Graphics in pdfTeX

There is no separate back-end, so PDFTeX has to support all those backed features that DVI drivers provide.

Graphics in pdfTEX

There is no separate back–end, so PDFTEX has to support all those backed features that DVI drivers provide.

This means that PDFTEX must handle color, graphics inclusion, transformations, hyperlinks and widgets itself.

Graphics in pdfTEX

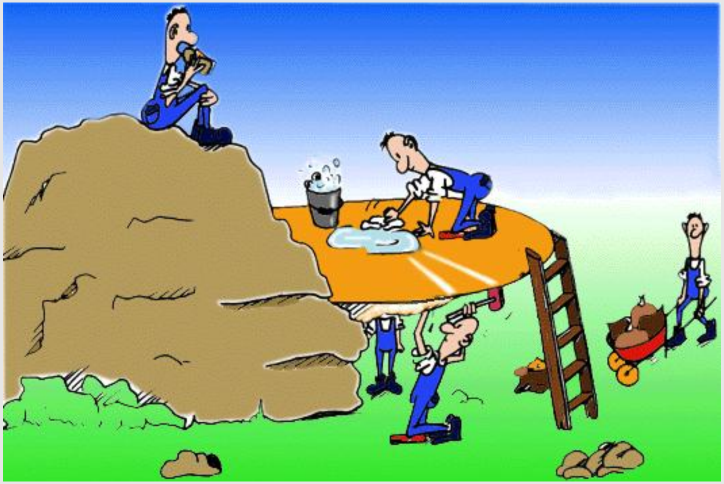
PDFTEX supports the following bitmap formats:

- PNG
- JPG
- TIFF

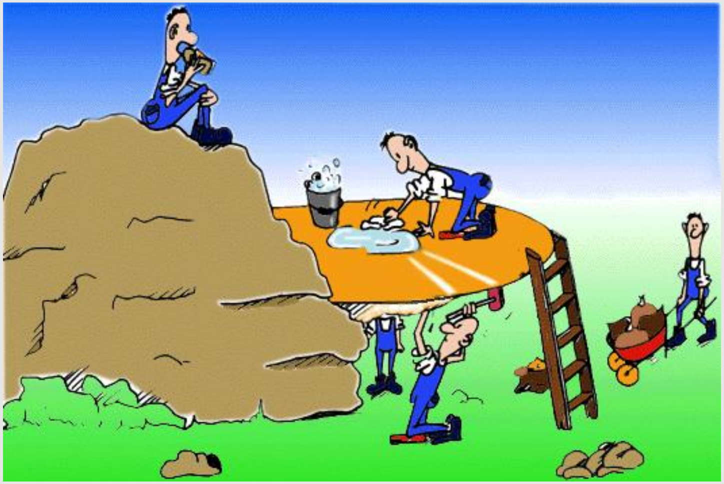
Graphics in pdfTEX

And of course, PDFTEX also supports vector graphics, like:

- PDF
- METAPOST
- literals



The three bitmap graphic formats PNG, JPG and TIFF are stored in compressed streams with lossless and lossy compression schemes.



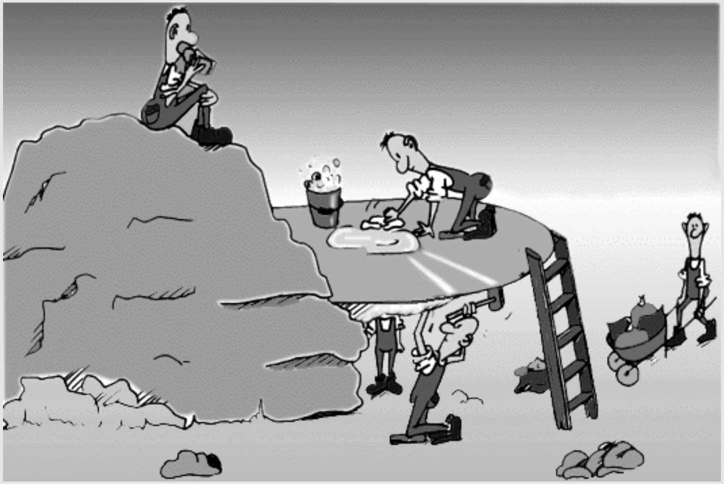
The three bitmap graphic formats PNG, JPG and TIFF are stored in compressed streams with lossless and lossy compression schemes.

This means that sometimes graphics are decompressed when being loaded and recompressed while being embedded.

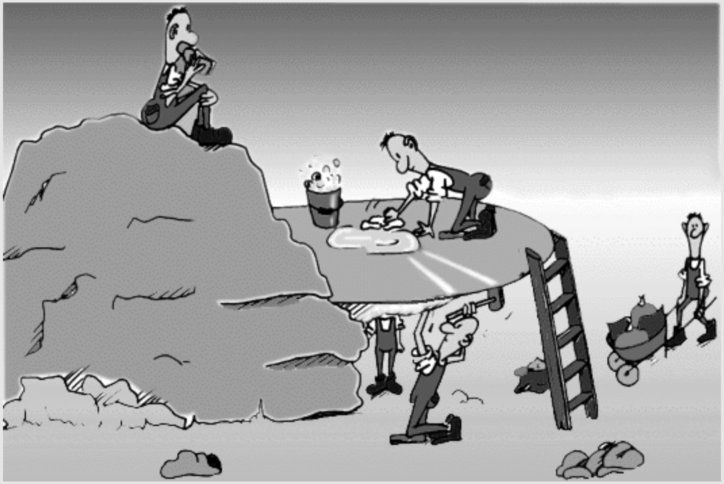
The three bitmap graphic formats PNG, JPG and TIFF are stored in compressed streams with lossless and lossy compression schemes.

This means that sometimes graphics are decompressed when being loaded and recompressed while being embedded.

PDFTEX stores graphics in xform objects and refers to them afterwards, so that reusing graphics is very efficient.

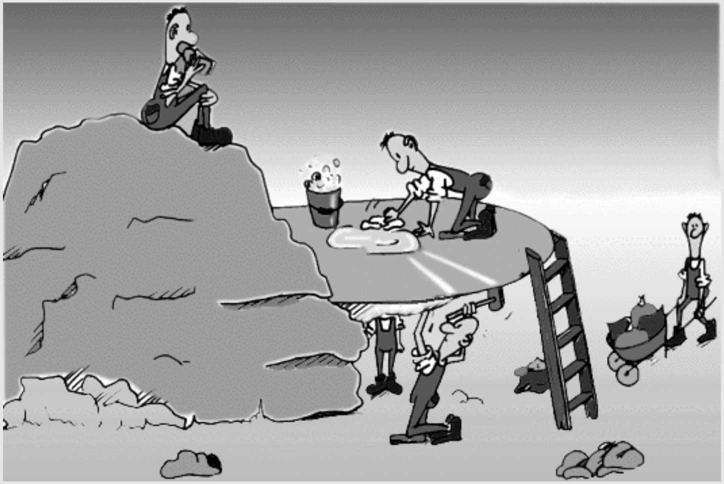


We can also specify an alternative graphic for printing.



We can also specify an alternative graphic for printing.

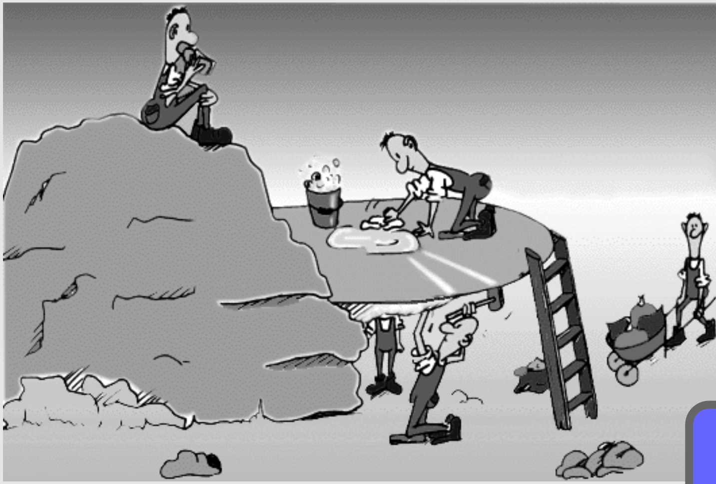
This saves bandwidth when viewing the document.



We can also specify an alternative graphic for printing.

This saves bandwidth when viewing the document.

Unfortunately alternative images are only supported for (identical) bitmap images.

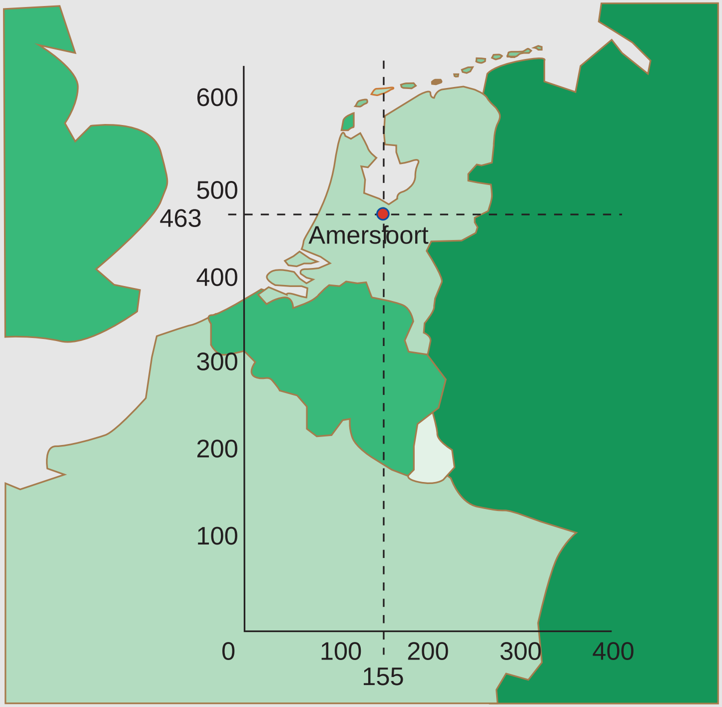


We can also specify an alternative graphic for printing.

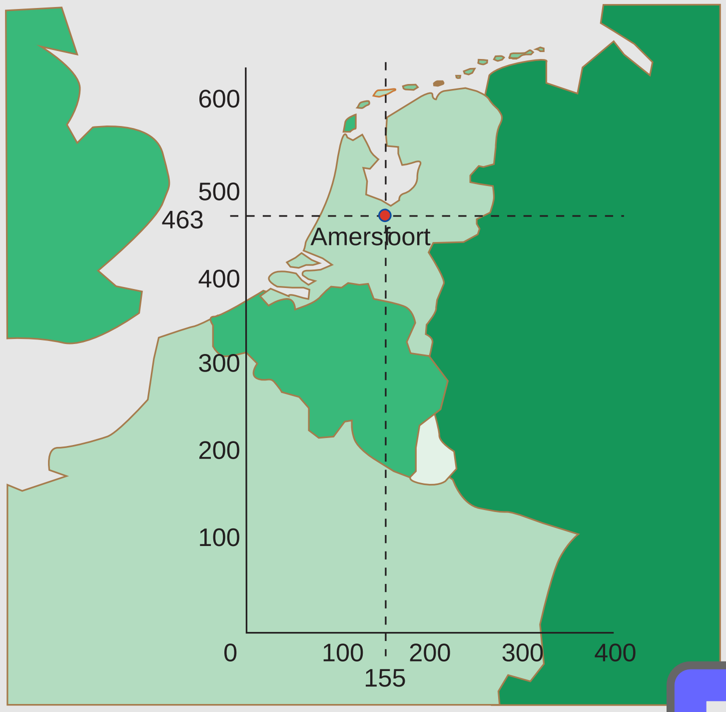
This saves bandwidth when viewing the document.

Unfortunately alternative images are only supported for (identical) bitmap images.

Because some applications twist the truth about dimensions of graphics, you can specify the resolution as known to you.

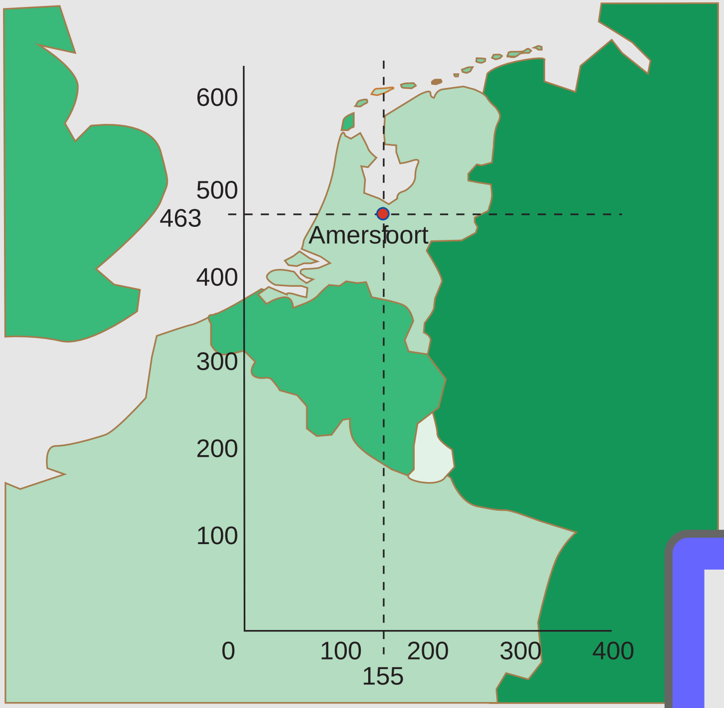


By supporting PDF inclusion, PDFTEX can include nearly every graphic you want.



By supporting PDF inclusion, PDFTEX can include nearly every graphic you want.

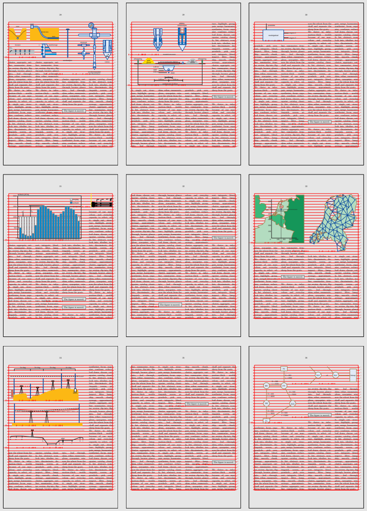
When doing so, PDFTEX tries hard to make sure that resources and fonts are not interfering.



By supporting PDF inclusion, PDFTEX can include nearly every graphic you want.

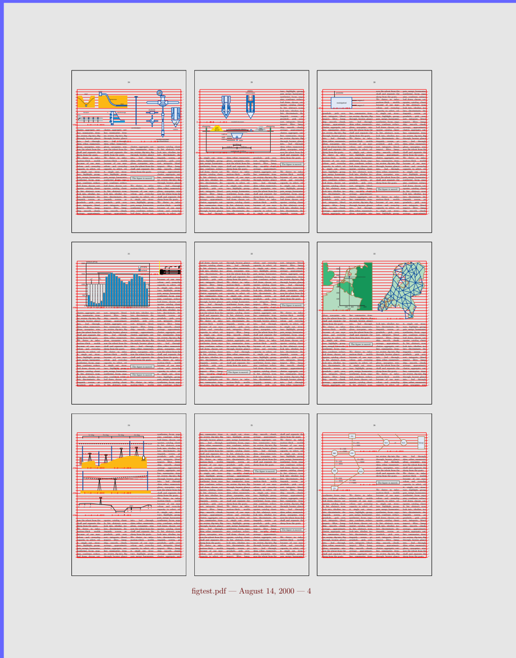
When doing so, PDFTEX tries hard to make sure that resources and fonts are not interfering.

As a bonus, PDFTEX will share as many fonts as possible.



figtest.pdf — August 14, 2000 — 4

Because PDFTEX can pick up an arbitrary page from a file, you can use PDFTEX as postprocessor.



Because PDFTEX can pick up an arbitrary page from a file, you can use PDFTEX as postprocessor.

This feature can be used to produce booklets and image sheets, to reprocess documents, to typeset annotated pages, etc.

```
\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

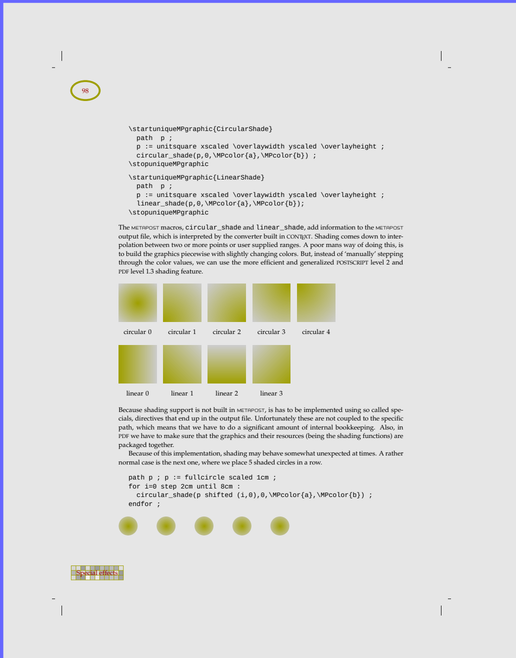
linear 0 linear 1 linear 2 linear 3

Because shading support is not built in METAPOST, is 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 ;
```

PDFTEX can handle METAPOST output rather well.



98

```
\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 0 linear 1 linear 2 linear 3

Because shading support is not built in METAPOST, is 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 ;
```

Special effects

PDFTEX can handle METAPOST output rather well.

Because PDFTEX gives you access to PDF datastructures, even nasty METAPOST trickery can be supported.

This is just a dummy text, kerned by TEX and typeset in a circle using METAPOST.

Donald Knuth has spent the past several years working on a system allowing him to control many aspects of the design of his forthcoming books—from the typesetting and layout down to the very shapes of the letters! Seldom has an author had anything remotely like this power to control the final appearance of his or her work. Knuth's TEX typesetting system has become well–known and available in many countries around the world. By contrast, his METAFONT system for designing families of typefaces has not become as well known or available.

In his article "The Concept of a Meta–Font", Knuth sets forth for the

Figure 9.4 One more time Hofstadter's quotation.

9.6 Random graphics

Given enough time and paper, we can probably give you some

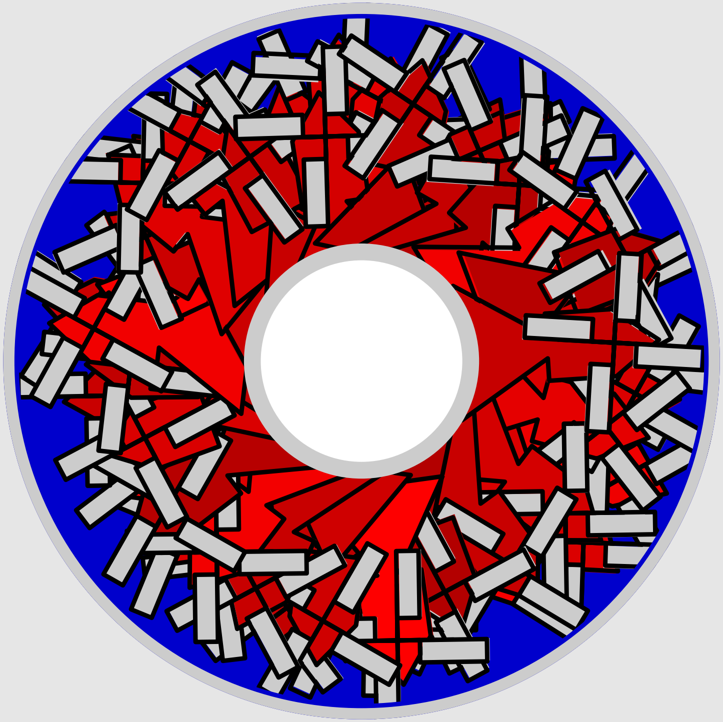
reasons why METAPOST is fun. To mention a few: you can enhance the layout with graphic ornaments, you can tune your graphics at runtime, and simple high quality graphics can be very effective.

The previous graphics draws exactly 1001 lines in a scratch–numbers–in–a–wall fashion. In 1998, the NTG did an survey among its members, and in the report, we used this fuzzy counter to enhance the rather dull tables.

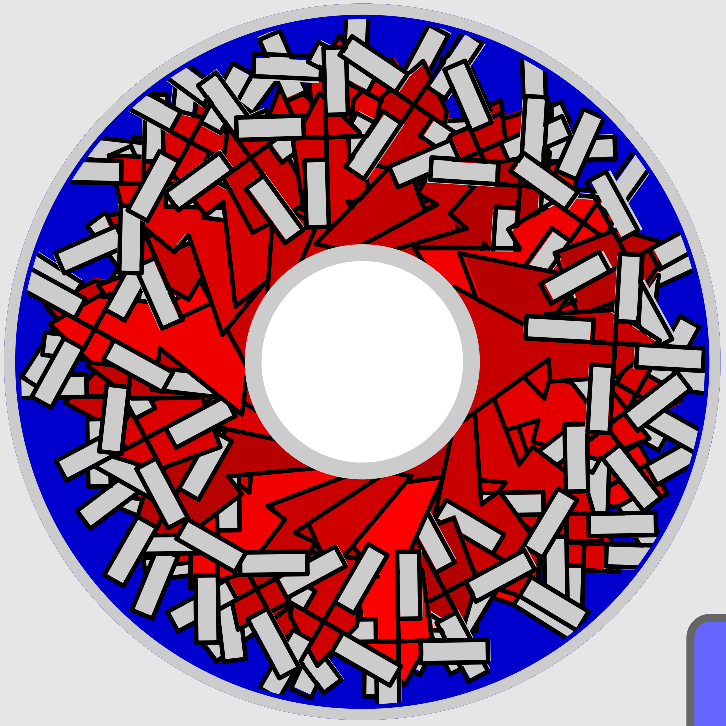
system	%	# users
Atari	10.4	𝍸 𝍸 𝍷
MSDOS	49.1	𝍸 𝍸 𝍸 𝍸 𝍸 𝍸 𝍸 𝍸 𝍸 𝍸 𝍷𝍷
OS/2	9.4	𝍸 𝍸
MacOS	5.7	𝍸 𝍷
UNIX	51.9	𝍸 𝍸 𝍸 𝍸 𝍸 𝍸 𝍸 𝍸 𝍸 𝍸 𝍸
WINDOWS	64.2	𝍸 𝍸 𝍸 𝍸 𝍸 𝍸 𝍸 𝍸 𝍸 𝍸 𝍸 𝍸 𝍸 𝍷𝍷𝍷

Table 9.1 Operating system (n=106).

Because we leave the job to TEX, font inclusion is taken care of very efficiently.



You can use PDFTEX for making stand-alone PDF images of METAPOST graphics or other supported formats.



You can use PDFTEX for making stand-alone PDF images of METAPOST graphics or other supported formats.

You can also launch applications that can show an image in a format not supported by PDF.

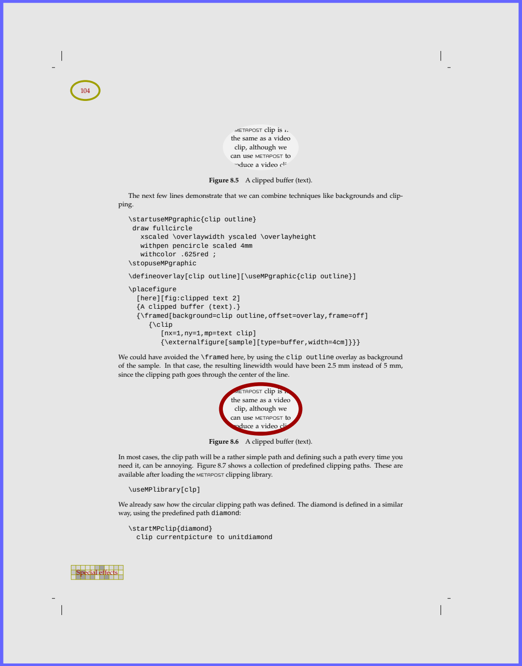
PDFTEX does support **movies**, that themselves are supported by means of plug-ins.

PDFTEX does support **movies**, that themselves are supported by means of plug-ins.

Thereby, you have complete control over the movies **appearance**

In the beginning of the last century, film came without sound.

But today, Acrobat brings you movies without images!


104

METAPOST clip is n… the same as a video clip, although we can use METAPOST to …oduce a video cli…

Figure 8.5 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 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.

METAPOST clip is n… the same as a video clip, although we can use METAPOST to …oduce a video cli…

Figure 8.6 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 8.7 shows a collection of predefined clipping paths. These are available after loading the METAPOST clipping library.

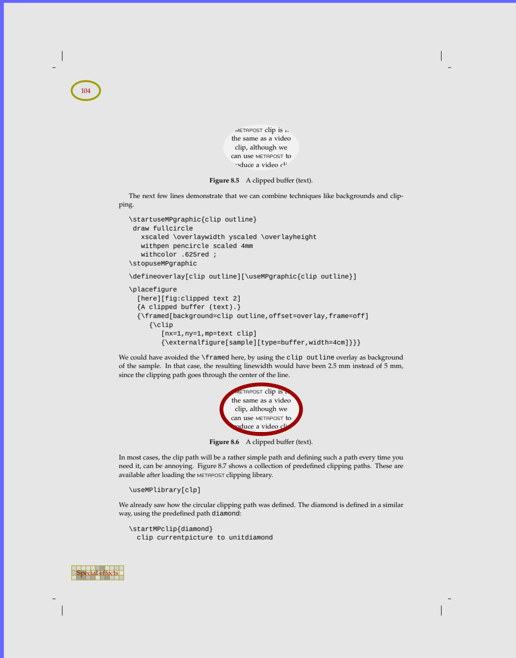
```
\useMPlibrary[clp]
```

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
```

Special effects

Graphic support as described so far can be implemented using the xform, ximage, annotation and literal PDF inclusion.



104

METAPOST clip is n the same as a video clip, although we can use METAPOST to roduce a video cli

Figure 8.5 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 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.

METAPOST clip is n the same as a video clip, although we can use METAPOST to roduce a video cli

Figure 8.6 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 8.7 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
```

Special effects

Graphic support as described so far can be implemented using the xform, ximage, annotation and literal PDF inclusion.

Using literal PDF permits you to play around with graphics and apply clipping, scaling, rotation and more.

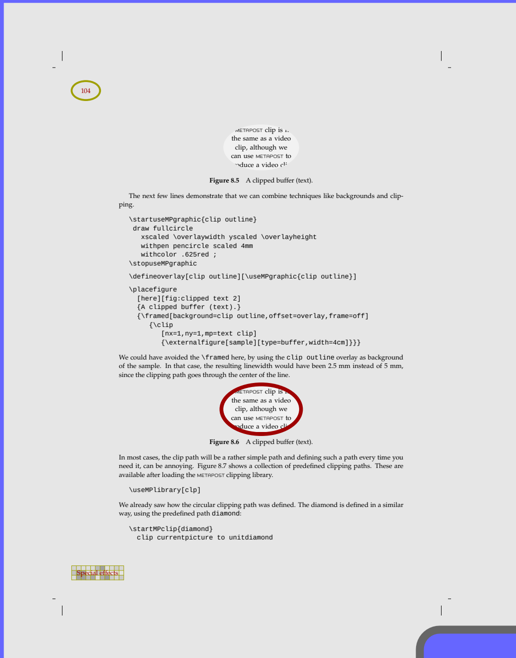


Figure 8.5 A clipped buffer (text).

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 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 8.6 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 8.7 shows a collection of predefined clipping paths. These are available after loading the METAPOST clipping library.

```
\useMPlibrary[clp]
```

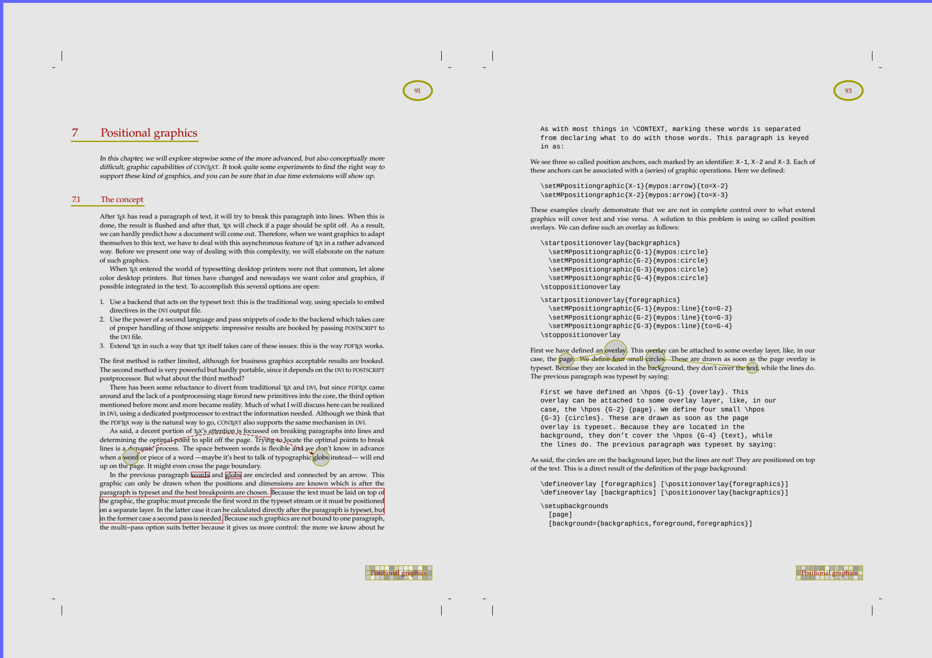
We already saw how the circular clipping path was defined. The diamond is defined in a similar way, using the predefined path `diamond`:

```
\startMPclip{diamond}
  clip currentpicture to unitdiamond
```


Graphic support as described so far can be implemented using the xform, ximage, annotation and literal PDF inclusion.

Using literal PDF permits you to play around with graphics and apply clipping, scaling, rotation and more.

Certain specials as supported by DVI drivers can be supported by PDFTEX too, like TPIC and EMTEX specials.



91

7 Positional graphics

In this chapter, we will explore stepwise some of the more advanced, but also conceptually more difficult, graphic capabilities of CONTEXT. It took quite some 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.

7.1 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, if possible 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: 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 works.

The first method is rather limited, although for business graphics acceptable results are booked. 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 since PDFTEX came around and the lack of a postprocessing stage forced new primitives into the core, the third option mentioned before more and more became reality. Much of what I will discuss here can be realized in DVI, using a dedicated postprocessor to extract the information needed. Although we think that the PDFTEX way is the natural way to go, CONTEXT also supports the same mechanism in DVI.

As said, a decent portion of TEX's attention is focussed 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 words and globs are encircled and connected by an arrow. This graphic can only be drawn when the positions and dimensions are known which is 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

Positional graphics

93

```
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 so called 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 are not in complete control over to what extend graphics will cover text and vise versa. A solution to this problem is using so called 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:
```

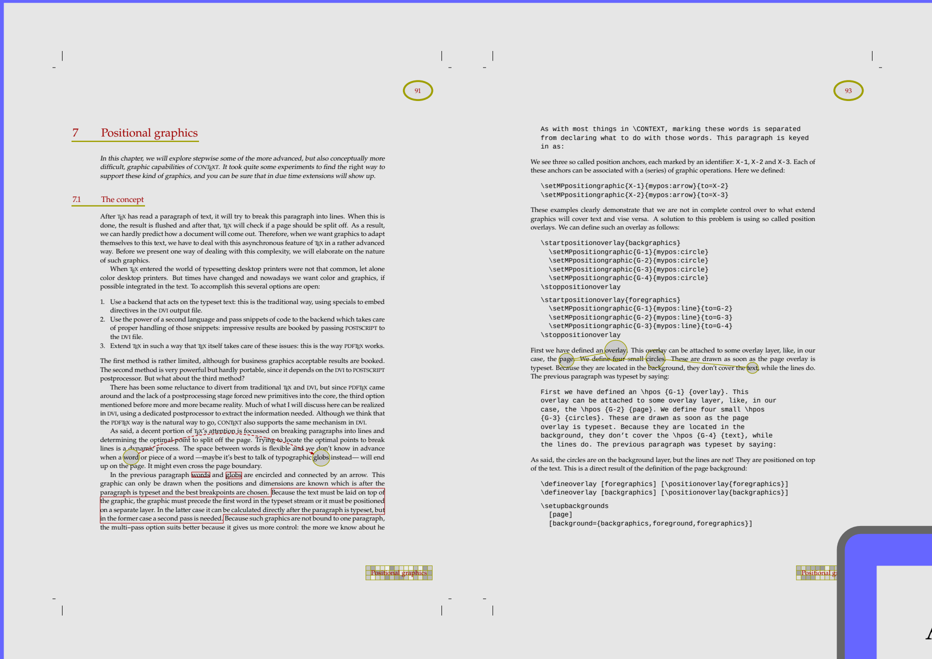
As said, the circles are on the background layer, but the lines are not! They are positioned on top of the text. This is a direct result of the definition of the page background:

```
\defineoverlay [foregraphics] [\positionoverlay{foregraphics}]
\defineoverlay [backgraphics] [\positionoverlay{backgraphics}]

\setupbackgrounds
  [page]
  [background={backgraphics,foreground,foregraphics}]
```

Positional graphics

Although still officially experimental, PDFTEX can provide positional information, which permits you to go even further with embedding graphics.



91

7 Positional graphics

In this chapter, we will explore stepwise some of the more advanced, but also conceptually more difficult, graphic capabilities of CONTEXT. It took quite some 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.

7.1 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, if possible 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: 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 works.

The first method is rather limited, although for business graphics acceptable results are booked. 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 since PDFTEX came around and the lack of a postprocessing stage forced new primitives into the core, the third option mentioned before more and more became reality. Much of what I will discuss here can be realized in DVI, using a dedicated postprocessor to extract the information needed. Although we think that the PDFTEX way is the natural way to go, CONTEXT also supports the same mechanism in DVI.

As said, a decent portion of TEX's attention is focussed 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 words and globs are encircled and connected by an arrow. This graphic can only be drawn when the positions and dimensions are known which is 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

Positional graphics

93

```
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 so called 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 are not in complete control over to what extend graphics will cover text and vise versa. A solution to this problem is using so called 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:

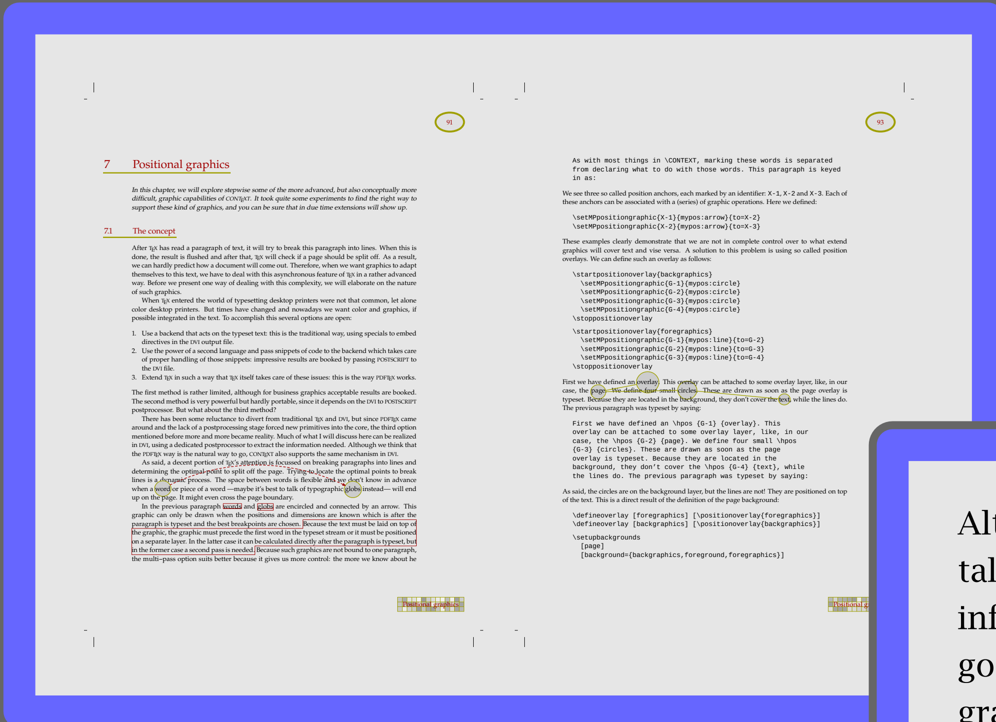
```
\defineoverlay [foregraphics] [\positionoverlay{foregraphics}]
\defineoverlay [backgraphics] [\positionoverlay{backgraphics}]

\setupbackgrounds
  [page]
  [background={backgraphics,foreground,foregraphics}]
```

Positional g

Although still officially experimental, PDFTEX can provide positional information, which permits you to go even further with embedding graphics.

Although this can also be done using DVI, the PDFTEX method is smoother, as well as more comfortable.



91

7 Positional graphics

In this chapter, we will explore stepwise some of the more advanced, but also conceptually more difficult, graphic capabilities of CONTEXT. It took quite some 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.

7.1 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, if possible 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: 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 works.

The first method is rather limited, although for business graphics acceptable results are booked. 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 since PDFTEX came around and the lack of a postprocessing stage forced new primitives into the core, the third option mentioned before more and more became reality. Much of what I will discuss here can be realized in DVI, using a dedicated postprocessor to extract the information needed. Although we think that the PDFTEX way is the natural way to go, CONTEXT also supports the same mechanism in DVI.

As said, a decent portion of TEX's attention is focussed 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 words and globs are encircled and connected by an arrow. This graphic can only be drawn when the positions and dimensions are known which is 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

Positional graphics

93

```
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 so called 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 are not in complete control over to what extend graphics will cover text and vise versa. A solution to this problem is using so called 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}]
```

Positional g

Although still officially experimental, PDFTEX can provide positional information, which permits you to go even further with embedding graphics.

Although this can also be done using DVI, the PDFTEX method is smoother, as well as more comfortable.

More details on this can be found in the upcoming MetaFun manual.