M3D PACKAGE MANUAL

Anthony Phan, October 5, 2011

Introduction

In the second half of the nineties (of the past century), I discovered TEX, MetaFont and, then, MetaPost. I was truly enthousiastic about this last tool since it allows a very simple inclusion of images produced by some MetaFont-like language into TEX documents. My first attempt in 3D pictures with MetaPost was very simple: the projection system was absolutely rigid and the picture was composed of a few lines and labels and only 4 filled flat surfaces whose colors where static. Then I understood that it is necessary to have a parametrizable frame since one does not always know if a picture, viewed from a certain angle, will be meaningfull or not. So the first and main step was done: having a parametrizable frame, dealing with space coordinates, drawing contours if their projection on the screen is well-oriented.

Then I heard of Denis Roegel's package "3D". Its syntax didn't fit what I could have in mind about 3D-programming in MetaPost, but it features facilities for `eps` to `gif` conversion and animation. I stolled these animation facilities in order to have fun and also to find the best angle for my picture. Then I began to make more complex designs intended to illustrate my web pages...

What was my *leitmotiv* was to keep the closest as possible to usual MetaFont/MetaPost programming, but also to consider every 3-dimensional object as enough complex to deserve a quite technical code. So I've come to think that what could be nice should be to have a stable and powerfull basic program and add to it libraries of common objects. Some more complex objects could then be build up from those more basic ones by moving them, rotating them and rescaling them. My idea about objects is certainly not related to object-programming, it is just the naive notion of solid bodies.

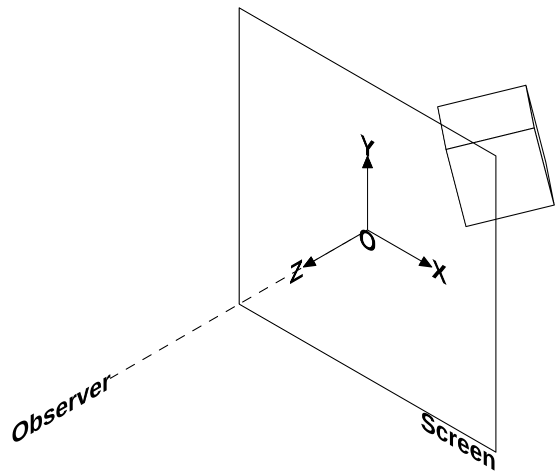
The aim one can have with such a project is boundless. The package "m3D" remains still in progress. I'm only glad to be able to use it when a friend ask me if I can draw a better picture than he does.

Also one of my believes is that any MetaPost programmer prefers to build his/her own macros' system than relying on someone else's programs—especially when these ones are claimed to be unstable. Such a programmer would simply have an overview of the syntax, of some special hacks, may publish his/her own programs on Internet and in this way give some feed back to every other people interested in such programming.

§ 1. Basic concepts

As anyone knows, MetaPost provides standard facilities for manipulating 3 dimensional vectors with the variable type "color": type-check, addition, substraction, scalar multiplication. What may be seen as missing is affine transformations of 3 dimensional vectors or pictures but no one would complain about that since MetaPost is a 2-dimensional oriented programming language and anyone would accept using some higher level control sequences for such tasks.

§ 2. Coordinates system



§ 3. Euler Angles

Given to orthonormal frames $(\mathbf{Ox}, \mathbf{Oy}, \mathbf{Oz})$ and $(\mathbf{Ox}', \mathbf{Oy}', \mathbf{Oz}')$, Euler angles (θ, ϕ, ψ) are real numbers such that

$$\begin{cases} \mathbf{Ox}' = \cos\phi\cos\theta \times \mathbf{Ox} + \cos\phi\sin\theta \times \mathbf{Oy} + \sin\phi \times \mathbf{Oz} \\ \mathbf{Oy}' = -(\cos\psi\sin\theta + \sin\psi\sin\phi\cos\theta) \times \mathbf{Ox} \\ \qquad\quad + (\cos\psi\cos\theta - \sin\psi\sin\phi\sin\theta) \times \mathbf{Oy} + \sin\psi\cos\phi \times \mathbf{Oz} \\ \mathbf{Oy}' = (\sin\psi\sin\theta - \cos\psi\sin\phi\cos\theta) \times \mathbf{Ox} \\ \qquad\quad - (\sin\psi\cos\theta + \cos\psi\sin\phi\sin\theta) \times \mathbf{Oy} + \cos\psi\cos\phi \times \mathbf{Oz} \end{cases}$$

They can be determined by the knowledge of the direction of $\mathbf{Ox}'$ expressed in terms of $(\mathbf{Ox}, \mathbf{Oy}, \mathbf{Oz})$-coordinates—one gets θ and ϕ—, and of the plane generated by $(\mathbf{Ox}', \mathbf{Oy}')$ which can be expressed by the $(\mathbf{Ox}, \mathbf{Oy}, \mathbf{Oz})$-coordinates of a vector lying the $(\mathbf{Ox}', \mathbf{Oy}')$-plane which is not colinear to $\mathbf{Ox}'$—one gets finally ψ.

So, a control sequence named `Angles` is defined in `m3Dplain.mp`. Its parameters are two vectors (or colors), say p and q, and it returns the triple (or color), say (θ, ϕ, ψ) of the corresponding Euler angles in the following way: if p is the null or a too small vector, it returns $(0, 0, 0)$; else, θ and ϕ are computed in such way that $\mathbf{Ox}'$ would have the same direction as p; then again, if q appears to be quite colinear to p, ψ is set to 0, or else ψ is set to the correct value that helps to define the whole $(\mathbf{Ox}', \mathbf{Oy}', \mathbf{Oz}')$-orthonormal frame such that the (p, q)-vector plane is equal to the $(\mathbf{Ox}', \mathbf{Oy}')$-vector plane with $\mathbf{Ox}'$ sharing the same direction as p.

§ 4. Projection system

§ 5. Rendering parameters

There are many parameters used by m3D, they are described below. We put a dagger mark $(^\dagger)$ in the descriptions of the parameters which should not be changed—mostly for æsthetic reasons—within a figure.

`ObsZ :=` internal numeric†. It is the distance between the screen or the sheet of paper and the observer. It should be given with metrical units.

`Resolution :=` internal numeric†. Some predefined objects use this paramer to determin the number of steps for their drawing. It should be given with metrical units.

`LightSource :=` color†. It is the location of the light source.

`LightColor :=` color†. It is the color of the light source (used with *specularity* only).

`LightAtInfinity :=` boolean†. It indicates whether the light source is a true point in space or a direction.

`Luminosity :=` internal numeric†. It is the intensity of the incident light. It should lie between 0 and 1.

`Contrast :=` internal numeric†. It is the usual contrast parameter. It should lie between 0 and 1.

`Specularity :=` internal numeric. It quantifies what fraction of incident light is reflected by objects. It should lie between 0 and 1 and can be changed within a figure in order to render different kinds of matter.

`Phong :=` internal numeric. It quantifies the spread of reflected light on objects. It should be a small integer and can be changed within a figure in order to render different kinds of matter.

`Fog :=` internal numeric†. A "fog" can be applied when rendering objects. The value 0 means no fog, 1 a linear fog with exponential decay, 2 a spherical fog with exponential decay. Colors fade to the `background` color.

`FogHalf :=` internal numeric†. It is the rate of the exponential decay of the fog. It should be a positive number given with metrical units.

`FogZ :=` internal numeric†. It is the z-coordinate relative to the screen below which fog can be applied. It should be given with metrical units.

`FinePlotFlag :=` boolean. This boolean is used by some control sequence like `plot3D`.

`mthreeDfont :=` string†. The name of the (vector) font that should be used for text in the 3 dimensional space.

`ShadedTextFlag :=` boolean. This boolean indicates whether special effects are applied when rendering a text in the 3 dimensional space or not.

`ObjectColor :=` color. This the color which is used when filling a facet in space. It can be changed at any time in order to get figures with plenty of colors.

§ 6. About the light source

There is only one light source defined in `m3Dplain.mp`. It can be located at infinity (`LightAtInfinity := true`) or at some point in the scenery (`LightAtInfinity := false`). In both cases its coordinates `LightSource` refer to the global screen frame and are not modified when objects are translated or rotated, that is that the light is fixed for the observer.

If one wants to link the light source to a peculiar object, one has to write within the definition of the object something like

```
LightSource := GDir(x,y,z)
```

if the light source is located at infinity, or

```
LightSource := GCoord(x,y,z)
```

if the light source is located at some point in space (`(x,y,z)` are here *local* coordinates). One should notice that it is barely the only direct use of the control sequences `GDir` and `GCoord`.

If one needs more than one light source, one has to define anew the control sequence `Light` of `m3Dplain.mp` (good luck since it is already a heavy machinery).

§ 7. Looking at objects from outside, inside or both

Usually the observer looks at solid objects from their outsides. Thus, their facets are drawn if the projection of their oriented contours appears to be positively oriented. This is why the default meaning of the control sequence `Orientation` is `Outside`. One can reverse the situation by telling

```
Inside;
```

and later turn back to the default situation with

```
Outside;
```

One can also try the following

`OutsideIn;` or `InsideOut;`

In these cases, facets are always drawn. The difference between the two is that light effects are reverted in the last case. If one wishes to achieve even weirder effects, he/she can play with the definition of `Orientation` and the value of `Outside_` with the basis of what is written in `m3Dplain.mp`. For instance,

```
def OutsideOut =
  vardef Orientation tertiary c =
    Outside_ := if turningnumber c < 0: - fi 1;  1
  enddef;
  Outside_:=1
enddef;
```

which is defined in `m3Dplain.mp` solves some difficulties with the Klein bottle.

§ 8. Ordering and hidden bodies

Drawing a single convex body is an easy task: one just have to draw every facet which contour is seen, or projected, as a positively (resp. negatively) oriented path when drawing its exterior (resp. interior) side. So, hidden facets are no problem in these cases. Switching between inside and outside can be done with `Inside` and `Outside` control sequences. They simply reverse the condition about orientation.

When dealing with non-convex bodies, one has to decompose these bodies into their convex parts and draw them in a proper order. Thus, we come to ordering. A fairly tricky control sequence named `QuickSort` has been designed for this purpose. Its argument is some text that must contain at least two terms and its output is a control sequence named `SortedList` which content is the previous (expanded) list ordered with respect to a `SortCriterion`. `SortCriterion` is a control sequence with two arguments (members of the list which is to be sorted) whose replacement text is a boolean. By default, these arguments are triples and the condition is about their actual depth relatively to the current observer. Thus `QuickSort(`*list of triples*`)` would output `SortedList` whose replacement text is just the expected ordered list of triples. One can change this just by adapting `SortCriterion` in order to sort numerics, pairs, strings, ...

It wouldn't be very elegant to sort things this way if one wants to perform a list of actions with respect to the depth of a list of points in space. A more natural way to do so is to use the following procedure:

`OnDepth;`
`Refpoint` *triple*`;`
`Action (`*delimited control sequences*`);`

```
...
endOnDepth;
```

What this procedure does is the following: save and reset a few things at the `OnDepth` statement; increase the `Action_counter` and stores the current reference point (a *triple*) at `Refpoint` invocation; stores the *delimited control sequences* into a variable control sequence numbered (here there is a little trick that I have been looking for a very very long time) with the current `Action_counter`; at `endOnDepth` orders the list $1, \ldots,$ `Action_counter` with respect to the depth of the reference points and then performs actions with respect to the sorted list. The most interesting thing with this procedure is that actions may depend on some parameters just as loop or macro parameters. Note that the `SortCriterion` has a special and temporary meaning when performing `endOnDepth`: its two arguments are then some indexes in $1, \ldots,$ `Action_counter` and the comparison is done between the depths of the two corresponding reference points.

§ 9. Integrating text

This is clear that for some nice and funny pictures, one has to integrate text in the three dimensional space, for instance when moving a text around a picture as I've done once or twice. This is rather particular. Defining any general scheme for doing so seems to me rather pointless: it is too complicated, it is hard to imagine what people would like to do, etc. Anyway, some control sequence that allows to move flat text around would be a basic stuff. Also, basic programming of such things may help to design special control sequences for more complicated tasks.

9.1. *Simple text.* — An object named `simpletext` is defined in `m3Dplain.mp`. Its specific parameters consist, first, in a string describing the alignment (`"left"`, `"justify"`, `"center"` or `"right"`) and a string telling where on the text the reference point should be (typically `"right"`, `"urt"`, `"top"`, `"ulft"`, `"left"`, `"llft"`, `"bot"`, `"lrt"` or [say!] `"center"`), then, in a list of strings which would be displayed one above another.

For instance, at a top most level (*see* further on about the scale parameter),

```
UseObject(simpletext, Origin, (90, 0, 90), 10pt, "justify", "ulft",
   "Come let me sing into your ear;",
   "Those dancing days are gone,",
   "All that silk and satin gear;",
   "Crouch upon a stone,",
   "Wrapping that foul body up",
   "In as foul a rag:",
   "I carry the sun in a golden cup;",
   "The moon in a silver bag."));
```

would display, at basepoint `Origin`, with frame the current one rotated by $(90, 0, 90)$, with scale 10 pt, the first part of J.B. Yeats' poem "Those dancing days are gone". If it is reasonable, each line would be justified and the basepoint would correspond to the upper left corner of the text.

The `simpletext` object uses a font named by the string `mthreeDfont` (default value: `"rphvb"`). Its design size is defined by the numerical variable named `mtheeDfontsize` (default value: `10pt`). The numerical parameter `baselineskip` parameter has its usual role (default value: `12pt`). These parameters are only related to the font, not to the `CurrentScale`. Thus, when using `simpletext` into an object, beware of the *scale* given in statements like

`UseObject(simpletext,` *origin*`,` *Euler angles*`,` *scale*`,`
alignment string`,` *location string*`,` *list of strings*`)`

since *scale* must be a *local scale*.

Justification is obtained by stretching the normal space width of the font up to a `TextStretchFactor` (whose default value is 2 which is quite large). Many successive spaces into a sentence count as as many spaces: *they are not reduced into a single space.*

Each character is drawn separately just to make sure that the affine transformation acting on the character would be approximatively correct if the projection is not linear. *Since every character would have a special scaling, one must set* `prologues` *to 1 or 2 in order to not overflow MetaPost capacities (the whole font will simply be declared at the beginning of the eps figure and not every character with its own transformation).* Of course, one should use PostScript^{TM} fonts for such use. This is why we have introduced `mthreeDfont`.

9.2. *Curved text.* — Planned but not released.

Remark. — Remember that annotating graphics can still be done with usual control sequence like

`label.`*loc*`(`*label*`, proj(`x`, `y`, `z`))`

since things like `simpletext` or `curvedtext` are rather for very special effects.

§ 10. Animations

10.1. *Introduction.* — Denis Roegel demonstrated that it is possible to use usual Unix tools to merge a list of MetaPost outputs into an animated GIF image (3D package). I've learned a lot from his "metapost to shell" script. The idea is the following: first, keep track of the maximum boundary limits of every eps outputs; then convert these eps outputs into PostScript^{TM} or eps files with these maximum boundary limits; convert these last files to, say, simple GIF images; then merge all these images into an animation.

10.2. *How-to with m3D.* — With `m3Dplain` (as for `3D`), (hidden) numerics named `xmin_`, `xmax_`, `ymin_` and `ymax_` are updated at every figure ends through a control sequence named `compute_bbox` (embedded by `m3Dplain.mp` into `extra_endfig`). Executing

`Animate(`*numeric*`, `*boolean or color*`)`

at the end of the file would produce an external file whose name is, by default, `animate-script`. Once the MetaPost job finished, under Unix-like system, execute

```
bash animate-script
```

from a console (`xterm` or such) in the right directory. The final output is `jobname.gif` where `jobname` is the actual name of the MetaPost program.

10.3. *External programs.* — This script requires the following programs: `sed` and `convert`. We choose to use `sed` to change the boundary parameters of every MetaPost output—the resulting temporary files are named `jobname.xxx.eps` where `jobname.xxx` is the name of one of the MetaPost outputs. The PostScript^{TM} to GIF conversion was performed with the Netpbm library in Roegel's macros and their merging into an animation was performed by `gifmerge` (a non standard but very nice Unix program freely available on the web). These last years, ImageMagick (copyrighted first by Dupont de Nemours, then by ImageMagick Studio, but quite free in fact) has spread over almost every Linux distribution. It is a very high quality tool that converts anything into everything, even animations. One of the basic control sequence is `convert` which is the one called by the former script.

10.4. *Details.* — The following provides more detailed explanations.

Animate(*numeric*, *boolean or color*) Control sequence whose first parameter is the border in bp provided with no units, and the second one is a color or a boolean. When the second parameter is a color, this color will be made transparent in the animation. When it is a boolean, background (color) will be made transparent if the boolean is true, and no transparency will be made if this boolean is false.

AnimateScript String variable, name of the (bash) script file which is output by Animate.

AnimateFormat String variable, format of the animated image to be constructed. Its default value is "gif", but one can change it to, say, "mpg" or "mng". The format must be understood by the convert command, and also some other programs should be available (mpg2encode for "mpg" format).

AnimateQuality Numerical variable, typical values are 1, 2, 4...

AnimateDelay Numerical variable, time in 1/100 seconds between every image in the animation.

AnimateLoop Numerical variable, parameter for the animation, its default value is equal to 0 (infinite loop).

compute_bbox and also xmin_, xmax_, ymin_, ymax_ have been explained before.

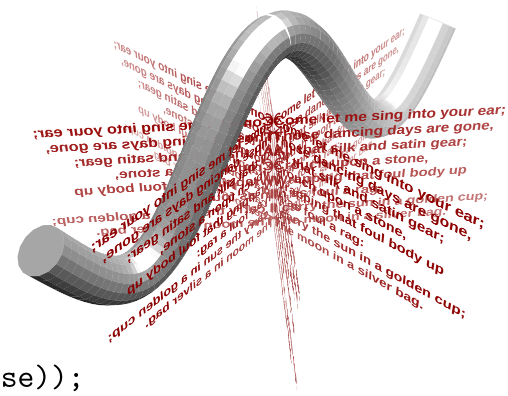
§ 11. Outputting encapsuled PostScript directly

```
DirectEPS filename;
...
endDirectEPS;
```

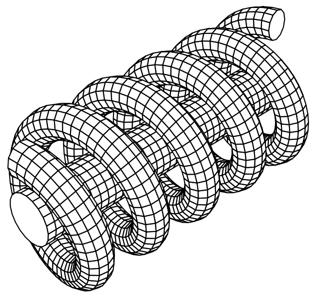
§ 12. Some samples

The first picture is made with Fill equal to TechnoFill (I have to change this name one day), the next one with Fill equal to WireFill (more conventional). I have also added a text (W.B. Yeats) for testing the simpletext object. There is, in the second picture, a cylinder but the most important is the use of an object named tube: given a path in space through $x(t)$, $y(t)$, $z(t)$, a radius r, a range for t, this object is what one can expect it to be. Also computations are quite fragile (second order) and may leads to unexpected and ugly effects. If the object cylinder is defined in m3Dlib01.mp, tube is defined in m3Dplain.mp since I think it may be a basic tool.

```
let Fill = SolidFill;%TechnoFill;
ShadedTextFlag := true;
TextColor := red;
beginfig(thisfig);
  interim prologues := 1;
  OnDepth;
    Refpoint(1,0,0);
    Action
    (UseObject(etube, Origin, (0, 90, 0), 1cm,
  "(cosd(t*90), 0, t)", 0.25, -3, 0, true, false));
    Refpoint(-1,0,0);
    Action
    (UseObject(etube, Origin, (0, 90, 0), 1cm,
  "(cosd(t*90), 0, t)", 0.25, 0, 3, false, true));
```

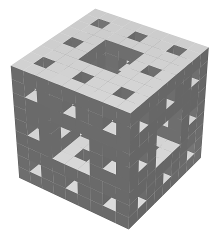


```
    for a = 0 step 45 until 315:
      Refpoint Dir(a+95,0);
      Action
      (UseObject(simpletext, Origin, (a+95, 0, 90), 5pt, "left", "ulft",
     "Come let me sing into your ear;",
     "Those dancing days are gone,",
     "All that silk and satin gear;",
     "Crouch upon a stone,",
     "Wrapping that foul body up",
     "In as foul a rag:",
     "I carry the sun in a golden cup;",
     "The moon in a silver bag."));
    endfor
  endOnDepth;
endfig;
```

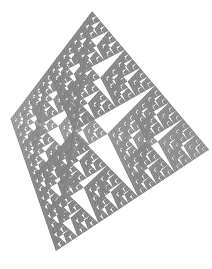


```
let Fill = WireFill;
beginfig(thisfig);
  ObjectColor := (1, 215/255, 0);
  for i = -2 upto 2:
    UseObject(tube, Origin, (0, -90, 0), 0.75cm,
      "(cosd(t*360), sind(t*360), t)", 0.25, i-0.5, i,
      if i = -2: true else: false fi, false); endfor
  ObjectColor := 0.5white;
  UseObject(cylinder, (-2.5, 0, 0)*0.75cm, (0, -90, 0), 0.75cm, 0.4, 5);
  ObjectColor := (1, 215/255, 0);
  for i = -2 upto 2:
    UseObject(tube, Origin, (0, -90, 0), 0.75cm,
      "(cosd(t*360), sind(t*360), t)", 0.25, i, i+0.5,
      if i = 2: true else: false fi, false); endfor
endfig;
```

Here there are two Sierpinski–Menger objects: the sponge (object named `sierpinskip_sponge`) and the gasket (object named `sierpinskip_gasket`). The sponge is drawn with `Fill` equal to `SolidFill` and the gasket with `Fill` equal to `SolidWireFill`. Both objects are defined in `m3Dlib01.mp`. The gasket—since it grows as 4^n where n is the level of recursion—is a lot easier to draw than the sponge—which grows as 20^n. Reaching the level 3 on my current MetaPost implementation for the sponge was not an easy task.



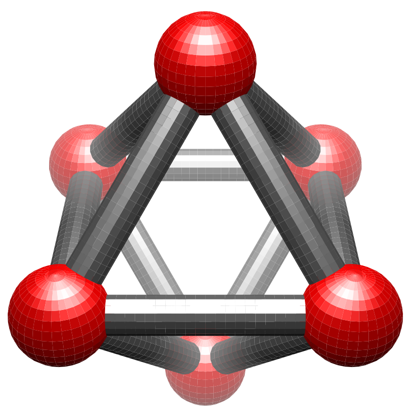
```
ObjectColor:=0.75white;
let Fill = SolidFill;
beginfig(thisfig);
  UseObject(sierpinski_sponge, Origin, (10,0,0), 1.5cm, 2);
endfig;
```



```
beginfig(thisfig);
  UseObject(sierpinski_gasket, Origin, (30,0,0), 1.5cm, 5);
endfig;
```

```
let Fill = SolidFill;
beginfig(thisfig);
  save u; u=3cm;
  ObjectColor := (1, 215/255, 0);
  ObjectPath :=
  (eps, 0.65){right}
  ...{up}(0.45, 1)
  --(0.5, 1){down}
  ...{left}(0.1,0.6)% bowl
  {right}...{down}(0.15,0.55)
  ...{down}(0.075,0.35){down}
  ...{down}(0.075, 0.2)
  ...(0.15,0.15){down}
  ...{left}(0.1,0.1){right}
  ...{right}(0.4,0.05)
  --(0.4,0){left}
  ...{left}(eps,0.05);
  OnDepth;
    % contents
    Refpoint (0,0,0.8u);
    Action (ObjectColor := red;
      UseObject(revolution, (0, 0, 0), Origin, u,
   (eps, ypart point 0.8 of ObjectPath)..point 0.8 of ObjectPath);
      ObjectColor := (1, 215/255, 0););
    % bowl
    Refpoint (0,0,u);
    Action (UseObject(revolution, (0, 0, 0), Origin, u,
   subpath (0.75,3) of ObjectPath););
    % stem
    Refpoint (0,0,0.5u);
    Action (UseObject(revolution, (0, 0, 0), Origin, u,
   subpath (3,8) of ObjectPath));
    % foot
    Refpoint (0,0,0.25u);
    Action (UseObject(revolution, (0, 0, 0), Origin, u,
   subpath (8,11) of ObjectPath));
  endOnDepth;
endfig;
```

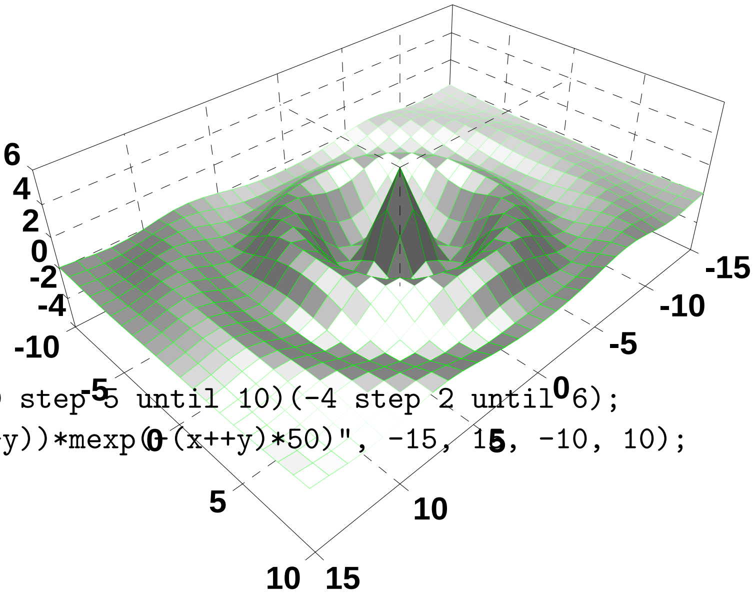
```
Object molecule =
  M1 = (0, 0, 1); M2 = (1, 0, 0); M3 = (0, 1, 0);
  M4 = (-1, 0, 0); M5 = (0, -1, 0); M6 = (0, 0, -1);
  save srad, lrad, j; srad := 0.25; lrad := 0.1;
  OnDepth;
    for i = 1 upto 6:
      Refpoint M[i];
      Action(ObjectColor:=red;
        UseObject(sphere, M[i], Origin, srad));
    endfor
```



```
    for i = 1, 6:
      for j = 2, 3, 4, 5:
        Refpoint 0.5[M[i], M[j]];
        Action(ObjectColor := 0.375white;
          SpheresLink(M[i], M[j], srad, srad, lrad));
      endfor
    endfor
    for i = 2 upto 5:
      Refpoint 0.5[M[i], M[if i = 5: 2 else: i+1 fi]];
      Action(ObjectColor := 0.375white;
        SpheresLink(M[i], M[if i = 5: 2 else: i+1 fi], srad, srad, lrad));
    endfor
  endOnDepth;
endObject;
let Fill = SolidFill;
beginfig(thisfig);
  UseObject(molecule, Origin, Origin, 2cm);
endfig;
```

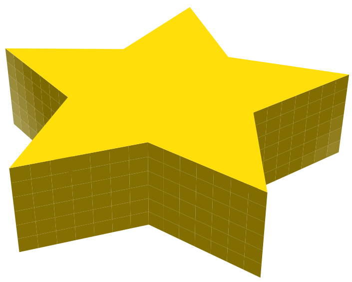
This last example shows the use of the object named `cylinderlike` which is defined in `m3Dplain.mp`.

```
Resolution := 2mm;
let Fill = SolidWireFill;
ObjectColor := 0.5white;
PenColor := green;
%FinePlotFlag:=true;
beginfig(thisfig);
  interim prologues := 1;
  pickup thin.nib;
  Euler(0,0,0,0.2cm);
  Frame(-15 step 5 until 15)(-10 step 5 until 10)(-4 step 2 until 6);
  Plot3D("4cosd(180/3.14159*(x++y))*mexp(-(x++y)*50)", -15, 15, -10, 10);
  FrameMark (0,0,4);
  endFrame;
endfig;
```



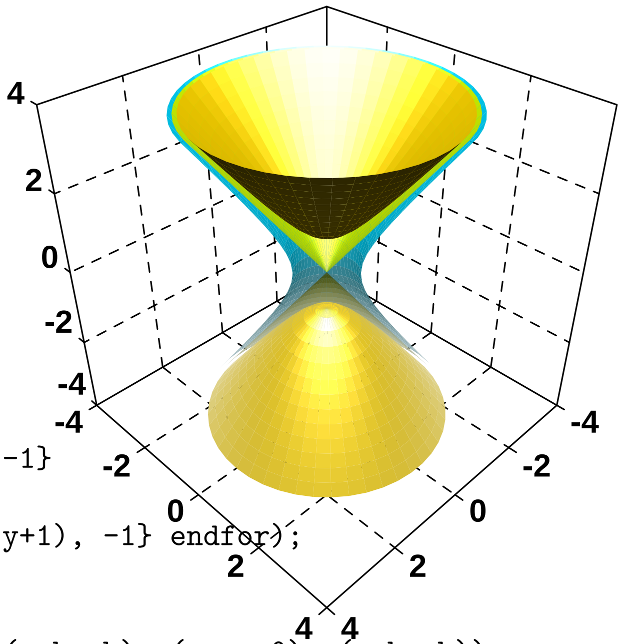
Its specific parameters are an xOy-cycle path and the height of the cyclinder. Thus, former picture as be obtained with

```
let Fill = SolidFill;
ObjectColor := (1, 215/255, 0);
beginfig(thisfig);
  UseObject(cylinderlike,(0,0,0),(0,0,0),1cm,
    for i= 0 upto 4:
      2dir(i/5*360)--dir((i+0.5)/5*360)--
    endfor cycle, 1);
endfig;
```



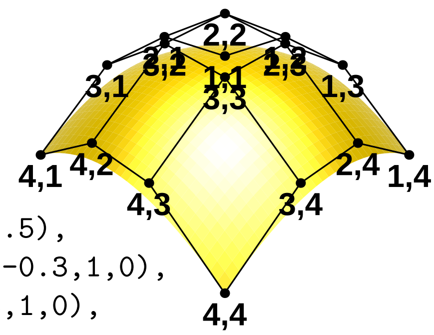
which is rather simple. Then a complex example of the object `revolution`

```
beginfig(thisfig);
  interim prologues:=1;
  Euler(0,0,0,0.5cm);
  save h, dh, r; h=4; dh = 1; r=0.75;
  Frame(-4 step 2 until 4)
    (-4 step 2 until 4)
    (-4 step 2 until 4);
  Inside;
  % hyperboloide  une nappe
  ObjectColor := (0, 215/255, 1);
  UseObject(revolution,Origin,Origin,1,
    (r*sqrt(h*h+1), h){-r*h/sqrt(h*h+1),-1}
    for y = h-dh step -dh until -h-eps:
      ...(r*sqrt(y*y+1), y){-r*y/sqrt(y*y+1), -1} endfor);
  % cone
  ObjectColor := (215/255, 1, 0);
  UseObject(revolution,Origin,Origin,1, (r*h, h)--(eps, 0)--(r*h,-h));
  % hyperboloide  deux nappes
  ObjectColor := (1, 215/255, 0);
  for side = "Inside", "Outside":
    scantokens side;
    UseObject(revolution,Origin,Origin,1,
      reverse((eps, 1){right} for y = 1+dh step dh until h+eps:
     ...(r*sqrt(y*y-1), y){r, sqrt(y*y-1)/y} endfor));
  endfor
  UseObject(revolution,Origin,Origin,1,
    (eps, -1){right} for y = 1+dh step dh until h+eps:
      ...(r*sqrt(y*y-1), -y){r, -sqrt(y*y-1)/y} endfor);
  endFrame;
endfig;
```



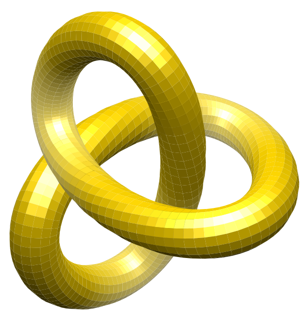
and a Bézier patch which is obtained with the control sequence `BezierPatch` (control points are shown if an only if `tracingchoices` > 0).

```
tracingchoices:=1;
beginfig(thisfig);
  interim prologues := 1;
  interim CurrentScale := 1.5cm;
  BezierPatch(10,
    (-1,-1,-0.5), (-1,-0.3,0), (-1,0.3,0), (-1,1,-0.5),
    (-0.3,-1,0), (-0.3,-0.3,0.5), (-0.3,0.3,0.5), (-0.3,1,0),
    (0.3,-1,0), (0.3,-0.3,0.5), (0.3,0.3,0.5), (0.3,1,0),
    (1,-1,-0.5), (1,-0.3,0), (1,0.3,0), (1,1,-0.5));
endfig;
```



The object named `tube` may be too simple in some complex situations like drawing knots:

```
Resolution:=1mm;
Ox:=(1,0,0);
Oy:=(0,1,0);
Oz:=(0,0,1);
```

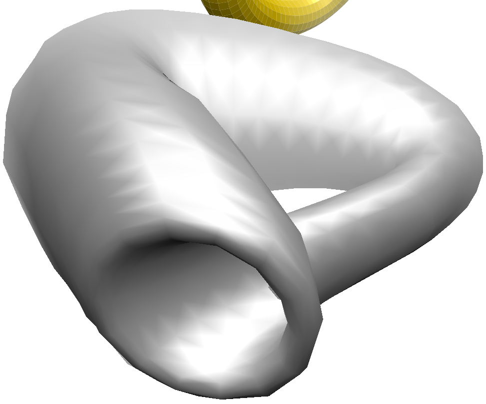


```
beginfig(thisfig);
  UseObject(tube,Origin,Origin,0.5cm,
    "(cosd(t)+2cosd(2t), sind(t)-2sind(2t), 2sind(3t))",
    0.5, 360, 0, false, false);
endfig;
```

That's why more complex version of this object has been defined: `etube` (enhanced tube).

```
beginfig(thisfig);
  UseObject(etube,Origin,Origin,0.5cm,
    "(cosd(t)+2cosd(2t), sind(t)-2sind(2t), 2sind(3t))",
    0.5, 360, 0, false, false);
endfig;
```

At last an example of direct EPS output with the use of smooth shading:



```
Resolution:=1cm;
Ox := (-sqrt(1/2), -sqrt(1/6), sqrt(1/3));
Oy := (sqrt(1/2), -sqrt(1/6), sqrt(1/3));
Oz := (0, sqrt(2/3), sqrt(1/3));
ObjectColor := 0.75white;
DirectEPS jobname&"."&decimal(thisfig);
   Euler(120, 0, 0, 0.5cm);
   OutsideOut;
   UseObject(klein_bottle, Origin, Origin, 1);
endDirectEPS;
```

Table of Contents