too. This is possible because we wrap the whole in an image using the `image` macro. This macro temporary saves the current picture, and at the end puts the old `currentpicture` under the new one.

We wrap the whole in a `vardef`. This means that the image is returned as if it was a variable. Actually, the last thing in a `vardef` should be a proper return value, in our case a picture. This also means that we may not end the `vardef` with a semi colon. So, when the content of the `vardef` is expanded, we get something

```
draw some_picture ... ;
```

Because we are still drawing something, we can add transform directives and set attributes, like the color.

The second `for` loop demonstrates two nice features. Instead of repeating the draw operation by copying code, we apply it to a list, in our case a list of paths. This list contains two simple line paths. Because an `image` starts with a fresh `currentpicture`, we can safely use the bounding box data to determine the height of the line.

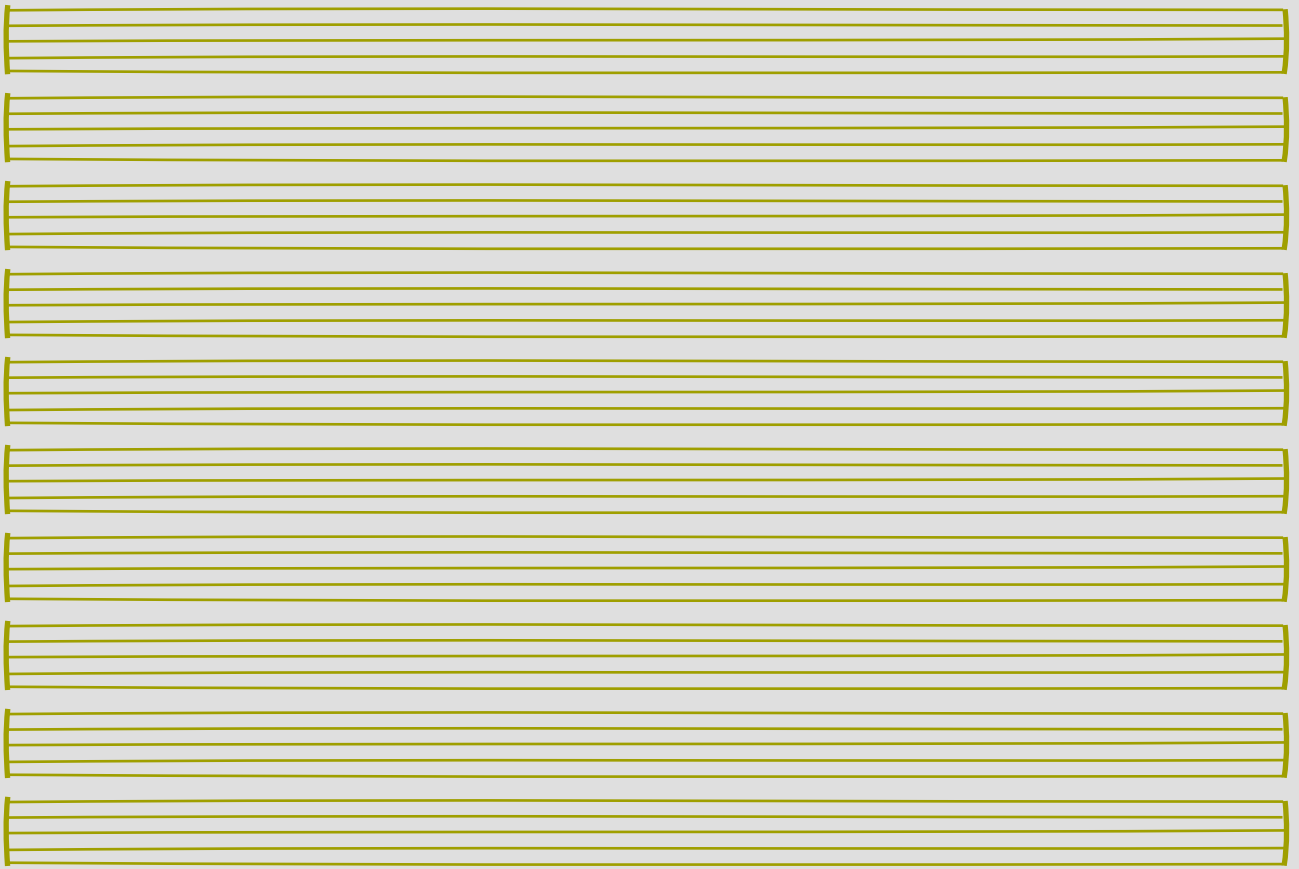
The next step in producing the sheets of paper is to put several bars on a page, preferable with the width of the current text. This time we will use a reusable graphic, because each bar is the same.

```
\startreusableMPgraphic{bars}
  \includeMPgraphic{bar} ;
  draw MusicBar (TextWidth, 6pt, 1pt, 2pt) withcolor .625yellow ;
\stopreusableMPgraphic
```

Instead of going trough the trouble of letting METAPOST calculate the positions of the bars, we will use TEX. We put 12 bars on a page and let TEX take care of the inter–bar spacing. Because we only want stretchable space between bars, called glue in TEX, we need to remove the last added glue.

```
\startstandardmakeup[doublesided=no,page=]
  \dorecurse{10}{\reuseMPgraphic{bars}\vfill}\removelastskip
\stopstandardmakeup
```

It may add to the atmosphere of handy–work if you slightly randomize the lines. We leave it up to the reader to figure out how the code should be changed to accomplish this.



The complete result is shown on the next page.

13.9 The euro symbol

When Patrick Gundlach posted a nice METAPOST version of the euro symbol to the CONTEXT discussion list, he added the comment "The official construction is ambiguous: how thick are the horizontal bars? How much do they stick out to the left? Is this thing a circle or what? Are the angles on the left side of the bars the same as the one on the right side? . . . " The alternative below is probably not as official as his, but permits a finetuning. You are warned: whatever you try, the euro *is* and *will remain* an ugly symbol.

We use a couple of global variables to control the euro shape within reasonable bounds. Then we define two circles. Next we define a vertical line that we use in a couple of cut and paste operations. Watch how the top left point of the outer circle determines the slant of the line that we use to slice the vertical bars.

```
boolean trace_euro ; trace_euro := false ;

vardef euro_symbol = image ( % begin_of_euro

if unknown euro_radius   : euro_radius   := 2cm             ; fi ;
if unknown euro_width    : euro_width    := 3euro_radius/16 ; fi ;
if unknown euro_r_offset : euro_r_offset := euro_width      ; fi ;
if unknown euro_l_offset : euro_l_offset := euro_radius/32  ; fi ;
if unknown euro_l_shift  : euro_l_shift  := euro_r_offset   ; fi ;
if unknown euro_v_delta  : euro_v_delta  := euro_width/4    ; fi ;

save
  outer_circle, inner_circle, hor_bar,
  right_line, right_slant, top_slant,  bot_slant,
```

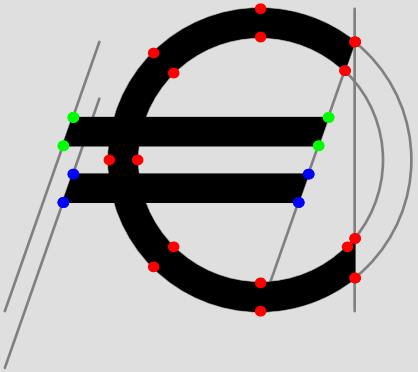
```
  euro_circle, euro_topbar, euro_botbar ;

path
  outer_circle, inner_circle, hor_bar,
  right_line, right_slant, top_slant, bot_slant,
  euro_circle, euro_topbar, euro_botbar ;

outer_circle := fullcircle scaled euro_radius ;
inner_circle := fullcircle scaled (euro_radius-euro_width) ;

if trace_euro : for i = outer_circle, inner_circle :
  draw i withpen pencircle scaled 1pt withcolor .5white ;
endfor ; fi ;

right_line :=
  (lrcorner outer_circle -- urcorner outer_circle)
  shifted (-euro_r_offset,0) ;

outer_circle := outer_circle cutbefore right_line ;

right_slant :=
  point 0 of outer_circle
  -- origin shifted (0,ypart lrcorner outer_circle) ;

euro_circle := buildcycle(outer_circle, right_line,
  reverse inner_circle, reverse right_slant) ;

hor_bar := (-euro_radius,0) -- (euro_radius,0) ;

top_slant :=
  right_slant shifted (-euro_radius+euro_r_offset-euro_l_offset,0) ;
```

```
bot_slant :=
  top_slant shifted (0,-euro_l_shift) ;

if trace_euro : for i = right_line, right_slant, top_slant, bot_slant :
  draw i withpen pencircle scaled 1pt withcolor .5white ;
endfor ; fi ;

euro_topbar := buildcycle
  (top_slant, hor_bar shifted (0, euro_v_delta),
   right_slant, hor_bar shifted (0, euro_v_delta+euro_width/2)) ;

euro_botbar := buildcycle
  (bot_slant, hor_bar shifted (0,-euro_v_delta),
   right_slant, hor_bar shifted (0,-euro_v_delta-euro_width/2)) ;

for i = euro_circle, euro_topbar, euro_botbar :
  draw i withpen pencircle scaled 0 ;
endfor ;
for i = euro_circle, euro_topbar, euro_botbar :
  fill i withpen pencircle scaled 0 ;
endfor ;

if trace_euro :
  drawpoints euro_circle withcolor red   ;
  drawpoints euro_topbar withcolor green ;
  drawpoints euro_botbar withcolor blue  ;
fi ;

) enddef ; % end_of_euro
```

We only set a parameter when it is not yet set. This has the advantage that we don't have to set them when we change one. This way of manipulating paths (cutting and building) does not always work well because of rounding errors, but here it does work.

```
euro_radius := 4cm ; trace_euro := true ; draw euro_symbol ;
```

For educational purposes, we have added a bit of tracing. When enabled, the euro shows up as:



Of course it would be best to define the euro as one shape, but we won't go though that process right now. By packaging the combined paths in an image, we can conveniently color the euro symbol:

```
draw euro_symbol withcolor .625red ;
```

You may wonder why we both draw and fill the euro, using a pen with zero width. We've done this in order to demonstrate the `redraw` and `refill` macros.

```
redraw currentpicture withpen pencircle scaled 4pt withcolor .625yellow ;
refill currentpicture withcolor .625white ;
setbounds currentpicture to boundingbox currentpicture enlarged 2pt ;
```

13.10 Killing time

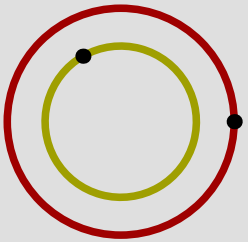
Not seldom TEX users want to use this program and its meta–relatives as general purpose tools, even at the cost of quite some effort or suboptimal results. Imagine that you are under way from our planet to Mars. After a long period of sleep you wake up and start wondering on what track you are. You even start questioning the experts that send you on your way, so you pop open your laptop, launch your editor and start metaposting.

First you need to determine the begin and end points of your journey. For this it is enough to know the relative angle of the paths that both planets follow as well as the path themselves. We assume circular paths.

```
path a ; a := fullcircle scaled 3cm ;
path b ; b := fullcircle scaled 2cm rotated 120 ;

draw a withpen pencircle scaled 1mm withcolor .625red ;
draw b withpen pencircle scaled 1mm withcolor .625yellow ;

draw point 0 of a withpen pencircle scaled 2mm ;
draw point 0 of b withpen pencircle scaled 2mm ;
```

The rotation 120 can be calculated from the relative starting points and time the journey will take. Alternatively we can use the time along the path, but this would be a bit more fuzzy later on.[16]

[16] In case you wonder why METAPOST talks about the time on a path, you now have a cue.



After a bit of playing with drawing paths between the two points, you decide to make a macro. We want to feed the angle between the paths but also the connecting path. So, we have to pass a path, but unfortunately we don't have direct access to the points. By splitting the argument definition we can pass an expression first, and a wildcard argument next.

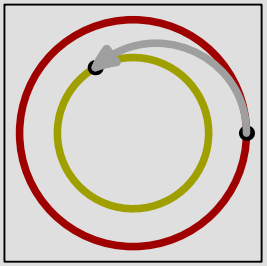
```
\startuseMPgraphic{gamble}
def Gamble (expr rot) (text track) =
  path a ; a := fullcircle scaled 3cm ;
  path b ; b := fullcircle scaled 2cm rotated rot ;

  pair aa ; aa := point 0 of a ;
  pair bb ; bb := point 0 of b ;
  path ab ; ab := track ;

  draw a  withpen pencircle scaled 1mm withcolor .625red ;
  draw b  withpen pencircle scaled 1mm withcolor .625yellow ;

  draw aa withpen pencircle scaled 2mm ;
  draw bb withpen pencircle scaled 2mm ;

  drawarrow ab withpen pencircle scaled 1mm withcolor .625white ;
```

```
    setbounds currentpicture to boundingbox a enlarged 2mm ;
    draw boundingbox currentpicture withpen pencircle scaled .25mm ;
  enddef ;
  \stopuseMPgraphic
```

Because at this distance nobody will bother us with the thickness of the pen and colors, we code them the hard way. We create our own universe by setting a fixed boundingbox.

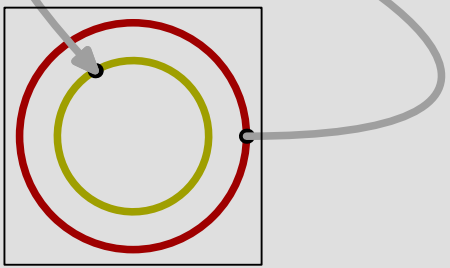
We leave the Earth in the most popular way, straight upwards and after a few cycles, we leave it parallel to the surface. The path drawn reminds much of the trajectories shown in popular magazines.

```
\startMPcode
\includeMPgraphic{gamble} ;
Gamble(120, aa {(0,1)} .. bb) ;
\stopMPcode
```



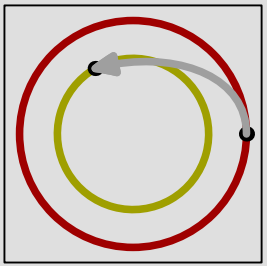
According to METAPOST, when we leave the Earth straight upwards and want a smooth trajectory, we have to pass through outer space.

```
\startMPcode
\includeMPgraphic{gamble} ;
Gamble(120,aa {(1,0)} .. bb) ;
\stopMPcode
```



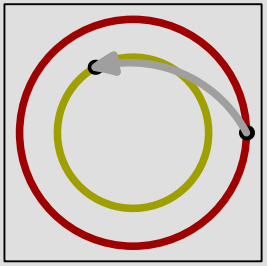
Given that we want a smooth path as well as a short journey, we can best follow Mars' path. Here we face the risk that when we travel slower than Mars does, we have a problem.

```
\startMPcode
\includeMPgraphic{gamble} ;
Gamble(120,aa {dir 90} .. {precontrol 0 of b rotated 90} bb) ;
\stopMPcode
```



We can even travel a shorter path when we leave Earth at the surface that faces the point of arrival.

```
\startMPcode
\includeMPgraphic{gamble} ;
Gamble(120,aa .. {precontrol 0 of b rotated 90} bb) ;
\stopMPcode
```



In the end we decide that although the trajectories look impressive, we will not trust our lives to METAPOST. A beautiful path is not neccessarily a good path. But even then, this macro provides a nice way to experiment with directions, controls and tensions.

14

METAFUN macros

CONTEXT comes with a series of METAPOST modules. In this chapter we will summarize the most important TEX and METAPOST macros. More information can be found in the documentation of the modules.

There are several ways to use the power of METAFUN, either or not using CONTEXT.

1. You can create an independent `mp` file and process it with the METAPOST program or MPTOPDF. Depending on what features you use, the succes of a run depends on the proper set up of the programs that take care of running TEX for `btex`.

2. You can embed the graphic in a `\startMPpage` construct and process it with CONTEXT MKIV (which is the future anyway). In that case you have the full METAFUN functionality available. If for some reason you still want to use MKII, you need to use TEXEXEC as before processing the file, it will do a couple of checks on the file. It will also make sure that the temporary files (`mpt` for `textext`'s and `mpo` for outline fonts) are taken care of too.

3. You can integrate the graphic in the document flow, using buffers, METAPOST code sections, or (reusable) graphic containers. In that case the graphics are processed runtime or between runs. This happens automatically.

Unless you want to write low level CONTEXT code yourself, there is no real reason to look into the modules that deal with METAPOST support. The traditional (partly generic) code is collected in:

`supp-mps.tex`	low level inclusion macros and housekeeping
`supp-mpe.tex`	experimental extensions (like specials)
`supp-pdf.tex`	METAPOST to PDF conversion

Especially the last two can be used in other macro packages. However, in CONTEXT we have regrouped the code (plus more) in other files:

`meta-***.tex`	definition and managing
`mlib-***.tex`	processing and conversion

The last category will certainly grow. Some of these modules are preloaded, others can be loaded using the command `\useMPlibrary`, like

```
\useMPlibrary[clp,txt]
```

for loading the predefined clipping paths and text tricks.

The METAPOST code is organized in files named `mp-****.mp`. The core file is `mp-tool.mp` and this file can comfortably be used in stand–alone graphics. The file `metafun.mp` is used to load the collection of modules into a format. The collection of METAPOST code files will grow in due time, but as long as you use the METAFUN format, you don't have to keep track of the organization of files. Most files relate to subsystems and are loaded automatically, like the files that implement page layout support and flow charts.

Although this document is the main source of information, occasionally the source code of METAFUN, and in many cases the source code of CONTEXT may contain additional information and examples.

A METAPOST syntax

In the METAFONT book as well as the METAPOST manual, you can find the exact specification of the language. Below you find the full METAPOST syntax, to which we add the basic METAFUN extensions. If this page looks too cryptic, you can safely skip to the next chapter.

A.1 Syntax diagrams

The following syntax diagrams are derived from the diagrams in the METAPOST manual. The → represents 'means' and the | symbol stands for 'or'.

The diagrams describe the hard coded METAPOST syntax as well as most of the macros and variables defined in the plain METAPOST format that belongs to the core of the system. They also include some of the more fundamental METAFUN commands.

⟨atom⟩
→ ⟨variable⟩ ⟨argument⟩
| ⟨number or fraction⟩
| ⟨internal variable⟩
| (⟨expression⟩)
| `begingroup` ⟨statement list⟩ ⟨expression⟩ `endgroup`
| ⟨nullary op⟩
| `btex` ⟨typesetting command⟩ `etex`
| ⟨pseudo function⟩

⟨primary⟩
→ ⟨atom⟩
| (⟨numeric expression⟩ , ⟨numeric expression⟩)
| (⟨numeric expression⟩ , ⟨numeric expression⟩ , ⟨numeric expression⟩)
| (⟨numeric expression⟩ , ⟨numeric expression⟩ , ⟨numeric expression⟩ , ⟨numeric expression⟩)
| ⟨of operator⟩ ⟨expression⟩ `of` ⟨primary⟩
| ⟨of operator⟩ ⟨expression⟩ `along` ⟨primary⟩
| ⟨of operator⟩ ⟨expression⟩ `on` ⟨primary⟩
| ⟨unary op⟩ ⟨primary⟩
| `str` ⟨suffix⟩
| `z` ⟨suffix⟩
| ⟨numeric atom⟩ `[` ⟨expression⟩ , ⟨expression⟩ `]`
| ⟨scalar multiplication op⟩ ⟨primary⟩

⟨secondary⟩
→ ⟨primary⟩
| ⟨secondary⟩ ⟨primary binop⟩ ⟨primary⟩
| ⟨secondary⟩ ⟨transformer⟩

⟨tertiary⟩
→ ⟨secondary⟩
| ⟨tertiary⟩ ⟨secondary binop⟩ ⟨secondary⟩

⟨subexpression⟩
→ ⟨tertiary⟩
| ⟨path expression⟩ ⟨path join⟩ ⟨path knot⟩

⟨expression⟩
→ ⟨subexpression⟩
| ⟨expression⟩ ⟨tertiary binop⟩ ⟨tertiary⟩
| ⟨path subexpression⟩ ⟨direction specifier⟩
| ⟨path subexpression⟩ ⟨path join⟩ `cycle`

⟨path knot⟩
→ ⟨tertiary⟩

⟨path join⟩
→ `--`
| ⟨direction specifier⟩ ⟨basic path join⟩ ⟨direction specifier⟩

⟨direction specifier⟩
→ ⟨empty⟩
| `{ curl` ⟨numeric expression⟩ `}`
| `{` ⟨pair expression⟩ `}`
| `{` ⟨numeric expression⟩ `,` ⟨numeric expression⟩ `}`

⟨basic path join⟩
→ `..`
| `...`
| `..` ⟨tension⟩ `..`
| `..` ⟨tension⟩ `..`
| `..` ⟨controls⟩ `..`

⟨tension⟩
→ `tension` ⟨numeric primary⟩
| `tension atleast` ⟨numeric primary⟩
| `tension` ⟨numeric primary⟩ `and` ⟨numeric primary⟩

⟨controls⟩
→ controls ⟨pair primary⟩
| controls ⟨pair primaryiandhpair primary⟩

⟨argument⟩
→ ⟨symbolic token⟩

⟨number or fraction⟩
→ ⟨number⟩ `/` ⟨number⟩
| ⟨number not followed by ' `/` ⟨numberi⟩ ' ⟩

⟨scalar multiplication op⟩
→ `+` | `-`
| ⟨ ' ⟨number or fractioni⟩ ' not followed by ' ⟨add op⟩ ⟨number⟩ ' ⟩

⟨transformer⟩
→ `rotated` ⟨numeric primary⟩
| `scaled` ⟨numeric primary⟩
| `shifted` ⟨pair primary⟩
| `slanted` ⟨numeric primary⟩
| `transformed` ⟨transform primary⟩
| `xscaled` ⟨numeric primary⟩
| `yscaled` ⟨numeric primary⟩
| `zscaled` ⟨pair primary⟩
| `xyscaled` ⟨numeric or pair primary⟩
| `reflectedabout` `(` ⟨pair expression⟩ `,` ⟨pair expressioni⟩ `)`
| `rotatedaround` `(` ⟨pair expression⟩ `,` ⟨numeric expression⟩ `)`
| `xsized` ⟨numeric primary⟩
| `ysized` ⟨numeric primary⟩
| `xysized` ⟨numeric or pair primary⟩
| `enlarged` ⟨numeric or pair primary⟩
| `leftenlarged` ⟨numeric primary⟩
| `rightenlarged` ⟨numeric primary⟩
| `topenlarged` ⟨numeric primary⟩
| `bottomenlarged` ⟨numeric primary⟩
| `randomized` ⟨numeric or pair or color primary⟩
| `cornered` ⟨numeric or pair⟩
| `smoothed` ⟨numeric or pair⟩
| `sqeezed` ⟨numeric or pair primary⟩
| `superellipsed` ⟨numeric primary⟩
| `randomshifted` ⟨numeric or pair primary⟩
| `uncolored` ⟨color primary⟩

| `softened` ⟨numeric or color primary⟩

⟨numeric or pair primary⟩
→ ⟨numeric primary⟩
| ⟨pair primary⟩

⟨numeric or pair or color primary⟩
→ ⟨numeric primary⟩
| ⟨pair primary⟩
| ⟨color primary⟩

⟨numeric or color primary⟩
→ ⟨numeric primary⟩
| ⟨color primary⟩

⟨nullary op⟩
→ `false` | `normaldeviate` | `nullpen` | `nullpicture` | `pencircle` | `true` | `whatever`

⟨unary op⟩
→ ⟨type⟩
| abs | acos | acosh | angle | arclength | ASCII | asin | asinh | atan
| bbox | blackpart | bluepart | bot | bottomboundary | bounded | boundingbox
| ceiling | center | char | clipped | colormodel | condition | cosd | cos | cosh | cot
| cotd | cyanpart | cycle | dashpart | decimal | ddecimal | dddecimal | dir | exp
| filled | floor | fontpart | fontsize | grayed | greenpart | greypart
| hex | innerboundingbox | inv | invcos | inverse | inverted | invsin
| known | leftboundary | length | lft | llcorner | ln | log | lrcorner
| magentapart | makepath | makepen | mexp | mlog | not | oct | odd | outerboundingbox
| pathpart | penpart | redpart | reverse | rightboundary | round | rt
| simplified | sin | sind | sinh | sqr | sqrt | stroked
| tan | tand | textpart | textual | top | topboundary
| ulcorner | uniformdeviate | unitvector | unknown | unspiked | urcorner
| xpart | xxpart | xypart | yellowpart | ypart | yxpart | yypart

⟨type⟩
→ boolean | color | cmykcolor | numeric | pair | path
| pen | picture | rgbcolor | string | transform

⟨primary binop⟩
→ * | / | ** | and
| dotprod | div | infont | mod

⟨secondary binop⟩
→ `+` | `-` | `++` | `+-+` | `or`
| `intersectionpoint` | `intersectiontimes`

⟨tertiary binop⟩
→ `&` | `<` | `<=` | `<>` | `=` | `>` | `>=`
| `cutafter` | `cutbefore` | `cutends`

⟨of operator⟩
→ `arctime` | `direction` | `directiontime` | `directionpoint` | `penoffset` | `point`
| `postcontrol` | `precontrol` | `subpath` | `substring`

⟨variable⟩
→ ⟨tag⟩ ⟨suffix⟩

⟨suffix⟩
→ ⟨empty⟩
| ⟨suffix⟩ ⟨subscript⟩
| ⟨suffix⟩ ⟨tag⟩
| ⟨suffix parameter⟩

⟨subscript⟩
→ ⟨number⟩
| `[` ⟨numeric expression⟩ `]`

⟨internal variable⟩
→ `ahangle` | `ahlength` | `bboxmargin` | `charcode` | `day` | `defaultcolormodel` | `defaultpen`
| `defaultscale` | `hour` | `labeloffset` | `linecap` | `linejoin` | `minute` | `miterlimit`
| `month` | `outputformat` | `outputtemplate` | `pausing` | `prologues` | `showstopping` | `time`
| `tracingoutput` | `tracingcapsules` | `tracingchoices` | `tracingcommands` | `tracingequations`
| `tracinglostchars tracingmacros` | `tracingonline` | `tracingrestores tracingspecs`
| `tracingstats` | `tracingtitles truecorners` | `warningcheck` | `year`
| ⟨symbolic token defined by `newinternal` ⟩

⟨pseudo function⟩
→ `min` (⟨expression list⟩)
| `max` (⟨expression list⟩)
| `incr` (⟨numeric variable⟩)
| `decr` (⟨numeric variable⟩)
| `dashpattern` (⟨on/off list⟩)
| `interpath` (⟨numeric expression⟩ , ⟨path expression⟩ , ⟨path expression⟩)
| `buildcycle` (⟨path expression list⟩)
| `thelabel` ⟨label suffix⟩ (⟨expression⟩ , ⟨pair expression⟩)
| `thefreelabel` (⟨expression⟩ , ⟨pair expression⟩ , ⟨pair expression⟩)
| `anglebetween` (⟨path expression⟩ , ⟨path expression⟩ , ⟨expression⟩)
| `pointarrow` (⟨path expression⟩ , ⟨numeric or pair primary⟩ , ⟨numeric expression⟩ , ⟨numeric expression⟩)
| `leftarrow` (⟨path expression⟩ , ⟨numeric or pair primary⟩ , ⟨numeric expression⟩)
| `centerarrow` (⟨path expression⟩ , ⟨numeric or pair primary⟩ , ⟨numeric expression⟩)
| `rightarrow` (⟨path expression⟩ , ⟨numeric or pair primary⟩ , ⟨numeric expression⟩)
| `paired` (⟨numeric or pair⟩)
| `tripled` (⟨numeric or color⟩)
| `remappedcolor` (⟨color expression⟩)

⟨path expression list⟩
→ ⟨path expression⟩
| ⟨path expression list⟩ , ⟨path expression⟩

⟨on/off list⟩
→ ⟨on/off list⟩ ⟨on/off clause⟩
| ⟨on/off clause⟩

⟨on/off clause⟩
→ `on` ⟨numeric tertiary⟩
| `off` ⟨numeric tertiary⟩

⟨boolean expression⟩ → ⟨expression⟩
⟨cmykcolor expression⟩ → ⟨expression⟩
⟨color expression⟩ → ⟨expression⟩
⟨numeric atom⟩ → ⟨atom⟩
⟨numeric expression⟩ → ⟨expression⟩
⟨numeric primary⟩ → ⟨primary⟩
⟨numeric tertiary⟩ → ⟨tertiary⟩
⟨numeric variable⟩ → ⟨variable⟩ | ⟨internal variable⟩
⟨pair expression⟩ → ⟨expression⟩
⟨pair primary⟩ → ⟨primary⟩
⟨path expression⟩ → ⟨expression⟩
⟨path subexpression⟩ → ⟨subexpression⟩
⟨pen expression⟩ → ⟨expression⟩

⟨picture expression⟩ → ⟨expression⟩
⟨picture variable⟩ → ⟨variable⟩
⟨rgbcolor expression⟩ → ⟨expression⟩
⟨string expression⟩ → ⟨expression⟩
⟨suffix parameter⟩ → ⟨parameter⟩
⟨transform primary⟩ → ⟨primary⟩

⟨program⟩
→ ⟨statement list⟩ `end`

⟨statement list⟩
→ ⟨empty⟩
| ⟨statement list⟩ ; ⟨statement⟩

⟨statement⟩
→ ⟨empty⟩
| ⟨equation⟩
| ⟨assignment⟩
| ⟨declaration⟩
| ⟨macro definition⟩
| ⟨compound⟩
| ⟨pseudo procedure⟩
| ⟨command⟩

⟨compound⟩
→ `begingroup` ⟨statement list⟩ `endgroup`
| `beginfig (` ⟨numeric expression⟩ `) ;` ⟨statement list⟩ ⟨;⟩ `endfig`

⟨equation⟩
→ ⟨expression⟩ `=` ⟨right-hand side⟩

⟨assignment⟩
→ ⟨variable⟩ `:=` ⟨right-hand side⟩
| ⟨internal variable⟩ `:=` ⟨right-hand side⟩

⟨right-and side⟩
→ ⟨expression⟩
| ⟨equation⟩
| ⟨assignment⟩

⟨declaration⟩
→ ⟨type⟩ ⟨declaration list⟩

⟨declaration list⟩
→ ⟨generic variable⟩
| ⟨declaration list⟩ `,` ⟨generic variable⟩

⟨generic variable⟩
→ ⟨Symbolic token⟩ ⟨generic suffix⟩

⟨generic suffix⟩
→ ⟨empty⟩
| ⟨generic suffix⟩ ⟨tag⟩
| ⟨generic suffix⟩ `[]`

⟨macro de nition⟩
→ ⟨macro heading⟩ `=` ⟨replacement text⟩ `enddef`

⟨macro heading⟩
→ `def` ⟨Symbolic token⟩ ⟨delimited part⟩ ⟨undelimited part⟩
| `vardef` ⟨generic variable⟩ ⟨delimited part⟩ ⟨undelimited part⟩
| `vardef` ⟨generic variable⟩ `@#` ⟨delimited part⟩ ⟨undelimited part⟩
| ⟨binary def⟩ ⟨parameter⟩ ⟨symbolic token⟩ ⟨parameter⟩

⟨delimited part⟩
→ ⟨empty⟩
| ⟨delimited part⟩ `(` ⟨parameter type⟩ ⟨parameter tokens⟩ `)`

⟨parameter type⟩
→ `expr`
| `suffix`
| `text`

⟨parameter tokens⟩
→ ⟨parameter⟩
| ⟨parameter tokens⟩ , ⟨parameter⟩

⟨parameter⟩
→ ⟨Symbolic token⟩

⟨undelimited part⟩
→ ⟨empty⟩
| ⟨parameter type⟩ ⟨parameter⟩
| ⟨precedence level⟩ ⟨parameter⟩
| `expr` ⟨parameter⟩ `of` ⟨parameter⟩

⟨precedence level⟩
→ `primary`
| `secondary`
| `tertiary`

⟨binary def⟩
→ ⟨primarydef⟩
| ⟨secondarydef⟩
| ⟨tertiarydef⟩

⟨pseudo procedure⟩
→ `drawoptions` (⟨option list⟩)
| `label` ⟨label suffix⟩ (⟨expression⟩ , ⟨pair expression⟩)
| `dotlabel` ⟨label suffix⟩ (⟨expression⟩ , ⟨pair expression⟩)
| `labels` ⟨label suffix⟩ (⟨point number list⟩)
| `dotlabels` ⟨label suffix⟩ (⟨point number list⟩)
| `textext` ⟨label suffix⟩ (⟨expression⟩)
| `freelabel` (⟨expression⟩ , ⟨pair expression⟩ , ⟨pair expression⟩)
| `freedotlabel` (⟨expression⟩ , ⟨pair expression⟩ , ⟨pair expression⟩)
| `remapcolor` (⟨color expression⟩ , ⟨color expression⟩)
| `resetcolormap`
| `recolor` ⟨picture expression⟩

⟨point number list⟩
→ ⟨suffix⟩ | ⟨point number list⟩ , ⟨suffix⟩

⟨label suffix⟩
→ ⟨empty⟩
| `lft` | `rt` | `top` | `bot` | `ulft` | `urt` | `llft` | `lrt` | `raw` | `origin`

⟨command⟩
→ `clip` ⟨picture variable⟩ `to` ⟨path expression⟩
| `interim` ⟨internal variable⟩ `:=` ⟨right-hand side⟩
| `let` ⟨symbolic token⟩ `=` ⟨symbolic token⟩
| `pickup` ⟨expression⟩
| `randomseed :=` ⟨numeric expression⟩
| `save` ⟨symbolic token list⟩
| `setbounds` ⟨picture variable⟩ `to` ⟨path expression⟩
| `shipout` ⟨picture expression⟩
| `special` ⟨string expression⟩
| ⟨addto command⟩
| ⟨drawing command⟩
| ⟨font metric command⟩
| ⟨newinternal command⟩
| ⟨message command⟩
| ⟨mode command⟩
| ⟨show command⟩
| ⟨special command⟩
| ⟨tracing command⟩

⟨show command⟩
→ `show` ⟨expression list⟩
| `showvariable` ⟨symbolic token list⟩
| `showtoken` ⟨symbolic token list⟩
| `showdependencies`

⟨symbolic token list⟩
→ ⟨symbolic token⟩
| ⟨symbolic token⟩ , ⟨symbolic token list⟩

⟨expression list⟩
→ ⟨expression⟩
| ⟨expression list⟩ , ⟨expression⟩

⟨addto command⟩
→ `addto` ⟨picture variable⟩ `also` ⟨picture expression⟩ ⟨option list⟩
| `addto` ⟨picture variable⟩ `contour` ⟨path expression⟩ ⟨option list⟩
| `addto` ⟨picture variable⟩ `doublepath` ⟨path expression⟩ ⟨option list⟩

⟨option list⟩
→ ⟨empty⟩
| ⟨drawing option⟩ ⟨option list⟩

⟨drawing option⟩
→ `withcolor` ⟨color expression⟩
| `withrgbcolor` ⟨rgbcolor expression⟩
| `withcmykcolor` ⟨cmykcolor expression⟩
| `withoutcolor`
| `withprescript` ⟨string expression⟩
| `withostscript` ⟨string expression⟩

| `withpen` ⟨pen expression⟩
| `dashed` ⟨picture expression⟩
| `withshade` ⟨numeric expression⟩

⟨drawing command⟩
→ `draw` ⟨picture expression⟩ ⟨option list⟩
| ⟨fill type⟩ ⟨path expression⟩ ⟨option list⟩

⟨fill type⟩
→ `fill` | `unfill` | `refill`
| `draw` | `undraw` | `redraw`
| `filldraw` | `drawfill` | `unfilldraw`
| `drawarrow` | `drawdblarrow`
| `cutdraw`

⟨newinternal command⟩
→ `newinternal` ⟨internal type⟩ ⟨symbolic token list⟩
| ⟨newinternal⟩ ⟨symbolic token list⟩

⟨message command⟩
→ `errhelp` ⟨string expression⟩
| `errmessage` ⟨string expression⟩
| `filenametemplate` ⟨string expression⟩
| `message` ⟨string expression⟩

⟨mode command⟩
→ `batchmode`
| `nonstopmode`
| `scrollmode`
| `errorstopmode`

⟨special command⟩
→ `fontmapfile`
| `fontmapline`
| `special`

⟨tracing command⟩
→ `tracingall`
| `loggingall`
| `tracingnone`

⟨if test⟩
→ `if` ⟨boolean expression⟩ `:` ⟨balanced tokens⟩ ⟨alternatives⟩ `fi`

⟨alternatives⟩
→ ⟨empty⟩
| `else :` ⟨balanced tokens⟩
| `elseif` ⟨boolean expression⟩ ⟨:⟩ ⟨balanced tokens⟩ ⟨alternatives⟩

⟨loop⟩
→ ⟨loop header⟩ : ⟨loop text⟩ `endfor`

⟨loop header⟩
→ `for` ⟨symbolic token⟩ `=` ⟨progression⟩
| `for` ⟨symbolic token⟩ `=` ⟨for list⟩
| `for` ⟨symbolic token⟩ `within` ⟨picture expression⟩
| `forsuffixes` ⟨symbolic token⟩ `=` ⟨suffix list⟩
| `forever`

⟨progression⟩
→ ⟨numeric expression⟩ `upto` ⟨numeric expression⟩
| ⟨numeric expression⟩ `downto` ⟨numeric expression⟩
| ⟨numeric expression⟩ `step` ⟨numeric expression⟩ `until` ⟨numeric expression⟩

⟨for list⟩
→ ⟨expression⟩
| ⟨for list⟩ `,` ⟨expression⟩

⟨suffix list⟩
→ ⟨suffix⟩
| ⟨suffix list⟩ , ⟨suffix⟩

A.2 Left overs

There are a few more concepts and commands available in METAFUN, like color remapping, shading and graphic inclusion. Because they have their own small syntax world, we present them here.

You may consider shades to be internally represented by a hidden datastructure. The end user has access to a shade by means of a pointer, expressed in a numeric.

⟨pseudo procedure⟩
→ `linear\_shade` (⟨path expression⟩ , ⟨numeric expression⟩ , ⟨color expression⟩ , ⟨color expression⟩)
| `circular\_shade` (⟨path expression⟩ , ⟨numeric expression⟩ , ⟨color expression⟩ , ⟨color expression⟩)

⟨pseudo function⟩
→ `define\_linear\_shade` (⟨pair expr⟩ , ⟨pair expr⟩ , ⟨color expr⟩ , ⟨color expr⟩)
| `define\_circular\_shade` (⟨pair expr⟩ , ⟨pair expr⟩ , ⟨path expr⟩ , ⟨path expr⟩ , ⟨color expr⟩ , ⟨color expr⟩)
| `predefined\_linear\_shade` (⟨path expr⟩ , ⟨numeric expr⟩ , ⟨color expr⟩ , ⟨color expr⟩)
| `predefined\_circular\_shade` (⟨path expr⟩ , ⟨numeric expr⟩ , ⟨color expr⟩ , ⟨color expr⟩)

External figures are just files, so the string passed as first argument needs to be a valid filename. Additionally, they need to be given dimensions.

⟨pseudo procedure⟩
→ `externalfigure` ⟨string expression⟩ ⟨transformer⟩

An external METAPOST graphic can be loaded by filename and figure number. The normal transformations can be applied.

⟨pseudo procedure⟩
→ `loadfigure` ⟨string expression⟩ ⟨figure number⟩ ⟨transformer⟩

⟨figure number⟩
→ `number` ⟨numeric expression⟩

A graphic text is (normally) an outline representation of a snippet of text typeset by TEX. This procedure has a couple of dedicated options.

⟨pseudo procedure⟩
→ `graphictext` ⟨string expression⟩ ⟨transformer⟩ ⟨text option list⟩
| `regraphictext` ⟨transformer⟩ ⟨text option list⟩

⟨text option list⟩
→ ⟨empty⟩
| ⟨text drawing option⟩ ⟨text option list⟩

⟨text drawing option⟩
→ ⟨drawing option⟩
| `reversefill`
| `outlinefill`
| `withdrawcolor` ⟨color expression⟩
| `withfillcolor` ⟨color expression⟩

⟨pseudo procedure⟩
→ `resetgraphictextdirective`
| `graphictextdirective` ⟨string expression⟩

⟨internal variable⟩
→ `graphictextformat`

B

This document

This document is produced in CONTEXT and can serve as an example of how to integrate METAPOST graphics into TEX. In this appendix we will discuss some details of producing this document.

We did not use any special tricks, so most of the examples you have seen, were coded just as shown. We used buffers to ensure that the code used to produce the accompanying graphic is identical to the typeset code in the document. Here is an example.

```
\startbuffer[dummy]
draw fullcircle
  xscaled 3cm yscaled 2cm
  rotatedaround(origin,30)
  withcolor .625red ;
\stopbuffer
```

Instead of using `\getbuffer`, we used the following command:

```
\startlinecorrection[blank]
\processMPbuffer[dummy]
\stoplinecorrection
```

The line correction commands take care of proper spacing around the graphic. If you want to process more buffers at once, you can pass their names as a comma separated list.

Alternatively, we could have said:

```
\startuseMPgraphic{dummy}
  draw fullcircle
    xscaled 3cm yscaled 2cm
    rotatedaround(origin,30)
    withcolor .625red ;
\stopuseMPgraphic
```

When including this graphic, we again take care of spacing.

```
\startlinecorrection[blank]
\useMPgraphic{dummy}
\stoplinecorrection
```

The first version of this manual was produced with PDFTEX and calls out to METAPOST. Because the number of graphics is large, we processed that version using the `--automp` directive (at that moment we were using TEXEXEC). And even then runtime was so unconveniently long that updating this manual became less and less fun. The current version is produced with LUATEX and CONTEXT MKIV, which brings down the runtime (including runtime calls to independent CONTEXT runs for the outline examples) to some 45 seconds on a 2.2 Gig Dell M90. Given that (at the time of writing this) over 1700 graphics are generated on the fly, this is not bad at all. On my current machine, a Dell M6700 with an Intel Core i7-3840QM running at 2.8 (3.9) Ghz (and Windows 8) the runtime is just above 20 seconds all-in and a few seconds less when using LUAJITTEX.

The document style is not that complicated. The main complication in such a document is to make sure that METAPOST is operating under the same font regime, but in MKIV this happens automatically. As document font we use the URW Palatino for the running text combined with Computer Modern Typewriter.

Because this document is available as paper and screen document, some large graphics are scaled down in the screen version.

We don't use any special tricks in typesetting this document, but when we added the section about transparency, a dirty trick was needed in a few cases order to get the described results. Because the screen document has gray backgrounds, exclusive transparencies come out 'wrong'. In the function drawing example we use the following trick to get a black background behind the graphics only. We have a buffer that contains a few lines of code:

```
picture savedpicture ;
savedpicture := currentpicture ;
currentpicture := nullpicture ;
draw savedpicture withcolor black ;
draw savedpicture ;
```

Since we use buffers for the graphics as well, we can now process a buffer with name `example` as follows:

```
\processbuffer[example,wipe]
```

This means that the example code is included two times. After it is processed, we recolor the currentpicture black, and after that we add the original picture once again.

C

Reference

In this chapter, we will demonstrate most of the drawing related primitives and macros as present in plain METAPOST and METAFUN extensions.

If a path is shown and/or a transformation is applied, we show the original in red and the transformed path or point in yellow. The small dark gray crosshair is the origin and the black rectangle the bounding box. In some drawings, in light gray we show the points that makeup the path.

This list describes traditional METAPOST and the stable part of METAFUN. As METAPOST evolves, new primitives are added but they are not always that relevant to us. If you browse the METAFUN sources you will for sure notice more functionality then summarized here. Most of that is meant for usage in CONTEXT and not exposed to the user. Other macros are still somewhat experimental but might become more official at some point. When METAPOST version 2 is ready this reference will be updated accordingly.

C.1

Paths

pair

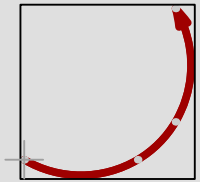
(1,.5)

metapost concept

pair .. pair

```
(0,0)..(.75,0)..(1,.25)..(1,1)
```

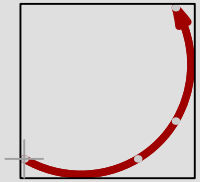
metapost macro



pair ... pair

```
(0,0)..(.75,0)...(1,.25)..(1,1)
```

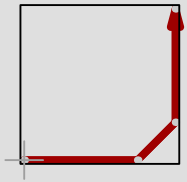
metapost macro



pair -- pair

```
(0,0)--(.75,0)--(1,.25)--(1,1)
```

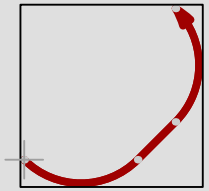
metapost macro



pair --- pair

```
(0,0)..(.75,0)---(1,.25)..(1,1)
```

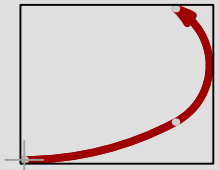
metapost macro



pair softjoin pair

```
(0,0)..(.75,0) softjoin (1,.25)..(1,1)
```

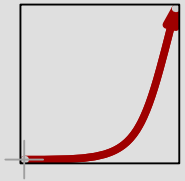
metapost macro



controls pair

```
(0,0)..controls (.75,0)..(1,1)
```

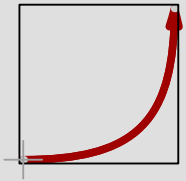
metapost primitive



controls pair and pair

```
(0,0)..controls (.75,0) and (1,.25)..(1,1)
```

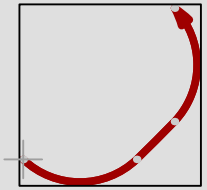
metapost primitive



tension numeric

```
(0,0)..(.75,0)..tension 2.5..(1,.25)..(1,1)
```

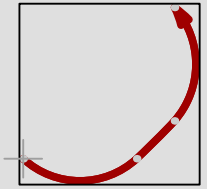
metapost primitive



tension num.. and num..

```
(0,0)..(.75,0)..tension 2.5 and 1.5..(1,.25)..(1,1)
```

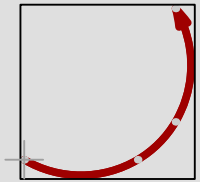
metapost primitive



tension atleast numeric

```
(0,0)..(.75,0)..tension atleast 1..(1,.25)..(1,1)
```

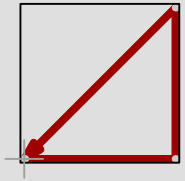
metapost primitive



cycle

```
(0,0)--(1,0)--(1,1)--cycle
```

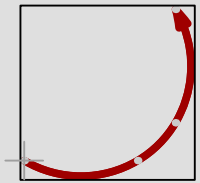
metapost primitive



curl numeric

```
(0,0)curl 1..(.75,0)..(1,.25)..(1,1)
```

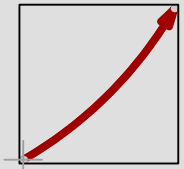
metapost primitive



dir numeric

(0,0)dir 30..(1,1)

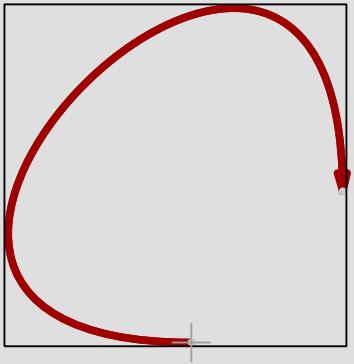
metapost primitive



left

(0,0)left..(1,1)

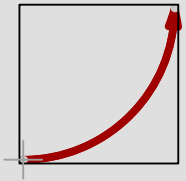
metapost macro



right

(0,0)right..(1,1)

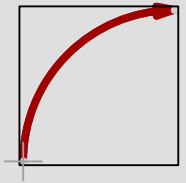
metapost macro



up

(0,0)up..(1,1)

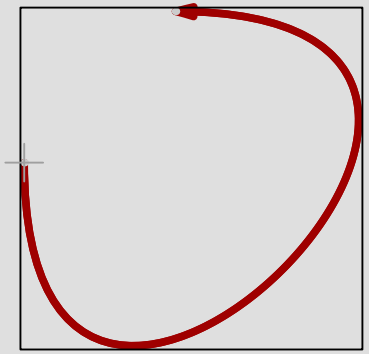
metapost macro



down

`(0,0)down..(1,1)`

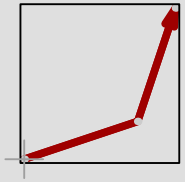
`metapost macro`



path & path

`(0,0)..(.75,.25) & (.75,.25)..(1,1)`

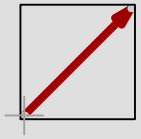
`metapost primitive`



unitvector

`origin--unitvector(1,1)`

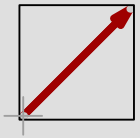
`metapost variable`



dir

origin--dir(45)

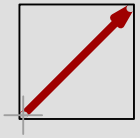
metapost primitive



angle

origin--dir(angle(1,1))

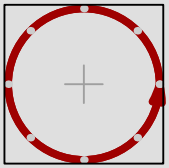
metapost primitive



fullcircle

fullcircle

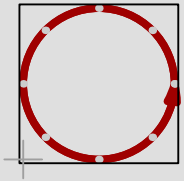
metapost variable



unitcircle

unitcircle

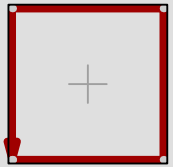
metafun variable



fullsquare

fullsquare

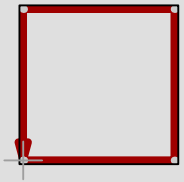
metafun variable



unitsquare

unitsquare

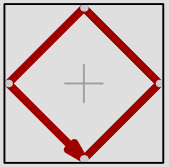
metapost variable



fulldiamond

fulldiamond

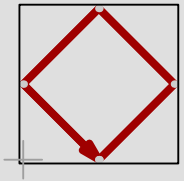
metafun variable



unitdiamond

unitdiamond

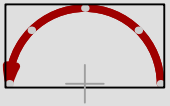
metafun variable



halfcircle

halfcircle

metapost variable



quartercircle

quartercircle

metapost variable

llcircle

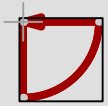
llcircle

metafun variable

lrcircle

lrcircle

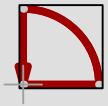
metafun variable



urcircle

urcircle

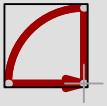
metafun variable



ulcircle

ulcircle

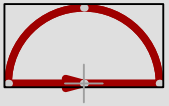
metafun variable



tcircle

tcircle

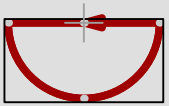
metafun variable



bcircle

bcircle

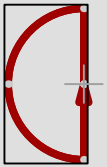
metafun variable



lcircle

lcircle

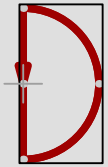
metafun variable



rcircle

rcircle

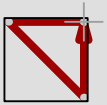
metafun variable



lltriangle

lltriangle

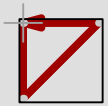
metafun variable



lrtriangle

lrtriangle

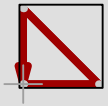
metafun variable



urtriangle

urtriangle

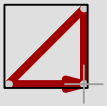
metafun variable



ultriangle

ultriangle

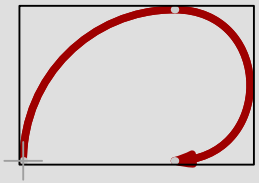
metafun variable



flex(pair,pair,pair)

flex ((0,0),(1,1),(1,0))

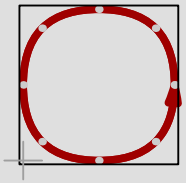
metapost macro



superellipse(pair,p..,p..,p..,num..)

superellipse((1,.5),(.5,1),(0,.5),(.5,0),.75)

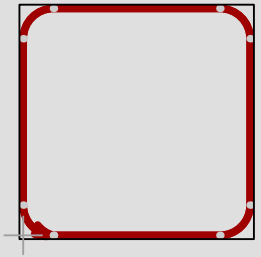
metapost macro



path smoothed numeric/pair

`unitsquare scaled 1.5 smoothed .2`

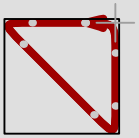
metafun macro



path cornered numeric/pair

`lltriangle scaled 1.5 cornered .2`

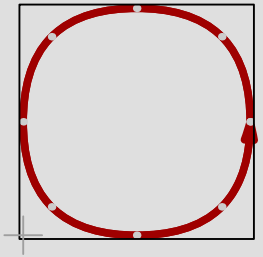
metafun macro



path superellipsed numeric

`unitsquare scaled 1.5 superellipsed .75`

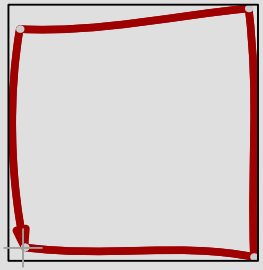
metafun macro



path randomized numeric/pair

`unitsquare scaled 1.5 randomized (.2,.2)`

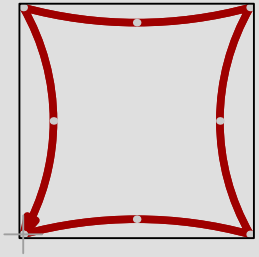
`metafun macro`



path squeezed numeric/pair

`unitsquare scaled 1.5 squeezed (.2,.1)`

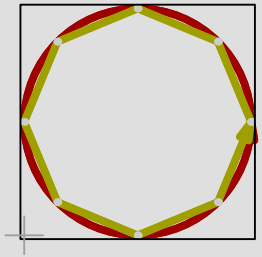
`metafun macro`



punked path

`punked unitcircle scaled 1.5`

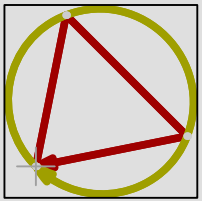
`metafun macro`



curved path

`curved ((0,0)--(.2,1)--(1,.2)--cycle)`

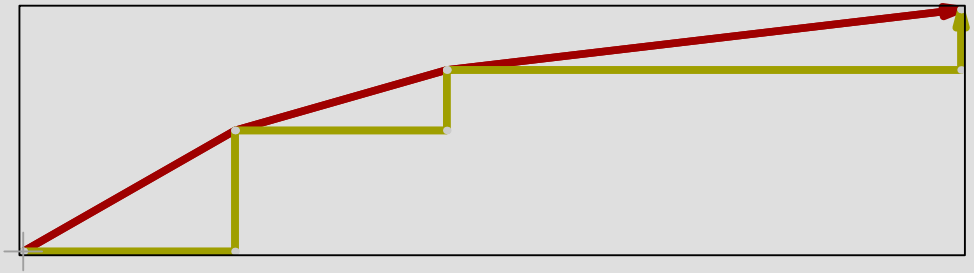
`metafun macro`



laddered path

`laddered ((0,0)--(1.4,.8)--(2.8,1.2)--(6.2,1.6))`

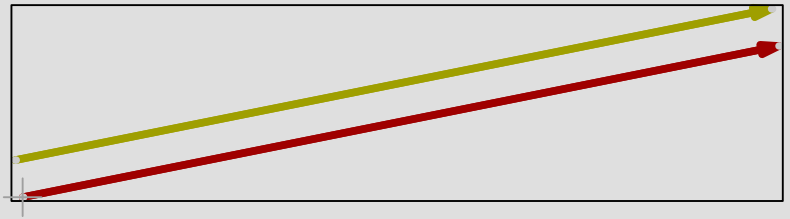
metafun macro



path paralleled distance

`((0,0)--(5,1)) paralleled .25`

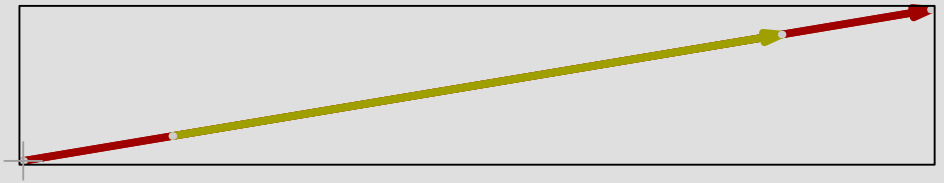
metafun macro



shortened path

`((0,0)--(6,1)) shortened 1`

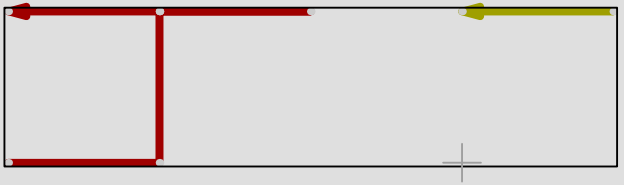
metafun macro



unspiked path

`unspiked ((0,0)--(1,0)--(1,1)--(2,1)--(1,1)--(0,1))`

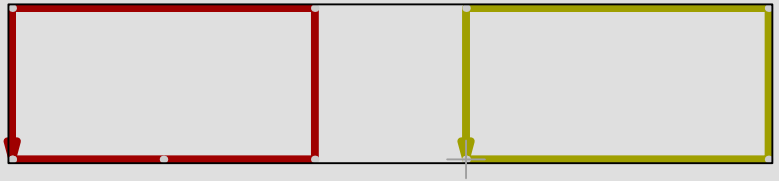
metafun macro



simplified path

`simplified ((0,0)--(1,0)--(2,0)--(2,1)--(0,1)--cycle)`

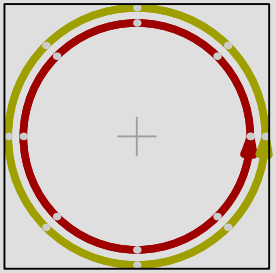
metafun macro



path blownup numeric/pair

`(fullcircle scaled 1.5) blownup .1`

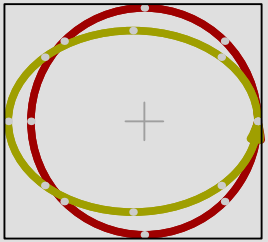
metafun macro



path stretched numeric/pair

```
(fullcircle scaled 1.5) stretched (1.1,0.8)
```

metafun macro



roundedsquare(num..,num..,num..)

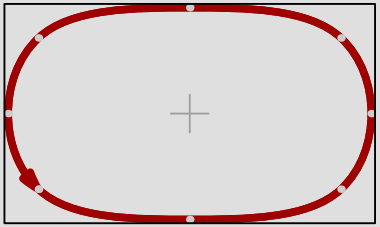
```
roundedsquare(2,1,.2)
```

metafun macro

tensecircle(num..,num..,num..)

```
tensecircle(2,1,.2)
```

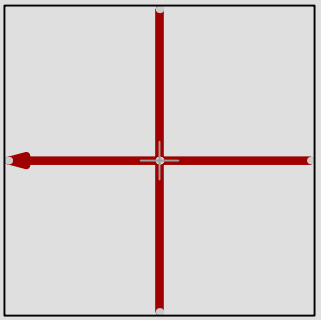
metafun macro



pair crossed size

```
origin crossed 1
```

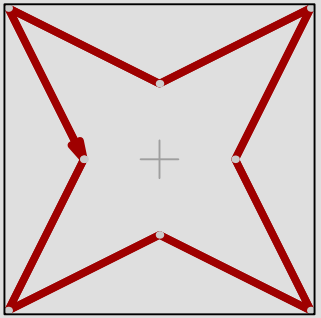
metafun macro



path crossed size

```
fullcircle scaled 2 crossed .5
```

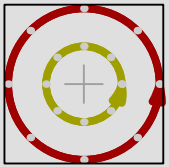
metafun macro



path scaled numeric

`fullcircle scaled .50`

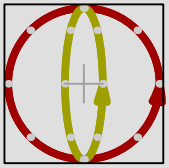
metapost primitive



path xscaled numeric

`fullcircle xscaled .25`

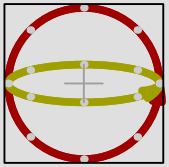
metapost primitive



path yscaled numeric

`fullcircle yscaled .25`

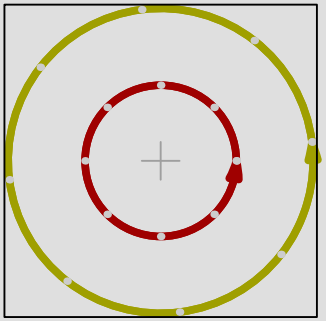
metapost primitive



path zscaled pair

```
fullcircle zscaled (2,.25)
```

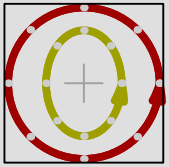
metapost primitive



path xyscaled numeric/pair

```
fullcircle xyscaled (.5,.7)
```

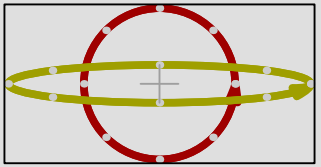
metapost primitive



path xyscaled pair

```
fullcircle xyscaled (2,.25)
```

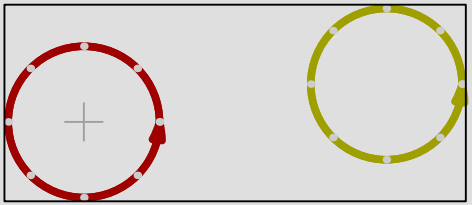
metapost primitive



path shifted pair

`fullcircle shifted (2,.25)`

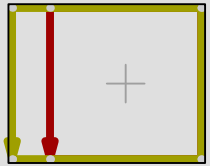
`metapost primitive`



path leftenlarged numeric

`fullsquare leftenlarged .25`

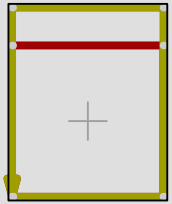
`metafun macro`



path topenlarged numeric

`fullsquare topenlarged .25`

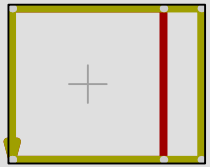
`metafun macro`



path rightenlarged numeric

`fullsquare rightenlarged .25`

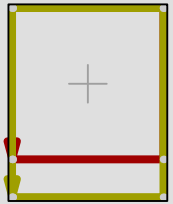
metafun macro



path bottomenlarged numeric

`fullsquare bottomenlarged .25`

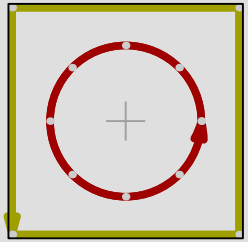
metafun macro



path enlarged numeric

`fullcircle enlarged .25`

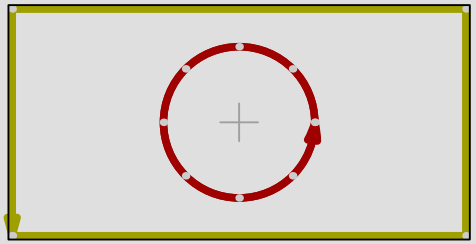
metafun macro



path enlarged pair

`fullcircle enlarged (1,.25)`

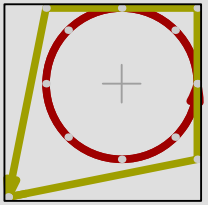
metafun macro



path llenlarged numeric

`fullcircle llenlarged .25`

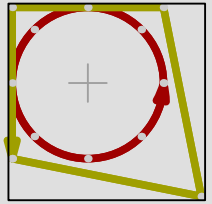
metafun macro



path lrenlarged numeric

`fullcircle lrenlarged .25`

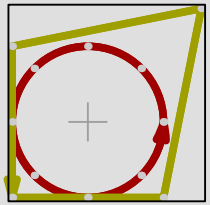
metafun macro



path urenlarged numeric

`fullcircle urenlarged .25`

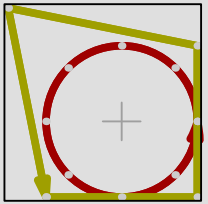
`metafun macro`



path ulenlarged numeric

`fullcircle ulenlarged .25`

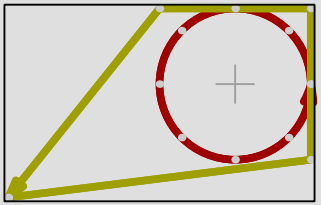
`metafun macro`



path llenlarged pair

`fullcircle llenlarged (1,.25)`

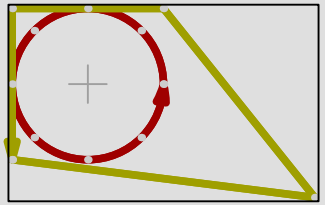
`metafun macro`



path lrenlarged pair

`fullcircle lrenlarged (1,.25)`

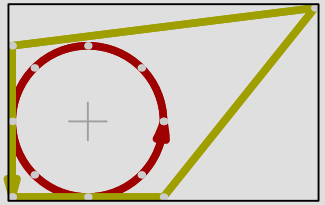
`metafun macro`



path urenlarged pair

`fullcircle urenlarged (1,.25)`

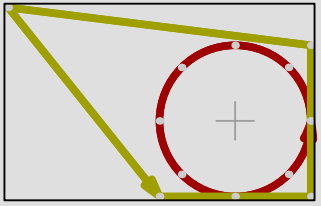
`metafun macro`



path ulenlarged pair

`fullcircle ulenlarged (1,.25)`

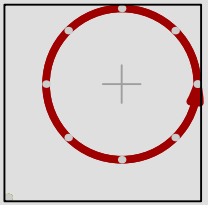
`metafun macro`



path llmoved numeric

`fullcircle llmoved .25`

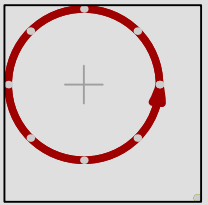
`metafun macro`



path lrmoved numeric

`fullcircle lrmoved .25`

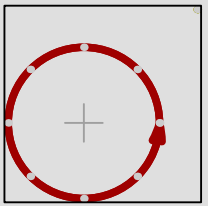
`metafun macro`



path urmoved numeric

`fullcircle urmoved .25`

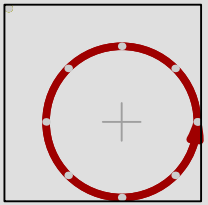
`metafun macro`



path ulmoved numeric

`fullcircle ulmoved .25`

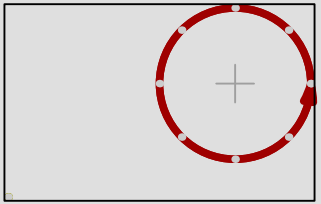
metafun macro



path llmoved pair

`fullcircle llmoved (1,.25)`

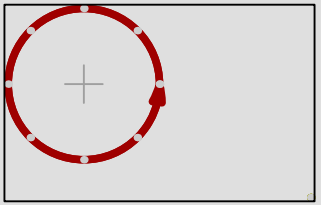
metafun macro



path lrmoved pair

`fullcircle lrmoved (1,.25)`

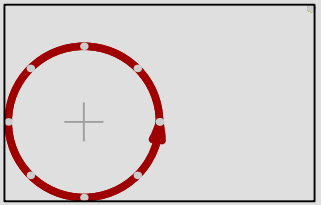
metafun macro



path urmoved pair

`fullcircle urmoved (1,.25)`

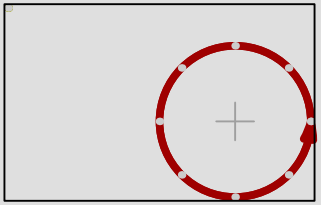
`metafun macro`



path ulmoved pair

`fullcircle ulmoved (1,.25)`

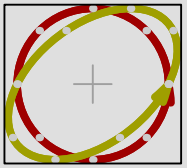
`metafun macro`



path slanted numeric

`fullcircle slanted .5`

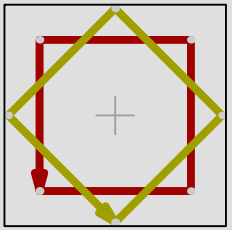
`metapost primitive`



path rotated numeric

`fullsquare rotated 45`

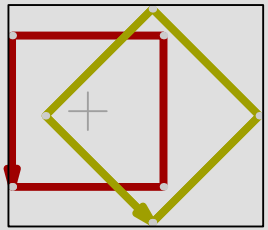
`metapost primitive`



path rotatedaround(pair,numeric)

`fullsquare rotatedaround((.25,.5),45)`

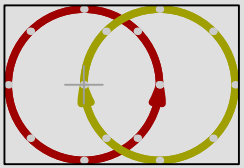
`metapost macro`



path reflectedabout(pair,pair)

`fullcircle reflectedabout((.25,-1),(.25,+1))`

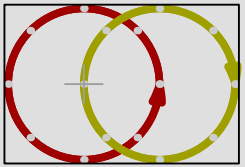
`metapost macro`



reverse path

```
reverse fullcircle shifted(.5,0)
```

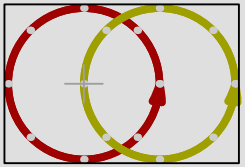
metapost primitive



counterclockwise path

```
counterclockwise fullcircle shifted(.5,0)
```

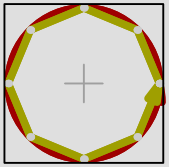
metapost macro



tensepath path

```
tensepath fullcircle
```

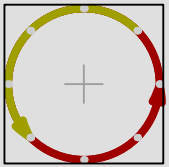
metapost macro



subpath (numeric,numeric) of path

```
subpath (1,5) of fullcircle
```

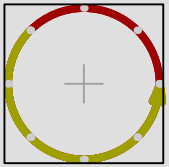
metapost primitive



path cutbefore pair

```
fullcircle cutbefore point 3 of fullcircle
```

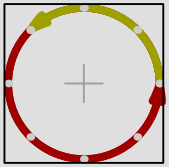
metapost macro



path cutafter pair

```
fullcircle cutafter point 3 of fullcircle
```

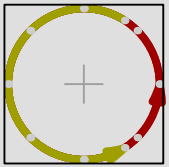
metapost macro



path cutends .1

`fullcircle cutends .5`

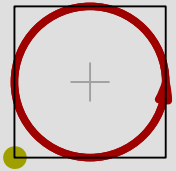
`metapost macro`



llcorner path

`llcorner fullcircle`

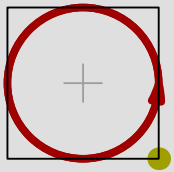
`metapost primitive`



lrcorner path

`lrcorner fullcircle`

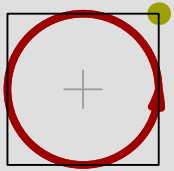
`metapost primitive`



urcorner path

urcorner fullcircle

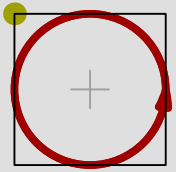
metapost primitive



ulcorner path

ulcorner fullcircle

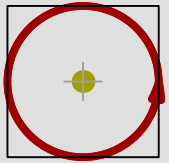
metapost primitive



center path

center fullcircle

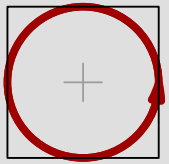
metapost macro



boundingbox path

`boundingbox fullcircle`

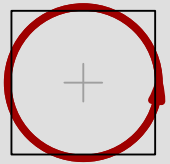
`metafun macro`



innerboundingbox path

`innerboundingbox fullcircle`

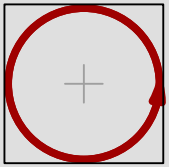
`metafun macro`



outerboundingbox path

`outerboundingbox fullcircle`

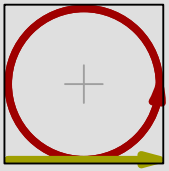
`metafun macro`



bottomboundary path

bottomboundary fullcircle

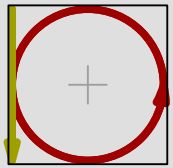
metafun macro



leftboundary path

leftboundary fullcircle

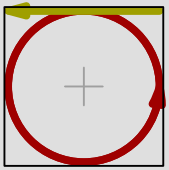
metafun macro



topboundary path

topboundary fullcircle

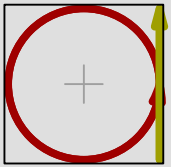
metafun macro



rightboundary path

```
rightboundary fullcircle
```

metafun macro



bbwidth path

```
draw textext(decimal bbwidth (fullcircle xscaled 100 yscaled 200))
```

metafun macro

100

bbwidth path

```
draw textext(decimal bbheight (fullcircle xscaled 100 yscaled 200))
```

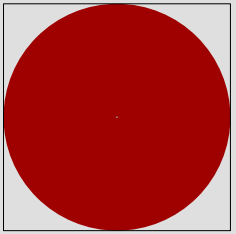
metafun macro

200

path/picture xsized numeric

```
currentpicture xsized 5cm
```

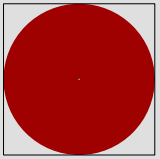
metafun macro



path/picture ysized numeric

currentpicture ysized 2cm

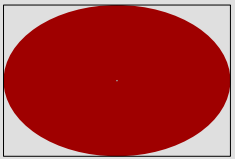
metafun macro



path/picture xysized numeric

currentpicture xysized (3cm,2cm)

metafun macro

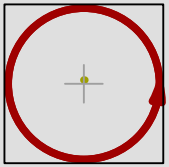


C.3 Points

top pair

top center fullcircle

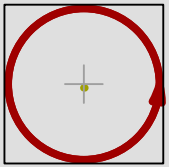
metapost macro



bot pair

bot center fullcircle

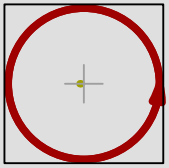
metapost macro



lft pair

lft center fullcircle

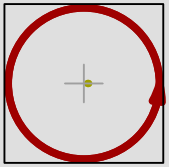
metapost macro



rt pair

rt center fullcircle

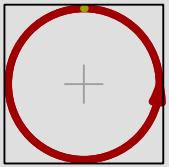
metapost macro



point numeric of path

point 2 of fullcircle

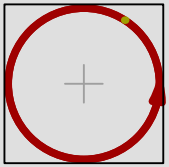
metapost primitive



point numeric on path

point .5 on fullcircle

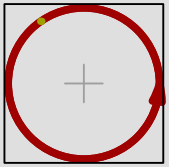
metafun macro



point numeric along path

point 1cm along fullcircle

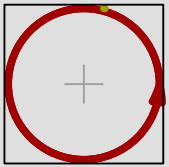
metafun macro



precontrol numeric of path

`precontrol 2 of fullcircle`

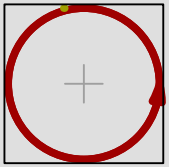
`metapost primitive`



postcontrol numeric of path

`postcontrol 2 of fullcircle`

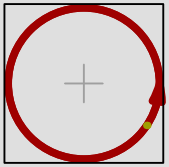
`metapost primitive`



directionpoint pair of path

`directionpoint (2,3) of fullcircle`

`metapost primitive`



numeric[pair,pair]

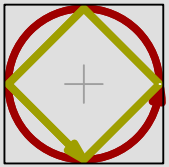
.5[(0,0),(1,1)]

metapost concept

path intersectionpoint path

fullcircle intersectionpoint fulldiamond

metapost macro

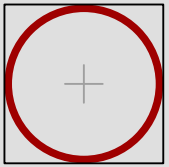


C.4 Attributes

withcolor rgbcolor

withcolor (.625,0,0)

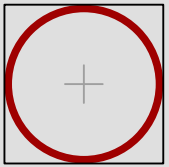
metapost primitive



withrgbcolor rgbcolor

```
withrgbcolor (.625,0,0)
```

metapost primitive



withcmykcolor cmykcolor

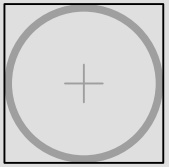
```
withcmykcolor (.625,0,0,0)
```

metapost primitive

withgray numeric

```
withgray .625
```

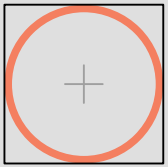
metapost primitive



withcolor cmyk(c,m,y,k)

withcolor cmyk(0,.625,.625,0)

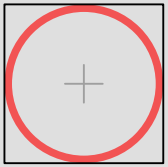
metafun macro



withcolor transparent(num,num,color)

withcolor transparent(1,.625,red)

metafun macro



withshade numeric

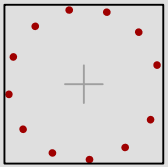
metafun macro

Shades need to be declared before they can be (re)used.

dashed withdots

dashed withdots

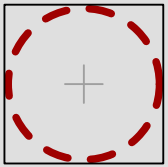
metapost primitive



dashed evenly

dashed evenly

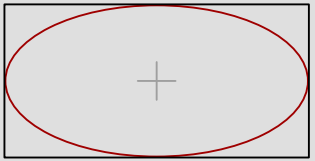
metapost primitive



withpen pencircle transform

withpen pencircle scaled 2mm

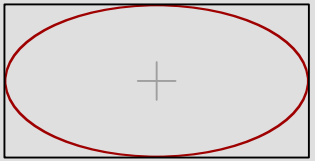
metapost macro



withpen pensquare transform

withpen pensquare scaled 2mm

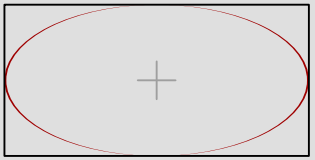
metapost macro



withpen penrazor transform

`withpen penrazor scaled 2mm`

metapost macro



withpen penspeck transform

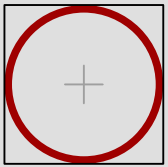
`withpen penspeck`

metapost macro

draw

`draw fullcircle`

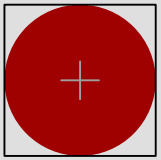
metapost macro



fill

```
fill fullcircle
```

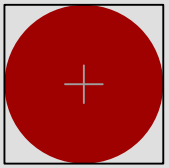
metapost macro



filldraw

```
filldraw fullcircle
```

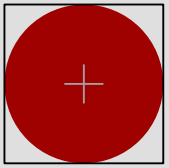
metapost macro



drawfill

```
drawfill fullcircle
```

metapost macro



drawdot

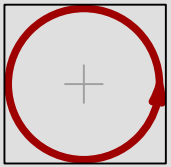
```
drawdot origin
```

metapost macro

drawarrow

drawarrow fullcircle

metapost macro



undraw

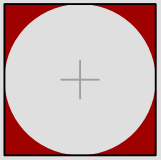
undraw fullcircle

metapost macro

unfill

unfill fullcircle

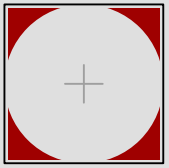
metapost macro



unfilldraw

unfilldraw fullcircle

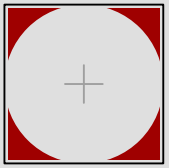
metapost macro



undrawfill

undrawfill fullcircle

metapost macro



undrawdot

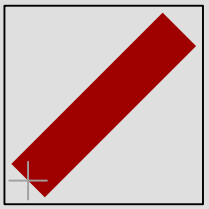
undrawdot origin

metapost macro

cutdraw

origin--(1,1)

metapost macro



butt

`linecap := butt`

`metapost variable`

rounded

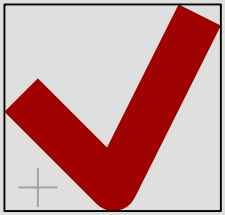
`linecap := rounded`

`metapost variable`

squared

`linecap := squared`

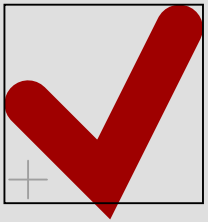
`metapost variable`



mitered

`linejoin := mitered`

metapost variable



rounded

`linejoin := rounded`

metapost variable

beveled

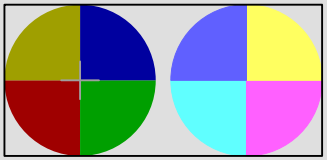
`linejoin := beveled`

metapost variable

inverted picture

`inverted currentpicture`

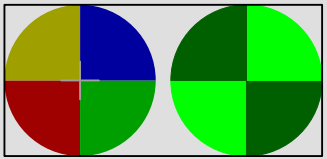
`metafun macro`



picture uncolored color

`currentpicture uncolored green`

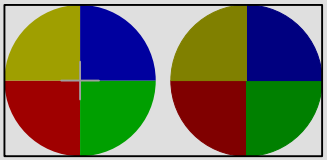
`metafun macro`



picture softened numeric

`currentpicture softened .8`

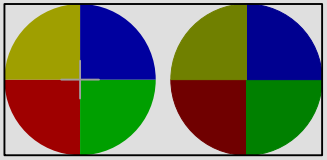
`metafun macro`



picture softened color

currentpicture softened (.7,.8,.9)

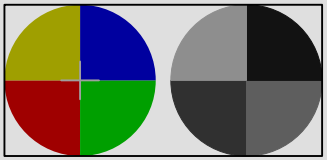
metafun macro



grayed picture

grayed currentpicture

metafun macro



C.5 Text

label

label("MetaFun",origin)

metapost macro

label.top

label.top("MetaFun",origin)

metapost macro

label.bot

```
label.bot("MetaFun",origin)
```

metapost macro

label.lft

```
label.lft("MetaFun",origin)
```

metapost macro

label.rt

```
label.rt("MetaFun",origin)
```

metapost macro

label.llft

```
label.llft("MetaFun",origin)
```

metapost macro

label.lrt

```
label.lrt("MetaFun",origin)
```

metapost macro

label.urt

```
label.urt("MetaFun",origin)
```

metapost macro

label.ulft

```
label.ulft("MetaFun",origin)
```

metapost macro

btex text etex

```
draw btex MetaTeX etex
```

metapost primitive

textext(string)

```
draw textext("MetaFun")
```

metafun macro

graphictext string ...

```
graphictext "MetaFun"
```

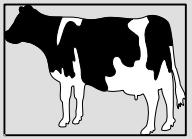
metafun macro

C.6 Graphics

loadfigure string number numeric ...

```
loadfigure "mycow.mp" number 1 scaled .25
```

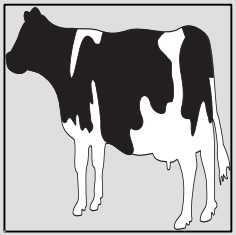
metafun macro



externalfigure string ...

```
draw externalfigure "mycow.pdf" scaled 3cm
```

metafun macro



addbackground text

```
addbackground withcolor .625 yellow
```

metafun macro

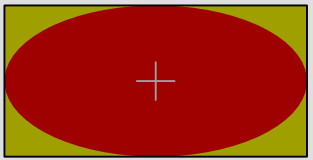
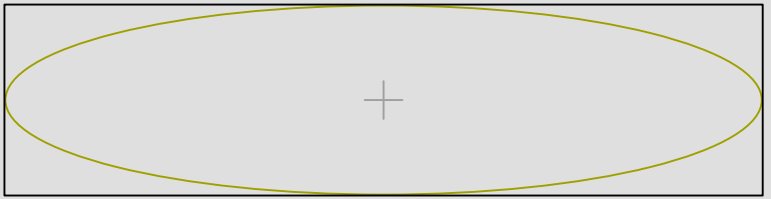


image (text)

```
draw image(draw fullcircle) xscaled 4cm yscaled 1cm
```

metapost macro



Index

For them

I owe much inspiration to both my parents. My mother Jannie constantly demonstrates me that computer graphics will never improve nature. She also converted one of my first METAPOST graphics into a patchwork that will remind me forever that handcraft is more vivid than computer artwork. My father Hein has spent a great deal of his life teaching math, and I'm sure he would have loved METAPOST. I inherited his love for books. I therefore dedicate this document to them.

Colofon

This manual is typeset with CONTEXT MKIV. No special tricks are used and everything you see in here, is available for CONTEXT users. The text is typeset in Palatino and Computer Modern Typewriter. We used LUATEX as TEX processing engine. Since this document is meant to be printed in color, some examples will look sub–optimal when printed in black and white.

Graphics

The artist impression of one of Hasselts canals at **page 340** is made by Johan Jonker. The CDROM production process graphic at **page 334** is a scan of a graphic made by Hester de Weert.

Copyright

Publisher

publisher: Boekplan, NL

isbn-ean: 978-94-90688-02-8
website: www.boekplan.nl

Info

internet: www.pragma-ade.com
support: ntg-context@ntg.nl
context: www.contextgarden.net