Drawing Boxes with MetaPost

John D. Hobby

Abstract

This paper describes a package for drawing boxes of different shapes. The `boxes` package has been implemented as an extension to the MetaPost graphics language and is a mandatory part of every MetaPost installation.

Contents

1 Introduction

This document describes auxiliary macros not included in Plain MetaPost that make it convenient to do things that *pic* is good at [1]. What follows is a description of how to use the macros contained in the file `boxes.mp`. This file is included in a special directory reserved for MetaPost macros and support software[1] and can be accessed by giving the MetaPost command `input boxes` before any figures that use the box making macros. The syntax for the `input` command is

`input` ⟨file name⟩

where a final "`.mp`" can be omitted from the file name. The `input` command looks first in the current directory and then in the special macro directory. Users interested in writing macros may want to look at the `boxes.mp` file in this directory.

Since the advent of the `boxes` package several alternative packages for drawing boxes of all kinds have been developed by the MetaPost community. The most widely known ones are `MetaObj`, `MetaUML`, `expressg`, and `blockdraw_mp`. If you intend to create lots of structural drawings, flow charts, *etc.*, those packages might be an interesting ressource, too.

2 Rectangular Boxes

The main idea of the box-making macros is that one should say

`boxit.`⟨suffix⟩`(`⟨picture expression⟩`)`

where the ⟨suffix⟩ does not start with a subscript.[2] This creates pair variables ⟨suffix⟩`.c`, ⟨suffix⟩`.n`, ⟨suffix⟩`.e`, ... that can then be used for positioning the picture before drawing it with a separate command such as

`drawboxed(`⟨suffix list⟩`)`

The argument to `drawboxed` should be a comma-separated list of box names, where a box name is a ⟨suffix⟩ with which `boxit` has been called.

[1] The name of this directory is likely to be something like `/usr/lib/mp/lib`, but this is system dependent.

[2] Some early versions of the box making macros did not allow any subscripts in the `boxit` suffix.

For the command `boxit.bb(pic)`, the box name is `bb` and the contents of the box is the picture `pic`. In this case, `bb.c` the position where the center of picture `pic` is to be placed, and `bb.sw`, `bb.se`, `bb.ne`, and `bb.nw` are the corners of a rectangular path that will surround the resulting picture. Variables `bb.dx` and `bb.dy` give the spacing between the shifted version of `pic` and the surrounding rectangle, and `bb.off` is the amount by which `pic` has to be shifted to achieve all this.

When the `boxit` macro is called with box name b, it gives linear equations that force b`.sw`, b`.se`, b`.ne`, and b`.nw` to be the corners of a rectangle aligned on the x and y axes with the box contents centered inside as indicated by the gray rectangle in Figure 1. The values of b`.dx`, b`.dy`, and b`.c` are left unspecified so that the user can give equations for positioning the boxes. If no such equations are given, macros such as `drawboxed` can detect this and give default values. The default values for `dx` and `dy` variables are controlled by the internal variables `defaultdx` and `defaultdy`.

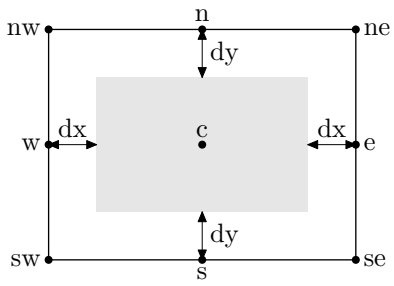

Figure 1: The relationship between the picture given to `boxit` and the associated variables. The picture is indicated by a gray rectangle.

If b represents a box name, `drawboxed(`b`)` draws the rectangular boundary of box b and then the contents of the box. This bounding rectangle can be accessed separately as `bpath` b, or in general

`bpath` ⟨box name⟩

It is useful in combination with operators like `cutbefore` and `cutafter` in order to control paths that enter the box. For example, if a and b are box names and p is a path from a`.c` to b`.c`,

`drawarrow` p `cutbefore bpath` a `cutafter bpath` b

draws an arrow from the edge of box a to the edge of box b.

Figure 2 shows a practical example including some arrows drawn with `cutafter bpath` ⟨box name⟩. It is instructive to compare Figure 2 to the similar figure in the pic manual [1]. The figure uses a macro

`boxjoin(`⟨equation text⟩`)`

to control the relationship between consecutive boxes. Within the ⟨equation text⟩, `a` and `b` represent the box names given in consecutive calls to `boxit` and the ⟨equation text⟩ gives equations to control the relative sizes and positions of the boxes.

For example, the second line of input for the above figure contains

```
boxjoin(a.se=b.sw; a.ne=b.nw)
```

This causes boxes to line up horizontally by giving additional equations that are invoked each time some box `a` is followed by some other box `b`. These equations are first invoked on the next line when box `a` is followed by box `ni`. This yields

```
a.se=ni.sw; a.ne=ni.nw
```

```
input boxes
beginfig(49);
boxjoin(a.se=b.sw; a.ne=b.nw);
boxit.a(btex\strut$\cdots$ etex);    boxit.ni(btex\strut$n_i$ etex);
boxit.di(btex\strut$d_i$ etex);      boxit.ni1(btex\strut$n_{i+1}$ etex);
boxit.di1(btex\strut$d_{i+1}$ etex); boxit.aa(btex\strut$\cdots$ etex);
boxit.nk(btex\strut$n_k$ etex);      boxit.dk(btex\strut$d_k$ etex);
drawboxed(di,a,ni,ni1,di1,aa,nk,dk); label.lft("ndtable:", a.w);
interim defaultdy:=7bp;
boxjoin(a.sw=b.nw; a.se=b.ne);
boxit.ba(); boxit.bb(); boxit.bc();
boxit.bd(btex $\vdots$ etex); boxit.be(); boxit.bf();
bd.dx=8bp; ba.ne=a.sw-(15bp,10bp);
drawboxed(ba,bb,bc,bd,be,bf); label.lft("hashtab:",ba.w);
vardef ndblock suffix $ =
  boxjoin(a.sw=b.nw; a.se=b.ne);
  forsuffixes $$=$1,$2,$3: boxit$$(); ($$dx,$$dy)=(5.5bp,4bp);
  endfor; enddef;
ndblock nda;  ndblock ndb;  ndblock ndc;
nda1.c-bb.c = ndb1.c-nda3.c = (whatever,0);
xpart ndb3.se = xpart ndc1.ne = xpart di.c;
ndc1.c - be.c = (whatever,0);
drawboxed(nda1,nda2,nda3, ndb1,ndb2,ndb3, ndc1,ndc2,ndc3);
drawarrow bb.c -- nda1.w;
drawarrow be.c -- ndc1.w;
drawarrow nda3.c -- ndb1.w;
drawarrow nda1.c{right}..{curl0}ni.c cutafter bpath ni;
drawarrow nda2.c{right}..{curl0}di.c cutafter bpath di;
drawarrow ndc1.c{right}..{curl0}ni1.c cutafter bpath ni1;
drawarrow ndc2.c{right}..{curl0}di1.c cutafter bpath di1;
drawarrow ndb1.c{right}..nk.c cutafter bpath nk;
drawarrow ndb2.c{right}..dk.c cutafter bpath dk;
x.ptr=xpart aa.c;   y.ptr=ypart ndc1.ne;
drawarrow subpath (0,.7) of (z.ptr..{left}ndc3.c) dashed evenly;
label.rt(btex \strut ndblock etex, z.ptr); endfig;
```

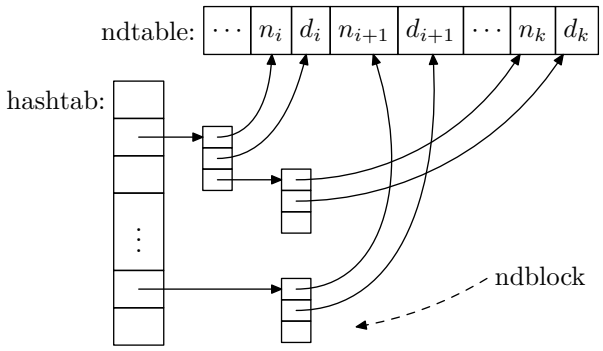

Figure 2: MetaPost code and the corresponding figure

The next pair of boxes is box `ni` and box `di`. This time the implicitly generated equations are

```
ni.se=di.sw; ni.ne=di.nw
```

This process continues until a new `boxjoin` is given. In this case the new declaration is

```
boxjoin(a.sw=b.nw; a.se=b.ne)
```

which causes boxes to be stacked below each other.

After calling `boxit` for the first eight boxes `a` through `dk`, the box heights are constrained to match but the widths are still unknown. Thus the `drawboxed` macro needs to assign default values to the ⟨box name⟩`.dx` and ⟨box name⟩`.dy` variables. First, `di.dx` and `di.dy` get default values so that all the boxes are forced to be large enough to contain the contents of box `di`.

The macro that actually assigns default values to `dx` and `dy` variables is called `fixsize`. It takes a list of box names and considers them one at a time, making sure that each box has a fixed size and shape. A macro called `fixpos` then takes this same list of box names and assigns default values to the ⟨box name⟩`.off` variables as needed to fix the position of each box. By using `fixsize` to fix the dimensions of each box before assigning default positions to any of them, the number of needing default positions can usually be cut to at most one.

Since the bounding path for a box cannot be computed until the size, shape, and position of the box is determined, the `bpath` macro applies `fixsize` and `fixpos` to its argument. Other macros that do this include

`pic` ⟨box name⟩

where the ⟨box name⟩ is a suffix, possibly in parentheses. This returns the contents of the named box as a picture positioned so that

`draw pic`⟨box name⟩

draws the box contents without the bounding rectangle. This operation can also be accomplished by the `drawunboxed` macro that takes a comma-separated list of box names. There is also a `drawboxes` macro that draws just the bounding rectangles.

Another way to draw empty rectangles is by just saying

`boxit`⟨box name⟩`()`

with no picture argument as is done several times in Figure 2. This is like calling `boxit` with an empty picture. Alternatively the argument can be a string expression instead of a picture expression in which case the string is typeset in the default font.

3 Circular and Oval Boxes

Circular and oval boxes are a lot like rectangular boxes except for the shape of the bounding path. Such boxes are set up by the `circleit` macro:

`circleit`⟨box name⟩`(`⟨box contents⟩`)`

where ⟨box name⟩ is a suffix and ⟨box contents⟩ is either a picture expression, a string expression, or ⟨empty⟩.

The `circleit` macro defines pair variables just as `boxit` does, except that there are no corner points ⟨box name⟩`.ne`, ⟨box name⟩`.sw`, etc. A call to

```
circleit.a(...)
```

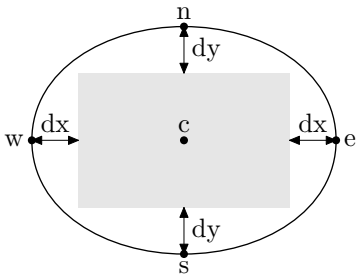

Figure 3: The relationship between the picture given to `circleit` and the associated variables. The picture is indicated by a gray rectangle.

gives relationships among points `a.c`, `a.s`, `a.e`, `a.n`, `a.w` and distances `a.dx` and `a.dy`. Together with `a.c` and `a.off`, these variables describe how the picture is centered in an oval as can be seen from the Figure 3.

The `drawboxed`, `drawunboxed`, `drawboxes`, `pic`, and `bpath` macros work for `circleit` boxes just as they do for `boxit` boxes. By default, the boundary path for a `circleit` box is a circle large enough to surround the box contents with a small safety margin controlled by the internal variable `circmargin`. Figure 4 gives a basic example of the use of `bpath` with `circleit` boxes.

```
vardef drawshadowed(text t) =
  fixsize(t);
  forsuffixes s=t:
    fill bpath.s shifted (1pt,-1pt);
    unfill bpath.s;
    drawboxed(s);
  endfor
enddef;

beginfig(51)
circleit.a(btex Box 1 etex);
circleit.b(btex Box 2 etex);
b.n = a.s - (0,20pt);
drawshadowed(a,b);
drawarrow a.s -- b.n;
endfig;
```

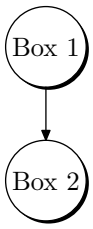

Figure 4: MetaPost code and the resulting figure. Note that the `drawshadowed` macro used here is not part of the `boxes.mp` macro package.

A full example of `circleit` boxes appears in Figure 5. The oval boundary paths around "Start" and "Stop" are due to the equations

`aa.dx=aa.dy;` and `ee.dx=ee.dy`

after

`circleit.ee(btex\strut Stop etex)` and `circleit.ee(btex\strut Stop etex)`.

The general rule is that `bpath.`c comes out circular if c`.dx`, c`.dy`, and c`.dx` $-$ c`.dy` are all unknown. Otherwise, the macros select an oval big enough to contain the given picture with the safety margin `circmargin`.

```
vardef cuta(suffix a,b) expr p =
  drawarrow p cutbefore bpath.a cutafter bpath.b;
  point .5*length p of p
enddef;

vardef self@# expr p =
  cuta(@#,@#) @#.c{curl0}..@#.c+p..{curl0}@#.c  enddef;

beginfig(52);
verbatimtex \def\stk#1#2{$\displaystyle{\matrix{#1\cr#2\cr}}$} etex
circleit.aa(btex\strut Start etex); aa.dx=aa.dy;
circleit.bb(btex \stk B{(a|b)^*a} etex);
circleit.cc(btex \stk C{b^*} etex);
circleit.dd(btex \stk D{(a|b)^*ab} etex);
circleit.ee(btex\strut Stop etex); ee.dx=ee.dy;
numeric hsep;
bb.c-aa.c = dd.c-bb.c = ee.c-dd.c = (hsep,0);
cc.c-bb.c = (0,.8hsep);
xpart(ee.e - aa.w) = 3.8in;
drawboxed(aa,bb,cc,dd,ee);
label.ulft(btex$b$etex, cuta(aa,cc) aa.c{dir50}..cc.c);
label.top(btex$b$etex, self.cc(0,30pt));
label.rt(btex$a$etex, cuta(cc,bb) cc.c..bb.c);
label.top(btex$a$etex, cuta(aa,bb) aa.c..bb.c);
label.llft(btex$a$etex, self.bb(-20pt,-35pt));
label.top(btex$b$etex, cuta(bb,dd) bb.c..dd.c);
label.top(btex$b$etex, cuta(dd,ee) dd.c..ee.c);
label.lrt(btex$a$etex, cuta(dd,bb) dd.c..{dir140}bb.c);
label.bot(btex$a$etex, cuta(ee,bb) ee.c..tension1.3 ..{dir115}bb.c);
label.urt(btex$b$etex, cuta(ee,cc) ee.c{(cc.c-ee.c)rotated-15}..cc.c);
endfig;
```

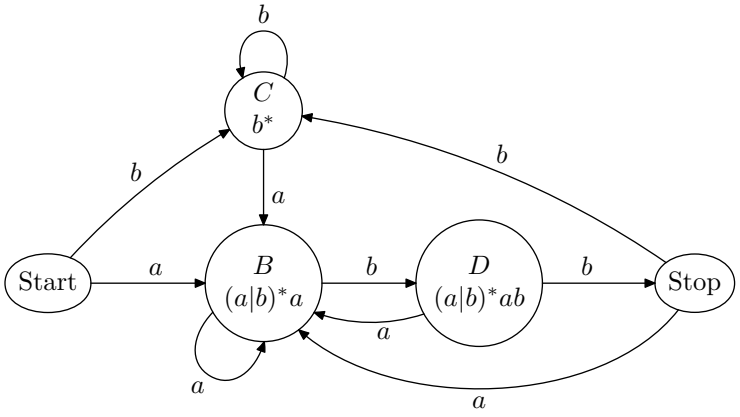

Figure 5: MetaPost code and the corresponding figure

A Reference manual

Tables 1 to 3 summarize macros, box variables and internal variables provided by the `boxes` package.

Table 1: Function-Like Macros

Name	Arguments	Result	Page	Explanation
`boxit`	suffix, picture	–	1	Define a box containing the picture
`boxit`	suffix, string	–	4	Define a box containing text
`boxit`	suffix, ⟨empty⟩	–	4	Define an empty box
`boxjoin`	equations	–	2	Give equations for connecting boxes
`bpath`	suffix	path	2	A box's bounding circle or rectangle
`circleit`	suffix, picture	–	4	Put picture in a circular box
`circleit`	suffix, picture	–	4	Put a string in a circular box
`circleit`	suffix, ⟨empty⟩	–	4	Define an empty circular box
`drawboxed`	list of suffixes	–	1	Draw the named boxes and their contents
`drawboxes`	list of suffixes	–	4	Draw the named boxes
`drawunboxed`	list of suffixes	–	4	Draw contents of named boxes
`fixpos`	list of suffixes	–	4	Solve for the size and position of the named boxes
`fixsize`	list of suffixes	–	4	Solve for size of named boxes
`pic`	suffix	picture	4	Box contents shifted into position

Table 2: Box variables

Variable	Explanation	Validity
⟨box name⟩`.c`	center point	
⟨box name⟩`.n`	top center point	
⟨box name⟩`.s`	bottom center point	
⟨box name⟩`.w`	center left point	
⟨box name⟩`.e`	center right point	
⟨box name⟩`.nw`	top left corner	`boxit` only
⟨box name⟩`.ne`	top right corner	`boxit` only
⟨box name⟩`.sw`	bottom left corner	`boxit` only
⟨box name⟩`.se`	bottom right corner	`boxit` only
⟨box name⟩`.dx`	horizontal clearance	
⟨box name⟩`.dy`	vertical clearance	
⟨box name⟩`.off`	actual location of box contents	

References

[1] Brian W. Kernighan. Pic—a graphics language for typesetting. In *Unix Research System Papers, Tenth Edition*, pages 53–77. AT&T Bell Laboratories, 1990.

Table 3: Internal variables with numeric values

Name	Page	Explanation
`circmargin`	5	clearance around contents of a circular or oval box
`defaultdx`	2	usual horizontal space around box contents (default `3bp`)
`defaultdy`	2	usual vertical space around box contents (default `3bp`)

Index