# Graphics Applications

## Creating 3D animations with METAPOST

Denis Roegel

### Abstract

METAPOST can be used to create animations. We show here an example of animation of polyhedra, introducing the `3d` package.

### Introduction

METAPOST (Hobby (1992); see also the description in Goossens et al. (1997)) is a drawing language very similar to METAFONT, but whose output is PostScript. METAPOST is especially suited for geometrical and technical drawings, where a drawing can naturally be decomposed in several parts, related in some logical way. Knuth is using METAPOST for the revisions of and additions to *The Art Of Computer Programming* (Knuth, 1997), and it is or will be a component of every standard TEX distribution.

Unfortunately, METAPOST is still quite bare and the user is only offered the raw power — a little bit like the TEX user who only has plain TEX at his/her disposal. The lack of libraries is certainly due to the infancy of METAPOST (which came in the public domain at the beginning of 1995) and thus to the small number of its users.

In this paper, we present a way to produce animations using METAPOST. The technique is quite general and we illustrate it through the `3d` package.

### Animations

The World Wide Web has accustomed us to various animations, especially `java` animations. Common components of web pages are animated GIF images.

Producing animations in METAPOST is actually quite easy. A number of $n$ images will be computed and their sequence produces the animation. The animation will be similar to a movie, with no interaction. More precisely, if `an_image(`$i$`)` produces a picture parameterized by $i$, it suffices to wrap this macro between `beginfig` and `endfig`:

```
def one_image_out(expr i)=
  beginfig(<figure number>);
    an_image(i);
  endfig;
enddef;
```

and to loop over `one_image_out`:

```
for j:=1 upto 100:one_image_out(j);endfor;
```

Assuming that `<figure number>` is equal to the parameter of `an_image`, the compilation of this program will produce a hundred files with extensions `.1`, `.2`, ..., `.100`. All these files are PostScript files and all we need to do is to find a way to collate them in one piece. How to do this depends on the operating system. On UNIX for instance, one can use `Ghostscript` to transform a PostScript file into `ppm` and then transform each `ppm` file into GIF with `ppmtogif`. These programs are part of the `NETPBM` package (Davidsen, 1993). Finally, a program such as `gifmerge` (Müller, 1996) creates an animated GIF file (GIF89A) out of the hundred individual simple GIFs. However, various details must be taken care of. For instance, only a part of `Ghostscript`'s output is needed and selection can be made with `pnmcut`.

The whole process of creating an animation out of METAPOST's outputs can be summed up in a shell script, similar to the one in figure 1. As we will see, this script (including the arguments of `awk` and `pnmcut`) can be generated automatically by METAPOST itself.

### Objects in space

**Introduction** The author applied this idea to the animation of objects in space. The macros in the `3d.mp` package[1] provide a basis for the representation of three-dimensional objects. The basic components of the objects are the points or the vectors. Both are stored as triplets. More precisely, we have three arrays[2] of type `numeric`:

```
numeric vect[]x,vect[]y,vect[]z;
```

Vector $i$'s components are `vect[`$i$`]x`, `vect[`$i$`]y` and `vect[`$i$`]z`. It is then straightforward to define the usual operations on vectors using this convention. For instance, vector addition is defined as:

```
def vect_sum(expr k,i,j)=
  vect[k]x:=vect[i]x+vect[j]x;
  vect[k]y:=vect[i]y+vect[j]y;
  vect[k]z:=vect[i]z+vect[j]z;
enddef;
```

[1] On CTAN, under `graphics/metapost/macros/3d`. The code is documented with `MFT` (Knuth, 1989) and illustrated with METAPOST. This paper describes version 1.0 of the macros.

[2] METAPOST has a few simple types such as `numeric`, `boolean`, `string`, `path`, .... It also has pairs (`pair`) and triples (`color`). We might have cheated and stored points as colors, but instead, we found it interesting to illustrate a construction equivalent to PASCAL's records or C's structures. In METAPOST, instead of having a list or an array of structures, we use several lists or arrays, so that a record is a cross-section over several arrays.

```
#! /bin/sh

/bin/rm -f animpoly.log
for i in `ls animpoly.*| grep 'animpoly.[0-9]'`;do
echo $i
echo '=============='
# shift each picture so that it lies in the page:
awk  < $i '{print} /^%%Page: /{print "172 153 translate\n"}' > $i.ps
# produce ppm format:
gs -sDEVICE=ppmraw -sPAPERSIZE=a4 -dNOPAUSE -r36 -sOutputFile=$i.ppm -q -- $i.ps
/bin/rm -f $i.ps
# produce gif:
ppmquant 32 $i.ppm | pnmcut 15 99 141 307 | ppmtogif > `expr $i.ppm : '\(.*\)ppm'`gif
/bin/rm -f $i.ppm
done
/bin/rm -f animpoly.gif
# merge the gif files:
gifmerge -10 -l1000 animpoly.*.gif > animpoly.gif
/bin/rm -f animpoly.*.gif
```

**Figure 1**: Script created by METAPOST (with some additional comments)

Often, we need some scratch vectors or vectors local to a macro. A simple vector allocation mechanism solves the problem: we use a stack of vectors and we reserve and free vectors only on top of the stack. For instance, the allocation of a vector is defined by:

```
def new_vect=incr(last_vect_) enddef;
```

where `last_vect_` is the index of the top of the stack. Hence, a vector is manipulated by its index on the stack. Writing `v:=new_vect;` lets `v` be the index of the newly allocated vector.

Freeing a vector is also easy and is only allowed at the top of the stack:

```
def free_vect(expr i)=
  if i=last_vect_:
     last_vect_:=last_vect_-1;
  else: errmessage("Vector " &
     decimal i & " can't be freed!");
  fi;
enddef;
```

How these macros are used is made explicit in the `vect_rotate` macro which does a rotation of a vector `v` around a vector `axis` by an angle `alpha`. This rotation is illustrated in figure 2. $\vec{v}$ is written as the sum of $\vec{h}$ and $\vec{a}$ where $\vec{h} \perp \vec{a}$. If $\vec{b}$ is $\overrightarrow{axis}/\|\overrightarrow{axis}\|$, $\vec{c}$ is computed as the vector product of $\vec{b}$ and $\vec{a}$ and $\vec{a}$ is then rotated in a simple way resulting in $\vec{f}$.

The vectors declared with `new_vect` are freed in the inverse order. The `vect_rotate` macro makes use of a few other macros: `vect_mod` computes the modulus of a vector; `vect_dprod(a,b)` is the dot product of vectors `a` and `b`; `vect_mult(b,a,x)` lets vector `b` equal vector `a` multiplied by the scalar `x`; `vect_sum` and `vect_diff` compute as their first argument the sum or the difference of the two other vectors; `vect_prod(c,a,b)` lets vector `c` equal the vectorial product of vectors `a` and `b`. These macros are described in appendix A.

```
vardef vect_rotate(expr v,axis,alpha)=
  save v_a,v_b,v_c,v_d,v_e,v_f;
  v_a:=new_vect;v_b:=new_vect;
  v_c:=new_vect;v_d:=new_vect;
  v_e:=new_vect;v_f:=new_vect;
  v_g:=new_vect;v_h:=new_vect;
  vect_mult(v_b,axis,1/vect_mod(axis));
  vect_mult(v_h,v_b,vect_dprod(v_b,v));
  vect_diff(v_a,v,v_h);
  vect_prod(v_c,v_b,v_a);
  vect_mult(v_d,v_a,cosd(alpha));
  vect_mult(v_e,v_c,sind(alpha));
  vect_sum(v_f,v_d,v_e);
  vect_sum(v,v_f,v_h);
  free_vect(v_h);free_vect(v_g);
  free_vect(v_f);free_vect(v_e);
  free_vect(v_d);free_vect(v_c);
  free_vect(v_b);free_vect(v_a);
enddef;
```

The `3d` package defines other macros in order to set the observer, to compute a reference matrix, etc. Provision is given for manipulating objects.

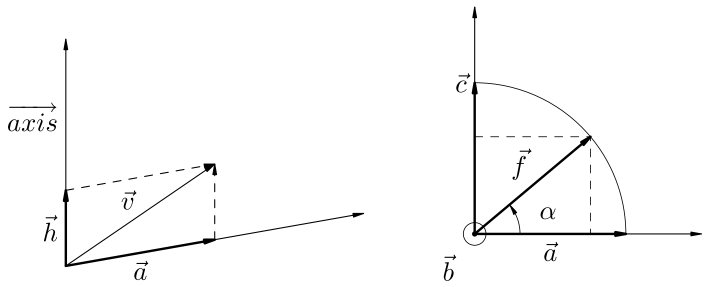



**Figure 2**: Vector rotation

**Objects and classes** The 3d package understands a notion of *class*. A *class* is a parameterized object. For instance, we have the class of regular tetrahedra, the class of regular cubes, etc. Our classes are the lowest level of abstraction and classes can not be composed. They can only be *instanciated*. When we need a specific tetrahedron, we call a generic function to create a tetrahedron, but with an identifier specific to one instance.

A class is a set of vertices in space, together with a way to draw faces, and therefore edges. The author's focus was to manipulate (and later animate) polyhedra. As an example, the `poly.mp` package provides the definition of each of the five regular convex polyhedra.

Each class consists of two macros: one defines the points, the other calls the first macro and defines the faces. Each macro has a parameter which is a string identifying the particular instance of that class.

The points of a regular tetrahedron are defined in `set_tetrahedron_points`, an example of the general macro name `set_`⟨*class*⟩`_points`. Five points are defined, four of them with `set_obj_point`, a macro which defines points *local* to an object. The first four points are the vertices and the fifth is the center of the tetrahedron. `set_obj_point`'s first parameter is the point number and the other three are the cartesian coordinates. The first three points are in a plane and the fourth is obtained with the `new_face_point` macro, which folds a face (see the description in appendix A for more details). The `new_face_point` macro is used with the angle `an` which is computed in advance. Once the four points are set, the object is normalized, which means that it is centered with respect to the list of vertices given as parameter (here `1,2,3,4`) and the last vertex is put on a sphere of radius 1, centered on the origin. Therefore, point 5 is the center of the tetrahedron, and the tetrahedron is set symmetrically with respect to the origin.

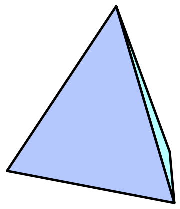
All five convex regular polyhedra are defined in this way and may therefore be inscribed in a sphere of radius 1.

```
def set_tetrahedron_points(expr inst)=
  set_obj_point(1,0,0,0);
  set_obj_point(2,1,0,0);
  set_obj_point(3,cosd(60),sind(60),0);
  sinan=1/sqrt(3);
  cosan=sqrt(1-sinan**2);
  an=180-2*angle((cosan,sinan));
  new_face_point(4,1,2,3,an);
  normalize_obj(inst)(1,2,3,4);
  set_obj_point(5,0,0,0);
enddef;
```

The second macro, `def_tetrahedron` defines the number of points and faces of the instance, calls the previous macro and defines the faces with the macro `set_obj_face`. The first argument of that macro is a *local* face number, the second is a list of vertices such that the list goes clockwise when the face is visible. The last argument is the color of the face in RGB.

```
vardef def_tetrahedron(expr inst)=
  new_obj_points(inst,5);
  new_obj_faces(inst,4);
  set_tetrahedron_points(inst);
  set_obj_face(1,"1,2,4","b4fefe");
  set_obj_face(2,"2,3,4","b49bc0");
  set_obj_face(3,"1,4,3","b4c8fe");
  set_obj_face(4,"1,3,2","b4fe40");
enddef;
```

The result of the drawing is:



A more complex example is the icosahedron which is defined below.

```
def set_icosahedron_points(expr inst)=
  set_obj_point(1,0,0,0);
  set_obj_point(2,1,0,0);
  set_obj_point(3,cosd(60),sind(60),0);
  cosan=1-8/3*cosd(36)*cosd(36);
  sinan=sqrt(1-cosan*cosan);
  an=180-angle((cosan,sinan));
  new_face_point(4,1,2,3,an);
  new_face_point(5,2,3,1,an);
  new_face_point(6,3,1,2,an);
  new_face_point(7,2,4,3,an);
  new_face_point(8,3,5,1,an);
  new_face_point(9,1,6,2,an);
  new_face_point(10,3,4,7,an);
  new_face_point(11,3,7,5,an);
  new_face_point(12,1,8,6,an);
  % 1 and 10 are opposite vertices
  normalize_obj(inst)(1,10);
  % center of icosahedron
  set_obj_point(13,0,0,0);
enddef;
```

```
vardef def_icosahedron(expr inst)=
  save cosan,sinan,an;
  new_obj_points(inst,13);
  new_obj_faces(inst,20);
  set_icosahedron_points(inst);
  set_obj_face(1,"3,2,1","b40000");
  set_obj_face(2,"2,3,4","ff0fa1");
  set_obj_face(3,"3,7,4","b49b49");
  set_obj_face(4,"3,5,7","b49bc0");
  set_obj_face(5,"3,1,5","b4c8fe");
  set_obj_face(6,"1,8,5","b4fefe");
  set_obj_face(7,"1,6,8","b4fe40");
  set_obj_face(8,"1,2,6","45d040");
  set_obj_face(9,"2,9,6","45a114");
  set_obj_face(10,"2,4,9","45a1d4");
  set_obj_face(11,"9,4,10","4569d4");
  set_obj_face(12,"4,7,10","112da1");
  set_obj_face(13,"7,5,11","b4fefe");
  set_obj_face(14,"5,8,11","b49bc0");
  set_obj_face(15,"8,6,12","45a114");
  set_obj_face(16,"6,9,12","b49b49");
  set_obj_face(17,"8,12,11","b40000");
  set_obj_face(18,"7,11,10","45a1d4");
  set_obj_face(19,"12,10,11","b4c8fe");
  set_obj_face(20,"9,10,12","ff0fa1");
enddef;
```

Since all points of the objects are stored in a unique global array, they are internally accessed by the local numbers and an offset defined by the macro `new_obj_points`. The icosahedron example shows a systematic use of the `new_face_point` macro to compute a point on an adjacent face. Displaying such an icosahedron results in the figure 3.

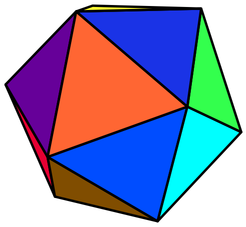
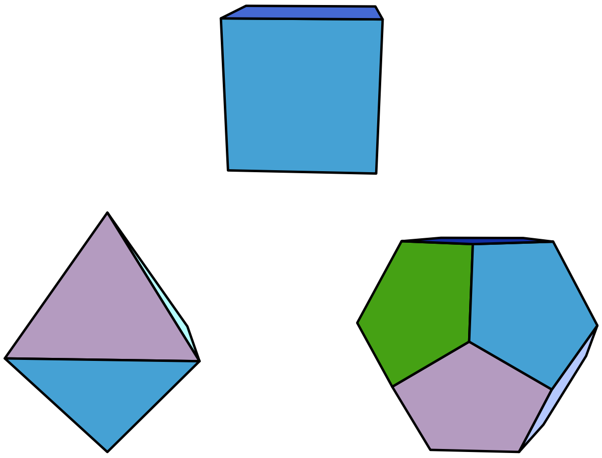


**Figure 3**: An icosahedron

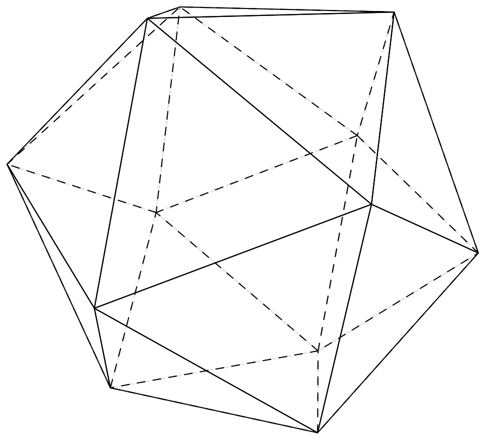
The other three regular convex polyhedra are:



The dodecahedron code is a bit special, since the vertices are built using ten additional points corresponding to face centers. These points are defined as an array of variables `fc1` through `fc10` with `new_points(fc)(10)`. `free_points(fc)(10)` frees them when they are no longer necessary. An excerpt of the dodecahedron code is:

```
def set_dodecahedron_points(expr inst)=
  new_points(fc)(10);% face centers
  set_point(fc1,0,0,0);
  set_obj_point(1,1,0,0);
  set_obj_point(2,cosd(72),sind(72),0);
  rotate_in_plane(3,fc1,1,2);
  ...
  free_points(fc)(10);
enddef;
```

Finally, wire drawings can be obtained by setting the boolean `filled_faces` to false:



**Animating objects** The animation of one or several objects involves the object(s) and an observer. The animation is a set of images and from an image to the next one, the observer as well as the objects can move. For instance the macro `one_image` in `3d.mp` is:

```
def one_image(expr name,i,a,rd,ang)=
  beginfig(i);
    set_point(Obs,
     -rd*cosd(a*ang),-rd*sind(a*ang),1);
    Obs_phi:=90;Obs_dist:=2;
    % fix point 1 of object |name|
    point_of_view_obj(name,1,Obs_phi);
    draw_obj(name);
    rotate_obj_pv(name,1,vect_I,ang);
    % show the rotation point
    draw_point(name,1);
    draw_axes(red,green,blue);
  endfig;
enddef;
```

The parameters of this macro are a name of an object (`name`), an image index (`i`), and three values defining the position of the observer. The observer (`Obs` is a global point and set with `set_point`, not with `set_obj_point`) follows a circle of radius `rd`. The parameter `a`, which is usually a function of `i`, determines the number of rotation steps of the observer, each step being a rotation of angle `ang`. The distance between the observer and the projection plane is 2 (see figure 4).

The orientation of the observer is defined by three angles (see figure 5). The `Obs_phi` angle is given and the two others are computed with a call to `point_of_view_obj(name,1,Obs_phi)` which constrains the observer to look towards point 1 of object `name`. Therefore, this point will seem fixed on the animation and `draw_point(name,1)` draws it later so that this feature can be observed. There is nothing special about that point, except that it remains fix when the object is rotated. The object is drawn with `draw_obj(name)` and `rotate_obj_pv` rotates the object `name` by `ang` degrees around an axis going through point 1 and directed by vector `vect_I` ($\vec{\imath}$). The reference vectors ($\vec{\imath}$, $\vec{\jmath}$ and $\vec{k}$) are drawn in red, green and blue with `draw_axes`.

Finally, a complete animation of an icosahedron is obtained with

```
animate_object("icosahedron",1,100,100);
```

which generates files `anim.101`, ..., `anim.200` from the main file `anim.mp`. The first parameter of `animate_object` is the name of the object to animate, the second and third parameters are minimal and maximal values of the index loop and the fourth parameter is an offset added to the index loop in order to get the file extension, which must lie in the interval 0..4096.

After each image is drawn, the values of the current bounding box are used to compute the bounding box of the sequence of images. The internal values `xmin_`, `ymin_`, `xmax_` and `ymax_` hold the minimal and maximal values of the coordinates of the past images' corners. They are updated just before each image is shipped out.

**Putting the pieces together** Once all the views have been computed, they can be used separately (see for instance the five views of figure 6) or more interestingly, they can be merged. This task is made almost straightforward by METAPOST itself. Indeed, every time `animate_object` is used, a shell script named `create_animation.sh` is generated, as a side-effect of a call to `show_animation_bbox`. The script is similar to that shown in figure 1. This script uses the values computed for the global bounding box of the sequence of images, for these values are necessary in order to extract the right parts of the images and get correct alignments; the parts are extracted with `pnmcut`.[3] If you have the programs used in this script (`Ghostscript`, etc.), you can just run it with `sh create_animation.sh` on UNIX. You may need to adapt it to your needs, and for that purpose, you can modify the macro `write_script` in `3d.mp`.

Some examples are included in the `3d` distribution, and they can be viewed for instance with `netscape` or special programs such as `xanim`.

[3] One might think of using `Ghostscript` for generating an excerpt of an image, but if `Ghostscript` is used to generate the bounding box of an image, it will in general not be possible to have a good alignment between all images. The sizes of the excerpts are only known when all images have been produced.

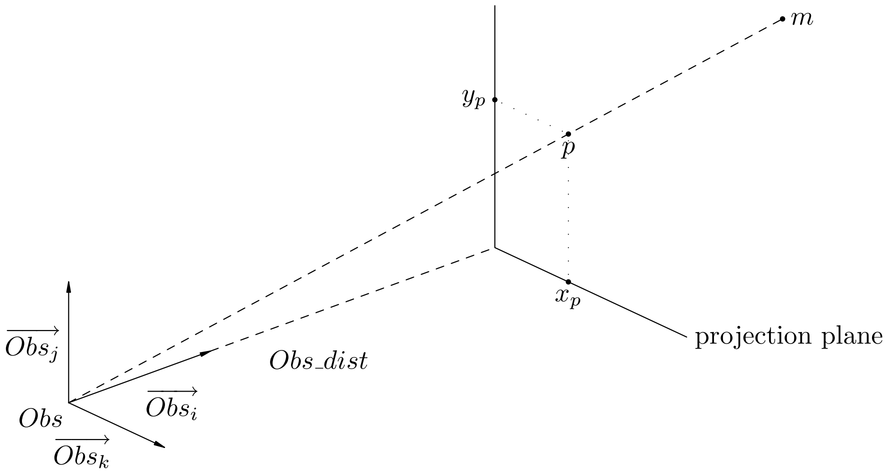



**Figure 4**: Projection on the screen

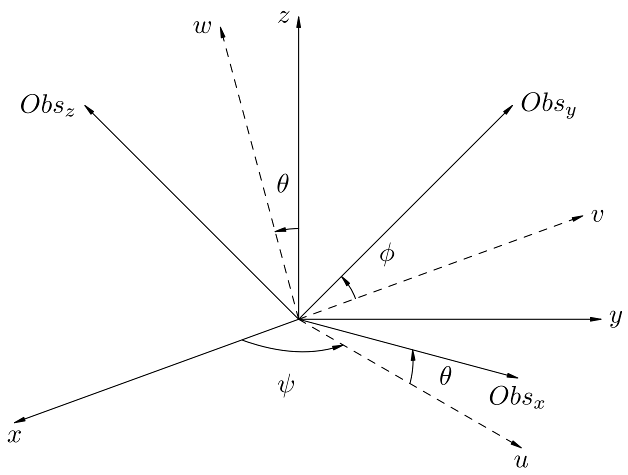



**Figure 5**: Orientation of the observer

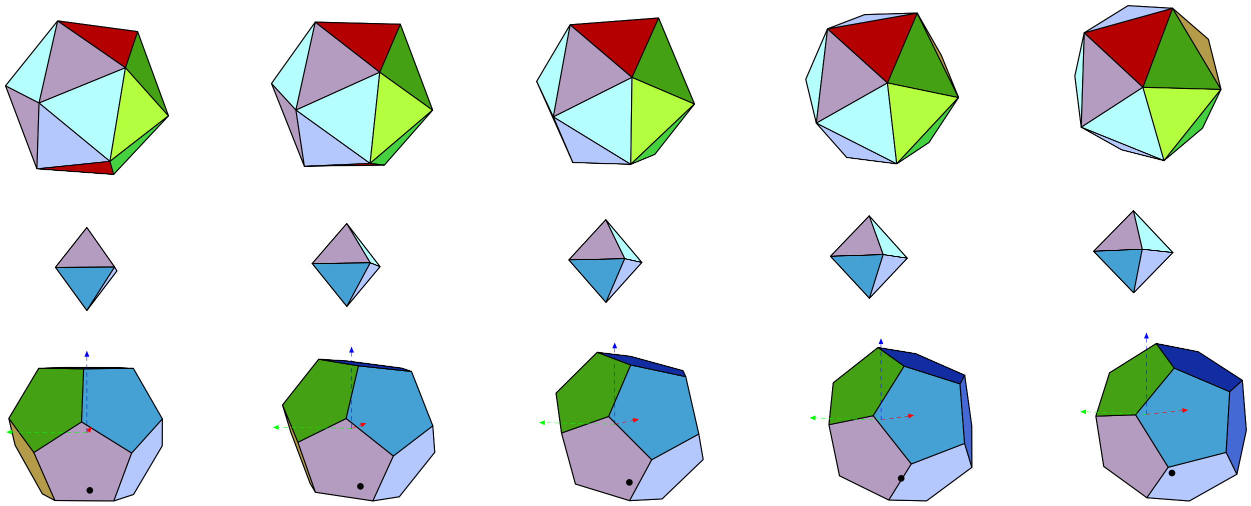


**Figure 6**: Five views of an animation

**Future**

It is quite easy to improve and extend the `3d` macros but the author decided to go no further for the moment. Other objects can be implemented easily and new algorithms can be added. For instance in order to take light sources or shadows into account, one can compute the angles under which a face gets its light, and the angle under which this very face is seen, in order to decide how much darker or lighter it must be rendered. Another problem is to represent overlapping objects correctly. In the current implementation, each object is drawn independently from the other objects, so that the overlapping may be wrong. One solution is to sort all the faces according to their distance to the observer and, if two faces can not be ordered, to split them. Then, the faces can be drawn starting with the most distant, and ending with the closest one. Appendix B explains the internal representation of the objects and shows that this algorithm can be implemented without much surgery to the present code.

**Acknowledgments**

Thanks to John Hobby who always answers all my queries on the METAFONT mailing list. Thanks to Alain Filbois who helped me with the shell script syntax, to Thomas Lambolais and Thomas Genet who gave some feedback on this work, and to Dominique Larchey who pointed out a shortcoming in the conclusion. Thanks to Denis Barbier who was one of the first users of these macros and contributed the animated crayons in the distribution. Thanks to Bogusław Jackowski who made valuable comments on some peculiarities of the code. And finally, special thanks to Ulrik Vieth who not only pushed me to polish my code and this paper more than I had first intended, but also made it possible to use METAPOST under `web2c`.

**References**

Davidsen, Bill. “NETPBM”. 1993. Available for instance at `ftp://ftp.wustl.edu/graphics/graphics/packages/NetPBM`.

Goossens, Michel, S. Rahtz, and F. Mittelbach. *The LaTeX Graphics Companion*. Addison-Wesley, Reading, MA, USA, 1997.

Hobby, John D. “A User’s Manual for MetaPost”. Technical Report 162, AT&T Bell Laboratories, Murray Hill, New Jersey, 1992.

Knuth, Donald E. “MFT, version 2.0”. 1989. Standard TeX distribution.

Knuth, Donald E. *The Art of Computer Programming, volumes 1, 2 and 3*. Addison-Wesley Publishing Company, 1997. New editions.

Müller, René K. “GIFMerge, version 1.33”. 1996. Available at `http://www.iis.ee.ethz.ch/~kiwi/GIFMerge/`.

**Appendix A**
**Summary of the 3d package**

**Types** The commands in the `3d` package take parameters of several different types. The types are described here.

- An ⟨*avn*⟩ (*Absolute Vector Number*) is the internal number identifying a vector in the `vect` array (an integer).
- An ⟨*apn*⟩ (*Absolute Point Number*) refers to a vector in the same way as an ⟨*avn*⟩ (an integer).
- A ⟨*lpn*⟩ (*Local Point Number*) is a number identifying a point *within* an object (an integer). Two ⟨*lpn*⟩s with the same value can correspond to different points in different objects.
- An ⟨*afn*⟩ (*Absolute Face Number*) is the internal number identifying a face.
- A ⟨*lfn*⟩ (*Local Face Number*) is a number identifying a face *within* an object (an integer). As for points, two ⟨*lfn*⟩s with the same value can correspond to different faces in different objects.
- A ⟨*cl*⟩ (*Class*) is a string representing a class, for instance `"tetrahedron"`. It may only contain letters and underscores.
- An ⟨*obj*⟩ (*Object*) is a string representing an object, that is an instance of a class. Such a string may only contain letters and underscores.
- A ⟨*vl*⟩ (*Vertex List*) is a list of integers, where each integer identifies a vertex. For instance, `1,7` is the list of vertices 1 and 7.
- A ⟨*vsl*⟩ (*Vertex String List*) is a string corresponding to a list of integers, where each integer identifies a vertex. For instance, `"1,2,6,5"` is the list of vertices 1, 2, 6 and 5.
- ⟨*hc*⟩ (*Hex Color*) is a string representing a color with the three RGB components in hexadecimal and in the range 0..255. For instance, `"b4fe40"`.
- ⟨*col*⟩ (*Color*) is a standard METAPOST color (a triplet of RGB components in the range 0..1), such as `red`.
- ⟨*str*⟩ (*String*) is a string.
- ⟨*pair*⟩ (*Pair*) is a pair of numerics.
- ⟨*num*⟩ (*Numeric*) is a number.
- ⟨*bool*⟩ (*Boolean*) is a boolean.

**Low level vector commands** The low level vector commands define the classical operations in vector algebra.

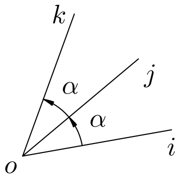
- vect_def($\langle avn\rangle$,$x$,$y$,$z$): defines vector $\langle avn\rangle$ as $(x, y, z)$;
- set_point; synonym of vect_def: a point is stored in the same array as vectors.
- set_obj_point($\langle lpn\rangle$,$x$,$y$,$z$): this defines the point $\langle lpn\rangle$ as $(x, y, z)$;
- vect_def_vect($\langle avn\rangle_1$,$\langle avn\rangle_2$): vector $\langle avn\rangle_1$ becomes equal to vector $\langle avn\rangle_2$;
- vect_sum($\langle avn\rangle_1$,$\langle avn\rangle_2$,$\langle avn\rangle_3$): the vector $\langle avn\rangle_1$ becomes the sum of vectors $\langle avn\rangle_2$ and $\langle avn\rangle_3$.
- vect_translate($\langle avn\rangle_1$,$\langle avn\rangle_2$): add vector $\langle avn\rangle_2$ to vector $\langle avn\rangle_1$; vector $\langle avn\rangle_2$ remains unchanged.
- vect_diff($\langle avn\rangle_1$,$\langle avn\rangle_2$,$\langle avn\rangle_3$): the vector $\langle avn\rangle_1$ becomes the difference between vectors $\langle avn\rangle_2$ and $\langle avn\rangle_3$.
- vect_dprod($\langle avn\rangle_1$,$\langle avn\rangle_2$) $\rightarrow \langle num\rangle$: returns the dot product of vectors $\langle avn\rangle_1$ and $\langle avn\rangle_2$.
- vect_mod($\langle avn\rangle$) $\rightarrow \langle num\rangle$: returns the modulus of vector $\langle avn\rangle$.
- vect_prod($\langle avn\rangle_1$,$\langle avn\rangle_2$,$\langle avn\rangle_3$): the vector $\langle avn\rangle_1$ becomes the vector product of vectors $\langle avn\rangle_2$ and $\langle avn\rangle_3$.
- vect_mult($\langle avn\rangle_1$,$\langle avn\rangle_2$,$\langle num\rangle$): $\langle avn\rangle_1$ becomes vector $\langle avn\rangle_2$ scaled by $\langle num\rangle$.
- mid_point($\langle avn\rangle_1$,$\langle avn\rangle_2$,$\langle avn\rangle_3$): vector (or point) $\langle avn\rangle_1$ becomes the mid-point of vectors (or of the line joining the points) $\langle avn\rangle_2$ and $\langle avn\rangle_3$.
- vect_rotate($\langle avn\rangle_1$,$\langle avn\rangle_2$,$a$): vector $\langle avn\rangle_1$ is rotated around vector $\langle avn\rangle_2$ by the angle $a$.

**Operations on objects** Several operations apply globally on objects:

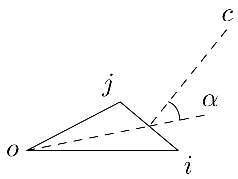
- assign_obj($\langle obj\rangle$,$\langle cl\rangle$): create $\langle obj\rangle$ as an instance of class $\langle cl\rangle$.
- reset_obj($\langle obj\rangle$): put $\langle obj\rangle$ back where it was just after it was initialized.
- put_obj($\langle obj\rangle$,$\langle avn\rangle$,$s$,$\psi$,$\theta$,$\phi$): object $\langle obj\rangle$ is scaled by $s$, shifted by vector $\langle avn\rangle$ and oriented with the angles $\psi$, $\theta$, $\phi$, as for the observer orientation (figure 5).
- rotate_obj_pv($\langle obj\rangle$,$\langle lpn\rangle$,$\langle avn\rangle$,$a$): object $\langle obj\rangle$ is rotated around an axis going through local point $\langle lpn\rangle$ and directed by vector $\langle avn\rangle$; the rotation is by $a$ degrees.
- rotate_obj_abs_pv($\langle obj\rangle$,$\langle apn\rangle$,$\langle avn\rangle$,$a$): the object $\langle obj\rangle$ is rotated around an axis going through absolute point $\langle apn\rangle$ and directed by vector $\langle avn\rangle$; the rotation is by $a$ degrees.
- rotate_obj_pp($\langle obj\rangle$,$\langle lpn\rangle_1$,$\langle lpn\rangle_2$,$a$): $\langle obj\rangle$ is rotated around an axis going through local points $\langle lpn\rangle_1$ and $\langle lpn\rangle_2$; the rotation is by $a$ degrees.
- translate_obj($\langle obj\rangle$,$\langle avn\rangle$): object $\langle obj\rangle$ is translated by vector $\langle avn\rangle$.
- scale_obj($\langle obj\rangle$,$v$): object $\langle obj\rangle$ is scaled by $v$.

**Building new points in space** Three macros are especially useful for the definition of regular polyhedra:

- rotate_in_plane($k$,$o$,$i$,$j$): get point $k$ from point $j$ by rotation around point $o$ by an angle $\alpha$ equal to the angle from $i$ to $j$; $i$, $j$ and $k$ are of type $\langle lpn\rangle$, whereas $o$ is of type $\langle apn\rangle$.




- new_face_point($c$,$o$,$i$,$j$,$\alpha$): the middle $m$ of points $i$ and $j$ is such that $(\widehat{\overrightarrow{om}, \overrightarrow{mc}}) = \alpha$ and $\overrightarrow{mc}$ is $\overrightarrow{om}$ rotated around $\overrightarrow{ji}$. $c$, $o$, $i$ and $j$ are of type $\langle lpn\rangle$.




- new_abs_face_point($c$,$o$,$i$,$j$,$\alpha$): similar to the previous definition, but $c$ and $o$ are of type $\langle apn\rangle$.

**Drawing points, axes, objects**

- draw_point($\langle obj\rangle$,$\langle lpn\rangle$): draw point $\langle lpn\rangle$ in object $\langle obj\rangle$.
- draw_axes($\langle col\rangle_1$,$\langle col\rangle_2$,$\langle col\rangle_3$): draw vectors $\vec{\imath}$, $\vec{\jmath}$ and $\vec{k}$ in colors $\langle col\rangle_1$, $\langle col\rangle_2$ and $\langle col\rangle_3$.
- draw_obj($\langle obj\rangle$): draw object $\langle obj\rangle$.

**Setting faces**

- set_face($\langle afn\rangle$,$\langle vsl\rangle$,$\langle hc\rangle$): set absolute face $\langle afn\rangle$ as delimited by the vertex list $\langle vsl\rangle$ (local point numbers) and colored by color $\langle hc\rangle$.
- set_obj_face($\langle lfn\rangle$,$\langle vsl\rangle$,$\langle hc\rangle$): set local face $\langle lfn\rangle$ as delimited by the vertex list $\langle vsl\rangle$ (local point numbers) and colored by color $\langle hc\rangle$.

**View points, distance**

- `compute_reference(`$\psi$`,`$\theta$`,`$\phi$`)`: defines the orientation of the observer by the three angles $\psi$, $\theta$ and $\phi$. See figure 5.
- `point_of_view_obj(`⟨*obj*⟩`,`⟨*lpn*⟩`,`$\phi$`)`: the orientation of the observer is defined as looking local point ⟨*lpn*⟩ of object ⟨*obj*⟩, with an angle of $\phi$;
- `point_of_view_abs(`⟨*apn*⟩`,`$\phi$`)`: the observer's orientation is defined as looking absolute point ⟨*apn*⟩, with an angle of $\phi$;
- `obs_distance(`$v$`)(`⟨*obj*⟩`,`⟨*lpn*⟩`)`: let $v$ equal the distance between the observer and local point ⟨*lpn*⟩ in object ⟨*obj*⟩.

**Vector and point allocation**

- `new_vect`→ ⟨*avn*⟩: return a new vector;
- `new_point`: synonym of `new_vect`;
- `new_points(`$v$`)(`$n$`)`: defines the absolute points $v_1, \ldots, v_n$, using `new_point`;
- `free_vect(`⟨*avn*⟩`)`: free vector ⟨*avn*⟩;
- `free_point(`⟨*apn*⟩`)`: free point ⟨*apn*⟩;
- `free_points(`$v$`)(`$n$`)`: frees the absolute points $v_1, \ldots, v_n$, using `free_point`.

**Debugging**

- `show_vect(`⟨*str*⟩`,`⟨*avn*⟩`)`: shows vector ⟨*avn*⟩, with string ⟨*str*⟩.
- `show_point`: synonym of `show_vect`
- `show_pair(`⟨*str*⟩`,`⟨*pair*⟩`)`: this shows a numeric pair, with string ⟨*str*⟩.

**Normalization**

- `normalize_obj(`⟨*obj*⟩`,`⟨*vl*⟩`)`: normalize object ⟨*obj*⟩ with respect to the list of vertices ⟨*vl*⟩.

**Parameters**

- `Obs_dist` → ⟨*num*⟩: distance between the observer and the projection plane.
- `h_field` → ⟨*num*⟩: horizontal field of view (default: 100 degrees)
- `v_field` → ⟨*num*⟩: vertical field of view (default: 70 degrees)
- `Obs_phi` → ⟨*num*⟩: angle $\phi$ for the orientation of the observer;
- `Obs_theta` → ⟨*num*⟩: angle $\theta$ for the orientation of the observer;
- `Obs_psi` → ⟨*num*⟩: angle $\psi$ for the orientation of the observer;
- `drawing_scale` → ⟨*num*⟩: scale factor applied for drawing;
- `filled_faces` → ⟨*bool*⟩: if `true`, the faces are drawn filled; if `false`, only the edges are drawn, and hidden edges are drawn dashed;
- `draw_contours` → ⟨*bool*⟩: if `true`, the contours of the faces are drawn, and the lines have the thickness `contour_width`; if `false`, the contours are not drawn;
- `contour_width` → ⟨*num*⟩: dimension used for drawing contours of faces (default: 1pt).

**Constants** These values represent constant objects such as reference vectors, and should not be changed.

- `vect_null` → ⟨*avn*⟩: internal index for $\vec{0}$.
- `vect_I` → ⟨*avn*⟩: internal index for $\vec{\imath}$.
- `vect_J` → ⟨*avn*⟩: internal index for $\vec{\jmath}$.
- `vect_K` → ⟨*avn*⟩: internal index for $\vec{k}$.
- `point_null` → ⟨*apn*⟩: internal index for $\vec{0}$.
- `Obs` → ⟨*apn*⟩: observer's internal point number.

**Defining new object points and faces**

- `new_obj_points(`⟨*obj*⟩`,`⟨*num*⟩`)`: defines points 1 to ⟨*num*⟩ in object ⟨*obj*⟩; must be used before setting the points;
- `new_obj_faces(`⟨*obj*⟩`,`⟨*num*⟩`)`: defines ⟨*num*⟩ faces in object ⟨*obj*⟩; must be used before setting the faces;

**Offsets**

- `pnt(`⟨*lpn*⟩`)` → ⟨*apn*⟩: returns the absolute point number for a given local point index.
- `face(`⟨*lfn*⟩`)` → ⟨*afn*⟩: returns the absolute face number for a given local face index.

**Standard classes** Five standard classes are defined in `poly.mp`: they define the five regular convex polyhedra. For each class ⟨*class*⟩, there are two macros:

- `set_`⟨*class*⟩`_points` (e.g. `set_cube_points`)
- `def_`⟨*class*⟩ (e.g. `def_cube`)

Each of these macros is defined with a parameter which is the instance name.

**Standard animations** The `3d` package provides a few standard animations using the convex polyhedra. In each of these animations, the observer follows a circular path pictured in figure 7. Each standard animation is divided into two macros. The first, such as `animate_object`, defines the class(es) that are used and sets the objects. The second, such as `one_image`, sets the observer, draws the object(s) and moves the object(s) and the observer. The file `animpoly.mp` gives examples of the use of the standard animations.

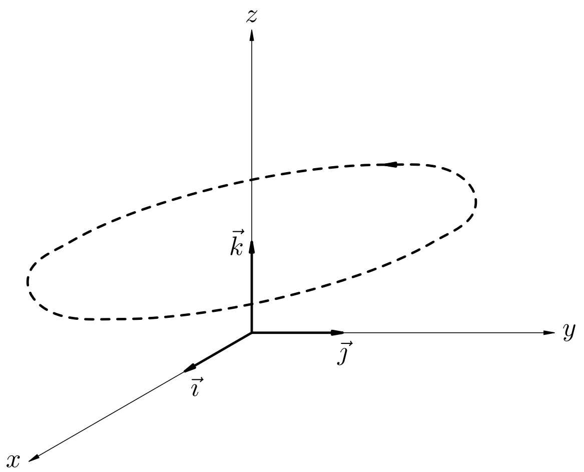



**Figure 7**: Motion of the observer

## Appendix B Coding an object

In order to extend the `3d` package, it is necessary to understand how the objects are coded. We give here an overview of this coding, but the reader is advised to peek in the code to get a better understanding on how all the functions interact.

First, an object has a name, for instance `"box"`. The macro `box_class` (which can be called with `obj_class_("box")`) is the string corresponding to the class of `"box"`, for instance `"cube"`. The variable `cube_point_offsetbox`, of type `numeric`, and obtained with `obj_point_offset_("box")`, is equal to the absolute index of the last point of the previous object. A cube is defined with $8 + 1$ points. Assuming it was defined after an icosahedron ($12+1$ points) named `"ico"`, `cube_point_offsetbox` will be a `numeric` equal to 13. `cube_pointsbox` (obtained with `obj_points_("box")`) is a macro equal to 9. The variable `cube_face_offsetbox`, similar to `cube_point_offsetbox`, obtained with a call to `obj_face_offset_("box")`, equals 20. The macro `cube_facesbox` (obtained by `obj_faces_("box")`) is equal to 6.

The `obj_name` macro is extended each time a new object is defined. To an absolute face number, it associates an object name. Hence, it is possible to go through all faces. `last_point_offset_` and `last_face_offset_` are the absolute numbers of the last points and faces defined up to now.

```
def obj_name(expr i)=
  if i<1: elseif i<=20:"ico"
          elseif i<=26:"box"
  fi;
enddef;
```

`pnt(i)` gives the absolute vector corresponding to local point $i$. `ipnt_(i)` is the absolute point number, that is $i$ plus the number of points defined beforehand in other objects. `points_[j]` is the absolute vector corresponding to absolute object point $j$. Similarly, `face(i)` is the absolute face corresponding to local face $i$.

The list of vertices of absolute face number $i$ is `face_points_[i]`. The color of absolute face number $i$ is `face_color_[i]`.

When the macros `pnt` or `face` are to be used, the calls `define_current_point_offset_("box")` and `define_current_face_offset_("box")` must be issued.

◇ Denis Roegel
CRIN (Centre de Recherche en Informatique de Nancy)
Bâtiment LORIA
BP 239
54506 Vandœuvre-lès-Nancy
FRANCE
roegel@loria.fr
URL: http://www.loria.fr/~roegel