3DLDF User and Reference Manual

3-dimensional drawing with MetaPost output.
Manual edition 1.1.5.1 for 3DLDF Version 1.1.5.1
July 2007

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 18 July 2007. 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 18 July 2007), 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 18 July 2007.

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 [Function Index], page 291, 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 Section 11.2.1 [Converting EPS Files], page 76, 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 Section 1.1 [Sources of Information], page 1, 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).

---

[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).

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 Chapter 39 [Adding a File], page 272, and Chapter 41 [Changes], page 277, 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

---

[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 Section 1.1 [Sources of Information], page 1.

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 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

[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.

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.

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 Chapter 11 [Installing and Running 3DLDF], page 73, 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

---

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

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

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 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 Section 11.1.1 [Template Functions], page 73, 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 Chapter 1 [Introduction], page 1, 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 Chapter 11 [Installing and Running 3DLDF], page 73, 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 Chapter 4 [Transforms], page 18).

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)
```

---

[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.

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 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.

```
Point p(1, 1, 1);
p.shear(1);
p.show("p:");
⊣ p: (2, 1, 1)
p.set(1, 1, 1);
p.shear(1, 1);
p.show("p:");
⊣ p: (3, 1, 1)
p.set(1, 1, 1);
p.shear(1, 1, 2, 2, 3, 3);
p.show("p:");
⊣ p: (3, 5, 7)
```

Fig. 1 demonstrates the effect of shearing the points of a rectangle in the x-y plane.

```
Point P0;
Point P1(3);
Point P2(3, 3);
Point P3(0, 3);
Rectangle r(p0, p1, p2, p3);
r.draw();
Rectangle q(r);
q.shear(1.5);
q.draw(black, "evenly");
```

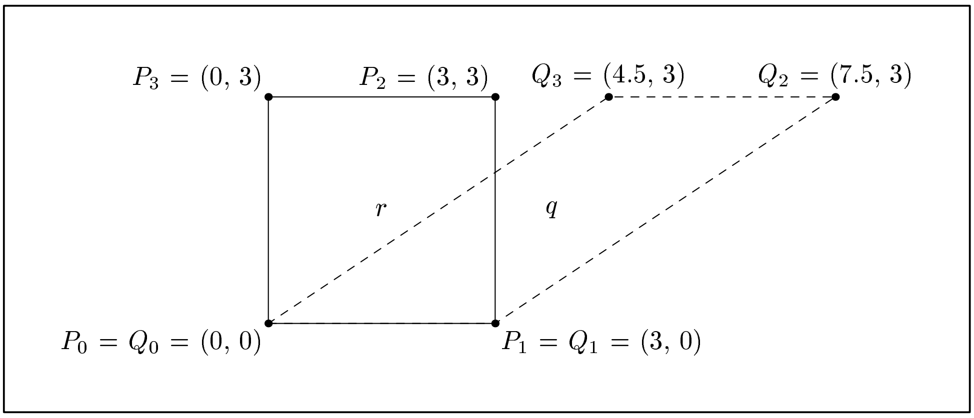


Figure 1.

3.4 Rotating

The function `rotate()` rotates a `Point` about one or more of the main axes. It takes three `real` arguments, specifying the angles of rotation in degrees about the x, y, and z-axes respectively. Only the first argument is required, the other two are 0 by default. If rotation about the y-axis, or the y and z-axes only are required, then 0 must be used as a placeholder for the first and possibly the second argument.

```
Point p(0, 1);
p.rotate(90);
p.show("p:");
⊣ p: (0, 0, -1)
p.rotate(0, 90);
p.show("p:");
⊣ p: (1, 0, 0)
p.rotate(0, 0, 90);
p.show("p:");
⊣ p: (0, 1, 0)
```

The rotations are performed successively about the x, y, and z-axes. However, rotation is not a commutative operation, so if rotation about the main axes in a different order is required, then `rotate()` must be invoked more than once:

```
Point A(2, 3, 4);
Point B(A);
A.rotate(30, 60, 90);
A.show("A:");
⊣ A: (-4.59808, -0.700962, 2.7141)
B.rotate(0, 0, 90);
B.rotate(0, 60);
B.rotate(30);
B.show("B:");
⊣ B: (-4.9641, 1.43301, -1.51795)
```

Rotation need not be about the main axes; it can also be performed about a line defined by two `Points`. The function `rotate()` with two `Point` arguments and a `real` argument for the angle of rotation (in degrees) about the axis. The `real` argument is optional, with 180° as the default.

```
Point p0 (-1.06066, 0, 1.06066);
Point p1 (1.06066, 0, -1.06066);
p1 *= p0.rotate(0, 30, 30);
p0.show("p0:");
⊣ p0: (-1.25477, -0.724444, 0.388228)
p1.show("p1:");
⊣ p1: (1.25477, 0.724444, -0.388228)
p0.draw(p1);
Point p2(1.06066, 0, 1.06066);
p2.show("p2:");
⊣ p2: (1.06066, 0, 1.06066)
```

```
Point p3(p2);
p3.rotate(p1, p0, 45);
p3.show("p3:");
⊣ p3 (1.09721, 1.15036, 1.17879)
Point p4(p2);
p4.rotate(p1, p0, 90);
p4.show("p4:");
⊣ p4: (0.882625, 2.05122, 0.485242)
Point p5(p2);
p5.rotate(p1, p0, 135);
p5.show("p5:");
⊣ p5: (0.542606, 2.17488, -0.613716)
Point p6(p2);
p6.rotate(p1, p0);
p6.show("p6:");
⊣ p6: (0.276332, 1.44889, -1.47433)
```

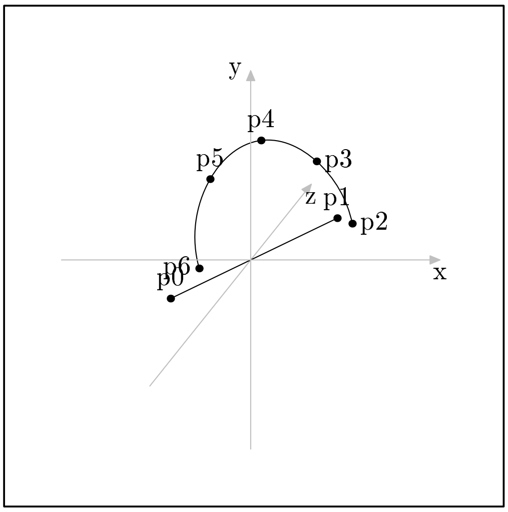


Figure 2.

I have sometimes gotten erroneous results using `rotate()` for rotation about two `Points`. It's usually worked to reverse the order of the `Point` arguments, or to change sign of the angle argument. I think I've fixed the problem, though.

4 Transforms

When `Points` are transformed using `shift()`, `shear()`, or one of the other transformation functions, the `world_coordinates` are not modified directly. Instead, another data member of `class Point` is used to store the information about the transformation, namely `transform` of type `class Transform`. A `Transform` object has a single data element of type `Matrix` and a number of member functions. A `Matrix` is simply a 4×4 array[1] of `reals` defined using `typedef real Matrix[4][4]`. Such a matrix suffices for performing all of the transformations (affine and perspective) possible in three-dimensional space.[2] Any combination of transformations can be represented by a single transformation matrix. This means that consecutive transformations of a `Point` can be "saved up" and applied to its coordinates all at once when needed, rather than updating them for each transformation.

`Transforms` work by performing matrix multiplication of `Matrix` with the homogeneous `world_coordinates` of `Points`. If a set of homogeneous coordinates $\alpha = (x, y, z, w)$ and

$$\texttt{Matrix}\ M = \begin{pmatrix} a & e & i & m \\ b & f & j & n \\ c & g & k & o \\ d & h & l & p \end{pmatrix}$$

then the set of homogeneous coordinates β resulting from multiplying α and M is calculated as follows:

$$\beta = \alpha \times M = ((xa+yb+zc+wd), (xe+yf+zg+wh), (xi+yj+zk+wl), (xm+yn+zo+wp))$$

Please note that each coordinate of β can be influenced by all of the coordinates of α.

Operations on matrices are very important in computer graphics applications and are described in many books about computer graphics and geometry. For 3DLDF, I've mostly used Huw Jones' *Computer Graphics through Key Mathematics* and David Salomon's *Computer Graphics and Geometric Modeling*.

It is often useful to declare and use `Transform` objects in 3DLDF, just as it is for `transforms` in Metafont. Transformations can be stored in `Transforms` and then be used to transform `Points` by means of `Point::operator*=(const Transform&)`.

```
1. Transform t;
2. t.shift(0, 1);
3. Point p(1, 0, 0);
4. p *= t;
```

---

[1] It is unfortunate that the terms "array", "matrix", and "vector" have different meanings in C++ and in normal mathematical usage. However, in practice, these discrepancies turn out not to cause many problems. Stroustrup, *The C++ Programming Language*, § 22.4, p. 662.

[2] In fact, none of the operations for transformations require all of the elements of a 4×4 matrix. In many 3D graphics programs, the matrix operations are modified to use smaller transformation matrices, which reduces the storage requirements of the program. This is a bit tricky, because the affine transformations and the perspective transformation use different elements of the matrix. I consider that the risk of something going wrong, possibly producing hard-to-find bugs, outweighs any benefits from saving memory (which is usually no longer at a premium, anyway). In addition, there may be some interesting non-affine transformations that would be worth implementing. Therefore, I've decided to use full 4×4 matrices in 3DLDF.

5. `p.show("p:");`
   ⊣ `p: (1, 1, 0)`

When a `Transform` is declared (line 1), it is initialized to an *identity matrix*. All identity matrices are square, all of the elements of the main diagonal (upper left to lower right) are 1, and all of the other elements are 0. So a 4×4 identity matrix, as used in 3DLDF, looks like this:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

If a matrix A is multiplied with an identity matrix I, the result is identical to A, i.e., $A \times I = A$. This is the salient property of an identity matrix.

The same affine transformations are applied in the same way to `Transforms` as they are to `Points`, i.e., the functions `scale()`, `shift()`, `shear()`, and `rotate()` correspond to the `Point` versions of these functions, and they take the same arguments:

```
Point p;
Transform t;
p.shift(3, 4, 5);
t.shift(3, 4, 5);
⇒ p.transform ≡ t
p.show_transform("p:");
⊣ p:
  Transform:
         0    0.707   0.707       0
    -0.866    0.354  -0.354       0
      -0.5   -0.612   0.612       0
         0        0       0       1
t.show("t:");
⊣ t:
         0    0.707   0.707       0
    -0.866    0.354  -0.354       0
      -0.5   -0.612   0.612       0
         0        0       0       1
```

4.1 Applying Transforms to Points

A `Transform` t is applied to a `Point` P using the binary `*=` operation (`Point::operator*=(const Transform&)`) which performs matrix multiplication of `P.transform` by `t`. See Section 22.6 [Point Reference; Operators], page 123.

```
Point P(0, 1);
Transform t;
t.rotate(90);
t.show("t:");
⊣ t:
```

```
   1       0       0       0
   0       0      -1       0
   0       1       0       0
   0       0       0       1
P *= t;
P.show_transform("P:");
⊣ P:
Transform:
   1       0       0       0
   0       0      -1       0
   0       1       0       0
   0       0       0       1
P.show("P:");
⊣ P: (0, 0, -1)
```

In the example above, there is no real need to use a `Transform`, since `P.rotate(90)` could have been called directly. As constructions become more complex, the power of `Transforms` becomes clear:

```
1. Point p0(0, 0, 0);
2. Point p1(10, 5, 10);
3. Point p2(16, 14, 32);
4. Point p3(25, 50, 99);
5. Point p4(12, 6, 88);
6. Transform a;
7. a.shift(2, 3, 4);
8. a.scale(1, 3, 1);
9. p2 *= p3 *= a;
10. a.rotate(p0, p1, 75);
11. p4 *= a;
12. p2.show("p2:");
    ⊣ p2: (18, 51, 36)
13. p3.show("p3:");
    ⊣ p3: (27, 159, 103)
14. p4.show("p4:");
    ⊣ p4: (24.4647, -46.2869, 81.5353)
```

In this example, a is shifted and scaled, and a is applied to both in line 9. This works, because the binary operation `operator*=(const Transform& t)` returns t, making it possible to chain invocations of `*=`. Following this, a is rotated $75°$ about the line through p_0 and p_1. Finally, all three transformations, which are stored in a, are applied to p_4.

4.2 Inverting Transforms

Inversion is another operation that can be performed on `Transforms`. This makes it possible to reverse the effect of a `Transform`, which may represent multiple transformations.

```
Point p;
Transform t;
t.shift(1, 2, 3);
t.scale(2, 3, 4);
t.rotate(45, 45, 30);
t.show("t:");
⊣ t:
   1.22   0.707    1.41       0
  0.238    2.59    -1.5       0
  -3.15    1.45       2       0
  -7.74    10.2    4.41       1
p *= t;
p.show("p:");
⊣ p: (-7.74, 10.2, 4.41)
Transform u;
u = t.inverse();
u.show("u:");
⊣ u:
  0.306  0.0265  -0.197 2.85e-09
  0.177   0.287  0.0906 -1.12e-09
  0.354  -0.167   0.125       0
     -1      -2      -3       1
p *= u;
p.show("p:");
⊣ p: (0, 0, 0)
u *= t;
u.show("u:");
⊣ u:
      1       0       0       0
      0       1       0       0
      0       0       1       0
      0       0       0       1
```

If `inverse()` is called with no argument, or with the argument `false`, it returns a `Transform` representing its inverse, and remains unchanged. If it is called with the argument `true`, it is set to its inverse.

Complete reversal of the transformations applied to a `Point`, as in the previous example, probably won't make much sense. However, partial reversal is a valuable technique. For example, it is used in `rotate()` for rotation about a line defined by two `Points`. The following example merely demonstrates the basic principle; an example that does something useful would be too complicated.

```
Transform t;
t.shift(3, 4, 5);
t.rotate(45);
t.scale(2, 2, 2);
Point p;
p *= t;
p.show("p:");
⊣ p: (6, 12.7279, 1.41421)
t.inverse(true);
p.rotate(90, 90);
p *= t;
p.show("p:");
⊣ p: (3.36396, -5.62132, -2.37868)
```

5 Drawing and Labeling Points

5.1 Drawing Points

It's all very well to declare `Points`, place them at particular locations, print their locations to standard output, and transform them, but none of these operations produce any MetaPost output. In order to do this, the first step is to use *drawing and filling commands*. The drawing and filling commands in 3DLDF are modelled on those in Metafont.

The following example demonstrates how to draw a dot specifying a `Color` (see Chapter 16 [Color Reference], page 87) and a *pen*[1].

```
Point P(0, 1);
P.drawdot(Colors::black, "pencircle scaled 3mm");
```

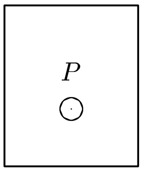


Figure 3.

In `drawdot()`, a `Color` argument precedes the `string` argument for the pen, so "`Colors::black`" must be specified as a placeholder in the call to `drawdot()`.[2]

The following example "undraws" a dot at the same location using a smaller pen. `undraw()` does not take a `Color` argument.

```
p.undrawdot("pencircle scaled 2mm");
```

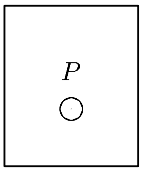


Figure 4.

For complete descriptions of `drawdot()` and `undrawdot()`, see Section 22.18 [Point Reference; Drawing], page 143.

Drawing and undrawing dots is not very exciting. In order to make a proper drawing it is necessary to *connect* the `Points`. The most basic way of doing this is to use the `Point` member function `draw()` with a `Point` argument:

```
Point p0;
Point p1(2, 2);
p0.draw(p1);
```

---

1 Pens are a concept from Metafont. In 3DLDF, there is currently no type "`Pen`". Pen arguments to functions are simply `strings`, and are written unaltered to `out_stream`. For more information about Metafont's `pens`, see Knuth, *The Metafontbook*, Chapter 4.

2 `Colors` are declared in the namespace `Colors`, so if you have a "`using`" declaration in the function where you use `drawdot()`, you can write "`black`" instead of "`Colors::black`". For more information about namespaces, see Stroustrup, *The C++ Programming Language*, Chapter 8.

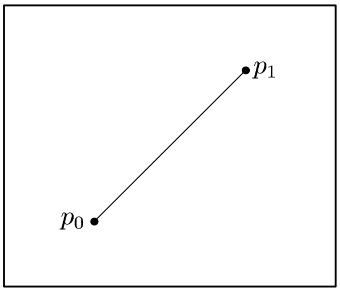


Figure 5.

`p0.draw(p1)` is equivalent in its effect to `p1.draw(p0)`.

The function `Point::draw()` takes a required `Point&` argument (a *reference*[3] to a `Point`) an optional `Color` argument, and optional `string` arguments for the dash pattern and the pen. The `string` arguments, if present, are passed unchanged to the output file. The empty `string` following the argument `p1` is a placeholder for the dash pattern argument, which isn't used here.

```
p0.draw(p1, Colors::gray, "", "pensquare scaled .5cm rotated 45");
```

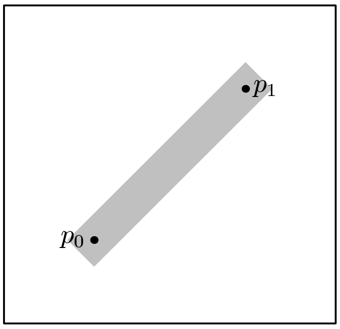


Figure 6.

The function `Point::undraw()` takes a required `Point&` argument and optional `string` arguments for the dash pattern and the pen. Unlike `Point::draw()`, a `Color` argument would have no meaning for `Point::undraw()`. The `string` arguments are passed unchanged to the output file.

`undraw()` can be used to "hollow out" the region drawn in Fig. 6. Since a dash pattern is used, portions of the middle of the region are not undrawn.

```
p0.undraw(p1, "evenly scaled 6", "pencircle scaled .2cm");
```

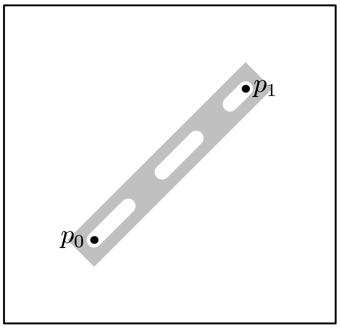


Figure 7.

[3] "A *reference* is an alternative name for an object. The main use of references is for specifying arguments and return values for functions in general and for overloaded operators (Chapter 11) in particular." Stroustrup, *The* C++ *Programming Language*, §5.5, p. 97.

For complete descriptions of `draw()` and `undraw()`, see Section 22.18 [Point Reference; Drawing], page 143.

5.2 Labeling Points

The labels in the previous examples were made by using the functions `Point::label()` and `Point::dotlabel()`, which make it possible to include TEX text in a drawing.

`label()` and `dotlabel()` take `string` arguments for the text of the label and the position of the label with respect to the `Point`. The label text is formatted using TEX, so it can contain math mode material between dollar signs. Please note that double backslashes must be used, where a single backslash would suffice in a file of MetaPost code, for example, for TEX control sequences. Alternatively, a `short` argument can be used for the label.

The position argument is optional, with `"top"` as the default. If the empty `string` `""` is used, the label will centered about the `Point` itself. This will usually only make sense for `label()`, because it would otherwise interfere with the dot. Valid arguments for the position are the same as in MetaPost: `"top"`, `"bot"` (bottom), `"lft"` (left), `"rt"` (right), `"ulft"` (upper left), `"urt"` (upper right), `"llft"` (lower left), and `"lrt"` (lower right).

```
Point p0;
Point p1(1);
Point p2(2);
Point p3(p0);
Point p4(p1);
Point p5(p2);
p3 *= p4 *= p5.shift(0, 1);
p0.draw(p1);
p1.draw(p2);
p2.draw(p5);
p5.draw(p4);
p4.draw(p3);
p3.draw(p0);
p0.label($p_0$, "");
p1.dotlabel(1);
p2.dotlabel("p2", "bot");
p3.dotlabel("This is $p_3$", "lft");
p4.label(4);
p5.label("$\\leftarrow p_5$", "rt");
```

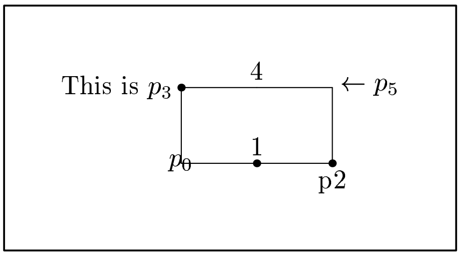


Figure 8.

For complete descriptions of `Point::label()` and `Point::dotlabel()`, see Section 22.19 [Points; Labelling], page 146.

6 Paths

Points alone are not enough for making useful drawings. The next step is to combine them into Paths, which are similar to Metafont's **paths**, except that they are three-dimensional. A Path consists of a number of Points and strings representing the *connectors*. The latter are not processed by 3DLDF, but are passed unchanged to the output file. They must be valid connectors for MetaPost, e.g.:

```
..
...
--
---
&
curl{2}..
{dir 60}..
{z1 - z2}..
.. tension 1 and 1.5..
..controls z1 and z2..
```

Usually, it will only make sense to use .. or --, and not ..., ---, tension, curl, controls, or any of the other possibilities, in Paths, unless you are sure that they will only be viewed with no foreshortening due to the perspective projection. This can be the case, when a Path lies in a plane parallel to one of the major planes, and is projected using parallel projection onto that plane. Otherwise, the result of using these connectors is likely to be unsatisfactory, because MetaPost performs its calculations based purely on the two-dimensional values of the points in the perspective projection. While the Points on the Path will be projected correctly, the course of the Path between these Points is likely to differ, depending on the values of the Focus used (see Section 9.2 [Focuses], page 62), so that different views of the same Path may well be mutually inconsistent. This problem doesn't arise with "--", since the perspective projection does not "unstraighten" straight lines, but it does with "..", even without tension, curl, or controls. The solution is to use enough Points, since a greater number of Points on a Path tends to reduce the number of possible courses through the Points.[1]

6.1 Declaring and Initializing Paths

There are various ways of declaring and initializing Paths. The simplest is to use the constructor taking two Point arguments:

```
Point A;
Point B(2, 2);
Path p(A, B);
p.draw();
```

[1] I believe that counter-examples could probably constructed, but for the most common cases, the principle applies.

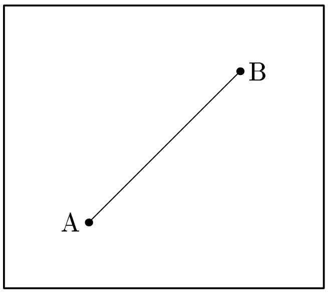


Figure 9.

Paths created in this way are important, because they are guaranteed to be linear, as long as no operations are performed on them that cause them to become non-linear. Linear Paths can be used to find *intersections*. See Section 26.17 [Path Intersections], page 197.

Paths can be declared and initialized using a single connector and an arbitrary number of Points. The first argument is a string specifying the connector. It is followed by a bool, indicating whether the Path is cyclical or not. Then, an arbitrary number of pointers to Point follow. The last argument must be 0.[2]

```
Point p[3];
p[0].shift(1);
p[1].set(1, 2, 2);
p[2].set(1, 0, 2);
Path pa("--", true, &p[0], &p[1], &p[2], 0);
pa.draw();
```

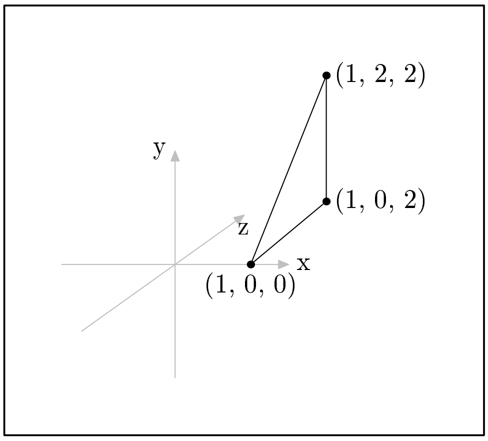


Figure 10.

Another constructor must be used for Paths with more than one connector and an arbitrary number of Points. The argument list starts with a pointer to Point, followed by string for the first connector. Then, pointer to Point arguments alternate with string arguments for the connectors. Again, the list of arguments ends in 0. There is no need for a bool to indicate whether the Path is cyclical or not; if it is, the last non-zero argument will be a connector, otherwise, it will be a pointer to Point.

```
Point p[8];
```

[2] It's easy to forget to use Point* arguments, rather than plain Point arguments, and to forget to end the list of arguments with 0. If plain Point arguments are used, compilation fails with GCC. With the DEC compiler, compilation succeeds, but a memory fault error occurs at run-time. If the argument list doesn't end in 0, neither compiler signals an error, but a memory fault error always occurs at run-time.

```
p[0].set(-2);
p[1].set(2);
p[2].set(0, 0, -2);
p[3].set(0, 0, 2);
p[4] = p[0].mediate(p[2]);
p[5] = p[2].mediate(p[1]);
p[6] = p[1].mediate(p[3]);
p[7] = p[3].mediate(p[0]);
p[4] *= p[5] *= p[6] *= p[7].shift(0, 1);
Path pa(&p[0], "..", &p[4], "...", &p[2],
        "..", &p[5], "...", &p[1], "..", &p[6],
        "...", &p[3], "..", &p[7], "...", 0);
pa.draw();
```

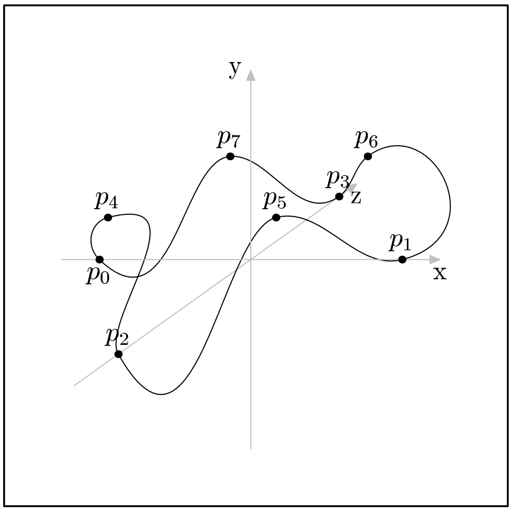


Figure 11.

As mentioned above (see Section 1.5.1 [Accuracy], page 7), specifying connectors is problematic for three-dimensional `Paths`, because MetaPost ultimately calculates the "most pleasing curve" based on the two-dimensional points in the MetaPost code written by 3DLDF.[3] For this reason, it's advisable to avoid specifying '`curl`', '`dir`', '`tension`' or control points in connectors. The more `Points` a (3DLDF) `Path` or other object contains, the less freedom MetaPost has to determine the (MetaPost) `path` through them. So a three-dimensional `Path` or other object in 3DLDF should have enough `Points` to ensure satisfactory results. The `Path` in Fig. 11 does not really have enough `Points`. It may require some trial and error to determine what a sufficient number of `Points` is in a given case.

`Paths` are very flexible, but not always convenient. 3DLDF provides a number of classes representing common geometric `Shapes`, which will be described in subsequent sections, and I intend to add more in the course of time.

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

6.2 Drawing and Filling Paths

The easiest way to draw a Path is with no arguments.

```
Point pt[5];
pt[0].set(-1, -2);
pt[1].set(0, -3);
pt[2].set(1, 0);
pt[3].set(2, 1);
pt[4].set(-1, 2);
Path pa("..", true, &pt[0], &pt[1], &pt[2], &pt[3], &pt[4], 0);
pa.draw();
```

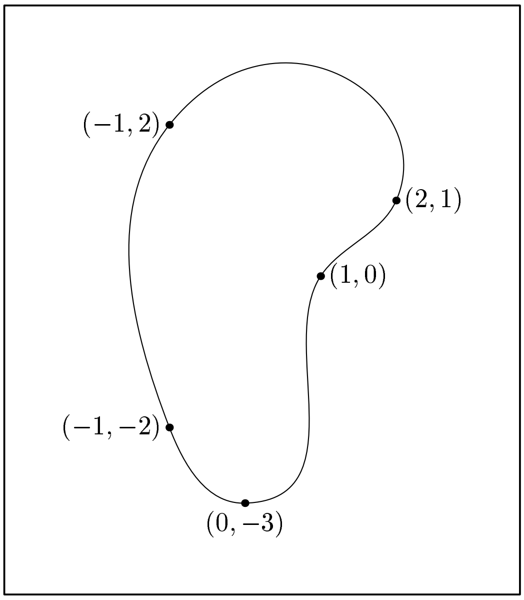


Figure 12.

Since pa is closed, it can be filled as well as drawn. The following example uses fill() with a Color argument, in order to avoid having a large splotch of black on the page. Common Colors are declared in the namespace Colors. See Chapter 16 [Color Reference], page 87.

```
pa.fill(Colors::gray);
```

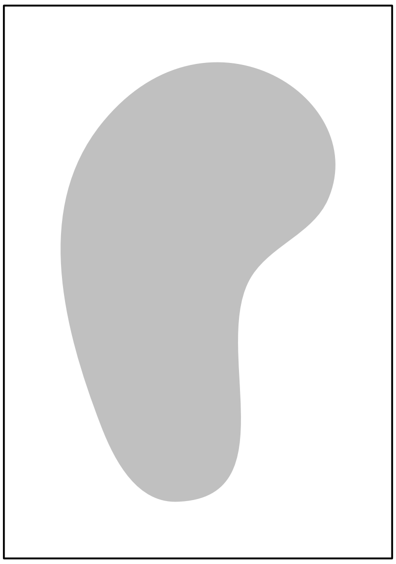


Figure 13.

Closed Paths can be filled and drawn, using the function filldraw(). This function draws the Path using the pen specified, or MetaPost's currentpen by default. A Color for drawing the Path can also be specified, otherwise, the default color (currently Colors::black) is used. In addition, the Path is filled using a second Color, which can be specified, or the background_color (Colors::background_color), by default. Filling a Path using the background color causes it to hide objects that lie behind it. See Section 9.3 [Surface Hiding], page 66, for a description of the surface hiding algorithm, and examples. Currently, this algorithm is quite primitive and only works for simple cases.

```
Point p0(-3, 0, 1);
Point p1(3, 1, 1);
p0.draw(p1);
pa.filldraw();
```

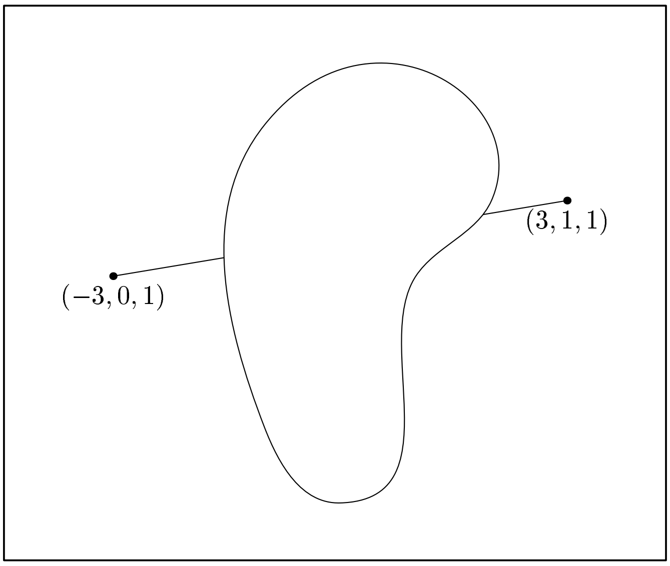


Figure 14.

The following example uses arguments for the `Colors` used for drawing and filling, and the pen. The empty string argument before the pen argument is a placeholder for the dash pattern argument.

```
pa.filldraw(black, gray, "",
   "pensquare xscaled 3mm yscaled 1mm rotated 60");
```

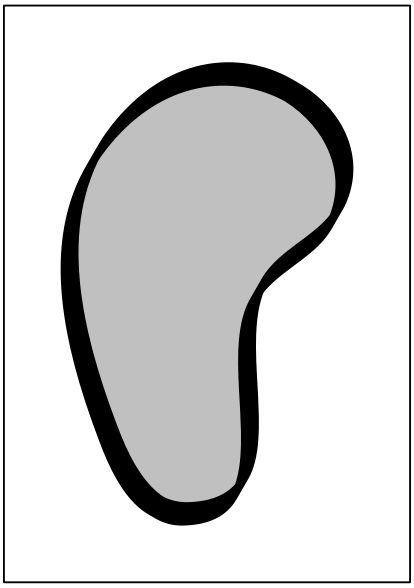


Figure 15.

`Paths` can also be "undrawn", "unfilled", and "unfilldrawn", using the corresponding functions:

```
pa.fill(gray);
p0.undraw(p1, "", "pencircle scaled 3mm");
```

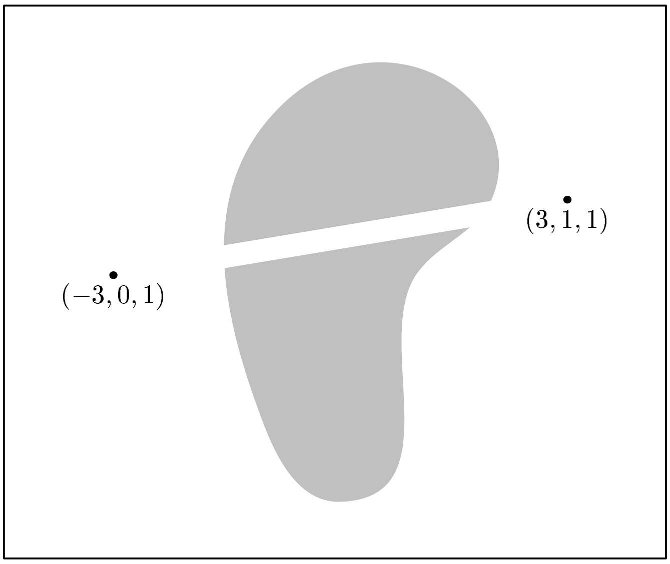


Figure 16.

```
pa.fill(gray);
```

```
Path q;
q = pa;
q.scale(.5, .5);
q.unfill();
```

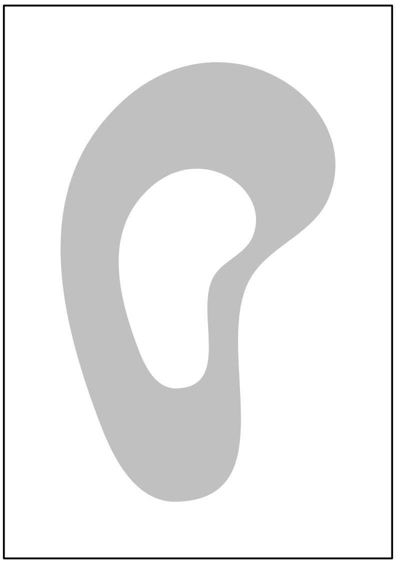


Figure 17.

The function `unfilldraw()` takes a `Color` argument for drawing the `Path`, which is `*Colors::background_color` by default. This makes it possible to unfill the `Path` while drawing the outline with a visible `Color`. On the other hand, it also makes it necessary to specify `*Colors::background_color` or `Colors::white`, if the user wants to use the dash pattern and/or pen arguments, without drawing the `Path`.

```
pa.fill(gray);
q.unfilldraw(white, "", "pensquare xscaled 3mm yscaled 1mm");
```

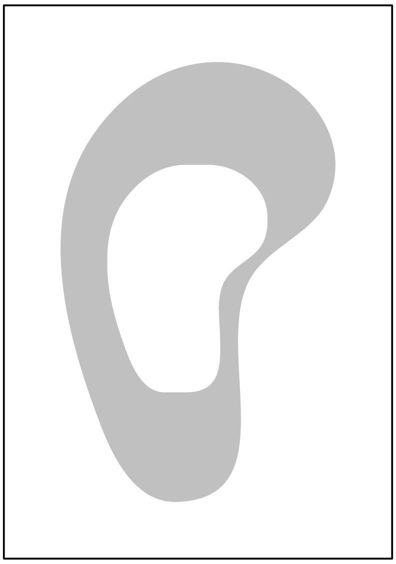


Figure 18.

The following example demonstrates the use of `unfilldraw()` with `black` as its `Color` argument. Unfortunately, it also demonstrates one of the limitations of the surface hiding algorith: The line from `p0` to `p1` is hidden by the filled `Path pa`. Since the portion of `pa` covered by `Path q` has been unfilled, $\overrightarrow{p_0p_1}$ should be visible as it passes through `q`. However, from the point of view of 3DLDF, there is no relationship between `pa` and `q`; nor does it "know" whether a `Path` has been filled or unfilled. If it's on a `Picture`, it will hide objects lying behind it, unless the surface hiding algorithm fails for another reason. See Section 9.3 [Surface Hiding], page 66, for more information.

```
p0.draw(p1);
pa.fill(gray);
q.unfilldraw(black, "", "pensquare xscaled 3mm yscaled 1mm");
```

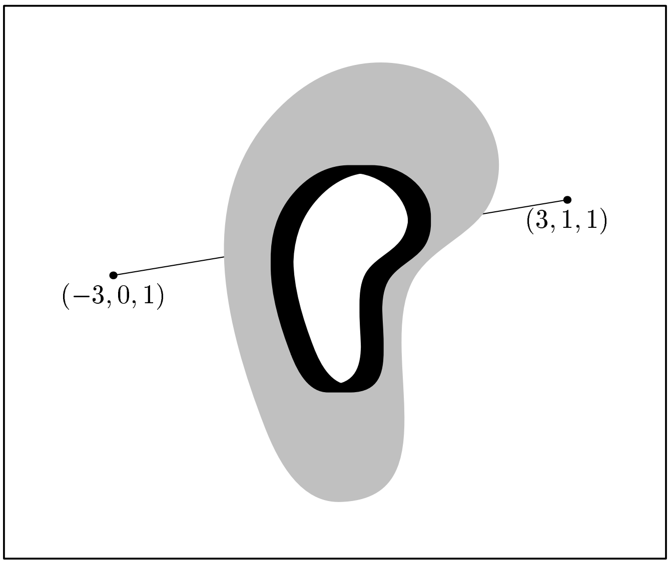


Figure 19.

See Section 26.12 [Paths; Drawing and Filling], page 179, for more information, and complete descriptions of the functions.

7 Plane Figures

3DLDF currently includes the following classes representing plane geometric figures: `Polygon`, `Reg_Cl_Plane_Curve` ("Regular Closed Plane Curve"), `Reg_Polygon` ("Regular Polygon"), `Rectangle`, `Ellipse` and `Circle`. `Polygon` and `Reg_Cl_Plane_Curve` are derived from `Path`, `Reg_Polygon` and `Rectangle` are derived from `Polygon`, and `Ellipse` and `Circle` are derived from `Reg_Cl_Plane_Curve`. `Polygon` and `Reg_Cl_Plane_Curve` are meant to be used as base classes only, so objects of these types should normally never be declared.

Since `Reg_Polygon`, `Rectangle`, `Ellipse`, and `Circle` all ultimately derive from `Path`, they are really just special kinds of `Path`. In particular, they inherit their drawing and filling functions from `Path`, and their transformation functions take the same arguments as the `Path` versions. They also have constructors and setting functions that work in a similar way, with a few minor differences, to account for their different natures. See Chapter 27 [Polygon Reference], page 198, Chapter 29 [Rectangle Reference], page 210, Chapter 31 [Ellipse Reference], page 221, and Chapter 32 [Circle Reference], page 237, for complete information on these classes.

7.1 Regular Polygons

The following example creates a pentagon in the x-z plane, centered about the origin, whose enclosing circle has a radius equal to 3cm.

```
default_focus.set(2, 3, -10, 2, 3, 10, 10);
Reg_Polygon p(origin, 5, 3);
p.draw();
```

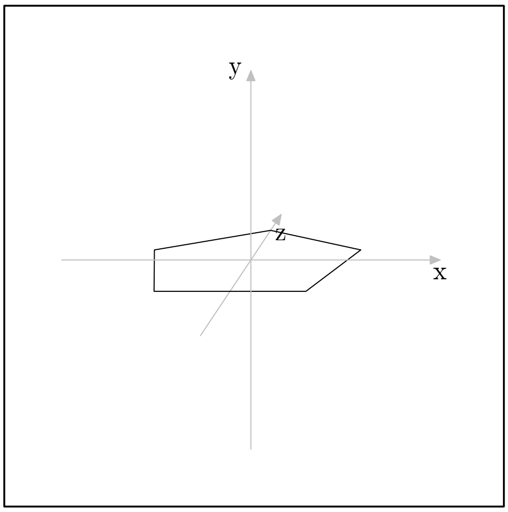


Figure 20.

Three additional arguments cause the pentagon to be rotated about the x, y, and z axes by the amount indicated. In this example, it's rotated 90° about the x-axis, so that it comes to lie in the x-y plane:

```
Reg_Polygon p(origin, 5, 3, 90);
```

```
p.draw();
```

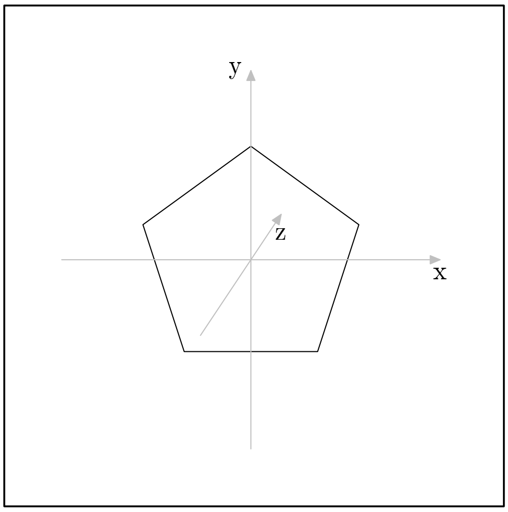


Figure 21.

In this example, it's rotated 36° about the y-axis, so that it appears to point in the opposite direction from the first example:

```
Reg_Polygon p(origin, 5, 3, 0, 36);
p.draw();
```

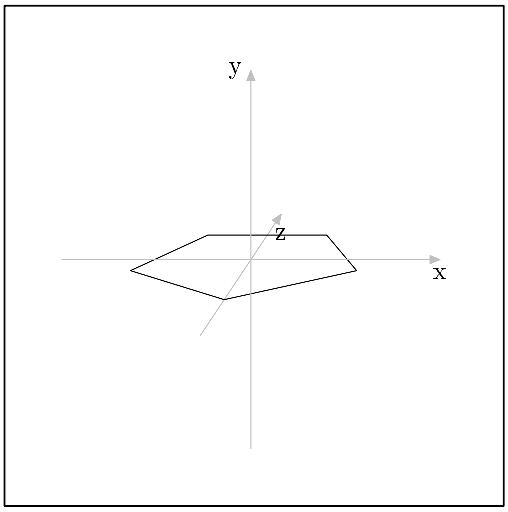


Figure 22.

In this example, it's rotated 90° about the z-axis, so that it lies in the z-y plane:

```
Reg_Polygon p(origin, 5, 3, 0, 0, 90);
p.draw();
```

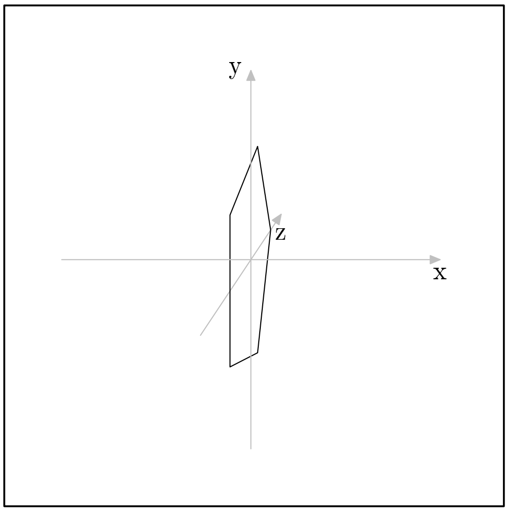


Figure 23.

In this example, it's rotated 45° about the x, y, and z-axes in that order:

```
Reg_Polygon p(origin, 5, 3, 45, 45, 45);
p.draw();
```

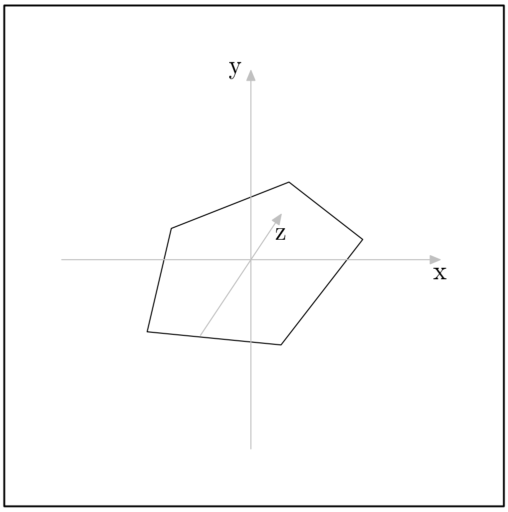


Figure 24.

`Reg_Polygons` need not be centered about the `origin`. If another `Point` *pt* is used as the first argument, the `Reg_Polygon` is first created with its center at the origin, then the specified rotations, if any, are performed. Finally, the `Reg_Polygon` is shifted such that its center comes to lie on *pt*:

```
Point P(-2, 1, 1);
Reg_Polygon hex(P, 6, 4, 60, 30, 30);
hex.draw();
```

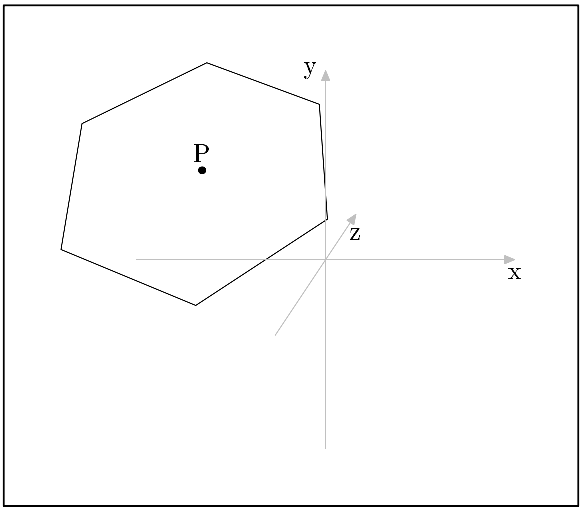


Figure 25.

In the following example, the `Reg_Polygon polygon` is first declared using the default constructor, which creates an empty `Reg_Polygon`. Then, the `polygon` is repeatedly changed using the setting function corresponding to the constructor used in the previous examples. Fig. 26 demonstrates that a given `Reg_Polygon` need not always have the same number of sides.

```
Point p(0, -3);
Reg_Polygon polygon;
for (int i = 3; i < 9; ++i)
  {
    polygon.set(p, i, 3);
    polygon.draw();
    p.shift(0, 1);
  }
```

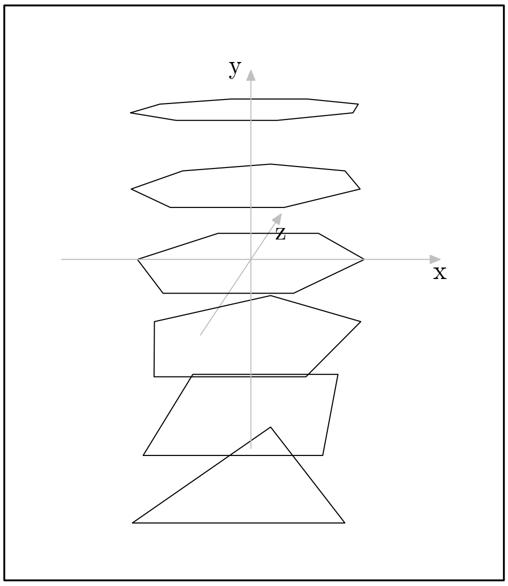


Figure 26.

7.2 Rectangles

A `Rectangle` can be constructed in the x-z plane by specifying a center `Point`, the width, and the height:

```
Rectangle r(origin, 2, 3);
r.draw();
```

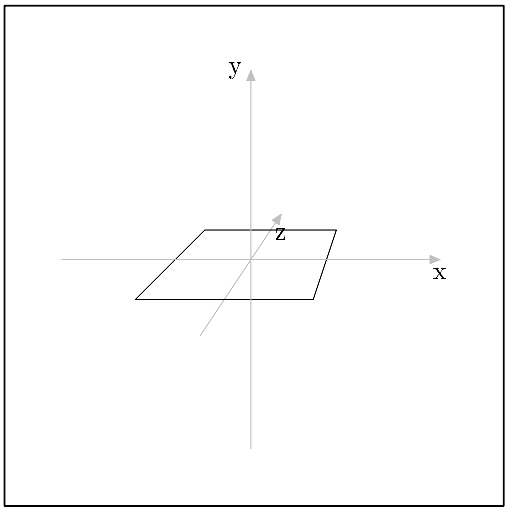


Figure 27.

Three additional arguments can be used to specify rotation about the x, y, and z-axes respectively:

```
Rectangle r(origin, 2, 3, 30, 45, 15);
r.draw();
```

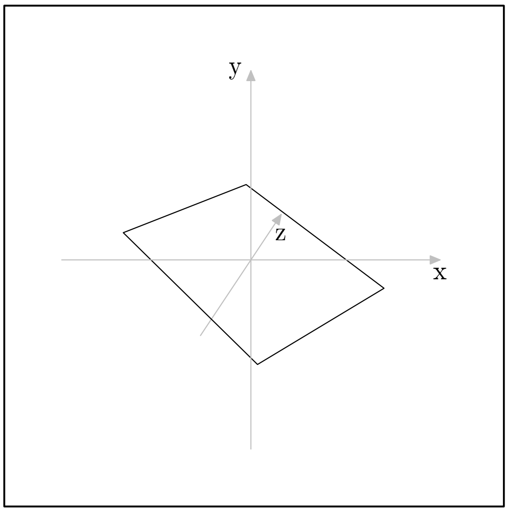


Figure 28.

If a `Point` p other than the origin is specified as the center of the `Rectangle`, the latter is first created in the x-z plane, centered about the origin, as above. Then, any rotations specified are performed. Finally, the `Rectangle` is shifted such that its center comes to lie at p:

```
Point p0(.5, 1, 3);
Rectangle r(p0, 4, 2, 30, 30, 30);
r.draw();
```

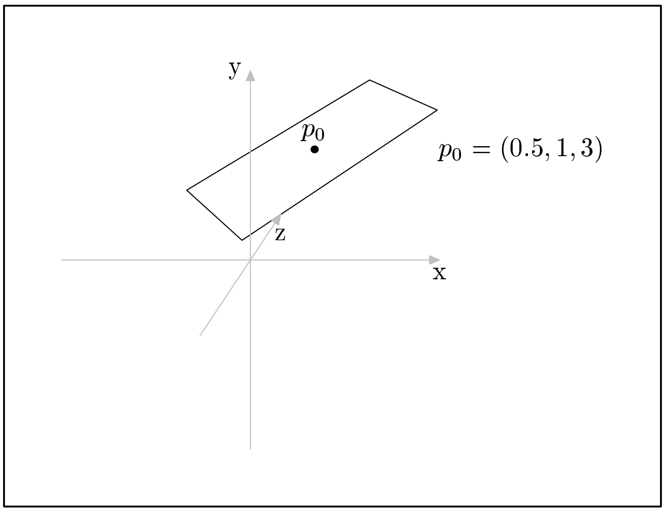


Figure 29.

This constructor has a corresponding setting function:

```
Rectangle r;
for (int i = 0; i < 180; i += 30)
  {
    r.set(origin, 4, 2, i);
    r.draw();
  }
```

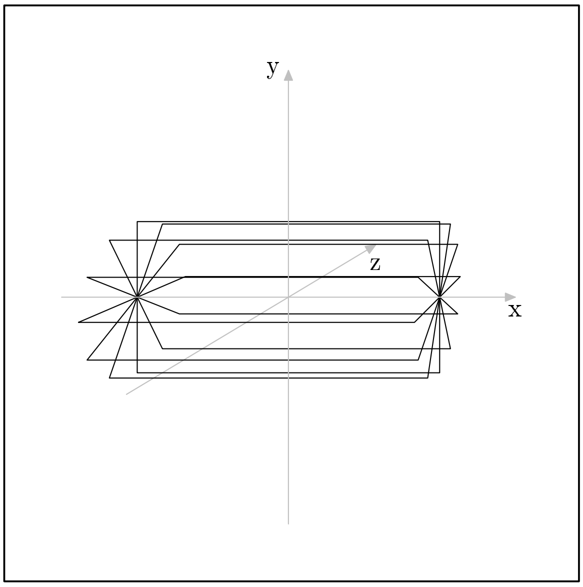


Figure 30.

`Rectangles` can also be specified using four `Points` as arguments, whereby they must be ordered so that they are contiguous in the resulting `Rectangle`:

```
Point pt[4];
pt[0].shift(-1, -2);
pt[2] = pt[1] = pt[0];
pt[1].rotate(180);
pt[3] = pt[1];
pt[2] *= pt[3].rotate(0, 180);
Rectangle r(pt[0], pt[2], pt[3], pt[1]);
r.draw();
```

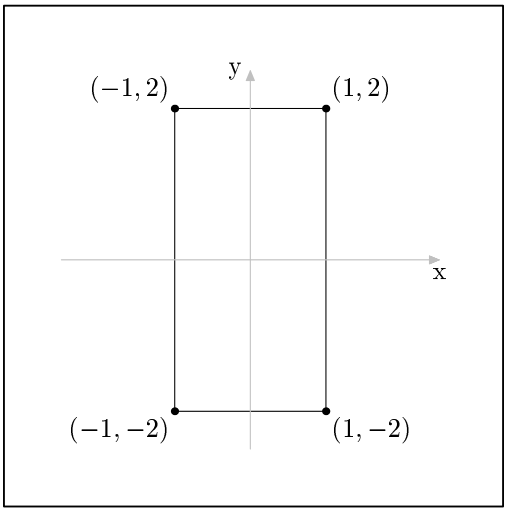


Figure 31.

This constructor checks whether the `Point` arguments are coplanar, however, it does not check whether they are really the corners of a valid rectangle; the user, or the code that calls this function, must ensure that they are. In the following example, `r`, although not rectangular, is a `Rectangle`, as far as 3DLDF is concerned:

```
pt[0].shift(0, -1);
pt[3].shift(0, 1);
Rectangle q(pt[0], pt[2], pt[3], pt[1]);
q.draw();
```

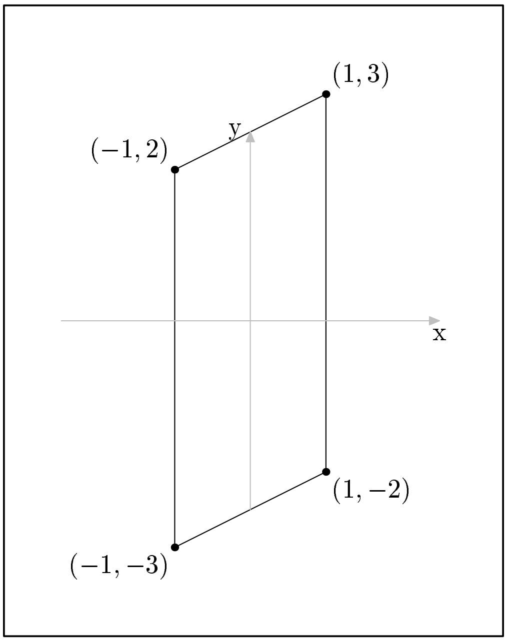


Figure 32.

This constructor is not really intended to be used directly, but should mostly be called from within other functions, that should ensure that the arguments produce a rectangular `Rectangle`. There is also no guarantee that transformations or other functions called on `Rectangle`, `Circle`, or other classes representing geometric figures won't cause them to become non-rectangular, non-circular, or otherwise irregular. Sometimes, this might even be desirable. I plan to add the function `Rectangle::is_rectangular()` soon, so that users can test `Rectangles` for rectangularity.

7.3 Ellipses

`Ellipse` has a constructor similar to those for `Reg_Polygon` and `Rectangle`. The first argument is the center of the `Ellipse`, and the following two specify the lengths of the horizontal and vertical axes respectively. The `Ellipse` is first created in the x-z plane, centered about the origin. The horizontal axis lies along the x-axis and the vertical axis lies along the z-axis. The three subsequent arguments specify the amounts of rotation about the x, y, and z-axes respectively and default to 0. Finally, `Ellipse` is shifted such that its center comes to lie at the `Point` specified in the first argument.

```
Point pt(-1, 1, 1);
Ellipse e(pt, 3, 6, 90);
e.draw();
```

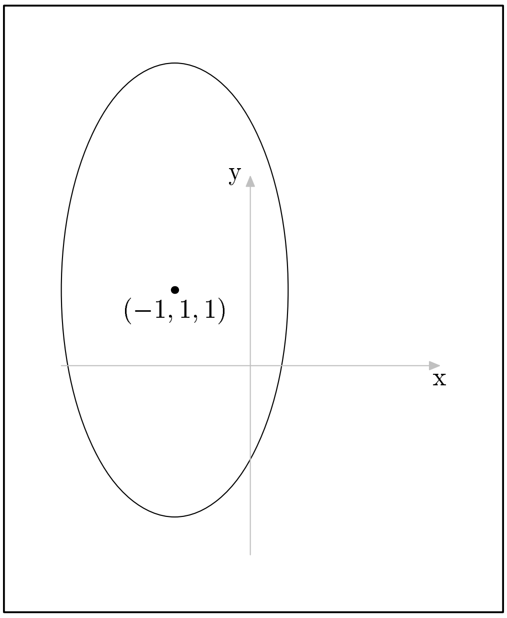


Figure 33.

As you may expect, this constructor has a corresponding setting function:

```
Ellipse e;
real h_save = 1.5;
real v_save = 2;
real h = h_save;
real v = v_save;
Point p(-1);
for (int i = 0; i < 5; ++i)
  {
      e.set(p, h, v, 90);
      e.draw();
      h_save += .25;
      v_save += .25;
      h *= sqrt(h_save);
      v *= sqrt(v_save);
      p.shift(0, 0, 2);
  }
```

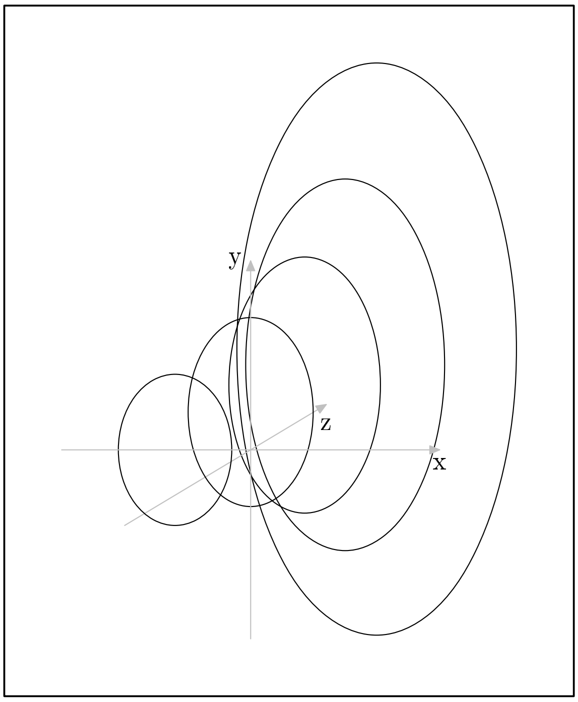


Figure 34.

7.4 Circles

`Circles` are constructed just like `Ellipses`, except that the vertical and horizontal axes are per definition the same, so there's only one argument for the diameter, instead of two for the horizontal and vertical axes:

```
Point P(0, 2, 1);
Circle c(P, 3.5, 90, 90);
c.draw();
```

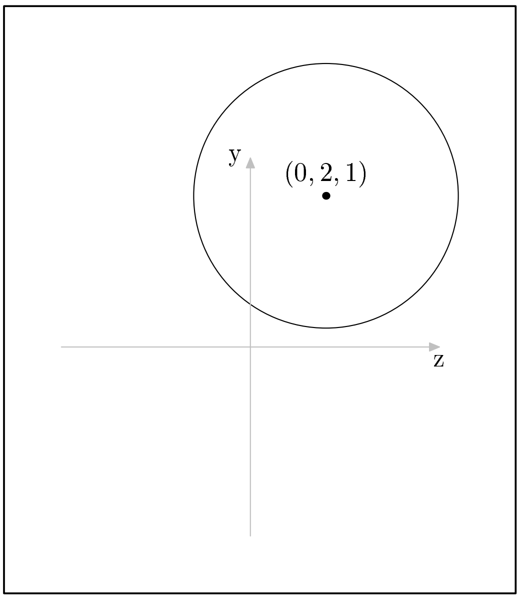


Figure 35.

This constructor, too, has a corresponding setting function:

```
Circle c;
Point p(-1, 0, 5);
for (int i = 0; i < 16; ++i)
  {
      c.set(p, 5, i * 22.5, 0, 0, 64);
      c.draw();
  }
```

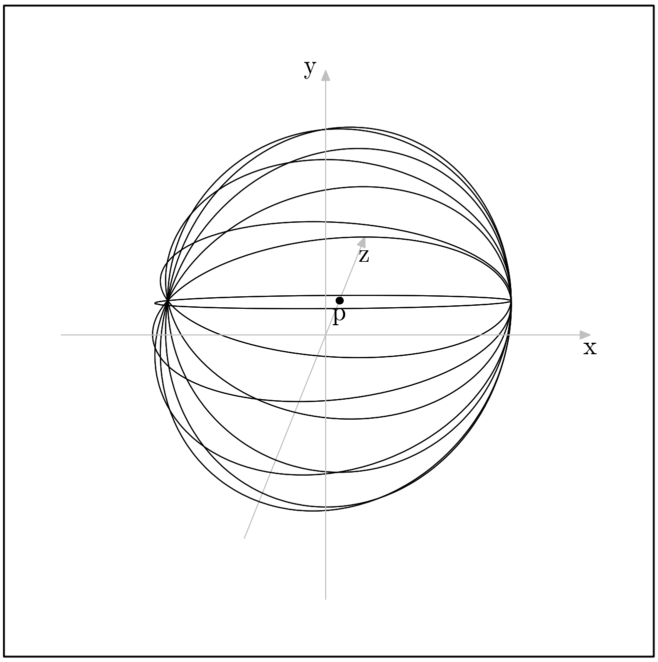


Figure 36.

In the preceding example, the last argument to `set()`, namely “64”, is for the number of `Points` used for constructing the perimeter of the `Circle`. The default value is 16, however, if it is used, foreshortening distorts the most nearly horizontal `Circle`. Increasing the number of points used improves its appearance. However, there may be a limit to how much improvement is possible. See Section 1.5.1 [Accuracy], page 7.

8 Solid Figures

8.1 Cuboids

A *cuboid* is a solid figure consisting of six rectangular faces that meet at right angles. A cube is a special form of cuboid, whose faces are all squares. The constructor for the class `Cuboid` follows the pattern familiar from the constructors for the plane figures: The first argument is the center of the `Cuboid`, followed by three `real` arguments for the height, width, and depth, and then three more `real` arguments for the angles of rotation about the x, y, and z-axes. The `Cuboid` is first constructed with its center at the origin. Its width, height, and depth are measured along the x, y, and z-axes respectively. If rotations are specified, it is rotated about the x, y, z-axes in that order. Finally, it is shifted such that its center comes to lie on its `Point` argument, if the latter is not the origin.

If the width, height, and depth arguments are equal, the `Cuboid` is a cube:

```
Cuboid c0(origin, 3, 3, 3, 0, 30);
c0.draw();
```

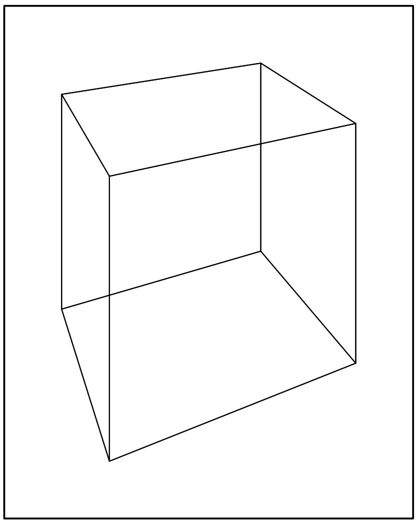


Figure 37.

In the following example, the `Cuboid` is "filldrawn", so that the lines dilineating the hidden surfaces of the `Cuboid` are covered.

```
Cuboid c1(origin, 3, 4, 5, 0, 30);
c1.filldraw();
```

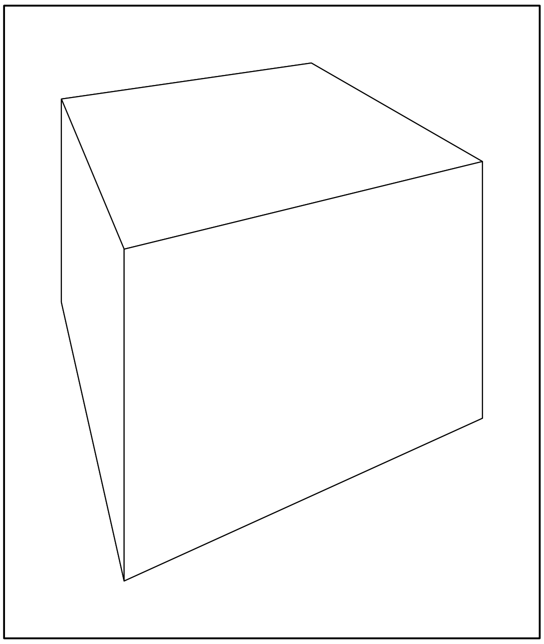


Figure 38.

8.2 Polyhedron

The class Polyhedron is meant for use only as a base class; no objects of type Polyhedron should be declared. Instead, there is a class for each of the different drawable polyhedra. Currently, 3DLDF defines only three: Tetrahedron, Dodecahedron, and Icosahedron. There's no need for a Cube class, because cubes can be drawn using Cuboid (see Section 8.1 [Cuboid Getstart], page 46).

Polyhedra have a high priority in my plans for 3DLDF. I intend to add Octahedron soon, which will complete the set of regular Platonic polyhedra. Then I will begin adding the semi-regular Archimedean polyhedra, and their duals.

The constructors for the classes derived from Polyhedron follow the pattern familiar from the classes already described. The constructors for the classes described below have identical arguments: First, a Point specifying the center, then a real for the diameter of the surrounding circle (*Umkreis*, in German) of one of its polygonal faces, followed by three real arguments for the angles of rotation about the main axes.

8.2.1 Tetrahedron

The center of a tetrahedron is the intersection of the lines from a vertex to the center of the opposite side. At least, in 3DLDF, this is the center of a Tetrahedron. I'm not 100° certain that this is mathematically correct.

```
Tetrahedron t(origin, 4);
t.draw();
t.get_center().dotlabel("$c$");
```

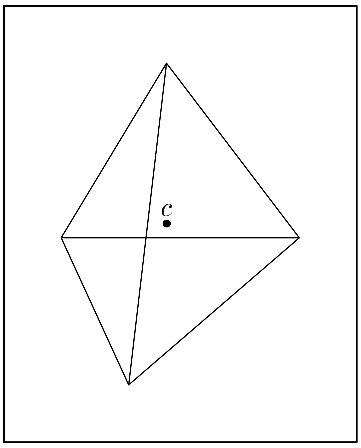
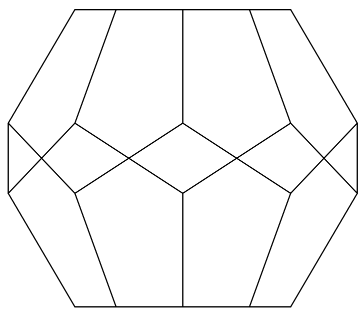


Figure 39.

8.2.2 Dodecahedron

A dodecahedron has 12 similar regular pentagonal faces. The following examples show the same `Dodecahedron` using different projections:

```
default_focus.set(2, 5, -10, 2, 5, 10, 10);
Dodecahedron d(origin, 3);
d.draw();
```



Parallel projection, x-y plane

Figure 40.

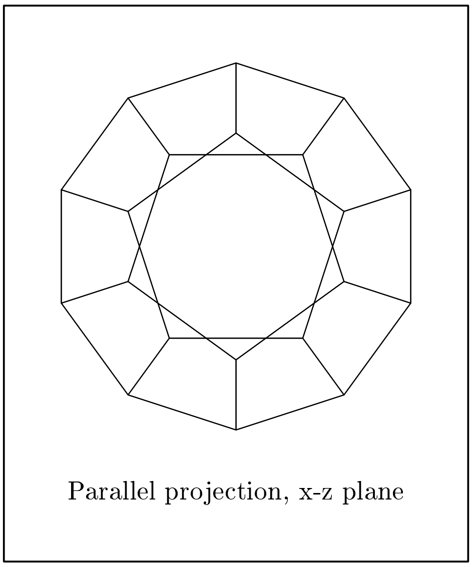


Figure 41.

Please note that the `Dodecahedron` in Fig. 42 is drawn, and not filldrawn!

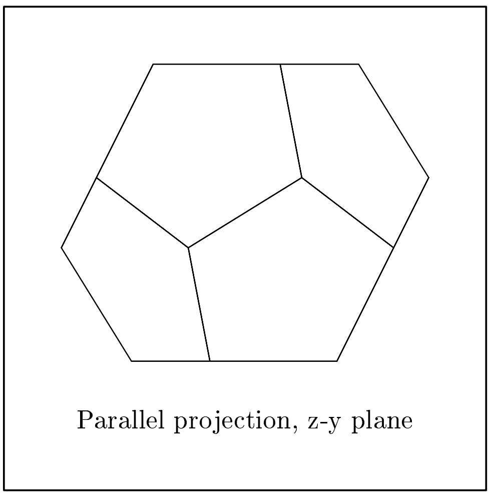


Figure 42.

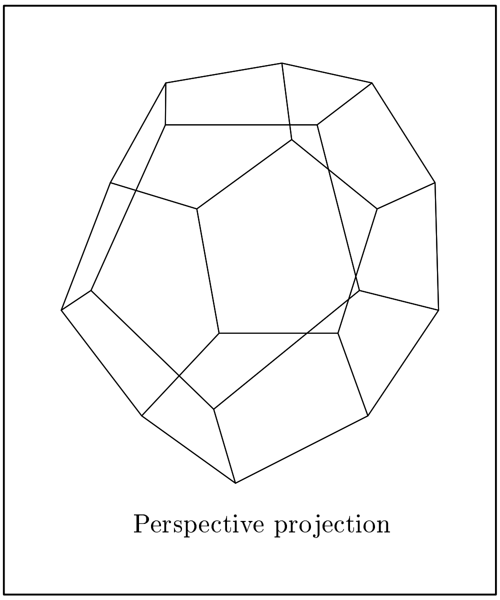


Figure 43.

In Fig. 44, `d` is filldrawn. In this case, the surface hiding algorithm has worked properly. See Section 9.3 [Surface Hiding], page 66.

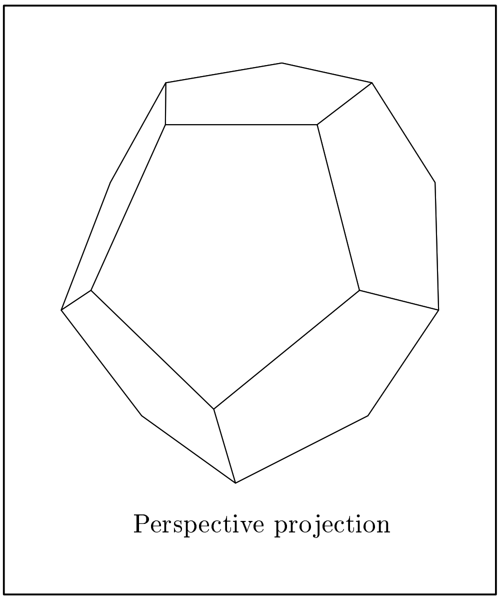
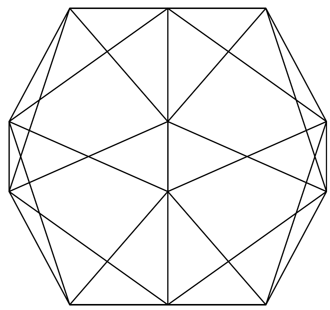


Figure 44.

8.2.3 Icosahedron

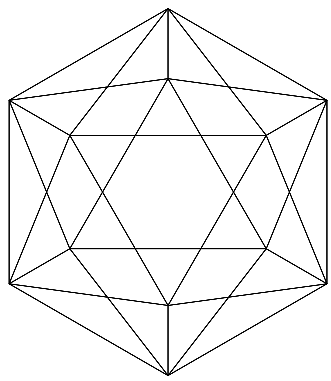
An icosahedron has 20 similar regular triangular faces. The following examples show the same `Icosahedron` using different projections:

```
default_focus.set(3, 0, -10, 2, 0, 10, 10);
Icosahedron i(origin, 3);
i.draw();
```



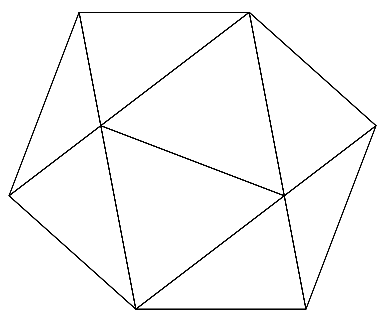
Parallel projection, x-y plane

Figure 45.



Parallel projection, x-z plane

Figure 46.



Parallel projection, z-y plane

Figure 47.

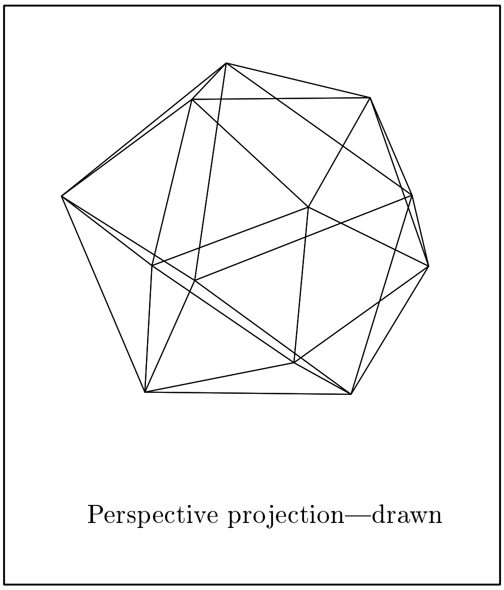


Figure 48.

In Fig. 49, `i` is filldrawn. In this case, the surface hiding algorithm has worked properly. See Section 9.3 [Surface Hiding], page 66.

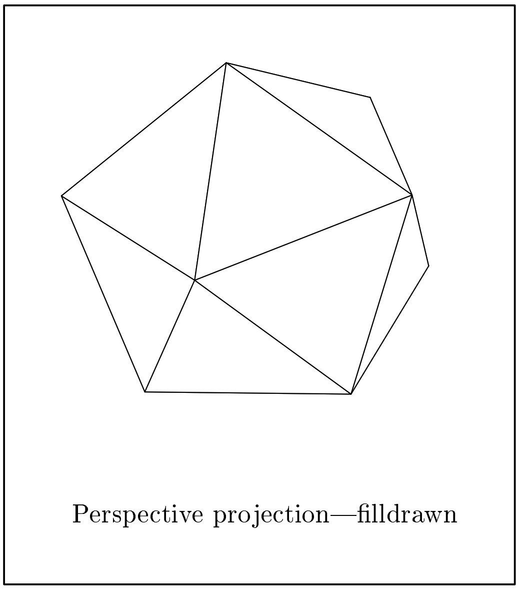


Figure 49.

9 Pictures

Applying drawing and filling operations to the drawable objects described in the previous chapters isn't enough to produce output. These operations merely modify the `Picture` object that was passed to them as an argument (`current_picture`, by default).

`Pictures` in 3DLDF are quite different from `pictures` in MetaPost. When a drawing or filling operation is applied to an object O, a copy of O, C, is allocated on the free store, a pointer to `Shape` S is pointed at C, and S is pushed onto the `vector<Shape*> shapes` on the `Picture` P, which was passed as an argument to the drawing or filling command. The arguments for the pen, dash pattern, `Color`, and any others, are used to set the corresponding data members of C (not O).

In order to actually cause MetaPost code to be written to the output file, it is necessary to invoke P`.output()`. Now, the appropriate version of `output()` is applied to each of the objects pointed to by a pointer on P`.shapes`. `output()` is a pure virtual function in `Shape`, so all classes derived from `Shape` must have an `output()` function. So, if `shapes[0]` points to a `Path`, `Path::output()` is called, if `shapes[1]` points to a `Point`, `Point::output()` is called, and if `shapes[2]` points to an object of a type derived from `Solid`, `Solid::output()` is called. `Point`, `Path`, and `Solid` are namely the only classes derived from `Shape` for which a version of `output()` is defined. All other `Shapes` are derived from one of these classes. These `output()` functions then write the MetaPost code to the output file through the output file stream `out_stream`.

```
beginfig(1);
default_focus.set(0, 0, -10, 0, 0, 10, 10);
Circle c(origin, 3, 90);
c.draw();
c.shift(1.5);
c.draw();
current_picture.output();
endfig(1);
```

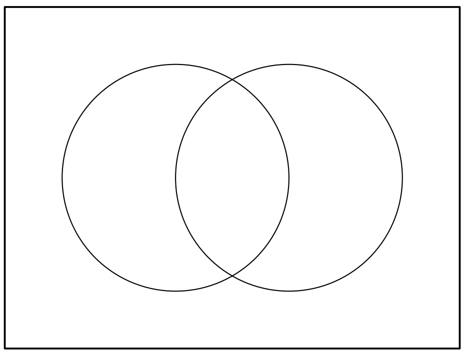


Figure 50.

The C++ code for Fig. 50 starts with the command `beginfig(1)` and ends with the command `endfig(1)`. They simply write "`beginfig(`⟨*arg*⟩`)`" and "`endfig()`" to `out_stream`, The optional `unsigned int` argument to `endfig()` is not written to `out_stream`, it's merely "syntactic sugar" for the user.

In MetaPost, the `endfig` command causes output and then clears `currentpicture`. This is not the case in 3DLDF, where `Picture::output()` and `Picture::clear()` must be invoked explicitly:

```
beginfig(1);
Point p0;
Point p1(1, 2, 3);
p0.draw(p1);
current_picture.output();
endfig(1);

beginfig(2);
current_picture.clear();
Circle C(origin, 3);
C.fill();
current_picture.output();
endfig(2);
```

In Fig. 51, two `Pictures` are used within a single figure.

```
beginfig(1);
Picture my_picture;
default_focus.set(0, 0, -10, 0, 0, 10, 10);
Circle c(origin, 3, 90);
c.draw(my_picture);
my_picture.output();
c.shift(1.5);
c.fill(light_gray);
current_picture.output();
endfig(1);
```

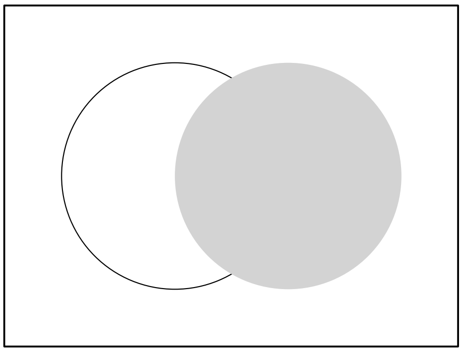


Figure 51.

Multiple objects, or complex objects made up of sub-objects, can be stored in a `Picture`, so that operations can be applied to them as a group:

```
default_focus.set(7, 5, -10, 7, 5, 10, 10);
Cuboid c0(origin, 5, 5, 5);
c0.shift(0, 0, 3);
c0.draw();
Circle z0(c0.get_rectangle_center(0), 2.5, 90, 0, 0, 64);
```

```
z0.draw();
Circle z1(z0);
z1.shift(0, 0, -1);
z1.draw();
int i;
int j = z0.get_size();
for (i = 0; i < 8; ++i)
    z0.get_point(i * j/8).draw(z1.get_point(i * j/8));
Cuboid c1(c0.get_rectangle_center(4), 5, 3, 3);
c1.shift(0, 2.5);
c1.draw();
Rectangle r0 = *c1.get_rectangle_ptr(3);
Point p[10];
for (i = 0; i < 4; ++i)
  p[i] = r0.get_point(i);
p[4] = r0.get_mid_point(0);
p[5] = r0.get_mid_point(2);
p[6] = p[4].mediate(p[5], 2/3.0);
Circle z2(p[6], 2, 90, 90, 0, 16);
z2.draw();
Circle z3 = z2;
z3.shift(3);
z3.draw();
j = z2.get_size();
for (i = 0; i < 8; ++i)
    z2.get_point(i * j/8).draw(z3.get_point(i * j/8));
p[7] = c0.get_rectangle_center(2);
p[7].shift(-4);
p[8] = c0.get_rectangle_center(3);
p[8].shift(4);
current_picture.output();
current_picture.rotate(45, 45);
current_picture.shift(10, 0, 3);
current_picture.output();
```