# Blocks and Arrows with MetaPost

**Abstract**
Typesetting of blocks and arrows in ConTEXt with MetaPost.

**Keywords**
MetaPost, ConTeXt, color, drawing, block, arrow, label.

## Introduction

There were a number of reasons for the development of my own package for drawing blocks and arrows. The first is the past experience with the use of LaTEX for these. Its pictures always have a touch of imperfection caused by the restricted set of line and arrow directions available. Using home made sets of arrowheads and lines in more directions than in the LaTEX-fonts, alleviates this problem somewhat but not enough. Drawing with MetaFont gives a lot more satisfaction, albeit with more effort. So it was decided to develop macros for block and arrow-drawing with a sufficient level of sophistication.

## Blocks

As an appetizer figure 1 shows a catalogue of the shapes preprogrammed in the package. After the presentation of the drawing API it will be told how one can easily add its own shapes.

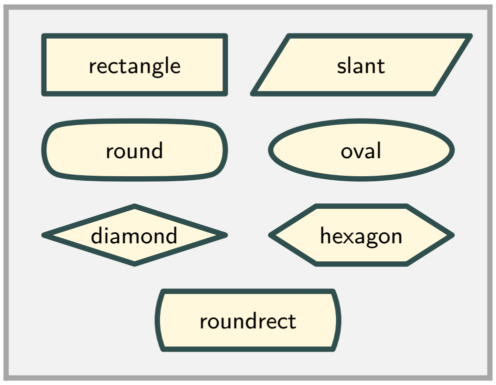

**Figure 1.** Block catalogue

All shapes are drawn from the path of a unit figure ($x = 0 \cdots 1, y = 0 \cdots 1$) and should be sized by setting their width and height. The following shapes are present:

- *rectangle* – unit square scaled in the $x$ and the $y$-direction;
- *slant* – square with a parametrized horizontal shearing;
- *round* – parametrized superellipse;
- *oval* – basically a unit circle;
- *diamond* – a lozenge;
- *hexagon* – six corners regularly spaced;
- *roundrect* – square with parametrically curved vertical sides.

The shapes in figure 1 are placed with the following macro call, where *rectangle*, *slant*, etc. are substituted for *shape*:

```
Block.shape (center, width, height, labeltext);
```

The *rectangle* is the default shape when calling `Block` without the shape modifier. The parameter `center` gives the distance from the origin over which the center of the shape is displaced. The parameters `width` and `height` scale the shape to the dimensions required. Finally `labeltext` is a *string* typeset in the center.[1]

Some of these shapes are parametrized; in the macro call above this parameter is given default value 0. Figure 2 shows the effect of various parameter values on the *slant* and the *roundrect* shapes. These figures are drawn with a more elaborate version of the previous macro:

```
VarBlock.shape (param, rotation, center, width,
                height, labeltext, outline);
```

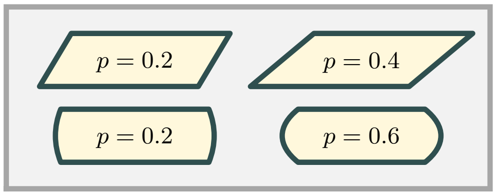

**Figure 2.** Parametrized shapes

There are some extra parameters here. The shape parameter is `param`, the `rotation` is an angle (degrees, counterclockwise) rotating the shape around its center before being shifted in place, and `outline` is a boolean governing the drawing of the frame around the shape. The programmed default is taken for `param` = 0, but as we will see this can be easily changed by altering the shape definition. Figure 3 illustrates the rotation of a shape with various angles.

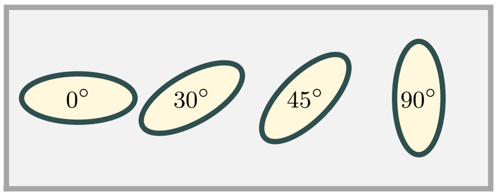

**Figure 3.** Rotated shapes

The `outline` parameter makes it possible to omit the framelines, leaving the colored shape only. Note in figure 4 the small difference in size between the two shapes, caused by the centering of the linedrawing on the boundary of the shape.[2] In order to facilitate drawing of shapes without outline one can use macro `OBlock` in the same manner as Block:[3]

```
OBlock.shape(center, width, height, labeltext);
```

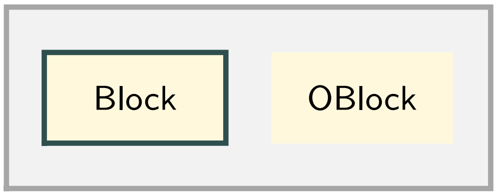

**Figure 4.** Outline parameter

The coloring of pictures is based on a set of four different colors; the management of those is explained in section “Color management”. The distinct colors are:

1. the color for linedrawing
2. the color for the interior of shapes
3. the color for the labeltext
4. the color for the general background

In the example of figure 5 two sets of colors are used.

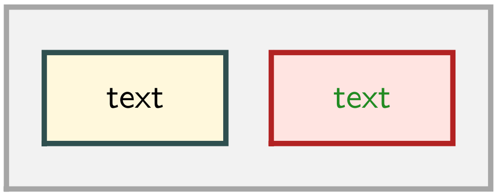

**Figure 5.** Different coloring

The rectangular frames around the figures are drawn by a simplified shape drawer. There are three macros:

```
Frame (width, height, gap);
Framed (width, height, gap, framepen);
Framed (width, height, gap, framepen) modifier;
```

The first macro produces a rectangular block of size $(gap + width + gap) \times (gap + height + gap)$ filled with the general backgroundcolor. The second one uses the *pen* parameter `framepen` to surround the block with framelines in the color for linedrawing. The third gives the possibility to individually modify the surrounding framelines in linestyle and color. The gap is an enlargement on all sides of the block; it was introduced in order to prevent lines drawn at the borders of the drawing area from spilling partly over into the surroundings. The resulting frame is placed at position $(-gap, -gap)$.

As is the case with `Framed` one can have a modifier on the framelines following calls to `Block`, `OBlock` and `VarBlock` too. The right side of figure 6 has been done with modifier `dashed evenly withcolor 0.7yellow`.[4]

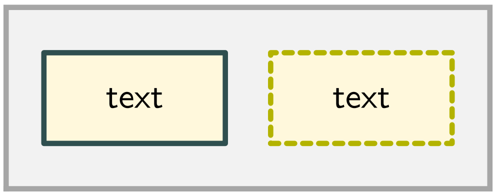

**Figure 6.** Trailing modifier

## Defining shapes

It is quite easy to define shapes yourself and call them up with `Block.shape`. Lets have a look at how shapes are done. Shape drawing is effected by macro `Form` that expects to receive a *path* of the shape in one of its parameters; it rotates and shifts the path as required, fills the resulting path, places the labeltext and possibly surrounds it with framelines. All this is done in the same way for all shapes.

The path of the shape is constructed by helper macro `varblock@#`. For example `Block.oval` uses a predefined *string* `blockf.oval` which contains the path expression for a circle fitting within the square bounded by $x = 0 \cdots 1, y = 0 \cdots 1$. That expression, possibly parametrized with the shape parameter, is scanned and processed, the resulting path being scaled to its proper width and height. Afterwards the string `blockl.oval` is scanned for any postprocessing action that might be needed. The *round* shape (actually a superellipse) needs such postprocessing; for the others this is just a noop i.e. an empty string.

All that is needed therefore is the definition of these two strings. As an example a fourpointed star is developed; its four points ending at the midpoints of the sides of the unit rectangle, the intersections between them parametrically placed on the diagonals (see figure 7).

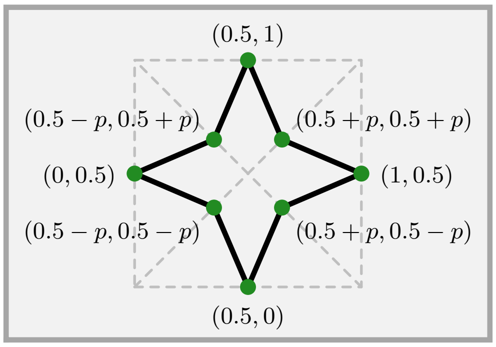

**Figure 7.** Plan of fourpointed star

In the code for the fourpointed star below note the *numeric* variable `p_`. The expansion of `VarBlock` assigns `param` to this variable.[5]

```
% Define path for fourstar.
vardef unitfourstar =
       (0.5,0)
       -- (0.5+p_,0.5-p_) -- (1,0.5)
       -- (0.5+p_,0.5+p_) -- (0.5,1)
       -- (0.5-p_,0.5+p_) -- (0,0.5)
       -- (0.5-p_,0.5-p_) -- cycle
enddef;
% Define scan strings for fourstar.
string blockf.fourstar, blockl.fourstar;
blockf.fourstar :=
    "if p_ = 0: p_ := 0.12 fi; unitfourstar";
blockl.fourstar := "";
```

This code is used in the placement of the two stars in figure 8 with:

```
Block.fourstar (.., "star");
VarBlock.fourstar (0.05, 45, .. , "", true);
```

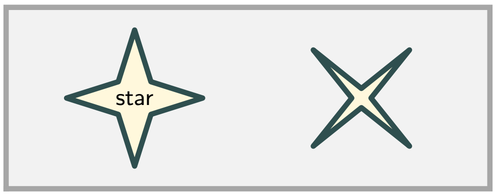

**Figure 8.** Fourpointed stars

## Labels

Besides the standard `label` macro in *plain* one can use the `Label` macro instead. The differences are subtle, the most important one being the typesetting of the labeltext. In `Label` the text is placed with ConTEXt-macro `textext` into a `hbox` whose depth is forced to zero. The reason for this is seen in the top boxes of figure 9, which are typeset with the original `label` macro: the baselines in the two boxes differ because of their different depths. At the bottom the `Label` macro has put both texts neatly on the same baseline. The effect is clearly seen in the position of the e's and d's with respect to the dashed line. A second refinement is the fact that the labeltext can be both in the form of a*picture* or a *string*. Since the `Label` macro is used in the typesetting of shapes, this applies there too.

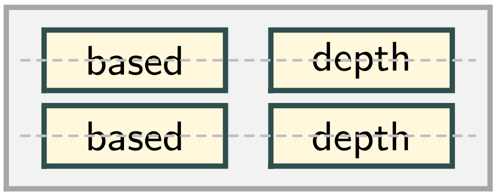

**Figure 9.** Label placement

## Arrows

Uneasiness grew over the shape of the arrows provided by the *plain* macros. This is illustrated in figure 10. The shape of the arrowheads on the left could be called somewhat more refined than those on the right. Moreover the *plain* macros distort the arrowhead at the end of a curved line, bending their flanks with the curve. It is, of course, a matter of taste to find this appalling. But more objectively stated, *plain* makes it impossible to consistently draw arrowheads in the same shape. A suitable alternative has given by David Salomon and is implemented here.[6]

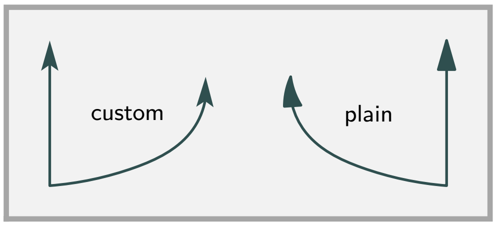

**Figure 10.** Arrow shapes compared

Figure 11 shows the geometry of the arrowhead. There are several parameters that can be varied:

- $l$ is the total length of the arrowhead, in the figure the distance between the two red bars;
- $r$ is the ratio that determines the distance from the tip to the point where the arrowhead is affixed to the incoming line (absolute length is $r \times l$);
- $\alpha$ is the opening angle at the tip and determines the distance between the wingtips;
- $\beta$ gives the curvature of the flanks (here $10^\circ$), the straight gray line is the one drawn for $\beta = 0^\circ$;
- $\gamma$ gives the curvature of the tailtips (here $2^\circ$), the straight gray line is the one drawn for $\gamma = 0^\circ$.

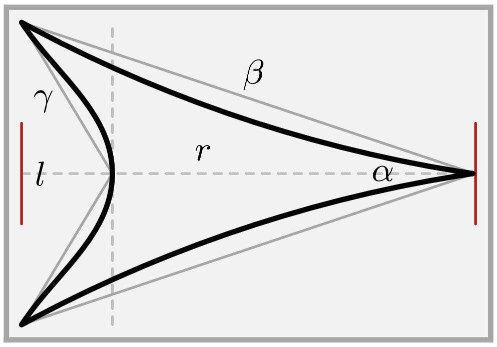

**Figure 11.** Arrowhead

With this geometry available, one can build many different arrowheads; figure 12 presents some. As can be seen, the heads may or may not have an outline. Be aware that not all parameter value combinations lead to pleasing looking shapes. Arrowshapes are defined with the following macro:

```
defineArrow (l,r,alfa,beta,gamma,outline) name;
```

The first five parameters are those from figure 11. The `outline` parameter is a boolean designating the drawing of an open arrowhead when `true`. In an open arrowhead interior and outline are filled with the colors for filling and linedrawing, respectively; `false` results in a solid head in the linedrawing color. It is necessary to select pen and colorset before the definition is executed, because these are used at that point. The parameter `name` is a user chosen unique designation by which the arrowhead later will be retrieved for placement in the drawing. Nota bene: this parameter must be a *string*. The heads thus defined keep their coloring even when a different colorset is selected later.

The arrows so defined are placed with the following macros for single and doublesided arrows, respectively. Here too the `name` must be a *string*:

```
drawArrow (name) path_expression;
drawdblArrow (name) path_expression;
```

The lines to the arrowheads adapt to the linecolor in effect at definition time, but this can be overridden with a modifier for both linestyle as well as color (see the rightmost arrow in figure 12).

The first of the next three macros defines a solid arrowhead for default usage.[7] The next two draw these default arrows, respectively single and doubleheaded; they are illustrated in figure 13. The drawing is fully in the current linedrawing color. However a modifier may be applied to change linestyle and/or drawing

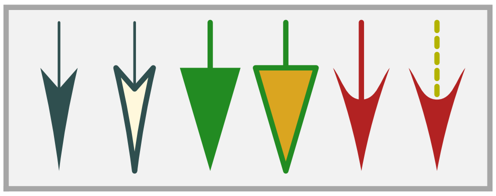

**Figure 12.** Arrowhead variations

color, as in the two arrows on the right. Changing the color for subsequent drawing can also be effected by changing the color for linedrawing.

```
defineDefaultArrow (l, r, alfa, beta, gamma);
arrowline path_expression;
dblarrowline path_expression;
```

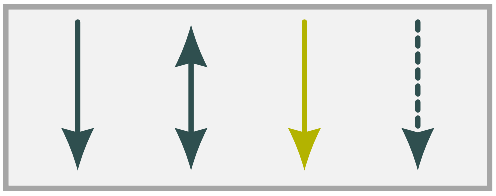

**Figure 13.** Default arrow drawing

## Color management

The coloring scheme introduced in section “Blocks”, consists of four colors used in linedrawing, filling the interior of shapes, drawing labeltext and the general background. The color for filling may be given an alpha mode and a factor for transparency. These colors can all be set individually with the four macros:

```
setFrameColor color;
setFillColor color;
setTextColor color;
setBackgroundColor color;
```

In modifiers they can be applied with a shortcut for `withcolor color` by using:

```
withFrameColor
withFillColor
withTextColor
withBackgroundColor
```

The alpha mode and fill can be changed through:

```
setFillMode mode;
setFillAlpha factor;
```

Sometimes one needs a different set of colors just momentarily, then the following macros come in handy. Their names speak for themselves. Note that saving cannot be nested, resaving causes the previous save to be lost; therefore do not forget to restore in case.

```
saveColors;
restoreColors;
```

There exist the following macros for the definition of a set of colors:

```
defineColors  (line,fill,text,background) name;
defineColorsTransparent
  (line,fill,text,background,mode,factor) name;
defineCurrentColors name;
defineDefaultColors
  (line, fill, text, background);
defineDefaultColorsTransparent
  (line, fill, text, background, mode, factor);
setDefaultColors;
```

As was done for arrow shapes, name defines a designator with which to call up the thus named set of colors. Nota bene: this parameter must be a *string*. If the transparency parameters are not explicitely given, they assume default values of 1 for mode and factor, leading to completely opaque colors. The third macro eases entering a definition for the colors currently set, the fourth and fifth define a default that can be called up by the last one. Colorsets are set by the next two macros; here too name must be a *string*. The second one below is just a set followed by a save.

```
setColors name;
setSaveColors name;
```

## Notes

1. This is not completely true; in section 4 it is mentioned that a *picture* is allowed also.
2. In case one wants to combine shapes with and without outlines, the individual widths and heights can be adjusted by the size of the pen in the respective dimension. The macros penX and penY provide these for the current pen.
3. For those inclined to perfection: compare the rectangles in figures 4 and 6 for the use of a square pen in the former.
4. Be aware that a dashed modifier applies to a pencircle, but has no effect with pensquare.
5. It arises because the formal parameters of the macro definition cannot be used in the blockf string, which is processed by scantokens. Also note the use of vardef in the path definition macro, def will not work here.
6. David Salomon, Arrows for Technical Drawings, TUGboat, Volume 13 (1992), No.2, p. 146–149.
7. Be aware that the implementation of the definition code thus not guarantee recycling of the memory for redefined entries. One should there not redefine too enthousiastically.

Hans van der Meer
hansm@science.uva.nl