3DLDF User and Reference Manual

3-dimensional drawing with MetaPost output.
Manual edition 1.1.5.1 for 3DLDF Version 1.1.5.1
January 2004

Laurence D. Finston

This is the 3DLDF User and Reference Manual, edition 1.1.5.1 for 3DLDF 1.1.5.1.
This manual was last updated on 16 January 2004.
3DLDF is a GNU package for three-dimensional drawing with MetaPost output. The author is Laurence D. Finston.

Short Contents

Table of Contents

1 Introduction

3DLDF is a free software package for three-dimensional drawing written by Laurence D. Finston, who is also the author of this manual. It is written in C++ using CWEB and it outputs MetaPost code.

3DLDF is a GNU package. It is part of the GNU Project of the Free Software Foundation and is published under the GNU General Public License. See the website `http://www.gnu.org` for more information. 3DLDF is available for downloading from `http://ftp.gnu.org/gnu/3dldf`. The official 3DLDF website is `http://www.gnu.org/software/3dldf`. More information about 3DLDF can be found at the author's website: `http://wwwuser.gwdg.de/~lfinsto1`.

Please send bug reports to:

`bug-3DLDF@gnu.org` and

Two other mailing lists may be of interest to users of 3DLDF: `help-3DLDF@gnu.org` is for people to ask other users for help and `info-3DLDF@gnu.org` is for sending announcements to users. To subscribe, send an email to the appropriate mailing list or lists with the word "subscribe" as the subject. The author's website is `http://wwwuser.gwdg.de/~lfinsto1`.

My primary purpose in writing 3DLDF was to make it possible to use MetaPost for three-dimensional drawing. I've always enjoyed using MetaPost, and thought it was a shame that I could only use it for making two-dimensional drawings. 3DLDF is a front-end that operates on three-dimensional data, performs the necessary calculations for the projection onto two dimensions, and writes its output in the form of MetaPost code.

While 3DLDF's data types and operations are modelled on those of Metafont and MetaPost, and while the only form of output 3DLDF currently produces is MetaPost code, it is nonetheless not in principle tied to MetaPost. It could be modified to produce PostScript code directly, or output in other formats. It would also be possible to modify 3DLDF so that it could be used for creating graphics interactively on a terminal, by means of an appropriate interface to the computer's graphics hardware.

The name "3DLDF" ("3D" plus the author's initials) was chosen because, while not pretty, it's unlikely to conflict with any of the other programs called "3D"-something.

1.1 Sources of Information

This handbook, and the use of 3DLDF itself, presuppose at least some familiarity on the part of the reader with Metafont, MetaPost, CWEB, and C++. If you are not familiar with any or all of them, I recommend the following sources of information:

Knuth, Donald Ervin. *The METAFONTbook*. Computers and Typesetting; C. Addison Wesley Publishing Company, Inc. Reading, Massachusetts 1986.

Hobby, John D. *A User's Manual for MetaPost*. AT & T Bell Laboratories. Murray Hill, NJ. No date.

Knuth, Donald E. and Silvio Levy. *The* **CWEB** *System of Structured Documentation*. Version 3.64—February 2002.

Stroustrup, Bjarne. *The* C++ *Programming Language*. Special Edition. Reading, Massachusetts 2000. Addison-Wesley. ISBN 0-201-70073-5.

The manuals for MetaPost and CWEB are available from the Comprehensive TEX Archive Network (CTAN). See one of the following web sites for more information:

Germany `http://dante.ctan.org`, `http://ftp.dante.de`
`http://www.dante.de`.

United Kingdom
`http://www.cam.ctan.org`
`http://ftp.tex.ac.uk`.

USA `http://www.tug.ctan.org`
`http://www.ctan.tug.org`.

1.2 About This Manual

This manual has been created using Texinfo, a documentation system which is part of the GNU Project, whose main sponsor is the Free Software Foundation. Texinfo can be used to generate online and printed documentation from the same input files.

For more information about Texinfo, see:

Stallmann, Richard M. and Robert J. Chassell. *Texinfo. The GNU Documentation Format.* The Free Software Foundation. Boston 1999.

For more information about the GNU Project and the Free Software Foundation, see the following web site: `http://www.gnu.org`.

The edition of this manual is 1.1.5.1 and it documents version 1.1.5.1 of 3DLDF. The edition number of the manual and the version number of the program are the same (as of 16 January 2004), but may diverge at a later date.

Note that "I", "me", etc., in this manual refers to Laurence D. Finston, so far the sole author of both 3DLDF and this manual. "Currently" and similar formulations refer to version 1.1.5.1 of 3DLDF as of 16 January 2004.

This manual is intended for both beginning and advanced users of 3DLDF. So, if there's something you don't understand, it's probably best to skip it and come back to it later. Some of the more difficult points, or ones that presuppose familiarity with features not yet described, are in the footnotes.

I firmly believe that an adequate program with good documentation is more useful than a great program with poor or no documentation. The ideal case, of course, is a great program with great documentation. I'm sorry to say, that this manual is not yet as good as I'd like it to be. I apologize for the number of typos and other errors. I hope they don't detract too much from its usefulness. I would have liked to have proofread and corrected it again before publication, but for reasons external to 3DLDF, it is necessary for me to publish now. I plan to set up an errata list on the official 3DLDF website (`http://www.gnu.org/software/3dldf`), and/or my own website (`http://wwwuser.gwdg.de/~lfinsto1`).

Unless I've left anything out by mistake, this manual documents all of the data types, constants and variables, namespaces, and functions defined in 3DLDF. However, some of the descriptions are terser than I would like, and I'd like to have more examples and illustrations. There is also more to be said on a number of topics touched on in this manual, and some topics I haven't touched on at all. In general, while I've tried to give complete information

on the "what and how", the "why and wherefore" has sometimes gotten short shrift. I hope to correct these defects in future editions.

1.2.1 Conventions

Data types are formatted like this: `int`, `Point`, `Path`. Plurals are formatted in the same way: `ints`, `Points`, `Paths`. It is poor typographical practice to typeset a single word using more than one font, e.g., `int`s, `Point`s, `Path`s. This applies to data types whose plurals do not end in "s" as well, e.g., the plural of the C++ `class Polyhedron` is `Polyhedra`.

When C++ functions are discussed in this manual, I always include a pair of parentheses to make it clear that the item in question is a function and not a variable, but I generally do not include the arguments. For example, if I mention the function `foo()`, this doesn't imply that `foo()` takes no arguments. If it were appropriate, I would include the argument type:

```
foo(int)
```

or the argument type and a placeholder name:

```
foo(int arg)
```

or I would write

```
foo(void)
```

to indicate that `foo()` takes no arguments. Also, I generally don't indicate the return type, unless it is relevant. If it is a member function of a class, I may indicate this, e.g.,, `bar_class::foo()`, or not, depending on whether this information is relevant. This convention differs from that used in the ⟨undefined⟩ [Function Index], page ⟨undefined⟩, which is generated automatically by Texinfo. There, only the name of the function appears, without parentheses, parameters, or return values. The class type of member functions may appear in the Function Index, (e.g., `bar_class::foo`), but only in index entries that have been entered explicitly by the author; such entries are not generated by Texinfo automatically.

Examples are formatted as follows:

```
Point p0(1, 2, 3);
Point p1(5, 6, 7.9);
Path pa(p0, p1);
p0.show("p0:");
 ⊣ p0: (1, 2, 3)
```

Examples can contain the following symbols:

⊣	Indicates output to the terminal when 3DLDF is run.
⇒	Indicates a result of some sort. It may precede a illustration generated by the code in the example.
error	Indicates that the following text is an error message.

This manual does not use all of the symbols provided by Texinfo. If you find a symbol you don't understand in this manual (which shouldn't happen), see page 103 of the Texinfo manual.

Symbols:

$\mathbb{N}$ The set of the natural numbers $\{0, 1, 2, 3, 4, \ldots\}$.

$\mathbb{Z}$ The set of the integers $\{\ldots, -3, -2, -1, 0, 1, 2, 3, 4, \ldots\}$.

$\mathbb{R}$ The set of the real numbers.

1.2.2 Illustrations

The illustrations in this manual have been created using 3DLDF. The code that generates them is in the Texinfo files themselves, that contain the text of the manual. Texinfo is based on TEX, so it's possible to make use of the latter's facility for writing ASCII text to files using TEX's `\write` command.

The file '`3DLDF-1.1.5.1/CWEB/exampman.web`' contains the C++ code, and the file '`3DLDF-1.1.5.1/CWEB/examples.mp`' contains the MetaPost code for generating the illustrations. 3DLDF was built using GCC 2.95 when the illustrations were generated. For some reason, GCC 3.3 has difficulty with them. It works to generate them in batches of about 50 with GCC 3.3.

MetaPost outputs Encapsulated PostScript files. These can be included in TEX files, as explained below. However, in order to display the illustrations in the HTML version of this manual, I had to convert them to PNG ("Portable Network Graphics") format (`http://www.libpng.org/pub/png/index.html`). See ⟨undefined⟩ [Converting EPS Files], page ⟨undefined⟩, for instructions on how to do this.

Please note that the illustrations cannot be shown in the Info output format!

If you have problems including the illustrations in the printed version, for example, if your installation doesn't have `dvips`, look for the following lines in '`3DLDF.texi`':

```
\doepsftrue    %% One of these two lines should be commented-out.
%\doepsffalse
```

Now, remove the '`%`' from in front of '`\doepsffalse`' and put one in front of '`\doepsftrue`'. This will prevent the illustrations from being included. This should only be done as a last resort, however, because it will make it difficult if not impossible to understand this manual.

The C++ code in an example is not always the complete code used to create the illustration that follows it, since the latter may be cluttered with commands that would detract from the clarity of the example. The actual code used always follows the example in the Texinfo source file, so the latter may be referred to, if the reader wishes to see exactly what code was used to generate the illustration.

You may want to skip the following paragraphs in this section, if you're reading this manual for the first time. Don't worry if you don't understand it, it's meaning should become clear after reading the manual and some experience with using 3DLDF.

The file '`3DLDF.texi`' in the directory '`3DLDF-1.1.5.1/DOC/TEXINFO`', the driver file for this manual, contains the following TEX code:

```
\newif\ifmakeexamples
\makeexamplestrue     %% One of these two lines should be commented-out.
%\makeexamplesfalse
```

When `texi2dvi` is run on '`3DLDF.texi`', `\makeexamplestrue` is not commented-out, and `\makeexamplesfalse` is, the C++ code for the illustrations is written to the file '`examples.web`'. If the EPS files don't already exist (in the directory

'3DLDF-1.1.5.1/DOC/TEXINFO/EPS'), the TEX macro \PEX, which includes them in the Texinfo files, will signal an error each time it can't find one. Just type 's' at the command line to tell TEX to keep going. If you want to be sure that these are indeed the only errors, you can type '<RETURN>' after each one instead.

texi2dvi 3DLDF.texi also generates the file 'extext.tex', which contains TEX code for including the illustrations by themselves.

'examples.web' must now be moved to '3DLDF-1.1.5.1/CWEB/' and ctangled, 'examples.c' must compiled, and 3DLDF must be relinked. ctangle examples also generates the header file 'example.h', which is included in 'main.web'. Therefore, if the contents of 'examples.h' have changed since the last time 'main.web' was ctangled, 'main.web' will have to be ctangled, and 'main.c' recompiled, before '3dldf' is relinked.[1]

Running 3dldf and MetaPost now generates the EPS (Encapsulated PostScript) files '3DLDFmp.1' through (currently) '3DLDFmp.199' for the illustrations. They must be moved to '3DLDF-1.1.5.1/DOC/TEXINFO/EPS'. Now, when texi2dvi 3DLDF.texi is run again, the dvips command '\epsffile' includes the EPS files for the illustrations in the manual. '3DLDF.texi' includes the line '\input epsf', so that '\epsffile' works. Of course, dvips (or some other program that does the job) must be used to convert '3DLDF.dvi' to a PostScript file. To see exactly how this is done, take a look at the '.texi' source files of this manual.[2]

In the '3DLDF.texi' belonging to the 3DLDF distribution, \makeexamplestrue will be commented-out, and makeexamplesfalse won't be, because the EPS files for the illustrations are included in the distribution.

The version of 'examples.web' in '3DLDF-1.1.5.1/CWEB' merely includes the files 'subex1.web' and 'subex2.web'. If you rename '3DLDF-1.1.5.1/CWEB/exampman.web' to 'examples.web', you can generate the illustrations.

1.3 CWEB Documentation

As mentioned above, 3DLDF has been programmed using CWEB, which is a "literate programming" tool developed by Donald E. Knuth and Silvio Levy. See ⟨undefined⟩ [Sources of Information], page ⟨undefined⟩, for a reference to the CWEB manual. Knuth's *TEX—The Program* and *Metafont—The Program* both include a section "How to read a WEB" (pp. x–xv, in both volumes).

CWEB files combine source code and documentation. Running ctangle on a CWEB file, for example, 'main.web', produces the file 'main.c' containing C or C++ code. Running cweave main.web creates a TEX file with pretty-printed source code and nicely formatted documentation. I find that using CWEB makes it more natural to document my code as I write it, and makes the source files easier to read when editing them. It does have certain consequences with regard to compilation, but these are taken care of by make. See ⟨unde-

[1] ctangle creates '<*filename*>.c' from '<*filename*>.web', so the compiler must compile the C++ files using the '-x c++' option. Otherwise, it would handle them as if they contained C code.

[2] If you want to try generating the illustrations yourself, you can save a little run-time by calling tex 3DLDF.texi the first time, rather than texi2dvi. The latter program runs TEX twice, because it needs two passes in order to generate the contents, indexing, and cross reference information (and maybe some other things, too).

fined⟩ [Adding a File], page ⟨undefined⟩, and ⟨undefined⟩ [Changes], page ⟨undefined⟩, for more information.

The CWEB files in the directory '`3DLDF-1.1.5.1/CWEB/`' contain the source code for 3DLDF. The file '`3DLDFprg.web`' in this directory is only ever used for cweaving; it is never ctangled and contains no C++ code for compilation. It does, however, include all of the other CWEB files, so that `cweave 3DLDFprg.web` generates the TEX file containing the complete documentation of the source code of 3DLDF.

The files '`3DLDF-1.1.5.1/CWEB/3DLDFprg.tex`', '`3DLDF-1.1.5.1/CWEB/3DLDFprg.dvi`', and '`3DLDF-1.1.5.1/CWEB/3DLDFprg.ps`' are included in the distribution of 3DLDF as a convenience. However, users may generate them themselves, should there be some reason for doing so, by entering `make ps` from the command line of a shell from the working directory '`3DLDF-1.1.5.1/`' or '`3DLDF-1.1.5.1/CWEB`'. Alternatively, the user may generate them by hand from the working directory '`3DLDF-1.1.5.1/CWEB/`' in the following way:

1. `cweave 3DLDFprg.web` generates '`3DLDFprg.tex`'.
2. `tex 3DLDFprg` or `tex 3DLDFprg.tex` generates '`3DLDFprg.dvi`'.
3. `dvips -o 3DLDFprg.ps 3DLDFprg` (possibly with additional options) generates '`3DLDFprg.ps`'.
4. `lpr -P<`*`print queue`*`> 3DLDFprg.ps` sends '`3DLDFprg.ps`' to a printer, on a UNIX or UNIX-like system.

The individual commands may differ, depending on the system you're using.

1.4 Metafont and MetaPost

Metafont is a system created by Donald E. Knuth for generating fonts, in particular for use with TEX, his well-known typsetting system.[3] Expressed in a somewhat simplified way, Metafont is a system for programming curves, which are then digitized and output in the form of run-time encoded bitmaps. (See Knuth's *The Metafontbook* for more information).

John D. Hobby modified Metafont's source code to create MetaPost, which functions in much the same way, but outputs encapsulated PostScript (EPS) files instead of bitmaps. MetaPost is very useful for creating graphics and is a convenient interface to PostScript. It is also easy both to imbed TEX code in MetaPost programs, for instance, for typesetting labels, and to include MetaPost graphics in ordinary TEX files, e.g., by using `dvips`.[4] Apart from simply printing the PostScript file output by `dvips`, there are many programs that can process ordinary or encapsulated PostScript files and convert them to other formats. Just two of the many possibilities are ImageMagick and GIMP, both of which can be used to create animations from MetaPost graphics.

However, MetaPost inherited a significant limitation from Metafont: it's not possible to use it for making three-dimensional graphics, except in a very limited way. One insuperable problem is the severe limitation on the magnitude of user-defined numerical variables in

[3] Knuth, Donald E. *The TEXbook*. Computers and Typesetting; A. Addison-Wesley Publishing Company. Reading, Massachusetts 1986.

[4] Rokicki, Tomas. *Dvips: A DVI-to-PostScript Translator* February 1997. Available from CTAN. See ⟨undefined⟩ [Sources of Information], page ⟨undefined⟩.

Metafont and MetaPost.[5] This made sense for Metafont's and MetaPost's original purposes, but they make it impossible to perform the calculations needed for 3D graphics.

Another problem is the data types defined in Metafont: Points are represented as pairs of real values and affine transformations as sets of 6 real values. This corresponds to the representation of points and affine transformations in the plane as a two-element vector on the one hand and a six element matrix on the other. While it is possible to work around the limitation imposed by having points be represented by only two values, it is impracticable in the case of the transformations.

For these reasons, I decided to write a program that would behave more or less like Metafont, but with suitable extensions, and the ability to handle three dimensional data; namely 3DLDF. It stores the data and performs the transformations and other necessary calculations and is not subject to the limitations of MetaPost and its data types. Upon output, it performs a perspective transformation, converting the 3D image into a 2D one. The latter can now be expressed as an ordinary MetaPost program, so 3DLDF writes its output as MetaPost code to a file.

In the following, it may be a little unclear why I sometimes refer to Metafont and sometimes to MetaPost. The reason is that Metafont inherited much of its functionality from Metafont. Certain operations in Metafont have no meaning in MetaPost and so have been removed, while MetaPost's function of interfacing with PostScript has caused other operations to be added. For example, in MetaPost, `color` is a data type, but not in Metafont. Unless otherwise stated, when I refer to Metafont, it can be assumed that what I say applies to MetaPost as well. However, when I refer to MetaPost, it will generally be in connection with features specific to MetaPost.

1.5 Caveats

1.5.1 Accuracy

When 3DLDF is run, it uses the three-dimensional data contained in the user code to create a two-dimensional projection. Currently, this can be a perspective projection, or a parallel projection onto one of the major planes. MetaPost code representing this projection is then written to the output file. 3DLDF does no *scan conversion*,[6] so all of the curves in the projection are generated by means of the algorithms MetaPost inherited from Metafont. These algorithms, however, are designed to find the "most pleasing curve"[7] given one or more two-dimensional points and connectors; they do not account for the the fact that the two-dimensional points are projections of three-dimensional ones. This can lead to unsatisfactory results, especially where extreme foreshortening occurs. In particular, '`curl`', `dir`, '`tension`', and control points should be used cautiously, or avoided altogether, when specifying connectors.

---

[5] "[...] METAFONT deals only with numbers in a limited range: A numeric token must be less than 4096, and its value is always rounded to the nearest multiple of $\frac{1}{65536}$." Knuth, *The METAFONTbook*, p. 50.

[6] *Scan conversion* is the process of digitizing geometric data. The ultimate result is a 2×2 map of pixels, which can be used for printing or representing the projection on a computer screen. The number of pixels per a given unit of measurement is the *resolution* of a given output device, e.g., 300 pixels per inch.

[7] Knuth, *The METAFONTbook*, Chapter 14, p. 127.

3DLDF operates on the assumption that, given an adequate number of points, MetaPost will produce an adequate approximation to the desired curve *in perspective*, since the greater the number of points given for a curve, the less "choice" MetaPost has for the path through them. My experience with 3DLDF bears this out. Generally, the curves look quite good. Where problems arise, it usually helps to increase the number of points in a curve.

A more serious problem is the imprecision resulting from the operation of rotation. Rotations use the trigonometric functions, which return approximate values. This has the result that points that should have identical coordinate values, sometimes do not. This has consequences for the functions that compare points. The more rotations are applied to points, the greater the divergence between their actual coordinate values, and the values they should have. So far, I haven't found a solution for this problem. On the other hand, it hasn't yet affected the usability of 3DLDF.

1.5.2 No Input Routine

3DLDF does not yet include a routine for reading input files. This means that user code must be written in C++, compiled, and linked with the rest of the program. I admit, this is not ideal, and writing an input routine for user code is one of the next things I plan to add to 3DLDF.

I plan to use Flex and Bison to write the input routine.[8] The syntax of the input code should be as close as possible to that of MetaPost, while taking account of the differences between MetaPost and 3DLDF.

For the present, however, the use of 3DLDF is limited to those who feel comfortable using C++ and compiling and relinking programs. Please don't be put off by this! It's not so difficult, and `make` does most of the work of recompiling and running 3DLDF. See ⟨undefined⟩ [Installing and Running 3DLDF], page ⟨undefined⟩, for more information.

1.6 Ports

I originally developed 3DLDF on a DECalpha Personal Workstation with two processors running under the operating system Tru64 Unix 5.1, using the DEC C++ compiler. I then ported it to a PC Pentium 4 running under Linux 2.4, using the GNU C++ compiler GCC 2.95.3, and a PC Pentium II XEON under Linux 2.4, using GCC 3.3. I am currently only maintaining the last version. I do not believe that it's worthwhile to maintain a version for GCC 2.95. While I would like 3DLDF to run on as many platforms as possible, I would rather spend my time developing it than porting it. This is something where I would be grateful for help from other programmers.

Although I am no longer supporting ports to other systems, I have left some conditionally compiled code for managing platform dependencies in the CWEB sources of 3DLDF. This may make it easier for other people who want to port 3DLDF to other platforms.

Currently, the files '`io.web`', '`loader.web`', '`main.web`', '`points.web`', and '`pspglb.web`' contain conditionally compiled code, depending on which compiler, or in the case of GCC, which version of the compiler, is used. The DEC C++ compiler defines the preprocessor macro '`__DECCXX`' and GCC defines '`__GNUC__`'. In order to distinguish between GCC 2.95.3

[8] Flex is a program for generating text scanners and Bison is a parser generator. They are available from `http://www.gnu.org`.

and GCC 3.3, I've added the macros '`LDF_GCC_2_95`' and '`LDF_GCC_3_3`' in '`loader.web`', which should be defined or undefined, depending on which compiler you're using. In the distribution, '`LDF_GCC_3_3`' is defined and '`LDF_GCC_2_95`' is undefined, so if you want to try using GCC 2.95, you'll have to change this (it's not guaranteed to work).

3DLDF 1.1.5.1 now uses Autoconf and Automake, and the '`configure`' script generates a '`config.h`' file, which is now included in '`loader.web`'. Some of the preprocessor macros defined in '`config.h`' are used to conditionally include library header files, but so far, there is no error handling code for the case that a file can't be included. I hope to improve the way 3DLDF works together with Autoconf and Automake in the near future.

3DLDF 1.1.5 is the first release that contains template functions. Template instantiation differs from compiler to compiler, so using template functions will tend to make 3DLDF less portable. See ⟨undefined⟩ [Template Functions], page ⟨undefined⟩, for more information. I am no longer able to build 3DLDF on the DECalpha Personal Workstation. I'm fairly sure that it would be possible to port it, but I don't plan to do this, since Tru64 Unix 5.1 and the DEC C++ compiler are non-free software.

1.7 Contributing to 3DLDF

So far, I've been the sole author and user of 3DLDF. I would be very interested in having other programmers contribute to it. I would be particularly interested in help in making 3DLDF conform as closely as possible to the GNU Coding Standards. I would be grateful if someone would write proper Automake and Autoconf files, since I haven't yet learned how to do so (I'm working on it).

See ⟨undefined⟩ [Introduction], page ⟨undefined⟩, for information on how to contact the author.

Using 3DLDF

Since 3DLDF does not yet have an input routine, user code must be written in C++ (in '`main.web`', or some other file) and compiled. Then, 3DLDF must be relinked, together with the new file of object code resulting from the compilation. For now, the important point is that the text of the examples in this manual represent C++ code. See ⟨undefined⟩ [Installing and Running 3DLDF], page ⟨undefined⟩, for more information.

2 Points

2.1 Declaring and Initializing Points

The most basic drawable object in 3DLDF is `class Point`. It is analogous to `pair` in Metafont. For example, in Metafont one can define a `pair` using the "z" syntax as follows:

```
z0 = (1cm, 1cm);
```

There are other ways of defining `pairs` in Metafont (and MetaPost), but this is the usual way.

In 3DLDF, a Point is declared and initialized as follows:

```
Point pt0(1, 2, 3);
```

This simple example demonstrates several differences between Metafont and 3DLDF. First of all, there is no analog in 3DLDF to Metafont's "z" syntax. If I want to have `Points` called "`pt0`", "`pt1`", "`pt2`", etc., then I must declare each of them to be a `Point`:

```
Point pt0(10, 15, 2);
Point pt1(13, 41, 5.5);
Point pt2(62.9, 7.02, 8);
```

Alternatively, I could declare an array of `Points`:

```
Point pt[3];
```

Now I can refer to `pt[0]`, `pt[1]`, and `pt[2]`.

In the Metafont example, the x and y-coordinates of the `pair z0` are specified using the unit of measurement, in this case, centimeters. This is currently not possible in 3DLDF. The current unit of measurement is stored in the static variable `Point::measurement_units`, which is a `string`. Its default value is `"cm"` for "centimeters". At present, it is best to stick with one unit of measurement for a drawing. After I've defined an input routine, 3DLDF should handle units of measurement in the same way that Metafont does.

Another difference is that the `Points pt0`, `pt1`, and `pt2` have three coordinates, x, y, and z, whereas `z0` has only two, x and y. Actually, the difference goes deeper than this. In Metafont, a `pair` has two parts, `xpart` and `ypart`, which can be examined by the user. In 3DLDF, a `Point` contains the following sets of coordinates:

```
world_coordinates
user_coordinates
view_coordinates
projective_coordinates
```

These are sets of 3-dimensional *homogeneous* coordinates, which means that they contain four coordinates: x, y, z, and w. Homogeneous coordinates are used in the affine and perspective transformations (see ⟨undefined⟩ [Transforms], page ⟨undefined⟩).

Currently, only `world_coordinates` and `projective_coordinates` are used in 3DLDF. The `world_coordinates` refer to the position of a `Point` in 3DLDF's basic, unchanging coordinate system. The `projective_coordinates` are the coordinates of the two-dimensional projection of the `Point` onto a plane. This projection is what is ultimately printed out or displayed on the computer screen. Please note, that when the coordinates of a `Point` are referred to in this manual, the `world_coordinates` are meant, unless otherwise stated.

`Points` can be declared and their values can be set in different ways.

```
Point pt0;
Point pt1(1);
Point pt2(2.3, 52);
Point pt3(4.5, 7, 13.205);
```

`pt0` is declared without any arguments, i.e., using the default constructor, so the values of its x, y, and z-coordinates are all 0.

`pt1` is declared and initialized with one argument for the x-coordinate, so its y and z-coordinates are initialized with the values of `CURR_Y` and `CURR_Z` respectively. The latter are static constant data members of class `Point`, whose values are 0 by default. They can be reset by the user, who should make sure that they have sensible values.

`pt2` is declared and initialized with two arguments for its x and y-coordinates, so its z-coordinate is initialized to the value of `CURR_Z`. Finally, `pt3` has an argument for each of its coordinates.

Please note that `pt0` is constructed using a the default constructor, whereas the other `Points` are constructed using a constructor with one required argument (for the x-coordinate), and two optional arguments (for the y and z-coordinates). The default constructor always sets all the coordinates to 0, irrespective of the values of `CURR_Y` and `CURR_Z`.

2.2 Setting and Assigning to Points

It is possible to change the value of the coordinates of `Points` by using the *assignment operator* `=` (`Point::operator=()`) or the function `Point::set()` (with appropriate arguments):

```
Point pt0(2, 3.3, 7);
Point pt1;
pt1 = pt0;
pt0.set(34, 99, 107.5);
pt0.show("pt0:");
⊣ pt0: (34, 99, 107.5)
pt1.show("pt1:");
⊣ pt1: (2, 3.3, 7)
```

In this example, `pt0` is initialized with the coordinates `(2, 3.3, 7)`, and `pt1` with the coordinates `(0, 0, 0)`. `pt1 = pt0` causes `pt1` to have the same coordinates as `pt0`, then the coordinates of `pt0` are changed to `(34, 99, 107.5)`. This doesn't affect `pt1`, whose coordinates remain `(2, 3.3, 7)`.

Another way of declaring and initializing `Points` is by using the *copy constructor*:

```
Point pt0(1, 3.5, 19);
Point pt1(pt0);
Point pt2 = pt0;
Point pt3;
pt3 = pt0;
```

In this example, `pt1` and `pt2` are both declared and initialized using the copy constructor; `Point pt2 = pt0` does not invoke the assignment operator. `pt3`, on the other hand, is declared using the default constructor, and not initialized. In the following line, `pt3 = pt0`

does invoke the assignment operator, thus resetting the coordinate values of `pt3` to those of `pt0`.

3 Transforming Points

`Points` don't always have to remain in the same place. There are various ways of moving or *transforming* them:

- Shifting. This is often called "translating", but the operation in Metafont that performs translation is called `shift`, so I call it "shifting".
- Scaling.
- Shearing.
- Rotating about an axis.

`class Point` has several member functions for applying these *affine transformations*[1] to a `Point`. Most of the arguments to these functions are of type `real`. As you may know, there is no such data type in C++. I have defined `real` using `typedef` to be either `float` or `double`, depending on the value of a preprocessor switch for conditional compilation.[2] 3DLDF uses many `real` values and I wanted to be able to change the precision used by making one change (in the file '`pspglb.web`') rather than having to examine all the places in the program where `float` or `double` are used. Unfortunately, setting `real` to `double` currently doesn't work.

3.1 Shifting

The function `shift()` adds its arguments to the corresponding `world_coordinates` of a `Point`. In the following example, the function `show()` is used to print the `world_coordinates` of `p0` to standard output.

```
Point p0(0, 0, 0);
p0.shift(1, 2, 3);
p0.show("p0:");
 ⊣ p0: (1, 2, 3)
p0.shift(10);
p0.show("p0:");
 ⊣ p0: (11, 2, 3)
p0.shift(0, 20);
p0.show("p0:");
 ⊣ p0: (11, 22, 3)
p0.shift(0, 0, 30);
p0.show("p0:");
 ⊣ p0: (11, 22, 33)
```

`shift` takes three `real` arguments, whereby the second and third are optional. To shift a `Point` in the direction of the positive or negative y-axis, and/or the positive or negative

[1] *Affine transformations* are operations that have the property that parallelity of lines is maintained. That is, if two lines (each determined by two points) are parallel before the transformation, they will also be parallel after the transformation. Affine transformations are discussed in many books about computer graphics and geometry. For 3DLDF, I've mostly used Jones, *Computer Graphics through Key Mathematics* and Salomon, *Computer Graphics and Geometric Modeling*.

[2] I try to avoid the use of preprocessor macros as much as possible, for the reasons given by Stroustrup in the *The* C++ *Programming Language*, §7.8, pp. 160–163, and *Design and Evolution of* C++, Chapter 18, pp. 423–426. However, conditional compilation is one of the tasks that only the preprocessor can perform.

z-axis only, then a 0 argument for the x direction, and possibly one for the y direction must be used as placeholders, as in the example above.

`shift()` can be invoked with a `Point` argument instead of `real` arguments. In this case, the x, y, and z-coordinates of the argument are used for shifting the `Point`:

```
Point a(10, 10, 10);
Point b(1, 2, 3);
a.shift(b);
a.show("a:")
 ⊣ a: (11, 12, 13)
```

Another way of shifting `Points` is to use the binary `+=` operator (`Point::operator+=()`) with a `Point` argument.

```
Point a0(1, 1, 1);
Point a1(2, 2, 2);
a0 += a1;
a0.show("a0:");
 ⊣ a0: (3, 3, 3)
```

3.2 Scaling

The function `scale()` takes three `real` arguments. The x, y, and z-coordinates of the `Point` are multiplied by the first, second, and third arguments respectively. Only the first argument is required; the default for the others is 1.

If one wants to perform scaling in either the y-dimension only, or the y and z-dimensions only, a dummy argument of 1 must be passed for scaling in the x-dimension. Similarly, if one wants to perform scaling in the z-dimension only, dummy arguments of 1 must be passed for scaling in the x and y-dimensions.

```
Point p0(1, 2, 3);
p0.scale(2, 3, 4);
p0.show("p0:");
 ⊣ p0: (2, 6, 12)
p0.scale(2);
p0.show("p0:");
 ⊣ p0: (4, 6, 12)
p0.scale(1, 3);
p0.show("p0:");
 ⊣ p0: (4, 18, 12)
p0.scale(1, 1, 3);
p0.show("p0:");
 ⊣ p0: (4, 18, 36)
```

3.3 Shearing

Shearing is more complicated than shifting or scaling. The function `shear()` takes six `real` arguments. If p is a `Point`, then `p.shear(a, b, c, d, e, f)` sets x_p to $x_p + ay_p + bz_p$, y_p to $y_p + cx_p + dz_p$, and z_p to $z_p + ex_p + fy_p$. In this way, each coordinate of a `Point` is modified based on the values of the other two coordinates, whereby the influence of the other coordinates on the new value is weighted according to the arguments.