Puzzling graphics in METAPOST

Hans Hagen – pragma@pi.net

In the beginning

If all stories are true, TEX is widely used among mathematicians and is not that populair among what we call α–oriented people. Maybe that's due to the command like interface of this system, although well designed macro packages could be of great help. Or does mathematical oriented use inhibit transfer to other domains.

When browsing through periodicals of user groups, one seldom finds articles on the use of METAFONT other then for making fonts. How can it be that people sometimes use TEX for non TEXnical purposes (like drawing things and calculating other things) while they oversee METAFONT as a tool that suits many drawing purposes other than fonts. And, every time I see some output, the quality strikes me.

Being no real mathematician I never came to reading the whole METAFONT book. I have generated some fonts, but that was all. Out of frustration about the quality of some drawing packages, I could not resist installing METAPOST, the POSTSCRIPT producing ancester of METAFONT.

However, METAPOST nearly drove me crazy, when I found out that including TEX output is possible, but only in close coorporation with tuned DVI to POSTSCRIPT drivers, unfortunately not the one I used. This forced me to write some TEX hacks first, but finally I was able to produce METAPOST output which included text typeset by TEX. For those who want to know the details: this dependancy concerns the inclusion of fonts. METAPOST expects DVI to POSTSCRIPT converters to sort this out, but one can trick TEX to force this behavior.[1] Another depency surfaced a while later: the macropackage one uses must leave TEX's vertical and horizontal offsets untouched, i.e. at it's initial values of 1 inch.

Visual proofing

Actually using METAPOST was stimulated by an (indeed) mathematical problem someone presented me. It took me just a few minutes to see that a sort of graphical proof was far more easy than some prove with a lot of $\sin(this)$ and $\cos(that)$ and long forgotten formulas.

I was expected to prove that the angle between the lines connecting the mid points of four squares drawn upon the four sides of a box is ninety degrees (see figure)

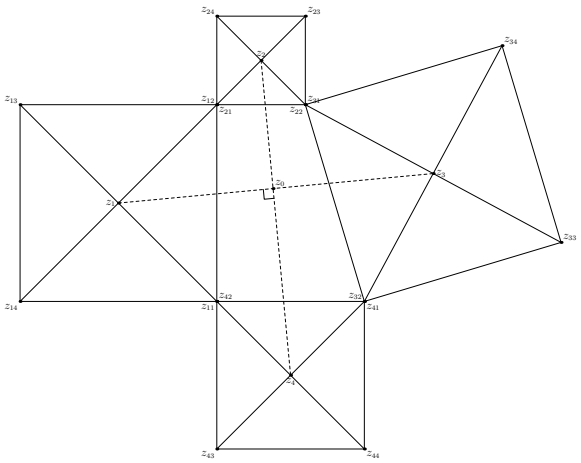


Figure 1 The problem.

[1] The module `m-metapost` handles those font matters.

When I was drawing the extremes of this graphic, I could not resist starting up the computer and setting METAPOST to work. And indeed, I had to find out that programming in this language is not that hard and sometimes is even more fun than programming in TEX.

Looking at the program and the graphics it produces, it looked, to me, that at least in educational settings it should be far more easy to convert people to TEX/METAPOST showing them METAPOST than showing them some bulky and complicated TEX macropackage.

I will explain a few details of the next METAPOST program, which probably holds no beauty for real advanced users. First we enable generation of some prologue material. This is needed when we include typeset objects. I still wonder why this is option not turned on by default; METAPOST files are always very small compared to those generated by drawing packages and the extra overhead is minimal.

```
prologues := 1;
```

For the sake of this article I've added some labels to the points we use. Generating these label however is optional and controled by a boolean:

```
boolean show_labels; show_labels := false;
```

We hide the problem in a macro, which accepts four pair of coordinates that determine the central four sided shape.

```
def draw_problem (expr p, q, r, s ) =
```

Because we want to call this macro more than once, we have to save the locally used values. Instead of declaring local variables, one can hide their use for the outside world. In most cases variables behave global. If we don't save them, the next call will lead to errors due to conflicting equations.

```
  save x, y, a, b, c, d, e, f, g, h;
```

We draw four squares and make use of METAPOST's powerful capability to solve equations. Watch the use of = which means that we just state dependencies. Assignments are realized by :=. The less assignments we use, the less errors we can make.

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

  a = x12 - x11 ; b = y12 - y11 ;

  z11 = (x11,   y11)   ; z12 = (x12,   y12)   ;
  z13 = (x12-b, y12+a) ; z14 = (x11-b, y11+a) ;

  c = x22 - x21 ; d = y22 - y21 ;

  z21 = (x21,   y21)   ; z22 = (x22,   y22)   ;
  z23 = (x22-d, y22+c) ; z24 = (x21-d, y21+c) ;

  e = x32 - x31 ; f = y32 - y31 ;

  z31 = (x31,   y31)   ; z32 = (x32,   y32)   ;
  z33 = (x32-f, y32+e) ; z34 = (x31-f, y31+e) ;

  g  = x42 - x41 ; h  = y42 - y41 ;

  z41 = (x41,   y41)   ; z42 = (x42,   y42)   ;
  z43 = (x42-h, y42+g) ; z44 = (x41-h, y41+g) ;
```

So far for the calculations to be done. It is possible to trace the way METAPOST solves its equations. Very handy indeed! Now we can draw the lines, but first we choose ourselves a decent linethickness.

```
pickup pencircle scaled .5pt;

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

The mid points can be calculated by METAPOST too. The next lines state that those points lay halfway the extremes.

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

We choose a alternative linestyle to show them in full glory.

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

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.

```
z0 = whatever[z1,z3] = whatever[z2,z4];
mark_rt_angle (z1, z0, z2) ;
```

The macro `mark_rt_angle` actually draws the angle symbol.

Next come the labels. This bunch of repetitive macros can surely be writen more compact when we use an auxiliary macro. But manipulating strings and tokens is yet behind my experience and except when writing macro–packages there is no real need for this. To keep things readable we did no optimalize the positioning.

```
if show_labels:
```

We use a bigger pen for drawing the dots.

```
    pickup pencircle scaled 5pt;
```

The next two lines show that TEX can do some typesetting work. We could have done without TEX, but why not go for the best.

```
    verbatimtex \def\Z   #1{$z_{\scriptscriptstyle #1}$}   etex;
    verbatimtex \def\ZZ#1#2{$z_{\scriptscriptstyle #1#2}$} etex;
```

The macro `dotlabel` draws a dot and puts a label. The `btex .. etex` construction moves control to TEX. This data is processed afterwards and the results are used in the next run. Including fonts is a bit tricky, because it's DVI–driver dependant. To prevent problems we secretly include the used glyphs in the TEX source and that way fool our DVI to POSTSCRIPT driver to include them. The (original) source of this text therefore includes:

```
\usemodule[metapost]         % a standard ConTeXt command
\switchtocorps[cmr,12pt,rm]  % we switch to the default font
\UseMetaPostFile{filename}   % a not yet embedded module primitive
\UseMetaPostProofFont{cmr10} % another module primitive
```

This module is independant of CONTEXT and is available for those who need it. So if the labels don't look like z_{12}, the wrong production tools were used. But let's switch back to the labels now:

```
    dotlabel.llft(btex \ZZ 11 etex, z11); dotlabel.ulft(btex \ZZ 12 etex, z12);
    dotlabel.ulft(btex \ZZ 13 etex, z13); dotlabel.llft(btex \ZZ 14 etex, z14);

    dotlabel.lrt (btex \ZZ 21 etex, z21); dotlabel.llft(btex \ZZ 22 etex, z22);
    dotlabel.urt (btex \ZZ 23 etex, z23); dotlabel.ulft(btex \ZZ 24 etex, z24);

    dotlabel.urt (btex \ZZ 31 etex, z31); dotlabel.ulft(btex \ZZ 32 etex, z32);
    dotlabel.urt (btex \ZZ 33 etex, z33); dotlabel.urt (btex \ZZ 34 etex, z34);

    dotlabel.lrt (btex \ZZ 41 etex, z41); dotlabel.urt (btex \ZZ 42 etex, z42);
    dotlabel.llft(btex \ZZ 43 etex, z43); dotlabel.lrt (btex \ZZ 44 etex, z44);

    dotlabel.urt (btex \Z  0  etex, z0);
    dotlabel.lft (btex \Z  1  etex, z1);  dotlabel.top (btex \Z  2  etex, z2);
    dotlabel.rt  (btex \Z  3  etex, z3);  dotlabel.bot (btex \Z  4  etex, z4);
```

And there we are.

```
  fi;

enddef;
```

I copied the meaning of `mark_rt_angle` from the manual and I hope that someday I write such macros myselve.

```
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;
```

We are going to draw a lot of pictures, so we define ourselves an extra macro. This time we hard–code some values. The fractions i and j are responsible for the visual iteration process.

```
def do_draw_problem ( expr n, i, j ) =
  beginfig ( n );
    draw_problem
      ( (400,400), (300,600),
        i[(300,600),(550,800)], j[(400,400),(550,500)] );
  endfig;
enddef;
```

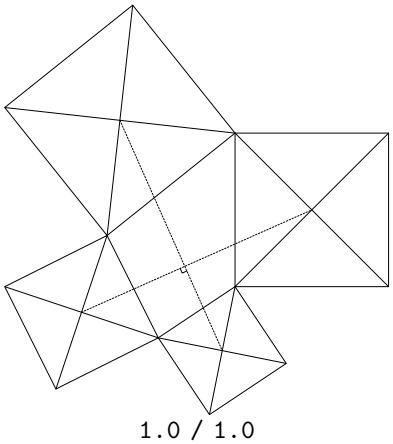
Of course we could have used some loop here, but defining an auxiliary macro probably takes more time than simply calling the drawing macro directly. The results are shown on a separate page.

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.

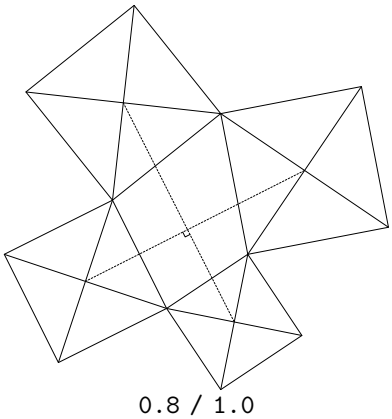
```
do_draw_problem ( 40 , 1.0 , 1.0 ); do_draw_problem ( 41 , 0.8 , 1.0 );
do_draw_problem ( 42 , 0.6 , 1.0 ); do_draw_problem ( 43 , 0.4 , 1.0 );
do_draw_problem ( 44 , 0.2 , 1.0 ); do_draw_problem ( 45 , 0.0 , 1.0 );

do_draw_problem ( 30 , 0.0 , 1.0 ); do_draw_problem ( 31 , 0.0 , 0.8 );
do_draw_problem ( 32 , 0.0 , 0.6 ); do_draw_problem ( 33 , 0.0 , 0.4 );
do_draw_problem ( 34 , 0.0 , 0.2 ); do_draw_problem ( 35 , 0.0 , 0.0 );
```

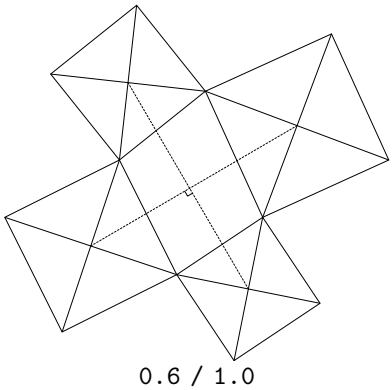
We already showed a picture with coordinates. This picture was generated using some alternative coordinates.



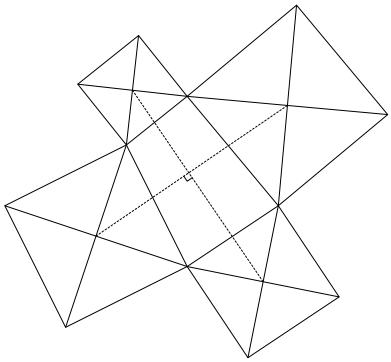
1.0 / 1.0



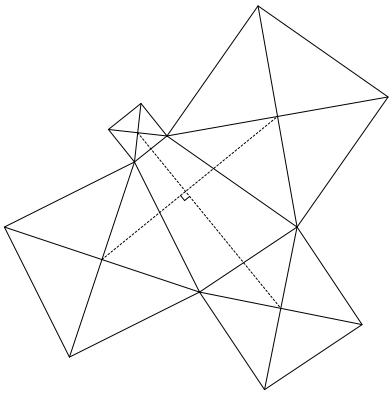
0.8 / 1.0



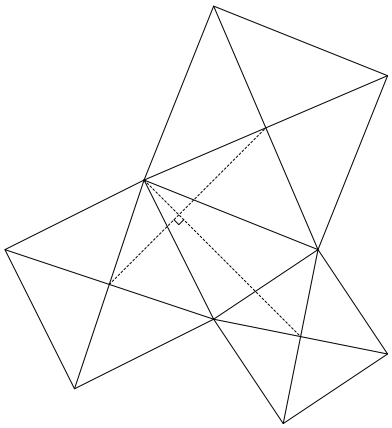
0.6 / 1.0



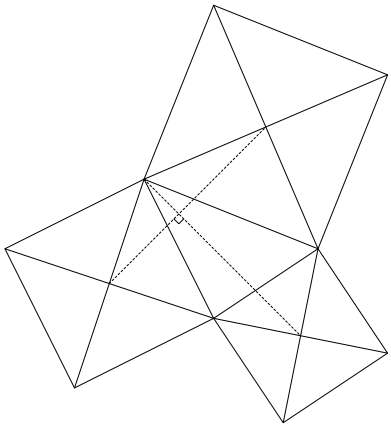
0.4 / 1.0



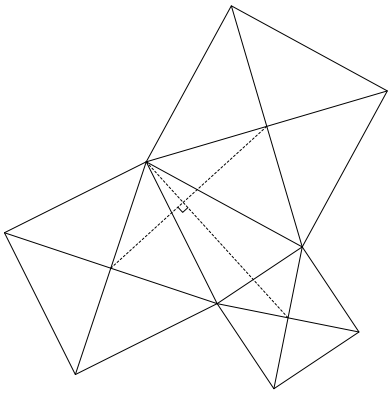
0.2 / 1.0



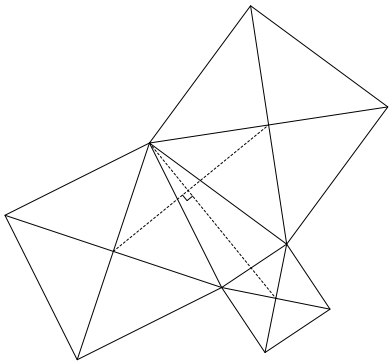
0.0 / 1.0



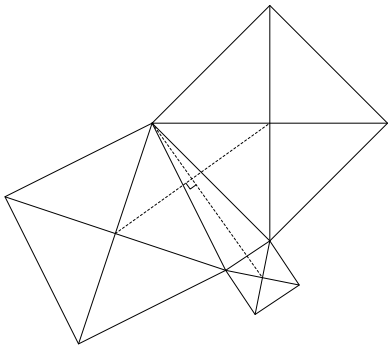
0.0 / 1.0



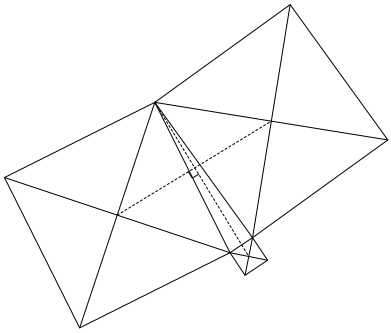
0.0 / 0.8



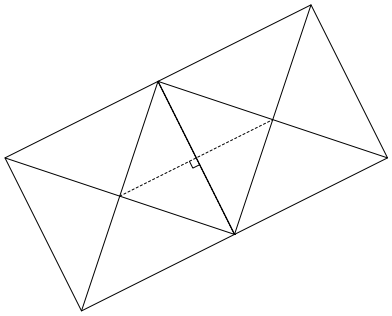
0.0 / 0.6



0.0 / 0.4



0.0 / 0.2



0.0 / 0.0

```
def do_draw_problem ( expr n, i, j ) =
  beginfig ( n );
    draw_problem
      ( (400,400), (400,600),
        i[(400,600),(550,600)], j[(400,400),(550,400)] );
  endfig;
enddef;

show_labels := true;

do_draw_problem ( 50 , 0.6 , 1.0 ) ;
```

Jigsawing

My second METAPOST project concerned our company logo, which I'll show in a few pages, when we combine it with the next one, which concerns jigsaw puzzles.

I still remember the times my parents and I spend whole evenings and sometimes part of the night solving jigsaw puzzles with thousands of pieces. The next piece of METAPOST code is therefore dedicated to my parents.

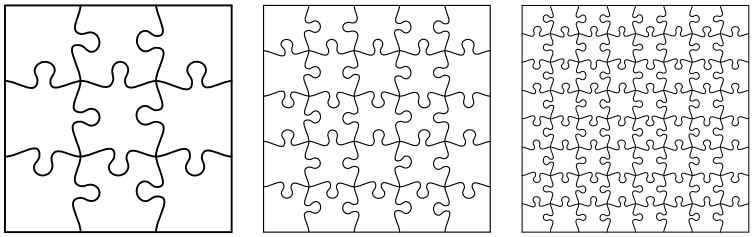


Figure 2 Some simple jigsaws.

This kind of graphics are not that hard to generate with METAPOST, although METAPOST has to work real hard to draw real big ones.

Producing jigsaw originals is just a matter of laying a jigsaw pattern over some kind of picture. The macros we show here however support the generation of separate pieces, an option that we needed for producing a company postcard and a hypertext company profile, in which we use hot spot puzzle pieces.

First we generate the whole jigsaw puzzle. We feed the main macro with the width, the height and some unit.

```
def PreDrawPuzzle ( expr ww, hh, uu ) =
```

We use the METAPOST random generator. Because we want to be able to regenerate previous puzzles, the seed needs to be set explicitly.

```
  randomseed  := random_seed;

  def rand (expr x) =
    (((x)+(uniformdeviate 10))/100)
  enddef;
```

We use the temporary variables w and h to store the width and height and also define our internal unit to be $uu/100$.

```
  w := ww;
  h := hh;
  u := uu/100;
```

The pieces of the puzzle show an alternating pattern. We just stick to this simple pattern, although some extra randomization can be used to generate some more randomized patterns. We use two boolean variables to keep track of the alternation:

```
boolean swapone; swapone := false;
boolean swaptwo; swaptwo := false;
```

We will store the curves of the puzzle in two path variables, *pw* and *ph*, which hold the horizontal and vertical curves. We also need an auxiliary one. All three variables are arrays.

```
path p[], pw[], ph[];
```

The next local macro generates the path. At least some minimal knowledge of METAPOST is needed to understand them.

```
def buildpath (expr len, num) =
```

We fill *num* columns and/or rows with curves.

```
  for j=1 upto num:
```

We need local variables and therefore call:

```
    clearxy;
```

Next we take care of the alternation:

```
    swaptwo := not swaptwo;
    if odd num:
      swapone := not swaptwo;
    else:
      swapone := swaptwo;
    fi;
```

The real work is done in the next few lines. We determine all relevant points on the curve. While doing so, the randomizer slightly offsets the curves from the midpoints. The values were determined experimentally.

```
    for i=0 upto len+1:
      x[i]c = ((i*100)*u);
      y[i]c = ((j*100)*u);
      x[i]m = (x[i]c+50u);
      if swapone:
        y[i]m = (y[i]c-25u)
      else:
        y[i]m = (y[i]c+25u)
      fi;
      x[i]l  = (rand(85)[x[i]c,x[i]m]);
      x[i]r  = (rand(15)[x[i]m,x[i+1]c]);
      y[i]l  = y[i]r=y[i]c;
      x[i]ll = (rand(75)[x[i]c,x[i]m]);
      x[i]rr = (rand(25)[x[i]m,x[i+1]c]);
      y[i]ll = y[i]rr=.50[y[i]c,y[i]m];
    endfor;
```

We only used equations and let METAPOST do the calculations by solving those. Next we draw the path, and in doing this we have to take care of the begin and end points. These actually lay outside the puzzle boundaries, else the curves would end ugly.

```
    p[j] :=
    for i=0 upto len:
      z[i]c..z[i]l..z[i]ll..z[i]m..z[i]rr..z[i]r..
    endfor
    z[len+1]c ;
```

Now that everything's done, we end the loop and the local macro definition.

```
  endfor;
enddef;
```

The boundaries of the puzzles are just straight lines.

```
pw[1]   := (100u,100u)--(100u,(h+1)*100u);
pw[w+1] := ((w+1)*100u,100u)--((w+1)*100u,(h+1)*100u);

ph[1]   := (100u,100u)--((w+1)*100u,100u);
ph[h+1] := (100u,(h+1)*100u)--((w+1)*100u,(h+1)*100u);
```

The next trickery uses the full strength of METAPOST: we clip of the parts of the curves that extend beyond the boundaries.

```
buildpath (w+1,h);

for k=2 upto h:
  ph[k] := p[k] cutbefore pw[1];
  ph[k] := ph[k] cutafter pw[w+1];
endfor;

buildpath (h+1,w);

for k=2 upto w:
  pw[k] := p[w-k+2] rotated 90 shifted ((w+2)*100u,0);
  pw[k] := pw[k] cutbefore ph[1];
  pw[k] := pw[k] cutafter ph[h+1];
endfor;
```

Drawing the whole puzzle can be done by flushing *pw* and *ph*, but as said before, we want to do some more. Therefore we save the puzzle piecewise in:

```
path phw[][];
```

Here another powerful feature shows up. METAPOST itself determines where the pieces begin and end.

```
for kh=1 upto h:
  for kw=1 upto w:
    phw[kh][kw] := buildcycle (ph[kh], pw[kw], ph[kh+1], pw[kw+1]);
  endfor;
endfor;
```

And now we're ready to use them.

```
enddef;
```

After the puzzle is predrawn, the horizontal curves are stored in *pw* and the vertical ones in *ph*. We also have the individual pieces available in *pwh*. The most efficient way of drawing a whole puzzle is using *pw* and *ph*, like in:

```
def DrawWholePuzzle  =
  for k=1 upto h+1:
    draw ph[k] withpen pencircle scaled 2.5u shifted (-100u,-100u);
  endfor;
  for k=1 upto w+1:
    draw pw[k] withpen pencircle scaled 2.5u shifted (-100u,-100u);
  endfor;
enddef;
```

An individual piece can be output by the next macro, which takes two coordinates.

```
def DrawPuzzlePiece (expr ww, hh) =
  draw phw[ww][hh] withpen pencircle scaled 2.5u shifted (-100u,-100u);
enddef;
```

The boundaries are drawn by another macro:

```
def DrawPuzzleBorder =
  draw pw[1]--ph[1]--pw[w+1]--ph[h+1]--cycle
    withpen pencircle scaled 2.5u shifted (-100u,-100u);
enddef;
```

Now let's draw some puzzles first. We want to get the same ones every time, so we first set *random_seed* to some value. We'll give them the same dimensions, using *puzzle_units*.

```
puzzle_size   = 3cm;
random_seed  := 10;

beginfig (101);
  PreDrawPuzzle (3, 3, puzzle_size/3);
  DrawWholePuzzle;
endfig;

beginfig (102);
  PreDrawPuzzle (5, 5, puzzle_size/5);
  DrawWholePuzzle;
endfig;

beginfig (103);
  PreDrawPuzzle (8, 8, puzzle_size/8);
  DrawWholePuzzle;
endfig;
```

As promissed I'll show a more advanced application now, but first we need to load an illustration, like the following logo.

```
input mp-logo; separation := 0;
```

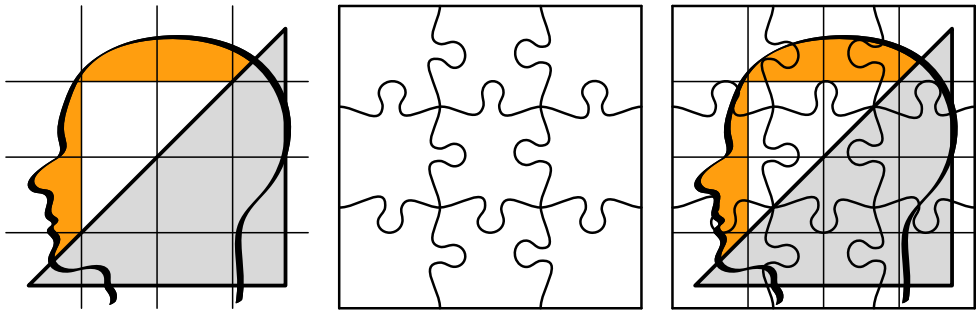
This one shows up as:

```
beginfig (201);
  PragmaLogo (1cm);
endfig;
```

And can be combined with the puzzle. Here we've realized the overlay in METAPOST, but one could as well let TEX do the job.

```
beginfig (202);
  PreDrawPuzzle (3, 3, 4/3cm);
  DrawWholePuzzle;
endfig;

beginfig (203);
  PragmaLogo (1cm);
  PreDrawPuzzle (3, 3, 4/3cm);
  DrawWholePuzzle;
endfig;
```



201 202 203

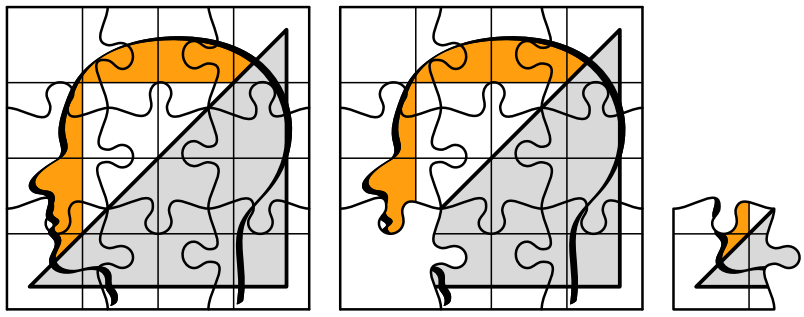
Figure 3 A real jigsaw.

Erasing a piece is quite straightforward. The next illustration can be used in tutorials on solving jigsaws.

```
def ErasePuzzlePiece (expr ww, hh) =
  erase fill phw[ww][hh] shifted (-100u,-100u);
enddef;

beginfig (204);
  PragmaLogo (1cm);
  PreDrawPuzzle (3, 3, 4/3cm);
  ErasePuzzlePiece (1, 1);
  DrawWholePuzzle;
endfig;
```

After we've drawn a picture, it's available in the METAPOST variable *currentpicture*, which means that we can use METAPOST's clip macro for clipping of a piece. Again, we defined ourselve a few auxiliary macro



203 204 205

Figure 4 Some tutorial jigsaw.

```
def ClipToPuzzlePiece (expr ww, hh) =
```

```
  clip currentpicture to (phw[ww][hh] shifted (-100u,-100u));
enddef;

beginfig (205);
  PragmaLogo (1cm);
  PreDrawPuzzle (3, 3, 4/3cm);
  ClipToPuzzlePiece (1, 1);
  DrawPuzzlePiece (1, 1);
endfig;
```

Clipping is not as efficient as one would expect it to be. We can think of looking through a window to the puzzle: the whole picture is output, but only part of it is visible. In our third example we'll build a large picture out of clipped parts. The resulting files are quite large and take some time to print. A more efficient solution would be to let the printer calculate the non clipped picture once and reuse it, but as far as I can see, this is not supported in METAPOST. (There is no real need for such functionality anyway, because one seldom uses the same sub picture more than once.)

We used TEX to produce a company postcard, that showed 3×3 separate pieces. This involves some picture handling and therefore close reading of the METAPOST manual. The postcard was typeset using TEX. Because the postcard was in color, we also had to take care of color separation. The company logo is tuned to this which means that the colored areas are printed slightly under the black lines. Such extensions are not that hard to program once one knows the tricks.

Enlarging illustrations as the one described here are a good illustration of the quality of METAPOST. Using magnifying tools in viewers can be very cruel to designers, but not to METAPOST. Let's spend a few more words on accuracy.

When one looks at the wonderful compact output of METAPOST one will notice that coordinates are specified in high precision. The exact dimensions of the visible part of the picture are specified in the `BoundingBox`. Although probably no POSTSCRIPT interpreter prevents this, the bounding box is often specified in whole numbers. If one considers the very high accuracy of TEX on the one hand and the visible difference between a few POSTSCRIPT points, one can imagine that, especially when scaling is applied, visual imperfectness can surface. We found it quite frustrating that we had to move up an illustration by .5pt, just to get it's embedded background in line with one we let TEX produce in a menu button. After we studied the source of the illustration, we found out that it specified a `HiResBoundingBox`, so now we let TEX use this one when it's there. Fortunately it's possible to instruct METAPOST to output such a high resolution bounding box. Its accuracy deserves it!

Moving around

I don't even know how the next kind of gimmick is called, but let's name it a shift puzzle. We just show this example, which has not that much use (except as an alternative puzzling kind of pagenumbering symbol) because it shows how pictures can be manipulated in METAPOST.

Because we don't want to be dependant of dimensions, we first declare ourselves a decent unit u, and some height h and width w constant that we'll use later on. We also tell METAPOST how many figures we want to generate.

```
u := .75cm;
w := 4;
h := 4;
n := 100;

input mp-logo; separation := 0; PragmaLogo (u);
```

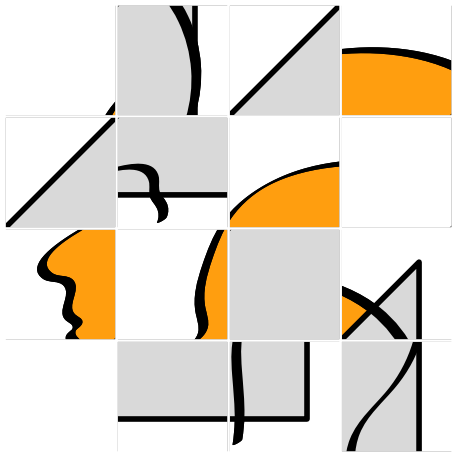


Figure 5 A shifting device.

After calling the macro, the picture is available in *currentpicture*. We are going to break this picture into smaller pieces, which we store in a 2-dimensional array of pictures:

```
picture r[][];
```

We need some more variables, like a scratchpicture *s*:

```
picture s;
```

The next operation is needed because we are going to move pictures around. This kind of manipulations sometimes make designing graphics a nuisance.

```
currentpicture := currentpicture shifted (u,u);
```

Next we split off the subpictures that we are going to move around.

```
for step_w:= 1 upto w+1:
  for step_h:= 1 upto h+1:
    r[step_w][step_h] := currentpicture shifted (-step_w*u,-step_h*u);
    clip r[step_w][step_h] to (unitsquare scaled u);
  endfor;
endfor;
```

The auxiliary macro below takes care of generating a picture.

```
current_w := 1; prev_w := current_w;
current_h := h; prev_h := current_h;

boolean cut_pieces; cut_pieces := false;

def generate_picture ( expr fig_number ) =
  beginfig (fig_number);
```

First we clear anything left.

```
    clearit;
```

This double loop builds a large picture out of the clipped ones we saved earlier.

```
    for step_w:= 1 upto w:
      for step_h:= 1 upto h:
```

```
        addto currentpicture also ( r[step_w][step_h] shifted (step_w*u,step_h*u));
      endfor;
    endfor;
    unfill unitsquare scaled u shifted (current_w*u,current_w*u);
```

This optional loops separate the pieces by thin white lines. By default, this option is turned off.

```
    if cut_pieces:
      for step_w:= 1 upto w+1:
        draw (step_w*u,u)--(step_w*u,(h+1)*u) withcolor white;
      endfor;
      for step_h:= 1 upto h+1:
        draw (u,step_h*u)--((w+1)*u,step_h*u) withcolor white;
      endfor;
    fi;

  endfig;
enddef;
```

We output a complete picture first. This picture could be output more efficient, but we saw no real reason for doing so.

```
generate_picture (0);
```

The next loop results in upto n pictures, each differing from the previous one. We enter an endless loop that is only exit when we have output enough pictures.

```
i := 0;
forever: exitunless i<n;
```

There are 4 different moves: up, down, left and right. We use METAFONT's random generator to choose one.

```
  x := (uniformdeviate 4);
  if x>3:
    if current_w<w: current_w := current_w+1; fi;
  elseif x>2:
    if current_w>1: current_w := current_w-1; fi;
  elseif x>1:
    if current_h<h: current_h := current_h+1; fi;
  else:
    if current_h>1: current_h := current_h-1; fi;
  fi;
```

We don't want too much quite legal but unwanted repetition so we keep track of the last one:

```
  if (current_w<>prev_w) or (current_h<>prev_h):
```

We swap some sub pictures,

```
    s := r[prev_w][prev_h];
    r[prev_w][prev_h] := r[current_w][current_h];
    r[current_w][current_h] := s;
```

write down the picture with

```
    i := i+1;
    generate_picture (i);
```

and finally save the last move:

```
    prev_w := current_w;
    prev_h := current_h;
```

We don't know in advance how many iterations are needed, but at some moment the loop is exit.

```
  fi;
endfor;
```

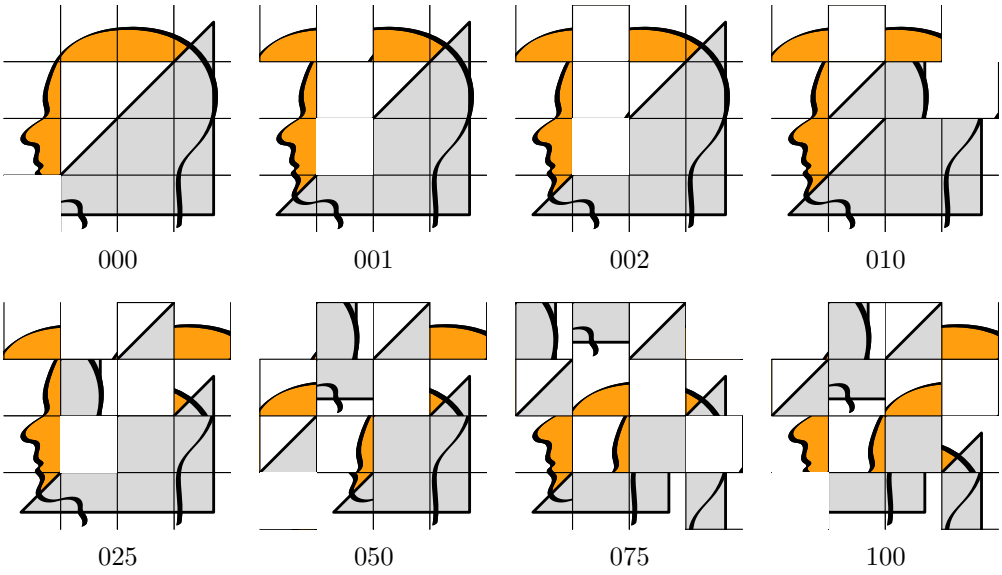


Figure 6 Some simple jigsaws.

It's as simple as that: starting from the solution METAPOST generates the problem. Showing all the results can be very illustrative, but we stick to a subset. The illustration we showed earlier was produced with:

```
cut_pieces := true; generate_picture (101);
```

This looks a bit more like a puzzle, doesn't it?

That's it

At the moment we're using METAPOST mainly for generating symbols and logo's of our customers. We plan to use this marvelous program for building libraries of symbols and pictograms and in due time we'll build ourselve a library (macropackage) that enables us to draw all those (simple) math illustrations that we now draw in drawing packages.