Colors and pictures

Abstract

This article is about MetaPost output. MetaPost produces graphics by means of `picture` variables that can contain a few different object types. The most important drawing object types can be colorized, so the first part of this article will talk about color data structures.

Colors

Color models

In order to understand how MetaPost handles color, it is necessary to understand a little bit about color models. Explaining that prerequisite knowledge in this article would make it much too long, so the assumption is made that you at least understand the difference between the basic principle behind greyscale, RGB, and CMYK specifications as ways to describe colors.

For once, this section does not start with a formal syntax. The formal specification would not really help because almost all the information we need cannot be seen in the expression syntax: parsing colors is easy for MetaPost. The interpretation that needs to happen after the reading has been done is the complicated bit.

MetaPost internally has four color models, any one of which can be chosen to do actual output with. Each of the color models also has an associated data type that can be used to define variables with that color model as its 'type':

☐ No model: `boolean`
☐ Greyscale: `numeric`
☐ RGB: `rgbcolor` (this is the initial default color model)
☐ CMYK: `cmykcolor`

None of these color models have an alpha/opacity component.

There is an internal variable `defaultcolormodel` that allows you to set a default color model:

```
defaultcolormodel := 5; % RGB
```

Each of the four color models MetaPost supports has an integer value associated with it, and these are the numerics used with `defaultcolormodel`. The numbers are: No model: `1`, Greyscale: `3`, RGB: `5`, and CMYK: `7`.
In case you are wondering: they are all odd because MetaPost uses the values `0`, `2`, `4`, and `6` internally to signify ⟨unknown⟩ variables in each of these color models.

When you ask MetaPost to create a graphic element (a path or picture, as will be discussed in the next section) there are primitive operations to specify both the color model and the color value that is to be used while adding this object to the picture it will become part of.

The next set of examples use `draw` as example of creating a graphic element. All of the examples produce output with the color value 'black', depending on how that is done within that particular color model. The examples use `draw` as an educational shortcut, but in reality they apply to one of the primitive operations that will be discussed in the next section. A typical definition of the `draw` macro does more work than just

a single primitive operation, so please focus on the color differences between the examples only.

First up, there is the option to use the current default:

```
draw p;
```

which uses a suitable 'black' definition for the current `defaultcolormodel`.

To use an explicit black greyscale when drawing a path:

```
draw p withgreyscale 0;
% or its alias:
draw p withcolor 0;
```

To use an explicit black RGB when drawing a path:

```
draw p withrgbcolor (0,0,0);
% or its alias:
draw p withcolor (0,0,0);
```

To use an explicit black CMYK when drawing a path:

```
draw p withcmykcolor (0,0,0,1);
% or its alias:
draw p withcolor (0,0,0,1);
```

From the above, you can see that `withcolor` is smart about what argument it gets and automatically picks the correct color model based on that value's specification. It will do the same thing if the value is a named variable. Use of the `withcolor` alias is recommended because it is shorter and (when used with color variables instead of literal values) it allows you to switch to a different color model without having to manually change every drawing command.

You will probably have noticed that the preceding examples covered only three of the four color models. The 'No color' color mode needs a bit more explanation.

The equivalent of

```
draw p;
```

is this:

```
draw p withcolor true;
```

Which uses the 'No model' color model to explicitly enable the black initialization. On its own that is not valuable. The real reason for the 'No model' is seen when the color model is used as a negation.

To skip black initialization when drawing a path, you can do this:

```
draw p withoutcolor;
% or its alias:
draw p withcolor false;
```

In this situation, the current object (`p`) will have no color information attached to it at all. No default 'black' will be output, so this object will be drawn with the color of the preceding object, if there is one. Be warned though that if this is the very first object to be output, it is likely it will still come out as black because usually printing systems start by initializing a default black color value.

Color variables

Variables of all color model types can be created using:

```
boolean mynocolor;
numeric mygreycolor;
rgbcolor myrgbcolor;
cmykcolor mycmykcolor;
```

As we saw earlier in the literal color model syntax examples, the input syntax for `rgbcolor` is a triplet of ⟨numeric expression⟩s inside parentheses, and for `cmykcolor` it is a quartet of ⟨numeric expression⟩s.

Just like `pair`s have `xpart` and `ypart` to access the parts of the variable, there are dedicated primitives for the RGB and CMYK color model parts as well.

For RGB:

```
redpart myrgbcolor;
greenpart myrgbcolor;
bluepart myrgbcolor;
```

For CMYK:

```
cyanpart mycmykcolor;
magentapart mycmykcolor;
yellowpart mycmykcolor;
blackpart mycmykcolor;
```

For orthogonality, there is also a primitive for the single greyscale part of a ⟨numeric⟩:

```
greypart mygreycolor;
```

These eight primitives can be used in equations, just like their `pair` counterparts.

The ⟨numeric expression⟩s that are used for the color parts in colors are treated a bit special when they are used as part of one of the primitive drawing commands. Most importantly, no error is produced when any of the parts are unknown or outside of the $[0, 1]$ range. They are just silently clipped to fit within the range. It is your responsibility as programmer to make sure that all the combination of ⟨numeric expression⟩s actually make sense as a color value.

Operations on colors

There are relatively few operations MetaPost can perform on RGB or CMYK colors as a singular object. Quite a lot of operations can already be done by manipulating the separate parts that were mentioned in the previous section, so there is little need for color-specific operators. Still, there are a few operations at 'top level' available.

You can multiply or divide color variables by a numeric:

```
rgbcolor myrgb;
myrgb = (0.5,0.5,0.5) * 1.5;
% => (0.75,0.75,0.75)
```

Or you can add or subtract two colors of the same type:

```
rgbcolor myrgb;
myrgb = (0.5,0.5,0.5) + (0.25,0.25,0.25);
% => (0.75,0.75,0.75)
```

You can also find the 'along-the-way' between two colors:

```
rgbcolor myrgb;
myrgb = .5[(0.5,0.5,0.5),(0.25,0.25,0.25)];
% => (0.375,0.375,0.375)
```

And colors can be negated:

```
rgbcolor myrgb;
myrgb = -(0.5,0.5,0.5);
% => (-0.5,-0.5,-0.5)
```

Finally, you can compare colors of the same type with each other:

```
rgbcolor myrgb, myrgba;
myrgb = (0.5,0.5,0.5);
myrgba = (0.25,0.25,0.25);
if myrgb > myrgba:
  message "true";
fi
```

Such tests process each component in order, and stop as soon as they notice a difference.

Operations on color variables like these may seem a bit useless at first glance, but the MetaPost macro packages that do three-dimensional drawings typically depend on color-based triplets or quartets as their data structures for points in space.

Pictures

MetaPost uses ⟨picture⟩s to internally store and eventually output graphical items. Here is the syntax tree for specifying ⟨picture⟩s:

⟨picture primary⟩ → ⟨picture variable⟩
  | `nullpicture`
  | `(`⟨picture expression⟩`)`

⟨picture secondary⟩ → ⟨picture primary⟩
  | ⟨picture secondary⟩⟨transformer⟩

⟨picture tertiary⟩ → ⟨picture secondary⟩

⟨picture expression⟩ → ⟨picture tertiary⟩

⟨addto command⟩ → `addto`⟨picture variable⟩`also`⟨picture expression⟩⟨option list⟩
  | `addto`⟨picture variable⟩`contour`⟨path expression⟩⟨option list⟩
  | `addto`⟨picture variable⟩`doublepath`⟨path expression⟩⟨option list⟩

⟨option list⟩ → ⟨empty⟩ | ⟨drawing option⟩⟨option list⟩

⟨drawing option⟩ → `withcolor`⟨color expression⟩
  | `withrgbcolor`⟨rgbcolor expression⟩
  | `withcmykcolor`⟨cmykcolor expression⟩
  | `withgreyscale`⟨numeric expression⟩
  | `withoutcolor`
  | `withprescript`⟨string expression⟩
  | `withpostscript`⟨string expression⟩
  | `withpen`⟨pen expression⟩
  | `dashed`⟨picture expression⟩

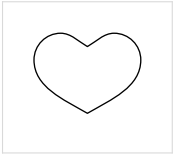
There are simple parts like ⟨transformer⟩ and subexpressions in parentheses that can be skipped because we have talked about those before. The expression part of this syntax diagram is quite unremarkable, except for mentioning the one predefined picture

variable: nullpicture. But this is a really important variable, because nullpicture is the way to create or reset a picture variable to the ⟨known⟩ (and empty) state.

Creating pictures and adding to them

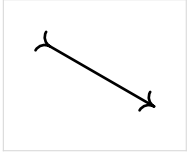
Before you look at the examples below, there is some information you should know. The examples use a predefined path with the name heart, like this:

```
path heart;
heart := (0,0){dir 30}..{up}(20,20)..
        {left}(10,30)..
        {dir -150}(0,25){dir 150}..
        {left}(-10,30)..{down}(-20,20)..
        {dir -30}cycle;
```



and a picture arrow_pic that already contains a simple graphic. The complete definition for that picture is:

```
path t_,h_,a_;
picture arrow;
arrow := nullpicture;
t_ := (-4,19){down}..{right}(0,15)
      {left}..{down}(-4,11);
h_ := t_ shifted (45,0);
a_ := (0,15)--(45,15);
def stroke_ =
  withpen pencircle scaled 1
enddef;
addto arrow doublepath t_ stroke_;
addto arrow doublepath a_ stroke_;
addto arrow doublepath h_ stroke_;
arrow := arrow rotated -30
               shifted (-25,12);
```



The last line of an example is typically

```
shipout A;
```

We will talk about the shipout command (and the withpen option) a bit later in the article. Now let's start with the examples …

You create a new picture variable using picture:

```
picture A;
```

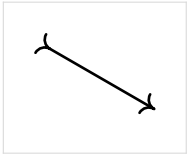
however, this creates a picture variable with the 'unknown' state. To convert it to a usable state, you *always* have to initialize it from another picture, for example:

```
picture A;
A = nullpicture;
```

Use an assignment (:=) instead of an equation (=) if you need to clear the receiving picture.

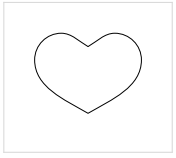
You can add another picture to a picture:

```
picture A,B;
A = nullpicture;
B = arrow;
addto A also B;
shipout A;
```



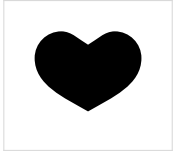
Add a stroked path to a picture:

```
picture A;
A = nullpicture;
addto A doublepath heart;
shipout A;
```



Add a filled path to a picture:

```
picture A;
A = nullpicture;
addto A contour heart;
shipout A;
```



The path must be cyclic for `contour` to work, because it needs a closed path to fill.

Adding a text label to a picture:

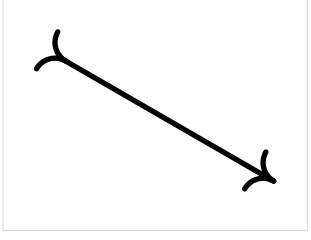
```
picture A,B;
A = nullpicture;
B = "a" infont "cmr10" scaled 4;
addto A also B;
shipout A;
```

The `infont` operation is a bit special because it literally creates a picture and therefore it wants to be paired with a non-initialized picture variable. There is no need to assign `nullpicture` to `B` first. In fact, if `B` is a 'known' variable at this point, you will get the `Redundant or inconsistent equation.` error.

All three of the ⟨addto command⟩ versions accept a list of options that we will discuss shortly.

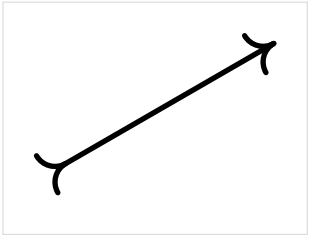
At the expression level, a picture expression can be transformed in all the normal ways:

```
picture A,B;
A = nullpicture;
B = arrow;
A := B scaled 2;
shipout A;
```



or (for example)

```
picture A,B;
A = nullpicture;
B = arrow;
addto A also B scaled 2 rotated 60;
shipout A;
```



Options to the addto command

The addto command forms accept various options.

We have already encountered the color options:

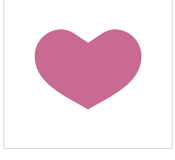
withcolor⟨color expression⟩
withrgbcolor⟨rgbcolor expression⟩
withcmykcolor⟨cmykcolor expression⟩
withgreyscale⟨numeric expression⟩
withoutcolor

It is useful to know that when multiple color options are specified, the last one in the sequence 'wins'.

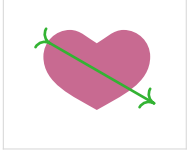
Here are a few examples:

```
picture A,B;
A = nullpicture;
B = arrow;
addto A also B
  withrgbcolor (0.2, 0.7, 0.2);
shipout A;
```

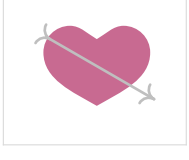
```
picture A;
A = nullpicture;
addto A contour heart
  withcmykcolor (0.2, 0.7, 0.2, 0);
shipout A;
```



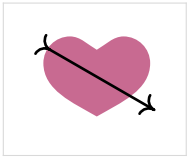
```
picture A;
A = nullpicture;
addto A contour heart
  withcmykcolor (0.2, 0.7, 0.2, 0);
addto A also arrow
  withcolor (0.2, 0.7, 0.2);
shipout A;
```



```
picture A;
A = nullpicture;
addto A contour heart
  withcmykcolor (0.2, 0.7, 0.2, 0);
addto A also arrow
  withgreyscale 0.75;
shipout A;
```



```
picture A;
A = nullpicture;
addto A contour heart
  withcmykcolor (0.2, 0.7, 0.2, 0);
addto A also arrow;
shipout A;
```

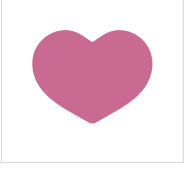


The `withpen` option allows specifying a pen:

```
picture A;
A = nullpicture;
addto A doublepath heart
  withpen pencircle scaled 5;
shipout A;
```

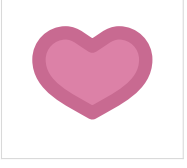
This also works for the `contour` case, where it then does 'filldraw':

```
picture A;
A = nullpicture;
addto A contour heart
  withcmykcolor (0.2, 0.7, 0.2, 0)
  withpen pencircle scaled 5;
shipout A;
```



The example above uses a single color for both the filling and the stroking. If you want to use separate colors for each, you have to add two items to the image:

```
picture A;
A = nullpicture;
addto A contour heart
  withcmykcolor (0.1, 0.6, 0.1, 0);
addto A doublepath heart
  withcmykcolor (0.2, 0.7, 0.2, 0)
  withpen pencircle scaled 5;
shipout A;
```



With `dashed`, it is possible to specify a dash pattern to use for stroking a path. This accepts a picture as argument, so that needs be exist first. One of the simplest examples looks like this:

```
picture A,B;
A = nullpicture;
B = nullpicture;
addto B doublepath (0,0)--(2,0);
addto B doublepath (6,0)--(8,0);
addto A doublepath heart
  dashed B;
shipout A;
```

Dash patterns are quite special `picture`s. When the dash pattern gets used, MetaPost flattens whatever the content of the picture is onto the x axis. The left-most and right-most x values define the bounds of the pattern. The set of produced x values will then be used as the pattern to use to stroke the path the dash pattern is applied to. MetaPost will repeat that whole picture as a pattern if needed, but it will initially start at $x = 0$. This allows shifting of the pattern.

Here are the dashes from the example above again, but now applied to a straight line. I also added a dot to show to (0,0) point:

```
picture A,B;
A = nullpicture;
B = nullpicture;
addto A doublepath (0,0)
  withpen pencircle;
addto B doublepath (0,0)--(2,0);
addto B doublepath (6,0)--(8,0);
addto A doublepath (0,0)--(48,0)
  dashed B;
shipout A;
```


Note that the dash picture produces a repeating pattern 2 units on, 4 units off, 2 units on. The middle dashes in the example output are the same width as the gaps because they consist of the last part of the first repetition and the first part of the second repetition (and that repeated five times).

Shifting the pattern to the left by two units allows it to start with a gap.

```
picture A,B;
A = nullpicture;
B = nullpicture;
addto A doublepath (0,0)
  withpen pencircle;
addto B doublepath (0,0)--(2,0);
addto B doublepath (6,0)--(8,0);
addto A doublepath (0,0)--(48,0)
  dashed (B shifted (-2,0));
shipout A;
```


There are lots of rules for dash patterns because MetaPost typically uses primitive support in the backend to handle the actual dashing (e.g. `setdash` for Encapsulated PostScript output):

- ☐ A dash pattern should not contain text or filled objects (so only non-cyclic paths are allowed)
- ☐ None of the paths may overlap when projected on the x axis (and all the y coordinates are ignored)
- ☐ Any used pens (`withpen`) are ignored.
- ☐ Color settings (`withcolor` c.s.) are simply not allowed at all.

The last two limitations come from the fact that a dash patterns uses the pen and color of the object they are applied to. Finally, `dashed` does not work well with pens other than pens derived from `pencircle`. Again, this is because of limitations in the backend(s).

The final two options are for specifying pre- or postscripts:

`withprescript` ⟨string expression⟩
`withpostscript` ⟨string expression⟩

These can be useful when generating EPS or SVG output. It is not possible to give an actual example inside this (ConTEXt-processed) article, because ConTEXt uses these primitives for its own purposes, unfortunately.

But here is a listing of an example that assumes the default EPS output mode in standalone MetaPost:

```
picture A;
A = nullpicture;
addto A doublepath (0,0)
  withprescript "start1"
  withprescript "start2"
  withpostscript "stop1"
  withpostscript "stop2";
shipout A;
end.
```

When the above is processed by MetaPost, it will create an output file containing the typical EPS preamble followed by:

```
start2
start1
 0 0 0 setrgbcolor 0 0 dtransform truncate idtransform setline...
newpath 0 0 moveto 0 0 rlineto stroke
stop1
stop2
showpage
```

The `withprescript` and `withpostscript` options are therefore a lot like `special` in TEX: if you are familiar with PostScript (or SVG, for that output format), you can use these options to tweak the output to support features that are not possible within MetaPost itself, like for example spot colors or transparency.

Two uses of each option are included in the example to show off the relative ordering in the output when either one of them is specified more than once.

Picture commands

Possibly the most important command that can be used with a picture is `shipout`, because that instructs MetaPost to open an output file for the picture and convert its contents to the correct format. Using the command itself is simple:

```
shipout A;
```

This uses the internal variables `outputformat` and `outputtemplate` to construct the filename to be used.

There is a command to clip a picture to a path:

```
picture A;
path p;
...
clip A to p;
```

This path can have any shape, but it must be cyclic.

Set the bounding box of a picture to a path:

```
picture A;
path p;
...
setbounds A to p;
```

the path must be cyclic, and is always simplified to a rectangle based on the smallest and largest x and y values of the path's explicit points.

You can ask for the corners of a picture:

```
picture A;
pair t;
...
t = llcorner A;
% also lrcorner, urcorner, ulcorner
```

Finally, it is possible to loop over de contents of a picture using the `within` operator.

Using the `for` ... `within` operation, it is possible to ask for the constituent parts of each of the drawing items in a picture. The part names are given in a condensed form in the following examples. By recombining the extracted parts, it is possible to completely reconstruct a picture.

For these tests, you may have to check the type with an `if` test (one of `filled`, `stroked`, `clipped`, `bounded`, `textual`, as discussed in the article about conditionals) first, because not all graphical objects have all parts.

Here is the list:

Pre- and postscripts:

```
string part;
for v within A:
  part := prescriptpart v;
% postscriptpart
endfor
```

Transformation parts:

```
numeric part;
for v within A:
  part := xpart v;
% ypart xxpart yypart xypart yxpart
endfor
```

Color model and/or color part

```
numeric part;
for v within A:
  part := colormodel v;
endfor
```

Color parts (RGB)

```
numeric part;
for v within A:
  part := redpart v;
  % bluepart greenpart
endfor
```

Color parts (CMYK)

```
numeric part;
for v within A:
  part := cyanpart v;
  % magentapart yellowpart blackpart
endfor
```

Color parts (grey)

```
numeric part;
for v within A:
  part := greypart v;
endfor
```

The dash part (which is itself a picture):

```
picture part;
for v within A:
  part := dashpart v;
endfor
```

The pen part:

```
pen part;
for v within A:
  part := penpart v;
endfor
```

The path part:

```
path part;
for v within A:
  part := pathpart v;
endfor
```

The text part of a label:

```
string part;
for v within A:
  part := textpart v;
endfor
```

The font part of a label:

```
string part;
for v within A:
  part := fontpart v;
endfor
```

Summary

That wraps up this article about the primitive operations on `pictures` and colors. As usual, many of the commands mentioned here are normally hidden behind macro definitions. In particular, as far as I know all of the MetaPost macro packages define a macro `draw` for adding stroked paths and a macro `fill` for adding filled paths to a picture. These are then used in combination with a predefined picture variable called `currentpicture`. Macros packages usually predefine the primary RGB colors `red`, `green`, and `blue` as well.

Maybe more higher level commands are available. For that, you will have to check the documentation of the macro package you are using.

Taco Hoekwater