An introduction to GNU 3DLDF

Laurence Finston

Abstract

This article is an introduction to GNU 3DLDF. GNU 3DLDF is a package for three-dimensional drawing with METAPOST and METAFONT output. It implements a language based on the METAFONT language with many additional data types and operations. It is designed for general technical drawings and a particular focus is intersections of geometrical figures.

Introduction

GNU 3DLDF is a package for three-dimensional drawing with METAPOST and METAFONT output. It implements a language based on the METAFONT language with many additional data types and operations.

METAFONT is a program for font design; its output is run-length encoded (i.e., compressed) bitmaps which may be converted to a form suitable for display on computer monitors or for printing. Since METAFONT was completed in 1984, scalable fonts have become the de facto standard, so METAFONT's bitmap format, though still usable, and despite the nice features of METAFONT fonts, unfortunately may be considered largely obsolete. In current TEX distributions, PostScript or OpenType versions of Knuth's Computer Modern fonts, originally programmed in METAFONT, are used by default.

METAPOST is a modified version of METAFONT that produces output in the form of PostScript code. While METAFONT is specifically designed for the purpose of font design, METAPOST may be used for technical drawings in general. However, while it includes some features not present in METAFONT, it has not diverged very far.

A first example

The following example is intended to give a first impression of 3DLDF. It shows a circle rotated and shifted in 3D space while at the origin, a set of arrows point in the directions of the positive and negative x-, y- and z-axes. The drawing is projected using the *perspective projection*, so that the arrows representing the z-axis are foreshortened.

Save the following 3DLDF code in a file named `minimal.ldf` (downloadable reference given at the end):

```
verbatim_metapost "prologues := 3;"
   & "outputtemplate := \"%j%3c.eps\";";
numeric frame_wd, frame_ht;
path frame;
frame_wd := frame_ht := 2cm;
frame := (-frame_wd, -frame_ht)
 -- (frame_wd, -frame_ht)
 -- (frame_wd, frame_ht)
 -- (-frame_wd, frame_ht)
 -- cycle;
pen medium_pen;
medium_pen := pencircle
   scaled (.375mm, .375mm, .375mm);
pickup medium_pen;
focus f;
set f with_position (-20cm, 20, -50)
  with_direction (-20cm, 20, 10)
  with_distance 70;

beginfig(1);
  circle c;
  c := (unit_circle scaled (1cm, 0, 1cm)
     rotated (50, 30, 0))
        shifted (2.25cm, .75cm, 2cm);
  draw frame shifted (1cm, 1cm);
  draw c;
  label("$c$", get_center c);
  drawdblarrow (-.5cm, 0, 0) -- (.5cm, 0, 0);
  drawdblarrow (0, -.5cm, 0) -- (0, .5cm, 0);
  drawdblarrow (0, 0, -.5cm) -- (0, 0, .5cm);
  label.top("$x$", (.5cm, 0));
  label.rt("$y$", (0, .5cm));
  label.lft("$z$", (0, 0, .5cm));
endfig with_focus f;
verbatim_metapost "end";
end;
```

Run the following commands:

```
3dldf minimal.ldf
mpost -numbersystem="double" minimal.mp
```

This is the result (`minimal001.eps`):

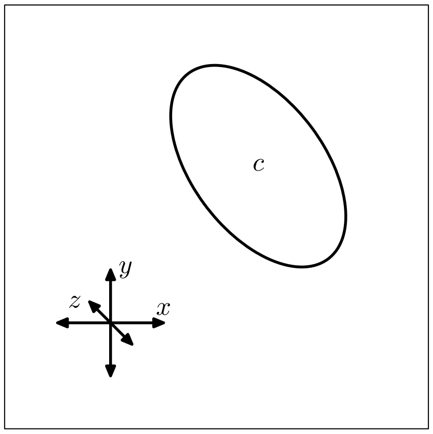

Fig. 1.

doi.org/10.47397/tb/43-3/tb135finston-3dldf

It is a “standalone” Encapsulated PostScript file (EPS) that can be viewed in a PostScript viewer such as Ghostview or Evince (Document Viewer) or included in a TeX file by means of the `\epsffile` macro defined by the `epsf.tex` file.

Save the following TeX code in `minimal.tex`:

```
\input epsf
\nopagenumbers \headline={}
%% DIN A4 Portrait
\special{papersize=210mm, 297mm}
\hsize=210mm \vsize=297mm
\parindent=0pt \parskip=0pt
\baselineskip=0pt
\advance\voffset by -1in
\advance\hoffset by -1in
\advance\hoffset by .75cm
\advance\voffset by 1cm
\def\epsfsize#1#2{#1}
\leftline{\epsffile{minimal001.eps}}
\bye
```

Run the following commands to create a PDF file containing the figure:

```
tex minimal.tex
dvipdfmx minimal.dvi
```

MetaPost output can also be read directly by pdfTeX and LuaTeX (the `{1}{1}` are scale factors):

```
\input supp-pdf
\convertMPtoPDF{minimal001.eps}{1}{1}
```

Motivation

In the 1980s I learned to make perspective drawings in the traditional way by hand, which is a tedious and error-prone procedure. In 1991, I had access to the computer-aided design (CAD) software AutoCAD and was excited by the possibility of using it to make “three-dimensional” drawings. Simultaneously, I had begun to learn to program in the computer language C.

Like a number of other interactive computer programs, such as Emacs or GIMP, AutoCAD implements a *command interpreter* in a language based on LISP. In the case of AutoCAD, it’s called “AutoLISP”. I soon discovered that I preferred “programming” my drawings rather than constructing them by pointing and clicking with a mouse.

In 1996, after several years of experience with TeX, I first used METAFONT for a project involving TeX for which I required some special characters. I enjoyed it and quickly discovered METAPOST and began using it for drawings. Soon I had the idea that it would be nice to have a 3D version of METAPOST, especially since AutoCAD was (and still is) very expensive and at the time required special equipment.

Some time later, I learned C++ for a job. At first I was reluctant but then I discovered that I liked the language and since then, had it in the back of my mind that I’d like to use it for some future project.

In 2002, I finally had the opportunity to do this and began to realize my idea of a “3D METAPOST” using C++. In 2003, 3DLDF was accepted into the GNU Project of the Free Software Foundation and I have continued to develop it since then.

The relationship of 3DLDF to METAFONT and METAPOST

As stated above, 3DLDF implements a language based on the METAFONT language. A 3DLDF program should therefore have a familiar “look and feel” to users of METAFONT and/or METAPOST. However, the implementation of 3DLDF has nothing to do with that of METAFONT or METAPOST. The latter were originally programmed with `WEB` in Pascal under the constraints on computer hard- and software that applied in the late 1970s to the mid 1980s. The “official” distributions of METAFONT and METAPOST use a version converted from Pascal to C using `web2c` (`https://tug.org/web2c`). However, the new versions remain close to the originals and have *not* been rewritten to reflect changes in computer hard- and software since the time when they were created. METAPOST only has added a feature enabling arithmetical calculations with a higher precision than in the original [4, Appendix A, “High-precision arithmetic with MetaPost”, p. 78].

METAFONT implements a *command interpreter* or just *interpreter*. There are many programs of this type and two tasks they require are *lexical scanning* or just *scanning* and *parsing*. METAFONT implements these functions with hand-written, optimized code intended by their author, Donald Knuth, to demonstrate the “Art of Computer Programming” (the title of his *magnum opus*, [6]) by efficiently solving a complex task under very strict constraints with respect to storage space and execution time.

While I can appreciate METAFONT as a work of technical mastery or even art, I don’t consider it an example of the best way to write a computer program in the 2000s or 2020s, when the constraints that applied when METAFONT was developed have simply ceased to exist and when computers and programming tools can be found on nearly every desktop or even in most people’s pockets.

For this reason and as a matter of practicality, I’ve programmed 3DLDF using C++ along with the very comprehensive C++ Standard Library and other software libraries such as the GNU Scientific Library and the pthreads library. For parsing, I use the package GNU Bison. One of the files Bison produces is a text document describing the grammar rules of the language implemented by the parsing function in Backus-Naur format. This is familiar to readers of *The TEXbook* and *The METAFONTbook*, where Knuth uses it to describe the grammar rules of the TEX and METAFONT languages, respectively.

For the main scanner, I do not use the Flex package often used for this purpose, although I do use it for other tasks within 3DLDF. The reason is that 3DLDF attempts to duplicate METAFONT's scanning procedure which operates according to a different principle than Flex. It is also particularly simple to implement as it requires only a single token of "lookahead" [7, ch. 6, "How METAFONT Reads What You Type", p. 49ff.].

If I were to have an idea how to perform a particular programming task in a new and more efficient manner, I believe it would be more useful if I were to contribute it to an existing software library or make it available in some other way, rather than simply incorporating it into a "monolithic" program. In fact, 3DLDF is implemented as a shared or unshared library and linked with a file ("compilation unit") containing a `main` function. The library may just as easily be linked with other `main` functions.

Main differences between METAFONT, METAPOST and 3DLDF

Since METAPOST is a development of METAFONT and remains so close to it, in the following I will generally refer only to METAFONT and in most cases, what I say will apply equally to METAPOST, unless otherwise noted or it refers to features not present in METAPOST, such as those involving digitization [7, ch. 24, "Discreteness and Discretion", p. 195ff.].

If, on the other hand, I refer to METAPOST, it will be because I'm referring to features specific to METAPOST and not present in METAFONT.

Most of the differences between METAFONT and 3DLDF are consequences of the addition of the third dimension. Other differences are due to the fact that METAFONT was specifically designed for the needs of type designers while 3DLDF is intended for general technical drawing. Finally, some are due to my own personal preferences.

A difference one must bear in mind is that METAFONT, METAPOST and 3DLDF each have different *canonical units*: In METAFONT, they are pixels, in METAPOST PostScript points, a.k.a. "big points" (**bp**) and in 3DLDF they are centimeters (**cm**). It is worth noting that in TEX, there are no canonical units and it will cause an error if a dimension is specified without units:

```
\dimen0=5\relax
! Illegal unit of measure (pt inserted).
[...]
? h
Dimensions can be in units of em, ex, in, pt,
pc, cm, mm, dd, cc, bp, or sp; but yours is
a new one!
```

`\relax` is needed because otherwise TEX will wait for further input containing the dimension specifier.

Equations and assignments. METAFONT supports programming in a "declarative" rather than an "imperative" style [7, p. 87]. This idea would appear to have been "in the air" at the time Knuth created METAFONT. However, in my opinion, like the New Math, it has not stood the test of time [2, pp. 1–2].

While for METAFONT, Knuth expresses a preference for the use of *equations* rather than *assignments*, unfortunately this is currently not possible for 3DLDF. METAFONT is able to keep track of dependencies and solve equations once sufficient data is available [9, ch. 28, "Dynamic linear equations", §585; ch. 29, "Dynamic nonlinear equations", §618] and [10].

I would like to implement this feature, but at the present time I don't know how to go about it nor whether it would be possible with 3D data. Nor do I consider this to be a priority. However, since I may yet do so, the `=` operator is reserved for equations, as in METAFONT, and `:=` must be used for assignments.

3DLDF does implement "declarative" forms of operations, such as transformations:

```
path q;
q := (0, 0, 0) -- (1, 1, 1)
   rotated (10, 15, 20);
```

However, for users who aren't afraid of hurting the computer's feelings, corresponding "imperative" operations are available as well:

```
rotate q (10, 15, 20);
```

In addition, 3DLDF implements the *operators* for assignment plus an arithmetical operation `+=`, `-=`, `*=`, and `/=` for different variable types, as appropriate. For example, all of them are available for **numeric**s, `+=` for vector-type variables (see "Vector-type objects", p. 328) and `*=` for applying transformations to **point**s, **path**s, etc.:

```
point p; p := (1, 2, 3);
point_vector pv; pv += p;
transform t;
t := identity scaled (3, 4, 5);
pv0 *= t;
show pv0;
```

results in:

```
point:
World coordinates:
(3.0000000, 8.0000000, 15.0000000, 1.0000000)
```

These "imperative" operators make it possible to avoid clumsy constructions such as `q := q rotated (10, 20, 30)` (which do also work, however).

Incidentally, the operators for assignment plus an arithmetical operation described above break the parsing rules of METAFONT. However, they were easy to implement and have never caused any problems.

Projections. For a 3D graphics program to be useful, the objects for which three-dimensional data are stored must be *projected* onto a two-dimensional plane for display on a computer screen or printing. 3DLDF implements two kinds of projection: parallel projection onto one of the major planes (x-y, x-z or y-z) and the *perspective projection.* In fact, it's possible to project a drawing onto an arbitrary plane by transforming the desired plane so that it comes to lie in a major plane, transforming the objects to be drawn in the same way, and then projecting them onto the latter plane. Other projections of interest, but not yet implemented, are projections onto a cylinder, sphere or other curved surfaces and the Mercator projection and other projections used for maps.

Though parallel projections are extremely useful, even for 3D drawings, it is the *perspective projection* that is most closely associated with 3D graphics. It essentially simulates the effect of instantaneously photographing a scene with a camera or viewing it with one immobile eye. The result of the perspective projection is as if a line were drawn from each point in a scene to a *focus* represented by single point (the camera lens or the lens of an eye) and the intersection of this line with a plane (the *plane of projection* or *picture plane*). This plane may be imagined to be between the focus and the scene (the normal procedure), behind the scene or behind the focus, in which case the image appears upside-down, as in a camera obscura or the retina.

In addition to the position of the focus, the direction of view and the distance from the focus to the plane of projection must be specified and the "upwards" direction calculated. In 3DLDF, this data is stored in an object of type **focus**. To change the upwards direction, the focus may be rotated about the line from the position through a point lying in the direction of view from the position.

```
focus f; set f with_position (-10cm, 20, -50)
       with_direction (-10cm, 20, 10)
       with_distance 70;
show f;
⟶
focus:
position:
World coordinates:
(-10.0000000, 20.0000000, -50.0000000, 1.0000000)
direction:
World coordinates:
(-10.0000000, 20.0000000, 10.0000000, 1.0000000)
up:
World coordinates:
(-10.0000000, 21.0000000, -50.0000000, 1.0000000)
distance == 70.00000000.
   axis == z
angle == 0.00000000
```

Pairs and points

The most basic "drawable" object type in METAFONT is the **pair**, which is used to represent a point in the plane. It consists of two numerical values, an x- and a y-coordinate:

```
pair p; % METAFONT
p = (1cm, 2cm);
```

In 3DLDF, **pair** is replaced by the object type **point**, which is used to represent a point in three-dimensional space:

```
point p; % 3DLDF
p := (1cm, 2cm, 3cm);
```

From the point of view of a user, a **point** has three coordinates, **x**, **y** and **z**. However, in fact, **point**s have an additional fourth coordinate, namely **w**. Such a set of coordinates is called *homogeneous*. The w-coordinate is normally 1 but will usually be $\neq 1$ when the **point** is projected using the perspective projection, as described in the previous section. In addition, the fourth coordinate is needed in order to be able to multiply the **point** with a 4×4 matrix, which shall occupy our attention below.

In 3DLDF, **point**s have four sets of coordinates, "world", "perspective", "user" and "view", of which currently only "world" and "perspective" are in use.

METAFONT's **pair**s and 3DLDF's **point**s are also used to represent the *difference* between two points in (2D or 3D) space, that is, the result of subtracting one point from another:

```
pair a, b; %% METAFONT
show a - b;
>> (-xpart b+xpart a,-ypart b+ypart a)
a = (2, 3);
b = (1, 2);
show a - b;
```

```
>> (1,1)
```

```
point a, b, c; %% 3DLDF
a := (2, 3);
b := (1, 2);
c := a - b;
show c;
⟶
point:
World coordinates:
(1.0000000, 1.0000000, 0.0000000, 1.0000000)
```

The result of subtracting one point from another is called a *vector*, which has a *magnitude* and a *direction*, but no location in space. **pair**s in METAFONT and **point**s in 3DLDF are used to represent both points in (2D or 3D) space and vectors, and the same object may be interpreted as a point or a vector, depending on circumstances.

In 2D, the magnitude is the distance to a point with the same x- and y-coordinates, i.e., $\sqrt{x^2+y^2}$ and similarly in 3D, with the added z-coordinate, i.e., $\sqrt{x^2+y^2+z^2}$ and the direction is that indicated by a line from the origin to that point. Such vectors are of great importance in 3D graphics. They should not be confused with the data type **vector** in C++ or other uses of the term. A *unit vector* may be created by dividing the x-, y- and z-coordinates of a **point** by the magnitude (input will be directly followed by output from now on):

```
point p[];
p0 := (2, 2, 2);
n := magnitude p0;
show n;
>> 3.4641
p1 := (2/n, 2/n, 2/n);
show p1;
point:
World coordinates:
(0.5773503, 0.5773503, 0.5773503, 1.0000000)
show magnitude p1;
>> 1
```

Since unit vectors are needed so often, 3DLDF implements the **unit_vector** operator for this purpose:

```
p2 := unit_vector p0;
show p2;
point:
World coordinates:
(0.5773503, 0.5773503, 0.5773503, 1.0000000)
show magnitude p2;
>> 1
```

In METAFONT programs, instead of using **pair**s, it is the convention to use **z** for representing points, e.g., `z = (4pt, 5pt);`. This convention is implemented by means of a **vardef** macro [7, p. 277]:

```
vardef z@#=(x@#,y@#) enddef
```

There is no correspondence to this macro in 3DLDF. For one thing, since **point**s already have a **z**-coordinate, **z** would be a poor choice for the default name for points and there is no other obvious choice, since "**p**" would be equally appropriate for **path**s and possibly even for **polygon**s or **parabolæ** (to say nothing of **polyhedra** or **paraboloid**s). Therefore, in 3DLDF, explicitly declared **point**s must always be used to represent points in space (and vectors).

Transformations

METAFONT implements the *transformations* shifting (translation), scaling and rotation by means of the object type **transform**, which consists of 6 numerical elements:

```
transform t; %% METAFONT code
show t;
>> (xpart t,ypart t,xxpart t,xypart t,
yxpart t,yypart t)
```

A **pair** may be "transformed" like this:

```
pair a, b;  %% METAFONT code
a = (1, 2);
transform t;
t = identity scaled (2, 4);
show t;
>> (0,0,2,0,0,4)
b = a transformed t;
show b;
>> (2,8)
```

The *identity* transformation is needed as a starting point for other transformations and looks like this:

```
show identity; %% METAFONT
>> (0,0,1,0,0,1)
```

In 3D, transformations are represented by 4×4 matrices of numerical values, which are called *transformation matrices*. Therefore, even if the perspective projection wasn't needed, **point**s would have to have 4 coordinates in order that they may be multiplied with transformation matrices. The identity matrix in 3DLDF looks like this:

```
show identity;
transform:
     1        0        0        0
     0        1        0        0
     0        0        1        0
     0        0        0        1
```

In 3DLDF, the syntax with **transformed** is supported. However, it also implements the operation "multiplication with assignment", so that the operator `*=` may be used to perform a matrix multiplication on a **point**:

```
point p;
p := (1, 2, 3);
transform t;
```

```
t := identity scaled (2, 3, 4);
show t;
transform:
      2       0       0       0
      0       3       0       0
      0       0       4       0
      0       0       0       1
p *= t;
show p;
point:
World coordinates:
(2.0000000, 6.0000000, 12.0000000, 1.0000000)
```

Paths

While **pair**s in METAFONT and **point**s in 3DLDF are the basic building blocks of drawings, **path**s are an essential element of any font or technical drawing: Without **path**s, all you have is a scatter plot.

In METAFONT, **pair**s and **path**s are the only "drawable" types. Font design tends to use free-form curves, so Knuth clearly didn't consider it necessary to define data types for algebraic curves, for example. `plain` METAFONT does define **quartercircle**, **halfcircle**, **fullcircle** and **unitsquare**, but these are constants of type **path**. METAFONT doesn't store any additional information about them, such as their centers or radii. It also defines **superellipse** as a macro.

3DLDF, on the other hand, is intended for general technical drawing. Many technical drawings consist solely of straight lines and where curves are used, by far most often they are circles, circular arcs, ellipses and elliptical arcs. Ellipses play a special role because circles appear elliptical in perspective. A common tool for technical drawing is (or was) stencils for drawing perspective ellipses.

Other curves occur frequently in drawings for special purposes: helices (non-planar) for representing screw threads, epi- and hypocycloids for gear teeth [1, pp. 53–55], parabolæ for trajectories, catenaries for hanging chains or ropes, Cartesian ovals for optics, etc. There are many interesting algebraic curves that appear in illustrations of works on geometry but are as rare as hen's teeth in technical drawings.

Among the plane figures consisting of straight lines, triangles, squares and rectangles are the most common, while other regular polygons are occasionally used.

Because of their importance in technical drawings and because the subject particularly interests me, in addition to **path**, 3DLDF defines the following object types.

For representing polygons:

- **triangle**
- **rectangle**
- **polygon**
- **reg_polygon** (regular polygon)

For the conic sections, 3DLDF implements the following types:

- **circle**
- **ellipse**
- **parabola**
- **hyperbola**

In addition, and unlike METAFONT, 3DLDF defines **superellipse** as an object type.

3DLDF also has types for solid geometric figures. Figures consisting of straight lines:

- **cuboid**
- **polyhedron**

Figures with a simply curved surface:

- **cone**
- **cylinder**

Quadric surfaces:

- **sphere**
- **ellipsoid**
- **paraboloid**
- **hyperboloid**

In addition, 3DLDF implements many predefined constants of these types:

```
unit_square  unit_pentagon unit_circle
unit_ellipse unit_cuboid   unit_sphere
unit_ellipsoid ...
```

The planar figures `unit_ellipse`, etc., are constructed in the x-z plane:

```
beginfig(2);
  ellipse e;
  e := unit_ellipse scaled (2cm, 0, 1cm);
  draw e;
endfig with_projection parallel_x_z;
```

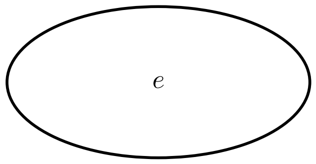

Fig. 3.

Ultimately, the solid figures consist of planar **path**s, a **polyhedron** of **polygon**s, a **sphere** of **circle**s, an **ellipsoid** of **ellipse**s, etc., and additional information is stored for the center, foci, axis lengths or other salient features of the figure.

They may be used in technical drawings for *wireframe* constructions. 3DLDF doesn't implement any form of surface hiding or rendering. In the case of cuboids or polyhedra, surface hiding may be done "by hand". For drawings that aren't too complex, this may work well enough:

```
  cuboid c[];
  rectangle r[];
  c0 := unit_cuboid scaled (1.5, 1.5, 1.5);
  c1 := c0;
  c2 := c0;
  rotate c1 (0, 30);
  shift c1  (-1.5cm, 0);
  draw c1;
  for i = 0 upto 5:
    r[i] := get_rectangle(i) c1;
  endfor;
  rotate c2 (0, -30);
  shift c2 (1.5cm, 0);
  draw c2;
  for i = 0 upto 5:
    r[i+6] := get_rectangle(i) c2;
  endfor;
  unfilldraw r6;
  unfilldraw r9;
  unfilldraw r10;
```

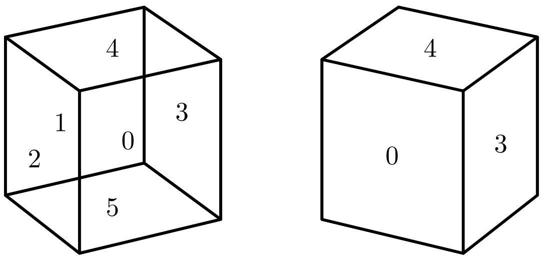

Fig. 4.

For curved surfaces, such as spheres or ellipsoids, this isn't possible, as finding the curve that represents the edge of a curved surface from a particular point of view is non-trivial. Please notice the left and right edges where the nearly horizontal circles appear to extend slightly past the nearly vertical ones.

```
sphere s;
s := unit_sphere scaled (2.5, 2.5, 2.5);
rotate s (10, 11, 10);
draw s;
```

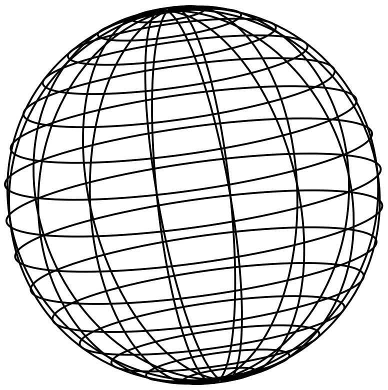
Fig. 5.

In the long run, the best solution for this problem would be to have 3DLDF produce output in a form suitable as input for a specialized rendering program, such as Blender (`https://blender.org`).

Path details. In METAFONT, **path**s are implemented as Bézier curves, which are a kind of *spline curve*. They may be specified in various ways, but are ultimately stored in the form [7, p. 13]:

$p = z_0$ .. controls u_0 and v_1 .. z_1 $\langle etc. \rangle$
z_{n-1} controls u_{n-1} and v_n .. z_n

As in this example:

```
tracingonline := 1; % METAFONT
path p;
p = fullcircle xscaled 2 yscaled 3;
show p;
>> Path at line 4:
(1,0)..controls (1,0.39784)
   and (0.89465,0.77939)
 ..(0.70712,1.06068)..
 controls (0.51959,1.34198)
 and (0.26523,1.5) ..(0,1.5)
```

Bézier curves are *invariant* under the affine transformations translation (shifting), rotation, and scaling, as long as they are scaled by the same amount in all dimensions [7, p. 132].

Unfortunately, Bézier curves are *not* invariant under the non-affine perspective projection, which is essential for 3D graphics. There is a generalization of the Bézier curve, that is, another spline curve, that is invariant under this projection, namely non-uniform rational B-splines or NURBs. With Bézier curves, conceptually, the control points exert a "pull" on the path and the amount of "pull" is equal for all of the control points. NURBs, on the other hand, have an additional "weight" parameter which makes it possible to "weight" the control points individually, so that they can exert different amounts of "pull". A Bézier curve is therefore equivalent to a similar NURB where all of the weights are 1.

One of the most important features of METAFONT is that, given a set of points on a **path**, it will attempt to draw the "most pleasing" curve through it [7, ch. 3, "Curves", p. 13ff.]. It provides a set of operations for the user to influence the shape of the curve by giving "hints": **dir**, **tension**, **atleast**, etc.

NURBs are not yet implemented in 3DLDF, although this is planned. **path**s are stored as the **point**s on the **path** and *connectors*, also known as *path joins*, may be specified in the same way as in METAFONT, i.e., with control points, **tension**, **dir**, {**point**}, $\{(x, y)\}$, $\{(x, y, z)\}$, and **atleast**. They are *not* converted to the form using "controls", as above, within 3DLDF but rather the connectors are written verbatim to the output (METAPOST and METAFONT code), except that any **point**s referred to are transformed along with the **path** whenever a transformation, including the projection upon output, is applied to the **path**. They therefore will only produce correct results for 2D objects that lie in the plane of projection, or a plane parallel to it, when a parallel projection is being used.

When METAFONT or METAPOST is run on the 3DLDF output, it will attempt to draw the "most pleasing" curve through the points on the path, accounting for any hints given with **dir**, **tension**, etc. *However*, METAFONT and METAPOST's idea of the "most pleasing" curve is based purely on the two-dimensional points on the path: They don't "know" that they represent the projection of a three-dimensional curve and the results are likely to be neither pleasing nor correct, *unless* the path is constrained by containing a sufficient number of points. In the special case that a parallel projection is used and the path lies in the plane of projection or a plane parallel to it, the result will, however, be correct, as it will be equivalent to just using METAFONT or METAPOST in the first place.

```
circle c;
set c with_center origin with_diameter 4
   with_point_count 8;
rotate c (-10, 0);
draw c;
```

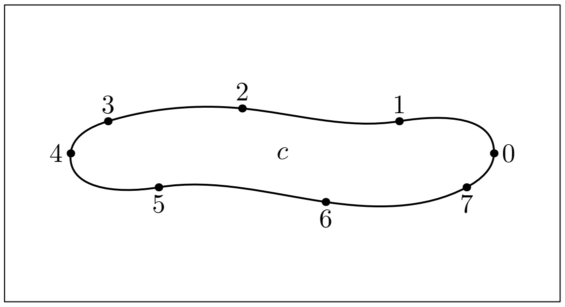

Fig. 6.

```
circle c;  % same, but with more points
set c with_center origin with_diameter 4
   with_point_count 64;
rotate c (-10, 0);
draw c;
```

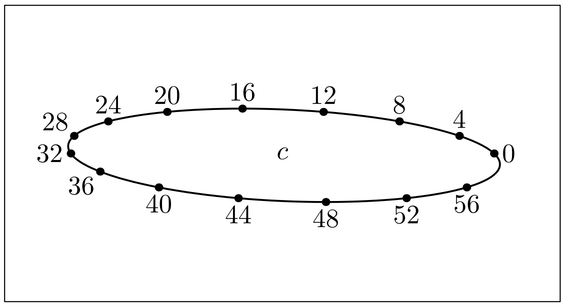

Fig. 7.

When projecting a 3D path using the perspective projection, **tension**, **dir**, $\{(x, y)\}$ and **atleast** are very likely to produce erroneous results, especially if an insufficient number of points have been specified. If the path is constrained sufficiently, this may not be noticeable; however, if any of these hints are used, the best practice is to replace the connectors with `..` before outputting the path, e.g.:

```
path p;
p := origin{up} .. {right}(1, 1){left}
     .. {down}(2, 0);
clear_connectors p;
p += ..;
```

Usually, there is no need to use hints, unless it is intended to use a parallel projection; nevertheless, the situation might arise, especially when working with METAFONT or METAPOST code "borrowed" from another source, such as the original sources for Computer Modern or any PostScript font accessed by means of the **glyph** command.

The situation is somewhat different with control points. Control points may be specified by the user and paths obtained by calling METAPOST from within 3DLDF will always contain them (unless they are subsequently removed).

Intersections. In METAFONT, since all **path**s are constructed in the same way, intersections are found by a method particular to Bézier curves (not described in *The METAFONTbook*). Furthermore, since METAFONT doesn't "know" that two **path**s p_0 `--` p_1 and p_2 `--` p_3 represent straight lines, their intersection is only found if it lies on the line segments they represent, *not* if the lines that contain them intersect.

In contrast to METAFONT, 3DLDF does "know" that a **path** like $(0, 0)$ `--` $(1, 1)$ represents a straight line, that a **circle** represents a circle, and so on, and this is of importance in functions that attempt to find the intersections of objects in a drawing.

In addition, 3DLDF *does* find the intersection point of two lines (within reason) rather than just the intersection points of the line segments. Not within reason would be two lines that are almost parallel so that their intersection point lies outside the region where the computer could calculate it reliably.

```
point p[];
path q[];
q0 := origin -- (2cm, 2cm);
q1 := (0, 2cm) -- (2cm, 0);
draw q0;
draw q1;
p0 := q0 intersectionpoint q1;
```

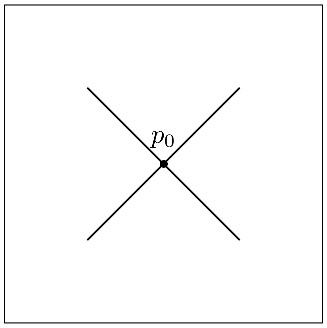

Fig. 8.

```
q0 := origin -- (2cm, 2cm);
q1 := (3cm, 0) -- (2.5cm, 2cm);
draw q0;
draw q1;
p0 := q0 intersectionpoint q1;
```

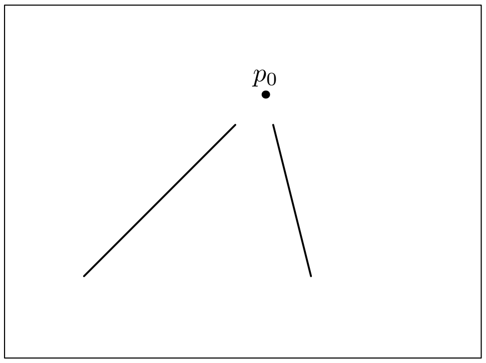

Fig. 9.

In this example, the result of **intersectionpoint** is stored in a **point**. In fact, the result is really an object of a *compound type* named **bool_point** (not present in METAFONT) consisting of a **boolean** and a **point**. In the case of the **intersectionpoint** operation, the **boolean** part of the **bool_point** indicates whether the **point** (if found) lies on one or both of the line segments.

If the lines do not intersect, the **point** returned in the **bool_point** is `INVALID_POINT` whose coordinates are the triple `(INVALID_REAL, INVALID_REAL, INVALID_REAL)`. Actually, neither `INVALID_POINT` nor `INVALID_REAL` are "invalid" from the point of view of the hardware or C++: `INVALID_REAL` is simply an arbitrary numerical value used within 3DLDF for testing whether an operation has succeeded or not. In fact, it is the largest **float** value available on a given computer.

The ability of 3DLDF to find the intersections of lines is useful as a way of compensating for its inability to interactively solve linear equations like METAFONT. In the latter, intersections are typically found using the "nullary" operation **whatever**, which is defined in `plain.mf` and `plain.mp` as a **vardef** macro [7, p. 264]:

```
path q;  % METAPOST
q0 = origin -- (2cm, 2cm);
q1 = (3cm, 0) -- (2.5cm, 2cm);
draw q0;
draw q1;
z0 = whateverorigin, (2cm, 2cm);
z0 = whatever(3cm, 0), (2.5cm, 2cm);
dotlabel.lft(btex $z_0$ etex, z0);
```

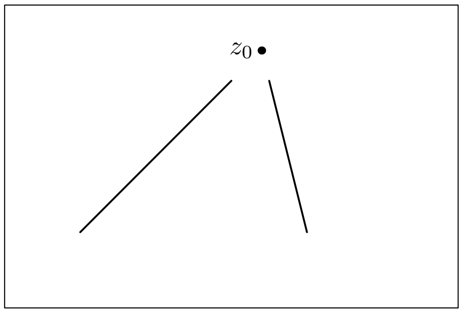

Fig. 10.

Intersections in 3DLDF. Finding the intersections of algebraic curves and surfaces in the plane and in 3D space is a focus of 3DLDF and a particular interest of mine. In some cases this is straightforward, whereas in others it is less so, or would involve higher mathematics currently beyond my abilities. Depending on the objects, their intersections may be points, curves, planes or curved surfaces.

The routines for finding intersections as in the previous examples all use the algebraic formulæ for the objects involved. One problem with this approach is that calculations involving real values (**float**s, **double**s or **long double**s) on a computer are not exact, so that there are always rounding errors. These in turn may cause intersections that exist to not be found. In particular, transforming objects by means of matrix multiplication tends to introduce rounding errors. For numerical a, b and ϵ, the solution is not to compare $a = b$, but rather $||a| - |b|| < \epsilon$ where ϵ is some small value appropriate to the circumstances.

Another problem is that transformations may cause objects to "go out of shape". 3DLDF places no restrictions on the transformations that may be applied to a drawable object (**point**, **path**, **circle**, etc.). This feature would be useful, for example, when projecting a geometrical object like a circle onto a curved surface: The resulting figure will not be circular, but it retains the object type **circle** and its "center" will be the projection of the center of the original object onto the surface.

In order to deal with this problem, 3DLDF implements tests for whether objects of a given type still fulfill the definition of that type. For example, objects may be tested for circularity with the operator **is_circular**, for whether they are elliptical with **is_elliptical**, etc. In contrast, the operations **is_circle**, **is_ellipse**, etc., test for the object type:

```
beginfig(11);
circle c[];
c0 := unit_circle;
c1 := c0 sheared (1, 1.5, 1.25);
draw c0;
draw c1;
show is_circle c0;
>> true
show is_circle c1;
>> true
show is_circular c0;
>> true
show is_circular c1;
>> false
endfig with_projection parallel_x_z;
```

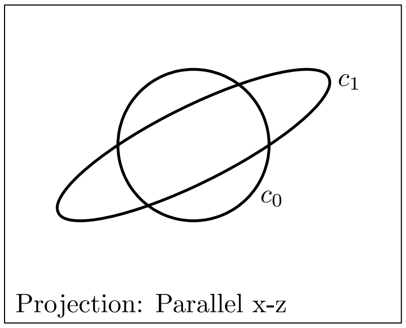

Fig. 11.

Vector-type objects

In METAFONT, *arrays* may be declared like this:

```
numeric n[];
path p[][];
```

Now, the user can assign values to the variables **n**⟨*suffix*⟩ and **p**⟨*suffix*⟩⟨*suffix*⟩:

```
tracingonline := 1; % METAFONT
n0 = 10;
n[n0 - 5] = 6;
show n0;
>> 10
show n[n0 - 5];
>> 6
a = 3;
p[a][n0] = origin .. (1, 1), -- (2, 0) .. cycle;
show p[3][10];
>> Path at line 10:
(0,0)..controls (-0.11444,0.59329)
   and (0.40671,1.11444)
 ..(1,1)..controls (1.33333,0.66667)
 and (1.66667,0.33333)
 ..(2,0)..controls (1.78766,-1.10071)
 and (0.21234,-1.10071) ..cycle
```

METAFONT makes it possible to declare variables using various combinations of *tags* and *suffixes* in a very flexible way [7, ch. 7, "Variables", p. 53ff.]. This works in exactly the same way in 3DLDF. However, continuing the previous example, in METAFONT, a variable *n* would be completely independent of `n0`, `n[n0 - 5]`, etc.:

```
numeric n[];
*n0 := 10;
*n := 5;
*show n0;
>> 10
show n;
>> 5
```

To access all the variables of the form `n`⟨*suffix*⟩, they must be accessed individually. A loop may be used, but then the suffixes must follow a pattern suitable for use as a loop index:

```
for i = 0 upto 5:
   n[i] = 2i;
   show n[i];
endfor;
>> 0
>> 2
>> 4
>> 6
>> 8
>> 10
```

In order to make it more convenient to access all of the members of an array, 3DLDF implements the notion of *vector-types*. In this case, the term *vector* is used in the sense common in the context of computer programming, namely for one-dimensional arrays.

For most types, such as **boolean**, **transform**, **point**, **path**, etc., there is a corresponding vector-type: **boolean_vector**, **transform_vector**, **point_vector**, **path_vector**, etc. For some more rarely-used or specialized types, there is no corresponding vector-type.

There are operations that take vector-type objects as their arguments and operate on all of the objects on the vector. In addition, the operator += may be implemented for a vector-type, where appropriate. For example, it adds a **path** to a **path_vector**:

```
path_vector pv;
pv += (1, 1) -- (2, 2);
pv += origin .. (1, 1, 1) .. (-1, 2, 2)
      .. (3.5, 0, 10);
show pv;
>> path_vector:
size of vector: 2
0:
type:  PATH_TYPE
surface_hiding_ctr:  0
decomposition_level:  0
points.size() == 2
connector_type_vector.size() == 1
points:
(1.00000000, 1.00000000, 0.00000000) --
   (2.00000000, 2.00000000, 0.00000000) ;
 etc.

1:
type:  PATH_TYPE
surface_hiding_ctr:  0
decomposition_level:  0
points.size() == 4
connector_type_vector.size() == 3
points:
(0.00000000, 0.00000000, 0.00000000)
   .. (1.00000000, 1.00000000, 1.00000000)
   .. (-1.00000000, 2.00000000, 2.00000000)
   .. (3.50000000, 0.00000000, 10.00000000) ;
 etc.
```

Vector-type variables can also be used as the return types for operations so that operations may return more than one object. For example, the operation ⟨*ellipse tertiary*⟩ **intersectionpoints** ⟨*ellipse secondary*⟩ returns 0 to 4 points:

```
ellipse e[];
e0 := unit_ellipse scaled (2cm, 0, 1cm);
e1 := e0 rotated (0, 45) shifted
   (.25cm, 0, -.125cm);
rotate e0 (0, -20);
draw e0;
draw e1;
point_vector P;
P := e0 intersection_points e1;
show size P;
>> 4
```

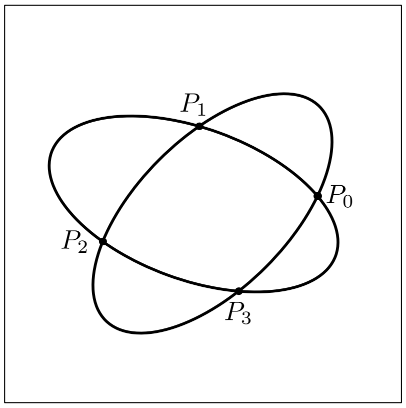

Fig. 12.

It is not possible to declare an array of vector-type objects.

Output and labels

In METAFONT, output is caused by the **shipout** primitive, which "ships out" a character. Normally, it is not called directly by the user, but rather indirectly by the **endchar** macro defined in `plain.mf`, which calls the macro **shipit**, which in turn calls **shipout** with *currentpicture* as its argument.

In METAPOST, **endfig** calls **shipout**, causing PostScript code to be written to the output file [4, p. 46].

In 3DLDF, **endfig** causes *current_picture* to be written to the METAPOST output and **endchar** causes it to be written to the METAFONT output. However, in addition, the **output** operator will cause the **picture** given as its argument, which may be **current_picture**, to be output in the form of METAPOST or METAFONT code to be written to the corresponding output file if called between **beginfig** and **endfig** on the one hand or **beginchar** and **endchar** on the other. This is useful for performing a primitive kind of surface hiding by hand and for creating figures or characters using more than one projection.

output doesn't clear the **picture** passed to it as an argument, so it may be output again, by **output**, **endfig** and/or **endchar**. If it should be cleared, it must be cleared explicitly using the **clear** command.

```
beginfig(1);
  draw unit_circle scaled (2cm, 0, 2cm);
  output current_picture with_projection
     parallel_x_z;
  clear current_picture;
  draw unit_ellipse scaled (3cm, 0, 2cm)
     rotated (90, 0);
   (etc.)
endfig with_focus f;
```

In METAFONT, labels are only used when developing a font and disappear in the final output. The placement is also usually determined by METAFONT and not by the user. They are therefore less important than in METAPOST and 3DLDF, since labels are an essential part of technical drawings.

In 3DLDF, labels work as in METAPOST, with only a few differences. METAPOST typesets labels by default using TEX. The `-tex` option may be used to set the name of the program to be called, e.g., `-tex=latex`.

In METAPOST, a plain **string** is typeset in the default font. On my system, this is Computer Modern Roman 10pt (`cmr10`), but it could be any PostScript font. To use TEX macros in the labels, **btex** ... **etex** or the macro **TEX** must be used. If METAPOST is called with the `-troff` argument, then `troff` is used to typeset material surrounded by **btex** and **etex** (or **verbatimtex** and **etex**). The first argument to **label** or **dotlabel** may be a **string** or a **picture**. See [4, ch. 8, "Integrating Text and Graphics"] for more information.

3DLDF passes the text in labels to METAPOST largely unchanged, so how they are handled depends on how METAPOST is called by the user after 3DLDF has run. However, **btex** and **etex** are always added to the beginning and end, respectively, of the label text. If any user found this undesirable, it would be easy to add an option to suppress this. A **picture** is not permitted as the first argument to **label** or **dotlabel**, but a number may be used, e.g., `dotlabel(0, p0)`.

In 3DLDF, a **picture** contains two kinds of items: 1) *drawable* ones, namely **point**s, **path**s, objects derived from **path** such as **circle** and **ellipse** and **solid**s, such as **sphere** and **ellipsoid** and 2) labels. When a **picture** is output, the drawable items are output first, followed by the labels, so that the latter aren't overwritten. The latter effect can be achieved by use of the **output** command to output a **picture** containing labels prior to **endfig**.

In 3DLDF programs, label commands may also appear between **beginchar** and **endchar**. When a **picture** is output, any labels on it are ignored. Thus the METAFONT output of 3DLDF will never contain labels, even when **mode** is **proofing**. To test the appearance of a character, it's best to produce a corresponding METAPOST figure, where one has the advantage of METAPOST's superior labelling facilities.

Labels in 3DLDF are purely two-dimensional items. Conceptually, they always lie in the plane of projection. The **point** specified as the second argument of **label** or **dotlabel** is projected like all of the other points on the picture and the label is placed in relation to its projection. The **point** may be transformed but the label itself may also be transformed separately. If rotation is desired, it should be rotated about the z-axis. Shifting and rotating make sense; other translations are likely to produce undesirable results.

Calling METAPOST from within 3DLDF

Having numerous object types to store information about algebraic curves and surfaces has certain advantages compared to storing them simply as spline curves, e.g., the ability to access their centers, foci, normals, radii, etc. On the other hand, it's nice to be able to find the intersections of arbitrary curves and surfaces, even free-form ones. Since 3DLDF does not yet implement NURBs, this is unfortunately not possible for the general case of intersections of arbitrary objects in 3D. However, for the special case of coplanar objects, it *is* possible by means of calling METAPOST (not METAFONT!) "indirectly" from within 3DLDF.

METAPOST is called instead of METAFONT because in METAPOST, the type used for arithmetical calculations may be specified using the option `-numbersystem`, allowing for almost arbitrary precision. In METAFONT, calculations are performed using 32-bit fixed-point numbers, with a strict limit on the magnitude of a $\langle$*numerical token*$\rangle$, namely < 4096 [7, pp. 50 and 63].

METAFONT implements many operations on **path**s, as described in *The METAFONTbook*. Because 3DLDF only stores the **point**s on a **path** and the connectors (or *path joins*), and does not calculate the "in-between" points, it is not possible to implement any of the operations that depend on this data. However, in the case of planar **path**s, it is possible to access these operations by calling METAPOST from within 3DLDF.

The operations in question are the following, where n stands for a numerical value, t for a numerical "time" value, p for a **pair** and q for a **path**:

- $p =$ **direction** t **of** q;
- $t =$ **directiontime** p **of** q;
- $p_0 =$ **directionpoint** p_1 **of** q;
- $p =$ **point** t **of** q;
- $p =$ **subpath** (t_0, t_1) **of** q;
- $p = q_0$ **intersectiontimes** q_1;
- $p = q_0$ **intersectionpoint** q_1;

The operation **angle** on a **pair** can be useful in combination with operations on **path**s:

- $n =$ **angle** p;
- $n =$ **angle direction** t **of** p;

Some of these operations have correspondences in 3DLDF for arbitrary (3D) **path**s, but they don't necessarily have exactly the same meaning or work the same way. In some cases, the names of the 3DLDF operations for operating on planar paths by calling METAPOST differ from the names of the corresponding operations in METAFONT names in order to avoid name clashes.

In the following listing, m and u are integers, v is a **numeric_vector** and w is a **point_vector**.

- q_0 := **get_metapost_path** q_1 $\langle options\rangle$
- v := **direction_metapost of** q $\langle options\rangle$
- n := **angle_direction_metapost** n **of** q $\langle options\rangle$
- v := q **intersectiontimes** q $\langle options\rangle$
- v := q **intersectiontimes_all** q $\langle options\rangle$
- p := q **intersectionpoint_metapost** q $\langle options\rangle$
- w := q **intersectionpoints_metapost** q $\langle options\rangle$
- q_0 := **resolve** q_1 (m, n) **to** u $\langle options\rangle$
- q_1 := **subpath** (n, n) **of** q
- q_0 := q_1 **normalized** $\langle options\rangle$

Commands:

- **call_metapost** $\langle$*string expression*$\rangle$ **(** $\langle$*path vector variable optional*$\rangle$ $\langle$*point vector variable optional*$\rangle$ $\langle$*numeric vector variable optional*$\rangle$ **)** $\langle$*options*$\rangle$
- **call_tex** $\langle$*string expression*$\rangle$ **,** $\langle$*numeric vector variable*$\rangle$ $\langle$*save optional*$\rangle$
- **normalize** $\langle$*path variable*$\rangle$ $\langle$*options*$\rangle$

In order for this to work, the 3DLDF path must first be put into the x-y plane. Then, a corresponding METAPOST path is created, the desired operations are performed on it, and the generated data is returned. In the case of points, they must be transformed by the inverse of the transformation that placed the path into the x-y plane.

```
circle c[];
transform t[];
c0 := unit_circle scaled (1cm, 0, 1cm)
   rotated (90, 0);
t0 := (identity rotated (0, 60, 20))
   shifted (1.9cm, 1cm);
c0 *= t0;
c1 := c0 normalized save;
draw c0;
draw c1;
```

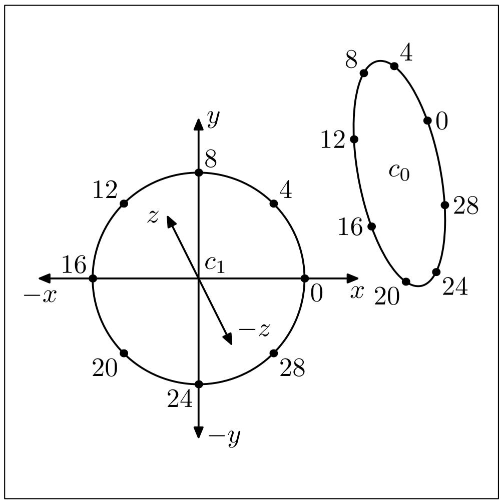

Fig. 13.

```
ellipse e[];
transform t[];
e0 := unit_ellipse scaled (1.75cm, 0, 1.25cm)
   rotated (90, 0);
t0 := (identity rotated (0, 60, 20))
   shifted (1.9cm, 1cm);
e0 *= t0;
e1 := e0 normalized save;
draw e0;
draw e1 with_color dark_gray;
```

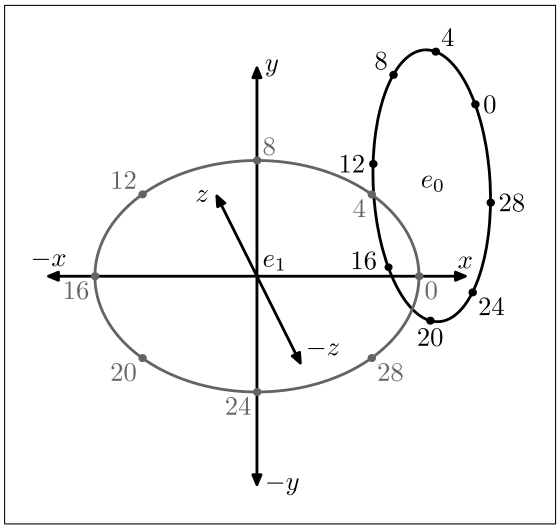

Fig. 14.

Further information and getting help

This article is merely intended to provide an introduction to GNU 3DLDF and to make it possible for the reader to make a start on using it, if desired. 3DLDF is a large and complex program and to cover its use and the ideas behind it comprehensively would require considerably more space. Many topics touched upon here could be enlarged upon in subsequent articles.

Unless one already knows METAFONT or METAPOST, the best way to learn 3DLDF is to start with *METAPOST, A User's Manual* [4]. For a deeper understanding of METAFONT, METAPOST and 3DLDF, *The METAFONTbook* is indispensable. After this, for learning 3DLDF itself, the GNU 3DLDF website provides further information and many examples of code and the generated graphics: `https://www.gnu.org/software/3dldf`.

The mailing list `help-3dldf@gnu.org` is available for users for discussion or to ask for help. See `https://www.gnu.org/software/3dldf/#Mailing_lists` for instructions on how to subscribe.

Unfortunately, the last edition of *The 3DLDF User and Reference Manual* (`https://www.gnu.org/software/3dldf/#Documentation`) is from 2006 and sadly out-of-date. At that time, 3DLDF was not yet interactive, as I had not yet implemented the scanning and parsing routines. It was more of a software library and drawings had to be created by writing C++ code to be linked to the latter. While it provides a good starting point for anyone who wants to know how 3DLDF works internally, it, of course, doesn't document the many features I've added since 2006.

Links

The GNU 3DLDF website:
`https://www.gnu.org/software/3dldf`

The author's personal website:
`https://laurence-finston.de`

METAPOST on the Web:
`https://tug.org/metapost.html`

CTAN, METAFONT package page:
`https://www.ctan.org/pkg/metafont`

Bibliography

[1] Cundy, H. Martyn and A.P. Rollet. *Mathematical Models.* Oxford: Oxford University Press, 1961.

[2] Gardner, Martin. *Mathematical Carnival.* New York: The Mathematical Association of America, 1989.

[3] Gonçalves, L. N. *FEATPOST and a Review of 3D METAPOST Packages.* In *TEX, XML, and Digital Typography. TUG 2004.* Lecture Notes in Computer Science, vol. 3130, p. 112ff. Berlin, Heidelberg: Springer, 2004. `https://doi.org/10.1007/978-3-540-27773-6_8`

[4] Hobby, John D. and the MetaPost development team. *METAPOST, A User's Manual.* 2020. `https://tug.org/metapost`

[5] Jones, Huw. *Computer Graphics through Key Mathematics.* London, UK: Springer-Verlag Limited, 2001.

[6] Knuth, Donald E. *The Art of Computer Programming.* Boston: Addison Wesley Publishing Company, 2019.

[7] Knuth, Donald E. *The METAFONTbook.* Reading, Massachusetts: Addison Wesley Publishing Company, 1986.

[8] Knuth, Donald E. *The TEXbook.* Reading, Massachusetts: Addison Wesley Publishing Company, 1986.

[9] Knuth, Donald E. *METAFONT: The Program. Computers & Typesetting*, vol. D. Reading, Massachusetts: Addison Wesley Publishing Company, 1986.

[10] Ramsey, Norman. *A Simple Solver for Linear Equations Containing Nonlinear Operators.* `https://www.cs.tufts.edu/~nr/noweb/examples/solver.html`

[11] Wikipedia. *Non-uniform rational B-spline.* `https://en.wikipedia.org/wiki/Non-uniform_rational_B-spline`

◇ Laurence Finston
Germany
`Laurence.Finston@gmx.de`