

CONTEXT

METAPOST to PDF conversion

group: CONTEXT Support Macros

version: 1997.05.21

date: 1997 July 25

author: Hans Hagen

copyright: PRAGMA / Hans Hagen & Ton Otten

These macros are written as generic as possible. Some general support macro's are loaded from a small module especially made for non `CONTEXT` use. In this module I use a matrix transformation macro written by Tanmoy Bhattacharya. Thanks to extensive testing of Sebastian Ratz I was able to complete this module within reasonable time. First we take care of non-`CONTEXT` use:

```
1 \ifx \undefined \writestatus \input supp-mis.tex \relax \fi

This module handles some PDF conversion and insertions topics. The macros use the PDFTEX primitive
\pdfliteral.

2 \writestatus{loading}{Context Support Macros / PDF}

3 \unprotect

4 \ifx\pdfliteral\undefined
  \def\pdfliteral#1{\message{[ignored pdfliteral: #1]}}
\fi
```

`\convertPDFt..`

PDFTEX supports verbatim inclusion of PDF code. The following macro takes care of inserting externally defined illustrations in PDF format. According to a suggestion Tanmoy Bhattacharya posted to the PDFTEX mailing list, we first skip lines until `stream` is reached and then copy lines until `endstream` is encountered. This scheme only works with vectorized graphics in which no indirect references to objects are used. Bitmaps also don't work. Interpreting their specifications is beyond the current implementation.

```
\convertPDFtoPDF
  {filename}
  {x scale} {y scale}
  {x offset } {y offset}
  {width} {height}
```

When the scales are set to 1, the last last four values are the same as the bounding box, e.g.

```
\convertPDFtoPDF{mp-pra-1.pdf} {1} {1}{-1bp}{-1bp}{398bp}{398bp}
\convertPDFtoPDF{mp-pra-1.pdf}{.5}{.5} {0bp} {0bp}{199bp}{199bp}
```

Keep in mind, that this kind of copying only works for pure and valid pdf code (without fonts).

The scanning and copying is straightforward and quite fast. To speed up things we use two constants.

```
5 \def\@@PDFstream@@ {stream}
\def\@@PDFendstream@@ {endstream}
```

`\PDFmediabox..`

If needed, the macros can scan for the mediabox that specifies the dimensions and offsets of the graphic. When we say:

```
\PDFmediaboxpreferredtrue
```

the mediabox present in the file superseded the user specified, already scaled and calculated offset and dimensions. Beware: the user supplied values are not the bounding box ones!

```
6 \newif\ifPDFmediaboxpreferred

7 \def\setPDFboundingbox#1#2#3#4#5#6%
  {\dimen0=#1\dimen0=#5\dimen0
   \ScaledPointsToBigPoints{\number\dimen0}\PDFxoffset
   \dimen0=#3\dimen0=#5\dimen0
   \xdef\PDFwidth{\the\dimen0}%
```

```

\dimen0=#2\dimen0=#6\dimen0
\ScaledPointsToBigPoints{\number\dimen0}\PDFyoffset
\dimen0=#4\dimen0=#6\dimen0
\xdef\PDFheight{\the\dimen0}%
\global\let\PDFxoffset=\PDFxoffset
\global\let\PDFyoffset=\PDFyoffset}

8 \def\setPDFmediabox#1[#2 #3 #4 #5]#6\done%
  {\dimen2=#2bp\dimen2=-\dimen2
   \dimen4=#3bp\dimen4=-\dimen4
   \dimen6=#4bp\advance\dimen6 by \dimen2
   \dimen8=#5bp\advance\dimen8 by \dimen4
   \setPDFboundingbox{\dimen2}{\dimen4}{\dimen6}{\dimen8}\PDFxscale\PDFyscale}

9 \def\checkPDFmediabox#1/MediaBox#2#3\done%
  {\ifx#2\relax \else
   \message{mediabox}%
   \setPDFmediabox#2#3\done
  \fi}

```

We use the general macro `\doprocessfile` and feed this with a line handling macro that changed it's behavior when the stream operators are encountered.

```

10 \def\handlePDFline%
    {\ifx\@@PDFstream@@\fileline
     \let\doprocessPDFline=\copyPDFobject
     \startPDFtoPDF
    \else\ifPDFmediaboxpreferred
     \expandafter\checkPDFmediabox\fileline/MediaBox\relax\done
    \fi\fi}

11 \def\copyPDFobject%
    {\ifx\@@PDFendstream@@\fileline
     \ifPDFmediaboxpreferred
     \let\doprocessPDFline=\findPDFmediabox
    \else
     \let\doprocessPDFline=\relax
    \fi
    \else
     \advance\scratchcounter by 1
     \pdfliteral{\fileline}%
    \fi}

12 \def\findPDFmediabox%
    {\expandafter\checkPDFmediabox\fileline/MediaBox\relax\done}

```

The main conversion macro wraps the PDF codes in a box that is output as an object. The graphics are embedded in `q` and `Q` and are scaled and positioned using one transform call (`cm`). This saves some additional scaling.

```

13 \def\startPDFtoPDF%
    {\setbox0=\vbox\bgroup
     \message{[PDF to PDF \PDFfilename]}%
     \forgetall
     \scratchcounter=0

```

```

\let\stopPDFtoPDF=\dostopPDftoPDF}

14 \def\dostopPDFtoPDF%
    {\ifnum\scratchcounter<0 \scratchcounter=1 \fi
     \message{(\the\scratchcounter\space lines)}}%
    \egroup
    \wd0=\PDFwidth
    \vbox to \PDFheight
        {\forgetall
         \vfill
         \pdfliteral{q}%
         \pdfliteral{1 0 0 1 \PDFxoffset\space \PDFyoffset\space cm}%
         \pdfliteral{\PDFxscale\space 0 0 \PDFyscale\space 0 0 cm}%
         \box0
         \pdfliteral{Q}}}

15 \def\stopPDFtoPDF%
    {\message{[PDF to PDF \PDFfilename\space not found]}}

16 \def\convertPDFtoPDF#1#2#3#4#5#6#7%
    {\bgroup
     \def\PDFfilename{#1}%
     \def\PDFxscale {#2}%
     \def\PDFyscale {#3}%
     \setPDFboundingbox{#4}{#5}{#6}{#7}{1}{1}%
     \uncatcodespecials
     \endlinechar=-1
     \let\doprocessPDFline=\handlePDFline
     \doprocessfile\scratchread\PDFfilename\doprocessPDFline
     \stopPDFtoPDF
    \egroup}

```

The next set of macros implements METAPost to PDF conversion. Because we want to test as fast as possible, we first define the POSTSCRIPT operators that METAPost uses. We don't define irrelevant ones, because these are skipped anyway.

```

17 \def \PScurveto      {curveto}
    \def \PSlineto      {lineto}
    \def \PSmoveto      {moveto}
    \def \PSshowpage    {showpage}
    \def \PSnewpath     {newpath}
    \def \PSfshow       {fshow}
    \def \PSclosepath   {closepath}
    \def \PSfill        {fill}
    \def \PSstroke      {stroke}
    \def \PSclip        {clip}
    \def \PSrlineto     {rlineto}
    \def \PSsetlinejoin {setlinejoin}
    \def \PSsetlinecap  {setlinecap}
    \def \PSsetmiterlimit {setmiterlimit}
    \def \PSsetgray     {setgray}
    \def \PSsetrgbcolor {setrgbcolor}
    \def \PSsetdash     {setdash}
    \def \PSgsave       {gsave}

```

```

\def \PSgrestore      {grestore}
\def \PStranslate     {translate}
\def \PSScale         {scale}
\def \PSconcat        {concat}
\def \PSdtransform    {dtransform}
18 \def \PSBoundingBox {BoundingBox:}
\def \PSHiResBoundingBox {HiResBoundingBox:}
\def \PSExactBoundingBox {ExactBoundingBox:}
\def \PSPage          {Page:}

```

In POSTSCRIPT arguments precede the operators. Due to the fact that in some translations we need access to those arguments, as well as that sometimes we have to skip them, we stack them up. The stack is one-dimensional for non path operators and two-dimensional for operators inside a path. This is because we have to save the whole path for (optional) postprocessing. Values are pushed onto the stack by:

```
\setMPargument {value}
```

They can be retrieved by the short named macros:

```

\gMPa {number}
\sMPa {number}

```

When scanning a path specification, we also save the operator, using

```
\setMPkeyword {n}
```

The path drawing operators are coded for speed: `clip`, `stroke`, `fill` and `fillstroke` become 1, 2, 3 and 4.

When processing the path this code can be retrieved using

```
\getMPkeyword{n}
```

When setting an argument, the exact position on the stack depend on the current value of the *counters* `\nofMPsegments` and `\nofMParguments`.

```

19 \newcount\nofMPsegments
\newcount\nofMParguments

```

These variables hold the coordinates. The argument part of the stack is reset by:

```
\resetMPstack
```

We use the prefix `@@MP` to keep the stack from conflicting with existing macros. To speed up things bit more, we use the constant `\@@MP`.

```

20 \def\@@MP{\@@MP}
21 \def\setMPargument#1%
    {\advance\nofMParguments by 1
    \expandafter\def
    \csname\@@MP\the\nofMPsegments\the\nofMParguments\endcsname%
    {\do#1}}
22 \def\gMPa#1%
    {\csname\@@MP0#1\endcsname}

```

```

23 \def\gMPs#1%
    {\csname\@MP\the\nofMPsegments#1\endcsname}

24 \def\setMPkeyword#1
    {\expandafter\def\csname\@MP\the\nofMPsegments0\endcsname{#1}%
     \advance\nofMPsegments by 1
     \nofMParguments=0\relax}

25 \def\getMPkeyword#1%
    {\csname\@MP#10\endcsname}

```

When we reset the stack, we can assume that all further comment is to be ignored as well as handled in strings. By redefining the reset macro after the first call, we save some run time.

```

26 \def\resetMPstack%
    {\catcode'\%=\@active
     \let\handleMPgraphic=\handleMPendgraphic
     \def\resetMPstack{\nofMParguments=0\relax}%
     \resetMPstack}

```

The arguments are saved with the preceding command `\do`. By default this command expands to nothing, but when we deal with strings it's used to strip off the (and).

Strings are kind of tricky, because characters can be passed verbatim (`hello`), by octal number (`\005`) or as command (`\()`). We therefore cannot simply ignore (and), the way we do with [and]. Another complication is that strings may contain characters that normally have a special meaning in \TeX , like \$ and {}.

A previous solution made \ an active character and let it look ahead for a number or character. We had to abandon this scheme because of the need for verbatim support. The next solution involved some *catcode* trickery but works well.

```

27 \def\octalMPcharacter#1#2#3%
    {\char'#1#2#3\relax}

28 \bgroup
    \catcode'\|=\@comment
    \catcode'\%=\@active
    \catcode'\[=\@active
    \catcode'\]=\@active
    \catcode'\{=\@active
    \catcode'\}=\@active
    \catcode'B=\@begingroup
    \catcode'E=\@endgroup
    \gdef\ignoreMPspecials|
        B\def%BE|
        \def[BE|
        \def]BE|
        \def{BE|
        \def}BEE
    \gdef\obeyMPspecials|
        B\def%B\char 37\relax E|
        \def[B\char 91\relax E|
        \def]B\char 93\relax E|
        \def{B\char123\relax E|

```

```

\def}B\char125\relax EE
\gdef\setMPspecials|
  B\catcode'\%=\@@active
  \catcode'\[=\@@active
  \catcode'\]=\@@active
  \catcode'\{=\@@active
  \catcode'\}=\@@active
  \catcode'\$=\@@letter
  \catcode'\_=\@@letter
  \catcode'\#=\@@letter
  \catcode'\^=\@@letter
  \catcode'\&=\@@letter
  \catcode'\|=\@@letter
  \catcode'\~=\@@letter
\def\B\char40\relax E|
\def\B\char41\relax E|
\def\B\char92\relax E|
\def\0B\octalMPcharacter0E|
\def\1B\octalMPcharacter1E|
\def\2B\octalMPcharacter2E|
\def\3B\octalMPcharacter3E|
\def\4B\octalMPcharacter4E|
\def\5B\octalMPcharacter5E|
\def\6B\octalMPcharacter6E|
\def\7B\octalMPcharacter7E|
\def\8B\octalMPcharacter8E|
\def\9B\octalMPcharacter9EE
\egroup

```

We use the comment symbol as a sort of trigger:

```

29 \bgroup
   \catcode'\%=\@@active
   \gdef\startMPscanning{\let%=\startMPconversion}
\egroup

```

In earlier versions we used the sequence

```
\expandafter\handleMPsequence\input filename\relax
```

Persistent problems in L^AT_EX however forced us to use a different scheme. Every POSTSCRIPT file starts with a %, so we temporary make this an active character that starts the scanning and redefines itself. (The problem originates in the redefinition by L^AT_EX of the \input primitive.)

```

30 \def\startMPconversion%
   {\catcode'\%=\@@ignore
    \ignoreMPspecials
    \handleMPsequence}

```

Here comes the main loop. Most arguments are numbers. This means that they can be recognized by their \lccode. This method saves a lot of processing time. We could speed up the conversion by handling the path separately.

```

31 \def\dohandleMPsequence#1#2 %
   {\ifnum\lccode'#1=0

```



```

\setMPargument{#1#2}%
\else
\edef\somestring{#1#2}%
\ifx\somestring\PSmoveto
\edef\lastMPmoveX{\gMPa1}%
\edef\lastMPmoveY{\gMPa2}%
\pdfliteral{\gMPa1 \gMPa2 m}%
\resetMPstack
\else\ifx\somestring\PSnewpath
\let\handleMPsequence=\handleMPpath
\else\ifx\somestring\PSgsave
\pdfliteral{q}%
\resetMPstack
\else\ifx\somestring\PSgrestore
\pdfliteral{Q}%
\resetMPstack
\else\ifx\somestring\PSdtransform % == setlinewidth
\let\handleMPsequence=\handleMPdtransform
\else\ifx\somestring\PSconcat
\pdfliteral{\gMPa1 \gMPa2 \gMPa3 \gMPa4 \gMPa5 \gMPa6 cm}%
\resetMPstack
\else\ifx\somestring\PSsetrgbcolor
\pdfliteral{\gMPa1 \gMPa2 \gMPa3 rg \gMPa1 \gMPa2 \gMPa3 RG}%
\resetMPstack
\else\ifx\somestring\PSsetgray
\pdfliteral{\gMPa1 g \gMPa1 G}%
\resetMPstack
\else\ifx\somestring\PStranslate
\pdfliteral{1 0 0 1 \gMPa1 \gMPa2 cm}%
\resetMPstack
\else\ifx\somestring\PSsetdash
\handleMPsetdash
\resetMPstack
\else\ifx\somestring\PSsetlinejoin
\pdfliteral{\gMPa1 j}%
\resetMPstack
\else\ifx\somestring\PSsetmiterlimit
\pdfliteral{\gMPa1 M}%
\resetMPstack
\else\ifx\somestring\PSfshow
\handleMPfshow
\resetMPstack
\else\ifx\somestring\PSsetlinecap
\pdfliteral{\gMPa1 J}%
\resetMPstack
\else\ifx\somestring\PSrlineto
\pdfliteral{\lastMPmoveX\space \lastMPmoveY\space l S}%
\resetMPstack
\else\ifx\somestring\PSscale
\pdfliteral{\gMPa1 0 0 \gMPa2 0 0 cm}%
\resetMPstack
\else
\handleMPgraphic{#1#2}%

```

```

\fi\fi\fi\fi\fi\fi\fi\fi
\fi\fi\fi\fi\fi\fi\fi\fi
\fi
\handleMPsequence}

```

Beginning and ending the graphics is taken care of by the macro `\handleMPgraphic`, which is redefined when the first graphics operator is met.

```

32 \def\handleMPendgraphic#1%
    {\ifx\somestring\PSshowpage
      \let\handleMPsequence=\finishMPgraphic
    \else
      \setMPargument{#1}%
    \fi}

33 \def\handleMPbegingraphic#1%
    {\ifx\somestring\PSBoundingBox
      \let\handleMPsequence=\handleMPboundingbox
    \else\ifx\somestring\PSHiResBoundingBox
      \let\handleMPsequence=\handleMPboundingbox
    \else\ifx\somestring\PSExactBoundingBox
      \let\handleMPsequence=\handleMPboundingbox
    \else\ifx\somestring\PSPage
      \let\handleMPsequence=\handleMPpage
    \else
      \setMPargument{#1}%
    \fi\fi\fi\fi}

34 \let\handleMPgraphic=\handleMPbegingraphic

```

We check for three kind of bounding boxes: the normal one and two high precision ones:

```

BoundingBox: llx lly ucx ucy
HiResBoundingBox: llx lly ucx ucy
ExactBoundingBox: llx lly ucx ucy

```

The dimensions are saved for later use.

```

35 \def\handleMPboundingbox #1 #2 #3 #4
    {\dimen0=#1pt\dimen0=-\MPxscale\dimen0
      \dimen2=#2pt\dimen2=-\MPyscale\dimen2
      \xdef\MPxoffset{\withoutpt{\the\dimen0}}%
      \xdef\MPyoffset{\withoutpt{\the\dimen2}}%
      \dimen0=#1bp\dimen0=-\dimen0
      \dimen2=#2bp\dimen2=-\dimen2
      \advance\dimen0 by #3bp
      \dimen0=\MPxscale\dimen0
      \xdef\MPwidth{\the\dimen0}%
      \advance\dimen2 by #4bp
      \dimen2=\MPyscale\dimen2
      \xdef\MPheight{\the\dimen2}%
      \nofMParguments=0
      \let\handleMPsequence=\dohandleMPsequence
      \handleMPsequence}

```

We use the `page` comment as a signal that stackbuilding can be started.

```

36 \def\handleMPpage #1 #2
    {\nofMParguments=0
     \let\handleMPsequence=\dohandleMPsequence
     \handleMPsequence}

```

METAPOST draws its dots by moving to a location and invoking `0 0 rlineto`. This operator is not available in PDF. Our solution is straightforward: we draw a line from $(current_x, current_y)$ to itself. This means that the arguments of the preceding `moveto` have to be saved.

```

37 \def\lastMPmoveX{0}
    \def\lastMPmoveY{0}

```

These saved coordinates are also used when we handle the texts. Text handling proved to be a bit of a nuisance, but finally I saw the light. It proved that we also had to take care of `(split arguments)`.

```

38 \def\handleMPfshow%
    {\setbox0=\hbox
     {\obeyMPspecials
      \edef\size{\gMPa{\the\nofMParguments} }%
      \advance\nofMParguments by -1
      \font\temp=\gMPa{\the\nofMParguments} at \size bp
      \advance\nofMParguments by -1
      \temp
      \ifnum\nofMParguments=1
        \def\do{##1}{##1}%
        \gMPa1%
      \else
        \scratchcounter=1
        \def\do{##1}{##1}%
        \gMPa{\the\scratchcounter}\space
        \def\do{}%
        \loop
          \advance\scratchcounter by 1
          \ifnum\scratchcounter<\nofMParguments
            \gMPa{\the\scratchcounter}\space
          \repeat
        \def\do{##1}{##1}%
        \gMPa{\the\scratchcounter}%
      \fi
      \unskip}%
     \dimen0=\lastMPmoveY bp
     \advance\dimen0 by \ht0
     \ScaledPointsToBigPoints{\number\dimen0}\lastMPmoveY
     \pdfliteral{n q 1 0 0 1 \lastMPmoveX\space\lastMPmoveY\space cm}%
     \dimen0=\ht0
     \advance\dimen0 by \dp0
     \box0
     \vskip-\dimen0
     \pdfliteral{Q}}

```

Most operators are just converted and keep their arguments. Dashes however need a bit different treatment, otherwise PDF viewers complain loudly. Another complication is that one argument comes after the `]`. When reading the data, we simply ignore the array boundary characters. We save ourselves some redundant newlines and at the same time keep the output readable by packing the literals.

```

39 \def\handleMPsetdash%
    {\bgroup
     \def\somestring{[]}%
     \scratchcounter=1
     \loop
     \ifnum\scratchcounter<\nofMParguments
       \edef\somestring{\somestring\space\gMPa{\the\scratchcounter}}%
       \advance\scratchcounter by 1
     \repeat
     \edef\somestring{\somestring]\gMPa{\the\scratchcounter} d}%
     \pdfliteral{\somestring}%
    \egroup}

```

The `setlinewidth` commands look a bit complicated. There are two alternatives, that always look the same. As John Hobby says:

```

x 0 dtransform exch truncate exch idtransform pop setlinewidth
0 y dtransform truncate idtransform setlinewidth pop

```

These are just fancy versions of `x setlinewidth` and `y setlinewidth`. The `x 0 ...` form is used if the path is *primarily vertical*. It rounds the width so that vertical lines come out an integer number of pixels wide in device space. The `0 y ...` form does the same for paths that are *primarily horizontal*. The reason why I did this is Knuth insists on getting exactly the widths \TeX intends for the horizontal and vertical rules in `btex...etex` output. (Note that PostScript scan conversion rules cause a horizontal or vertical line of integer width n in device space to come out $n + 1$ pixels wide, regardless of the phase relative to the pixel grid.)

The common operator in these sequences is `dtransform`, so we can use this one to trigger setting the linewidth.

```

40 \def\handleMPdtransform%
    {\ifdim\gMPa1pt>\!zeropoint
     \pdfliteral{\gMPa1 w}%
     \def\next##1 ##2 ##3 ##4 ##5 ##6 {\handleMPsequence}%
    \else
     \pdfliteral{\gMPa2 w}%
     \def\next##1 ##2 ##3 ##4 {\handleMPsequence}%
    \fi
    \let\handleMPsequence=\dohandleMPsequence
    \resetMPstack
    \next}

```

The most complicated command is `concat`. METAPOST applies this operator to `stoke`. At that moment the points set by `curveto` and `moveto`, are already fixed. In PDF however the `cm` operator affects the points as well as the pen (stroke). Like more PDF operators, `cm` is defined in a bit ambiguous way. The only save route for non-circular penshapes, is saving the path, recalculating the points and applying the transformation matrix in such a way that we can be sure that its behavior is well defined. This comes down to inverting the path and applying `cm` to that path as well as the pen. This all means that we have to save the path.

In METAPOST there are three ways to handle a path p :

```
draw p; fill p; filldraw p;
```

The last case outputs a `gsave fill grestore` before `stroke`. Handling the path outside the main loops saves about 40% run time.¹ Switching between the main loop and the path loop is done by means of the recursively called macro `\handleMPsequence`.

```
41 \def\handleMPpath%
    {\chardef\finiMPpath=0
     \let\closeMPpath=\relax
     \let\flushMPpath=\flushnormalMPpath
     \resetMPstack
     \nofMPsegments=1
     \let\handleMPsequence=\dohandleMPpath
     \dohandleMPpath}
```

Most paths are drawn with simple round pens. Therefore we've split up the routine in two.

```
42 \def\flushnormalMPpath%
    {\scratchcounter=\nofMPsegments
     \nofMPsegments=1
     \loop
       \expandafter\ifcase\getMPkeyword{\the\nofMPsegments}\relax
       \pdfliteral{\gMPs1 \gMPs2 l}%
     \or
       \pdfliteral{\gMPs1 \gMPs2 \gMPs3 \gMPs4 \gMPs5 \gMPs6 c}%
     \or
       \pdfliteral{\lastMPmoveX\space \lastMPmoveY\space 1 S}%
     \or
       \edef\lastMPmoveX{\gMPs1}%
       \edef\lastMPmoveY{\gMPs2}%
       \pdfliteral{\lastMPmoveX\space \lastMPmoveY\space m}%
     \fi
     \advance\nofMPsegments by 1\relax
     \ifnum\nofMPsegments<\scratchcounter
       \repeat}

43 \def\flushconcatMPpath%
    {\scratchcounter=\nofMPsegments
     \nofMPsegments=1
     \loop
       \expandafter\ifcase\getMPkeyword{\the\nofMPsegments}\relax
       \doMPconcat{\gMPs1}\a{\gMPs2}\b
       \pdfliteral{\a\space \b\space 1}%
     \or
       \doMPconcat{\gMPs1}\a{\gMPs2}\b
       \doMPconcat{\gMPs3}\c{\gMPs4}\d
       \doMPconcat{\gMPs5}\e{\gMPs6}\f
       \pdfliteral{\a\space \b\space \c\space \d\space \e\space \f\space c}%
     \or
       \bgroup
       \noMPtranslate
       \doMPconcat\lastMPmoveX\a\lastMPmoveY\b
       \pdfliteral{\a\space \b\space 1 S}%
       \egroup
     \or
```

¹ We can save some more by following the METAPOST output routine, but for the moment we keep things simple.

```

\edef\lastMPmoveX{\gMPs1}%
\edef\lastMPmoveY{\gMPs2}%
\doMPconcat\lastMPmoveX\lastMPmoveY\b
\pdfliteral{\a\space \b\space m}%
\fi
\advance\nofMPsegments by 1\relax
\ifnum\nofMPsegments<\scratchcounter
\repeat}

```

The transformation of the coordinates is handled by one of the macros Tanmoy posted to the PDFTEX mailing list. I rewrote and optimized the original macro to suit the other macros in this module.

```
\doMPconcat {x position} \xresult {y position} \yresult
```

By setting the auxiliary *dimensions* `\dimen0` upto `\dimen10` only once per path, we save over 20% run time. Some more speed was gained by removing some parameter passing. These macros can be optimized a bit more by using more constants. There is however not much need for further optimization because penshapes usually are round and therefore need no transformation. Nevertheless we move the factor to the outer level and use bit different `pt` removal macro. Although the values represent base points, we converted them to pure points, simply because those can be converted back.

```

44 \def\MPconcatfactor{256}
45 \def\doMPreducedimen#1
    {\count0=\MPconcatfactor
    \advance\dimen#1 \ifdim\dimen#1>\!!zeropoint .5\else -.5\fi\count0
    \divide\dimen#1 \count0\relax}
46 \def\doMPexpanddimen#1
    {\multiply\dimen#1 \MPconcatfactor\relax}
47 \def\presetMPconcat%
    {\dimen 0=\gMPs1 pt \doMPreducedimen 0 % r_x
    \dimen 2=\gMPs2 pt \doMPreducedimen 2 % s_x
    \dimen 4=\gMPs3 pt \doMPreducedimen 4 % s_y
    \dimen 6=\gMPs4 pt \doMPreducedimen 6 % r_y
    \dimen 8=\gMPs5 pt \doMPreducedimen 8 % t_x
    \dimen10=\gMPs6 pt \doMPreducedimen10 } % t_y
48 \def\noMPtranslate% use this one grouped
    {\dimen 8=\!!zeropoint % t_x
    \dimen10=\!!zeropoint} % t_y
49 \def\doMPconcat#1#2#3#4%
    {\dimen12=#1 pt \doMPreducedimen12 % p_x
    \dimen14=#3 pt \doMPreducedimen14 % p_y
    %
    \dimen16 \dimen 0
    \multiply \dimen16 \dimen 6
    \dimen20 \dimen 2
    \multiply \dimen20 \dimen 4
    \advance \dimen16 -\dimen20
    %
    \dimen18 \dimen12
    \multiply \dimen18 \dimen 6

```

```

\dimen20 \dimen14
\multiply \dimen20 \dimen 4
\advance \dimen18 -\dimen20
\dimen20 \dimen 4
\multiply \dimen20 \dimen10
\advance \dimen18 \dimen20
\dimen20 \dimen 6
\multiply \dimen20 \dimen 8
\advance \dimen18 -\dimen20
%
\multiply \dimen12 -\dimen 2
\multiply \dimen14 \dimen 0
\advance \dimen12 \dimen14
\dimen20 \dimen 2
\multiply \dimen20 \dimen 8
\advance \dimen12 \dimen20
\dimen20 \dimen 0
\multiply \dimen20 \dimen10
\advance \dimen12 -\dimen20
%
\doMPreducedimen16
\divide \dimen18 \dimen16 \doMPexpanddimen18
\divide \dimen12 \dimen16 \doMPexpanddimen12
%
\edef#2{\withoutpt{\the\dimen18}}%      % p_x^\prime
\edef#4{\withoutpt{\the\dimen12}}%      % p_y^\prime

```

The following explanation of the conversion process was posted to the PDFTEX mailing list by Tanmoy. The original macro was part of a set of macro's that included sinus and cosinus calculation as well as scaling and translating. The METAPOST to PDF conversion however only needs transformation.

Given a point (U_x, U_y) in user coordinates, the business of POSTSCRIPT is to convert it to device space. Let us say that the device space coordinates are (D_x, D_y) . Then, in POSTSCRIPT (D_x, D_y) can be written in terms of (U_x, U_y) in matrix notation, either as

$$\begin{pmatrix} D_x & D_y & 1 \end{pmatrix} = \begin{pmatrix} U_x & U_y & 1 \end{pmatrix} \begin{pmatrix} s_x & r_x & 0 \\ r_y & s_y & 0 \\ t_x & t_y & 1 \end{pmatrix} \quad (1)$$

or

$$\begin{pmatrix} D_x \\ D_y \\ 1 \end{pmatrix} = \begin{pmatrix} s_x & r_y & t_x \\ r_x & s_y & t_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} U_x \\ U_y \\ 1 \end{pmatrix} \quad (2)$$

both of which is a shorthand for the same set of equations:

$$D_x = s_x U_x + r_y U_y + t_x \quad (3)$$

$$D_y = r_x U_x + s_y U_y + t_y \quad (4)$$

which define what is called an 'affine transformation'.

POSTSCRIPT represents the 'transformation matrix' as a six element matrix instead of a 3×3 array because three of the elements are always 0, 0 and 1. Thus the above transformation is written in postscript as

$[s_x r_x r_y s_y t_x t_y]$. However, when doing any calculations, it is useful to go back to the original matrix notation (whichever: I will use the second) and continue from there.

As an example, if the current transformation matrix is $[s_x r_x r_y s_y t_x t_y]$ and you say $[a b c d e f]$ concat, this means:

Take the user space coordinates and transform them to an intermediate set of coordinates using array $[a b c d e f]$ as the transformation matrix.

Take the intermediate set of coordinates and change them to device coordinates using array $[s_x r_x r_y s_y t_x t_y]$ as the transformation matrix.

Well, what is the net effect? In matrix notation, it is

$$\begin{pmatrix} I_x \\ I_y \\ 1 \end{pmatrix} = \begin{pmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} U_x \\ U_y \\ 1 \end{pmatrix} \quad (5)$$

$$\begin{pmatrix} D_x \\ D_y \\ 1 \end{pmatrix} = \begin{pmatrix} s_x & r_x & t_x \\ r_y & s_y & t_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} I_x \\ I_y \\ 1 \end{pmatrix} \quad (6)$$

where (I_x, I_y) is the intermediate coordinate.

Now, the beauty of the matrix notation is that when there is a chain of such matrix equations, one can always compose them into one matrix equation using the standard matrix composition law. The composite matrix from two matrices can be derived very easily: the element in the i^{th} horizontal row and j^{th} vertical column is calculated by 'multiplying' the i^{th} row of the first matrix and the j^{th} column of the second matrix (and summing over the elements). Thus, in the above:

$$\begin{pmatrix} D_x \\ D_y \\ 1 \end{pmatrix} = \begin{pmatrix} s'_x & r'_y & t'_x \\ r'_x & s'_y & t'_y \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} U_x \\ U_y \\ 1 \end{pmatrix} \quad (7)$$

with

$$\begin{aligned} s'_x &= s_x a + r_y b \\ r'_x &= r_x a + s_y b \\ r'_y &= s_x c + r_y d \\ s'_y &= r_x c + s_y d \\ t'_x &= s_x e + r_y f + t_x \\ t'_y &= r_x e + s_y f + t_y \end{aligned} \quad (8)$$

In fact, the same rule is true not only when one is going from user coordinates to device coordinates, but whenever one is composing two 'transformations' together (transformations are 'associative'). Note that the formula is not symmetric: you have to keep track of which transformation existed before (i.e. the equivalent of $[s_x r_x r_y s_y t_x t_y]$) and which was specified later (i.e. the equivalent of $[a b c d e f]$). Note also that the language can be rather confusing: the one specified later 'acts earlier', converting the user space coordinates to intermediate coordinates, which are then acted upon by the pre-existing transformation. The important point is that order of transformation matrices cannot be flipped (transformations are not 'commutative').

Now what does it mean to move a transformation matrix before a drawing? What it means is that given a point (P_x, P_y) we need a different set of coordinates (P'_x, P'_y) such that if the transformation acts on (P'_x, P'_y) , they produce (P_x, P_y) . That is we need to solve the set of equations:

$$\begin{pmatrix} P_x \\ P_y \\ 1 \end{pmatrix} = \begin{pmatrix} s_x & r_y & t_x \\ r_x & s_y & t_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P'_x \\ P'_y \\ 1 \end{pmatrix} \quad (9)$$

Again matrix notation comes in handy (i.e. someone has already solved the problem for us): we need the inverse transformation matrix. The inverse transformation matrix can be calculated very easily: it is

$$\begin{pmatrix} P'_x \\ P'_y \\ 1 \end{pmatrix} = \begin{pmatrix} s'_x & r'_y & t'_x \\ r'_x & s'_y & t'_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ 1 \end{pmatrix} \quad (10)$$

where, the inverse transformation matrix is given by

$$\begin{aligned} D &= s_x s_y - r_x r_y \\ s'_x &= s_y / D \\ s'_y &= s_x / D \\ r'_x &= -r_x / D \\ r'_y &= -r_y / D \\ t'_x &= (-s_y t_x + r_y t_y) / D \\ t'_y &= (r_x t_x - s_x t_y) / D \end{aligned} \quad (11)$$

And you can see that when expanded out, this does give the formulas:

$$P'_x = \frac{s_y(p_x - t_x) + r_y(t_y - p_y)}{s_x * s_y - r_x * r_y} \quad (12)$$

$$P'_y = \frac{s_x(p_y - t_y) + r_x(t_x - p_x)}{s_x * s_y - r_x * r_y} \quad (13)$$

The code works by representing a real number by converting it to a dimension to be put into a $\langle dimension \rangle$ register: 2.3 would be represented as 2.3pt for example. In this scheme, multiplying two numbers involves multiplying the $\langle dimension \rangle$ registers and dividing by 65536. Accuracy demands that the division be done as late as possible, but overflow considerations need early division.

Division involves dividing the two $\langle dimension \rangle$ registers and multiplying the result by 65536. Again, accuracy would demand that the numerator be multiplied (and/or the denominator divided) early: but that can lead to overflow which needs to be avoided.

If nothing is known about the numbers to start with (in concat), I have chosen to divide the 65536 as a 256 in each operand. However, in the series calculating the sine and cosine, I know that the terms are small (because I never have an angle greater than 45 degrees), so I chose to apportion the factor in a different way.

The path is output using the values saved on the stack. If needed, all coordinates are recalculated.

```
50 \def\processMPpath%
    {\flushMPpath
    \closeMPpath
    \pdfliteral{\ifcase\finiMPpath W n\or S\or f\or B\fi}%
    \let\handleMPsequence=\dohandleMPsequence
    \resetMPstack
    \nofMPsegments=0
    \handleMPsequence}
```

This macro interprets the path and saves it as compact as possible.

The main conversion command is

The dimensions are derived from the bounding box. So we only have to say:

CONTEXT Support Macros CONTEXT

```

53 \def\convertMPtoPDF#1#2#3%
    {\bgroup
     \message{[MP to PDF #1]}%
     \setMPspecials
     \startMPscanning
     \def\do{}%
     \edef\MPxscale{#2}%
     \edef\MPyscale{#3}%
     \setbox0=\vbox\bgroup
     \forgetall
     \offinterlineskip
     \pdfliteral{q}%
     \let\handleMPsequence=\dohandleMPsequence
     \input #1\relax}

54 \def\finishMPgraphic%
    {\pdfliteral{Q}%
     \egroup
     \wd0=\MPwidth
     \vbox to \MPheight
     {\forgetall
      \vfill
      \pdfliteral{q \MPxscale\space 0 0 \MPyscale\space
        \MPxoffset\space \MPyoffset\space cm}%
      \box0
      \pdfliteral{Q}}}%
    \egroup}

```

This kind of conversion is possible because METAPOST does all the calculations. Converting other POSTSCRIPT files would drive both me and T_EX crazy.

```

55 \protect \endinput

```

METAPOST to PDF conversion

```
\convertMPtoPDF 3  
\convertPDFtoPDF 1
```

```
\PDFmediaboxprefered 1
```