METAPOST and patterns

Piotr Bolek
ul. Szkolna 15, 05-180 Pomiechówek, Poland
Phone: (48) 22–785 43 39
P.Bolek@ia.pw.edu.pl

Abstract

In this paper the METAPOST macros for defining and using patterns are presented. METAPOST is an excellent graphics program which gives the user access to many PostScript features. But there is no way to access the *Pattern Color Space* of PostScript Level 2. The `mpattern` package is the author's attempt to give users of METAPOST a comfortable way of accessing this feature of PostScript. This package allows the user to define patterns using arbitrary METAPOST code, modify the pattern transformation matrix and specify vertical and horizontal displacement of adjacent pattern cells. Examples of defining and using patterns are shown.

Introduction

METAPOST is a very good graphics program. It takes the best from METAFONT, PostScript and TeX. From METAFONT, it borrows the declarative programming model and a way of describing graphic objects. It has a very comfortable interface to TeX and PostScript features. The user can typeset labels using TeX commands and fonts and modify the graphic state parameters of PostScript, such as painting color, line thickness and dashing patterns, as well as the way of line ending and joining. But, the possibility of defining and using patterns is lacking.

Patterns are very useful and comfortable. Once defined, they behave like ordinary colors — they are automatically tiled and clipped on the edges of the painted area by the PostScript interpreter. They can be used for easy definition of textures such as stripes, waves, checkers, hexagons and many more.

Direct implementation of the interface to patterns in METAPOST is impossible without modifying the sources of the METAPOST program itself. The solution proposed by the author is different. The main part of the pattern package is written as METAPOST macros, but the figures in which the patterns are used must be postprocessed by a simple perl script. This script is a simple wrapper that calls the METAPOST program, finds the figures in which the patterns were used and postprocesses these figures. The user who wants to define and use patterns in METAPOST figures must use this wrapper script (called `mpp`, which stands for "METAPOST with Patterns") instead of direct invocation of the METAPOST program.

Patterns in PostScript

There are two types of patterns in PostScript — uncolored and colored. Uncolored patterns do not specify any color and act as stencils for painting with separately specified colors. In uncolored patterns, operators that specify colors are not allowed. The colored pattern specifies the colors used for painting the pattern cell.

Definition of patterns in PostScript consists of several elements. The pattern is defined as a special kind of dictionary. (The PostScript data structure acts as an associative array with elements which may have different types.) The main element of the pattern dictionary is `PaintProc` — the arbitrary PostScript procedure which is executed to paint a single pattern cell.

The other important elements of pattern definition are: pattern bounding box (`BBox`) which is used to clip the drawing made by `PaintProc` and horizontal and vertical spacing between adjacent pattern cells (`XStep`, `YStep`). This spacing may differ from the values implied by the dimensions of the pattern bounding box — the contents of adjacent cells will then overlap or there will be gaps between cells. Values of these parameters must be different from zero — either positive or negative.

The pattern shape can be modified by the arbitrary affine transformation specified during definition of the pattern — pattern cells can be scaled, rotated or skewed.

PostScript patterns and METAPOST

How is METAPOST related to PostScript pattern color space? The main part of the pattern definition is the PaintProc procedure. It is an arbitrary PostScript procedure — and the purpose of METAPOST is to produce arbitrary PostScript procedures. Therefore, the picture produced by METAPOST can be used as a definition of the pattern PaintProc. METAPOST knows the picture bounding box so we can also use this information if we need it.

METAPOST also can be used to specify the pattern transformation matrix which will be used to change orientation, size or shape of the basic pattern cell. The METAPOST `transform` type contains the same information as the PostScript transformation matrix. We can specify the transformation of a pattern cell using comfortable METAPOST (METAFONT) transformation expressions.

The `mpattern` package

The `mpattern` package is the interface to the PostScript Pattern Color Space from METAPOST. Using this package, we can define patterns using arbitrary METAPOST commands. We can also specify the bounding box of a pattern and spacing information (`XStep`, `YStep`). It is possible to use expressions of the type `transform` to specify an arbitrary affine transformation which will be applied to our pattern. The patterns defined with this package are colored patterns.

Once defined, patterns can be used in natural way — by using the `withpattern` operator, similar to `withcolor`, `withpen`, etc.

Here is a simple example. Assuming that we have already defined the path `bean`, we can define and use a pattern like that below:

```
beginpattern(checker);
  fill unitsquare scaled 4mm rotated 45;
endpattern;
beginfig(1);
  fill bean withpattern checker;
  draw bean;
endfig;
```

The result is shown on figure 1.

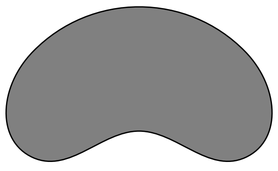

Figure 1

As we can see, the pattern is defined with two macros: `beginpattern` and `endpattern`. The former has one parameter — the name of a pattern. This name will be used later to identify the pattern when the user wants to use it as a filling for a closed path. Between these two macros, the user is allowed to use any valid METAPOST commands.

In our example, the pattern bounding box is not specified. In such situations, it is calculated by METAPOST, and in fact is identical with the bounding box of the implicitly defined picture. When the bounding box should be different from the default, we can specify it using the `patternbbox` macro. We can modify our pattern very easily by specifying the center of our square as the lower-left vertex of the bounding box. It will clip the basic cell of our pattern, ignoring everything outside the bounding box.

```
beginpattern(checker_clip);
  fill unitsquare scaled 4mm rotated 45;
  z1=llcorner currentpicture;
  z2=urcorner currentpicture;
  z1'=.5[z1,z2];
  patternbbox(z1',z2);
endpattern;
```

We can also change the spacing of the pattern without modification of its bounding box. We can specify the vertical and horizontal spacing separately (`patternxstep` and `patternystep` macros), or both of them at once (`patternstep`).

```
beginpattern(checker_ovl);
  fill unitsquare scaled 4mm rotated 45;
  patternxstep(4mm);
endpattern;
beginpattern(checker_gap);
  fill unitsquare scaled 4mm rotated 45;
  patternstep(6mm,7mm);
endpattern;
```

These three modifications of our first pattern are shown on figure 2.

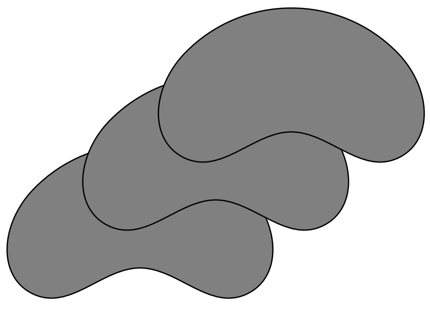

Figure 2

Using the `mpattern` package we can also specify arbitrary transformation of pattern cells. Ordinary METAPOST expressions of the type `transform` are used for this purpose. The `patterntransform` macro the user allows to specify the transformation of the pattern. The argument of this macro should contain the type `transform`. To rotate our example pattern we can define it as follows:

```
beginpattern(checker_rot);
  fill unitsquare scaled 4mm rotated 45;
  patterntransform(identity rotated 22);
endpattern;
```

Patterns may also be translated, scaled and slanted. The transformations can be joined in the usual way:

```
beginpattern(checker_sl);
  fill unitsquare scaled 4mm rotated 45;
  patterntransform(identity rotated 45
                   slanted .2);
endpattern;
```

Examples of transformed patterns are shown on figure 3.

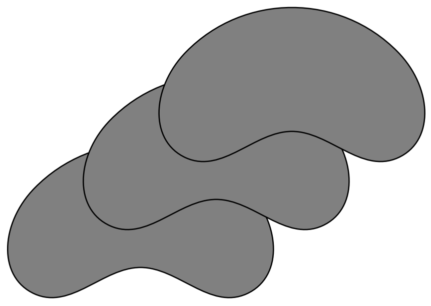

Figure 3

Now we know all the macros for defining and using patterns available to users:

- `beginpattern`, `endpattern`: the couple of macros which enclose the definition of basic pattern cell.
- `patternbbox`: the macro which allows the explicit specification of the pattern bounding box. If this macro is not used in the definition of the pattern, then the bounding box of the whole picture acting as the pattern cell is used. This macro may have two parameters of the type `pair`, or four `numeric` parameters.
- `patternxstep`, `patternystep`, `patternstep`: the macros for specifying spacing of the pattern cells.
- `patterntransform`: the macro for changing the shape of a pattern cell. The argument of this macros must have the type `transform` and it represents the transformation which will be applied to the pattern. This transformation will take place *after* determining the bounding box and spacing of the pattern. Therefore the real size and shape of the basic pattern cell can be different from the ones specified in the pattern definition. Any valid METAPOST transform expression can be used as the argument of this macro.
- `patterncolor`: the macro used to assign color to the defined pattern. In the first stage of processing, the pattern is defined as a color which is replaced by the pattern itself during the postprocessing stage. The colors assigned to patterns are generated automatically, but we can force the use of concrete colors for this purpose. The argument of the `patterncolor` macro must be a number from range $[0, 1]$ and is interpreted as gray level (0 — black, 1 — white). Manual specification of a color tied with patterns requires that this color not be used for other (ordinary) purposes — because every object painted with this color will be painted with the pattern.

  The use of only gray levels in the `patterncolor` macro is by the implementation of assignment of colors to patterns. The assignment information is stored in an array indexed by the colors, which is possible only when the colors are monochrome. This limitation may be relaxed in the future.
- `withpattern`: the primary operator which can be used for drawing shapes filled with earlier defined patterns. It can be used in a way similar to `withpen` or `withcolor` operators.

Implementation of the package

The `mpattern` package consists of two parts. The first is the METAPOST code in which the user interface and working macros are defined. The second is the simple perl script which invokes METAPOST and postprocesess its output.

The processing of the patterns takes place in two steps and is managed by the simple perl script called `mpp`. In the first step, the METAPOST program is invoked. METAPOST code placed between `beginpattern`, `endpattern` macros is processed as the figure with a high number — the default is 999, but if this number is used by the user, then 998 is used, and so on. Problems can arise when the user uses all the picture numbers from 0 to 999, but hopely this is a highly improbable situation. In the `endpattern` macro, the PostScript code generated by the pattern picture is read and remembered as the PostScript pattern definition in the string variable. When the user uses the `withpattern` operator, this code will be placed at the beginning of the picture in which the pattern is to be used. This is performed with a `special` command. Information about every defined and used pattern is stored as comments in output files and in the log file. This information is used in the second step of processing patterns.

The PostScript code defining the pattern is constructed by the macro `endpattern`. The tiling information supplied by the user in the `patternbbox` and `pattern[xy]?step`[1] macros is converted to a form suitable for PostScript. PostScript code generated from an implicitly defined picture is used as the body of the pattern `PaintProc` procedure. All of these elements are placed together into a pattern dictionary and stored in string variables. The METAPOST transform expression, given as an argument to `patterntransform`, is converted to a PostScript transformation matrix and used in definition of the pattern to modify the coordinate system in which the pattern will be painted.

Every defined pattern is joined with the color which will be replaced by the pattern in the second step. The area to be filled with the pattern is filled by METAPOST with this color. The colors used for this purpose should not collide with "ordinary" colors used by the user. At the moment, colors which are to be replaced by patterns are constructed as

```
k * epsilon * white
```

where `k` is the number of the defined pattern, `epsilon` is the smallest number in METAPOST, and `white` is the white color. Using the macro `patterncolor` in the definition of the pattern, the user can explicitly specify the color (gray level) which will be used with the defined pattern. Ensuring that pattern colors will not collide with ordinary colors is, in this case, left to the user.

Every pattern is remembered in a variable with the same name as the argument of `beginpattern`, so the user should not try to use such a variable for other purposes.

The second step in processing patterns is performed by a perl script. Pictures in which patterns are used are found with the help of information stored in the log file. Then for every such picture, substitution from colors-to-patterns is performed. See the small example below.

If the pattern `checker` was defined first — "his" color will be (`epsilon * white`), and the line

```
0.00002 setgray
```

[1] Regular expression notation is used here.

in the output file will be replaced with

```
checker setpattern
```

and definition of the pattern `checker` will be placed at the top of this file. If there are several different patterns used in one picture, then several pattern definitions will be placed at the top of output file.

Pattern examples

We have already seen several examples of patterns. These were checkers — the simplest possible ones. Now let us try to define more interesting and useful patterns.

Line patterns. The basic patterns are of course lines. It seems quite easy to define patterns from lines. But let us have a look at figure 4.

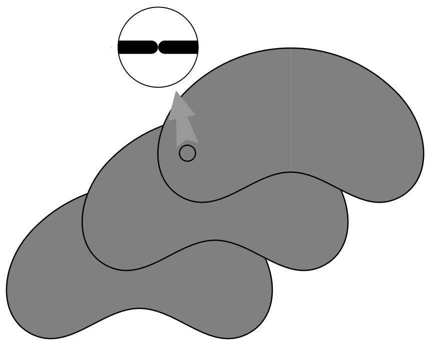

Figure 4

Patterns consisting of vertical and horizontal lines demonstrate something strange — at regular intervals they have thinner parts. This effect has two reasons — the pen used to draw lines is circular, and METAPOST calculates the bounding box of the pattern cell automatically. METAPOST is quite accurate when it calculates the bounding box of the picture, so we have what we wanted…

Knowing the reason for this unwanted effect, we can deal with it. Two possible solutions of this problem are shown in the listing below. The first solution is not to use a rounded pen (`lines_s`), and the second is to explicitly define the bounding box of the pattern in such a way that the rounded ends of the lines are cut off (`lines_ss`).

```
beginpattern(lines_h);
  draw origin--10left
    withpen pencircle scaled 2;
  patternystep(2mm);
endpattern;
beginpattern(lines_v);
  draw origin--10up
    withpen pencircle scaled 2;
  patternxstep(2mm);
```

```
endpattern;
beginpattern(lines_s);
  draw origin--10up
    withpen pensquare scaled 2;
  patternxstep(2mm);
  patterntransform(identity rotated -45);
endpattern;
beginpattern(lines_ss);
  draw origin--10up
    withpen pencircle scaled 2;
  patternxstep(2mm);
  patternbbox(left,10up+right);
  patterntransform(identity rotated 45);
endpattern;
```

Other patterns. Of course we are not limited to the definition of such simple patterns only. The following represent examples of two patterns and their transformations.

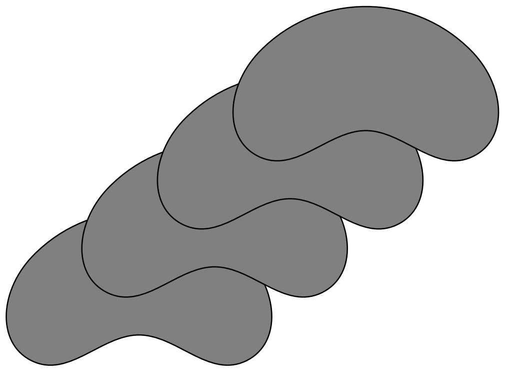

Figure 5

Definitions of our patterns are really simple, but they look quite interesting (fig. 5).

```
def wave_def=
  z1=origin; z2=5up+5right; z3=10right;
  draw z1{right}..z2..{right}z3;
  patternbbox(.25down,10right+5.25up);
enddef;
beginpattern(wave_i);
  wave_def;
endpattern;
beginpattern(wave_ii);
  wave_def;
  patterntransform(identity slanted .9
    rotated 35 xscaled 1.5);
endpattern;
def fish_def=
  z1=origin; z2=5right+5.up;
  path p; p=z1{up}..z2;
  draw p;
  draw p xscaled -1 shifted (5right+5up);
  draw currentpicture xscaled -1;
enddef;
beginpattern(fish_i);
  fish_def;
endpattern;
beginpattern(fish_ii);
  fish_def;
  patterntransform(identity slanted .9
    rotated 67 xscaled 1.5);
endpattern;
```

Interesting patterns can be defined using a `for` loop. Below we define a path which is later rotated, and as a result, we obtain quite interesting looking textures.

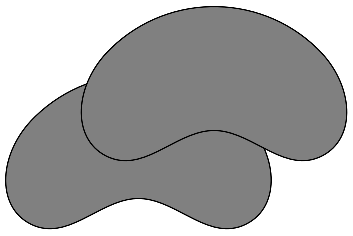

Figure 6

```
beginpattern(p_a);
   z1=(0,0); z2=5up;
   path p;
   p=z1{right rotated -10}..{up}z2;
   for i=1 upto 4:
      draw p rotated (i*360/4);
   endfor;
   patternbbox(-y2,-y2,y2,y2);
endpattern;
beginpattern(p_b);
   z1=(0,0); z2=5up;
   path p;
   p=z1{left}..z1+(-2,3)..{dir 65}z2;
   for i=1 upto 4:
      draw p rotated (i*360/4);
   endfor;
   patternbbox(-y2,-y2,y2,y2);
endpattern;
```

Colored patterns. All previously defined patterns were black and white; but we can of course define patterns with more colors. Here are three examples (fig. 7). (Editor's note: Figures 7 and 8 may be viewed in color at `http://www.tug.org/TUGboat/Articles/tb60/bolek-cfigs.pdf`.) Because the definitions of these patterns are not as simple as those above, it may be interesting to see the shape of the basic pattern cells (fig. 8). The code defining these patterns is contained in Appenix A.

Of course we can use fonts in our patterns. Quite interesting examples of patterns in which texts was used are shown in fig. 9.

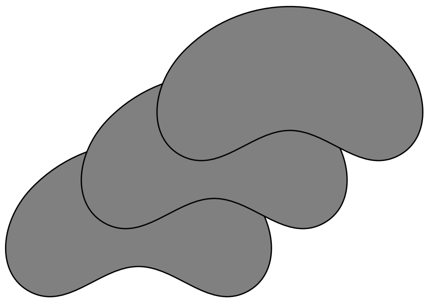
Figure 7

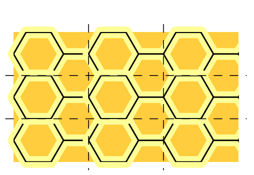
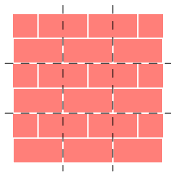
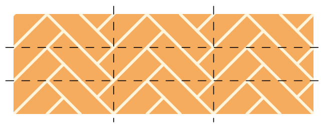
Figure 8

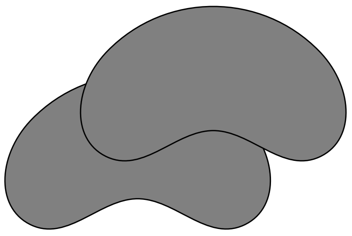
Figure 9

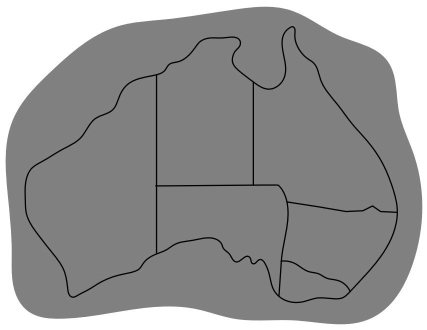
Figure 10

But using text in patterns may be dangerous. The PostScript created by processing the TEX document including a figure with "text" patterns can cause errors in PostScript devices or interpreters. The problems occurs only when the fonts used in the patterns are bitmapped fonts. When we use Type 1 fonts, the file is processed without errors. It seems that the dvips interface to PK fonts is not safe enough when used in patterns. So if we are going to use text in patterns, we should use only Type 1 fonts. Times New Roman is used in fig. 9.

The final example (fig. 10) is an illustration from a chapter about map coloring in book about combinatorics.

A Appendix

The definitions of patterns used in figure 7 are shown below.

```
beginpattern(Honey);
   path p;
   alpha:=360/6;
   for i=0 upto 5:
      z[i]=(3mm*up) rotated (i*alpha-30);
   endfor;
   p=z0--z1--z2--z3--z4--z5--cycle;
   z6=z5+z0-z1;
   z1b=(x2,y3); z2b=(x6,y0);
   patternbbox(z1b,z2b);
   fill z1b--(x1b,y2b)--z2b--(x2b,y1b)--cycle withcolor ((255, 193,  37)/255);
   drawoptions(withpen pencircle scaled 4 withcolor (red+green+.5blue));
   draw p;
   draw z5--z6;
   draw (z1--z2--z3) shifted (z6-z2);
   drawoptions();
   draw p;
   draw z5--z6;
   draw (z1--z2--z3) shifted (z6-z2);
endpattern;

beginpattern(Brick);
   u:=3mm;
   fill unitsquare scaled 2u withcolor (red+.4green+.4blue);
   draw unitsquare xscaled 2u yscaled u withcolor white;
   draw (u,u)--(u,2u) withcolor white;
   draw (0,2u)--(2u,2u) withcolor white;
   patternbbox(origin,(2u,2u));
endpattern;

beginpattern(Floor);
   u:=2mm;
   fill unitsquare xscaled 6u yscaled 2u withcolor ((238, 154,  73)/255);
   drawoptions(withpen pencircle scaled 1 withcolor ((255, 241, 210)/255));
   draw origin--(2u,2u)--(4u,0);
   draw (4u,2u)--(6u,0);
   draw (2u,0)--(3u,u);
   draw (5u,u)--(6u,2u);
   draw (-u,u)--(u,3u);
   draw (5u,3u)--(7u,u);
   patternbbox(origin,(6u,2u));
endpattern;
```

Patterns with texts from figure 9

```
beginpattern(txt_i);
   picture l;
   l=thelabel(btex\font\q=ptmr8r\q MetaPost etex, origin);
   draw l;
   z1=llcorner currentpicture;
   z2=urcorner currentpicture;
   draw l shifted ((y2-y1)*up+.5(x2-x1)*right);
   draw l shifted ((y2-y1)*up+.5(x2-x1)*left);
   patternbbox(z1,(x2,2[y1,y2]));
endpattern;

beginpattern(txt_ii);
   picture l;
   l=thelabel(btex\font\q=ptmri8r\q MetaPost etex, origin);
   draw l;
   z1=llcorner currentpicture;
   z2=urcorner currentpicture;
   draw l shifted ((y2-y1)*up+.2(x2-x1)*right);
   draw l shifted ((y2-y1)*up+.8(x2-x1)*left);
   patternbbox(z1,(x2,2[y1,y2]));
   patterntransform(identity rotated 60);
endpattern;
```