Object-Oriented Graphics with MetaObj

Abstract
MetaOBJ is a macro package for MetaPost [1], a programming language for graphics producing PostScript output, based on the well-known MetaFont. MetaOBJ is written and maintained by Denis B. Roegel. It has been released under the LPPL (LaTeX Project Public License) and is available from CTAN.
The cool thing about MetaOBJ is that it provides very high-level object-oriented macros which simplify the construction of complicated drawings by defining objects of arbitrary complexity and combining them to larger structures. This is already reflected in the name of the package: MetaOBJ is a shortcut for "MetaPost Objects". The cover picture of the MetaOBJ manual [2], reproduced in figure 1, gives a first impression of the kind of graphics which can be produced.

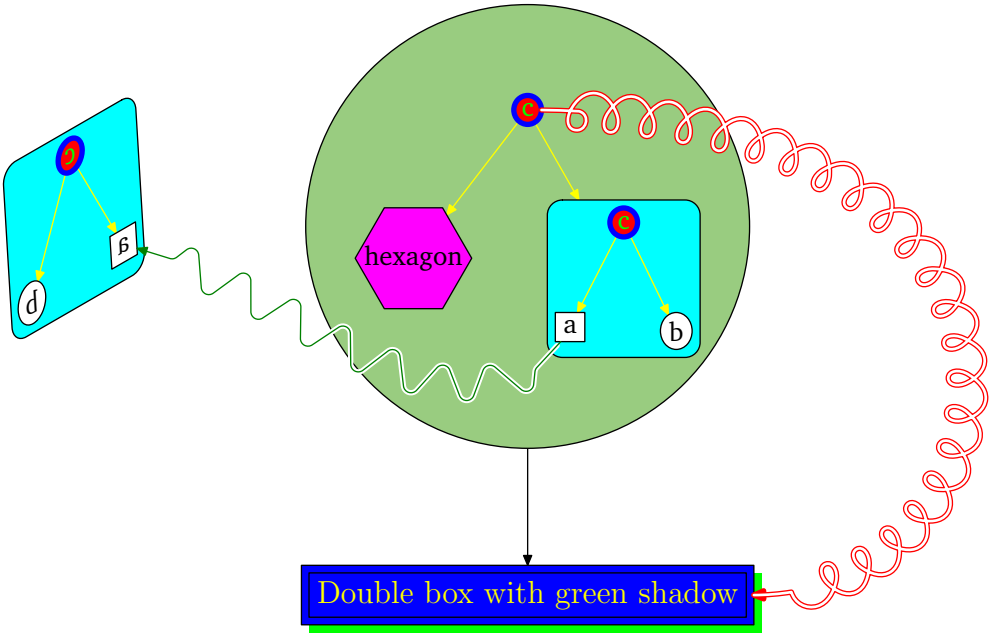


Figure 1 Title graphic of the MetaOBJ manual, taken from [2].

Introduction

It is not the aim of this paper to cover all possibilities and intricate details of MetaOBJ. Although many examples and topics are borrowed from the excellent user manual [2], the following cannot be and is not intended as a replacement. I will rather try to portray MetaOBJ from my point of view – the point of view of a user interested in getting qualitatively excellent graphics done relatively easily. So, let me start by saying some words about my TeX-background.

I would characterize myself as an experienced, but not expert user of TeX. During my studies, I used LaTeX for nearly all kinds of documents I had to produce. For

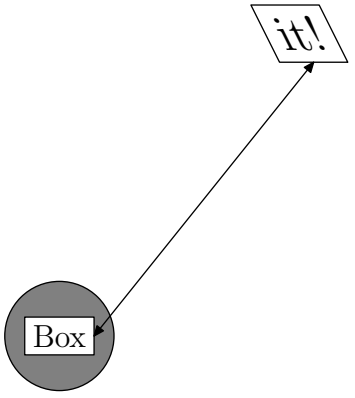
graphics, I used either PSTRICKS or external software (which is of course annoying because of difficulties concerning consistency and exactness). About three years ago, I made first contact with ConTEXt and was immediately mesmerized by its possibilities, the rapid development and future potential. And of course, I came to know MetaPost this way. Since the few graphics I have to produce today happen to be of a rather schematical and technical kind, there can presumably be no better and more satisfactory way to create them than by using a graphic programming language like MetaPost. Therefore, the system has become my favored choice.

When I first saw MetaOBJ, I started to use it because it provides features with similar output to many PSTRICKS commands, works well with ConTEXt's MetaPost integration, and does not hamper direct PDF output. Anybody plunging deeper into MetaOBJ will notice that this does not do full justice to its true strengths and intentions, but, as stated above, in the same way my intention is to give an impression of what is available to the user easily, without focussing too much on object-orientation, implementation or extensions.

After some small examples to quicken the reader's appetite, I am going to show how objects can be combined into more complex graphics like the MetaOBJ cover graphic shown in figure 1 and how some PSTRICKS-like graphics can be produced. Finally, a custom class of objects will be defined to produce schematic representations of certain molecules.

Example Graphics

Let us start right now with a first MetaOBJ graphic. The code is virtually self-explanatory. But as simple as it is, figure 2 already shows a lot of MetaOBJ features:

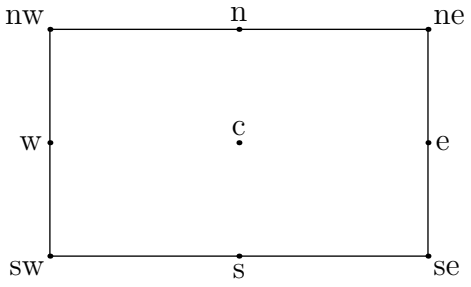



```
newBox.box1(btex Box etex);
newCircle.circ(box1) "circmargin(2mm)",
  "filled(true)", "fillcolor(.5white)";
newBox.box2(btex it! etex);
scaleObj(box2, 3/2);
slantObj(box2, -1/2);
box2.w - circ.e = (2cm, 4cm);
box1.c = origin;
drawObj(circ, box2);
drawdblarrow box1.e -- box2.s;
```

Figure 2 A first example.

- Objects are created as instances of classes by `newSomething` constructors. In the example, we have objects of classes `Box` and `Circle`.
- Objects have names by which they can be accessed: `box1`, `box2` and `circ` in the example.
- Most objects of the MetaOBJ standard library can be customized by the use of options like `filled` or `fillcolor`.
- Objects accept and store all kind transformations supported by MetaPost: `scaleObj`, `slantObj` and so on with self-explanatory names.
- Objects can contain pictures (and more, like paths and other variables, as we will see later). Most importantly, they can contain other objects: `box1` is a subobject of `circ`.
- Objects are floating, relative positioning to each other is possible: `box2.w - circ.e = (2cm, 4cm)`. Thus, objects can be only partly defined.

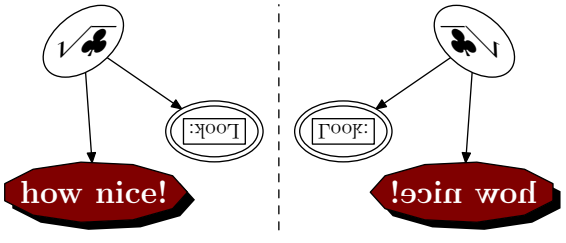
- Standard MetaOBJ objects provide a standard interface. They have a set of points in the directions of the compass, as shown in figure 3. The interface for accessing the points and setting the position of the object is compatible to the the `boxes.mp` package by John D. Hobby.




```
show_empty_boxes := true;
newEmptyBox.e(5cm, 3cm)
  "framed(true)";
e.c = origin;
drawObj(e);
```

Figure 3 Points of the MetaOBJ standard interface.

- Specifying one point of an object defines the position of all points of the object (at least for the standard objects): `box1.c = origin`. As we will see, objects store equations determining the relative positions of their points.
- Once absolutely positioned, they can be drawn with a call to `drawObj`, which internally calls the respective drawing function of the class, i. e. `drawCircle` or `drawBox`.
- Finally, the line `drawdblarrow box1.e - box2.s` shows that making an object a subobject of another object does not hamper us from accessing it directly. This is true for any depth of nesting and is an important advantage compared to approaches which do not preserve the object structure, e. g. by storing just pictures inside.




```
newDEllipse.ell(new_Box_(btex Look: etex)());
save t;
rotateObj(ell, 180);
scaleObj(ell, 2/3);
newPolygon.pol(btex \bf how nice! etex, 11)
  "filled(true)", "fillcolor(.5red)",
  "picturecolor(white)", "shadow(true)";
newCircle.root(btex $\sqrt\clubsuit$ etex);
slantObj(root, 1/2);
newTree.t(root)(pol, ell);
rebindVisibleObj(t);
t.c = origin;
drawObj(t);
duplicateObj(t2, t);
reflectObj(t2)(origin, (0,1));
draw t.se + (2mm, 0) -- t.ne + (2mm, 0)
  dashed evenly;
t2.e = t.e + (4mm, 0);
drawObj(t2);
```

Figure 4 Another example graphic.

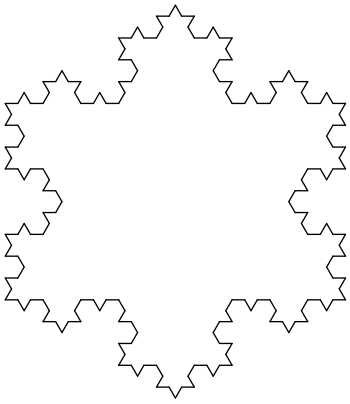
In addition to the above, the second example (figure 4) shows some more points worth mentioning:

- There are *streamlined* versions of the standard constructors, returning an object of the respective type. They can be used to pass the new object to another object immediately, like the `Box` in `newDEllipse.ell(new_Box_(btex`

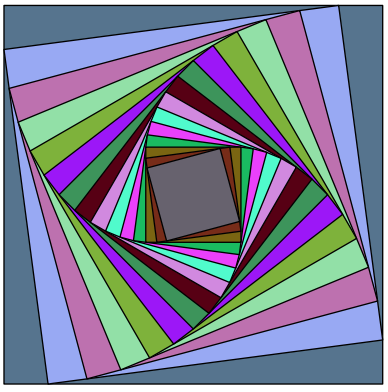
`Look: etex)())`, or to assign the object to a variable. Streamlined constructors are distinguished from the normal ones by underscores as in `new_Box` and have a slightly different syntax.

- An in-depth, identical yet independent clone can be created by `duplicateObj`.
- Classes for regular arrangement of objects can be defined, such as the `Tree` and `Matrix` classes of the standard library.

There are of course many more features which cannot all be commented here. Two last examples in figure 5 show that it is also possible to define recursive objects. `VonKochFlake` and `RecursiveBox` are part of the standard MetaOBJ library.



```
newVonKochFlake.flake(3);
scaleObj(flake,.45);
flake.c=origin;
drawObj(flake);
```



```
newRecursiveBox.rb(14)
  "rotangle(7.5)";
randomizeRecursiveBox(rb);
scaleObj(rb, .2);
rb.c = origin;
drawObj(rb);
```

Figure 5 `VonKochFlake` and `RecursiveBox`

Actually, I have cheated a little bit, since the following code has also been used to produce the colorful `RecursiveBox`.

```
def randomcolor =
     (uniformdeviate 1, uniformdeviate 1, uniformdeviate 1)
  enddef;

  vardef randomizeRecursiveBox(suffix n) =
    if known n.sub :
      randomizeRecursiveBox(obj(n.sub));
    fi;
    ExecuteOptions(n)("fillcolor(randomcolor)");
  enddef;

  setObjectDefaultOption("RecursiveBox")("filled")(true);
```

Connections and Graphs

The previous examples have already shown many of the basic classes of the MetaOBJ library, and how these can be combined to composite objects. A common need that has not yet been addressed is how to draw connections between objects. And, what about that promised PSTRICKS functionality? Indeed, the commands for connecting objects in MetaOBJ are very similar to those from PSTRICKS. Figure 6 summarizes

some of the possibilities. As you can see, the connections can operate on objects as in `ncline(A)(B)` as well as on points, which means that you can write something like `nccurve(A)(origin)`.

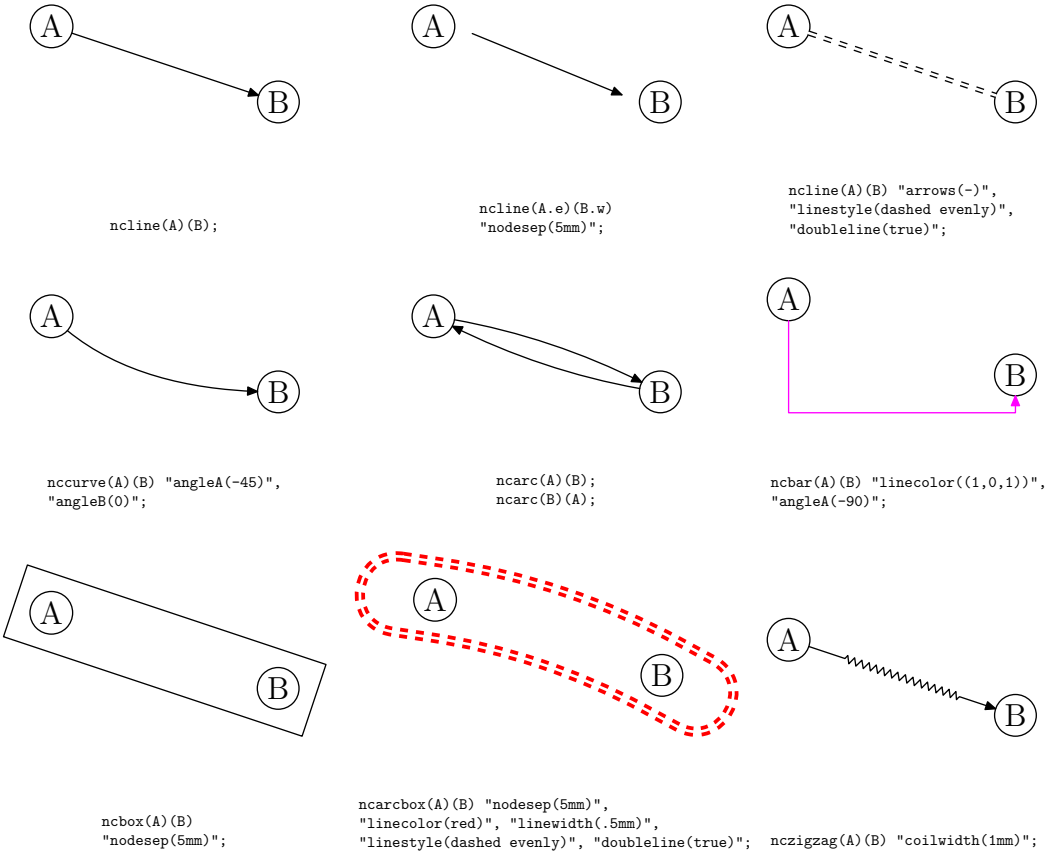


Figure 6 Some of the node connections defined by MetaOBJ. For a complete overview of types and options, please refer to the manual.

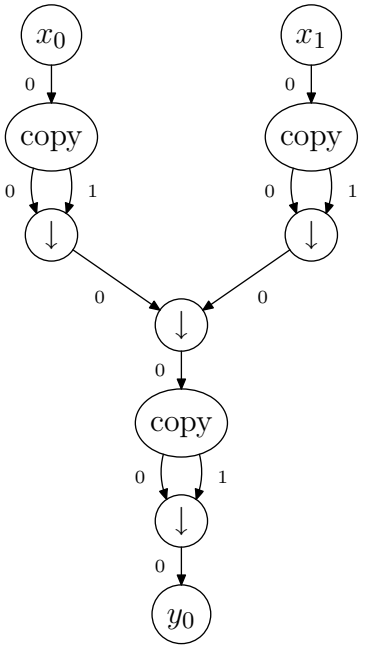
Several options trigger the appearance of the connections. It should also be mentioned that the examples show only one type of connections: the *immediate* ones, which are drawn instantly, meaning that the object positions have to be already defined. MetaOBJ also provides *deferred* versions of all connections, which are memorized with an object and are drawn when the object is drawn. The commands are identical except for the object name as a suffix: `ncline.A(A)(B)` stores the connection between objects `A` and `B` together with object `A`.

The MetaOBJ manual title graphic (see figure 1) uses nearly everything mentioned so far and a little bit more, so it may be a good time to reveal its source. The code should be easily understandable by now, and hopefully, you will be delighted to see how a relatively complicated graphic can be programmed in such an intuitive and simple manner.

```
newBox.a("a");
  newEllipse.b("b");
  newEllipse.c("c") "filled(true)", "fillcolor(red)","picturecolor(green)",
                    "framecolor(blue)","framewidth(2pt)";
  newTree.t(c)(a,b) "linecolor((1,1,0))";
  newBox.aa(t) "filled(true)", "fillcolor((0,1,1))","rbox_radius(2mm)";
  aa.c=origin;
```

```
newHexagon.xa("hexagon") "fit(false)","filled(true)","fillcolor((1,0,1))";
newEllipse.xc("c") "filled(true)", "fillcolor(red)","picturecolor(green)",
                   "framecolor(blue)","framewidth(2pt)";
newTree.xt(xc)(xa,aa) "linecolor((1,1,0))";
newCircle.xaa(xt) "filled(true)", "fillcolor((.6,.8,.5))";
newDBox.db(btex Double box with green shadow etex)
        "shadow(true)", "shadowcolor(green)",
        "filled(true)","fillcolor(blue)","picturecolor((1,1,0))";
newTree.nt(xaa)(db);
drawObj(nt);
nccoil(xc)(db) "angleA(0)","angleB(180)",
  "coilwidth(5mm)","linetension(0.8)","linecolor(red)",
  "doubleline(true)","posB(e)";
duplicateObj(dt,aa);
reflectObj(dt,origin,up);
slantObj(dt,.5);
rotateObj(dt,30);
dt.c=nt.c-(6cm,-1cm);
drawObj(dt);
nczigzag(a)(treepos(obj(dt.sub))(1))
  "angleA(-120)","coilwidth(7mm)","linecolor(.5green)","linearc(1mm)",
  "border(2pt)";
```

Apart from trees, which have been used in the examples so far, matrices are another way of arranging objects in a regular way provided by MetaOBJ (as by PSTRICKS). For simplification of connections between tree nodes and matrix elements, there are special connection commands which refer to the connected elements by their coordinates in the container object. The example in figure 7 shows how this is done for a matrix. In addition, you can see that labels can be added to connections (and to all other objects) easily.




```
newCircle.xnull(btex $x_0$\strut etex);
newCircle.xeins(btex $x_1$\strut etex);
newCircle.ynull(btex $y_0$\strut etex);
for i=1 upto 4 :
  newCircle.pfeil[i](btex $\downarrow$\strut etex);
  newEllipse.copy[i](btex copy\strut etex);
endfor
newMatrix.mat(7,3)(
  xnull,nb,xeins, copy1,nb,copy2, pfeil1,nb,pfeil2,
  nb,pfeil3,nb,   nb,copy3,nb,    nb,pfeil4,nb,
  nb,ynull,nb) "hsep(5mm)", "vsep(5mm)";
mcline.Obj(mat)(1,1,2,1) "name(a)";
ObjLabel.Obj(mat)(btex $\scriptscriptstyle0$ etex)
  "labpathname(a)", "labdir(lft)";

% ... (other labels, not shown here)

mcline.Obj(mat)(6,2,7,2) "name(l)";
ObjLabel.Obj(mat)(btex $\scriptscriptstyle0$ etex)
  "labpathname(l)", "labdir(lft)";
mat.c=origin;
drawObj(mat);
```

Figure 7 Example graph showing the use of matrices, matrix connections and labels.

Custom Objects

At some point, you will probably wish to create your own classes. In principle, all you need to define for a `Nice` class is a constructor `newNice`, a drawing function `drawNice` and a bounding path `BpathNice`. MetaOBJ helps you with many predefined standard components and functions.

Before creating new objects, let us first look at a simple standard object, `EmptyBox`. Here is the constructor:

```
vardef newEmptyBox@#(expr dx,dy) text options=
     ExecuteOptions(@#)(options);
     assignObj(@#,"EmptyBox");
     StandardInterface;
     ObjCode StandardEquations,
       "@#ise-@#isw=(" & decimal dx & ",0)",
       "@#ine-@#ise=(0," & decimal dy & ")";
   enddef;
```

The constructor takes two dimensions as arguments – used as width and height of the box – as well as a potentially empty list of options. These are processed by `ExecuteOptions`. `assignObj` takes care of making the new object a member of the correct class (and some more internal things). Then, `StandardInterface` declares the standard points shown in figure 3 as well as, unmentioned before, a second set of these points (`ine`, `in`, `ine`, and so on) for use from within the object (don't bother about the reasons for now). The equations determining the relations between the object points are stored as strings using `ObjCode`, where `StandardEquations` is an abbreviation for `(PureStandardEquations & StandardInnerEquations)`, which in turn are defined as

```
def PureStandardEquations=
     ("@#se-@#sw=@#ne-@#nw;" & % parallelogram equation
     "xpart(@#se-@#ne)=0;" &
     "ypart(@#se-@#sw)=0;" &
     "@#n=.5[@#ne,@#nw];" &    % North
     "@#s=.5[@#se,@#sw];" &    % South
     "@#e=.5[@#ne,@#se];" &    % East
     "@#w=.5[@#nw,@#sw];" &    % West
     "@#c=.5[@#n,@#s];" )      % Center
   enddef;

   def StandardInnerEquations=
      ("@#ine=@#ne;@#inw=@#nw;@#isw=@#sw;@#ise=@#se;@#in=@#n;@#is=@#s;" &
     "@#ie=@#e;@#iw=@#w;@#ic=@#c;")
   enddef;
```

Drawing an `EmptyBox` is simple – it is shown only if enabled (and you won't see much unless it is filled or framed). In any case, paths stored with the object are drawn:

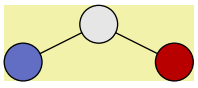
```
def drawEmptyBox(suffix n)=
     if show_empty_boxes:
       drawFramedOrFilledObject_(n);
     fi;
     drawMemorizedPaths_(n);
   enddef;
```

The bounding path (used e. g. for filling) is also very simple:

```
def BpathEmptyBox(suffix n)=StandardBpath(n) enddef;
```

an abbreviation for `(n.inw-n.isw-n.ise-n.ine-cycle)`. And that's it. Of course, the definitions get more complicated for more sophisticated objects, but the principle remains the same.

Now we have everything to go ahead and create our own classes. I once came across the need to draw a schematic representation for a cyclic peptide. Maybe you know that peptide molecules are polymers, composed of monomers called amino acids. Anyway, I wanted to represent each amino acid as three circles (the main chain atoms) connected by lines (covalent bonds), with a color box in the background, as shown in figure 8.



```
newAminoAcid.X;
X.c=origin;
drawObj(X);
```

Figure 8 Schematic representation of an amino acid.

Admittedly, the code is not very nice, and of course I would do it differently if I did it again. But due to lack of time for a rewrite, here it is. After some setup:

```
% ad : atom circle diameter
  % dx : x distance between two atom centers
  % dy : y distance

  numeric ad, dx, dy, hyp, offset;
    ad = 5mm;
    dx = 2ad;
    dy = 1/2dx;
    hyp = sqrt(dx**2+dy**2);

  % ahlength : length of arrow heads
  ahlength := 3/4ad;

  % colors for atoms N, C, CA and background
  color NColor, CColor, CAColor, AAColor;
    NColor = (.38,.431,.769);
    CAColor = .9white;
    CColor = (.722,0,0);
    AAColor = (.95,.95,.67);
```

we define the constructor for an `AminoAcid`:

```
vardef newAminoAcid@# =
    assignObj(@#,"AminoAcid");
    StandardInterface;
    save N,CA,C; string N,CA,C;
    N=newobjstring_; CA=newobjstring_; C=newobjstring_;
    newCircle.obj(N)(nullpicture) "circmargin(.5ad)",
      "filled(true)","fillcolor(NColor)";
    newCircle.obj(CA)(nullpicture) "circmargin(.5ad)",
      "filled(true)","fillcolor(CAColor)";
    newCircle.obj(C)(nullpicture) "circmargin(.5ad)",
      "filled(true)","fillcolor(CColor)";
    SubObject(N,obj(N)); SubObject(CA,obj(CA)); SubObject(C,obj(C));
    ObjCode StandardEquations,
      "@#isw=obj(@#N).sw",
      "@#ine=(xpart obj(@#C).e,ypart obj(@#CA).n)",
      "obj(@#CA).c-obj(@#N).c=(dx,dy)",
      "obj(@#C).c-obj(@#CA).c=(dx,-dy)";
    StandardTies;
    ncline.@#(obj(@#N))(obj(@#CA)) "arrows(-)";
    ncline.@#(obj(@#CA))(obj(@#C)) "arrows(-)";
  enddef;
```

As you can see, three `Circles` are created for the atoms and registered as subobjects of the `AminoAcid` using the `SubObject` function. `newobjstring_` provides a new

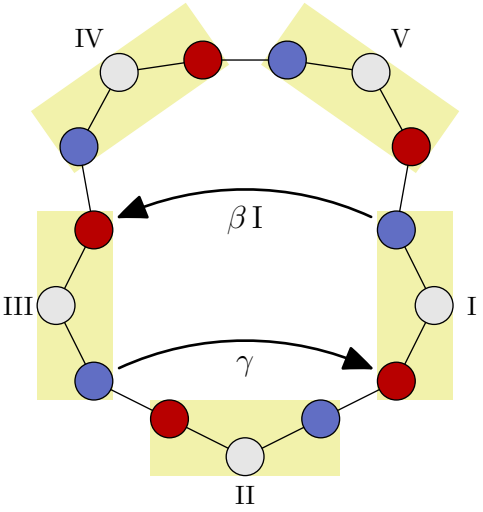
unique name for each subobject. In addition to the `StandardEquations`, there are some relations arranging the subobjects. `StandardTies` memorizes the connection between the object and its subobjects. Finally, the lines representing the bonds are added to the `AminoAcid` object using `ncline`.

```
def BpathAminoAcid(suffix n)=StandardBpath(n) enddef;

  def drawAminoAcid(suffix n)=
    fill BpathAminoAcid(n) withcolor AAColor;
    drawObj(obj(n.N),obj(n.CA),obj(n.C));
    drawMemorizedPaths_(n);
  enddef;
```

The bounding path is not special at all. `drawAminoAcid` is straightforward, too: the background is filled, and the subobjects as well as the stored paths are drawn.

Now, the amino acids had to be combined to a peptide, as shown in figure 9 for a pentapeptide.




```
newPentaPeptide.p;
lowerTurn.p(btex $\gamma$ etex);
upperTurn.p(btex $\beta\,$I etex);
p.c=origin;
drawObj(p);
```

Figure 9 Schematic drawing of pentapeptide.

The `PentaPeptide` class was defined as follows.

```
vardef newPentaPeptide@#=
     assignObj(@#,"PentaPeptide");
     StandardInterface;
     save I,II,III,IV,V; string I,II,III,IV,V;
     forsuffixes s = I, II, III, IV, V :
       s=newobjstring_;
       newAminoAcid.obj(s);
       SubObject(s,obj(s));
     endfor
     rotateObj(obj(IV),35); rotateObj(obj(V),-35); rotateObj(obj(I),270);
     rotateObj(obj(II),180); rotateObj(obj(III),90);
     ObjCode StandardEquations,
        "xpart .5[obj(obj(@#III).C).c,obj(obj(@#I).N).c]=xpart obj(@#II).c",
       "ypart obj(obj(@#III).C).c=ypart obj(obj(@#I).N).c",
        "obj(obj(@#I).C).c-obj(obj(@#II).N).c=(dx,dy)",
       "xpart .5[obj(@#V).c,obj(@#IV).c]=xpart obj(@#II).c",
        "obj(obj(@#V).N).c-obj(obj(@#IV).C).c=(hyp,0)",
       "ypart obj(obj(@#IV).N).c-ypart obj(obj(@#III).C).c="
       &"sqrt(hyp**2-(xpart obj(obj(@#III).C).c-xpart obj(obj(@#IV).N).c)**2)",
        "@#isw=(xpart obj(@#III).n,ypart obj(@#II).n)",
       "@#ine=(xpart obj(@#I).n,ypart obj(@#IV).n)";
```

```
    StandardTies;
  ncline.@#(obj(obj(@#I).C))(obj(obj(@#II).N)) "arrows(-)";
   ncline.@#(obj(obj(@#II).C))(obj(obj(@#III).N)) "arrows(-)";
  ncline.@#(obj(obj(@#III).C))(obj(obj(@#IV).N)) "arrows(-)";
  ncline.@#(obj(obj(@#IV).C))(obj(obj(@#V).N)) "arrows(-)";
  ncline.@#(obj(obj(@#V).C))(obj(obj(@#I).N)) "arrows(-)";
  ObjLabel.obj(@#I)(btex I etex) "labpoint(n)", "labshift((.5ad,0))";
  ObjLabel.obj(@#II)(btex II etex) "labpoint(n)", "labshift((0,-.5ad))";
  ObjLabel.obj(@#III)(btex III etex) "labpoint(n)", "labshift((-.5ad,0))";
  ObjLabel.obj(@#IV)(btex IV etex) "labpoint(n)", "labshift((-.5ad,.5ad))";
  ObjLabel.obj(@#V)(btex V etex) "labpoint(n)", "labshift((.5ad,.5ad))";
enddef;

def drawPentaPeptide(suffix n)=
  drawObj(obj(n.I),obj(n.II),obj(n.III),obj(n.IV),obj(n.V));
  drawMemorizedPaths_(n);
enddef;
```

And finally two convenient abbreviations.

```
vardef lowerTurn@#(expr name)=
    nccurve.@#(obj(obj(@#.III).N))(obj(obj(@#.I).C)) "name(turnlow)",
      "angleA(30)", "angleB(-30)", "nodesepA(.5ad)", "nodesepB(.5ad)",
      "linewidth(2pt)";
    ObjLabel.@#(name) "labpathname(turnlow)", "labdir(bot)";
  enddef;

  vardef upperTurn@#(expr name)=
    nccurve.@#(obj(obj(@#.I).N))(obj(obj(@#.III).C)) "name(turnup)",
      "angleA(150)", "angleB(-150)", "nodesepA(.5ad)", "nodesepB(.5ad)",
      "linewidth(2pt)";
    ObjLabel.@#(name) "labpathname(turnup)", "labdir(bot)";
  enddef;
```

Several wishes remain unaddressed so far, for example parameterization of labels. But this is another story, which may be told another day.

Final Remarks

In the course of writing this article, I had to realize that it is not trivial to point out the distinctive features of MetaOBJ without getting too lengthy or quoting the whole manual. I hope that some of the Maps readers will feel like trying MetaOBJ themselves after reading. In this case, I have achieved my aim. My thanks and all credits for MetaOBJ go to Denis B. Roegel for creating this awesome package and making it publicly available.

References

[1] J. D. Hobby, A User's Manual for MetaPost, *AT&T Bell Laboratories Computing Science Technical Report 162*, **1992**.

[2] D. B. Roegel, The MetaOBJ tutorial and reference manual, **2002**, `http://www.tug.org/tex-archive/graphics/metapost/contrib/macros/metaobj/doc/momanual.pdf`.

Eckhart W. Guthöhrlein
eckhart.guthoehrlein@uni-bielefeld.de