1. Copyright and License.
Copyright © 2003, 2004, 2005, 2006, 2007 The Free Software Foundation

See the section ⟨ GNU Free Documentation License 1561 ⟩ for the copying conditions that apply **to this document**.

The program GNU 3DLDF documented here is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 3 of the License, or (at your option) any later version. See the section ⟨ GNU General Public License 1562 ⟩ in this document.

GNU 3DLDF is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with GNU 3DLDF; if not, write to the Free Software Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

GNU 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. GNU 3DLDF is available for downloading from `http://www.gnu.org/software/3dldf/LDF.html`. It is also available from `http://www.dante.de/software/ctan/`, the Dante www-server and from `http://wwwuser.gwgd.de/~lfinsto1`, the author's website.

Please send bug reports to:
`lfinsto1@gwdg.de`

The mailing list `help-3DLDF@gnu.org` is available for people to ask other users for help. The mailing list `info-3DLDF@gnu.org` is for sending announcements to users. To subscribe to these mailing lists, send an email with "subscribe ⟨*email-address*⟩" as the subject.

The author can be contacted at:
Laurence D. Finston
c/o Free Software Foundation, Inc.
51 Franklin St, Fifth Floor
Boston, MA 02110-1301
USA

lfinsto1@gwdg.de
s246794@stud.uni-goettingen.de

Web site: `http:://wwwuser.gwdg.de/~lfinsto1`

2. Introduction (`3DLDF.web`).
This book contains the program code of 3DLDF, along with explanations. For information on *using* 3DLDF see the *3DLDF User and Reference Manual*, which should have been in the distribution along with this book.

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.

In the text sections of the `CWEB` documentation, I note things about C++ that may be of interest even if they are obvious to people with experience.

The various files that make up 3DLDF are ctangled and compiled separately (see the chapter "Compiling" in the *3DLDF User and Reference Manual*) and the resulting object files are then linked. However, the file `3DLDF.web`, which contains no C++ code and is not ctangled, includes the other `.web` files, so that cweave processes them as if they were all one file.

To write my `.web` files, I wrote a `cweb-mode` for Emacs and a number of Emacs-Lisp functions to go with it. It is not currently included in the 3DLDF distribution (Version 1.1.5.1), but I may include it in a later version. However, GNU is at work at an official cweb-mode of its own, so you might want to use it instead, if it's available.

Plurals of types are typeset with the "s" in the same font as the type, e.g., "**Points**" and not "**Point**s". It's not considered good typographical practice to typeset words with letters from different fonts. The second example does have the advantage that it's somewhat clearer what the actual name of the type is, but I think the first argument is weightier.

See `http://www-cs-faculty.stanford.edu/knuth/cweb.html` for more information about CWEB. The WEB (for Pascal) and CWEB packages are available from the CTAN archive, `ftp.dante.de` and `http:://www.dante.de`.

Donald Knuth's books *TEX: The Program* and *METAFONT: The Program* each include a section "How to read a WEB", which may be helpful.

Log

[LDF 2002.11.18.] Changed name of this file from `cweavedriver.web` to `cwdriver.web`. It now has fewer than 8 letters and can be used under DOS.

[LDF 2003.08.16.] Changed name of this file from `cwdriver.web` to `3DLDF.web` sometime since 2002.11.18. Forgot to note it here.

[LDF 2003.11.12.] Removed the version control identifiers from the CWEB files for the distribution of 3DLDF 1.1. They're still used in my development versions.

---

3. Formatting commands. `cweave` formats "`==`" as "$\equiv$", "`!=`" as "$\neq$", and "`!`" as "$\neg$". Programmers who use these tokens must type them as "`==`", "`!=`, and "`!`".

The following formatting commands are for types defined in C++ or in the C++ Standard Library, but not handled correctly by `cweave`.

format *bitset* *int*
format *bool* *int*
format *bools* *bool*
format *ifstream* *int*
format *key_type* *int*
format *mapped_type* *int*
format *map* *int*
format *ofstream* *int*
format *stat* *int*
format *stringstream* *int*
format *numeric_limits* *int*
format *pair* *int*
format *string* *char*
format *tm* *int*
format *valarray* *int*
format *vector* *int*

4. This section contains commands for inputting the CWEB source files, which are invisible in the `cweave` output.

5. Preprocessor variables and library files (`loader.web`). [LDF 2002.10.15.] It would, of course, be possible to put this code into a `.h` file directly, but it's convenient to have a CWEB file so that it can be `cweaved` along with the rest of 3DLDF.

[LDF 2003.07.18.] Set the preprocessor macro `LDF_GCC_3_3` to 1 in order to compile using gcc version 3.3 20030226 (prerelease) (SuSE Linux). Set it to 0 in order to compile using GCC version 2.95.3 20010315 (SuSE). This can be faster than using GCC 3.3, especially with respect to linking.

Log

[LDF 2003.08.21.] Now including `plfmvar.h`. It contains **#define** and **#undef** preprocessor commands for conditional compilation. There's a different version of this file in each of the subdirectories used for compiling with a different combination of operating system, compiler, and compiler version.

[LDF 2003.07.25.] Modified the conditional constructions governing compilation slightly.

[LDF 2003.08.14.] Now including `getopt.h` for the GCC versions under Linux. It's for processing the command line options.

[LDF 2003.08.14.] Now including `streambuf.h`, if `LDF_GCC_2_95` is defined, otherwise `ios`. This is for stream formatting.

[LDF 2003.08.29.] Removed `getopt.h` to `main.web`, because it's only used there.

[LDF 2003.09.03.] Added **#define** `LDF_PUBLIC` in order to be able to conditionally include `plfmvar.h`. The latter is not included in the version for distribution. Instead, the preprocessor variables are defined or undefined here.

[LDF 2003.11.12.] Removed the version control identifiers from the CWEB files for the distribution of 3DLDF 1.1. They're still used in my development versions.

⟨ Version control identifier 5 ⟩ ≡

```
static string rcs_id = "$Id: loader.web,v 1.4 2004/01/12 21:30:27 lfinsto1 Exp $";
```

See also sections 13, 53, 63, 76, 93, 164, 241, 250, 307, 635, 659, 696, 982, 1017, 1099, 1141, 1273, 1317, 1331, 1443, 1451, 1470, 1536, and 1543.

This code is used in sections 11, 51, 61, 74, 91, 162, 239, 248, 305, 633, 657, 694, 980, 1015, 1097, 1139, 1271, 1315, 1329, 1441, 1449, 1468, 1534, 1541, and 1558.

6. Configuration file. This section includes `config.h`, which is generated by `configure`. This is new in 3DLDF 1.1. The `configure` script generated by Autoconf tests whether certain library files are present, and defines preprocessor variables in `config.h` accordingly. These can be used for conditionally compiling code, so that library files are only included if they are really present. However, it will be necessary to add code for handling the case that they aren't present. I haven't done this yet, although I have put in conditional code using these variables in a couple of places. TO DO: Work on this. [LDF 2003.11.12.]

Autoconf does not per default check the version of the compiler that's used, and I'm not sure whether this would really be sensible. 3DLDF already contains conditionally compiled code based on whether the DEC C++ compiler, or the GNU C++ compiler (GCC) version 2.95 or version 3.3 is used. If the DEC compiler is used, the preprocessor variables `LDF_GCC_2_95` or `LDF_GCC_3_3` must be undefined by hand below. It defines `__DECCXX` itself. If GCC is used, one of them must be defined, and the other undefined. Per default, `LDF_GCC_3_3` is defined and `LDF_GCC_2_95` is undefined. This is because GCC 3.3 is, in general, an improvement over GCC 2.95. However, I usually use GCC 2.95 myself, because linking is significantly faster on the computer I use. [LDF 2003.11.12.]

Log

[LDF 2003.11.12.] Added this section.

[LDF 2003.12.17.] `config.h` is now not included if I'm with the DEC C++ compiler. This is because because building with Autoconf, etc., doesn't work on the DEC Alpha machine I'm using.

⟨ Include files 6 ⟩ ≡
```
#ifndef __DECCXX
#include "config.h"
#endif
```

See also sections 7, 8, 9, 14, 54, 64, 77, 94, 165, 242, 251, 308, 636, 660, 697, 983, 1018, 1100, 1142, 1274, 1318, 1332, 1444, 1452, 1471, 1537, and 1544.

This code is used in sections 12, 51, 61, 74, 91, 162, 239, 248, 305, 633, 657, 694, 980, 1015, 1097, 1139, 1271, 1315, 1329, 1441, 1449, 1468, 1534, 1541, and 1558.

7. Library files.

Log

[LDF 2003.12.17.] Changed the conditional in which `_GNU_SOURCE`, `LDF_GCC_3_3`, and `LDF_GCC_2_95` are defined or undefined. Working on compiling with the DEC C++ compiler.

⟨ Include files 6 ⟩ +≡
```
#define LDF_PUBLIC
#ifndef __DECCXX
#ifndef _GNU_SOURCE
#define _GNU_SOURCE
#endif
#define LDF_GCC_3_3
#undef LDF_GCC_2_95
#endif
#ifdef __DECCXX
#define __USE_STD_IOSTREAM   /* Needed for "sstream" below. */
#endif
#include <algorithm>
#include <deque>
#include <exception>
#ifdef __GNUC__
#if HAVE_FLOAT_H
#include <float.h>   /* [LDF 2002.12.13.] Needed for FLT_MAX. */
#endif
#endif
#include <fstream>
#include <functional>
#include <iomanip>
#ifdef LDF_GCC_2_95   /* For stream formatting. [LDF 2003.08.14.] */
#include <streambuf.h>
#else
#include <ios>
#endif
#include <iostream>
#include <iterator>
#include <math.h>
#ifdef LDF_GCC_3_3
#include <new>
#else
```

```
#ifdef LDF_GCC_2_95
#include <new.h>
#endif
#endif
#include <sstream>
#include <stdarg.h>
#include <stdexcept>
#ifdef __GNUC__
#include <stdio.h>
#endif
#ifdef __DECCXX
#include <stdlib.h>
#elif HAVE_STDLIB_H
#include <stdlib.h>
#endif
```

8. `streambuf.h` is included above, if `LDF_GCC_2_95` is defined, instead of `ios`, which is included in all other cases. [LDF 2003.08.14.]

⟨ Include files 6 ⟩ +≡

```
#include <string>
#include <valarray>
#include <vector>
```

9.

Log

[LDF 2003.07.18.] Added "**using namespace** *std*". This is needed with GCC 3.3, but not with GCC 2.95 or the DEC C++ compiler.

⟨ Include files 6 ⟩ +≡

```
using namespace std;
```

10. Putting `loader` together.

11. This is what's compiled. It's not necessary to include `stdlib.h` when using GCC 3.3, but I think it's safer. [LDF 2004.01.06.]

Log

[LDF 2004.01.06.] Added this section. It simplifies the rules for building the executable `3dldf` if `loader.c` is compiled.

```
#ifdef __DECCXX
#include <stdlib.h>
#elif HAVE_STDLIB_H
#include <stdlib.h>
#endif
#include <string>
  using namespace std;
  ⟨ Version control identifier 5 ⟩
```

12. This is what's written to the `loader.h`.

Log

[LDF 2004.01.06.] Added this section.

⟨ `loader.h` 12 ⟩ ≡
⟨ Include files 6 ⟩

13. Global items (`pspglb.web`). Typedefs, global variables and constants, and some non-class functions. (`pspglb.web`.)

Log

[LDF 2003.11.12.] Removed the version control identifiers from the CWEB files for the distribution of 3DLDF 1.1.1. They're still used in my development versions.

[LDF 2003.12.10.] Put the version control identifiers back into my release versions for 3DLDF 1.1.4. I've already put some of them back in, now I'm doing the rest of them. However, the release versions are now in their own RCS repository.

⟨ Version control identifier 5 ⟩ +≡
static string *rcs_id* = `"$Id: pspglb.web,v 1.14 2004/01/16 16:30:41 lfinsto1 Exp $"`;

14. Include files.

⟨ Include files 6 ⟩ +≡

```
#include "loader.h"
#ifdef __DECCXX
#include <limits>
#else
#ifdef LDF_GCC_3_3
#include <limits>
#else
#ifdef LDF_GCC_2_95
#if HAVE_LIMITS_H
#include <limits.h>
#endif
#endif
#endif
#endif
#include <bitset>
```

15. Type definitions. [LDF 2002.10.15.] Currently, all floating point variables are declared as **reals**. I've defined **real** in order to make it easy to switch between using *floats* and *doubles* simply by changing the value of the **#if** expression.

I try to avoid using preprocessor commands (see Introduction), but this is one of the cases where there's no better alternative to using the preprocessor (I don't consider commenting out the unwanted version preferable to using the preprocessor).

Log

[LDF 2002.04.10.] Added formatting commands.

[LDF 2002.04.10.] Added declaration of **bool_real**.

[LDF 2002.12.11.] Added the macros `LDF_REAL_FLOAT` and `LDF_REAL_DOUBLE`. They're needed below, where `MAX_REAL` and `INVALID_REAL` are declared in the GNU/Linux version (using GCC).

[LDF 2003.06.03.] Added *real_short*. It's the return type of **Plane** :: *get_distance*( ).

[LDF 2003.12.30.] Now using # **define** instead of @*d* for the definitions of `LDF_REAL_FLOAT` and `LDF_REAL_DOUBLE`. If @*d* is used, the definitions are only written to `pspglb.c`. Using # **define**, and writing ⟨ Type definitions 15 ⟩ to `pspglb.h` makes the definitions available to the files that include `pspglb.h`.

```
format real float
format real_pair real
format bool_pair real_pair
format real_pair real_short
format Matrix int
```

⟨ Type definitions 15 ⟩ ≡

```
#define LDF_REAL_FLOAT 1
#define LDF_REAL_DOUBLE 0
#if LDF_REAL_FLOAT
  typedef float real;
#elif LDF_REAL_DOUBLE
  typedef double real;
#else    /* Default. LDF 2003.12.17. */
  typedef float real;
#endif
  typedef real Matrix[4][4];
  typedef pair<real, real> real_pair;
  typedef pair<real, signed short> real_short;
  typedef pair<bool, bool> bool_pair;
  typedef pair<bool, real> bool_real;
```

See also sections 312, 313, 315, and 317.

This code is cited in section 15.

This code is used in sections 51, 52, 633, and 634.

16. Utility classes.

⟨ Utility classes 16 ⟩ ≡

```
  struct real_triple {
    real first;
    real second;
    real third;
    real_triple()
    : first(0), second(0), third(0) { }
    real_triple(real a, real b, real c)
    : first(a), second(b), third(c) { }
  };
```

This code is used in sections 51 and 52.

17. Global variables.

18. For compilation. [LDF 2003.08.25.] GCC 2.95 doesn't have the **numeric_limits** template, and GCC 3.3 doesn't seem to have it either.

Log

[LDF 2002.12.11.] BUG FIX: Discovered that the way this was before, `MAX_REAL` = `INVALID_REAL` − *real_limits.epsilon*( ) caused `MAX_REAL` and `INVALID_REAL` to be equal! I didn't notice the problem until

I started to port 3DLDF to GNU/Linux. It also doesn't work to use MAX_REAL = INVALID_REAL − *real_limits.min*( ).

⟨ Global variables 18 ⟩ ≡
valarray ⟨**real**⟩ *null_coordinates*(4);

See also sections 19, 20, 78, 234, 302, and 630.

This code is cited in section 25.

This code is used in sections 51, 91, 239, 305, and 633.

19. MAX_REAL is the second largest **real** value. MAX_REAL_SQRT is convenient to have for testing when computing distances.

!! KLUDGE: Using the macros FLT_MAX or DBL_MAX because the **numeric_limits** template doesn't seem to be available under GNU/Linux using GCC, at least not on the computer I'm using. [LDF 2002.12.11.]

Log

[LDF 2003.12.08.] Changed the definition of MAX_REAL. Previously, it was calculated using .00000003 ∗ FLT_MAX, which was a kludge.

[LDF 2003.12.29.] Changed the way MAX_REAL and MAX_REAL_SQRT are declared. They can no longer be **const**, because the value of MAX_REAL is set at the beginning of *main*( ) using *get_second_largest* < *Real* > ( ). The value of MAX_REAL_SQRT is set after this. Their values should never change after this!

MAX_REAL_SQRT must be initialized here, because it's used in **Point** :: *magnitude*( ). [LDF 2003.12.29.]

⟨ Global variables 18 ⟩ +≡
```
#ifdef __DECCXX
  numeric_limits<real> real_limits;
  extern const real INVALID_REAL = real_limits.max();
#else
#if LDF_REAL_DOUBLE
  extern const real INVALID_REAL = DBL_MAX;
#else    /* LDF_REAL_FLOAT, or not specified. LDF 2003.12.08. */
  extern const real INVALID_REAL = FLT_MAX;
#endif
#endif
  real MAX_REAL = 0;
  real MAX_REAL_SQRT = 0;
```

20. [LDF 2003.08.14.] VERBOSE_GLOBAL is *false* by default. It is set to *true* by the command line option "--verbose". If VERBOSE_GLOBAL is *true*, the local *verbose* variables in functions are set to *true*.

Log

[LDF 2003.08.14.] Added VERBOSE_GLOBAL and SILENT_GLOBAL.

⟨ Global variables 18 ⟩ +≡
```
  bool VERBOSE_GLOBAL = false;
  bool SILENT_GLOBAL = false;
```

21.

⟨ Declarations for the header file 21 ⟩ ≡

```
extern bool VERBOSE_GLOBAL;
extern bool SILENT_GLOBAL;
extern const bool ldf_real_float;
extern const bool ldf_real_double;
extern real MAX_REAL;
extern real MAX_REAL_SQRT;
```

See also sections 23, 26, 29, 235, 237, 303, 320, 631, 655, and 692.

This code is cited in section 24.

This code is used in sections 52, 240, 306, 634, 658, and 695.

22.

Log

[LDF 2003.08.14.] Added VERSION_3DLDF and COPYRIGHT_3DLDF.

⟨ Global constants 22 ⟩ ≡

```
#if LDF_REAL_FLOAT
  extern const bool ldf_real_float = 1;
  extern const bool ldf_real_double = 0;
#elif LDF_REAL_DOUBLE
  extern const bool ldf_real_float = 0;
  extern const bool ldf_real_double = 1;
#else     /* Defaults. LDF 2003.12.17. */
  extern const bool ldf_real_float = 1;
  extern const bool ldf_real_double = 0;
#endif
  extern const string VERSION_3DLDF = "1.1.5.1";
  extern const string COPYRIGHT_3DLDF = "Copyright␣(C)␣2003,␣2004␣by␣Laurence␣D.␣Finston.";
  extern const string DISCLAIMER_3DLDF = "3DLDF␣comes␣with␣ABSOLUTELY␣NO␣WARRANTY\
      ;\nfor␣details␣see␣the␣file␣COPYING,\nwhich␣you␣should␣have␣received\nin␣the␣dis\
      tribution␣of␣3DLDF␣1.1.5.1\nThis␣is␣free␣software,␣and␣you␣are␣welcome\nto␣redis\
      tribute␣it␣under␣certain␣conditions;\nfor␣details,␣again,␣see␣the␣file␣COPYING.\\
      n\nPlease␣send␣bug␣reports␣to␣the␣author␣at:\n\nEmail:␣␣␣␣lfinsto1@gwdg.de\n␣␣␣␣\
      ␣␣␣␣␣␣or␣s246794@stud.uni-goettingen.de\nWeb␣site:␣http://wwwuser.gwdg.de/~lfinsto1";
```

See also sections 28, 159, 236, and 319.

This code is used in sections 51, 162, 239, and 633.

23.

Log

[LDF 2003.11.28.] Changed VERSION_3DLDF from a **real** to a **string**. This is necessary, because I now have versions with three digits separated by periods.

⟨ Declarations for the header file 21 ⟩ +≡

```
extern const string VERSION_3DLDF;
extern const string COPYRIGHT_3DLDF;
extern const string DISCLAIMER_3DLDF;
extern const bool ldf_real_float;
extern const bool ldf_real_double;
```

24. TO DO: Find out why the library version of *trunc*( ) can't be found in the version for GCC 2.95 under Linux! [LDF 2002.12.10.]

The problem doesn't exist for GCC 3.3 under Linux. [LDF 2003.08.14.]

Log

[LDF 2003.08.14.] Put this function declaration in ⟨ Declare utility functions 24 ⟩. Formerly, it was in ⟨ Declarations for the header file 21 ⟩.

Changed the conditional from **#ifdef** `__GNUC__` to **#ifdef** `LDF_GCC_2_95`, because the library version of *trunc*( ) is found when compiling with GCC 3.3 under Linux.

⟨ Declare utility functions 24 ⟩ ≡
#ifdef `LDF_GCC_2_95`
  double *trunc*(**double** *d*);
#endif

See also section 31.
This code is cited in section 24.
This code is used in section 52.

25.

Log

[LDF 2003.08.14.] Put this function definition into ⟨ Define utility functions 25 ⟩. Formerly, it was in ⟨ Global variables 18 ⟩.

```
⟨ Define utility functions 25 ⟩ ≡
#ifdef LDF_GCC_2_95    /* KLUDGE!! [LDF 2002.12.1.] trunc( ) isn't available on the Linux machine
        gwdg-wb02.gwdg.de! Find out why not! */
  double trunc(double d)
  {
    int i;
    i = static_cast⟨int⟩(d);
    return static_cast⟨double⟩(i);
  }
#endif
```

See also section 32.

This code is cited in section 25.

This code is used in section 51.

26. For the header file.

```
⟨ Declarations for the header file 21 ⟩ +≡
  extern valarray⟨real⟩ null_coordinates;
#ifdef __DECCXX
  extern numeric_limits⟨real⟩ real_limits;
#endif
```

27. Global constants. INVALID_REAL is the largest possible **real** value, where **real** is either a synonym for **float** or for **double**, depending on how it's defined. Values are set to INVALID_REAL or functions return it when something has gone wrong. INVALID_REAL is also used for the **real** values in INVALID_TRANSFORM and INVALID_POINT. Another possibility would be to use exception handling, but so far I've found it convenient to use INVALID_REAL instead. Since the largest **float** is so large, and *epsilon*( ) for *floats* is so small, the loss of the largest possible valid value is insignificant. Using exception handling has its advantages, and if it turns out to be useful, I'll put in exception handling code, but using an otherwise valid value to signal exceptional conditions or errors does have the advantage of simplifying the path of execution through the program code. [LDF 2002.10.16.] Modified [LDF 2002.10.20.]

Log

[LDF 2002.09.25.] Added this section. Previously, I declared and initialized my global constants in the header file. This meant that each compilation unit that loaded pspglb.h had its own version of PI, INVALID_REAL, etc. I didn't know that *consts* had internal linkage by default and that I could make their linkage external by using **extern** in the declaration with the initialization, and put a second declaration, also with **extern**, in the header file. This is what I've done now.

[LDF 2003.06.03.] Added INVALID_REAL_SHORT.

---

28. For compilation.

```
⟨ Global constants 22 ⟩ +≡
  extern const real PI = 4.0 * atan(1.0);
  extern const real_pair INVALID_REAL_PAIR(INVALID_REAL, INVALID_REAL);
  extern const real_short INVALID_REAL_SHORT(INVALID_REAL, 0);
```

29. For the header file.

⟨ Declarations for the header file 21 ⟩ +≡
```
extern const real PI;
extern const real INVALID_REAL;
extern const real_pair INVALID_REAL_PAIR;
extern const real_short INVALID_REAL_SHORT;
```

30. Utility functions.

31. Solve quadratic equation. [LDF 2002.09.03.] TO DO: Maybe add functions for solving cubic and quartic equations, if this is practicable.

Log

[LDF 2003.06.1.] Changed return type from **pair**⟨**real**, **real**⟩ to **real_pair**, which is equivalent.

⟨ Declare utility functions 24 ⟩ +≡
```
real_pair solve_quadratic(real a, real b, real c);
```

32.

⟨ Define utility functions 25 ⟩ +≡

```
real_pair solve_quadratic(real a, real b, real c)
{
  real_pair p;
  try {
    p.first = (-b + sqrt((b * b) - (4 * a * c)))/(2 * a);
  }
  catch(...)
  {
    p.first = INVALID_REAL;
  }
  try {
    p.second = (-b - sqrt((b * b) - (4 * a * c)))/(2 * a);
  }
  catch(...)
  {
    p.second = INVALID_REAL;
  }
  return p;
}
```

33. System information.

Log

[LDF 2003.12.29.] Added this section.

---

34. Declare namespace System.

Log

[LDF 2003.12.29.] Added this section.

⟨ Declare **namespace System** 34 ⟩ ≡

```
namespace System {
  ⟨ Declare System functions 36 ⟩
}
```

See also section 65.

This code is used in sections 51, 52, 74, and 75.

35. Endianness.

Log

[LDF 2003.12.29.] Added this section.

---

36. Get endianness. *get_endianness*( ) returns the following values:
0 if the processor is little-endian.
1 if the processor is big-endian.
-1 if the endianness cannot be determined.

It is called by *is_little_endian*( ) and *is_big_endian*( ). [LDF 2003.12.21.]

This function has been adapted from Harbison, Samuel P., and Guy L. Steele Jr. *C, A Reference Manual*, pp. 163–164. [LDF 2003.12.29.]

Log

[LDF 2003.12.29.] Added this function.

⟨ Declare **System** functions 36 ⟩ ≡
```
signed short get_endianness(const bool verbose = false);
```
See also sections 38, 40, 43, 45, 47, 66, 71, and 72.

This code is used in sections 34, 52, 65, and 75.

37.

⟨ Define **System** functions 37 ⟩ ≡
```
signed short System::get_endianness(const bool verbose)
{
  union {
    long Long;
    char Char[sizeof(long)];
  } u;
  u.Long = 1;
  if (u.Char[0] ≡ 1) {
    if (verbose) cout ≪ "Processor␣is␣little-endian." ≪ endl ≪ endl ≪ flush;
    return 0;
  }
  else if (u.Char[sizeof(long) − 1] ≡ 1) {
    if (verbose) cout ≪ "Processor␣is␣big-endian." ≪ endl ≪ endl ≪ flush;
    return 1;
  }
  else {
    cerr ≪ "ERROR!␣In␣System::get_endianness():\n" ≪
        "Can't␣determine␣endianness.␣Returning␣-1" ≪ endl ≪ endl ≪ flush;
    return −1;
  }
}
```
See also sections 39, 41, 44, 46, 48, 67, and 68.

This code is used in sections 51, 74, and 75.

38. Is big endian.

Log

[LDF 2003.12.29.] Added this function.

⟨ Declare **System** functions 36 ⟩ +≡
```
bool is_big_endian(const bool verbose = false);
```

39.

⟨ Define **System** functions 37 ⟩ +≡

```
bool System::is_big_endian(const bool verbose)
{
  return (get_endianness(verbose) ≡ 1);
}
```

40. Is little endian.

Log

[LDF 2003.12.29.] Added this function.

⟨ Declare **System** functions 36 ⟩ +≡

```
bool is_little_endian(const bool verbose = false);
```

41.

⟨ Define **System** functions 37 ⟩ +≡

```
bool System::is_little_endian(const bool verbose)
{
  return (get_endianness(verbose) ≡ 0);
}
```

42. Register width.

Log

[LDF 2003.12.29.] Added this section.

43. Get register width.

Log

[LDF 2003.12.29.] Added this function.
[LDF 2004.1.2.] Changed the name of this function from *get_processor_size*( ) to *get_register_width*( ).

⟨ Declare **System** functions 36 ⟩ +≡

```
unsigned short get_register_width( );
```

44.

⟨ Define **System** functions 37 ⟩ +≡

```
unsigned short System::get_register_width( )
{
  return (sizeof(void *) * CHAR_BIT);
}
```

45. Is 32 bit.

Log

[LDF 2003.12.29.] Added this function.

⟨ Declare **System** functions 36 ⟩ +≡

```
bool is_32_bit( );
```

46.

⟨ Define **System** functions 37 ⟩ +≡

```
bool System::is_32_bit()
{
  return (get_register_width() ≡ 32);
}
```

47. Is 64 bit.

Log

[LDF 2003.12.29.] Added this function.

⟨ Declare **System** functions 36 ⟩ +≡

```
bool is_64_bit();
```

48.

⟨ Define **System** functions 37 ⟩ +≡

```
bool System::is_64_bit()
{
  return (get_register_width() ≡ 64);
}
```

49. Forward declarations. [LDF 2002.10.16.] In the files that are compiled first, some classes refer to other classes that haven't been defined yet. Forward declarations make it possible to do this. TO DO: GET CITATION from Stroustrup.

Log

[LDF 2002.04.10.] Added the forward declaration of **bool_real_point**. It's needed because it's used as the return value of **Point**::*intersection_point*( ), which is, of course, declared within the declaration of **class Point**. However, **bool_real_point** can only be defined *after* **Point** is defined. This forward declaration solves the problem.

[LDF 2003.07.16.] Added forward declaration of **Ellipse**. It's needed, because I've declared **Ellipse** to be a **friend** of **Path**. Formerly, **Circle** was a **friend** of **Path**, but now it must be **Ellipse**, because I've made the "segment" functions *segment*( ), *half*( ), and *quarter*( ) members of **Ellipse** instead of **Circle**.

⟨ Forward declarations 49 ⟩ ≡

```
struct bool_point;
struct bool_real_point;
class Circle;
class Ellipse;
struct Focus;
struct Line;
class Path;
class Picture;
struct Plane;
class Point;
```

This code is used in sections 51 and 52.

50. Putting `pspglb` together.

51. This is what's compiled.

⟨ Include files 6 ⟩
⟨ Version control identifier 5 ⟩
⟨ Type definitions 15 ⟩
⟨ Utility classes 16 ⟩
⟨ Global variables 18 ⟩
⟨ Global constants 22 ⟩
⟨ Define utility functions 25 ⟩
⟨ Declare **namespace System** 34 ⟩
⟨ Define **System** functions 37 ⟩
⟨ Forward declarations 49 ⟩

52. This is what's written to the `pspglb.h`.

⟨ `pspglb.h` 52 ⟩ ≡
⟨ Type definitions 15 ⟩
⟨ Utility classes 16 ⟩
⟨ Declarations for the header file 21 ⟩
⟨ Declare **namespace System** 34 ⟩
/∗ This doesn't work, apparently because it's incompatible with the use of `sstream`. ∗/
#if 0
#ifdef `__DECCXX` /∗ Using the DEC C++ Compiler. ∗/
const real `PI` = `__CXXL_PI`;
#endif
#endif
⟨ Declare utility functions 24 ⟩
⟨ Forward declarations 49 ⟩
⟨ Declare **System** functions 36 ⟩

53. Dynamic allocation for Shapes.

Log

[LDF 2003.12.29.] Added this template function.
[LDF 2004.1.2.] Moved the code in this file from `pspglb.web` to `creatnew.web`.

⟨ Version control identifier 5 ⟩ +≡
static string *rcs_id* = "$Id:␣creatnew.web,v␣1.4␣2004/01/12␣21:27:44␣lfinsto1␣Exp␣$";

54. Include files.

⟨ Include files 6 ⟩ +≡
#include "loader.h"
#include "pspglb.h"

55. Dynamic allocation for Shapes.

56. Pointer argument.

⟨ Declare *create_new*( ) 56 ⟩ ≡
template⟨**class** *C*⟩ *C*∗*create_new*(**const** *C*∗*arg*);

See also section 58.

57.

⟨ Define *create_new*( ) 57 ⟩ ≡

```
template<class C> C*create_new(const C*arg)
{
  C * obj = new(C);
  obj->set_on_free_store();
  if (arg ≠ 0) *obj = *arg;
  return obj;
}
```

See also section 59.

This code is used in sections 61 and 62.

58. Reference argument.

⟨ Declare *create_new*( ) 56 ⟩ +≡

```
template<class C> C*create_new(const C&arg);
```

59.

⟨ Define *create_new*( ) 57 ⟩ +≡

```
template<class C> C*create_new(const C&arg)
{
  C * obj = new(C);
  obj->set_on_free_store();
  *obj = arg;
  return obj;
}
```

60. Putting `creatnew` together.

61. This is what's compiled. I don't really need to compile the definition of *create_new*( ) here, because it must be included in all of the files that instantiate it, anyway. However, that may become unnecessary later, in which case it will have to be compiled here. In addition, if there's something wrong with the definition, it may be helpful to catch the error here. [LDF 2004.1.2.]

⟨ Include files 6 ⟩
⟨ Version control identifier 5 ⟩
⟨ Define *create_new*( ) 57 ⟩

62. This is what's written to the `creatnew.h`. The file `creatnew.h` must be included by all files that define specializations of *create_new*( ). [LDF 2003.12.29.]

⟨ `creatnew.h` 62 ⟩ ≡
  ⟨ Define *create_new*( ) 57 ⟩

63. Get second-largest real value.

⟨ Version control identifier 5 ⟩ +≡
  static string *rcs_id* = `"$Id:␣gsltmplt.web,v␣1.4␣2004/01/12␣21:29:56␣lfinsto1␣Exp␣$"`;

64. Include files.

⟨ Include files 6 ⟩ +≡
#include `"loader.h"`
#include `"pspglb.h"`

65. Declare namespace System.

Log

[LDF 2004.1.2.] Added this section.

⟨ Declare **namespace System** 34 ⟩ +≡
  namespace System {
    ⟨ Declare **System** functions 36 ⟩
  }

66. Get second largest. This function calculates the second-largest **real** value. It should be called using **float** or **double** as a parameter, e.g., *get_second_largest* < **float** > (`FLT_MAX`) or *get_second_largest* < **double** > (`DBL_MAX`). `FLT_MAX` or `DBL_MAX` must be passed as an argument. On systems with the **numeric_limits** template, *real_limits.max*( ) could be used instead. [LDF 2003.12.29.]

get_second_largest( ) determines which unsigned integral type has the same size as the template parameter *Real*. The locally declared type *i_type* is defined to be a synonym for this type using **typedef**. *ip* is a pointer to *i_type*. It is assigned a value by casting a pointer to `MAX_VAL` to the type of *ip*, i.e., a pointer to *i_type*. Then, 1 is subtracted from ∗*ip*, and the *ip* is cast back to a pointer to *Real*. This is the second largest *Real* value. [LDF 2003.12.29.]

This algorithm works on all of the machines I've tested. It doesn't matter whether they are big or little-endian, whether they have 32 or 64-bit processors. If the exponent of a floating point type is stored in its low order byte or bytes, then this will fail. I haven't run into this problem yet, though. The commented-out code in ⟨ Loop for testing bits 70 ⟩ may help in finding the second-largest *Real* value in this case. [LDF 2003.12.29.]

Log

[LDF 2003.12.29.] Added this function.
[LDF 2004.1.2.] Moved this section from `pspglb.web` to `creatnew.web`.

⟨ Declare **System** functions 36 ⟩ +≡
  template⟨**class** *Real*⟩ *Real get_second_largest*(*Real*`MAX_VAL`, **bool** *verbose* = *false*);
  float *get_second_largest*(**float**, **bool**);
  double *get_second_largest*(**double**, **bool**);

67.

⟨ Define **System** functions 37 ⟩ +≡

68.

⟨ Define **System** functions 37 ⟩ +≡

```
template⟨class Real⟩Real System::get_second_largest(RealMAX_VAL, bool verbose)
{
  const unsigned short USHORT_SIZE = sizeof(unsigned short);
  const unsigned short UINT_SIZE = sizeof(unsigned int);
  const unsigned short ULONG_SIZE = sizeof(unsigned long);
  const unsigned short ULONG_LONG_SIZE = sizeof(unsigned long long);
  const unsigned short Real_SIZE = sizeof (Real);
  const unsigned short FLT_SIZE = sizeof(float);
  const unsigned short DBL_SIZE = sizeof(double);
  const unsigned short LONG_DBL_SIZE = sizeof(long double);
  const bool Real_EQ_USHORT = (Real_SIZE ≡ USHORT_SIZE);
  const bool Real_EQ_UINT = (Real_SIZE ≡ UINT_SIZE);
  const bool Real_EQ_ULONG = (Real_SIZE ≡ ULONG_SIZE);
  const bool Real_EQ_ULONG_LONG = (Real_SIZE ≡ ULONG_LONG_SIZE);
  if (verbose) {
    cout ≪ "USHORT_SIZE␣␣==␣" ≪ USHORT_SIZE ≪ endl ≪ flush;
    cout ≪ "UINT_SIZE␣␣==␣" ≪ UINT_SIZE ≪ endl ≪ flush;
    cout ≪ "ULONG_SIZE␣==␣" ≪ ULONG_SIZE ≪ endl ≪ flush;
    cout ≪ "ULONG_LONG_SIZE␣==␣" ≪ ULONG_LONG_SIZE ≪ endl ≪ flush;
    cout ≪ "FLT_SIZE␣␣==␣" ≪ FLT_SIZE ≪ endl ≪ flush;
    cout ≪ "DBL_SIZE␣␣==␣" ≪ DBL_SIZE ≪ endl ≪ flush;
    cout ≪ "LONG_DBL_SIZE␣␣==␣" ≪ LONG_DBL_SIZE ≪ endl ≪ flush;
    cout ≪ "Real_SIZE␣␣==␣" ≪ Real_SIZE ≪ endl ≪ flush;
  }
  Real *rp;
  if (Real_EQ_USHORT) {
    if (verbose) cout ≪ "Real_EQ_USHORT\n";
    typedef unsigned short i_type;
    ⟨ Calculate second-largest Real 69 ⟩
  }
  else if (Real_EQ_UINT) {
    if (verbose) cout ≪ "Real_EQ_UINT␣\n";
    typedef unsigned int i_type;
    ⟨ Calculate second-largest Real 69 ⟩
  }
  else if (Real_EQ_ULONG) {
    if (verbose) cout ≪ "Real_EQ_ULONG␣\n";
    typedef unsigned long i_type;
    ⟨ Calculate second-largest Real 69 ⟩
  }
  else if (Real_EQ_ULONG_LONG) {
    if (verbose) cout ≪ "Real_EQ_ULONG_LONG\n";
    typedef unsigned long long i_type;
    ⟨ Calculate second-largest Real 69 ⟩
  }
  else {
```

```
      cerr ≪ "ERROR!␣In␣main():\n" ≪ "Apparently,␣Real␣doesn't␣have␣the␣same␣size␣" ≪
          "as␣any␣unsigned␣integral␣type.\n" ≪ "There␣must␣be␣some␣mistake.\n" ≪
          "Exiting␣with␣return␣value␣-1.\n\n" ≪ flush;
      return −1;
    }
    if (verbose) {
      cout.precision(25);
      cout ≪ "MAX_VAL␣==␣" ≪ MAX_VAL ≪ endl ≪ "*rp␣==␣" ≪ *rp ≪ endl ≪
          "(MAX_VAL␣==␣*rp)␣==␣" ≪ (MAX_VAL ≡ *rp) ≪ endl ≪ "(MAX_VAL␣-␣*rp)␣==␣" ≪
          (MAX_VAL − *rp) ≪ endl ≪ "MAX_VAL␣>␣*rp␣==␣" ≪ (MAX_VAL > *rp) ≪ endl ≪
          "MAX_VAL␣<␣*rp␣==␣" ≪ (MAX_VAL < *rp) ≪ endl ≪ flush;
    }
    if (MAX_VAL ≡ *rp) {
      cerr ≪ "ERROR!␣In␣System::get_second_largest<Real>():\n" ≪
          "MAX_VAL␣==␣*rp.␣Exiting␣with␣return␣value␣1" ≪ endl ≪ endl ≪ flush;
      exit(1);
    }
    else if (MAX_VAL < *rp) {
      cerr ≪ "ERROR!␣In␣System::get_second_largest<Real>():\n" ≪
          "MAX_VAL␣<␣*rp.␣Exiting␣with␣return␣value␣1" ≪ endl ≪ endl ≪ flush;
      exit(1);
    }
    return *rp;
  }
```

69. Calculate second-largest Real.

⟨ Calculate second-largest *Real* 69 ⟩ ≡

```
  { i_type *ip = reinterpret_cast⟨i_type *⟩(&MAX_VAL);
  if (verbose) cout ≪ "*ip␣==␣" ≪ *ip ≪ endl ≪ flush;
  i_type bit_pattern_i_type;
  i_type result;
  bit_pattern_i_type = 1;
  bitset⟨sizeof(i_type) * CHAR_BIT⟩ b;
  b = *ip;
  if (verbose) cout ≪ "b␣(MAX_VAL)␣==␣" ≪ b ≪ endl ≪ flush;
  b = bit_pattern_i_type;
  if (verbose) cout ≪ "b␣(bit_pattern_i_type)␣==␣" ≪ b ≪ endl ≪ flush;
  result = bit_pattern_i_type ⊕ *ip;
  if (verbose) cout ≪ "result␣==␣" ≪ result ≪ endl ≪ flush;
  b = result;
  if (verbose) cout ≪ "b␣(result)␣==␣" ≪ b ≪ endl ≪ flush;
  rp = reinterpret_cast ⟨ Real *> (&result);
  if (verbose) cout ≪ "*rp␣==␣" ≪ *rp ≪ endl ≪ flush;
  }
```

This code is used in section 68.

70. Loop for testing bits.

```
⟨ Loop for testing bits 70 ⟩ ≡
#if 0
  int counter; for (int i = 0; i < (sizeof (Real) ∗ CHAR_BIT); ++i) {
  if (verbose) cout ≪ "i␣==␣" ≪ i ≪ endl ≪ flush;   /∗ This has only been needed on the DEC Alpha,
        so far. ∗/   /∗ This is the case that the highest-order bit of the ∗/   /∗ mantissa is 1, and all
        of the other bits (in particular, ∗/   /∗ all the bits of the exponent) are 0. In this case, ∗rp ∗/
      /∗ is not a number (NAN). The GNU compiler copes with this, ∗/   /∗ the DEC compiler signals
        a floating point error and dumps ∗/   /∗ core (I believe). I wasn't able to catch the error with
        ∗/   /∗ try and catch. ∗/   /∗ START HERE. Change (8 + 1) to (FLT_EXP + 1), except ∗/
      /∗ FLT_EXP ∗/   /∗ isn't the right name. Find it, and put here. This ∗/
      /∗ assumes the exponent is at the left, which may not be ∗/   /∗ true. Skipping this bit pattern
        is necessary on the DEC ∗/   /∗ ALPHA, because the float is not a number. ∗/
      /∗ It must be 23 for float, and 52 for double. ∗/
  if (i ≡ (sizeof (Real) ∗ CHAR_BIT) − (12)) {
    if (verbose) cout ≪ "i␣==␣" ≪ i ≪ ".␣␣This␣produces␣NAN.␣␣Continuing.\n\n" ≪ flush;
    continue;
  }
  bit_pattern_i_type = 1;
  bit_pattern_i_type ≪= i;
  if (verbose)
    cout ≪ "bit_pattern_i_type␣(1␣<<␣" ≪ i ≪ ")␣==␣" ≪ bit_pattern_i_type ≪ endl ≪ flush;
  b = bit_pattern_i_type;
  if (verbose) cout ≪ "b␣(bit_pattern_i_type)␣==␣" ≪ b ≪ endl ≪ flush;
  result = bit_pattern_i_type ⊕ ∗ip;
  if (verbose) cout ≪ "result␣==␣" ≪ result ≪ endl ≪ flush;
  b = result;
  if (verbose) cout ≪ "b␣(result)␣==␣" ≪ b ≪ endl ≪ flush;
  rp = reinterpret_cast ⟨ Real ∗> (&result);
  if (verbose) cout ≪ "∗rp␣==␣" ≪ ∗rp ≪ endl ≪ flush;
  if (∗rp ≤ 0) {
    if (verbose) cout ≪ "∗rp␣<=␣0.\nContinuing.\n\n" ≪ flush;
    continue;
  }
  else if (∗rp ≥ MAX_VAL) {
    if (verbose) cout ≪ "∗rp␣>=␣␣MAX_VAL\nContinuing." ≪ endl ≪ flush;
    continue;
  }
  else if (second_largest_real ≥ ∗rp) {
    if (verbose) cout ≪ "second_largest_real␣>=␣∗rp\nContinuing." ≪ "\n" ≪ flush;
    continue;
  }
  else if (∗rp > second_largest_real ∧ ∗rp < MAX_VAL) {
    if (verbose) cout ≪ "∗rp␣>␣second_largest_real␣&&␣" ≪ "∗rp␣<␣MAX_VAL" ≪ endl ≪
          "Setting␣second_largest_real␣to␣∗rp" ≪ "␣and␣counter␣to␣i␣(" ≪ i ≪ ").\n" ≪ flush;
    second_largest_real = ∗rp;
    counter = i;
  }
  else {
    if (verbose) cout ≪ "Some␣other␣condition.␣Continuing.\n";
    continue;
  }
```

```
  }    /* for */
#endif
```

This code is cited in section 66.

71. Template function instantiations.

Log

[LDF 2003.12.29.] Added this section.

⟨ Declare **System** functions 36 ⟩ +≡
  float *get_second_largest*(**float** `MAX_VAL`, **bool** *verbose*);

72.

⟨ Declare **System** functions 36 ⟩ +≡
  double *get_second_largest*(**double** `MAX_VAL`, **bool** *verbose*);

73. Putting `gsltmplt` together.

74. This is what's compiled. I don't really need to compile the definition of *get_second_largest*( ) here, because it must be included in all of the files that instantiate it, anyway. However, that may become unnecessary later, in which case it will have to be compiled here. In addition, if there's something wrong with the definition, it may be helpful to catch the error here. [LDF 2004.1.2.]

⟨ Include files 6 ⟩
⟨ Version control identifier 5 ⟩
⟨ Declare **namespace System** 34 ⟩
⟨ Define **System** functions 37 ⟩

75. This is what's written to the `gsltmplt.h`. The file `gsltmplt.h` must be included by all files that define specializations of *get_second_largest*( ). [LDF 2003.12.29.]

⟨ `gsltmplt.h` 75 ⟩ ≡
 ⟨ Declare **namespace System** 34 ⟩
 ⟨ Declare **System** functions 36 ⟩
 ⟨ Define **System** functions 37 ⟩

76. I/O (`io.web`).

Log

[LDF 2003.11.12.] Removed the version control identifiers from the CWEB files for the distribution of 3DLDF 1.1. They're still used in my development versions.

[LDF 2003.12.10.] Put the version control identifiers back into my release versions for 3DLDF 1.1.4. I've already put some of them back in, now I'm doing the rest of them. However, the release versions are now in their own RCS repository.

⟨ Version control identifier 5 ⟩ +≡
 static string *rcs_id* = `"$Id:␣io.web,v␣1.4␣2004/01/12␣21:30:03␣lfinsto1␣Exp␣$"`;

77. Include files.

⟨ Include files 6 ⟩ +≡
#**include** `"loader.h"`
#**include** `"pspglb.h"`
#**include** `<time.h>`

78. Global variables. [LDF 2002.10.16.] *in_stream* is an input stream attached to a file with user code for input. Currently, it is used, but it fulfills no useful function, because I haven't defined an input routine yet. *out_stream* is an output stream attached to the file of METAPOST code that 3DLDF currently produces as its output. *tex_stream* is an output stream attached to a file of TEX code. The user can write TEX code to this file and load it into `persp.tex` or use it for some other purpose. 3DLDF makes no use of it itself.

Log

[LDF 2002.08.30.] Added *tex_stream* so that I can include TEX code in my user code. Code written by 3DLDF to *tex_stream* will be loaded by `persp.tex`, or whatever TEX file includes the PostScript file generated by METAPOST from the output of 3DLDF. User code is currently in `main.web`. In production versions user code will be in `user.web`.

[LDF 2003.07.16.] Added *fig_num*.

---

 format *ifstream* *int*
 format *ofstream* *int*

⟨ Global variables 18 ⟩ +≡
 ifstream *in_stream*;
 ofstream *out_stream*;
 ofstream *tex_stream*;
 unsigned short *fig_num*;

79. **extern** declarations for the global variables.

⟨ **extern** variable declarations 79 ⟩ ≡
  extern ifstream *in_stream*;
  extern ofstream *out_stream*;
  extern ofstream *tex_stream*;
  extern unsigned short *fig_num*;

This code is used in section 92.

80. I/O functions.

81. Initialize I/O.

Log

[LDF 2003.08.29.] Changed, so that *in_stream* isn't opened.

⟨ Declare I/O functions 81 ⟩ ≡
  void *initialize_io*(**string** *in_stream_name*, **string** *out_stream_name*, **string** *tex_stream_name*, **char** ∗*program_name*);

See also sections 84, 86, and 88.

This code is used in section 92.

82.

⟨ Define I/O functions 82 ⟩ ≡
  void *initialize_io*(**string** *in_stream_name*, **string** *out_stream_name*, **string** *tex_stream_name*, **char** ∗*program_name*){ **time_t** *tt*;
    tm ∗*lt*;
    tt = *time*(0);
    lt = *localtime*(&*tt*);
    string *datestamp*(*asctime*(*lt*));
    datestamp.*erase*(*datestamp*.*size*( ) − 1);   /∗ Remove terminal line-feed. ∗/

See also sections 83, 85, 87, and 89.

This code is used in section 91.

83. Open *out_stream* and *tex_stream*. *in_stream* is currently not opened. [LDF 2003.08.29.]

```
⟨ Define I/O functions 82 ⟩ +≡
#if 0
  in_stream.open(in_stream_name.c_str());
#endif
  out_stream.open(out_stream_name.c_str());
#ifdef __DECCXX
  out_stream.setf(ios_base::fixed, ios_base::floatfield);
#else
#ifdef __GNUC__
  out_stream.setf(ios::fixed, ios::floatfield);
#endif
#endif
  tex_stream.open(tex_stream_name.c_str());    /* Write datestamp to out_stream. */
  out_stream << "%%%% This is " << out_stream_name << "." << endl << "%%%% Generated on " <<
      datestamp << " from " << program_name << ".\n\n";    /* Write datestamp to tex_stream. */
  tex_stream << "%%%% This is " << tex_stream_name << "." << endl << "%%%% Generated on " <<
      datestamp << " from " << program_name << ".\n\n"; }
```

84. Write footers. [LDF 2002.10.16.] Footers can be written to *outputstream* and *tex_stream*. I use them for Local Variables lists for Emacs. Other users may not want this, which is why this code is commented out here.

```
⟨ Declare I/O functions 81 ⟩ +≡
  void write_footers();
```

85.

⟨ Define I/O functions 82 ⟩ +≡

```
void write_footers()
{
#if 0
  out_stream << endl << endl << "%%␣L" << "ocal␣Variables:" << endl << "%%␣mode:Metafont" <<
      endl << "%%␣eval:(if␣metapost-keymap␣nil␣(load␣\"metapost-keymap\"))" <<
      endl << "%%␣eval:(use-local-map␣metapost-mode-map)" << endl <<
      "%%␣eval:(local-set-key␣[f9]␣'mp-file)" << endl << "%%␣run-mp-on-file:\"persp.mp\"" <<
      endl << "%%␣run-cweb-on-file:\"main.web\"" << endl <<
      "%%␣run-tex-on-file:\"persp.tex\"" << endl << "%%␣run-dvips-on-file:\"persp.ps\"" <<
      endl << "%%␣End:" << endl << endl;
  tex_stream << "\n\n%%␣L" << "ocal␣Variables:\n" << "%%␣mode:tex\n" <<
      "%%␣eval:(local-set-key␣[f9]␣'mp-file)\n" << "%%␣run-mp-on-file:\"persp.mp\"\n" <<
      "%%␣run-cweb-on-file:\"main.web\"\n" << "%%␣run-tex-on-file:\"persp.tex\"\n" <<
      "%%␣run-dvips-on-file:\"persp.ps\"\n" << "%%␣End:\n";
#endif
  return;
}
```

86. Begin figure.

Log

[LDF 2003.07.16.] Added *silent* argument, and a message printed conditionally to *stdout*, saying which figure is being started. This should help in finding where errors occur.

[LDF 2003.07.16.] Made non-inline.

[LDF 2003.08.17.] Made *silent* non-**const**. Setting it to *true*, if `SILENT_GLOBAL` is *true*.

⟨ Declare I/O functions 81 ⟩ +≡

```
void beginfig(unsigned short i, bool silent = false);
```

87.

⟨ Define I/O functions 82 ⟩ +≡

```
void beginfig(unsigned short i, bool silent)
{
  if (SILENT_GLOBAL) silent = true;
  fig_num = i;
  out_stream << "beginfig(" << fig_num << ");\n";
  if (¬silent) cout << "Beginning␣figure␣" << fig_num << "." << endl << flush;
  return;
}
```

88. End figure. The **unsigned short** argument is "syntactic sugar". It's ignored by *endfig*( ), but may be convenient for a user for keeping track of what figure is being ended.

Log

[LDF 2003.07.16.] Added *silent* argument, and a message printed conditionally to *stdout*, saying which figure is being ended. This should help in finding where errors occur.

[LDF 2003.07.16.] Made non-inline.

[LDF 2003.08.17.] Made *silent* non-**const**. Setting it to *true*, if SILENT_GLOBAL is *true*.

⟨ Declare I/O functions 81 ⟩ +≡

```
void endfig(unsigned short i = 0, bool silent = false);
```

89.

⟨ Define I/O functions 82 ⟩ +≡

```
void endfig(unsigned short i, bool silent)
{
  if (SILENT_GLOBAL) silent = true;
  out_stream << "endfig" << ";\n";
  if (¬silent) cout << "Ending␣figure␣" << fig_num << "." << endl << endl << flush;
  return;
}
```

90. Putting I/O together.

91. This is what's compiled.

⟨ Include files 6 ⟩
⟨ Version control identifier 5 ⟩
⟨ Global variables 18 ⟩
⟨ Define I/O functions 82 ⟩

92. This is what's written to `io.h`.

⟨ `io.h` 92 ⟩ ≡
  ⟨ **extern** variable declarations 79 ⟩
  ⟨ Declare I/O functions 81 ⟩

93. Color (`colors.web`).

Log

[LDF 2003.11.12.] Removed the version control identifiers from the CWEB files for the distribution of 3DLDF 1.1. They're still used in my development versions.

[LDF 2003.12.10.] Put the version control identifiers back into my release versions for 3DLDF 1.1.4. I've already put some of them back in, now I'm doing the rest of them. However, the release versions are now in their own RCS repository.

⟨ Version control identifier 5 ⟩ +≡
  static string *rcs_id* = `"$Id: colors.web,v 1.7 2004/01/12 21:27:38 lfinsto1 Exp $"`;

94. Include files.

⟨ Include files 6 ⟩ +≡
#**include** `"loader.h"`
#**include** `"pspglb.h"`
#**include** `"creatnew.h"`
#**include** `"io.h"`

95. Color class definition. [LDF 2002.09.25.] !! Remember to change the constructors, setting functions, and assignment operator if I add or change anything here!!

Log

[LDF 2002.10.06.] Added. *on_free_store*.

⟨ Define **class Color** 95 ⟩ ≡
  class Color {
    string *name*;
    bool *use_name*;
    bool *on_free_store*;
    real *red_part*;
    real *green_part*;
    real *blue_part*;
  public: ⟨ Declare **Color** functions 97 ⟩
  };

This code is used in sections 162 and 163.

96. Constructors and setting functions.

97. Default constructor. [LDF 2002.10.06.] Added code to definition. Previously, it was empty.

⟨ Declare **Color** functions 97 ⟩ ≡
  Color( );

See also sections 99, 102, 104, 107, 109, 114, 116, 118, 121, 123, 125, 127, 129, 131, 133, 135, 138, 141, 142, 143, 144, 146, 149, and 151.

This code is used in section 95.

98.

⟨ Define **Color** functions 98 ⟩ ≡

```
Color::Color()
{
   red_part = green_part = blue_part = 0.0;
   name = "";
   use_name = false;
   on_free_store = false;
}
```

See also sections 100, 103, 105, 108, 110, 115, 117, 119, 122, 124, 126, 128, 130, 132, 134, 136, 139, 145, 150, and 152.
This code is used in section 162.

99. Copy constructor. !! Remember to add or change code here if I add or change anything in the **class** definition!!

Log

[LDF 2002.09.25.] Added this function.

⟨ Declare **Color** functions 97 ⟩ +≡

```
Color(const Color &c, const string n = "", const bool u = true);
```

100.

⟨ Define **Color** functions 98 ⟩ +≡

```
Color::Color(const Color &c, const string n, const bool u)
{
   name = n;
   red_part = c.get_red_part();
   green_part = c.get_green_part();
   blue_part = c.get_blue_part();
   if (n ≠ "" ∧ u ≡ true) {
      use_name = true;
   }
   else use_name = false;
   on_free_store = false;     /* LDF 2002.10.06. Added. */
   return;
}
```

101. Name and unsigned short arguments.

102. Constructor.

⟨ Declare **Color** functions 97 ⟩ +≡

```
Color(const string n, const unsigned short r, const unsigned short g, const unsigned short
      b, const bool u = true);
```

103.

⟨ Define **Color** functions 98 ⟩ +≡

Color :: **Color**(**const string** n, **const unsigned short** r, **const unsigned short** g, **const unsigned short** b, **const bool** u)
  : *name*(n) {
    name = n;
    if ($n \neq$ "" $\wedge u \equiv$ *true*) {
      use_name = *true*;
    }
    else *use_name* = *false*;
    on_free_store = *false*;   /∗ LDF 2002.10.06. Added. ∗/
    red_part = $r/255.0$;
    green_part = $g/255.0$;
    blue_part = $b/255.0$;
  }

104. Setting function.

⟨ Declare **Color** functions 97 ⟩ +≡

void *set*(**const string** n, **const unsigned short** r, **const unsigned short** g, **const unsigned short** b, **const bool** u = *false*);

105.

⟨ Define **Color** functions 98 ⟩ +≡

void Color :: *set*(**const string** n, **const unsigned short** r, **const unsigned short** g, **const unsigned short** b, **const bool** u)
  {
    name = n;
    if ($n \neq$ "" $\wedge u \equiv$ *true*) {
      use_name = *true*;
    }
    else *use_name* = *false*;
    red_part = $r/255.0$;
    green_part = $g/255.0$;
    blue_part = $b/255.0$;
  }

106. Three real arguments. [LDF 2002.10.09.] Added the following constructor and setting function. They are for unnamed **Colors**. The DEC compiler can't distinguish between **real** and **unsigned short** arguments, so the overloaded functions must differ in another way. In this case, these versions have no *name* argument. I believe that users are most likely to declare **Colors** using **real** arguments when they plan to modify them, in which case the *output*( ) function should write the red, green and blue values to *out_stream* rather than *name*. If it turns out to be necessary, more constructors can be added or the existing ones can be changed.

107. Constructor.

⟨ Declare **Color** functions 97 ⟩ +≡

Color(**const real** r, **const real** g, **const real** b);

108.

⟨ Define **Color** functions 98 ⟩ +≡

```
Color::Color(const real r, const real g, const real b)
{
  name = "";
  use_name = false;
  on_free_store = false;
  if (r < 0) {
    cerr << "WARNING! In Color::Color() (three real arguments):\n" <<
        "Red part argument < 0.  Setting red_part to 0.\n\n";
    red_part = 0;
  }
  else if (r > 1) {
    cerr << "WARNING! In Color::Color() (three real arguments):\n" <<
        "Red part argument > 1.  Setting red_part to 1.\n\n";
    red_part = 1;
  }
  else red_part = r;
  if (g < 0) {
    cerr << "WARNING! In Color::Color() (three real arguments):\n" <<
        "Green part argument < 0.  Setting green_part to 0.\n\n";
    green_part = 0;
  }
  else if (g > 1) {
    cerr << "WARNING! In Color::Color() (three real arguments):\n" <<
        "Green part argument > 1.  Setting green_part to 1.\n\n";
    green_part = 1;
  }
  else green_part = g;
  if (b < 0) {
    cerr << "WARNING! In Color::Color() (three real arguments):\n" <<
        "Blue part argument < 0.  Setting blue_part to 0.\n\n";
    blue_part = 0;
  }
  else if (b > 1) {
    cerr << "WARNING! In Color::Color() (three real arguments):\n" <<
        "Blue part argument > 1.  Setting blue_part to 1.\n\n";
    blue_part = 1;
  }
  else blue_part = b;
  return;
}
```

109. Setting function.

⟨ Declare **Color** functions 97 ⟩ +≡

```
void set(const real r, const real g, const real b);
```

110.

⟨ Define **Color** functions 98 ⟩ +≡

```
void Color::set(const real r, const real g, const real b)
{
  name = "";
  use_name = false;
  on_free_store = false;
  if (r < 0) {
    cerr << "WARNING!␣In␣Color::Color()␣(three␣real␣arguments):\n" <<
        "Red␣part␣argument␣<␣0.␣␣Setting␣red_part␣to␣0.\n\n";
    red_part = 0;
  }
  else if (r > 1) {
    cerr << "WARNING!␣In␣Color::Color()␣(three␣real␣arguments):\n" <<
        "Red␣part␣argument␣>␣1.␣␣Setting␣red_part␣to␣1.\n\n";
    red_part = 1;
  }
  else red_part = r;
  if (g < 0) {
    cerr << "WARNING!␣In␣Color::Color()␣(three␣real␣arguments):\n" <<
        "Green␣part␣argument␣<␣0.␣␣Setting␣green_part␣to␣0.\n\n";
    green_part = 0;
  }
  else if (g > 1) {
    cerr << "WARNING!␣In␣Color::Color()␣(three␣real␣arguments):\n" <<
        "Green␣part␣argument␣>␣1.␣␣Setting␣green_part␣to␣1.\n\n";
    green_part = 1;
  }
  else green_part = g;
  if (b < 0) {
    cerr << "WARNING!␣In␣Color::Color()␣(three␣real␣arguments):\n" <<
        "Blue␣part␣argument␣<␣0.␣␣Setting␣blue_part␣to␣0.\n\n";
    blue_part = 0;
  }
  else if (b > 1) {
    cerr << "WARNING!␣In␣Color::Color()␣(three␣real␣arguments):\n" <<
        "Blue␣part␣argument␣>␣1.␣␣Setting␣blue_part␣to␣1.\n\n";
    blue_part = 1;
  }
  else blue_part = b;
}
```

111. Pseudo-constructor for dynamic allocation.

Log

[LDF 2003.12.30.] Replaced **Color** :: *create_new_color* ( ) with specializations of **template**⟨**class** C⟩ C∗*create_new* ( ).

---

112. Pointer argument.

⟨ Declare non-member template functions for **Color** 112 ⟩ ≡

```
Color *create_new(const Color *c);
```

See also section 113.
This code is used in sections 162 and 163.

113. Reference argument.

Log

[LDF 2004.1.2.] Added this declaration.

⟨ Declare non-member template functions for **Color** 112 ⟩ +≡
  Color ∗*create_new*(**const Color** &*c*);

114. Assignment. [LDF 2002.09.24.] Added this operator function.

⟨ Declare **Color** functions 97 ⟩ +≡
  void operator=(**const Color** &*c*);

115.

⟨ Define **Color** functions 98 ⟩ +≡

```
void Color::operator=(const Color &c)
{
  name = "";
  use_name = false;
  red_part = c.red_part;
  green_part = c.green_part;
  blue_part = c.blue_part;
}
```

116. Equality. [LDF 2002.09.25.] Changed so that only *red_part*, *green_part* and *blue_part* are compared. This way, **Colors** that differ only in *name* and/or *use_name* are considered to be equal. [LDF 2002.09.24.] Added this operator function.

⟨ Declare **Color** functions 97 ⟩ +≡
  bool operator≡(**const Color** &*c*) **const**;

117.

⟨ Define **Color** functions 98 ⟩ +≡

```
bool Color::operator≡(const Color &c) const
{
  return ((red_part ≡ c.red_part) ∧ (green_part ≡ c.green_part) ∧ (blue_part ≡ c.blue_part));
}
```

118. Inequality. [LDF 2002.09.24.] Added this operator function.

⟨ Declare **Color** functions 97 ⟩ +≡
  bool operator≠(**const Color** &*c*) **const**;

119.

⟨ Define **Color** functions 98 ⟩ +≡
bool Color :: **operator**≠(**const Color** &*c*) **const**
{
  return ¬(∗*this* ≡ *c*);
}

120. Modifying.

121. Set on free store.

Log

[LDF 2003.12.30.] Added this function.

⟨ Declare **Color** functions 97 ⟩ +≡
bool *set_on_free_store*(**bool** *b* = *true*);

122.

⟨ Define **Color** functions 98 ⟩ +≡
bool Color :: *set_on_free_store*(**bool** *b*)
{
  on_free_store = *b*;
  return *b*;
}

123. Set name.

⟨ Declare **Color** functions 97 ⟩ +≡
void *set_name*(**const string** *s*);

124.

⟨ Define **Color** functions 98 ⟩ +≡
void Color :: *set_name*(**const string** *s*)
{
  name = *s*;
}

125. Set use name.

⟨ Declare **Color** functions 97 ⟩ +≡
void *set_use_name*(**const bool** *b*);

126.

⟨ Define **Color** functions 98 ⟩ +≡
void Color :: *set_use_name*(**const bool** *b*)
{
  use_name = *b*;
}

127. Modify.

⟨ Declare **Color** functions 97 ⟩ +≡
void *modify*(**const real** *r*, **const real** *g* = 0, **const real** *b* = 0);

128.

⟨ Define **Color** functions 98 ⟩ +≡

```
void Color::modify(const real r, const real g, const real b)
{
  red_part += r;
  green_part += g;
  blue_part += b;
  if (red_part > 1) {
    cerr << "WARNING!␣In␣Color::modify():\n" << "red_part␣is␣greater␣than␣1:␣" << red_part <<
        endl << "Setting␣red_part␣to␣1.\n\n";
    red_part = 1;
  }
  else if (red_part < 0) {
    cerr << "WARNING!␣In␣Color::modify():\n" << "red_part␣is␣less␣than␣0:␣" << red_part <<
        endl << "Setting␣red_part␣to␣0.\n\n";
    red_part = 0;
  }
  if (green_part > 1) {
    cerr << "WARNING!␣In␣Color::modify():\n" << "green_part␣is␣greater␣than␣1:␣" <<
        green_part << endl << "Setting␣green_part␣to␣1.\n\n";
    green_part = 1;
  }
  else if (green_part < 0) {
    cerr << "WARNING!␣In␣Color::modify():\n" << "green_part␣is␣less␣than␣0:␣" << green_part <<
        endl << "Setting␣green_part␣to␣0.\n\n";
    green_part = 0;
  }
  if (blue_part > 1) {
    cerr << "WARNING!␣In␣Color::modify():\n" << "blue_part␣is␣greater␣than␣1:␣" <<
        blue_part << endl << "Setting␣blue_part␣to␣1.\n\n";
    blue_part = 1;
  }
  else if (blue_part < 0) {
    cerr << "WARNING!␣In␣Color::modify():\n" << "blue_part␣is␣less␣than␣0:␣" << blue_part <<
        endl << "Setting␣blue_part␣to␣0.\n\n";
    blue_part = 0;
  }
  return;
}
```

129. Set red part.

⟨ Declare **Color** functions 97 ⟩ +≡

```
void set_red_part(const real r);
```

130.

⟨ Define **Color** functions 98 ⟩ +≡

```
void Color::set_red_part(const real r)
{
  if (r > 1) {
    cerr << "WARNING!␣In␣Color::set_red_part():\n" << "r␣is␣greater␣than␣1:␣" << r << endl <<
        "Setting␣red_part␣to␣1.\n\n";
    red_part = 1;
  }
  else if (r < 0) {
    cerr << "WARNING!␣In␣Color::set_red_part():\n" << "r␣is␣less␣than␣0:␣" << r << endl <<
        "Setting␣red_part␣to␣0.\n\n";
    red_part = 0;
  }
  else red_part = r;
  return;
}
```

131. Set green part.

⟨ Declare **Color** functions 97 ⟩ +≡

```
void set_green_part(const real g);
```

132.

⟨ Define **Color** functions 98 ⟩ +≡

```
void Color::set_green_part(const real g)
{
  if (g > 1) {
    cerr << "WARNING!␣In␣Color::set_green_part():\n" << "g␣is␣greater␣than␣1:␣" << g <<
        endl << "Setting␣green_part␣to␣1.\n\n";
    green_part = 1;
  }
  else if (g < 0) {
    cerr << "WARNING!␣In␣Color::set_green_part():\n" << "g␣is␣less␣than␣0:␣" << g << endl <<
        "Setting␣green_part␣to␣0.\n\n";
    green_part = 0;
  }
  else green_part = g;
  return;
}
```

133. Set blue part.

⟨ Declare **Color** functions 97 ⟩ +≡

```
void set_blue_part(const real b);
```

134.

⟨ Define **Color** functions 98 ⟩ +≡

```
void Color::set_blue_part(const real b)
{
  if (b > 1) {
    cerr << "WARNING!␣In␣Color::set_blue_part():\n" << "b␣is␣greater␣than␣1:␣" << b <<
        endl << "Setting␣blue_part␣to␣1.\n\n";
    blue_part = 1;
  }
  else if (b < 0) {
    cerr << "WARNING!␣In␣Color::set_blue_part():\n" << "b␣is␣less␣than␣0:␣" << b << endl <<
        "Setting␣blue_part␣to␣0.\n\n";
    blue_part = 0;
  }
  else blue_part = b;
  return;
}
```

135. Show.

⟨ Declare **Color** functions 97 ⟩ +≡

```
void show(string text = "") const;
```

136.

⟨ Define **Color** functions 98 ⟩ +≡

```
void Color::show(string text) const
{
  if (text ≡ "") text = "Color:";
  cout << text << endl;
  cout << "name␣==␣" << get_name() << endl;
  cout << "use_name␣==␣" << get_use_name() << endl;
  cout << "red_part␣==␣" << get_red_part() << endl;
  cout << "green_part␣==␣" << get_green_part() << endl;
  cout << "blue_part␣==␣" << get_blue_part() << endl << endl;
  return;
}
```

137. Returning elements and information.

138. Is on free store.

Log

[LDF 2004.01.06.] Made non-inline.

⟨ Declare **Color** functions 97 ⟩ +≡

```
bool is_on_free_store() const;
```

139.

⟨ Define **Color** functions 98 ⟩ +≡
```
bool Color::is_on_free_store() const
{
  return on_free_store;
}
```

140. Get Color parts. [LDF 2002.09.24.] These functions always return a **real**; the argument *decimal* can't make them return an **unsigned short**.

141. Get red part.

⟨ Declare **Color** functions 97 ⟩ +≡
```
inline real get_red_part(bool decimal = false) const
{
  if (decimal) return trunc((red_part * 255) + .5);
  else return red_part;
}
```

142. Get green part.

⟨ Declare **Color** functions 97 ⟩ +≡
```
inline real get_green_part(bool decimal = false) const
{
  if (decimal) return trunc((green_part * 255) + .5);
  else return green_part;
}
```

143. Get blue part.

⟨ Declare **Color** functions 97 ⟩ +≡
```
inline real get_blue_part(bool decimal = false) const
{
  if (decimal) return trunc((blue_part * 255) + .5);
  else return blue_part;
}
```

144. Get use name.

⟨ Declare **Color** functions 97 ⟩ +≡
```
bool get_use_name() const;
```

145.

⟨ Define **Color** functions 98 ⟩ +≡
```
bool Color::get_use_name() const
{
  return use_name;
}
```

146. Get name.

⟨ Declare **Color** functions 97 ⟩ +≡
```
inline string get_name() const
{
  return name;
}
```

147. Output operator.

⟨ Declare non-member non-template functions for **Color** 147 ⟩ ≡
```
ostream & operator<<(ostream & o, const Color &c);
```
This code is used in section 163.

148.

⟨ Define non-member non-template functions for **Color** 148 ⟩ ≡
```
ostream & operator<<(ostream & o, const Color &c)
{
  if (c.get_use_name() == true) {
    o << c.get_name();
  }
  else {
    o << "(" << c.get_red_part() << ", " << c.get_green_part() << ", " << c.get_blue_part() << ")";
  }
  return o;
}
```
This code is used in section 162.

149. Define Colors in METAPOST.

⟨ Declare **Color** functions 97 ⟩ +≡
```
void define_color_mp() const;
```

150.

⟨ Define **Color** functions 98 ⟩ +≡

```
void Color::define_color_mp() const
{
  if (¬out_stream.is_open()) {
    cerr ≪ "ERROR!␣In␣Color::define_color_mp():\n" ≪
        "out_stream␣is␣closed!␣␣Returning.\n" ≪ flush;
    return;
  }
  if (name ≡ "") {
    cerr ≪ "ERROR!␣In␣Color::define_colors_mp():\n" ≪
        "name␣is␣empty.␣␣Not␣doing␣anything␣and␣returning.\n" ≪ flush;
    return;
  }
  out_stream ≪ "color␣" ≪ name ≪ ";" ≪ "␣␣" ≪ name ≪ "␣=␣(" ≪ get_red_part() ≪ ",␣" ≪
      get_green_part() ≪ ",␣" ≪ get_blue_part() ≪ ");\n" ≪ flush;
  return;
}
```

151. Initialize Colors. [LDF 2002.09.25.] This function presupposes the existence of **namespace Colors**.

⟨ Declare **Color** functions 97 ⟩ +≡

```
static void initialize_colors();
```

152.

⟨ Define **Color** functions 98 ⟩ +≡

```
void Color::initialize_colors()
{
  using namespace Colors;
  if (¬out_stream.is_open()) {
    cerr ≪ "ERROR! In Color::initialize_colors():\n" ≪
        "out_stream is closed!  Returning.\n" ≪ flush;
    return;
  }
  out_stream ≪ "%% Color definitions.\n\n";
#if 0   /* [LDF 2002.09.25.] These colors are already defined in METAPOST and their definitions are
        not likely to change. However, if they do, I can comment these function calls back in. */
  red.define_color_mp();
  green.define_color_mp();
  blue.define_color_mp();
  black.define_color_mp();
  white.define_color_mp();
  background.define_color_mp();
#endif
  yellow.define_color_mp();
  cyan.define_color_mp();
  magenta.define_color_mp();
  orange.define_color_mp();
  violet.define_color_mp();
  purple.define_color_mp();
  yellow_green.define_color_mp();
  green_yellow.define_color_mp();
  blue_violet.define_color_mp();
  gray.define_color_mp();
  light_gray.define_color_mp();
  violet_red.define_color_mp();
  default_background.define_color_mp();
    /* [LDF 2002.09.25.] Currently, this function does nothing if I'm using all of the colors. */
  out_stream ≪ "\n%% End of Color definitions.\n\n";
  return;
}
```

153. Namespace Colors. Here I can put either ⟨ Major **Colors** 156 ⟩ or ⟨ All **Colors** 0 ⟩ into ⟨ Declare **namespace Colors** 153 ⟩, and comment out the other, depending on what I want. This prevents too much unneeded code from being processed. ⟨ All **Colors** 0 ⟩ is very long, so I neither want to compile it, write the **extern** declarations from it to `colors.h`, nor print out the code when I run `cweave`, unless I really want to use it. [LDF 2002.09.25.]

⟨ Declare **namespace Colors** 153 ⟩ ≡

```
  ⟨ Major Colors 156 ⟩    /* ⟨ All Colors 0 ⟩ */
```

This code is cited in section 153.

This code is used in section 162.

154. [LDF 2002.09.25.] Here I can put either ⟨ **extern** Major **Colors** 157 ⟩ or ⟨ **extern** All **Colors** 0 ⟩ into ⟨ **extern namespace Colors** declaration 154 ⟩

⟨ **extern namespace Colors** declaration 154 ⟩ ≡
⟨ **extern** Major **Colors** 157 ⟩

This code is cited in sections 154 and 158.

This code is used in section 163.

155. Major Colors. The colors "red", "green", "blue", "black", and "white" are already defined in METAPOST, however, we need them here in order to access the **Color** functions for them.

!! [LDF 2002.09.24.] If this definition isn't explicitly written to the header file, as it is below, this causes real problems!! It took me awhile to find out that this was the cause.

156. Internal (with initialization). [LDF 2002.09.25.] !! If I add **Colors** here, remember to add them in the "External" section below, and in the definition of **Color** :: *initialize_colors* ( ) below.

Log

[LDF 2002.10.26.] Added *help_color*.

⟨ Major **Colors** 156 ⟩ ≡

```
namespace Colors {    /* Primaries, additive. */
  extern const Color red("red", 255, 0, 0, true);
  extern const Color green("green", 0, 255, 0, true);
  extern const Color blue("blue", 0, 0, 255, true);    /* Primaries, subtractive. */
  extern const Color cyan("cyan", 0, 255, 255, true);
  extern const Color yellow("yellow", 255, 255, 0, true);
  extern const Color magenta("magenta", 255, 0, 255, true);    /* [LDF 2002.09.27.] The convention
      that I use is that colors like "orange_red" are reds and colors like "red_orange" are oranges. */
   /* Red. */
  extern const Color orange_red("orange_red", 255, 69, 0);
  extern const Color violet_red("violet_red", 208, 32, 144);    /* Pink. */
  extern const Color pink("pink", 255, 192, 203);    /* Blue. */    /* Yellow. */
  extern const Color green_yellow("green_yellow", 173, 255, 47);    /* Orange. */
  extern const Color orange("orange", 255, 165, 0, true);    /* Violet. */
  extern const Color violet("violet", 238, 130, 238, true);
  extern const Color purple("purple", 160, 32, 240, true);
  extern const Color blue_violet("blue_violet", 138, 43, 226);    /* Green. */
  extern const Color yellow_green("yellow_green", 154, 205, 50);
   /* "Unbunt" Colors (blacks, whites, and grays). */
  extern const Color black("black", 0, 0, 0, true);
  extern const Color white("white", 255, 255, 255, true);
  extern const Color gray("gray", 192, 192, 192);
  extern const Color light_gray("light_gray", 211, 211, 211);
   /* Defaults. [LDF 2002.09.27.] Note that default_color, help_color and background_color are pointers
      and that default_background is a plain Color. It can be used to access the original background
      color (currently white), if the user points the background_color at some other Color. */
  extern const Color default_background("default_background", 255, 255, 255, true);
  extern const Color *default_color = &black;
  extern const Color *background_color = &default_background;
  extern const Color *help_color = &green;    /* LDF 2002.10.26. Added. */
   /* [LDF 2002.09.25.] !! TO DO: default_background is a convenience, in case I change "background"
      in the METAPOST code. Check METAPOST documentation!! I believe it has something similar.
      */
```

```
  }
```

This code is cited in sections 153 and 158.

This code is used in section 153.

157. External.

```
⟨ extern Major Colors 157 ⟩ ≡
  namespace Colors {    /* [LDF 2002.09.27.] The ordering should be as above for the internal
        declarations. */    /* Primaries, additive. */
    extern const Color red;
    extern const Color green;
    extern const Color blue;    /* Primaries, subtractive. */
    extern const Color cyan;
    extern const Color yellow;
    extern const Color magenta;    /* Red. */
    extern const Color orange_red;
    extern const Color violet_red;    /* Pink. */
    extern const Color pink;    /* Blue. */    /* Yellow. */
    extern const Color green_yellow;    /* Orange. */
    extern const Color orange;    /* Violet. */
    extern const Color violet;
    extern const Color purple;
    extern const Color blue_violet;    /* Green. */
    extern const Color yellow_green;    /* "Unbunt" Colors (black, white, and grays). */
    extern const Color black;
    extern const Color white;
    extern const Color gray;
    extern const Color light_gray;    /* Defaults. */
    extern const Color default_background;
    extern Color *default_color;
    extern Color *help_color;    /* LDF 2002.10.26. Added. */
    extern Color *background_color;
  }
```

This code is cited in section 154.

This code is used in section 154.

158. All Colors. !! IF `colall.web` IS CHANGED, I WILL HAVE TO MAKE SURE THAT THIS FILE is recompiled!! `cmpl` does not check the state of `colall.web`. ?? Should I do something about this, or is it not worth it? If I work on `colall.web`, I could just put the code back in here.

[LDF 2002.09.26.] If I want all of the colors declared in `colall.web`, I can uncomment the following line and use ⟨ All **Colors** 0 ⟩ instead of ⟨ Major **Colors** 156 ⟩ in ⟨ **extern namespace Colors** declaration 154 ⟩ above.

`@i colall.web` (Commented out).

159. Global constants.

Log

[LDF 2002.10.26.] Added *help_color_vector*.

```
⟨ Global constants 22 ⟩ +≡
  namespace Colors {
    extern const vector⟨const Color *⟩ default_color_vector(1, default_color);
```

```
    extern const vector⟨const Color *⟩ help_color_vector(1, help_color);
      /* LDF 2002.10.26. Added. */
    extern const vector⟨const Color *⟩ background_color_vector(1, background_color);
  }
```

160.

⟨ **extern** global constant declarations 160 ⟩ ≡

```
  namespace Colors {
    extern const vector⟨const Color *⟩ default_color_vector;
    extern const vector⟨const Color *⟩ help_color_vector;      /* LDF 2002.10.26. Added. */
    extern const vector⟨const Color *⟩ background_color_vector;
  }
```

This code is used in section 163.

161. Putting Color together.

162. This is compiled.

⟨ Include files 6 ⟩
⟨ Version control identifier 5 ⟩;
⟨ Define **class Color** 95 ⟩
⟨ Declare **namespace Colors** 153 ⟩
⟨ Define **Color** functions 98 ⟩
⟨ Declare non-member template functions for **Color** 112 ⟩
⟨ Define non-member non-template functions for **Color** 148 ⟩
⟨ Global constants 22 ⟩

163. This is written to `colors.h`.

⟨ `colors.h` 163 ⟩ ≡
⟨ Define **class Color** 95 ⟩
⟨ **extern namespace Colors** declaration 154 ⟩
⟨ Declare non-member template functions for **Color** 112 ⟩
⟨ Declare non-member non-template functions for **Color** 147 ⟩
⟨ **extern** global constant declarations 160 ⟩

164. Transformations (`transform.web`).

Log

[LDF 2003.11.12.] Removed the version control identifiers from the CWEB files for the distribution of 3DLDF 1.1. They're still used in my development versions.

[LDF 2003.12.10.] Put the version control identifiers back into my release versions for 3DLDF 1.1.4. I've already put some of them back in, now I'm doing the rest of them. However, the release versions are now in their own RCS repository.

⟨ Version control identifier 5 ⟩ +≡
static string *rcs_id* = `"$Id:␣transfor.web,v␣1.5␣2004/01/12␣21:33:44␣lfinsto1␣Exp␣$"`;

165. Include files.

⟨ Include files 6 ⟩ +≡
#include `"loader.h"`
#include `"pspglb.h"`
#include `"io.h"`
#include `"colors.h"`

166. Transform class definition. The **Transform** class has a 4×4 transformation matrix as its only data member and a number of member functions. **Points**, **Pictures**, and *Focuses* contain **Transforms** as data members.

!! Remember to add items to **operator**=( ) if if I add them to the class definition here.

Log

[LDF 2003.07.04.] Removed **friend** declaration for **Focus**. I've added *set_element*( ) and *get_element*( ), which are used in the **Focus** functions, so the latter need no longer be a **friend** of **Transform**.

---

format *Transform* *int*

⟨ Define **class Transform** 166 ⟩ ≡
class Transform {
  friend class Point;
  Matrix *matrix*; /∗ When I've got things working, I can try to optimize use of storage by not storing the parts of the matrix that I don't need. This is a little complicated, because the row or column which isn't needed differs between the affine and perspective transformations. ∗/
public: ⟨ Declare **Transform** functions 168 ⟩
};

This code is used in sections 239 and 240.

167. Constructors.

168. Default constructor. (No arguments). Initializes a new transformation matrix as the identity matrix

⟨ Declare **Transform** functions 168 ⟩ ≡
Transform( );

See also sections 170, 172, 174, 176, 178, 180, 182, 185, 187, 190, 192, 195, 197, 200, 202, 203, 205, 211, 212, 213, 216, 218, 221, 223, 226, and 232.

This code is used in section 166.

169.

⟨ Define **Transform** functions 169 ⟩ ≡
```
Transform::Transform()
{
  reset();
}
```

See also sections 171, 173, 175, 177, 179, 181, 183, 186, 188, 191, 193, 196, 198, 201, 204, 206, 207, 208, 209, 217, 219, 222, 224, 227, 228, 229, 230, 231, 233, 416, 424, 425, 426, 427, 428, 429, 430, 431, 432, 439, and 759.

This code is used in sections 239, 633, and 980.

170. Constructor with one real argument. All elements of *matrix* are set to the **real** argument r.

⟨ Declare **Transform** functions 168 ⟩ +≡
```
Transform(real r);
```

171.

⟨ Define **Transform** functions 169 ⟩ +≡
```
Transform::Transform(real r)
{
  for (int i = 0; i < 4; i++)
    for (int j = 0; j < 4; j++) matrix[i][j] = r;
}
```

172. Constructor with 16 real arguments. [LDF 2002.09.06.] Added this constructor. This constructor makes it possible to specify all of the elements of *matrix*.

⟨ Declare **Transform** functions 168 ⟩ +≡
```
Transform(real r0_0, real r0_1, real r0_2, real r0_3, real r1_0, real r1_1, real r1_2, real r1_3, real
    r2_0, real r2_1, real r2_2, real r2_3, real r3_0, real r3_1, real r3_2, real r3_3);
```

173.

⟨ Define **Transform** functions 169 ⟩ +≡

```
Transform::Transform(real r0_0, real r0_1, real r0_2, real r0_3, real r1_0, real r1_1, real r1_2, real
        r1_3, real r2_0, real r2_1, real r2_2, real r2_3, real r3_0, real r3_1, real r3_2, real r3_3)
{
  matrix[0][0] = r0_0;
  matrix[0][1] = r0_1;
  matrix[0][2] = r0_2;
  matrix[0][3] = r0_3;
  matrix[1][0] = r1_0;
  matrix[1][1] = r1_1;
  matrix[1][2] = r1_2;
  matrix[1][3] = r1_3;
  matrix[2][0] = r2_0;
  matrix[2][1] = r2_1;
  matrix[2][2] = r2_2;
  matrix[2][3] = r2_3;
  matrix[3][0] = r3_0;
  matrix[3][1] = r3_1;
  matrix[3][2] = r3_2;
  matrix[3][3] = r3_3;
}
```

174. Assignment. Sets *matrix* to be identical to the *matrix* of another **Transform**. !! Remember to add items here if I add them to the class definition.

⟨ Declare **Transform** functions 168 ⟩ +≡

```
Transform operator=(const Transform &t);
```

175.

⟨ Define **Transform** functions 169 ⟩ +≡

```
Transform Transform::operator=(const Transform &t)
{
  for (int i = 0; i < 4; i++)
    for (int j = 0; j < 4; j++) matrix[i][j] = t.matrix[i][j];
  return t;
}
```

176. Reset to identity matrix.

⟨ Declare **Transform** functions 168 ⟩ +≡

```
void reset();
```

177.

⟨ Define **Transform** functions 169 ⟩ +≡

```
void Transform::reset()
{
  for (int i = 0; i < 4; i++)    /* Rows. */
    for (int j = 0; j < 4; j++)    /* Columns. */
    {
      matrix[i][j] = (i ≡ j) ? 1 : 0;
    }
}
```

178. Setting values.

Log

[LDF 2003.07.04.] Added this function.

⟨ Declare **Transform** functions 168 ⟩ +≡

```
void set_element(const unsigned short row, const unsigned short col, real r);
```

179.

⟨ Define **Transform** functions 169 ⟩ +≡

```
void Transform::set_element(const unsigned short row, const unsigned short col, real r)
{
  matrix[row][col] = (fabs(r) < epsilon()) ? 0 : r;
  return;
}
```

180. Clean. *clean*( ) changes elements in *matrix* whose absolute values are < *epsilon*( ) to 0.

⟨ Declare **Transform** functions 168 ⟩ +≡

```
void clean();
```

181.

⟨ Define **Transform** functions 169 ⟩ +≡

```
void Transform::clean()
{
  real eps = epsilon();
  for (int i = 0; i < 4; i++)    /* Rows. */
    for (int j = 0; j < 4; j++)    /* Columns. */
    {
      if (fabs(matrix[i][j]) < eps) matrix[i][j] = 0;
    }
}
```

182. Epsilon. Minimum magnitude of values stored in *matrix*. [LDF 2002.10.16.] The value returned by *epsilon*( ) has to be fairly large because of the poor precision resulting from the use *floats* and the Standard Library versions of the trigonometric functions. There is currently no equality operator for **Transform**, but the precision of the **Transforms** affects that of **Points**. TO DO: I hope to be able to solve the problem by finding routines for calculating the trigonometric functions more accurately (and faster) by using integers and bitwise shifts. If this doesn't work out, I could try redefining **real** as **double** (which I don't want to do), or I could try to use **double** explicitly when using the trigonometric functions. In the latter case, I would have to truncate the *doubles* to *floats* eventually, so I don't know if this would have any benefit.

Log

[LDF 2004.1.2.] Now returning different values, depending on whether **real** is **float** or **double**. TO DO: Try to find out what values would be best. It will be necessary to check how good the value for **double** is.

[LDF 2004.1.2.] Made *epsilon*( ) **static** and non-inline.

⟨ Declare **Transform** functions 168 ⟩ +≡

```
static real epsilon();
```

183.

⟨ Define **Transform** functions 169 ⟩ +≡

```
real Transform::epsilon()
{
#if LDF_REAL_DOUBLE
  return .000000001;
#else
  return .00001;
#endif
}
```

184. Test for identity matrix. [LDF 2002.11.16.] TO DO: I should check the elements on the main diagonal for whether they differ from 1 by an amount <*epsilon*( ). If so, they should be set to 1.

185. Non-const version.

⟨ Declare **Transform** functions 168 ⟩ +≡

```
bool is_identity();
```

186.

⟨ Define **Transform** functions 169 ⟩ +≡

```
bool Transform::is_identity()
{
  clean();
  for (int i = 0; i < 4; i++)
    for (int j = 0; j < 4; j++)
      if ((i ≡ j ∧ matrix[i][j] ≠ 1) ∨ ((i ≠ j) ∧ matrix[i][j] ≠ 0)) return false;
  return true;
}
```

187. const version.

⟨ Declare **Transform** functions 168 ⟩ +≡

```
bool is_identity() const;
```

188.

⟨ Define **Transform** functions 169 ⟩ +≡

```
bool Transform::is_identity() const
{
  Transform t;
  t = *this;
  t.clean();
  for (int i = 0; i < 4; i++)
    for (int j = 0; j < 4; j++)
      if ((i ≡ j ∧ matrix[i][j] ≠ 1) ∨ ((i ≠ j) ∧ matrix[i][j] ≠ 0)) return false;
  return true;
}
```

189. Querying.

190. Get element.

Log

[LDF 2003.07.04.] Added this function.

⟨ Declare **Transform** functions 168 ⟩ +≡

```
real get_element(const unsigned short row, const unsigned short col) const;
```

191.

⟨ Define **Transform** functions 169 ⟩ +≡

```
real Transform::get_element(const unsigned short row, const unsigned short col) const
{
  return matrix[row][col];
}
```

192. Show.

⟨ Declare **Transform** functions 168 ⟩ +≡

```
void show(string text = "") const;
```

193.

⟨ Define **Transform** functions 169 ⟩ +≡

```
void Transform::show(string text) const
{
  if (text ≡ "") text = "Transform:";
  cout ≪ text ≪ "\n";
  for (int i = 0; i < 4; i++) {
    for (int j = 0; j < 4; j++) {   /* !! [LDF 2002.02.07.] Can't use left here, and I can't include
          ios. This causes an error with a reference to something in the C++ manual. Must write to K.
          Heuer and ask if he can fix it. */
      cout ≪ setw(7) ≪ setprecision(3) ≪ matrix[i][j] ≪ "␣";
    }
    cout ≪ endl;
  }
  cout ≪ endl ≪ flush;
}
```

194. Affine transformations. [LDF 2002.10.16.] The functions for the affine transformations all return a **Transform** representing the transformation, not **this*. This makes it possible to chain expressions using **operator**∗=( ), e.g.,

```
Point pt0;
Point pt1(1,2,3);
Point pt2(3,4,5);
Transform t;
t.rotate(90,90,90);
pt0 *= pt1 *= pt2 *= t;
```

pt0, *pt1*, and *pt2* are all rotated 90° around the x, y, and z-axes.

195. Scale.

Log

[LDF 2002.10.15.] BUG FIX: If the absolute value of an argument is < *epsilon*( ), the argument is now set to 0 instead of 1. Setting it to 1 causes no scaling to take place, which is not the effect of multiplying the corresponding coordinate by a number of very small magnitude.

[LDF 2003.03.25.] BUG FIX: Fixed conditional that tests whether all the arguments are 1.

⟨ Declare **Transform** functions 168 ⟩ +≡

```
Transform scale(real x, real y = 1, real z = 1);
```

196.

⟨ Define **Transform** functions 169 ⟩ +≡

```
Transform Transform::scale(real x, real y, real z)
{
  Transform t;
  if (x ≡ 1 ∧ y ≡ 1 ∧ z ≡ 1) {
    cerr ≪ "WARNING!␣In␣Transform::scale():\n" ≪
        "All␣arguments␣==␣1.␣␣Returning␣identity␣transformation." ≪ endl ≪ endl ≪ flush;
    return t;
  }
  real eps = epsilon();
  t.matrix[0][0] = (fabs(x) ≥ eps) ? x : 0;
  t.matrix[1][1] = (fabs(y) ≥ eps) ? y : 0;
  t.matrix[2][2] = (fabs(z) ≥ eps) ? z : 0;
  *this *= t;
  clean();
  return t;
}
```

197. Shear. [LDF 2002.10.15.] Replaced the dummy definition of this function with a proper one.

⟨ Declare **Transform** functions 168 ⟩ +≡

```
Transform shear(real xy, real xz = 0, real yx = 0, real yz = 0, real zx = 0, real zy = 0);
```

198.

⟨ Define **Transform** functions 169 ⟩ +≡

```
Transform Transform::shear(real xy, real xz, real yx, real yz, real zx, real zy)
{
  Transform t;
  real eps = epsilon();
  if (fabs(xy) < eps) xy = 0;
  if (fabs(xz) < eps) xz = 0;
  if (fabs(yx) < eps) yx = 0;
  if (fabs(yz) < eps) yz = 0;
  if (fabs(zx) < eps) zx = 0;
  if (fabs(zy) < eps) zy = 0;
  if (xy ≡ 0 ∧ xz ≡ 0 ∧ yx ≡ 0 ∧ yz ≡ 0 ∧ zx ≡ 0 ∧ zy ≡ 0) {
    cerr ≪ "WARNING!␣In␣Transform::shear():\n" ≪
        "All␣arguments␣are␣0.␣␣Returning␣identity␣transformation." ≪ endl ≪ endl ≪ flush;
    return t;
  }
  t.matrix[1][0] = xy;
  t.matrix[2][0] = xz;
  t.matrix[0][1] = yx;
  t.matrix[2][1] = yz;
  t.matrix[0][2] = zx;
  t.matrix[1][2] = zy;
  (*this) *= t;
  clean();
  return t;
}
```

199. Shift. (Translation.)

200. real arguments.

⟨ Declare **Transform** functions 168 ⟩ +≡

```
Transform shift(real x, real y = 0, real z = 0);
```

201.

⟨ Define **Transform** functions 169 ⟩ +≡

```
Transform Transform::shift(real x, real y, real z)
{
  real eps = epsilon();
  Transform t;
  if (x ≡ 0 ∧ y ≡ 0 ∧ z ≡ 0) return t;
  t.matrix[3][0] = (fabs(x) > eps) ? x : 0;
  t.matrix[3][1] = (fabs(y) > eps) ? y : 0;
  t.matrix[3][2] = (fabs(z) > eps) ? z : 0;
  t.clean();
  (*this) *= t;
  clean();
  return t;
}
```

202. Point argument. [LDF 2002.04.24.] Added this function. It must be defined in `points.web`, because **Point** is an incomplete type here.

⟨ Declare **Transform** functions 168 ⟩ +≡
Transform *shift*(**const Point** &*p*);

203. Shift with multiplication. [LDF 2002.08.22.] Added this function. It takes **real** arguments and multiplies the appropriate elements of *matrix* by them.

⟨ Declare **Transform** functions 168 ⟩ +≡
Transform *shift_times*(**real** x, **real** $y = 1$, **real** $z = 1$);

204.

⟨ Define **Transform** functions 169 ⟩ +≡
```
Transform Transform::shift_times(real x, real y, real z)
{
  bool DEBUG = false;   /* true */
  if (DEBUG) {
    show("Before␣multiplication");
  }
  matrix[3][0] *= x;
  matrix[3][1] *= y;
  matrix[3][2] *= z;
  if (DEBUG) {
    show("After␣multiplication");
  }
  return *this;
}
```

205. Rotation around the main axes. *rotate*( ) will perform rotation about the x, y and z-axes in that order if called with multiple, non-zero arguments. Rotation only about the y and/or z-axis requires one or two dummy 0 arguments so that *rotate*( ) "knows" about which axis (or axes) to rotate.

⟨ Declare **Transform** functions 168 ⟩ +≡
Transform *rotate*(**real** x, **real** $y = 0$, **real** $z = 0$);

206.

⟨ Define **Transform** functions 169 ⟩ +≡

```
Transform Transform::rotate(real x, real y, real z){ bool DEBUG = false;      /* true */
    Transform t_all;
    real eps = epsilon();
    if (x ≡ 0 ∧ y ≡ 0 ∧ z ≡ 0) {
      if (DEBUG) cerr ≪ "In␣rotate(real,␣real,␣real)\n" ≪
          "0␣rotation␣about␣all␣axes.␣Returning␣identity␣matrix.\n" ≪ flush;
      return t_all;
    }
    Transform t_x;
    Transform t_y;
    Transform t_z;
    real ssin;
    real ccos;
    real temp1;
    real temp2;
    int i;
```

207. Rotation around the x-axis.

⟨ Define **Transform** functions 169 ⟩ +≡

```
if (x ≠ 0) {      /* !! Reversed direction of rotation because I didn't like the way it was. */
  x *= −PI/180.0;      /* Convert to radians. */
  ssin = sin(x);
  ccos = cos(x);
  for (i = 0; i < 4; i++) {
    temp1 = (t_x.matrix[i][1] * ccos) − (t_x.matrix[i][2] * ssin);
    temp2 = (t_x.matrix[i][1] * ssin) + (t_x.matrix[i][2] * ccos);
    t_x.matrix[i][2] = (fabs(temp2) ≥ eps) ? temp2 : 0;
    t_x.matrix[i][1] = (fabs(temp1) ≥ eps) ? temp1 : 0;
  }
}      /* if */
```

208. Rotation around the y-axis.

⟨ Define **Transform** functions 169 ⟩ +≡

```
if (y ≠ 0) {      /* !! Reversed direction of rotation because I didn't like the way it was. */
  y *= −PI/180.0;
  ssin = sin(y);
  ccos = cos(y);
  for (i = 0; i < 4; i++) {
    temp1 = (t_y.matrix[i][0] * ccos) + (t_y.matrix[i][2] * ssin);
    temp2 = (−t_y.matrix[i][0] * ssin) + (t_y.matrix[i][2] * ccos);
    t_y.matrix[i][2] = (fabs(temp2) ≥ eps) ? temp2 : 0;
    t_y.matrix[i][0] = (fabs(temp1) ≥ eps) ? temp1 : 0;
  }
}      /* if */
```

209. Rotation around the z-axis.

⟨ Define **Transform** functions 169 ⟩ +≡

```
if (z ≠ 0) {
  z *= PI/180.0;
  ssin = sin(z);
  ccos = cos(z);
  for (int i = 0; i < 4; i++) {
    temp1 = (t_z.matrix[i][0] * ccos) − (t_z.matrix[i][1] * ssin);
    temp2 = (t_z.matrix[i][0] * ssin) + (t_z.matrix[i][1] * ccos);
    t_z.matrix[i][1] = (fabs(temp2) ≥ eps) ? temp2 : 0;
    t_z.matrix[i][0] = (fabs(temp1) ≥ eps) ? temp1 : 0;
  }
}   /* if */
t_all *= t_x;
t_all *= t_y;
t_all *= t_z;
t_all.clean();
(*this) *= t_all;
clean();
return t_all; }   /* End of rotate(). */
```

210. Rotation around an arbitrary axis.

211. Point arguments. Defined in `points.web` because **Point** is an incomplete type here.

Log

[LDF 2002.4.7.] Added default value for *angle* ≡ 180.

[LDF 2003.05.02.] Changed name of this function from *rotate_around*( ) to *rotate*( ). This function now overloads *rotate*( ) with three **real** arguments.

⟨ Declare **Transform** functions 168 ⟩ +≡

```
Transform rotate(Point p0, Point p1, const real angle = 180);
```

212. Path argument. [LDF 2002.05.03.] Defined in `paths.web` because **Path** is an incomplete type here.

Log

[LDF 2002.05.03.] Added this function.
[LDF 2003.05.02.] Changed name of this function from *rotate_around*( ) to *rotate*( ). This function now overloads *rotate*( ) with three **real** arguments.

⟨ Declare **Transform** functions 168 ⟩ +≡
Transform *rotate*(**const Path** &*p*, **const real** *angle* = 180);

213. Alignment with an axis. Defined in `points.web`, because it uses **Points**, which haven't been defined yet.

⟨ Declare **Transform** functions 168 ⟩ +≡
Transform *align_with_axis*(**Point** *p0*, **Point** *p1*, **char** *axis* = 'z'); /* Default is the z-axis. */

214. Matrix multiplication.

215. With assignment.

216. real argument. [LDF 2002.11.19.] This function multiplies each element of **Matrix** by the **real** argument *r* and returns *r*. This makes it possible to chain invocations of this function. Not currently used anywhere, but it may turn out to be useful for something.

Log

[LDF 2002.08.22.] Added this function.
[LDF 2002.11.19.] Changed return value from **this* to *r*.

⟨ Declare **Transform** functions 168 ⟩ +≡
real operator*=(**real** *r*);

217.

⟨ Define **Transform** functions 169 ⟩ +≡
```
real Transform::operator*=(real r)
{
  for (int i = 0; i < 4; i++)
    for (int j = 0; j < 4; j++) matrix[i][j] *= r;
  clean();
  return r;
}
```

218. Transform argument.

Log

[LDF 2002.11.06.] If *t* is the identity **Transform**, it is returned right away. If **this* is, it is set to *t* using **operator**=( ). BUG FIX: Now *t* is always returned, instead of **this*. This makes it possible to chain expressions using this function.

⟨ Declare **Transform** functions 168 ⟩ +≡
Transform operator*=(**const Transform** &*t*);

219.

⟨ Define **Transform** functions 169 ⟩ +≡

```
Transform Transform::operator*=(const Transform &t)
{
  bool DEBUG = false;  /* true */
  if (DEBUG) cout << "Entering Transform::operator*=().\n" << flush;
  if (t.is_identity()) {
    if (DEBUG) cout << "t is the identity transformation. " << "Returning t.\n" << flush;
    return t;
  }
  else if (is_identity()) {
    if (DEBUG) cout << "*this is the identity transformation. " <<
        "Setting *this to t and returning t.\n" << flush;
    return (*this = t);
  }
  Matrix temp_matrix = {{0,0,0,0},{0,0,0,0},{0,0,0,0},{0,0,0,0}};
  for (int i = 0; i < 4; i++)
    for (int k = 0; k < 4; k++)
      for (int j = 0; j < 4; j++) temp_matrix[i][k] += matrix[i][j] * t.matrix[j][k];
  for (int i = 0; i < 4; i++)
    for (int j = 0; j < 4; j++) {
      matrix[i][j] = temp_matrix[i][j];
    }
  clean();
  if (DEBUG) cout << "Exiting Transform::operator*=().\n" << flush;
  return t;
}
```

220. Plain multiplication.

221. real argument. [LDF 2002.08.22.] Added this function. Not currently used anywhere, but it may turn out to be useful for something.

⟨ Declare **Transform** functions 168 ⟩ +≡

```
Transform operator*(const real r) const;
```

222.

⟨ Define **Transform** functions 169 ⟩ +≡

```
Transform Transform::operator*(const real r) const
{
  Transform t = *this;
  t *= r;
  t.clean();
  return t;
}
```

223. Transform argument.

⟨ Declare **Transform** functions 168 ⟩ +≡

```
Transform operator*(const Transform t) const;
```

224.

⟨ Define **Transform** functions 169 ⟩ +≡

```
Transform Transform::operator*(const Transform t) const
{
  Transform a = *this;
  a *= t;
  a.clean();
  return a;
}
```

225. Matrix inversion.

226. const version (no assignment). It would be easy to generate the inverses of the transformations that I call explicitly using *rotate*( ), *shift*( ), etc., as I go along. However, it is not possible to do this for the ones produced using **operator**∗( ) and **operator**∗=( ). So, since a matrix inversion routine is needed anyway, I don't bother to generate the inverses as I go along.

TO DO: Get format for references! *inverse*( ) uses the Gauß-Jordan algorithm with column pivot search. I've taken the algorithm from Stoer, Josef. *Numerische Mathematik 1*[1] and adapted it to C++.

Log

[LDF 2002.12.01.] !! Changed *hi* from **real** to **int** because of a warning, when I tried to compile under GNU/Linux. I think *hi* can be an **int**, but test to be sure!

⟨ Declare **Transform** functions 168 ⟩ +≡

```
Transform inverse() const;
```

[1] Stoer, Josef. *Numerische Mathematik 1.* Achte, neu bearbeitete und erweiterte Auflage. Springer-Verlag. Berlin 1999. ISBN 3-540-66154-9, page 205.

227.

⟨ Define **Transform** functions 169 ⟩ +≡

```
Transform Transform::inverse() const{ bool DEBUG = false;   /* true */
    if (DEBUG) cout << "Entering␣Transform::inverse()." << endl << flush;
    int i;
    int j;
    int k;
    int row;
    const int n = 4;
    real max;
    real hr;
    int hi;   /* [LDF 2002.12.01.] See above in "Log". */
    real hv[n];
    Transform t;
    if (DEBUG) {
      cout << "matrix␣=\n";
    }
    for (i = 0; i < 4; i++) {
      for (j = 0; j < 4; j++) {
        t.matrix[i][j] = matrix[i][j];
        if (DEBUG) {
          cout << matrix[i][j] << "␣";
        }
      }
      if (DEBUG) {
        cout << "\n";
      }
    }
    if (DEBUG) {
      cout << "\n";
      t.show("t");
      cout << "Enter␣<RETURN>␣to␣continue.\n\n" << flush;
      getchar();
    }
    int p[n];
    for (int j = 0; j < n; j++) p[j] = j;
    for (j = 0; j < n; j++) {
```

228. Pivot search.

⟨ Define **Transform** functions 169 ⟩ +≡

```
max = fabs(t.matrix[j][j]);
row = j;
for (i = j + 1; i < n; i++) {
   if (fabs(t.matrix[i][j]) > max) {
      max = fabs(t.matrix[i][j]);
      row = i;
   }   /* if */
}   /* inner for */
if (DEBUG) {
   cout << "max == " << max << endl << flush;
   cout << "row == " << row << endl << flush;
}
if (max ≡ 0) {
   cerr << "ERROR! In Transform::inverse():\n" <<
       "Matrix is singular.  Returning INVALID_TRANSFORM.\n" << flush;
   return INVALID_TRANSFORM;
}
```

229. Row exchange.

⟨ Define **Transform** functions 169 ⟩ +≡

```
if (row > j) {
   for (k = 0; k < n; k++) {
      hr = t.matrix[j][k];
      t.matrix[j][k] = t.matrix[row][k];
      t.matrix[row][k] = hr;
   }   /* for */
   hi = p[j];
   p[j] = p[row];
   p[row] = hi;
}   /* if */
if (DEBUG) cout << "Finished row exchange.\n" << flush;
```

230. Transformation.

⟨ Define **Transform** functions 169 ⟩ +≡

```
if (DEBUG) cout << "t.matrix[" << j << "][" << j << "] == " << t.matrix[j][j] << endl << flush;
hr = 1/t.matrix[j][j];
for (i = 0; i < n; i++) t.matrix[i][j] = hr * t.matrix[i][j];
t.matrix[j][j] = hr;
for (k = 0; k < n; k++)
   if (k ≠ j) {
      for (i = 0; i < n; i++)
         if (i ≠ j) t.matrix[i][k] -= t.matrix[i][j] * t.matrix[j][k];
      t.matrix[j][k] *= -hr;
   }
}   /* outer for */
if (DEBUG) cout << "Finished Transformation.\n" << flush;
```

231. Column exchange.

⟨ Define **Transform** functions 169 ⟩ +≡

```
for (i = 0; i < n; i++) {
  for (k = 0; k < n; k++) hv[p[k]] = t.matrix[i][k];
  for (k = 0; k < n; k++) t.matrix[i][k] = hv[k];
}   /* for */
if (DEBUG) {
  cout << "Finished column exchange.\n" << flush;
  cout << "Exiting Transform::inverse()." << endl << flush;
}
if (DEBUG) cout << "Exiting Transform::inverse()." << endl << flush;
return t; }
```

232. Non-const version (with assignment). [LDF 2002.10.20.] Added this function. I thought of calling in "*invert*()", but I decided against it, because I thought having two functions called "*inverse*()" and "*invert*()" would be confusing.

There is no point in calling this function with *assign* ≡ *false*, since it is equivalent to the **const** version above with no argument. If it is called with *assign* ≡ *false*, a warning is issued, the **const** version is invoked, and its return value is returned.

⟨ Declare **Transform** functions 168 ⟩ +≡

```
Transform inverse(bool assign);
```

233.

⟨ Define **Transform** functions 169 ⟩ +≡

```
Transform Transform::inverse(bool assign)
{
  bool DEBUG = false;   /* true */
  if (DEBUG) cout << "Entering Transform::inverse(bool assign).\n" << flush;
  if (assign ≡ false) {
    cerr << "WARNING: In Transform::inverse(bool assign):\n" <<
        "assign == false.  There's no reason to ever " <<
        "call this function with a \"false\" argument,\n" <<
        "but it doesn't do any harm.  Continuing.\n\n" << flush;
    return inverse();
  }
  if (DEBUG) cout << "Exiting Transform::inverse(bool assign).\n" << flush;
  return (*this = inverse());
}
```

234. Global variables.

⟨ Global variables 18 ⟩ +≡

```
Transform user_transform;
```

235.

⟨ Declarations for the header file 21 ⟩ +≡
  extern Transform *user_transform*;

236. Global constants.

⟨ Global constants 22 ⟩ +≡
  extern const Transform `INVALID_TRANSFORM(INVALID_REAL)`;
  extern const Transform `IDENTITY_TRANSFORM`;

237.

⟨ Declarations for the header file 21 ⟩ +≡
  extern const Transform `INVALID_TRANSFORM`;
  extern const Transform `IDENTITY_TRANSFORM`;

238. Putting Transform together.

239. This is what's compiled.

  ⟨ Include files 6 ⟩
  ⟨ Version control identifier 5 ⟩
  ⟨ Define **class Transform** 166 ⟩
  ⟨ Global variables 18 ⟩
  ⟨ Global constants 22 ⟩
  ⟨ Define **Transform** functions 169 ⟩

240. This is what's written to `transfor.h`.

⟨ `transfor.h` 240 ⟩ ≡
  ⟨ Define **class Transform** 166 ⟩
  ⟨ Declarations for the header file 21 ⟩

241. Shape (`shapes.web`). [LDF 2002.10.20.] **Shape** is an abstract class. This means that no objects of type **Shape** may be declared. **Shape** is used as a base class for all "drawable" classes, e.g., **Point**, **Path**, and **Dodecahedron**. All objects that are put onto a **Picture** must be either **Shapes** or **Labels**.

Log

[LDF 2003.11.12.] Removed the version control identifiers from the CWEB files for the distribution of 3DLDF 1.1. They're still used in my development versions.

[LDF 2003.12.10.] Put the version control identifiers back into my release versions for 3DLDF 1.1.4. I've already put some of them back in, now I'm doing the rest of them. However, the release versions are now in their own RCS repository.

⟨ Version control identifier 5 ⟩ +≡
  static string *rcs_id* = `"$Id:␣shapes.web,v␣1.4␣2004/01/12␣21:32:30␣lfinsto1␣Exp␣$"`;

242. Include files.

⟨ Include files 6 ⟩ +≡
#**include** `"loader.h"`
#**include** `"pspglb.h"`
#**include** `"io.h"`
#**include** `"colors.h"`
#**include** `"transfor.h"`

243. Shape class definition.

244. class Point is known when `shapes.c` is compiled, because it's declared (but not defined) in `transfor.web`, which is processed by `cmpl` first.

?? Apparently, both the return value and the argument types of pure virtual functions must be the same, otherwise the derived classes will cause compiler errors. Check where this is stated.

Log

[LDF 2003.05.16.] Added declarations of *get_minimum_z*( ) and *get_mean_z*( ).

---

  format *Shape* *int*

⟨ Define **Shape** class 244 ⟩ ≡
  class Shape {
protected: **static const signed short** `DRAWDOT`; /* **const** values used for output. */
  static const signed short `DRAW`;
  static const signed short `FILL`;
  static const signed short `FILLDRAW`;
  static const signed short `UNDRAWDOT`;
  static const signed short `UNDRAW`;
  static const signed short `UNFILL`;
  static const signed short `UNFILLDRAW`;

See also section 245.

This code is used in sections 248 and 249.

245. Shape function declarations. All **Shape** functions are pure virtual functions.

[LDF 2002.10.20.] I've thought about getting rid of *get_copy*( ) a couple of times, and using `create_new_`⟨*type*⟩( ) instead, but it's not possible: *get_copy*( ) is used in **Picture** functions for objects of types derived from **Shape** where the type is not known. The compiler must resolve to the correct version of *get_copy*( ), so a **virtual Shape** function is needed. The "`create_new_`⟨*type*⟩( )" functions are not **virtual Shape** functions, and can't be, because the names of the types are part of the name of the functions. I could solve this problem by renaming *get_copy*( ) *create_new*( ), but what I wanted to do was have a template function *create_new*( ) (or just *create*( )). So far, I haven't been able to get this to work. So, for the time being, I'm leaving things as they are.

[LDF 2002.10.23.] The default arguments to *show*( ) are necessary, since

⟨ Define **Shape** class 244 ⟩ +≡

```
public: virtual void show(string text = "", char coords = 'w', const bool do_persp = true, const
      bool do_apply = true, Focus *f = 0, const unsigned short proj = 0, const real factor = 1)
      const = 0;
  virtual Shape *get_copy() const = 0;
  virtual bool is_on_free_store() const = 0;
  virtual bool set_on_free_store(bool b = true) = 0;
  virtual void clear() = 0;
  virtual void output() = 0;
  virtual vector<Shape *> extract(const Focus &, const unsigned short proj, real factor) = 0;
  virtual Transform rotate(const real, const real, const real) = 0;
  virtual Transform scale(real, real, real) = 0;
  virtual Transform shear(real xy, real xz, real yx, real yz, real zx, real zy) = 0;
  virtual Transform shift(real, real, real) = 0;
  virtual Transform rotate(const Point &, const Point &, const real) = 0;
  virtual Transform operator*=(const Transform &) = 0;
  virtual void apply_transform(void) = 0;
  virtual bool set_extremes() = 0;
  virtual real get_minimum_z() const = 0;
  virtual real get_maximum_z() const = 0;
  virtual real get_mean_z() const = 0;
  virtual const valarray<real> get_extremes() const = 0;
  virtual void suppress_output() = 0;
  virtual void unsuppress_output() = 0; } ;
```

246. Static data members.

⟨ Define **static Shape** member variables 246 ⟩ ≡

```
  const signed short Shape::DRAWDOT = 1;
  const signed short Shape::DRAW = 2;
  const signed short Shape::FILL = 3;
  const signed short Shape::FILLDRAW = 4;
  const signed short Shape::UNDRAWDOT = −1;
  const signed short Shape::UNDRAW = −2;
  const signed short Shape::UNFILL = −3;
  const signed short Shape::UNFILLDRAW = −4;
```

This code is used in section 248.

247. Putting Shape together.

248. This is what's compiled.

⟨ Include files 6 ⟩
⟨ Version control identifier 5 ⟩
⟨ Define **Shape** class 244 ⟩
⟨ Define **static Shape** member variables 246 ⟩

249. This is what's written to `shapes.h`.

⟨ `shapes.h` 249 ⟩ ≡
⟨ Define **Shape** class 244 ⟩

250. Picture and Label (`pictures.web`).

Log

[LDF 2003.11.12.] Removed the version control identifiers from the CWEB files for the distribution of 3DLDF 1.1. They're still used in my development versions.

[LDF 2003.12.10.] Put the version control identifiers back into my release versions for 3DLDF 1.1.4. I've already put some of them back in, now I'm doing the rest of them. However, the release versions are now in their own RCS repository.

⟨ Version control identifier 5 ⟩ +≡
static string *rcs_id* = `"$Id:␣pictures.web,v␣1.4␣2004/01/12␣21:31:45␣lfinsto1␣Exp␣$"`;

251. Include files.

⟨ Include files 6 ⟩ +≡
#**include** `"loader.h"`
#**include** `"pspglb.h"`
#**include** `"io.h"`
#**include** `"colors.h"`
#**include** `"transfor.h"`
#**include** `"shapes.h"`

252. Label. [LDF 2002.10.20.] **Labels** are the only objects, other than **Shapes**, that can be put onto **Pictures**. They are created by the functions *label*( ) and *dotlabel*( ), which are currently defined for **Points** and **Paths**.

253. Label class definition. A **Label** contains a pointer to a **Point**, which is its location, a **bool** to indicate whether the label should have a dot or not, a **string** for the text of the label and a **string** for positioning the text with respect to the label. *pt* must be a pointer, because **Point** is an incomplete type here. *position* can be any of the strings used in METAFONT, i.e., `"top"`, `"bot"`, `"lft"`, `"rt"`, `"llft"`, `"ulft"`, `"lrt"`, or `"urt"`.

[LDF 2002.10.09.] **Labels** are currently only ever created on the free store.

format *Label* *int*

⟨ Define classes 253 ⟩ ≡
```
class Label {
  friend class Point;
  friend class Picture;
  Point *pt;
  bool dot;
  string text;
  string position;
public: static bool DO_LABELS;
  ⟨ Declare Label functions 255 ⟩
};
```
See also section 261.

This code is used in sections 305 and 306.

254. Static data members. `DO_LABELS` is used for globally enabling or suppressing putting **Labels** onto **Pictures**: If `DO_LABELS` is *false*, then *label*( ) and *dotlabel*( ) have no effect, i.e., no **Label** is put onto the **Picture**. Note that **Picture** has a **private** data member *do_labels*, which is for enabling or suppressing output of **Labels** for a single **Picture** (see below).

Log

[LDF 2002.10.20.] Added this section. `DO_LABELS` was formerly a global variable defined in `pspglb.web`.

⟨ Initialize **static Label** data members 254 ⟩ ≡
bool Label::`DO_LABELS` = *true*;

This code is used in section 305.

255. Declarations for Label functions. These must be defined in `points.web`, because they require operations on *pt*, and **Point** is an incomplete type in this file.

Log

[LDF 2002.10.23.] Added arguments *proj* and *factor*.

⟨ Declare **Label** functions 255 ⟩ ≡
void *output*(**const Focus** &*f*, **const unsigned short** *proj*, **real** *factor*, **const Transform** &*t*);
Label ∗*get_copy*( ) **const**;

This code is used in section 253.

256. namespace Projections.

Log

[LDF 2003.05.11.] Added `AXON`.

⟨ Declare namespace **Projections** 256 ⟩ ≡

```
namespace Projections {
  extern const unsigned short PERSP = 0;
  extern const unsigned short PARALLEL_X_Y = 1;
  extern const unsigned short PARALLEL_X_Z = 2;
  extern const unsigned short PARALLEL_Z_Y = 3;
  extern const unsigned short AXON = 4;
  extern const unsigned short ISO = 5;
};
```

This code is used in section 305.

257. External.

⟨ **extern** declaration of namespace **Projections** 257 ⟩ ≡

```
namespace Projections {
    extern const unsigned short PERSP;
    extern const unsigned short PARALLEL_X_Y;
    extern const unsigned short PARALLEL_X_Z;
    extern const unsigned short PARALLEL_Z_Y;
    extern const unsigned short AXON;
    extern const unsigned short ISO;
};
```

This code is used in section 306.

258. namespace Sorting. This namespace contains constants that are passed to **Picture** :: *output*( ) for determining how the **Shapes** on the **Picture** are sorted in order to determine the order in which they are output.

Log

[LDF 2003.05.16.] Added this namespace.

⟨ Declare namespace **Sorting** 258 ⟩ ≡

```
namespace Sorting {
    extern const unsigned short NO_SORT = 0;
    extern const unsigned short MAX_Z = 1;
    extern const unsigned short MIN_Z = 2;
    extern const unsigned short MEAN_Z = 3;
};
```

This code is used in section 305.

259. External.

⟨ **extern** declaration of namespace **Sorting** 259 ⟩ ≡

```
namespace Sorting {
    extern const unsigned short NO_SORT;
    extern const unsigned short MAX_Z;
    extern const unsigned short MIN_Z;
    extern const unsigned short MEAN_Z;
};
```

This code is used in section 306.

260. Picture.

format *Picture* *int*

261. Picture class definition. [LDF 2002.08.06.] Note that **Label** has a **public static** data member named DO_LABELS, which is used for globally enabling or suppressing putting **Labels** onto **Pictures** (see above).

Picture :: *do_labels*, on the other hand, is for enabling or suppressing the output of **Labels** for a single **Picture**. If a **Picture** is output when *do_labels* is *false* for that **Picture**, the **Labels** are not output. However, the **Labels** are still on the **Picture**. If *do_labels* is reset to *true* and the **Picture** is output again, the **Labels** *will* be output this time.

[LDF 2002.04.25.] Added *do_labels*. It's set to *true* in the constructors and can be set to *false* using *suppress_labels*( ).

⟨ Define classes 253 ⟩ +≡

```
class Picture {
    Transform transform;
    vector<Shape *> shapes;
    vector<Label *> labels;
    bool do_labels;
public: ⟨ Declare Picture functions 263 ⟩
};
```

262. Constructors.

263. Default constructor. (No arguments).

⟨ Declare **Picture** functions 263 ⟩ ≡

```
Picture();
```

See also sections 265, 267, 269, 270, 272, 274, 275, 276, 280, 283, 285, 286, 288, 289, 291, 293, 295, 298, 299, 300, and 301.

This code is used in section 261.

264.

⟨ Define **Picture** functions 264 ⟩ ≡
  Picture :: **Picture**( )
  : *do_labels*(*true*) { }

See also sections 271, 273, 277, 281, 284, 287, 290, 292, 294, 296, 417, 440, 587, 588, 589, 590, 592, 593, 594, 595, 596, 597, and 598.

This code is used in sections 305 and 633.

265. Copy constructor. !! PORTING. [LDF 2002.12.05.] Moved to `points.web` because I've moved **Picture** :: **operator**=( ) to `points.web`, so the latter is undefined in this file. I've had to do these things because of differences between the DEC compiler and the GNU compiler.

⟨ Declare **Picture** functions 263 ⟩ +≡
  Picture(**const Picture** &*p*);

266. Destructor. [LDF 2002.10.20.] **Picture** does not currently have a destructor. *clear*( ) takes care of deallocating memory and clearing *shapes* and *labels*. Defining a destructor would probably cause problems.

267. Assignment. TO DO: Add TEX macro for "PORTING". !! PORTING. [LDF 2002.12.05.] Moved to `points.web` because **Picture** :: *clear*( ) and **Label** :: *get_copy*( ) are undefined in this file. This didn't cause a problem with the DEC compiler, but it does with the GNU Compiler.

⟨ Declare **Picture** functions 263 ⟩ +≡
  void operator=(**const Picture** &*p*);

268. Adding elements.

269. Add Picture. This function must be defined in `points.web`, because it uses **Point**, which is an incompletely defined **class** here.

[LDF 2002.04.17.] It seems to be most useful to have the argument **Picture** *p* be non-**const**, in order to be able to shift it and add it to ∗*this* multiple times. For this to work, it must be possible to set *p.transform* to the identity matrix afterwards. It is possible to do this explicitly by calling *reset_transform*( ) on the **Picture** following the call to **operator**+=( ), but it's more convenient to have it done automatically. If it turns out to be useful, I can add a **const** version of this function.

Log

[LDF 2002.04.17.] Added this declaration.

⟨ Declare **Picture** functions 263 ⟩ +≡
  void operator+=(**const Picture** &*p*);

270. Add Shape.

⟨ Declare **Picture** functions 263 ⟩ +≡
  void operator+=(**Shape** ∗*s*);

271.

⟨ Define **Picture** functions 264 ⟩ +≡
```
void Picture::operator+=(Shape *s)
{
    shapes.push_back(s);
}
```

272. Add Label.

⟨ Declare **Picture** functions 263 ⟩ +≡
```
void operator+=(Label *label);
```

273.

⟨ Define **Picture** functions 264 ⟩ +≡
```
void Picture::operator+=(Label *label)
{
    labels.push_back(label);
}
```

274. Suppress Labels. [LDF 2002.04.25.] Added this function. Sometimes it's irritating to have the labels when a **Picture** is copied and transformed, and both the original and the transformed versions are output.

⟨ Declare **Picture** functions 263 ⟩ +≡
```
inline void suppress_labels()
{
    do_labels = false;
}
```

275. Unsuppress Labels.

Log

[LDF 2002.12.20.] Added this function.

⟨ Declare **Picture** functions 263 ⟩ +≡
```
inline void unsuppress_labels()
{
    do_labels = true;
}
```

276. Kill Labels.

Log

[LDF 2003.05.07.] Added this function.

⟨ Declare **Picture** functions 263 ⟩ +≡
```
void kill_labels();
```

277.

⟨ Define **Picture** functions 264 ⟩ +≡
```
void Picture::kill_labels()
{
  labels.clear();
}
```

278. Transformations. Transformations for **Pictures** are saved up, and then performed when the **Picture** is output.

279. Affine transformations.

280. Scale.

⟨ Declare **Picture** functions 263 ⟩ +≡
```
Transform scale(real x, real y = 1, real z = 1);
```

281.

⟨ Define **Picture** functions 264 ⟩ +≡
```
Transform Picture::scale(real x, real y, real z)
{
  return transform.scale(x, y, z);
}
```

282. Shift. (Translation.)

283. real version.

⟨ Declare **Picture** functions 263 ⟩ +≡
```
Transform shift(real x, real y = 0, real z = 0);
```

284. *shift*( ) returns a **Transform** representing the shift, *not* ∗*this*. This makes it possible to apply the transformation to other objects.

⟨ Define **Picture** functions 264 ⟩ +≡
```
Transform Picture::shift(real x, real y, real z)
{
  return transform.shift(x, y, z);
}
```

285. Point version. This function must defined in `points.web`, because **Point** is an incompletely defined type here.

⟨ Declare **Picture** functions 263 ⟩ +≡
```
Transform shift(const Point &p);
```

286. Rotation around the main axes. *rotate*( ) will perform rotation about the x, y and z-axes in that order if called with multiple, non-zero arguments. Rotation only about the y and/or z-axis requires one or two dummy 0 arguments so that *rotate*( ) "knows" about which axis (or axes) to rotate.

⟨ Declare **Picture** functions 263 ⟩ +≡
```
Transform rotate(const real x, const real y = 0, const real z = 0);
```

287.

⟨ Define **Picture** functions 264 ⟩ +≡
Transform Picture :: *rotate*(**const real** x, **const real** y, **const real** z)
{
return *transform.rotate*(x, y, z);
}

288. Rotation around an arbitrary axis. [LDF 2002.05.03.] This function is defined in `points.web`, because it has **Point** arguments, and **Point** is an incomplete type in this file.

Log

[LDF 2002.05.03.] Added this declaration.

[LDF 2003.05.02.] Changed name of this function from *rotate_around*( ) to *rotate*( ). This function now overloads *rotate*( ) with three **real** arguments.

⟨ Declare **Picture** functions 263 ⟩ +≡
Transform *rotate*(**const Point** &*p0*, **const Point** &*p1*, **const real** *angle* = 180);
/* Remember to add shear! */

289. Set transform.

Log

[LDF 2003.01.17.] Made non-inline and changed t from plain **Transform** to **const Transform** &.

⟨ Declare **Picture** functions 263 ⟩ +≡
Transform *set_transform*(**const Transform** &t);

290.

⟨ Define **Picture** functions 264 ⟩ +≡
Transform Picture :: *set_transform*(**const Transform** &t)
{
transform = t;
return t;
}

291. Multiplying transform.

Log

[LDF 2003.01.17.] Changed t from plain **Transform** to **const Transform** &.

⟨ Declare **Picture** functions 263 ⟩ +≡
Transform operator*=(**const Transform** &t);

292.

⟨ Define **Picture** functions 264 ⟩ +≡
```
Transform Picture::operator*=(const Transform &t)
{
  transform *= t;
  return t;
}
```

293. Show.

⟨ Declare **Picture** functions 263 ⟩ +≡
```
void show(string text = "", bool stop = false);
```

294.

⟨ Define **Picture** functions 264 ⟩ +≡
```
void Picture::show(string text, bool stop)
{
  cout << "Showing picture: " << text << "\n" << flush;
  transform.show("transform:");
  cout << "shapes.size() == " << shapes.size() << endl << flush;
  cout << "labels.size() == " << labels.size() << endl << flush;
  cout << "do_labels == " << do_labels << endl << flush;
  cout << "Showing shapes.\n";
  for (vector<Shape *>::iterator iter = shapes.begin(); iter != shapes.end(); ++iter) (**iter).show();
  if (stop) {
    cout << "Hit return to continue.\n" << flush;
    getchar();
  }
  cout << "Done showing picture.\n" << flush;
}
```

295. Show transform.

⟨ Declare **Picture** functions 263 ⟩ +≡
```
void show_transform(string text = "Transform from Picture:");
```

296.

⟨ Define **Picture** functions 264 ⟩ +≡
```
void Picture::show_transform(string text)
{
  transform.show(text);
}
```

297. Output. [LDF 2002.09.18.] Added the optional **real** arguments *min_x_proj*, *max_x_proj*, etc. The purpose of these is to suppress output of **Shapes** whose *projective_extremes* fall outside of these limits, whereby the "z" values are not currently checked. They are not set for a particular **Focus** or **Picture**, but for a particular invocation of *output*( ). I believe the default values are sufficiently generous, but they can always be changed if it turns out that they're not. Alternatively, I could store them in the **Picture** or the **Focus**, if that turns out be more convenient. They are checked in **Picture** :: *check_projection_limits*( ).

298. Focus argument.

⟨ Declare **Picture** functions 263 ⟩ +≡
```
void output(const Focus &f, const unsigned short projection = Projections::PERSP, real
    factor = 1, const unsigned short sort_value = Sorting::MAX_Z, const bool
    do_warnings = true, const real min_x_proj = −40, const real max_x_proj = 40, const real
    min_y_proj = −40, const real max_y_proj = 40, const real min_z_proj = −40, const real
    max_z_proj = 40);
```

299. No Focus argument.

⟨ Declare **Picture** functions 263 ⟩ +≡
```
void output(const unsigned short proj = Projections::PERSP, real factor = 1, const
    unsigned short sort_value = Sorting::MAX_Z, const bool do_warnings = true, const real
    min_x_proj = −40, const real max_x_proj = 40, const real min_y_proj = −40, const real
    max_y_proj = 40, const real min_z_proj = −40, const real max_z_proj = 40);
```

300. Clear. Defined in `points.web`.

⟨ Declare **Picture** functions 263 ⟩ +≡
```
void clear();
```

301. Reset transform. [LDF 2002.04.17.] Added this function.

⟨ Declare **Picture** functions 263 ⟩ +≡
```
inline void reset_transform()
{
  transform.reset();
}
```

302. Global variables.

⟨ Global variables 18 ⟩ +≡
```
Picture current_picture;
```

303.

⟨ Declarations for the header file 21 ⟩ +≡
```
extern Picture current_picture;
```

304. Putting Picture and Label together.

305. This is what's compiled.

⟨ Include files 6 ⟩
⟨ Version control identifier 5 ⟩
⟨ Declare namespace **Projections** 256 ⟩
⟨ Declare namespace **Sorting** 258 ⟩
⟨ Define classes 253 ⟩
⟨ Initialize **static Label** data members 254 ⟩
⟨ Global variables 18 ⟩
⟨ Define **Picture** functions 264 ⟩

306. This is what's written to `pictures.h`.

⟨ `pictures.h` 306 ⟩ ≡
⟨ **extern** declaration of namespace **Projections** 257 ⟩
⟨ **extern** declaration of namespace **Sorting** 259 ⟩
⟨ Define classes 253 ⟩
⟨ Declarations for the header file 21 ⟩

307. Point (`points.web`).
[LDF 2002.10.20.] **Point** is the most basic drawable (not fillable!) type. All of the other **Shapes** contain **Points** and are ultimately defined by their **Points** and the relationships among them. It is therefore understandable that `points.web` is by far the largest of the source files of 3DLDF and that **Point** has the most functions of any class in 3DLDF. Many of the functions in the other classes do little more than apply the **Point** version of the function to their **Points**.

Log

[LDF 2003.11.12.] Removed the version control identifiers from the CWEB files for the distribution of 3DLDF 1.1. They're still used in my development versions.

[LDF 2003.12.10.] Put the version control identifiers back into my release versions for 3DLDF 1.1.4. I've already put some of them back in, now I'm doing the rest of them. However, the release versions are now in their own RCS repository.

⟨ Version control identifier 5 ⟩ +≡
static string *rcs_id* = `"$Id:␣points.web,v␣1.6␣2004/01/12␣21:32:01␣lfinsto1␣Exp␣$"`;

308. Include files.

⟨ Include files 6 ⟩ +≡
```
#include "loader.h"
#include "pspglb.h"
#include "creatnew.h"
#include "io.h"
#include "colors.h"
#include "transfor.h"
#include "shapes.h"
#include "pictures.h"
```

309. Point class definition. [LDF 2002.10.20.]

- *world_coordinates* contains the coordinates of the **Point** in the global coordinate system.
- *user_coordinates* and *view_coordinates* are not currently used.
- *user_coordinates* is intended for use with a user-defined coordinate system. For example, it may be convenient to define a coordinate system based on a plane defined by an object in a drawing. The user-defined coordinate system will be defined in terms of the global coordinate system and the *world_coordinates* can be derived from the *user_coordinates* by using the appropriate transformation.
- *projective_coordinates* is used for projecting the three-dimensional **Points** onto two-dimensions for output. Currently, the only projection routines are for perspective and parallel projection. I plan to add others (axionometric, etc.) soon.
- *transform* is used for storing the transformations that are applied the **Point**. It is not necessary to update *world_coordinates* (or *user_coordinates*, if they're being used) every time a **Point** is transformed. The transformations can be saved up and their result applied to the **Point** when needed. This will be when the **Point** or the **Shape** or **Label** containing it is output, or when a function (such as *get_x*( )) requires up-to-date coordinate values.
- *drawdot_value*, *drawdot_color*, and *pen* are used in the drawing and undrawing functions and in **Point** :: *output*( ). They are explained below.

- *projective_extremes* is used in outputting **Pictures**. It's for culling **Points** that are invisible using a particular **Focus**, or that lie outside the boundaries passed as arguments to **Picture** :: *output*( ).
- *do_output* is for enabling or suppressing output of a **Point**. It's needed when a **Point** has been culled, as described above. Culling does not actually remove a **Point** (or any other **Shape**) from a **Picture**, so a way of suppressing output is needed. However, it must be possible to enable output again, because the **Picture** may be output again using a different **Focus** and/or different values for the boundaries.
- *measurement_units* is a **string** that is attached to the numerical values of the *projective_coordinates* when the METAPOST code is output to *out_stream*. It is currently `"cm"` and at the present time (2002.10.20), it's not a good idea to use more than one value for a single drawing. Changing this will only become urgent when I start writing the input routine. TO DO.

Log

[LDF 2003.04.01.] Added `WORLD_VALUES`, `PROJ_VALUES`, `USER_VALUES`, and `VIEW_VALUES`. They are used in *label*( ) for labelling **Points** using the values in *world_coordinates*, *projective_coordinates*, etc.

[LDF 2003.05.06.] Changed `WORLD_VALUE`, `PROJ_VALUE`, `USER_VALUE`, and `VIEW_VALUE` to `WORLD_VALUES`, `PROJ_VALUES`, `USER_VALUES`, and `VIEW_VALUES`. Added `WORLD_VALUES_X_Y`, `PROJ_VALUES_X_Y`, `USER_VALUES_X_Y`, and `VIEW_VALUES_X_Y` for suppressing the z-coordinate.

[LDF 2003.05.20.] Added `WORLD_VALUES_Z`.

---

format *Point Shape*

⟨ Define **class Point** 309 ⟩ ≡

```
class Point : protected Shape {
  friend Transform Transform::align_with_axis(Point, Point, char);
private: Transform transform;
  bool on_free_store;
  valarray<real> world_coordinates;
  valarray<real> user_coordinates;
  valarray<real> view_coordinates;
  valarray<real> projective_coordinates;
  signed short drawdot_value;
  const Color *drawdot_color;
  string pen;
protected:   /* LDF 2002.09.18. Added projective_extremes. It contains the minimum and maximum
        values for x, y, and z in projective_coordinates. */
  valarray<real> projective_extremes;
  bool do_output;   /* LDF 2002.09.18. Added this data member. */
public: static string measurement_units;
  static real CURR_Y;
  static real CURR_Z;
  static const short WORLD_VALUES;
  static const short PROJ_VALUES;
  static const short USER_VALUES;
  static const short VIEW_VALUES;
  static const short WORLD_VALUES_X_Y;
  static const short PROJ_VALUES_X_Y;
  static const short USER_VALUES_X_Y;
  static const short VIEW_VALUES_X_Y;
  static const short WORLD_VALUES_Z;
  ⟨ Declare Point constructors 324 ⟩
  ⟨ Declare Point functions 329 ⟩
};
```

This code is used in sections 633 and 634.

310.

Log

LDF 2003.04.01. Added initialization of `WORLD_VALUES`, `PROJ_VALUES`, `USER_VALUES`, and `VIEW_VALUES`. They are used in *label*( ) for labelling **Points** using the values in *world_coordinates*, *projective_coordinates*, etc. !! KLUDGE: Using the macro `SHRT_MAX` because the **numeric_limits** template doesn't seem to be available under GNU/Linux using GCC, at least not on the computer I'm using.

LDF 2003.05.06. Added initialization of `WORLD_VALUES_X_Y`, `PROJ_VALUES_X_Y`, `USER_VALUES_X_Y`, and `VIEW_VALUES_X_Y`

LDF 2003.05.22. BUG FIX: Changed `WORLD_VALUES_Z` so that it's one less than `VIEW_VALUES_X_Y`. Previously, it had the same value.

⟨ Define **static Point** data members 310 ⟩ ≡

```
string Point::measurement_units = "cm";
real Point::CURR_Y = 0;
real Point::CURR_Z = 0;
const short Point::WORLD_VALUES = SHRT_MAX;
const short Point::PROJ_VALUES = WORLD_VALUES − 1;
const short Point::USER_VALUES = WORLD_VALUES − 2;
const short Point::VIEW_VALUES = WORLD_VALUES − 3;
const short Point::WORLD_VALUES_X_Y = WORLD_VALUES − 4;
const short Point::PROJ_VALUES_X_Y = WORLD_VALUES − 5;
const short Point::USER_VALUES_X_Y = WORLD_VALUES − 6;
const short Point::VIEW_VALUES_X_Y = WORLD_VALUES − 7;
const short Point::WORLD_VALUES_Z = WORLD_VALUES − 8;
```

This code is used in section 633.

311. Type definitions and utility structures. Some of the types are simple enough to be defined using **typedef**, but others require **struct** definitions.

Log

[LDF 2002.04.10.] Added these formatting instructions. They are duplicated using "@s" in `cwdriver.web`.

```
format point_pair Point
format bool_point Point
format bool_point_pair Point
format bool_point_quadruple Point
format bool_real_point Point
```

312. point_pair and bool_point_pair.

⟨ Type definitions 15 ⟩ +≡

```
typedef pair⟨Point, Point⟩ point_pair;
typedef pair⟨bool_point, bool_point⟩ bool_point_pair;
```

313. bool_point.

Log

LDF 2002.04.15. Added this section. **bool_point** was formerly a simple **typedef**. I've had to change it to a **struct**, in order for **Point**::*intersection_points*( ) to return one.

LDF 2003.05.30. Removed the definition of the default constructor to the new section ⟨ Define **bool_point** functions 314 ⟩. See below for an explanation.

⟨ Type definitions 15 ⟩ +≡

```
struct bool_point {
  bool b;
  Point pt;
  bool_point();
  bool_point(bool bb, const Point &ppt)
  : b(bb), pt(ppt) { }
  void operator=(const bool_point &bp)
  {
    b = bp.b;
    pt = bp.pt;
  }
};
```

314.

Log

LDF 2003.05.30. Added this section, and the definition of **bool_point**(**void**). Previously, b and pt were not set, so their values were unpredictable. I had to remove the definition from the declaration of **bool_point**, because `INVALID_POINT` isn't defined, when the declaration is read by the compiler.

⟨ Define **bool_point** functions 314 ⟩ ≡

```
bool_point::bool_point()
{
  b = false;
  pt = INVALID_POINT;
}
```

This code is cited in section 313.

This code is used in section 633.

315. bool_point_quadruple. It would be possible to define this as a **pair** of **pairs**, but then the individual element s would be nested inconveniently.

⟨ Type definitions 15 ⟩ +≡

```
struct bool_point_quadruple {
  bool_point first;
  bool_point second;
  bool_point third;
  bool_point fourth;
  bool_point_quadruple();
  bool_point_quadruple(bool_point a, bool_point b, bool_point c, bool_point d)
  : first(a), second(b), third(c), fourth(d) { }
  void operator=(const bool_point_quadruple &arg)
  {
    first.b = arg.first.b;
    first.pt = arg.first.pt;
    second.b = arg.second.b;
    second.pt = arg.second.pt;
    third.b = arg.third.b;
    third.pt = arg.third.pt;
    fourth.b = arg.fourth.b;
    fourth.pt = arg.fourth.pt;
  }
};
```

316. Default Constructor for bool_point_quadruple.

Log

LDF 2003.06.1. Added this section. Redefined the default constructor **bool_point_quadruple**(**void**), so that *first*, *second*, *third*, and *fourth* are all set to `INVALID_BOOL_POINT`. In order to do this, it was necessary to remove the definition from the declaration of **bool_point_quadruple**, because when the compiler sees it, `INVALID_BOOL_POINT` isn't defined yet.

⟨ Define **bool_point_quadruple** functions 316 ⟩ ≡

```
bool_point_quadruple::bool_point_quadruple()
  : first(INVALID_BOOL_POINT), second(INVALID_BOOL_POINT), third(INVALID_BOOL_POINT),
      fourth(INVALID_BOOL_POINT) { }
```

This code is used in section 633.

317. bool_real_point. [LDF 2002.04.10.] Added this type. **Line** :: *intersection_point*( ) returns a **bool_real_point**. I may change **Point** :: *intersection_point*( ) so that it calls **Line** :: *intersection_point*( ) and returns a **bool_real_point**, too.

[LDF 2002.10.26.] !! KLUDGE: `\newline` inserted in the text above to avoid overfull boxes.

⟨ Type definitions 15 ⟩ +≡

```
struct bool_real_point {
  bool b;
  real r;
  Point pt;
  bool_real_point();   /* Default constructor. */
  bool_real_point(const bool_real_point &brp)
  : b(brp.b), r(brp.r), pt(brp.pt) { }   /* Copy constructor. */
  bool_real_point(const bool &bb, const real &rr, const Point &ppt)
  : b(bb), r(rr), pt(ppt) { }   /* Construcor with bool, real, and Point arguments. */
  void operator=(const bool_real_point &brp)   /* Assignment operator. */
  {
    b = brp.b;
    r = brp.r;
    pt = brp.pt;
  }
};
```

318. Default Constructor for bool_real_point.

Log

LDF 2003.06.1. Added this section. Redefined the default constructor **bool_real_point**(**void**), so that *b* is set to *false*, *r* is set to `INVALID_REAL`, and *pt* is set to `INVALID_BOOL_POINT`. In order to do this, it was necessary to remove the definition from the declaration of **bool_real_point**, because when the compiler sees it, `INVALID_REAL` and `INVALID_POINT` aren't defined yet.

⟨ Define **bool_real_point** functions 318 ⟩ ≡

```
bool_real_point::bool_real_point()
  : b(false), r(INVALID_REAL), pt(INVALID_POINT) { }
```

This code is used in section 633.

319. Global constants. [LDF 2002.09.25.] Changed this section. I now know that *consts* have internal linkage by default and that I must declare them with **extern** in order to give them external linkage.

⟨ Global constants 22 ⟩ +≡

```
extern const Point INVALID_POINT(INVALID_REAL, INVALID_REAL, INVALID_REAL);
extern const Point origin(0, 0, 0);
extern const bool_point INVALID_BOOL_POINT(false, INVALID_POINT);
extern const bool_point_pair INVALID_BOOL_POINT_PAIR(INVALID_BOOL_POINT,
    INVALID_BOOL_POINT);
extern const bool_real_point INVALID_BOOL_REAL_POINT(false, INVALID_REAL, INVALID_POINT);
extern const bool_point_quadruple INVALID_BOOL_POINT_QUADRUPLE(INVALID_BOOL_POINT,
    INVALID_BOOL_POINT, INVALID_BOOL_POINT, INVALID_BOOL_POINT);
```

320.

⟨ Declarations for the header file 21 ⟩ +≡

```
extern const Point INVALID_POINT;
extern const Point origin;
extern const bool_point INVALID_BOOL_POINT;
extern const bool_point_pair INVALID_BOOL_POINT_PAIR;
extern const bool_real_point INVALID_BOOL_REAL_POINT;
extern const bool_point_quadruple INVALID_BOOL_POINT_QUADRUPLE;
```

321. Constructors and setting functions.

322. The **valarrays** I use for the various sets of coordinates can be declared in the **class** declaration, but neither can their size be set nor can they be initialized. *null_coordinates* is defined in `pspglb.web` and is a **valarray** of **reals** with 4 elements = 0. Setting *world_coordinates*, etc. to *null_coordinates* makes them the right size.

323. Initialize coordinates and limits.

Log

[LDF 2002.4.3.] Now setting *world_coordinates*[3], *user_coordinates*[3], and *view_coordinates*[3] = 1. It fixes a bug that showed up when I tried to shift a **Point** with coordinates ≡ 0.

```
⟨ Initialize coordinates and limits 323 ⟩ ≡
#ifdef __DECCXX
  world_coordinates = null_coordinates;
  user_coordinates = null_coordinates;
  view_coordinates = null_coordinates;
  projective_coordinates = null_coordinates;
#endif
#ifdef __GNUC__
  world_coordinates.resize(4, 0);
  user_coordinates.resize(4, 0);
  view_coordinates.resize(4, 0);
  projective_coordinates.resize(4, 0);
#if 1
  pen = "";
#endif
  transform.reset();
#endif
  world_coordinates[3] = 1;
  user_coordinates[3] = 1;
  view_coordinates[3] = 1;
  projective_extremes.resize(6, 0);
```

This code is used in sections 325, 328, and 332.

324. Default version. No arguments.

```
⟨ Declare Point constructors 324 ⟩ ≡
  Point();
```

See also sections 327 and 331.

This code is used in section 309.

325.

⟨ Define **Point** constructors 325 ⟩ ≡
```
Point::Point()
{
   ⟨ Initialize coordinates and limits 323 ⟩
   on_free_store = false;
   do_output = true;
}
```
See also sections 328 and 332.

This code is used in section 633.

326. Three real values.

327. Constructor.

Log

[LDF 2002.12.01.] Made arguments **const**.

⟨ Declare **Point** constructors 324 ⟩ +≡
```
Point(const real x, const real y = CURR_Y, const real z = CURR_Z);
```

328.

⟨ Define **Point** constructors 325 ⟩ +≡
```
Point::Point(const real x, const real y, const real z)
{
   ⟨ Initialize coordinates and limits 323 ⟩
   on_free_store = false;
   do_output = true;
#if 0   /* [LDF 2002.10.23.] user_transform is not currently in use. It is intended for use in implementing
        user-defined coordinate systems. */
   if (user_transform.is_identity())
#endif
      world_coordinates[0] = x;
   world_coordinates[1] = y;
   world_coordinates[2] = z;
   world_coordinates[3] = 1;
}
```

329. Setting function.

Log

[LDF 2002.12.01.] Made arguments **const**.

[LDF 2003.03.25.] Changed this function, so that it returns **this* instead of **void**. This makes it possible to chain invocations of this function.

⟨ Declare **Point** functions 329 ⟩ ≡
```
const Point &set(const real x, const real y = CURR_Y, const real z = CURR_Z);
```
See also sections 333, 338, 340, 342, 344, 346, 349, 350, 352, 355, 357, 360, 362, 365, 367, 370, 372, 375, 377, 380, 382, 384, 385, 387, 389, 393, 396, 398, 400, 401, 404, 406, 408, 412, 414, 419, 421, 436, 438, 442, 446, 448, 450, 454, 456, 458, 460, 463, 464, 466, 467, 469, 470, 472, 473, 475, 477, 482, 484, 486, 488, 489, 491, 493, 495, 501, 505, 507, 510, 512, 518, 521, 523, 525, 527, 529, 532, 536, 538, 540, 542, 544, 546, 548, 551, 553, 555, 557, 560, 567, 569, 572, and 573.

This code is used in section 309.

330.

⟨ Define **Point** functions 330 ⟩ ≡

```
const Point &Point::set(const real x, const real y, const real z)
{
   Point p(x, y, z);
   *this = p;
   do_output = true;
   return *this;    /* LDF 2003.03.25. Added this. Formerly, the return value was void. */
}
```

See also sections 334, 339, 341, 343, 345, 347, 351, 356, 358, 361, 363, 366, 368, 371, 373, 376, 378, 381, 383, 386, 388, 390, 391, 394, 395, 397, 399, 405, 407, 409, 413, 415, 420, 422, 437, 443, 444, 445, 447, 449, 451, 455, 457, 459, 461, 476, 478, 483, 485, 487, 490, 492, 494, 496, 502, 506, 508, 511, 513, 519, 522, 524, 526, 528, 530, 533, 537, 539, 541, 543, 545, 547, 549, 552, 554, 556, 561, 562, 563, 564, 565, 566, 568, 570, 574, 575, 576, 577, 578, 579, 580, 581, 582, 583, 584, 585, 645, 647, 681, 760, 823, 824, 832, 833, 841, 842, 860, 861, 945, 948, 949, and 950.

This code is used in sections 633, 657, 694, and 980.

331. Copy constructor.

⟨ Declare **Point** constructors 324 ⟩ +≡

```
Point(const Point &p);
```

332.

⟨ Define **Point** constructors 325 ⟩ +≡

```
Point::Point(const Point &p)
{
   ⟨ Initialize coordinates and limits 323 ⟩
   *this = p;
   on_free_store = false;
   do_output = true;
}
```

333. Setting function. [LDF 2002.10.23.] This function is unnecessary, because it does nothing that the assignment operator can't do. However, I've tried to use *set*( ) a couple of times with a **Point** argument, so it's convenient to have it. If nothing else, it prevents compilation from failing occasionally.

Log

[LDF 2002.10.23.] Added this function.

⟨ Declare **Point** functions 329 ⟩ +≡

```
void set(const Point &p);
```

334.

⟨ Define **Point** functions 330 ⟩ +≡
  void Point :: *set*(**const Point** &*p*)
  {
    ∗*this* = *p*;
    do_output = *true*;
  }

335. Pseudo-constructor for dynamic allocation. *create_new* < **Point** > ( ) is meant to be used instead of **new** ( ) for dynamic allocation of **Points**. It calls the default constructor (without arguments) and then sets *on_free_store* to *true*.

[LDF 2002.10.11.] It is used in various **Point** functions, and in **Path** and some classes derived from **Path**, currently **Ellipse**, **Reg_Polygon**, and **Rectangle**. It is intended that objects of these types be declared, i.e., unlike **Reg_Cl_Plane_Curve**, they are not meant to be used only as base classes. **Reg_Cl_Plane_Curve** does not use *create_new* < **Point** > ( ) and it is unlikely that other classes of this kind will use it.

Log

[LDF 2003.12.30.] Replaced the non-template *create_new_point*( ) functions with versions of the *create_new*( ) template function. I believe the definitions shouldn't be necessary, but at present they are. Without them, the templates aren't instantiated.

[LDF 2003.12.30.] Changed the argument. It's now a **const Point** ∗.

[LDF 2003.12.30.] Removed default argument "0", because this caused a compiler error when using the DEC C++ compiler. Apparently, it suffices to declare a default argument in the template declaration.

---

336. Pointer argument.

⟨ Declare non-member template functions for **Point** 336 ⟩ ≡
  Point ∗*create_new*(**const Point** ∗*p*);

See also section 337.
This code is used in sections 633 and 634.

337. Referece argument.

Log

[LDF 2003.12.30.] Changed argument from **Point** to **const Point** &.

⟨ Declare non-member template functions for **Point** 336 ⟩ +≡
  Point ∗*create_new*(**const Point** &*p*);

338. Destructor.

Log

[LDF 2003.08.27.] Added a **virtual** destructor with an empty definition, because GCC with the "`-Wall`" option issued the following warning: "'class Point' has virtual functions but non-virtual destructor".

⟨ Declare **Point** functions 329 ⟩ +≡
  virtual ∼**Point**( );

339.

⟨ Define **Point** functions 330 ⟩ +≡

```
Point::~Point()
{ }
```

340. Assignment.

Log

[LDF 2003.03.25.] Changed this function, so that it returns *p* instead of **void**. This makes it possible to chain invocations of this function.

⟨ Declare **Point** functions 329 ⟩ +≡

```
const Point &operator=(const Point &p);
```

341.

⟨ Define **Point** functions 330 ⟩ +≡

```
const Point &Point::operator=(const Point &p)
{
    transform = p.transform;
    drawdot_value = p.drawdot_value;
    drawdot_color = p.drawdot_color;
#if 1
    pen = p.pen;
#endif
    world_coordinates = p.world_coordinates;
    user_coordinates = p.user_coordinates;
    view_coordinates = p.view_coordinates;
    projective_coordinates = p.projective_coordinates;
    projective_extremes = p.projective_extremes;    /* LDF 2002.09.18. Added this line. I think that it
        shouldn't be necessary to do this, but I've added it just to be sure. */
    do_output = true;    /* [LDF 2002.09.18.] Probably not necessary, but I might as well. */
    return p;    /* LDF 2003.03.25. Added this. Formerly, the return value was void. */
}
```

342. Set on free store.

Log

[LDF 2003.12.29.] Made non-inline.

⟨ Declare **Point** functions 329 ⟩ +≡

```
virtual bool set_on_free_store(bool b = true);
```

343.

⟨ Define **Point** functions 330 ⟩ +≡

```
bool Point::set_on_free_store(bool b)
{
  on_free_store = b;
  return b;
}
```

344. Clear. I need this function because it's a **virtual** function in **Shape**.

Log

[LDF 2002.10.27.] Redefined this function. Formerly, it was inline and empty. Now it sets all of the x, y, and z coordinates to 0, and resets *transform*. It doesn't seem worthwhile to set *drawdot_value*, *drawdot_color*, or *pen* to any particular values.

⟨ Declare **Point** functions 329 ⟩ +≡

```
void clear();
```

345.

⟨ Define **Point** functions 330 ⟩ +≡

```
void Point::clear()
{
  for (int i = 0; i < 4; i++) {
    world_coordinates[i] = user_coordinates[i] = view_coordinates[i] = projective_coordinates[i] = 0;
  }
  transform.reset();
  return;
}
```

346. Clean.

⟨ Declare **Point** functions 329 ⟩ +≡

```
void clean(int factor = 1);
```

347.

⟨ Define **Point** functions 330 ⟩ +≡

```
void Point::clean(int factor)
{
  apply_transform();
  real eps = epsilon() * factor;
  for (int i = 0; i < 4; i++)
    if (fabs(world_coordinates[i]) < eps) world_coordinates[i] = 0.0;
}
```

348. Returning elements and information.

349. Is identity.

⟨ Declare **Point** functions 329 ⟩ +≡

```
inline bool is_identity()
{
  return (transform.is_identity());
}
```

350. Epsilon.

Log

[LDF 2004.1.2.] Now returning different values, depending on whether **real** is **float** or **double**. TO DO: Try to find out what values would be best. It will be necessary to check how good the value for **double** is.

[LDF 2004.1.2.] Made *epsilon*( ) non-inline.

⟨ Declare **Point** functions 329 ⟩ +≡

```
static real epsilon();
```

351.

⟨ Define **Point** functions 330 ⟩ +≡

```
real Point::epsilon()
{
#if LDF_REAL_DOUBLE
  return .000000001;
#else
  return .00001;
#endif
}
```

352. Get Line. Defined in `lines.web`. Must be defined there, because **Line** is an incomplete type here.

[LDF 2002.04.12.] Removed this function to `lines.web`.

⟨ Declare **Point** functions 329 ⟩ +≡

```
Line get_line(const Point &pt) const;
```

353. Getting coordinates. ?? Change *get_x*( ), etc., back to **inline**??

354. Get all coordinates. [LDF 2002.09.19.] Added this function. ?? [LDF 2002.12.01.] Can I make the **Focus** ∗ argument **const**? What is the syntax for a pointer to a **const**, as opposed to a **const** pointer? Look up!! Make sure I change this is `3DLDF.texi` if I change it here!

355. Non-const version. [LDF 2002.09.19.] Added this function.

⟨ Declare **Point** functions 329 ⟩ +≡

```
valarray⟨real⟩ get_all_coords(char coords = 'w', const bool do_persp = true, const bool
    do_apply = true, Focus *f = 0, const unsigned short proj = Projections::PERSP, real
    factor = 1);
```

356.

⟨ Define **Point** functions 330 ⟩ +≡

```
valarray⟨real⟩ Point::get_all_coords(char coords, const bool do_persp, const bool do_apply, Focus
      *f, const unsigned short proj, real factor)
{
  if (do_apply) apply_transform();
  coords = tolower(coords);
  if (coords ≡ 'w') return world_coordinates;
  else if (coords ≡ 'v') return view_coordinates;
  else if (coords ≡ 'u') return user_coordinates;
  else if (coords ≡ 'p') {
    if (do_persp) {
      if (f ≡ 0) f = &default_focus;
      project(*f, proj, factor);
    }
    return projective_coordinates;
  }
  else {
    cerr ≪ "ERROR!␣In␣Point::get_all_coords():\n" ≪
        "Argument␣coords␣has␣invalid␣value:␣" ≪ coords ≪
        ".␣Returning␣world␣coordinates.\n" ≪ flush;
    return world_coordinates;
  }
}
```

357. const version. [LDF 2002.09.19.] Added this function.

⟨ Declare **Point** functions 329 ⟩ +≡

```
valarray⟨real⟩ get_all_coords(char coords = 'w', const bool do_persp = true, const bool
    do_apply = true, Focus *f = 0, const unsigned short proj = Projections::PERSP, real
    factor = 1) const;
```

358.

⟨ Define **Point** functions 330 ⟩ +≡

```
valarray⟨real⟩ Point::get_all_coords(char coords, const bool do_persp, const bool do_apply, Focus
      *f, const unsigned short proj, real factor) const
{
  Point p(*this);
  valarray⟨real⟩ v = p.get_all_coords(coords, do_persp, do_apply, f);
  return v;
}
```

359. Get coord. [LDF 2002.09.14.] Added *get_coord*( ). Fixing a bug that caused *get_x*('p'), etc., to call *project*( ) multiple times when doing **Path**::*output*( ).

360. Non-const version. [LDF 2002.10.27.] The argument *c* refers to either the x, y, z, or w coordinate.

⟨ Declare **Point** functions 329 ⟩ +≡

```
real get_coord(char c, char coords = 'w', const bool do_persp = true, const bool
    do_apply = true, Focus *f = 0, const unsigned short proj = Projections::PERSP, real
    factor = 1);
```

361.

⟨ Define **Point** functions 330 ⟩ +≡

```
real Point::get_coord(char c, char coords, const bool do_persp, const bool do_apply, Focus *f, const
        unsigned short proj, real factor)
{
  if (do_apply) apply_transform();
  if (f ≡ 0) f = &default_focus;
  unsigned short ctr;
  c = tolower(c);
  if (c ≡ 'x') ctr = 0;
  else if (c ≡ 'y') ctr = 1;
  else if (c ≡ 'z') ctr = 2;
  else if (c ≡ 'w') ctr = 3;
  else {
    cerr ≪ "ERROR!␣In␣Point::get_coord():␣" ≪ "Invalid␣c␣argument:␣" ≪ c ≪ ".\n" ≪
        "Using␣x\n" ≪ flush;
    ctr = 0;
  }
  coords = tolower(coords);
  if (coords ≡ 'w') return world_coordinates[ctr];
  else if (coords ≡ 'u') return user_coordinates[ctr];
  else if (coords ≡ 'p') {
    if (do_persp) project(*f, proj, factor);
    return projective_coordinates[ctr];
  }
  else if (coords ≡ 'v') return view_coordinates[ctr];
  else {
    cerr ≪ "ERROR!␣In␣Point::get_coord().␣Invalid␣argument␣char␣coords:␣" ≪ coords ≪
        endl ≪ "Returning␣INVALID_REAL.\n\n" ≪ flush;
    return INVALID_REAL;
  }
}
```

362. const version.

⟨ Declare **Point** functions 329 ⟩ +≡

```
real get_coord(char c, char coords = 'w', const bool do_persp = true, const bool
    do_apply = true, Focus *f = 0, const unsigned short proj = Projections::PERSP, real
    factor = 1) const;
```

363.

⟨ Define **Point** functions 330 ⟩ +≡
real Point :: *get_coord* (**char** *c*, **char** *coords*, **const bool** *do_persp*, **const bool** *do_apply*, **Focus** $*f$, **const unsigned short** *proj*, **real** *factor*) **const**
{
Point *p*(**this*);
return *p.get_coord*(*c*, *coords*, *do_persp*, *do_apply*, *f*, *proj*, *factor*);
}

364. Get x.

365. Non-const version.

⟨ Declare **Point** functions 329 ⟩ +≡
real *get_x*(**char** *coords* = 'w', **const bool** *do_persp* = *true*, **const bool** *do_apply* = *true*, **Focus** $*f = 0$, **const unsigned short** *proj* = **Projections** :: PERSP, **real** *factor* = 1);

366.

⟨ Define **Point** functions 330 ⟩ +≡
real Point :: *get_x*(**char** *coords*, **const bool** *do_persp*, **const bool** *do_apply*, **Focus** $*f$, **const unsigned short** *proj*, **real** *factor*)
{
return *get_coord*('x', *coords*, *do_persp*, *do_apply*, *f*, *proj*, *factor*);
}

367. const version.

⟨ Declare **Point** functions 329 ⟩ +≡
real *get_x*(**char** *coords* = 'w', **const bool** *do_persp* = *true*, **const bool** *do_apply* = *true*, **Focus** $*f = 0$, **const unsigned short** *proj* = **Projections** :: PERSP, **real** *factor* = 1) **const**;

368.

⟨ Define **Point** functions 330 ⟩ +≡
real Point :: *get_x*(**char** *coords*, **const bool** *do_persp*, **const bool** *do_apply*, **Focus** $*f$, **const unsigned short** *proj*, **real** *factor*) **const**
{
return *get_coord*('x', *coords*, *do_persp*, *do_apply*, *f*, *proj*, *factor*);
}

369. Get y.

370. Non-const version.

⟨ Declare **Point** functions 329 ⟩ +≡
real *get_y*(**char** *coords* = 'w', **const bool** *do_persp* = *true*, **const bool** *do_apply* = *true*, **Focus** $*f = 0$, **const unsigned short** *proj* = **Projections** :: PERSP, **real** *factor* = 1);

371.

⟨ Define **Point** functions 330 ⟩ +≡
```
real Point::get_y(char coords, const bool do_persp, const bool do_apply, Focus *f, const unsigned
        short proj, real factor)
  {
    return get_coord('y', coords, do_persp, do_apply, f, proj, factor);
  }
```

372. const version.

⟨ Declare **Point** functions 329 ⟩ +≡
```
real get_y(char coords = 'w', const bool do_persp = true, const bool do_apply = true, Focus
    *f = 0, const unsigned short proj = Projections::PERSP, real factor = 1) const;
```

373.

⟨ Define **Point** functions 330 ⟩ +≡
```
real Point::get_y(char coords, const bool do_persp, const bool do_apply, Focus *f, const unsigned
        short proj, real factor) const
  {
    return get_coord('y', coords, do_persp, do_apply, f, proj, factor);
  }
```

374. Get z.

375. Non-const version.

⟨ Declare **Point** functions 329 ⟩ +≡
```
real get_z(char coords = 'w', const bool do_persp = true, const bool do_apply = true, Focus
    *f = 0, const unsigned short proj = Projections::PERSP, real factor = 1);
```

376.

⟨ Define **Point** functions 330 ⟩ +≡
```
real Point::get_z(char coords, const bool do_persp, const bool do_apply, Focus *f, const unsigned
        short proj, real factor)
  {
    return get_coord('z', coords, do_persp, do_apply, f, proj, factor);
  }
```

377. const version.

⟨ Declare **Point** functions 329 ⟩ +≡
```
real get_z(char coords = 'w', const bool do_persp = true, const bool do_apply = true, Focus
    *f = 0, const unsigned short proj = Projections::PERSP, real factor = 1) const;
```

378.

⟨ Define **Point** functions 330 ⟩ +≡
```
real Point::get_z(char coords, const bool do_persp, const bool do_apply, Focus *f, const unsigned
        short proj, real factor) const
  {
    return get_coord('z', coords, do_persp, do_apply, f, proj, factor);
  }
```

379. Get w.

380. Non-const version.

⟨ Declare **Point** functions 329 ⟩ +≡
real *get_w*(**char** *coords* = 'w', **const bool** *do_persp* = *true*, **const bool** *do_apply* = *true*, **Focus** $*f = 0$, **const unsigned short** *proj* = **Projections**::PERSP, **real** *factor* = 1);

381.

⟨ Define **Point** functions 330 ⟩ +≡
```
real Point::get_w(char coords, const bool do_persp, const bool do_apply, Focus *f, const unsigned
        short proj, real factor)
  {
    return get_coord('w', coords, do_persp, do_apply, f, proj, factor);
  }
```

382. const version.

⟨ Declare **Point** functions 329 ⟩ +≡
real *get_w*(**char** *coords* = 'w', **const bool** *do_persp* = *true*, **const bool** *do_apply* = *true*, **Focus** $*f = 0$, **const unsigned short** *proj* = **Projections**::PERSP, **real** *factor* = 1) **const**;

383.

⟨ Define **Point** functions 330 ⟩ +≡
```
real Point::get_w(char coords, const bool do_persp, const bool do_apply, Focus *f, const unsigned
        short proj, real factor) const
  {
    return get_coord('w', coords, do_persp, do_apply, f, proj, factor);
  }
```

384. Get transform.

Log

[LDF 2002.10.27.] Made this function **const**.

⟨ Declare **Point** functions 329 ⟩ +≡
```
inline Transform get_transform() const
  {
    return transform;
  }
```

385. Get copy.

Log

[LDF 2002.10.27.] Made this function **const**.

⟨ Declare **Point** functions 329 ⟩ +≡
Shape **get_copy*( ) **const**;

386.

⟨ Define **Point** functions 330 ⟩ +≡
Shape ∗**Point**::*get_copy*( ) **const** { **Point** ∗*p* = *create_new* < **Point** > (0);
∗*p* = ∗*this*;
return static_cast⟨**Shape** ∗⟩(*p*); }

387. Is on free store.

Log

[LDF 2004.01.06.] Made non-inline.

⟨ Declare **Point** functions 329 ⟩ +≡
bool *is_on_free_store*( ) **const**;

388.

⟨ Define **Point** functions 330 ⟩ +≡
bool Point::*is_on_free_store*( ) **const**
{
return *on_free_store*;
}

389. Slope. [LDF 2002.10.27.] *slope*( ) returns the slope of the *trace* of the line from ∗*this* to *p* on the plane indicated by the **char** arguments *m* and *n*. These should be 'x', 'y', 'z', 'X', 'Y', or 'Z'.

Log

[LDF 2002.10.27.] Now using *world_coordinates* directly instead of "get" functions.
[LDF 2002.10.27.] Changed argument *p* from **const Point** & to **Point**.

⟨ Declare **Point** functions 329 ⟩ +≡
real *slope*(**Point** *p*, **char** *m* = 'x', **char** *n* = 'y') **const**;

390.

⟨ Define **Point** functions 330 ⟩ +≡

```
real Point::slope(Point p, char m, char n) const { bool DEBUG = false;   /* true */
    Point a(*this);
    a.apply_transform();
    p.apply_transform();
    if (a ≡ p) {
      cerr ≪ "ERROR!␣In␣Point::slope():\n" ≪ "Points␣are␣the␣same.␣Returning␣INVALID_\
          REAL\n\n" ≪ flush;
      if (DEBUG) {
        a.show("a");
        p.show("p");
      }
      return INVALID_REAL;
    }
    m = tolower(m);
    n = tolower(n);
    if (¬((m ≡ 'x' ∨ m ≡ 'y' ∨ m ≡ 'z') ∧ (n ≡ 'x' ∨ n ≡ 'y' ∨ n ≡ 'z') ∧ (m ≠ n))) {
      cerr ≪ "ERROR!␣In␣Point::slope():\n" ≪ "One␣or␣both␣char␣arguments␣are␣invalid␣\
          or␣they␣are␣the␣same:␣" ≪ m ≪ ",␣" ≪ n ≪ endl ≪ "Returning␣INVALID_REAL\n\n" ≪
          flush;
      return INVALID_REAL;
    }
    int ctr = m − 'x';
    real a_m_coord = a.world_coordinates[ctr];
    real p_m_coord = p.world_coordinates[ctr];
    ctr = n − 'x';
    real a_n_coord = a.world_coordinates[ctr];
    real p_n_coord = p.world_coordinates[ctr];
```

391. We often use *slope*( ) in order to find out whether a line has slope or not, so an error message is out of place here. A warning is too, probably, but I'm leaving this in here for now, just in case I change my mind.

⟨ Define **Point** functions 330 ⟩ +≡

```
  if (a_m_coord ≡ p_m_coord) {
#if 0
    cerr ≪ "WARNING!␣In␣Point::slope():\n" ≪ m ≪
        "␣coordinates␣of␣points␣are␣equal␣(no␣slope)!\n" ≪ "Returning␣INVALID_REAL\n\n";
#endif
    return INVALID_REAL;
  }
  return (a_n_coord − p_n_coord)/(a_m_coord − p_m_coord); }
```

392. Is on segment.

393. Non-const version. [LDF 2002.10.29.] *is_on_segment*( ) returns a **bool_real** with the **bool** indicating whether $*this$ lies on the line segment between $p0$ and $p1$, and a **real** value t representing the distance of $*this$ on the way from $p0$ to $p1$. If the **bool** is *true*, then $0 <= t <= 1$. If $t < 0$ or $t > 1$, then $*this$ lies on the line passing through $p0$ and $p1$, but not on the segment. If $*this$ doesn't lie on the line, t will be `INVALID_REAL`.

[LDF 2002.10.29.] To check whether $*this$ lies on the line, use *is_on_line*( ).

Log

[LDF 2002.10.29.] BUG FIX: Added code to check whether the unit vectors $*this - p0$ and $p1 - *this$ are equal before calculating r. Before I did this, *true* was returned for **Points** that weren't on the line segment.

[LDF 2002.10.29.] Now using *world_coordinates* directly instead of *get_x*( ), *get_y*( ), and *get_z*( ).

⟨ Declare **Point** functions 329 ⟩ +≡

bool_real *is_on_segment*(**Point** *p0*, **Point** *p1*);

394.

```
⟨ Define Point functions 330 ⟩ +≡
  bool_real Point::is_on_segment(Point p0, Point p1){ bool DEBUG = false;    /* true */
      if (DEBUG) cout ≪ "Entering␣Point::is_on_segment().\n" ≪ flush;
      apply_transform( );
      p0.apply_transform( );
      p1.apply_transform( );
      if (DEBUG) {
        show("this");
        p0.show("p0");
        p1.show("p1");
      }
      if (∗this ≡ INVALID_POINT ∨ p0 ≡ INVALID_POINT ∨ p1 ≡ INVALID_POINT) {
        cerr ≪ "ERROR!␣In␣Point::is_on_segment():\n" ≪ "One␣of␣the␣Points␣is␣invalid!␣" ≪
            "Returning␣false␣and␣INVALID_REAL.\n\n" ≪ flush;
        return pair⟨bool, real⟩(false, INVALID_REAL);
      }
      bool b;
      real r;
      if (p0 ≡ p1 ∧ ∗this ≡ p0) {
        cerr ≪ "ERROR!␣In␣Point::is_on_segment():\n" ≪
            "*this␣and␣the␣arguments␣p0␣and␣p1␣are␣all␣equal.␣" ≪
            "Returning␣false␣and␣INVALID_REAL.\n";
        return pair⟨bool, real⟩(false, INVALID_REAL);
      }
      else if (p0 ≡ p1) {
        cerr ≪ "ERROR!␣In␣Point::is_on_segment():\n" ≪ "Arguments␣p0␣and␣p1␣are␣equal.␣" ≪
            "Returning␣false␣and␣INVALID_REAL.\n";
        return pair⟨bool, real⟩(false, INVALID_REAL);
      }
      else if (∗this ≡ p0) {
        return pair⟨bool, real⟩(true, 0.0);
      }
      else if (∗this ≡ p1) {
        return pair⟨bool, real⟩(true, 1.0);
      }
            /* [LDF 2002.10.29.] Beginning of new code. */
      Point v0(∗this − p0);
      Point v1(p1 − ∗this);
      v0.apply_transform( );
      v1.apply_transform( );
      if (DEBUG) {
        v0.show("v0");
        v1.show("v1");
      }
      v0.unit_vector(true);
      v1.unit_vector(true);
      if (DEBUG) {
        v0.show("v0");
        v1.show("v1");
      }
```

```
Point v2(−v1);
if (DEBUG) {
  v2.show("v2");
}
if (v0 ≠ v1 ∧ v0 ≠ v2) {
  if (DEBUG) cout ≪ "Not␣on␣line.\n";
  return pair⟨bool, real⟩(false, INVALID_REAL);
}
      /* [LDF 2002.10.29.] End of new code. */
```

395. [LDF 2002.10.29.] Calculate how far ∗*this* is on the way from *p0* to *p1*.

LDF Undated. The value t can be calculated from either the x, y, or z-coordinates. We try them in order and return the first one that works. Because of the limited precision with which we are working, it's possible that the value of t can differ, depending on what coordinates are used to calculate it. In general, this will not be significant, since we'll mainly be needing this function to determine whether a **Point** is on a line segment; the exact value of t will usually not be significant.

⟨ Define **Point** functions 330 ⟩ +≡

```
if (p1.world_coordinates[0] ≠ p0.world_coordinates[0])
  r = (world_coordinates[0] − p0.world_coordinates[0])/(p1.world_coordinates[0] − p0.world_coordinates[0]);
else if (p1.world_coordinates[1] ≠ p0.world_coordinates[1])
  r = (world_coordinates[1] − p0.world_coordinates[1])/(p1.world_coordinates[1] − p0.world_coordinates[1]);
else if (p1.world_coordinates[2] ≠ p0.world_coordinates[2])
  r = (world_coordinates[2] − p0.world_coordinates[2])/(p1.world_coordinates[2] − p0.world_coordinates[2]);
else {
  cerr ≪ "ERROR!␣In␣Point::is_on_segment()\n" ≪
      "Can't␣calculate␣t.␣Returning␣false␣and␣INVALID_REAL.\n\n" ≪ flush;
  return pair⟨bool, real⟩(false, INVALID_REAL);
}
if (r ≥ 0 ∧ r ≤ 1) b = true;
else b = false;
return pair⟨bool, real⟩(b, r); }
```

396. const version.

Log

[LDF 2002.10.29.] Added this function.

⟨ Declare **Point** functions 329 ⟩ +≡

```
bool_real is_on_segment(const Point &p0, const Point &p1) const;
```

397.

⟨ Define **Point** functions 330 ⟩ +≡
  bool_real Point :: *is_on_segment*(**const Point** &*p0*, **const Point** &*p1*) **const**
  {
    Point *a*(∗*this*);
    return *a*.*is_on_segment*(*p0*, *p1*);
  }

398. Is on line. [LDF 2002.10.29.] TO DO: Maybe add a non-**const** version. This isn't urgent, though.

Log

[LDF 2002.10.29.] Added this function.

⟨ Declare **Point** functions 329 ⟩ +≡
  bool_real *is_on_line*(**const Point** &*p0*, **const Point** &*p1*) **const**;

399.

⟨ Define **Point** functions 330 ⟩ +≡
  bool_real Point :: *is_on_line*(**const Point** &*p0*, **const Point** &*p1*) **const**
  {
    bool_real *br* = *is_on_segment*(*p0*, *p1*);
    if (*br*.*second* $\neq$ `INVALID_REAL`) *br*.*first* = *true*;
    return *br*;
  }

400. Is on Plane. [LDF 2003.06.04.] This function returns *true*, if ∗*this* lies on the **Plane** *p*, otherwise *false*. It must be defined in `planes.web`, because **Plane** is an incomplete type here.

Log

[LDF 2003.06.04.] Added this function.

⟨ Declare **Point** functions 329 ⟩ +≡
  bool *is_on_plane*(**const Plane** &*p*) **const**;

401. Is in triangle. [LDF 2003.06.11.] This function returns *true*, if ∗*this* lies within the triangle defined by the three **Point** arguments, otherwise *false*. Defined in `paths.web`, because it uses **class Path**, which is an incompletely defined type here.

Log

[LDF 2003.06.11.] Added this function.
[LDF 2003.06.24.] Removed the argument *test_points*.

⟨ Declare **Point** functions 329 ⟩ +≡
  bool *is_in_triangle*(**const Point** &*p0*, **const Point** &*p1*, **const Point** &*p2*, **bool** *verbose* = *false*)
    const;

402. Transformations.

403. Affine transformations.

404. Rotation around the main axes.

Log

[LDF 2003.01.22.] Replaced body of function. **Transform** :: *rotate*( ) returns a **Transform** representing the rotation only, so I don't need to use a locally declared **Transform** *t* in this function.

⟨ Declare **Point** functions 329 ⟩ +≡
Transform *rotate*(**const real** x, **const real** $y = 0$, **const real** $z = 0$);

405.

⟨ Define **Point** functions 330 ⟩ +≡
Transform Point :: *rotate*(**const real** x, **const real** y, **const real** z)
{
return *transform.rotate*(x, y, z);
}

406. Scale.

Log

[LDF 2003.01.22.] Replaced body of function. **Transform** :: *scale*( ) returns a **Transform** representing the rotation only, so I don't need to use a locally declared **Transform** *t* in this function.

⟨ Declare **Point** functions 329 ⟩ +≡
Transform *scale*(**real** x, **real** $y = 1$, **real** $z = 1$);

407.

⟨ Define **Point** functions 330 ⟩ +≡
Transform Point :: *scale*(**real** x, **real** y, **real** z)
{
return *transform.scale*(x, y, z);
}

408. Shear.

Log

[LDF 2003.01.22.] Replaced body of function. **Transform** :: *shear*( ) returns a **Transform** representing the rotation only, so I don't need to use a locally declared **Transform** *t* in this function.

⟨ Declare **Point** functions 329 ⟩ +≡
Transform *shear*(**real** xy, **real** $xz = 0$, **real** $yx = 0$, **real** $yz = 0$, **real** $zx = 0$, **real** $zy = 0$);

409.

⟨ Define **Point** functions 330 ⟩ +≡
Transform Point :: *shear*(**real** xy, **real** xz, **real** yx, **real** yz, **real** zx, **real** zy)
{
return *transform.shear*(xy, xz, yx, yz, zx, zy);
}

410. Shift.

411. Point versions.

412. Three real arguments.

⟨ Declare **Point** functions 329 ⟩ +≡
Transform *shift*(**real** x, **real** $y = 0$, **real** $z = 0$);

413.

⟨ Define **Point** functions 330 ⟩ +≡
Transform Point :: *shift*(**real** x, **real** y, **real** z)
{
Transform t;
if $(x \neq 0 \lor y \neq 0 \lor z \neq 0)$ *transform* $*=$ $t.shift(x, y, z)$;
return t;
}

414. Point argument.

⟨ Declare **Point** functions 329 ⟩ +≡
Transform *shift*(**const Point** &p);

415.

⟨ Define **Point** functions 330 ⟩ +≡
Transform Point :: *shift*(**const Point** &p)
{
return *shift*(p.*get_x*( ), p.*get_y*( ), p.*get_z*( ));
}

416. Transform version. Point argument. [LDF 2002.04.24.] Added this function. It's declared in `transfor.web`, but must be defined here, because **Point** is an incomplete type there.

⟨ Define **Transform** functions 169 ⟩ +≡
Transform Transform :: *shift*(**const Point** &p)
{
return *shift*(p.*get_x*( ), p.*get_y*( ), p.*get_z*( ));
}

417. Picture version. Point argument. [LDF 2002.08.08.] Added this function. It's declared in `pictures.web`, but must be defined here, because **Point** is an incomplete type there.

⟨ Define **Picture** functions 264 ⟩ +≡
Transform Picture :: *shift*(**const Point** &p)
{
return *shift*(p.*get_x*( ), p.*get_y*( ), p.*get_z*( ));
}

418. Shift times.

[LDF 2003.01.19.] Note that *shift_times*( ) will only have an effect if it's called *after* a call to *shift*( ) and *before* an operation is applied that causes *apply_transform*( ) to be called.

Log

[LDF 2003.01.19.] Added this section.

419. Three real arguments.

Log

[LDF 2003.01.19.] Added this function.

[LDF 2003.01.22.] Got rid of local **Transform** *t*. It wasn't needed. Now just returning the return value of *transform.shift_times*( ).

⟨ Declare **Point** functions 329 ⟩ +≡
Transform *shift_times*(**real** x, **real** $y = 1$, **real** $z = 1$);

420.

⟨ Define **Point** functions 330 ⟩ +≡
Transform Point :: *shift_times*(**real** x, **real** y, **real** z)
{
  return *transform.shift_times*(x, y, z);
}

421. Point argument.

Log

[LDF 2003.01.19.] Added this function.

⟨ Declare **Point** functions 329 ⟩ +≡
Transform *shift_times*(**const Point** &*p*);

422.

⟨ Define **Point** functions 330 ⟩ +≡
Transform Point :: *shift_times*(**const Point** &*p*)
{
  return *transform.shift_times*(*p.get_x*( ), *p.get_y*( ), *p.get_z*( ));
}

423. Alignment with an axis. Declared in `transfor.web`. Defined here, because it needs **Points**. [LDF 2002.10.23.] *align_with_axis*( ) returns the **Transform** needed to align $\overrightarrow{p_0 p_1}$ with one of the main axes.

[LDF 2003.05.04.] BUG: TO DO: Try to find out why I sometimes get erroneous results with *rotate*(**Point**, **Point**, **real**) (formerly *rotate_around*( )). I think the problem may be here.

Log

[LDF 2002.12.10.] Made this function a **friend** in **class Point**. Now calling *p0.apply_transform*( ) and *p1.apply_transform*( ) at the beginning of this function and using *p0.world_coordinates* and *p1.world_coordinates* directly instead of *get_x*( ), *get_y*( ), and *get_z*( ).

424.

⟨ Define **Transform** functions 169 ⟩ +≡

```
Transform Transform::align_with_axis(Point p0, Point p1, char axis)
      /* Default is the z-axis. */
    { bool DEBUG = false;    /* true */
    if (DEBUG) cout << "Entering␣Transform::align_with_axis." << endl << flush;
    p0.apply_transform();    /* LDF 2002.12.10. Added these two lines. */
    p1.apply_transform();
    Transform t;
    axis = tolower(axis);    /* Upper- or lowercase is permitted for axis. */
    if (axis ≠ 'x' ∧ axis ≠ 'y' ∧ axis ≠ 'z') {
      cerr << "ERROR!␣In␣Transform::align_with_axis().␣Invalid␣\"axis\"␣argument:␣" <<
          axis << endl << "Returning␣identity␣matrix." << endl << flush;
      return t;
    }
    real angle;
```

425. [LDF 2002.10.23.] Shift *p0* to origin, and shift *p1* the same way, so that the relationship between them remains constant.

⟨ Define **Transform** functions 169 ⟩ +≡

```
  if (p0 ≠ origin) {
    if (DEBUG) {
      p0.show("p0");
      p1.show("p1");
    }
    t.shift(−p0);
    p1 *= t;
    p1.apply_transform();
    p0.apply_transform();
  }
  ⟨ Normalize point 433 ⟩    /* [LDF 2002.10.23.] Transform the Point so that it's x, y, and z coordinates
          are all positive. See below for the explanation. */
    if (DEBUG) {
      t.show("t␣outside␣of␣␣normalization");
    }
    Point proj_on_x_z_plane(p1);    /* [LDF 2002.10.23.] Get the projection of p1 on the x-z plane. */
  proj_on_x_z_plane.shift(0, −p1.world_coordinates[1]);
  if (DEBUG) proj_on_x_z_plane.show("proj_on_x_z_plane");
```

426. [LDF 2002.10.23.] If we're aligning with the x or y-axis, rotate *p1* onto the x-y plane and then to the x-axis.

⟨ Define **Transform** functions 169 ⟩ +≡
```
if (axis ≡ 'x' ∨ axis ≡ 'y') {
  Point pt_on_x_axis;
  pt_on_x_axis.set(1);
  angle = proj_on_x_z_plane.angle(pt_on_x_axis);
  if (DEBUG) cout ≪ "angle␣of␣projection:␣" ≪ angle ≪ endl ≪ flush;
  if (angle ≠ 0 ∧ angle ≠ INVALID_REAL) t *= p1.rotate(0, −angle);
  if (DEBUG) p1.show("p1␣after␣rotation␣to␣x-y␣plane");
  angle = p1.angle(pt_on_x_axis);
  if (DEBUG) cout ≪ "angle␣to␣x-axis:␣" ≪ angle ≪ endl ≪ flush;
  if (angle ≠ 0 ∧ angle ≠ INVALID_REAL) t *= p1.rotate(0, 0, −angle);
  if (DEBUG) p1.show("p1␣after␣rotation␣to␣x-axis");
}
```

427. [LDF 2002.10.23.] If we're aligning with the z-axis, rotate *p1* onto the y-z plane and then to the z-axis.

⟨ Define **Transform** functions 169 ⟩ +≡
```
else if (axis ≡ 'z') { Point pt_on_z_axis;
pt_on_z_axis.set(0, 0, 1);    /* [LDF 2002.10.23.] This assumes that proj_on_x_z_plane.get_z() ≥ 0. It
    should be, but if it isn't, the following error handling code takes care of the problem. */
if (proj_on_x_z_plane.get_z() < 0) {
  cerr ≪ "ERROR!␣In␣Transform::align_with_axis():\n" ≪
      "proj_on_x_z_plane.get_z()␣<␣0\n" ≪ "@<Normalize␣point@>␣should␣ensure␣that␣" ≪
      "this␣value␣is␣>=␣0.\n" ≪ "Handling␣the␣error,␣but␣find␣out␣why␣it␣happened!" ≪
      endl ≪ endl ≪ flush;
  pt_on_z_axis.set(0, 0, −1);
}
```

428.

Log

[LDF 2002.12.10.] Added the following conditional. Trying to fix a bug that occurred while porting to GNU/Linux.

⟨ Define **Transform** functions 169 ⟩ +≡
```
if (proj_on_x_z_plane.world_coordinates[0] ≡ 0 ∧ proj_on_x_z_plane.world_coordinates[1] ≡ 0) angle = 0;
else angle = proj_on_x_z_plane.angle(pt_on_z_axis);
```

429.

⟨ Define **Transform** functions 169 ⟩ +≡
```
if (DEBUG) cout ≪ "angle␣of␣projection:␣" ≪ angle ≪ endl ≪ flush;
if (angle ≠ 0 ∧ angle ≠ INVALID_REAL) t *= p1.rotate(0, angle);
if (DEBUG) p1.show("p1␣after␣rotation␣to␣z-y␣plane");
p1.apply_transform();
```

430.

Log

[LDF 2002.12.10.] Added the following conditional. Trying to fix a bug that occurred while porting to GNU/Linux.

[LDF 2003.06.13.] BUG FIX: Changed *proj_on_x_z_plane* to *p1* in the "**if**" part of the following conditional. The y-coordinate of *proj_on_x_z_plane* is always 0, so *angle* was always set to 0. I discovered this bug when I tried rotating a **Point** in the plane of a **Reg_Polygon** about a line from the center of the **Reg_Polygon** in the direction of its normal, and the resulting **Point** was not in the same plane.

```
⟨ Define Transform functions 169 ⟩ +≡
  if (p1.world_coordinates[1] ≡ 0) angle = 0;
  else angle = p1.angle(pt_on_z_axis);
```

431.

```
⟨ Define Transform functions 169 ⟩ +≡
  if (DEBUG) cout ≪ "angle␣to␣z-axis:␣" ≪ angle ≪ endl ≪ flush;
  if (angle ≠ 0 ∧ angle ≠ INVALID_REAL) t *= p1.rotate(−angle);
  if (DEBUG) p1.show("p1␣after␣rotation␣to␣z-axis");
  }
```

432. [LDF 2002.10.23.] If we're aligning with the y-axis, *p1* must be rotated from the x-axis (where it is now) around the z-axis by 90°. Then it will be on the y-axis.

```
⟨ Define Transform functions 169 ⟩ +≡
  if (axis ≡ 'y') {
    t *= p1.rotate(0, 0, 90);
    if (DEBUG) p1.show("p1␣after␣rotation␣to␣y-axis");
  }
  if (DEBUG) {
    cout ≪ "p1.magnitude()␣==␣" ≪ p1.magnitude() ≪ endl ≪ flush;
    t.show("t␣at␣end␣of␣align_with_axis()");
  }
  *this *= t;
  if (DEBUG) cout ≪ "Exiting␣Transform::align_with_axis." ≪ endl ≪ flush;
  return t; }
```

433. Normalize point. It makes it easier to determine the correct direction of rotation toward the x-y or y-z plane if *p1*'s coordinates are all >= 0, so we rotate it in order to make them so. The only case that requires more than a rotation around a single axis is the case that x_{p1}, y_{p1}, and z_{p1} are all < 0. It would be nice if I could replace this long conditional with a more elegant construction, but I don't know one.

```
⟨ Normalize point 433 ⟩ ≡
  {
    if (DEBUG) p1.show("p1 before normalization");
    if (p1.world_coordinates[0] < 0 ∧ p1.world_coordinates[1] ≥ 0 ∧ p1.world_coordinates[2] ≥ 0)
        /* x negative, y and z positive. */
      t *= p1.rotate(0, −90);
    else if (p1.world_coordinates[0] ≥ 0 ∧ p1.world_coordinates[1] < 0 ∧ p1.world_coordinates[2] ≥ 0)
        /* x positive, y negative, z positive. */
      t *= p1.rotate(90);
    else if (p1.world_coordinates[0] ≥ 0 ∧ p1.world_coordinates[1] ≥ 0 ∧ p1.world_coordinates[2] < 0)
        /* x positive, y positive, z negative. */
      t *= p1.rotate(−90);
    else if (p1.world_coordinates[0] < 0 ∧ p1.world_coordinates[1] < 0 ∧ p1.world_coordinates[2] ≥ 0)
        /* x negative, y negative, z positive. */
      t *= p1.rotate(0, 0, 180);
    else if (p1.world_coordinates[0] < 0 ∧ p1.world_coordinates[1] ≥ 0 ∧ p1.world_coordinates[2] < 0)
        /* x negative, y positive, z negative. */
      t *= p1.rotate(0, 180);
    else if (p1.world_coordinates[0] ≥ 0 ∧ p1.world_coordinates[1] < 0 ∧ p1.world_coordinates[2] < 0)
        /* x positive, y negative, z negative. */
      t *= p1.rotate(180);
    else if (p1.world_coordinates[0] < 0 ∧ p1.world_coordinates[1] < 0 ∧ p1.world_coordinates[2] < 0)
      /* All negative. */
    {
      real a = p1.world_coordinates[0];
      t *= p1.rotate(180, 180);
      t *= p1.shift(a);
      t *= p1.rotate(0, 180);
      t *= p1.shift(−a);
    }
    p1.apply_transform();
    if (DEBUG) {
      p0.show("p0 after normalization");
      p1.show("p1 after normalization");
      t.show("t after normalization");
    }
  }
```

This code is used in section 425.

434. Rotation around an arbitrary axis.

435. Point versions. [LDF 2002.4.7.] Added default value for *angle* ≡ 180.

436. Point arguments. This function first checks to see if $*this$ lies on the axis. It does this by creating unit vectors in the directions of $p1 - p0$ and $*this - p0$. If they are equal, or the latter is the former multiplied by -1, then we don't bother to perform the rotation. Otherwise, we call **Transform** :: *rotate*( ) (defined below).

Log

[LDF 2002.4.7.] Added default value for $angle \equiv 180$.
[LDF 2003.05.02.] Changed name of this function from *rotate_around*( ) to *rotate*( ). This function now overloads *rotate*( ) with three **real** arguments.

⟨ Declare **Point** functions 329 ⟩ +≡
```
Transform rotate(const Point &p0, const Point &p1, const real angle = 180);
```

437.

⟨ Define **Point** functions 330 ⟩ +≡
```
Transform Point::rotate(const Point &p0, const Point &p1, const real angle)
{
  Point a = p1 - p0;
  Point b = *this - p0;
  a.unit_vector(true);
  b.unit_vector(true);
  Transform t;
  if (a ≡ b ∨ a ≡ -b) {
    cerr ≪ "WARNING!␣In␣Point::rotate().\n" ≪ "Point␣to␣be␣rotated␣lies␣on␣axis.\n" ≪
        "Returning␣identity␣Transform.\n\n" ≪ flush;
    return t;
  }
  return transform.rotate(p0, p1, angle);
}
```

438. Path argument. Defined in `paths.web`, because **Path** is still an incomplete type in this compilation unit.

Log

[LDF 2002.04.07.] Added default value for $angle \equiv 180$.
[LDF 2003.05.02.] Changed name of this function from *rotate_around*( ) to *rotate*( ). This function now overloads *rotate*( ) with three **real** arguments.

⟨ Declare **Point** functions 329 ⟩ +≡
```
Transform rotate(const Path &p, const real angle = 180);
```

439. Transform version. Declared in `transfor.web`. [LDF 2002.09.29.] TO DO: Possible BUG!! Actually, the problem that occurred may just have to do choosing the direction of rotation. I've changed the place where the problem occurred, so I'll have to write a routine to test this.

Log

[LDF 2002.10.23.] Changed, so that the direction of $\overrightarrow{p_0 p_1}$ is tested. If it is parallel to the x or y-axis, then that axis is used for alignment. Otherwise, the z-axis is used. This may help reduce inaccuracies caused by rotations. Haven't tested it yet. TO DO: Test this!

[LDF 2002.11.03.] TO DO: See if I can't make **Point** arguments **const**.

[LDF 2003.05.02.] Changed name of this function from *rotate_around*( ) to *rotate*( ). This function now overloads *rotate*( ) with three **real** arguments.

⟨ Define **Transform** functions 169 ⟩ +≡

```
Transform Transform::rotate(Point p0, Point p1, const real angle)
{
    bool DEBUG = false;    /* true */
    if (DEBUG) cout ≪ "Entering␣Transform::rotate().\n";
    p0.apply_transform();
    p1.apply_transform();
    Point a = p1 − p0;
    a.unit_vector(true);
    char axis;
    if (a.get_x() ≡ 1 ∨ a.get_x() ≡ −1) axis = 'x';
    else if (a.get_y() ≡ 1 ∨ a.get_y() ≡ −1) axis = 'y';
    else axis = 'z';
    Transform t;
    t.align_with_axis(p0, p1, axis);
    Transform i = t.inverse();
    if (axis ≡ 'x') t.rotate(angle);
    else if (axis ≡ 'y') t.rotate(0, angle);
    else t.rotate(0, 0, angle);
    t *= i;
    t.clean();
    *this *= t;
    clean();
    return t;
}
```

440. Picture version. [LDF 2002.10.20.] *angle* is in degrees.

Log

[LDF 2003.05.02.] Changed name of this function from *rotate_around*( ) to *rotate*( ). This function now overloads *rotate*( ) with three **real** arguments.

⟨ Define **Picture** functions 264 ⟩ +≡

```
Transform Picture::rotate(const Point &p0, const Point &p1, const real angle)
{
  Transform t;
  t.rotate(p0, p1, angle);
  transform *= t;
  return t;
}
```

441. Projection. [LDF 2002.10.23.] For stylistic reasons, and for the sake of clean programming, I believe that the programmer who uses *project*( ) should ensure that *apply_transform*( ) has been invoked first. However, *transform* is checked in *project*( ) and *apply_transform*( ) is invoked, if required, so invoking *apply_transform*( ) explicitly beforehand is not strictly speaking necessary.

Log

[LDF 2002.09.09.] The new version now almost works Added division of *projective_coordinates* by the value calculated for w. However, it doesn't work when I use *hex_pattern1*( ). Find out why not!! TO DO: Add routine for calculating z. Then I can add sorting routine in **Picture**::*output*( ).

[LDF 2002.09.14.] I believe I've gotten the new version to work now. LOOK UP: Do I need to divide the derived z value by w? I don't think it's necessary. Since the z values of all of the **Points** would be divided by the same amount, their relative positions would remain the same, since only the relationship "closer or further away" matters, not the exact amounts.

[LDF 2002.09.16.] Added **Focus** argument to this function. Default is *default_focus*, but it was necessary to write a dummy version of this function in order to make this work, because *default_focus* doesn't exist at the time that this declaration is compiled.

[LDF 2002.09.18.] Changed name of this function from *persp_transform*( ) to *project* and added **Transform** argument.

[LDF 2003.05.09.] BUG FIX: Added loop, setting all elements of *projective_coordinates* to 0. This was done in the conditionally compiled code for the DEC compiler, but I forgot to do it for GCC when added the declaration of *temp_coordinates* and resized it. It took me about 6–7 hours to find this bug!

---

442. Focus argument.

Log

[LDF 2003.07.11.] Added defaults for *proj* and *factor*.

⟨ Declare **Point** functions 329 ⟩ +≡

```
bool project(const Focus &f, const unsigned short proj = Projections::PERSP, real factor = 1);
```

443.

⟨ Define **Point** functions 330 ⟩ +≡

```
bool Point::project(const Focus &f, const unsigned short proj, real factor){ bool DEBUG = false;
      /* true */
    if (DEBUG) cout << "Entering project().\n" << flush;
    if (¬transform.is_identity())      /* LDF 2002.10.23. Added, just to be sure. */
      apply_transform();
#ifdef __GNUC__
    valarray⟨real⟩ temp_coordinates;
    temp_coordinates.resize(4, 0);
    for (int i = 0; i < 4; ++i)      /* LDF 2003.05.09. Added this loop. */
      projective_coordinates[i] = 0;
#else
#ifdef __DECCXX
    valarray⟨real⟩ temp_coordinates = projective_coordinates = null_coordinates;
#endif
#endif
    int i;
    int j;      /* [LDF 2002.09.18.] Transform temp_coordinates by Focus::transform. */
    for (i = 0; i < 4; i++) {
      for (j = 0; j < 4; j++) {
        temp_coordinates[i] += world_coordinates[j] * f.get_transform_element(j, i);
      }
    }
```

444. Parallel projection.
[LDF 2002.11.06.] TO DO: Add a way of projecting onto a plane other than the x-y plane. It's possible to acheive the same effect by rotating the **Picture** before outputting it, but it would be nice to do so without changing the **Picture**.

The x and y projective coordinates are simply taken from the world coordinates.

Log

[LDF 2002.10.23.] Added this section.
[LDF 2002.12.18.] Changed `PARALLEL` to `PARALLEL_X_Y` and added `PARALLEL_X_Z` and `PARALLEL_Z_Y`.

```
⟨ Define Point functions 330 ⟩ +≡
  if (proj ≡ Projections::PARALLEL_X_Y ∨ proj ≡ Projections::PARALLEL_X_Z ∨ proj ≡
        Projections::PARALLEL_Z_Y) {
    using namespace Projections;
    if (factor ≡ 0) {
      cerr ≪ "ERROR!␣In␣Point::project():\n" ≪ "factor␣==␣0.␣␣Multiplying␣coordinates␣b\
          y␣0␣doesn't␣make␣sense.␣" ≪ "Using␣1␣instead.\n\n" ≪ flush;
      factor = 1;
    }
    unsigned short horizontal;
    unsigned short vertical;
    if (proj ≡ PARALLEL_X_Y ∨ proj ≡ PARALLEL_X_Z)    /* [LDF 2002.12.18.] Explain this!! */
    horizontal = 0;
    else horizontal = 2;
    if (proj ≡ PARALLEL_X_Y ∨ proj ≡ PARALLEL_Z_Y) vertical = 1;
    else vertical = 2;
    projective_coordinates[0] = world_coordinates[horizontal] * factor;
    projective_coordinates[1] = world_coordinates[vertical] * factor;
    projective_coordinates[2] = 0;
    projective_coordinates[3] = 1;
    if (DEBUG) {
      cout ≪ "projective_coordinates:␣(" ≪ projective_coordinates[0] ≪ ",␣" ≪
          projective_coordinates[1] ≪ ",␣" ≪ projective_coordinates[2] ≪ ",␣" ≪
          projective_coordinates[3] ≪ ")" ≪ endl ≪ endl ≪ flush;
    }
    return true;
  }
```

445. Perspective projection. !! KLUDGE: See below. [LDF 2002.11.08.] TO DO: Get numbers to output using *only* decimal notation!

⟨ Define **Point** functions 330 ⟩ +≡

```
if (temp_coordinates[2] + f.get_distance( ) ≡ 0) {
  cerr ≪ "ERROR!␣In␣Point::project():\n" ≪ "temp_coordinates[2]␣==␣" ≪ temp_coordinates[2] ≪
      ",␣f.distance␣==␣" ≪ f.get_distance( ) ≪ endl ≪ "Sum␣==␣0.␣Can't␣perform␣division." ≪
      endl ≪ "Setting␣projective_coordinates␣to␣INVALID_REAL" ≪ "␣and␣returning.\n" ≪
      flush;
  for (i = 0; i < 4; i++) projective_coordinates[i] = INVALID_REAL;
  return false;
}
else if (temp_coordinates[2] + f.get_distance( ) < 0) {
  cerr ≪ "ERROR!␣In␣Point::project():\n" ≪ "temp_coordinates[2]␣==␣" ≪
      temp_coordinates[2] ≪ ",␣f.distance␣==␣" ≪ f.get_distance( ) ≪
      endl ≪ "Their␣sum␣<␣0.␣Point␣lies␣behind␣focus." ≪ endl ≪
      "Setting␣projective_coordinates␣to␣INVALID_REAL" ≪ "␣and␣returning.\n" ≪ flush;
  for (i = 0; i < 4; i++) projective_coordinates[i] = INVALID_REAL;
  return false;
}
real save_z = temp_coordinates[2]/(temp_coordinates[2] + f.get_distance( ));
if (DEBUG) {
  cout ≪ "temp_coordinates[2]␣==␣" ≪ temp_coordinates[2] ≪ endl ≪ flush;
  cout ≪ "f.get_distance()␣==␣" ≪ f.get_distance( ) ≪ endl ≪ flush;
  cout ≪ "save_z␣==␣" ≪ save_z ≪ endl ≪ flush;
}
for (i = 0; i < 4; i++) {
  for (j = 0; j < 4; j++) {
    projective_coordinates[i] += temp_coordinates[j] ∗ f.get_persp_element(j, i);
  }
}
real eps = epsilon( );
if (projective_coordinates[3] ≡ 0) {
  cerr ≪ "ERROR!␣In␣Point::project():␣" ≪ "projective_coordinates[3]␣==␣0.\n" ≪
      "This␣will␣cause␣a␣floating␣point␣error.\n" ≪
      "Setting␣projective_coordinates␣to␣INVALID_REAL" ≪ "␣and␣returning.\n" ≪ flush;
  for (i = 0; i < 4; i++) projective_coordinates[i] = INVALID_REAL;
  return false;
}
for (i = 0; i < 4; i++) {
  projective_coordinates[i] /= projective_coordinates[3];
  if (fabs(projective_coordinates[i]) < eps) projective_coordinates[i] = 0;
}   /∗ [LDF 2002.09.14.] Set the z value of the perspective coordinates in order to be able to use it for
      my hidden surface algorith. ∗/
projective_coordinates[2] = (fabs(save_z) > eps) ? save_z : 0;
if (DEBUG) {
  cout ≪ "Perspective␣coordinates:\n(";
  for (i = 0; i < 4; i++) {
    cout ≪ projective_coordinates[i];
    if (i < 3) cout ≪ ",␣";
  }
  cout ≪ ")\n" ≪ flush;
```

```
}
for (i = 0; i < 2; i++)
     /* [LDF 2002.11.07.] KLUDGE. Added this loop. The value used for comparison is slightly larger
        than one that arose while I was testing the constructor of Trunc_Octahedron. eps was too small.
        */
  if (fabs(projective_coordinates[i]) ≤ 10.0 · 10^−05) projective_coordinates[i] = 0;
if (DEBUG) cout ≪ "Exiting␣project().\n" ≪ flush;
return true; }
```

446. No Focus argument. [LDF 2002.09.13.] Added this function. This dummy function just passes *default_focus* to *project* (**const Focus** &*f* ...). This is necessary because it's impossible make the argument *f* optional with *default_focus* as the default. This is because *project*( ) must be declared inside the **class** declaration of **Point**, whereas the declaration of **Focus** must be later, because it contains **Points**. It's not a problem to use *default_focus* inside of *project*( ), as long as *default_focus* is defined before the function is called.

⟨ Declare **Point** functions 329 ⟩ +≡

```
bool project(const unsigned short &proj = Projections::PERSP, real factor = 1);
```

447.

⟨ Define **Point** functions 330 ⟩ +≡

```
bool Point::project(const unsigned short &proj, real factor)
{
  return project(default_focus, proj, factor);
}
```

448. Applying transformations. This version applies the transformation stored in **Point** :: *transform*. !! Add a version that applies a **Transform** supplied as an argument!! [LDF 2002.12.08.] BUG FIX: See below.

⟨ Declare **Point** functions 329 ⟩ +≡

```
void apply_transform();
```

449.

⟨ Define **Point** functions 330 ⟩ +≡

```
void Point::apply_transform()
{
  bool DEBUG = false;   /* true */
  if (transform.is_identity())   /* If transform.matrix is the identity matrix, we don't need to bother
        to perform the matrix multiplication. */
    return;
  int i;
  int j;
  valarray<real> new_coordinates;
  new_coordinates.resize(4,0);   /* [LDF 2002.12.08.] BUG FIX. For GNU CC. */
  if (DEBUG) {
    cout << "x == " << world_coordinates[0] << endl << flush;
    cout << "y == " << world_coordinates[1] << endl << flush;
    cout << "z == " << world_coordinates[2] << endl << flush;
    cout << "w == " << world_coordinates[3] << endl << flush;
  }
  for (i = 0; i < 4; i++) {
    for (j = 0; j < 4; j++) {
      new_coordinates[i] += world_coordinates[j] * transform.matrix[j][i];
      if (new_coordinates[i] != 0) {
        if (DEBUG) {
          cout << "new_coordinates[" << i << "] == " << new_coordinates[i] << endl << flush;
        }
      }
    }
  }
  real eps = epsilon();
  for (i = 0; i < 4; i++) {
    if (DEBUG) {
      cout << "new_coordinates[" << i << "] == " << new_coordinates[i] << endl << flush;
    }
    world_coordinates[i] = (fabs(new_coordinates[i]) > eps) ? new_coordinates[i] : 0;
  }
  transform.reset();
}
```

450. Set transform to identity.

⟨ Declare **Point** functions 329 ⟩ +≡

```
void reset_transform();
```

451.

⟨ Define **Point** functions 330 ⟩ +≡
```
void Point::reset_transform()
{
  transform.reset();
}
```

452. Drawing.

453. Drawdot.

[LDF 2002.10.26.] *drawdot*( ) copies ∗*this* and puts the copy onto the **vector** *shapes* of the **Picture** argument *picture*. The data members *drawdot_value*, *drawdot_color*, and *pen* are only set on the copy, not on ∗*this*. All of the drawing and filling functions behave similarly.

[LDF 2003.05.30.] TO DO: Add code for allocating new **Color**, if *ddrawdot_color.use_name* is *false*, as in the drawing and filling functions for **Path** and **Solid**.

454. Normal version.

Log

[LDF 2003.07.11.] Made *ppen* and *drawdot*( ) itself **const**.

⟨ Declare **Point** functions 329 ⟩ +≡
```
void drawdot(const Color &ddrawdot_color = *Colors::default_color, const string ppen = "", Picture
    &picture = current_picture) const;
```

455.

⟨ Define **Point** functions 330 ⟩ +≡
```
void Point::drawdot(const Color &ddrawdot_color, const string ppen, Picture &picture) const{
        bool DEBUG = false;    /* true */
    if (DEBUG) cout << "Entering␣drawdot()" << "\n" << flush;
    Point *pt = create_new < Point > (0);
    *pt = *this;
    pt->drawdot_value = DRAWDOT;
    pt->drawdot_color = &ddrawdot_color;
#if 1
    pt->pen = ppen;
#endif
    picture += static_cast<Shape *>(pt);
    if (DEBUG) cout << "Exiting␣drawdot()" << "\n" << flush;
    }
```

456. Picture argument first.

Log

[LDF 2002.01.24.] Added this version.
[LDF 2003.01.31.] Removed default for *picture*. Having a default made calls to *drawdot*( ) with no arguments ambiguous.
[LDF 2003.07.11.] Made *ppen* and *drawdot*( ) itself **const**.

⟨ Declare **Point** functions 329 ⟩ +≡
```
void drawdot(Picture &picture, const Color &ddrawdot_color = *Colors::default_color, const string
    ppen = "") const;
```

457.

⟨ Define **Point** functions 330 ⟩ +≡
```
void Point::drawdot(Picture &picture, const Color &ddrawdot_color, const string ppen) const
{
  drawdot(ddrawdot_color, ppen, picture);
}
```

458. Undrawdot. [LDF 2002.10.26.] *undraw*( ) does not remove a dot from *picture*, but causes the METAPOST command **undrawdot** to be written to *out_stream* when *picture* is output.

⟨ Declare **Point** functions 329 ⟩ +≡
```
void undrawdot(string ppen = "", Picture &picture = current_picture);
```

459.

⟨ Define **Point** functions 330 ⟩ +≡
```
void Point::undrawdot(string ppen, Picture &picture){ Point *pt = create_new < Point > (0);
    *pt = *this;
    pt->drawdot_value = UNDRAWDOT;
    pt->drawdot_color = Colors::background_color;
#if 1
    pt->pen = ppen;
#endif
    picture += static_cast<Shape *>(pt); }
```

460. Picture argument first.

Log

[LDF 2002.01.24.] Added this version.
[LDF 2003.01.31.] Removed default for *picture*. Having a default made calls to *undrawdot*( ) with no arguments ambiguous.

⟨ Declare **Point** functions 329 ⟩ +≡
```
void undrawdot(Picture &picture, string ppen = "");
```

461.

⟨ Define **Point** functions 330 ⟩ +≡
void Point :: *undrawdot*(**Picture** &*picture*, **string** *ppen*)
{
undrawdot(*ppen*, *picture*);
}

462. Draw. [LDF 2002.10.26.] *draw*( ) creates a **Path** with the two **Points** ∗*this* and the argument *p*, and the connector `"--"`, calls **Path** :: *draw*( ) for it, and returns the **Path**. The latter is a line, i.e., **Path** :: *get_line_switch*( ) returns *true* for it.

draw( ) must be defined in `paths.web`, because **Path** is an incomplete type here.

463. Normal version.

Log

[LDF 2003.01.15.] Added the argument *aarrow*.

⟨ Declare **Point** functions 329 ⟩ +≡
Path *draw*(**const Point** &*p*, **const Color** &*ddraw_color* = ∗**Colors** :: *default_color*, **string** *ddashed* = `""`, **string** *ppen* = `""`, **Picture** &*picture* = *current_picture*, **bool** *aarrow* = *false*) **const**;

464. Picture argument first. [LDF 2003.01.15.] This function is convenient for when I want to pass a **Picture** argument.

Log

[LDF 2002.09.17.] Added this function.
[LDF 2003.01.15.] Added the argument *aarrow*.

⟨ Declare **Point** functions 329 ⟩ +≡
Path *draw*(**Picture** &*picture*, **const Point** &*p*, **const Color** &*ddraw_color* = ∗**Colors** :: *default_color*, **string** *ddashed* = `""`, **string** *ppen* = `""`, **bool** *aarrow* = *false*);

465. Draw arrow.

466. Normal version. [LDF 2003.01.15.] Defined in `paths.web`.

Log

[LDF 2003.01.15.] Added this function.
[LDF 2003.06.03.] Made *drawarrow*( ) **const**.

⟨ Declare **Point** functions 329 ⟩ +≡
Path *drawarrow*(**const Point** &*p*, **const Color** &*ddraw_color* = ∗**Colors** :: *default_color*, **string** *ddashed* = `""`, **string** *ppen* = `""`, **Picture** &*picture* = *current_picture*) **const**;

467. Picture argument first. [LDF 2003.01.15.] Defined in `paths.web`.

Log

[LDF 2003.06.03.] Made *drawarrow*( ) **const**.

⟨ Declare **Point** functions 329 ⟩ +≡
Path *drawarrow*(**Picture** &*picture*, **const Point** &*p*,
const Color &*ddraw_color* = ∗**Colors** :: *default_color*, **string** *ddashed* = "", **string** *ppen* = "")
const;

468. Undraw.

469. Normal version. This function must be defined in `paths.web`, because it uses **Path**, which is an incomplete type here.

Log

[LDF 2002.4.8.] Added this function.
[LDF 2002.11.03.] Changed this function, so that it returns the **Path** *pa*, instead of **void**.

⟨ Declare **Point** functions 329 ⟩ +≡
Path *undraw*(**const Point** &*pt*, **string** *ddashed* = "", **string** *ppen* = "", **Picture**
&*picture* = *current_picture*);

470. Picture argument first.

Log

[LDF 2002.09.17.] Added this function. It's convenient for when I want to pass a **Picture** argument.
[LDF 2002.11.03.] Changed this function, so that it returns the **Path** *pa*, instead of **void**.

⟨ Declare **Point** functions 329 ⟩ +≡
Path *undraw*(**Picture** &*picture*, **const Point** &*pt*, **string** *ddashed* = "", **string** *ppen* = "");

471. Draw help. [LDF 2002.10.26.] *draw_help*( ) is like *draw*( ), except that the **Path** is only drawn if the **static Path** data member *do_help_lines* ≡ *true*. This is convenient for drawing construction lines that shouldn't be output in the final version of a drawing. Also, the default color is ∗**Colors** :: *help_color*.

472. Normal version. [LDF 2002.4.8.] This function must be defined in `paths.web`, because it uses **Path**, which is an incomplete type here.

Log

[LDF 2002.4.8.] Added this function.
[LDF 2003.07.13.] Made this function **const**.

⟨ Declare **Point** functions 329 ⟩ +≡
Path *draw_help*(**const Point** &*pt*, **const Color** &*ddraw_color* = ∗**Colors** :: *help_color*, **string**
ddashed = "", **string** *ppen* = "", **Picture** &*picture* = *current_picture*) **const**;

473. Picture argument first. [LDF 2002.09.17.] This version is convenient for when I want to pass a **Picture** argument.

Log

[LDF 2002.09.17.] Added this function.
[LDF 2003.07.13.] Made this function **const**.

⟨ Declare **Point** functions 329 ⟩ +≡
Path *draw_help*(**Picture** &*picture*, **const Point** &*pt*,
  const Color &*ddraw_color* = ∗**Colors**::*help_color*, **string** *ddashed* = "", **string** *ppen* = "") **const**;

474. Showing.

475. Show. [LDF 2002.10.26.] The arguments:

string *text* — If *text* is non-empty, (i.e., not ""), it's written to standard output (`stdout`). If it is empty, or *show*( ) is called without any arguments, the default is used, namely `"Point:"`.

char *coords* — One of the characters `'w'`, `'p'`, `'u'`, or `'v'` should be used, to indicate which set of coordinates should be shown: *world_coordinates*, *projective_coordinates*, *user_coordinates*, or *view_coordinates*, respectively. The latter two exist, but are not currently used. The corresponding uppercase characters can also be used.

const bool *do_persp* — Only meaningful if the *projective_coordinates* are being shown (*coords* argument `'p'`). If *do_persp* ≡ *true*, then *project*( ) is called on ∗*this* before *projective_coordinates* are shown. This is usually what one wants. However, it may sometimes be useful to show the contents of *projective_coordinates*, without calling *project*( ), in which case *do_persp* should be *false*.

const bool *do_apply* — Usually, *apply_transform*( ) should be called on ∗*this* before showing a set of coordinates, so the default for *do_apply* is *true*. However, it may sometimes be useful to show the values of the coordinates without applying *transform*, in which case *do_apply* should be *false*.

Focus ∗*f* — Only meaningful if the *projective_coordinates* are being shown (*coords* argument `'p'`). Refers to the **Focus** used for projection. If the default is used, or 0 is passed as the argument explicitly, then the global variable *default_focus* is used.

const unsigned short *proj* — Only meaningful if the *projective_coordinates* are being shown (*coords* argument `'p'`). Refers to the projection used. Currently, I've only programmed the perspective and the parallel projections. The default is the perspective projection.

const real *factor* — Only meaningful if the *projective_coordinates* are being shown (*coords* argument `'p'`) and the parallel projection is being used. The x and y values in *projective_coordinates* are multiplied by *factor*, so it can be used to magnify or shrink the projected image. The default is 1 (no magnification or shrinking).

[LDF 2002.10.26.] TO DO: Add case `'a'` for *coords* for showing all of the sets of coordinates.

Log

[LDF 2002.10.26.] !! KLUDGE: In the text above, I've had to typeset "*projective_coordinates*" using "\it" explicitly in a couple of places, in order to get the hyphenation to work.

[LDF 2002.11.12.] Added "\relax" after the arguments to "\ARG" in the TeX code above in order to suppress a space at the beginning of the first line of the following indented paragraph. I couldn't figure out a way of suppressing the space within the definition of \ARG.

[LDF 2003.04.30.] Changed, so that a newline is not output following *text*.

⟨ Declare **Point** functions 329 ⟩ +≡

```
void show(string text = "", char coords = 'w', const bool do_persp = true, const bool
    do_apply = true, Focus *f = 0, const unsigned short proj = Projections::PERSP, const real
    factor = 1) const;
```

476.

⟨ Define **Point** functions 330 ⟩ +≡

```
void Point::show(string text, char coords, const bool do_persp, const bool do_apply, Focus
      *f, const unsigned short proj, const real factor) const
{
  bool DEBUG = false;   /* true */
  if (text ≡ "") text = "Point:";
  cout ≪ text ≪ "␣";
  coords = tolower(coords);
  if (coords ≡ 'w') ;   /* Do nothing. */
  else if (coords ≡ 'p') cout ≪ "Projective␣coordinates.\n" ≪ flush;
  else if (coords ≡ 'u') cout ≪ "User␣coordinates.\n" ≪ flush;
  else if (coords ≡ 'v') cout ≪ "View␣coordinates.\n" ≪ flush;
  else {
    cerr ≪ "WARNING!␣In␣|show()|:␣" ≪ "Invalid␣character␣for␣coords␣argument.\n" ≪
        "Showing␣world␣coordinates.\n" ≪ flush;
    coords = 'w';
  }
  if (*this ≡ INVALID_POINT) {
    cerr ≪ "Point␣is␣==␣INVALID_POINT.\nCan't␣show.␣Returning.\n" ≪ flush;
    return;
  }
  if (DEBUG) transform.show("Transform␣before␣apply_transform");
  valarray⟨real⟩ v = get_all_coords(coords, do_persp, do_apply, f, proj, factor);
  cout ≪ "(" ≪ v[0] ≪ ",␣" ≪ v[1] ≪ ",␣" ≪ v[2] ≪ ")\n" ≪ flush;
  if (DEBUG) {
    transform.show("Transform␣after␣apply_transform");
    cout ≪ "on_free_store␣==␣" ≪ on_free_store ≪ "\n";
  }
}
```

477. Show transform.

⟨ Declare **Point** functions 329 ⟩ +≡

```
void show_transform(string text = "");
```

478.

⟨ Define **Point** functions 330 ⟩ +≡

```
void Point::show_transform(string text)
{
  if (text ≡ "") text = "transform:";
  cout ≪ text ≪ endl;
  transform.show();
}
```

479. Outputting.

480. Output operator. [LDF 2002.10.26.] This function is used in **Path**::*output*( ) for writing the x and y values of the *projective_coordinates* to *out_stream*. All code using this function must ensure that *apply_transform*( ) and *project*( ) are called *first*!

Log

[LDF 2002.09.16.] Removed calls to *apply_transform*( ) and *project*( ).

⟨ Declare non-member non-template functions for **Point** 480 ⟩ ≡

```
ostream &operator≪(ostream &o, Point &p);
```

See also section 534.

This code is used in section 634.

481.

⟨ Define non-member non-template functions for **Point** 481 ⟩ ≡

```
ostream &operator≪(ostream &o, Point &p)
{
  o ≪ "(" ≪ p.get_x('p', false, false) ≪ Point::measurement_units ≪ ",␣" ≪ p.get_y('p', false,
      false) ≪ Point::measurement_units ≪ ")";
  return o;
}
```

See also section 535.

This code is used in section 633.

482. Suppress output. [LDF 2002.09.18.] Added this function. It's needed because trying to erase a **Shape** ∗ from *elements* in **Picture**::*output*( ) causes a memory fault.

⟨ Declare **Point** functions 329 ⟩ +≡

```
virtual void suppress_output();
```

483.

⟨ Define **Point** functions 330 ⟩ +≡

```
void Point::suppress_output()
{
  do_output = false;
}
```

484. Unsuppress output. [LDF 2002.09.18.] Added this function. It's needed because trying to erase a **Shape** ∗ from *elements* in **Picture**::*output*( ) causes a memory fault.

⟨ Declare **Point** functions 329 ⟩ +≡

```
virtual void unsuppress_output();
```

485.

⟨ Define **Point** functions 330 ⟩ +≡
  void Point :: *unsuppress_output*( )
  {
    do_output = *true*;
  }

486. Extract. [LDF 2002.10.26.] *extract*( ) is a pure **virtual** function in **Shape**. It's called by **Picture** :: *output*( ). Each of the **Shape** pointers on the **vector** *shapes* in the **Picture** must be "extracted". For **Points**, this means projecting the **Point** using the **Focus** passed to *extract*( ) as an argument. If *project*( ) succeeds, *extract*( ) returns a **vector** containing *this*. Otherwise, it returns an empty **vector**.

[LDF 2002.10.26.] A **vector** is returned rather than *this* by itself because it may sometimes be useful to return a collection of **Shape** pointers rather than a single one. This was formerly the case for **Cuboid**, but at the present time, no version of *extract*( ) returns a **vector** with more than one pointer to **Shape**.

Log

[LDF 2002.09.17.] Added **const Focus** &*f* argument and error handling code. Now, if the **Point** cannot be projected onto the projection plane using the **Focus** *f*, it is not put onto the **vector**⟨**Shape** ∗⟩ **Picture** :: *elements*, and consequently never reaches **Picture** :: *output*( ) and **Point** :: *output*( ).

⟨ Declare **Point** functions 329 ⟩ +≡
  vector⟨**Shape** ∗⟩ *extract*(**const Focus** &*f*, **const unsigned short** *proj*, **real** *factor*);

487.

⟨ Define **Point** functions 330 ⟩ +≡

```
vector⟨Shape *⟩ Point::extract(const Focus &f, const unsigned short proj, real factor)
{
  bool DEBUG = false;    /* true */
  vector⟨Shape *⟩ v;     /* [LDF 2002.09.16.] Added this error checking code. Check *this first, to
      make sure that it can be drawn with the current value of default_focus. */
  apply_transform();
  if (¬project(f, proj, factor)) {
    if (DEBUG)
      cerr ≪ "WARNING!␣In␣Path::extract():\n" ≪ "Point␣on␣Path␣cannot␣be␣projected.␣" ≪
          "Returning␣empty␣vector␣<Shape*>.\n" ≪ flush;
    return v;
  }
  if (DEBUG) {
    cout ≪ "world_coordinates:␣␣␣␣␣␣(" ≪ world_coordinates[0] ≪ ",␣" ≪
        world_coordinates[1] ≪ ",␣" ≪ world_coordinates[2] ≪ ",␣" ≪ world_coordinates[3] ≪
        ")\n" ≪ "projective_coordinates:␣(" ≪ projective_coordinates[0] ≪
        ",␣" ≪ projective_coordinates[1] ≪ ",␣" ≪ projective_coordinates[2] ≪ ",␣" ≪
        projective_coordinates[3] ≪ ")\n";
  }
  v.push_back(this);
  return v;
}
```

488. Get extremes. [LDF 2002.09.18.] Added this function. Any code that calls *get_extremes*( ) must ensure that *project*( ) has been invoked first.

⟨ Declare **Point** functions 329 ⟩ +≡

```
virtual inline const valarray⟨real⟩ get_extremes() const
{
  return projective_extremes;
}
```

489. Get minimum z. [LDF 2002.09.17.] Added this function.

⟨ Declare **Point** functions 329 ⟩ +≡

```
virtual real get_minimum_z() const;
```

490.

⟨ Define **Point** functions 330 ⟩ +≡
```
real Point::get_minimum_z() const
{
  bool DEBUG = false;   /* true */
  if (DEBUG) cout << "Entering Point::get_minimum_z()" << endl << flush;
  if (DEBUG) cout << "minimum_z == " << projective_extremes[4] << endl << flush;
  if (DEBUG) cout << "Exiting Point::get_minimum_z()" << endl << flush;
  return projective_extremes[4];
}
```

491. Get maximum z. [LDF 2002.09.17.] Added this function.

⟨ Declare **Point** functions 329 ⟩ +≡
```
virtual real get_maximum_z() const;
```

492.

⟨ Define **Point** functions 330 ⟩ +≡
```
real Point::get_maximum_z() const
{
  bool DEBUG = false;   /* true */
  if (DEBUG) cout << "Entering Point::get_maximum_z()" << endl << flush;
  if (DEBUG) cout << "maximum_z == " << projective_extremes[5] << endl << flush;
  if (DEBUG) cout << "Exiting Point::get_maximum_z()" << endl << flush;
  return projective_extremes[5];
}
```

493. Get mean z. [LDF 2003.05.16.] Added this function.

⟨ Declare **Point** functions 329 ⟩ +≡
```
virtual real get_mean_z() const;
```

494.

⟨ Define **Point** functions 330 ⟩ +≡
```
real Point::get_mean_z() const
{
  return ((projective_extremes[4] + projective_extremes[5])/2);
}
```

495. Set extremes. This function sets "extreme" values for the x, y, and z-coordinates. This is, of course, trivial for **Points**, because they only have one x, y and z-coordinate. So the maxima and minima for each coordinate are always the same.

[LDF 2002.10.20.] The programmer who uses *set_extremes*( ) must ensure that *apply_transform*( ) and *project*( ) are invoked before *set_extremes*( )!

Log

[LDF 2002.09.17.] Added this function.
[LDF 2002.09.18.] Changed the name of this function from *set_minimum_z*( ) to *set_extremes*( ).

⟨ Declare **Point** functions 329 ⟩ +≡
```
virtual bool set_extremes();
```

496.

⟨ Define **Point** functions 330 ⟩ +≡

```
bool Point::set_extremes()
{
  bool DEBUG = false;   /* true */
  if (DEBUG) cout << "Entering Point::set_extremes()" << endl << flush;
  for (int i = 0; i < 4; i++) {
    if (projective_coordinates[i] == INVALID_REAL) {
      cerr << "ERROR! In Point::set_extremes():\n" << "projective_coordinates[" << i << "]" <<
          " == INVALID_REAL.\n" << "Setting every element in projective_extremes " <<
          "to INVALID_REAL and returning false.\n" << flush;
      for (int j = 0; j < 6; j++) projective_extremes[j] = INVALID_REAL;
      return false;
    }
  }
  projective_extremes[0] = projective_coordinates[0];   /* min x */
  projective_extremes[1] = projective_coordinates[0];   /* max x */
  projective_extremes[2] = projective_coordinates[1];   /* min y */
  projective_extremes[3] = projective_coordinates[1];   /* max y */
  projective_extremes[4] = projective_coordinates[2];   /* min z */
  projective_extremes[5] = projective_coordinates[2];   /* max z */
  if (DEBUG) cout << "Exiting  Point::set_extremes()" << endl << flush;
  return true;
}
```

497. Comparison classes. [LDF 2003.05.16.] The function classes in this section are used in **Picture** :: *output*( ) to sort the pointers to **Shape** in **vector**⟨**Shape** ∗⟩ *elements*. The argument *sort_value*, which should be one of the constants in **namespace Sorting**, determines which one is used, or if *elements* shouldn't be sorted.

- If **Sorting** :: `MIN_Z` is passed to **Picture** :: *output*( ), *Compare_minimum_z* is used for sorting. The elements are sorted in descending order of the *minimum* z-value from their *projective_extremes*.
- If **Sorting** :: `MAX_Z` is passed to **Picture** :: *output*( ), *Compare_maximum_z* is used for sorting. The elements are then sorted in descending order of the *maximum* z-value from their *projective_extremes*.
- If **Sorting** :: `MEAN_Z` is passed to **Picture** :: *output*( ), *Compare_mean_z* is used for sorting. The elements are then sorted in descending order of of the mean of the minimum and maximum z-values from their *projective_extremes*.

In all three of these cases, the **Shapes** that are furthest from the **Focus** are output first, so that they can be covered, if necessary, by **Shapes** that are closer.

- If **Sorting** :: `NO_SORT` is passed to **Picture** :: *output*( ), *elements* is not sorted, and the **Shapes** are output in the order in which they were drawn or filled.

498. Compare minimum z.

Log

[LDF 2003.05.16.] Added this **class**.

⟨ Define comparison classes 498 ⟩ ≡

```
class Compare_minimum_z {
public: int operator()(const Shape *s1, const Shape *s2) const
  {
    return s1→get_minimum_z() > s2→get_minimum_z();
  }
};
```

See also sections 499 and 500.

This code is cited in section 596.

This code is used in sections 633 and 634.

499. Compare maximum z.

Log

[LDF 2002.09.17.] Added this **class**.
[LDF 2002.09.21.] Changed from "minimum z" to "maximum z". This works for the more common cases.

⟨ Define comparison classes 498 ⟩ +≡

```
class Compare_maximum_z {
public: int operator()(const Shape *s1, const Shape *s2) const
  {
    return s1→get_maximum_z() > s2→get_maximum_z();
  }
};
```

500. Compare mean z.

Log

[LDF 2002.09.17.] Added this **class**.

⟨ Define comparison classes 498 ⟩ +≡

```
class Compare_mean_z {
public: int operator()(const Shape *s1, const Shape *s2) const
  {
    return (((s1→get_minimum_z() + s1→get_maximum_z())/2) >
        ((s2→get_minimum_z() + s2→get_maximum_z())/2));
  }
};
```

501. Output. [LDF 2002.10.26.] *output*( ) is a pure **virtual** function in **Shape**. After the **Shape** pointers on the **vector Picture** :: *shapes* have been extracted, *output*( ) is called for each of the **Shapes** they point to (except for the ones, if any, where *project*( ) failed). *output*( ) writes the METAPOST code to *out_stream*.

Log

[LDF 2002.09.16.] Added **Focus** argument *f*. I want the default to be *default_focus*, but I can't put it in the declaration, as I normally do, because *default_focus* hasn't been defined yet. I've put it in the definition, and it seems to work. Sometimes it doesn't, and I don't know why, nor do I know why it works this time. If I run into problems, this may be the reason. If necessary, I can make a dummy version of this function with no argument that calls *this* version with *default_focus* as its argument.

[LDF 2002.09.17.] Changed the argument *f* from **Focus** to **const Focus** &. Removed the invocations of *apply_transform*( ) and *project*( ) and error handling code to *extract*( ).

[LDF 2002.10.23.] Removed the argument *f*. Since *extract*( ) takes care of applying *project*( ), the *projective_coordinates* are already set, so all *output*( ) needs to do is write them to *out_stream* with the proper METAPOST instructions.

⟨ Declare **Point** functions 329 ⟩ +≡

```
void output();
```

502.

⟨ Define **Point** functions 330 ⟩ +≡

```
void Point::output()
{
  bool DEBUG = false;  /* true */
  if (DEBUG) cout << "Entering Point::output().\n" << flush;
  if (do_output == false) {
    if (DEBUG) cout << "In Point::output(): do_output == false. Returning.\n" << flush;
    return;
  }
  if (drawdot_value == DRAWDOT) out_stream << "drawdot ";
  else if (drawdot_value == UNDRAWDOT) out_stream << "drawdot ";
  else  /* DRAWDOT */
  {
    cerr << "WARNING! Invalid |drawdot_value|: " << drawdot_value <<
        ". Using \"drawdot\"\n" << flush;
#if 0  /* !! Define a class for information on the run state. */
    if (¬Run_State::non_stop) getchar();
#endif
    out_stream << "drawdot ";
  }
  out_stream << "(" << projective_coordinates[0] << measurement_units << ", " <<
      projective_coordinates[1] << measurement_units << ")";
  if (drawdot_color ≠ Colors::default_color) out_stream << " withcolor " << *drawdot_color;
#if 1
  if (pen ≠ "") out_stream << " withpen " << pen;
#endif
  out_stream << ";\n";
  if (DEBUG) cout << "Exiting Point::output().\n" << flush;
}
```

503. Labelling.

504. Label.

505. string argument. [LDF 2002.10.27.] The arguments:

string *text_str* The text for the label.

string *position_str* Indicates the position of the label text relative to the **Point**. The same **strings** are permitted as in METAPOST. They are written unchecked to *out_stream*, so if an invalid **string** is used, it won't cause an error in 3DLDF, but it will in METAPOST. The permitted **strings** are: `"top"` (the default), `"bot"`, `"lft"`, `"rt"`, `"ulft"` (upper left), `"llft"` (lower left), `"urt"` (upper right), `"lrt"` (lower right), and `""` for putting the label right on top of the **Point**. The empty string must be used explicitly, because `"top"` is the default.

bool *dot* If *true*, then **dotlabel** is written to *out_stream* rather than **label**. This argument is mainly for use by the function *dotlabel*( ), which calls *label*( ) with *dot* ≡ *true*.

Picture &*picture* Indicates the **Picture** onto which the **Label** should be placed. The default is *current_picture* .

[LDF 2003.01.15.] TO DO: Add *pen* argument to *label*( ) and *dotlabel*( )!!

Log

[LDF 2002.05.14.] Changed *text_str* so that it is no longer optional. It doesn't make any sense to print empty labels, so I've made it a required argument.

[LDF 2002.11.12.] Added "`\relax`" after the arguments to "`\ARG`" in the TEX code above in order to suppress a space at the beginning of the first line of the following indented paragraph. I couldn't figure out a way of suppressing the space within the definition of `\ARG`.

[LDF 2003.07.09.] Made *text_str*, *position_str*, and *dot* arguments **const**.

⟨ Declare **Point** functions 329 ⟩ +≡
void *label*(**const string** *text_str*, **const string** *position_str* = `"top"`, **const bool** *dot* = *false*, **Picture** &*picture* = *current_picture*) **const**;

506.

⟨ Define **Point** functions 330 ⟩ +≡

```
void Point::label(const string text_str, const string position_str, const bool dot, Picture &picture)
      const { bool DEBUG = false;   /* true */
    if (DEBUG) cout ≪ "Entering␣Point::label()" ≪ "\n" ≪ flush;
    if (Label::DO_LABELS ≡ false) {
      if (DEBUG) cout ≪ "Label::DO_LABELS␣==␣false.␣Returning." ≪ "\n" ≪ flush;
      return;
    }
    if (*this ≡ INVALID_POINT) {
      cerr ≪ "WARNING!␣In␣Point::label():␣" ≪ "*this␣==␣INVALID_POINT.\nNot␣doing␣anyt\
          hing.\n\n" ≪ flush;
      return;
    }
    Label *lbl = new (Label); lbl→pt = create_new < Point > (0);
    *(lbl→pt) = *this;
    if (dot ≡ true) lbl→dot = true;
    lbl→position = position_str;
    if (text_str ≠ "") lbl→text = text_str;
    else lbl→text = "Pt.";
    picture += lbl;   /* [LDF 2002.10.27.] The Label is pushed onto the vector labels in picture. */
    if (DEBUG) cout ≪ "Exiting␣Point::label()" ≪ "\n" ≪ flush;
    return; }
```

507. short argument. [LDF 2003.04.01.] TO DO: Make non-**const** version of this function! TO DO: Make it possible to use `PROJ_VALUES` to use the values in *projective_coordinates* for the label. This will require adding arguments for use by *project*( ).

Log

[LDF 2003.04.01.] Changed this function so that it tests whether *text_short* is equal to `WORLD_VALUES`, `PROJ_VALUES`, `USER_VALUES`, or `VIEW_VALUES`, which are **public const static** data members in **Point**. If *text_short* is equal to `WORLD_VALUES`, ∗*this* is copied and *apply_transform*( ) is called on the copy. This is necessary, because this function is **const**. Then, the updated values in the *world_coordinates* vector of the copy are used for the label.

[LDF 2003.05.06.] Added comparison of *text_short* with `WORLD_VALUES_X_Y`, `PROJ_VALUES_X_Y`, `USER_VALUES_X_Y`, or `VIEW_VALUES_X_Y`, which are used for suppressing the z-coordinate, when the values from one of the sets of coordinates are used for the label. Also, no longer copying ∗*this*, since *get_x*( ), *get_y*( ), and *get_z*( ) are **const** anyway.

[LDF 2003.05.20.] Added "`WORLD_VALUES_Z`" case.

[LDF 2003.05.22.] BUG FIX: The "`WORLD_VALUES_Z`" case started with **if** instead of **else if**. This caused *s.str*( ) to have an erroneous five-digit integer following the closing parenthesis, when `WORLD_VALUES` or `WORLD_VALUES_X_Y` was used. I don't know why this should have been the case, but changing **if** to **else if** fixed the problem. It probably had something to do with the fact that `WORLD_VALUES_Z` had the same value as `VIEW_VALUES_X_Y`. I've fixed this above today, too.

[LDF 2003.06.06.] Changed the case, where *text_short* = `WORLD_VALUES` or *text_short* = `WORLD_VALUES_X_Y`: The coordinates surrounded by parentheses are now printed out using TeX's math mode, i.e., "(x, y, z)" instead of "(x, y, z)".

[LDF 2003.07.09.] Made *text_short*, *position_str*, and *dot* arguments **const**.

⟨ Declare **Point** functions 329 ⟩ +≡
 void *label*(**const short** *text_short*, **const string** *position_str* = "top", **const bool** *dot* = *false*, **Picture** &*picture* = *current_picture*) **const**;

508.

⟨ Define **Point** functions 330 ⟩ +≡

```
void Point::label(const short text_short, const string position_str, const bool dot, Picture
        &picture) const
{
  bool DEBUG = false;    /* true */
  stringstream s;
  if (text_short ≡ WORLD_VALUES ∨ text_short ≡ WORLD_VALUES_X_Y) {
    if (DEBUG) cout ≪ "It's WORLD_VALUES or WORLD_VALUES_X_Y.\n";
    s ≪ "$(" ≪ get_x() ≪ ", " ≪ get_y();
    if (text_short ≡ WORLD_VALUES) s ≪ ", " ≪ get_z();
    s ≪ ")$";
  }
  else if (text_short ≡ WORLD_VALUES_Z) {
    if (DEBUG) cout ≪ "It's WORLD_Z.\n";
    s ≪ get_z();
  }
  else if (text_short ≡ PROJ_VALUES) {
    if (DEBUG) cout ≪ "It's PROJ_VALUES\n";
    cerr ≪ "WARNING! In Point::label():" ≪ endl ≪ "text_short == PROJ_VALUES. " ≪
        "Haven't programmed this case yet.\n" ≪ "Returning.\n\n" ≪ flush;
    return;
  }
  else if (text_short ≡ USER_VALUES) {
    if (DEBUG) cout ≪ "It's USER_VALUES\n";
    cerr ≪ "WARNING! In Point::label():" ≪ endl ≪ "text_short == USER_VALUES. " ≪
        "Haven't programmed this case yet.\n" ≪ "Returning.\n\n" ≪ flush;
    return;
  }
  else if (text_short ≡ VIEW_VALUES) {
    if (DEBUG) cout ≪ "It's VIEW_VALUES\n";
    cerr ≪ "WARNING! In Point::label():" ≪ endl ≪ "text_short == VIEW_VALUES. " ≪
        "Haven't programmed this case yet.\n" ≪ "Returning.\n\n" ≪ flush;
    return;
  }
  else {
    if (DEBUG) cout ≪ "It's some other value.\n";
    s ≪ text_short;
  }
  if (DEBUG) cout ≪ "s.str() == " ≪ s.str() ≪ endl ≪ flush;
  label(s.str(), position_str, dot, picture);
  return;
}
```

509. Dotlabel. TO DO: Add an optional *pen* argument. If it's used, use *drawdot*( ) with the *pen* argument, together with *label*( ). When I do this, I should also add **real** arguments (to both *label*( ) and *dotlabel*( )) for shifting the position of the text, and a version with a **Point** argument for the same purpose. This is so that the dot won't cover the text. [LDF 2003.07.16.]

510. string argument.

Log

[LDF 2003.07.09.] Made *text_str* and *position_str* arguments **const**.

⟨ Declare **Point** functions 329 ⟩ +≡
void *dotlabel*(**const string** *text_str*, **const string** *position_str* = "top", **Picture** &*picture* = *current_picture*) **const**;

511.

⟨ Define **Point** functions 330 ⟩ +≡
```
void Point::dotlabel(const string text_str, const string position_str, Picture &picture) const
{
  label(text_str, position_str, true, picture);
}
```

512. short argument.

Log

[LDF 2003.07.09.] Made *text_short* and *position_str* arguments **const**.

⟨ Declare **Point** functions 329 ⟩ +≡
void *dotlabel*(**const short** *text_short*, **const string** *position_str* = "top", **Picture** &*picture* = *current_picture*) **const**;

513.

⟨ Define **Point** functions 330 ⟩ +≡
```
void Point::dotlabel(const short text_short, const string position_str, Picture &picture) const
{
  label(text_short, position_str, true, picture);
}
```

514. [LDF 2002.09.06.] Commented out ~**Label**( ). This was the cause of a bug that caused a memory fault when I tried to use a label in figure 2 (*beginfig*(2)) after having used it in figure 1 and then invoking *current_picture.clear*( ) in between.

⟨ Define **Label** functions 514 ⟩ ≡
```
#if 0
  Label::~Label()
  {
    if (pt ≠ 0) delete pt;
  }
#endif
```
See also sections 515 and 516.

This code is used in section 633.

515. Get copy of Label.

⟨ Define **Label** functions 514 ⟩ +≡
```
  Label * Label::get_copy() const{ Label *lbl = new (Label); lbl→pt = create_new < Point > (0);
      *(lbl→pt) = *pt;
      lbl→dot = dot;
      lbl→text = text;
      lbl→position = position;
      return lbl; }
```

516. Output Labels. [LDF 2002.10.23.] Declared in `pictures.web`. Must be defined here, because **Point** is an incomplete type there.

Log

[LDF 2002.10.23.] Added arguments *proj* and *factor*.

⟨ Define **Label** functions 514 ⟩ +≡
```
  void Label::output(const Focus &f, const unsigned short proj, real factor, const Transform &t)
  {
    if (¬t.is_identity()) *pt *= t;
    pt→apply_transform();
    if (¬pt→project(f, proj, factor)) {
      cerr ≪ "WARNING!␣In␣Label::output():\n" ≪ "Point␣in␣Label␣cannot␣be␣projected!␣" ≪
          "Not␣printing␣Label\n" ≪ flush;
      return;
    }
    if (dot ≡ true) out_stream ≪ "dot";
    out_stream ≪ "label";
    if (position ≠ "") out_stream ≪ "." ≪ position;
    out_stream ≪ "(btex␣" ≪ text ≪ "␣etex,␣(" ≪ pt→get_x('p',
        false) ≪ Point::measurement_units ≪ ",␣" ≪ pt→get_y('p',
        false) ≪ Point::measurement_units ≪ "));\n";
    return;
  }
```

517. Matrix operations.

518. Multiplication by a Transform with assignment.

Log

[LDF 2002.11.06.] BUG FIX: This function now returns *t* instead of *transform*. This makes it possible to chain expressions using **operator**∗=( ).

⟨ Declare **Point** functions 329 ⟩ +≡
```
Transform operator*=(const Transform &t);
```

519.

⟨ Define **Point** functions 330 ⟩ +≡
```
Transform Point::operator*=(const Transform &t)
{
  return (transform *= t);
}
```

520. Vector operations. [LDF 2002.10.27.] Note that the vector operations don't affect the w coordinate.

Log

[LDF 2002.10.27.] In the functions **operator**+( ),**operator**+=( ), **operator**−( ), and **operator**−=( ): It doesn't seem worth it to write non-**const** versions, although I could. Now using the elements of *p0.world_coordinates* directly instead of using *get_x*( ), *get_y*( ), and *get_z*( ). This is safe, as is calling *apply_transform*( ) on *p*, and saves the cost of three function calls.

---

521. Vector addition.

⟨ Declare **Point** functions 329 ⟩ +≡
```
Point operator+(Point p) const;
```

522.

⟨ Define **Point** functions 330 ⟩ +≡
```
Point Point::operator+(Point p) const
{
  Point a;
  a = *this;
  p.apply_transform();
  a.shift(p.world_coordinates[0], p.world_coordinates[1], p.world_coordinates[2]);
  return a;
}
```

523. Vector addition with assignment.

⟨ Declare **Point** functions 329 ⟩ +≡
```
void operator+=(Point p);
```

524.

⟨ Define **Point** functions 330 ⟩ +≡

```
void Point::operator+=(Point p)
{
  p.apply_transform();
  shift(p.world_coordinates[0], p.world_coordinates[1], p.world_coordinates[2]);
}
```

525. Vector subtraction.

⟨ Declare **Point** functions 329 ⟩ +≡

```
Point operator-(Point p) const;
```

526.

⟨ Define **Point** functions 330 ⟩ +≡

```
Point Point::operator-(Point p) const
{
  Point a(*this);
  p.apply_transform();
  a.shift(-p.world_coordinates[0], -p.world_coordinates[1], -p.world_coordinates[2]);
  return a;
}
```

527. Vector subtraction with assignment.

⟨ Declare **Point** functions 329 ⟩ +≡

```
void operator-=(Point p);
```

528.

⟨ Define **Point** functions 330 ⟩ +≡

```
void Point::operator-=(Point p)
{
  p.apply_transform();
  shift(-p.world_coordinates[0], -p.world_coordinates[1], -p.world_coordinates[2]);
}
```

529. Vector-scalar multipication with assignment.

Log

[LDF 2002.10.27.] Made argument *r* **const**. Changed return value from **Point** & to **void**.

[LDF 2003.05.14.] Changed return value from **void** to **real**. It now returns the argument *r*. This makes it possible to chain invocations of this function.

⟨ Declare **Point** functions 329 ⟩ +≡

```
real operator*=(const real r);
```

530. ?? I'm not sure whether multiplication with a scalar is commutative with transformations. I doubt it. Therefore, I apply *transform* before multiplying.

⟨ Define **Point** functions 330 ⟩ +≡
```
real Point::operator*=(const real r)
{
  apply_transform();
  for (int i = 0; i < 3; i++) world_coordinates[i] *= r;
  return r;
}
```

531. Vector-scalar multipication.

532. Member version (Point first).

Log

[LDF 2002.10.27.] Made this function and the argument r **const**.

⟨ Declare **Point** functions 329 ⟩ +≡
```
Point operator*(const real r) const;
```

533.

⟨ Define **Point** functions 330 ⟩ +≡
```
Point Point::operator*(const real r) const
{
  Point a(*this);
  a.apply_transform();
  a *= r;
  return a;
}
```

534. Non-member version (scalar first).

⟨ Declare non-member non-template functions for **Point** 480 ⟩ +≡
```
Point operator*(const real, const Point &p);
```

535.

⟨ Define non-member non-template functions for **Point** 481 ⟩ +≡
```
Point operator*(const real r, const Point &p)
{
  return p * r;
}
```

536. Unary minus.

Log

[LDF 2002.10.27.] Made this function **const**.

⟨ Declare **Point** functions 329 ⟩ +≡
```
Point operator-() const;
```

537.

⟨ Define **Point** functions 330 ⟩ +≡

```
Point Point::operator-() const
{
  Point a(*this);
  a.apply_transform();
  a *= -1;
  return a;
}
```

538. Vector-scalar division with assignment. ?? I'm not sure whether division with a scalar is commutative with transformations. I doubt it. Therefore, I apply *transform* before dividing.

Log

[LDF 2002.10.27.] Made the argument r **const**.

⟨ Declare **Point** functions 329 ⟩ +≡

```
void operator/=(const real r);
```

539.

⟨ Define **Point** functions 330 ⟩ +≡

```
void Point::operator/=(const real r)
{
  apply_transform();
  for (int i = 0; i < 3; i++) world_coordinates[i] /= r;
}
```

540. Vector-scalar division.

Log

[LDF 2002.10.27.] Made this function and the argument r **const**.

⟨ Declare **Point** functions 329 ⟩ +≡

```
Point operator/(const real r) const;
```

541.

⟨ Define **Point** functions 330 ⟩ +≡

```
Point Point::operator/(const real r) const
{
  Point a(*this);
  a.apply_transform();
  a /= r;
  return a;
}
```

542. Dot product.

Log

[LDF 2002.10.27.] Changed this function and argument *p* to **const**. Now using *world_coordinates* directly instead of *get_x*( ), *get_y*( ), and *get_z*( ).

[LDF 2003.07.11.] Changed, so that if the dot product is less than **Point** :: *epsilon*( ), 0 will be returned.

⟨ Declare **Point** functions 329 ⟩ +≡

```
real dot_product(Point p) const;
```

543.

⟨ Define **Point** functions 330 ⟩ +≡

```
real Point::dot_product(Point p) const
{
  Point a(*this);
  a.apply_transform();
  p.apply_transform();
  real r = ((a.world_coordinates[0] * p.world_coordinates[0]) + (a.world_coordinates[1] *
      p.world_coordinates[1]) + (a.world_coordinates[2] * p.world_coordinates[2]));
  if (fabs(r) < Point::epsilon()) r = 0;
  return r;
}
```

544. Cross product.

Log

[LDF 2002.10.27.] Changed this function and argument *p* to **const**. Now using *world_coordinates* directly instead of *get_x*( ), *get_y*( ), and *get_z*( ).

⟨ Declare **Point** functions 329 ⟩ +≡

```
Point cross_product(Point p) const;
```

545.

⟨ Define **Point** functions 330 ⟩ +≡

Point Point :: *cross_product*(**Point** p) **const**
{
  Point $a(*this)$;
  $a.apply\_transform()$;
  $p.apply\_transform()$;
  Point r;
  $r.world\_coordinates[0] = (a.world\_coordinates[1] * p.world\_coordinates[2]) - (a.world\_coordinates[2] * p.world\_coordinates[1])$; /* x. */
  $r.world\_coordinates[1] = (a.world\_coordinates[2] * p.world\_coordinates[0]) - (a.world\_coordinates[0] * p.world\_coordinates[2])$; /* y. */
  $r.world\_coordinates[2] = (a.world\_coordinates[0] * p.world\_coordinates[1]) - (a.world\_coordinates[1] * p.world\_coordinates[0])$; /* z. */
  return r;
}

546. Magnitude. [LDF 2002.10.27.]

The magnitude of a **Point** is its distance from the origin and is equal to $\sqrt{(x^2 + y^2 + z^2)}$.

Since *floats* are so large anyway, and since I can easily redefine **real** to use **double** or **double double**, (or whatever it's called Look up!!), it's not really necessary to use an algorithm to approximate $\sqrt{(x^2 + y^2 + z^2)}$ (viz., "Pythagorean addition" in Knuth, *Metafont: The Program.* (Get reference!!) However, it might be nice to use it anyway.

Log

[LDF 2002.10.27.] Made this function **const**. Now using *world_coordinates* directly instead of *get_x*( ), *get_y*( ), and *get_z*( ).

⟨ Declare **Point** functions 329 ⟩ +≡

real *magnitude*( ) **const**;

547.

⟨ Define **Point** functions 330 ⟩ +≡

```
real Point::magnitude() const
{
  bool DEBUG = true;    /* false */
  real r;
  real temp;
  Point a(*this);
  a.apply_transform();
  if ((a.world_coordinates[0] > MAX_REAL_SQRT) ∨ (a.world_coordinates[1] >
        MAX_REAL_SQRT) ∨ (a.world_coordinates[2] > MAX_REAL_SQRT)) {
    cerr ≪ "ERROR:␣In␣Point::magnitude().\n" ≪
        "Point␣has␣a␣coordinate␣too␣large␣for␣squaring!\n" ≪ "Returning␣INVALID_REAL.\n";
    return INVALID_REAL;
  }
  r = a.world_coordinates[0] * a.world_coordinates[0];
  temp = a.world_coordinates[1] * a.world_coordinates[1];
  if (MAX_REAL − r < temp) {
    cerr ≪ "In␣magnitude().\n";
    cerr ≪ "Point␣has␣too␣great␣a␣magnitude!\n";    /* !! This show() outputs to stdout. It
        would be nice to output it to stderr instead. Must write function for this. */
    cerr ≪ "Returning␣INVALID_REAL.\n";
    return INVALID_REAL;
  }
  r += temp;
  temp = a.world_coordinates[2] * a.world_coordinates[2];
  if (MAX_REAL − r < temp) {
    cerr ≪ "In␣magnitude().\n";
    cerr ≪ "Point␣has␣too␣great␣a␣magnitude!\n";    /* !! This show() outputs to stdout. It
        would be nice to output it to stderr instead. Must write function for this. */
    cerr ≪ "Returning␣INVALID_REAL.\n";
    return INVALID_REAL;
  }
  r += temp;
  return sqrt(r);
}
```

548. Angle between two vectors.

Log

[LDF 2002.10.27.] Made this function **const**.

[LDF 2003.07.27.] Made the argument p a **const Point** &. No longer copying **this*. Now using *dot_product*( ) instead of calculating the angle "by hand". Simplified the code of the function.

⟨ Declare **Point** functions 329 ⟩ +≡

```
real angle(const Point &p) const;
```

549.

⟨ Define **Point** functions 330 ⟩ +≡

```
real Point::angle(const Point &p) const
{
  bool DEBUG = false;    /* true */
  real mag = magnitude();
  real p_mag = p.magnitude();
  if (mag ≡ INVALID_REAL) {
    cerr ≪ "WARNING! In angle(). magnitude() failed." ≪ " Returning INVALID_REAL.\n";
    return INVALID_REAL;
  }
  else if (mag ≡ 0) {
    if (DEBUG) cerr ≪ "WARNING! In angle().\n" ≪ "*this has magnitude 0.\n" ≪
          "Returning INVALID_REAL.\n";
    return INVALID_REAL;
  }
  else if (p_mag ≡ INVALID_REAL) {
    cerr ≪ "WARNING! In angle(). p.magnitude() failed." ≪ " Returning INVALID_REAL.\n";
    return INVALID_REAL;
  }
  else if (p_mag ≡ 0) {
    if (DEBUG) cerr ≪ "WARNING! In angle().\n" ≪ "p has magnitude 0.\n" ≪
          "Returning INVALID_REAL.\n";
    return INVALID_REAL;
  }
  else return (180/PI * acos(dot_product(p)/(mag * p_mag)));
}
```

550. Unit vector.

Log

[LDF 2002.10.27.] Added a second version. If *assign* is not used, *unit_vector*( ) can be **const**, so I now have a **const** version with no argument and a non-**const** one for assignment that should be called with the argument *true*.

---

551. With assignment. This version should only ever be called with *true* as its argument. Using *false* will work, unless **this* is **const**, in which case it will cause an compilation error. [LDF 2002.10.27.]

If the optional *silent* argument is *true*, warning messages will be suppressed, otherwise, they will be issued. The **const** version below can't have an optional *silent* argument, because that would make a call to this function with one argument ambiguous.

Log

[LDF 2002.10.27.] If *magnitude*( ) fails, *unit_vector*( ) now returns INVALID_POINT instead of *origin*.

[LDF 2003.07.01.] Added the *silent* argument to suppress warning messages. I kept getting warnings when this function was called from intersection functions, in cases where it wasn't a problem, that a **Point** (vector) had 0 magnitude.

⟨ Declare **Point** functions 329 ⟩ +≡

```
Point unit_vector(const bool assign, const bool silent = false);
```

552.

⟨ Define **Point** functions 330 ⟩ +≡

```
Point Point::unit_vector(const bool assign, const bool silent)
{
  if (assign ≡ false) {
    if (¬silent) {
      cerr ≪ "WARNING!␣In␣Point::unit_vector():\n" ≪
          "Don't␣call␣this␣function␣with␣false␣as␣its␣argument.\n" ≪
          "Use␣unit_vector()␣without␣an␣argument␣instead.\n" ≪
          "Calling␣unit_vector()␣without␣an␣argument.\n\n" ≪ flush;
    }
    return unit_vector();
  }
  apply_transform();
  real m = magnitude();
  if (m ≡ 0)
    /* LDF 2002.04.10. Added this error handling code for the case where *this has no magnitude. */
  {
    if (¬silent) {
      cerr ≪ "WARNING!␣In␣Point::unit_vector().\n" ≪
          "Point␣(vector)␣has␣no␣magnitude.␣Returning␣INVALID_POINT.\n\n" ≪ flush;
    }
    return INVALID_POINT;
  }
  for (int i = 0; i < 3; i++) world_coordinates[i] /= m;
  world_coordinates[3] = 1;    /* [LDF 2002.10.27.] Setting the w-coordinate to 1, just to be sure. */
  return *this;
}
```

553. const (no assignment).

⟨ Declare **Point** functions 329 ⟩ +≡

```
Point unit_vector() const;
```

554.

⟨ Define **Point** functions 330 ⟩ +≡

```
Point Point::unit_vector() const
{
  Point a(*this);
  return a.unit_vector(true);
}
```

555. Mediation.

Log

[LDF 2003.12.09.] Changed from a non-member to a **const** member function.

⟨ Declare **Point** functions 329 ⟩ +≡

```
Point mediate(Point p, const real r = .5) const;
```

556.

⟨ Define **Point** functions 330 ⟩ +≡
Point Point :: *mediate*(**Point** p, **const real** r) **const**
{
  Point $t(*this)$;
  $t *= (1 - r)$;
  $p *= r$;
  return $(t + p)$;
}

557. Get normal. *get_normal*( ) must be defined in `paths.web`, because it uses a **Path** in its definition, which is an incompletely defined type in this file.

Log

[LDF 2003.07.11.] Added this declaration.

⟨ Declare **Point** functions 329 ⟩ +≡
Point *get_normal*(**const Point** &p, **const Point** &q) **const**;

558. Comparison.

559. Equality. !! I may have to adjust to value of *eps*. It would be nice to be able to use *epsilon*( ), but for other purposes *epsilon*( ) must be smaller. Transformations seem to cause fairly large inaccuracies in the values of the coordinates, so I need greater tolerance in the functions testing for equality and inequality.

This function could be formulated more succinctly, but I had some trouble getting it to work properly, so I'm leaving it in its more verbose form, in case I have to debug it some more.

560. Non-const version.

Log

[LDF 2002.10.27.] Revised this function. Now using **Point** a and **Point** q. Added *factor* and using it as the argument to *clean*( ) and for calculating *eps*. Since this function is an operator, it's not possible to pass *factor* as an argument, unfortunately. Using *clean*(*factor*) makes it possible to compare the coordinates with 0 directly rather than using *fabs*( ) and *eps*. Also, **operator**≡( ) now uses *world_coordinates* directly rather than *get_x*( ), *get_y*( ), and *get_z*( ).

[LDF 2003.07.09.] Made this function non-**const**, and added **const** version below.

⟨ Declare **Point** functions 329 ⟩ +≡
bool operator≡(**Point** p);

561.

⟨ Define **Point** functions 330 ⟩ +≡

```
bool Point::operator≡(Point p){ bool DEBUG = false;      /* true */
    if (DEBUG) cout ≪ "Entering␣Point::operator==()" ≪ "\n" ≪ flush;
    unsigned short factor = 10;     /* LDF 2002.10.27. Added. */
    clean(factor);
    p.clean(factor);
    real eps = epsilon() * factor;     /* This currently makes eps ≡ .0001. */
    real t_x = world_coordinates[0];
    real t_y = world_coordinates[1];
    real t_z = world_coordinates[2];
    real p_x = p.world_coordinates[0];
    real p_y = p.world_coordinates[1];
    real p_z = p.world_coordinates[2];
```

562. **Points** are frequently compared to INVALID_POINT, so it's best to suppress debugging output for these comparisons, because they're probably not the ones we're interested in.

⟨ Define **Point** functions 330 ⟩ +≡

```
if (t_x ≡ INVALID_REAL ∨ t_y ≡ INVALID_REAL ∨ t_z ≡ INVALID_REAL ∨ p_x ≡ INVALID_REAL ∨ p_y ≡
        INVALID_REAL ∨ p_z ≡ INVALID_REAL) DEBUG = false;
```

563. Debugging output.

⟨ Define **Point** functions 330 ⟩ +≡

```
if (DEBUG) {
    cout ≪ "t_x␣==␣" ≪ t_x ≪ endl ≪ flush;
    cout ≪ "t_y␣==␣" ≪ t_y ≪ endl ≪ flush;
    cout ≪ "t_z␣==␣" ≪ t_z ≪ endl ≪ flush;
    cout ≪ "p_x␣==␣" ≪ p_x ≪ endl ≪ flush;
    cout ≪ "p_y␣==␣" ≪ p_y ≪ endl ≪ flush;
    cout ≪ "p_z␣==␣" ≪ p_z ≪ endl ≪ flush;
}
```

564. Check whether the coordinates of both **Points** are all 0.

Log

[LDF 2002.10.27.] Now that *clean*(10) and *p.clean*(10) are called above, it's no longer necessary to compare the absolute values of the coordinates. I can just compare them with 0 instead.

⟨ Define **Point** functions 330 ⟩ +≡

```
if (t_x ≡ 0 ∧ p_x ≡ 0 ∧ t_y ≡ 0 ∧ p_y ≡ 0 ∧ t_z ≡ 0 ∧ p_z ≡ 0) {
    if (DEBUG) cout ≪ "All␣coordinates␣are␣0,␣returning␣true.\n";
    return true;
}
```

565. Get the signs of the coordinates.

Log

[LDF 2002.10.27.] As in the previous section, changed so that the coordinates are compared with 0, instead of using *fabs*( ) and *eps*.

⟨ Define **Point** functions 330 ⟩ +≡

```
signed short t_x_sign;
signed short t_y_sign;
signed short t_z_sign;
signed short p_x_sign;
signed short p_y_sign;
signed short p_z_sign;
if (t_x ≡ 0) t_x_sign = 0;
else if (t_x < 0) t_x_sign = −1;
else t_x_sign = 1;
if (t_y ≡ 0) t_y_sign = 0;
else if (t_y < 0) t_y_sign = −1;
else t_y_sign = 1;
if (t_z ≡ 0) t_z_sign = 0;
else if (t_z < 0) t_z_sign = −1;
else t_z_sign = 1;
if (p_x ≡ 0) p_x_sign = 0;
else if (p_x < 0) p_x_sign = −1;
else p_x_sign = 1;
if (p_y ≡ 0) p_y_sign = 0;
else if (p_y < 0) p_y_sign = −1;
else p_y_sign = 1;
if (p_z ≡ 0) p_z_sign = 0;
else if (p_z < 0) p_z_sign = −1;
else p_z_sign = 1;
if (DEBUG) {
  cout ≪ "t_x_sign␣==␣" ≪ t_x_sign ≪ endl ≪ flush;
  cout ≪ "t_y_sign␣==␣" ≪ t_y_sign ≪ endl ≪ flush;
  cout ≪ "t_z_sign␣==␣" ≪ t_z_sign ≪ endl ≪ flush;
  cout ≪ "p_x_sign␣==␣" ≪ p_x_sign ≪ endl ≪ flush;
  cout ≪ "p_y_sign␣==␣" ≪ p_y_sign ≪ endl ≪ flush;
  cout ≪ "p_z_sign␣==␣" ≪ p_z_sign ≪ endl ≪ flush;
}
if ((t_x_sign ≠ p_x_sign) ∨ (t_y_sign ≠ p_y_sign) ∨ (t_z_sign ≠ p_z_sign)) {
  if (DEBUG)
    cout ≪ "At␣least␣one␣coordinate␣pair␣has␣signs␣that␣differ.␣" ≪ "Returning␣false.\n";
  return false;
}
```

566. Get the difference between each pair of x, y, and z-coordinates.

⟨ Define **Point** functions 330 ⟩ +≡

```
t_x = fabs(t_x);
t_y = fabs(t_y);
t_z = fabs(t_z);
p_x = fabs(p_x);
p_y = fabs(p_y);
p_z = fabs(p_z);
real delta_x = fabs(t_x − p_x);
real delta_y = fabs(t_y − p_y);
real delta_z = fabs(t_z − p_z);
if (DEBUG) {
  cout ≪ "delta_x␣==␣" ≪ delta_x ≪ endl ≪ flush;
  cout ≪ "delta_y␣==␣" ≪ delta_y ≪ endl ≪ flush;
  cout ≪ "delta_z␣==␣" ≪ delta_z ≪ endl ≪ flush;
}
bool r;    /* [LDF 2002.10.27.] The return value. It's only needed for the sake of the debugging code.
      Otherwise, this function could just return the result of the following expression. */
r = (delta_x < eps ∧ delta_y < eps ∧ delta_z < eps);
if (DEBUG) cout ≪ "r␣==␣" ≪ r ≪ endl ≪ flush;
if (DEBUG) cout ≪ "Exiting␣Point::operator==()" ≪ "\n" ≪ flush;
return r; }
```

567. const version. [LDF 2003.07.09.] This function merely copies **this* and calls the non-**const** version on the copy. Here, *p* can be a **const Point** &, because this function does nothing but pass it to non-**const** version, where it is passed by value.

Log

[LDF 2003.07.09.] Added this version. Made the original version non-**const**.

⟨ Declare **Point** functions 329 ⟩ +≡

```
bool operator≡(const Point &p) const;
```

568.

⟨ Define **Point** functions 330 ⟩ +≡

```
bool Point::operator≡(const Point &p) const
{
  Point copy(*this);
  return (copy ≡ p);
}
```

569. Inequality.

⟨ Declare **Point** functions 329 ⟩ +≡

```
bool operator≠(const Point &p) const;
```

570.

⟨ Define **Point** functions 330 ⟩ +≡
bool Point :: **operator**≠(**const Point** &*p*) **const**
{
return ¬(∗*this* ≡ *p*);
}

571. Intersection.
[LDF 2002.10.27.] *intersection_point*( ) takes four **Point** arguments. It assumes that the first and second represent one line segment and the third and fourth another. It calculates the intersection point of the two lines, if any, and returns a **bool_point**. If the intersection point exists and is on both line segments, the **bool** is *true* and the **Point** is the intersection point. If the intersection point exists, but is not on both line segments, the **bool** is *false* and the **Point** is the intersection point. If no intersection point exists, i.e, the line segments are congruent or parallel, then the **bool** is *false* and the **Point** is `INVALID_POINT`.

LDF Date? Note that *intersection_point*( ) had to be defined as a **static** member function, because **Path** :: *intersection_point*( ) was not able to resolve the call of this version, when it was not callable as "**Point** :: *intersection_point*( )". I got a compiler error, because the call of *intersection_point*( ) inside **Path** :: *intersection_point*( ) with four references to **Points** as arguments was deemed to have "too many arguments" and the references to **Point** couldn't be converted to **const** references to **Path**.

572. Vector version. [LDF 2003.06.29.] Defined in `lines.web`, because **Line** is an incomplete type here.

The algorithm used in this function is taken from Jones, Huw. *Computer Graphics Through Key Mathematics*, pp. 208–311. See **References** for the complete reference.

!! It may be necessary or desirable to add **try**...**catch** blocks where calculations are performed below, just in case overflow occurs.

!! Under Linux, both this function and the version of *intersection_points*( ) using traces have failed in the same cases, which involved coplanar lines which had been rotated about the z-axis or the y and z-axes. I suspect it has to do with the routines for sine and cosine, since I've had trouble with rotation in the constructors for **Polyhedra**. [LDF 2003.06.29.]

Log

[LDF 2002.04.10.] Added this function. It replaces the old version, below.

[LDF 2002.04.12.] Removed the definition of this function to `lines.web`, because it requires the use of **Lines**, and **Line** is an incomplete type here.

[LDF 2003.06.29.] Started using this version again. Bug fixes I've made elsewhere seem to have made it function.

⟨ Declare **Point** functions 329 ⟩ +≡
static bool_point *intersection_point*(**const Point** &*pp0*, **const Point** &*pp1*, **const Point** &*qq0*, **const Point** &*qq1*);

573. Trace version. This function finds the intersection point of two lines by finding the intersection points of the traces of the lines on the major planes. I originally wrote it, because the vector version didn't work. Bug fixes elsewhere seem to have fixed the problem, so this version isn't really needed anymore. [LDF 2003.06.29.]

The **bool** argument *trace* serves only to distinguish this function from the vector version. It doesn't matter whether it's *true* or *false*. [LDF 2003.06.29.]

Log

[LDF 2002.10.27.] Changed the **const Point** & arguments to **Point**, because I had to copy them anyway in order to call *apply_transform*( ) on them.

[LDF 2003.06.29.] Added the **bool** argument *trace*, in order to be able to use both the vector and trace versions. Previously, the vector version didn't work, and was commented-out. Now, bug fixes elsewhere seem to have made the vector version work. Both versions, however, failed under Linux. See the TEX section for the vector version, above, for more information.

⟨ Declare **Point** functions 329 ⟩ +≡
static bool_point *intersection_point*(**Point** *p0*, **Point** *p1*, **Point** *q0*, **Point** *q1*, **const bool** *trace*);

574.

⟨ Define **Point** functions 330 ⟩ +≡
```
bool_point Point::intersection_point(Point p0, Point p1, Point q0, Point q1, const bool trace){
        bool DEBUG = false;     /* true */
    if (DEBUG) {
      cout << "Entering␣Point::intersection_point().\n" << flush;
      p0.show("p0");
      p1.show("p1");
      q0.show("q0");
      q1.show("q1");
    }
    bool_point bp;     /* Return value. */
    if (DEBUG) cout << "Error␣after␣here␣0.\n" << flush;
```

575. Apply the transformations, so we have the correct values for x, y, and z in each of the **Points**. Then assign them to variables.

⟨ Define **Point** functions 330 ⟩ +≡
```
p0.apply_transform();
p1.apply_transform();
q0.apply_transform();
q1.apply_transform();
real p0_x = p0.world_coordinates[0];
real p0_y = p0.world_coordinates[1];
real p0_z = p0.world_coordinates[2];
real p1_x = p1.world_coordinates[0];
real p1_y = p1.world_coordinates[1];
real p1_z = p1.world_coordinates[2];
real q0_x = q0.world_coordinates[0];
real q0_y = q0.world_coordinates[1];
real q0_z = q0.world_coordinates[2];
real q1_x = q1.world_coordinates[0];
real q1_y = q1.world_coordinates[1];
real q1_z = q1.world_coordinates[2];
if (DEBUG) cout << "Error␣after␣here␣1.\n" << flush;
```

576. Get deltas.

⟨ Define **Point** functions 330 ⟩ +≡

```
real delta_x_p = p1_x − p0_x;
real delta_y_p = p1_y − p0_y;
real delta_z_p = p1_z − p0_z;
real delta_x_q = q1_x − q0_x;
real delta_y_q = q1_y − q0_y;
real delta_z_q = q1_z − q0_z;
if (DEBUG) cout ≪ "Error␣after␣here␣2.\n" ≪ flush;
```

577. Slopes for line $\overrightarrow{p_0p_1}$.

⟨ Define **Point** functions 330 ⟩ +≡

```
real slope_p_x_y = (delta_x_p ≠ 0) ? delta_y_p/delta_x_p : INVALID_REAL;
real slope_p_x_z = (delta_x_p ≠ 0) ? delta_z_p/delta_x_p : INVALID_REAL;
real slope_p_z_y = (delta_z_p ≠ 0) ? delta_y_p/delta_z_p : INVALID_REAL;
if (DEBUG) cout ≪ "Error␣after␣here␣3.\n" ≪ flush;
```

578. Slopes for line $\overrightarrow{q_0q_1}$.

⟨ Define **Point** functions 330 ⟩ +≡

```
real slope_q_x_y = (delta_x_q ≠ 0) ? delta_y_q/delta_x_q : INVALID_REAL;
real slope_q_x_z = (delta_x_q ≠ 0) ? delta_z_q/delta_x_q : INVALID_REAL;
real slope_q_z_y = (delta_z_q ≠ 0) ? delta_y_q/delta_z_q : INVALID_REAL;
if (DEBUG) cout ≪ "Error␣after␣here␣4.\n" ≪ flush;
```

579. The traces on the x-y plane. *x_i*, *y_i*, *z_i*, *y_int_p*, and *y_int_q* are set to INVALID_REAL so that I can test for whether the routines below succeed in setting them correctly.

⟨ Define **Point** functions 330 ⟩ +≡

```
real x_i = INVALID_REAL;   /* x-coordinate of the intersection point. */
real y_i = INVALID_REAL;   /* y-coordinate of the intersection point. */
real z_i = INVALID_REAL;   /* z-coordinate of the intersection point. */
real y_int_p = INVALID_REAL;   /* y-intercept of p0p1. */
real y_int_q = INVALID_REAL;   /* y-intercept of q0q1. */
if (DEBUG) cout ≪ "Error␣after␣here␣5.\n" ≪ flush;
if (slope_p_x_y ≠ INVALID_REAL)   /* Δx_p ≠ 0. */
{
  y_int_p = p0_y − (slope_p_x_y ∗ p0_x);
}
if (slope_q_x_y ≠ INVALID_REAL)   /* Δx_q ≠ 0. */
{
  y_int_q = q0_y − (slope_q_x_y ∗ q0_x);
}
if (DEBUG) cout ≪ "Error␣after␣here␣6.\n" ≪ flush;
```

580.

- If both of the traces of $\vec{p}$ and $\vec{q}$ in the x-y plane are parallel to the y-axis (i.e., $\Delta x = 0$), we test whether $p_x = q_x$. If they are, then we set *x_i* to that value. If they're not, the lines don't intersect, so we return `INVALID_POINT`.
- If the trace of $\vec{p}$ or the trace of $\vec{q}$ in the x-y plane is parallel to the y-axis, we set *x_i* to its x-value, because in this case, the intersection point must have this x-value.
- If $\Delta x_p \neq 0$ and $\Delta x_q \neq 0$, we derive *x_i* using the slope and y-intercept of the lines.

⟨ Define **Point** functions 330 ⟩ +≡

```
if (y_int_p ≡ INVALID_REAL ∧ y_int_q ≡ INVALID_REAL)
  /* p⃗_xz and q⃗_xz are both parallel to the z-axis. */
{
  if (DEBUG) cout ≪ "Error␣after␣here␣7.\n" ≪ flush;
  if (p0_x ≡ q0_x)     /* They have the same value for x. */
  {
    if (DEBUG) cout ≪ "Error␣after␣here␣8.\n" ≪ flush;
    if (DEBUG) cout ≪ "Traces␣on␣x-y␣plane␣are␣coincident.\n" ≪ flush;
    x_i = p0_x;
    real y_int_p_z = INVALID_REAL;
    real y_int_q_z = INVALID_REAL;
    if (slope_p_z_y ≠ INVALID_REAL) y_int_p_z = p0_y − slope_p_z_y ∗ p0_z;
    if (slope_q_z_y ≠ INVALID_REAL) y_int_q_z = q0_y − slope_q_z_y ∗ q0_z;
    if (DEBUG) cout ≪ "Error␣after␣here␣9.\n" ≪ flush;
    if (slope_p_z_y ≡ INVALID_REAL ∧ slope_q_z_y ≡ INVALID_REAL) {
      if (DEBUG) cout ≪ "Both␣traces␣on␣z-y␣plane␣are␣vertical\n" ≪ flush;
      if (DEBUG) cout ≪ "Error␣after␣here␣10.\n" ≪ flush;
      if (p0_z ≡ q0_z) cerr ≪ "Lines␣are␣coincident.\n";
      else cerr ≪ "Lines␣do␣not␣intersect.\n";
      cerr ≪ "Returning␣INVALID_BOOL_POINT.\n\n" ≪ flush;
      return INVALID_BOOL_POINT;
    }
    else if (slope_p_z_y ≡ INVALID_REAL) {
      if (DEBUG) cout ≪ "The␣p-trace␣is␣vertical\n" ≪ flush;
      if (DEBUG) cout ≪ "Error␣after␣here␣11.\n" ≪ flush;
      z_i = p0_z;
      y_i = z_i ∗ slope_q_z_y + y_int_q_z;
      if (DEBUG) {
        cout ≪ "x_i␣==␣" ≪ x_i ≪ endl ≪ flush;
        cout ≪ "y_i␣==␣" ≪ y_i ≪ endl ≪ flush;
        cout ≪ "z_i␣==␣" ≪ z_i ≪ endl ≪ flush;
        cout ≪ "slope_q_z_y␣==␣" ≪ slope_q_z_y ≪ endl ≪ flush;
        cout ≪ "y_int_q_z␣==␣" ≪ y_int_q_z ≪ endl ≪ flush;
      }
    }
    else if (slope_q_z_y ≡ INVALID_REAL) {
      if (DEBUG) cout ≪ "The␣q-trace␣is␣vertical\n" ≪ flush;
      if (DEBUG) cout ≪ "Error␣after␣here␣12.\n" ≪ flush;
      z_i = q0_z;
      y_i = z_i ∗ slope_p_z_y + y_int_p_z;
      if (DEBUG) {
        cout ≪ "x_i␣==␣" ≪ x_i ≪ endl ≪ flush;
        cout ≪ "y_i␣==␣" ≪ y_i ≪ endl ≪ flush;
```

```
            cout ≪ "z_i␣==␣" ≪ z_i ≪ endl ≪ flush;
            cout ≪ "slope_p_z_y␣==␣" ≪ slope_p_z_y ≪ endl ≪ flush;
            cout ≪ "y_int_p_z␣==␣" ≪ y_int_p_z ≪ endl ≪ flush;
          }
        }
        else {
          if (DEBUG) cout ≪ "Neither␣trace␣is␣vertical\n" ≪ flush;
          if (DEBUG) cout ≪ "Error␣after␣here␣13.\n" ≪ flush;
          z_i = (y_int_q_z − y_int_p_z)/(slope_p_z_y − slope_q_z_y);
          y_i = slope_p_z_y ∗ z_i + y_int_p_z;
          if (DEBUG) {
            cout ≪ "x_i␣==␣" ≪ x_i ≪ endl ≪ flush;
            cout ≪ "y_i␣==␣" ≪ y_i ≪ endl ≪ flush;
            cout ≪ "z_i␣==␣" ≪ z_i ≪ endl ≪ flush;
            cout ≪ "slope_p_z_y␣==␣" ≪ slope_p_z_y ≪ endl ≪ flush;
            cout ≪ "y_int_p_z␣==␣" ≪ y_int_p_z ≪ endl ≪ flush;
            cout ≪ "slope_q_z_y␣==␣" ≪ slope_q_z_y ≪ endl ≪ flush;
            cout ≪ "y_int_q_z␣==␣" ≪ y_int_q_z ≪ endl ≪ flush;
          }
        }
      }
      else      /∗ They don't have the same value for x. ∗/
      {
        if (DEBUG) cout ≪ "Error␣after␣here␣14.\n" ≪ flush;
        cerr ≪ "Lines␣do␣not␣intersect:\n";
        cerr ≪ "(" ≪ p0_x ≪ ",␣" ≪ p0_y ≪ ",␣" ≪ p0_z ≪ ")␣--␣(" ≪ p1_x ≪ ",␣" ≪ p1_y ≪
            ",␣" ≪ p1_z ≪ ")\n(" ≪ q0_x ≪ ",␣" ≪ q0_y ≪ ",␣" ≪ q0_z ≪ ")␣--␣(" ≪ q1_x ≪
            ",␣" ≪ q1_y ≪ ",␣" ≪ q1_z ≪ ")\nReturning␣INVALID_BOOL_POINT.\n" ≪ flush;
        return INVALID_BOOL_POINT;
      }
    }
    else if (y_int_p ≡ INVALID_REAL)      /∗ p⃗_xy is parallel to the y-axis. ∗/
    {
      if (DEBUG) cout ≪ "Error␣after␣here␣15.\n" ≪ flush;
      x_i = p0_x;
      y_i = slope_q_x_y ∗ x_i + y_int_q;
    }
    else if (y_int_q ≡ INVALID_REAL)      /∗ q⃗_xy is parallel to the y-axis. ∗/
    {
      if (DEBUG) cout ≪ "Error␣after␣here␣16.\n" ≪ flush;
      x_i = q0_x;
      y_i = slope_p_x_y ∗ x_i + y_int_p;
    }
```

581. [LDF 2002.11.12.] !! BUG: Occurred when I tried to find an intersection of two lines in the x-z plane. This code shouldn't be reached. Rotating the objects 90° around the x-axis, putting them into the x-y plane, fixed the problem. Obviously, the case that the objects are in the x-z plane already isn't handled properly.

⟨ Define **Point** functions 330 ⟩ +≡

```
else    /* Neither q_xy nor q_xy is parallel to the y-axis. */
{
  if (DEBUG)    /* [LDF 2002.11.12.] Start working on finding bug here. */
  {
    cout << "Error after here 17.\n" << flush;
    cout << "slope_p_x_y == " << slope_p_x_y << endl << flush;
    cout << "slope_q_x_y == " << slope_q_x_y << endl << flush;
  }
  if (slope_p_x_y ≠ slope_q_x_y) {
    x_i = (y_int_q − y_int_p)/(slope_p_x_y − slope_q_x_y);
    y_i = slope_p_x_y * x_i + y_int_p;
  }
}
```

582. The trace on the x-z plane. We don't need to do this if we've calculated z_i above, in the case that the traces on the x-y plane are coincident.

⟨ Define **Point** functions 330 ⟩ +≡

```
if (z_i ≡ INVALID_REAL) {
if (DEBUG) cout << "Error after here 18.\n" << flush;
real z_int_p = INVALID_REAL;    /* z-intercept of p0p1. */
real z_int_q = INVALID_REAL;    /* z-intercept of q0q1. */
if (slope_p_x_z ≠ INVALID_REAL)    /* Δx_p ≠ 0. */
{
  z_int_p = p0_z − (slope_p_x_z * p0_x);
}
if (slope_q_x_z ≠ INVALID_REAL)    /* Δx_q ≠ 0. */
{
  z_int_q = q0_z − (slope_q_x_z * q0_x);
}
if (DEBUG) {
  cout << "z_int_p == " << z_int_p << endl << flush;
  cout << "z_int_q == " << z_int_q << endl << flush;
}
```

583. [LDF 2003.06.24.] x_i will be equal to INVALID_REAL, if the traces of the lines on the x-y plane were colinear.

Log

[LDF 2003.06.24.] Added this conditional.

⟨ Define **Point** functions 330 ⟩ +≡

```
if (x_i ≡ INVALID_REAL ∧ ¬(z_int_p ≡ INVALID_REAL ∨ z_int_q ≡ INVALID_REAL)) {
  x_i = (z_int_q − z_int_p)/(slope_p_x_z − slope_q_x_z);
  y_i = p0_y;
}
```

584. In the following case, $\vec{p}_{xz}$ and $\vec{q}_{xz}$ are both parallel to the z-axis. They have the same value for x. We've set *x_i* above, so there's no need to do so here again.

⟨ Define **Point** functions 330 ⟩ +≡

```
if (z_int_p ≡ INVALID_REAL ∧ z_int_q ≡ INVALID_REAL) {
  if (DEBUG) cout ≪ "Error␣after␣here␣19.\n" ≪ flush;
  if (p0_x ≠ q0_x) {
    if (DEBUG) cout ≪ "Error␣after␣here␣20.\n" ≪ flush;
    cerr ≪ "Lines␣do␣not␣intersect:\n" ≪ "(" ≪ p0_x ≪ ",␣" ≪ p0_z ≪ ",␣" ≪
        p0_z ≪ ")␣--␣(" ≪ p1_x ≪ ",␣" ≪ p1_z ≪ ",␣" ≪ p1_z ≪ ")\n(" ≪ q0_x ≪
        ",␣" ≪ q0_z ≪ ",␣" ≪ q0_z ≪ ")␣--␣(" ≪ q1_x ≪ ",␣" ≪ q1_z ≪ q1_z ≪
        ")\nReturning␣INVALID_BOOL_POINT.\n" ≪ flush;
    return INVALID_BOOL_POINT;
  }
}
else if (z_int_p ≡ INVALID_REAL) {
  if (DEBUG) cout ≪ "Error␣after␣here␣21.\n" ≪ flush;
  z_i = p0_z;
}
else if (z_int_q ≡ INVALID_REAL) {
  if (DEBUG) cout ≪ "Error␣after␣here␣22.\n" ≪ flush;
  z_i = q0_z;
}
else {
  if (DEBUG) cout ≪ "Error␣after␣here␣23.\n" ≪ flush;
  z_i = slope_p_x_z * x_i + z_int_p;
}
}
```

585. [LDF 2002.10.27.] If *x_i*, *y_i*, and *z_i* are all valid, set *bp.pt* using those values. Otherwise, set it to `INVALID_POINT`. If this **Point** is on both of the line segments $\overrightarrow{p_0 p_1}$ and $\overrightarrow{q_0 q_1}$, set *bp.b* to *true*, otherwise set it to *false*.

```
⟨ Define Point functions 330 ⟩ +≡
  if (DEBUG) {
    cout ≪ "Error␣after␣here␣24.\n" ≪ flush;
    cout ≪ "x_i␣==␣" ≪ x_i ≪ endl ≪ flush;
    cout ≪ "y_i␣==␣" ≪ y_i ≪ endl ≪ flush;
    cout ≪ "z_i␣==␣" ≪ z_i ≪ endl ≪ flush;
  }
  if (x_i ≡ INVALID_REAL ∨ y_i ≡ INVALID_REAL ∨ z_i ≡ INVALID_REAL) {
    return INVALID_BOOL_POINT;
    if (DEBUG) cout ≪ "Returning␣INVALID_BOOL_POINT.\n" ≪ flush;
  }
  else {
    bp.pt.set(x_i, y_i, z_i);
    if (DEBUG) bp.pt.show("bp.pt");
  }
  if (bp.pt.is_on_segment(p0, p1).first ≡ false ∨ bp.pt.is_on_segment(q0, q1).first ≡ false) bp.b = false;
  else bp.b = true;
  if (DEBUG) {
    cout ≪ "bp.b␣==␣" ≪ bp.b ≪ endl;
    cout ≪ "Exiting␣Point::intersection_point().\n" ≪ flush;
  }
  return bp; }
```

586. Picture functions. These functions must be defined here, because they use types which are incompletely defined in `pictures.web`.

587. Assignment operator. !! PORTING [LDF 2002.12.05.] Moved here from `pictures.web`. See that file for explanation.

```
⟨ Define Picture functions 264 ⟩ +≡
  void Picture::operator=(const Picture &p)
  {
    clear();
    transform = p.transform;
    for (vector⟨Shape *⟩::const_iterator iter = p.shapes.begin(); iter ≠ p.shapes.end(); iter++) {
      shapes.push_back((*iter)→get_copy());
    }
    Label *lbl;
    for (vector⟨Label *⟩::const_iterator iter = p.labels.begin(); iter ≠ p.labels.end(); iter++) {
      lbl = (*iter)→get_copy();
      labels.push_back(lbl);
    }
  }
```

588. Copy constructor. !! PORTING [LDF 2002.12.05.] Moved here from `pictures.web`. See that file for explanation.

```
⟨ Define Picture functions 264 ⟩ +≡
  Picture::Picture(const Picture &p)
  : do_labels(true) {
```

```
    *this = p;
  }
```

589. Combining Pictures. [LDF 2002.04.17.] Added this function.

Log

[LDF 2002.10.29.] Made *p* **const** and fixed bugs that changed *p* (see below).

[LDF 2002.10.29.] BUG FIX: Now, *p.transform* is applied to ∗(*shapes.back*()), previously it was applied to ∗∗*iter*, which was not what I wanted.

[LDF 2002.10.29.] BUG FIX: Now, *p.transform* is applied to ∗(*labels.back*()→*pt*), previously it was applied to ∗((∗*iter*)→*pt*), which is not what I wanted.

[LDF 2002.10.29.] BUG FIX: Now, *p.transform.reset*() is no longer called.

⟨ Define **Picture** functions 264 ⟩ +≡

```
  void Picture::operator+=(const Picture &p)
  {
    for (vector⟨Shape *⟩::const_iterator iter = p.shapes.begin(); iter ≠ p.shapes.end(); iter++) {
      shapes.push_back((*iter)→get_copy());

        /* [LDF 2002.10.29.] Normally, transform in a Picture is applied to its Shapes when it's output,
           however, it must be done now for the copies of the Shapes from Picture p that are copied
           onto *this, because p.transform is only known within p; the Shapes don't "know" about it. */
      *(shapes.back()) *= p.transform;
    }
    for (vector⟨Label *⟩::const_iterator iter = p.labels.begin(); iter ≠ p.labels.end(); iter++) {
      labels.push_back((*iter)→get_copy());

      *(labels.back()→pt) *= p.transform;
    }
  }
```

590. Clear Picture.

⟨ Define **Picture** functions 264 ⟩ +≡

```
  void Picture::clear()
  {
    bool DEBUG = false;    /* true */
    if (DEBUG) cout ≪ "Entering␣Picture::clear().\n" ≪ flush;
    if (shapes.size() ≤ 0 ∧ labels.size() ≤ 0) return;
    transform.reset();
    for (vector⟨Shape *⟩::iterator iter = shapes.begin(); iter ≠ shapes.end(); iter++) {
      (*iter)→clear();
      delete (*iter);
    }
    for (vector⟨Label *⟩::iterator iter = labels.begin(); iter ≠ labels.end(); iter++) {
        /* ?? I tried to use ~Label() here, but it didn't work. I got run-time errors having to do with
           "Unaligned access pid=299273..." (didn't understand). This works, though. If I change the
           definition of Label, I'll have to make corresponding changes here. */
      delete (*iter)→pt;
      delete (*iter);
    }
    shapes.clear();
```