3DLDF: The Program
Version 1.1.5.1
by Laurence D. Finston

January 2004

1. Copyright and License.

3DLDF is a GNU package. It is part of the GNU Project of the Free Software Foundation and is published under the GNU General Public License. See the website `http://www.gnu.org` for more information. 3DLDF is available for downloading from `http://www.gnu.org/3DLDF`. 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:
`bug-3DLDF@gnu.org`

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
Kreuzbergring 41
D-37075 Goettingen
Germany

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 = "$Id:␣shapes.web,v␣1.4␣2004/01/12␣21:32:30␣lfinstol␣Exp␣$";
```

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 ≠ 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 $\overrightarrow{p_0p_1}$ 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* ≡ 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* ≡ 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 $\overrightarrow{p_0p_1}$ 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 $dot \equiv true$.

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. */
  $r = (delta\_x < eps \land delta\_y < eps \land delta\_z < eps)$;
  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();
```

```
    labels.clear();
    if (DEBUG) cout << "Exiting Picture::clear().\n" << flush;
  }
```

591. Output. The arguments:

- *sort_value* is used to determine how to sort the **Shapes**. The values to be used are found in **namespace Sorting**. **Sorting**::NO_SORT is used, if they shouldn't be sorted, because we will have already drawn them in the order we want them rendered. **Sorting**::MAX_Z is used for sorting them according to their maximum z-coordinate for "furthest-first" output. **Sorting**::MIN_Z is used for sorting them according to their maximum z-coordinate for "nearest-last" output. So far, no other types of sorting have been defined.

The simple painter's algorithm implemented here for surface hiding fails for **Shapes** where one **Shape** is partly in front of and partly behind another. For these cases, it will be necessary to find the intersection points and divide the **Shapes** into parts. TO DO: Implement a routine for dividing up **Shapes**. This will not be done soon!

do_warnings: Sometimes we'll want use the *min_x_proj*, *max_x_proj*, etc., arguments to cut off parts of the image, or we'll deliberately place the **Focus** where it won't be able to "see" part of the image. In these cases, it will be annoying to see the warnings.

Log

[LDF 2002.09.21.] Added the arguments *do_sort* and *do_warnings*.

[LDF 2003.05.16.] Changed **bool** *do_sort* to **const unsigned short** *sort_value*. About to add **namespace Sorting** with constants for different ways of sorting, i.e., "no sort", "nearest-last", or "furthest-first".

---

592. Focus argument.

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

```
  void Picture::output(const Focus &f, const unsigned short proj, real factor, const
        unsigned short sort_value, const bool do_warnings, const real min_x_proj, const real
        max_x_proj, const real min_y_proj, const real max_y_proj, const real min_z_proj, const real
        max_z_proj){ bool DEBUG = false;    /* true */
    using namespace Sorting;
    if (DEBUG) {
      cout << "Entering Picture::output(const Focus& ...).\n" << flush;
      cout << "min_x_proj == " << min_x_proj << endl << flush;
      cout << "max_x_proj == " << max_x_proj << endl << flush;
      cout << "min_y_proj == " << min_y_proj << endl << flush;
      cout << "max_y_proj == " << max_y_proj << endl << flush;
    }    /* Check whether the vector shapes has anything in it. If it doesn't, return. */
    if (shapes.size() ≤ 0 ∧ labels.size() ≤ 0) {
      if (DEBUG) cout << "Picture is empty. Returning.\n" << flush;
      return;
    }
```

593. [LDF 2002.09.17.] Some **Shapes** may consist of other **Shapes**, and not have an *output*( ) function of their own, so we must extract their contents recursively until we get to **Shapes** that have one. So far, only **Point**, **Path**, and **Solid** have *output*( ) functions, and all other **Shapes** reduce to **Paths** or **Solids**.

[LDF 2002.09.17.] *extract*( ) checks that all of the **Points** contained in the **Shape** can be projected with the **Focus** that is being used. If any of them cannot be, then *extract*( ) returns an empty **vector** of **Shapes**. This means that any **Shape** must be entirely projectable; partial **Shapes** will not be output. problem, too.

Log

[LDF 2003.01.05.] Modified the TEX text above to account for the fact that I've added **Solid**.

[LDF 2003.01.05.] BUG FIX: Moved the code that causes *transform* to be applied to the elements of *shapes*. This is now done *before* the *extremes* are set. The way it was before didn't work properly, because *extract*( ) used the untransformed values to decide whether a **Shape** was projectable. In order to do this, I had to make *apply_transform*( ) a **Shape** function.

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

```
vector⟨Shape *⟩ v;
vector⟨Shape *⟩ elements;
vector⟨Shape *⟩::iterator iter;
bool do_transform = ¬transform.is_identity( );
DEBUG = false;    /* true */
if (DEBUG) {
  if (do_transform) {
    cout ≪ "Applying␣transform.\n";
    transform.show("transform:");
  }
  else cout ≪ "Not␣applying␣transform.\n";
}
for (iter = shapes.begin( ); iter ≠ shapes.end( ); ++iter) {
  if (do_transform) {
    (**iter) *= transform;
  }
  v = (*iter)→extract(f, proj, factor);
  if (DEBUG ∧ v.size( ) ≡ 0)
    cerr ≪ "WARNING!␣In␣Picture::output():\n" ≪ "extract()␣returned␣an␣empty␣vector.␣" ≪
        "Continuing.\n" ≪ flush;
  for (vector⟨Shape *⟩::iterator i = v.begin( ); i ≠ v.end( ); ++i) {
    elements.push_back(*i);
  }
}
DEBUG = false;
```

594. Set the extremes for the **Shape** and handle the error if it returns *false*. (LDF Undated)

Log

[LDF 2002.09.18.] Changed the error handling code below. Formerly, *get_minimum_z*( ) was invoked, but this is unnecessary, since *set_extremes*( ) returns *false* if something goes wrong with setting the extreme values for the **Shape**.

```
⟨ Define Picture functions 264 ⟩ +≡
  {    /* [LDF 2003.01.05.] Beginning of group. */
  valarray⟨real⟩ extremes(6,0); for (iter = elements.begin(); iter ≠ elements.end(); ++iter) {
  if (DEBUG) cout ≪ "About␣to␣set␣extremes.\n" ≪ flush;
  if (¬(**iter).set_extremes()) {
    cerr ≪ "ERROR!␣In␣Picture::output():\n" ≪ "set_extremes()␣returned␣false.␣" ≪
        "Suppressing␣output␣for␣this␣Shape*.\n" ≪ flush;
    (**iter).suppress_output();
  }
```

595. [LDF 2002.09.18.] Added this routine. It checks for whether the values in the **valarray** ⟨**real**⟩ *projective_extremes* in the **Shape** fall within the limits given by the *min_x_proj*, *max_x_proj*, *min_y_proj*, and *max_x_proj* arguments to this function (**Picture** :: *output*( )). (Note that *min_z_proj* and *max_z_proj* are currently not checked.) If they don't, the **Shape** ∗ is removed from *elements*. Note that the projected z-coordinates are not currently checked, but they are used for ordering the **Shapes** for output (furthest away first).

```
⟨ Define Picture functions 264 ⟩ +≡
  extremes = (**iter).get_extremes();
  if (DEBUG)    /* [LDF 2002.09.21.] Show the extremes for this Shape. */
  {
    for (int i = 0; i < 4; i++) {
      cout ≪ "extremes[" ≪ i ≪ "]␣==␣" ≪ extremes[i] ≪ "\n";
    }
    cout ≪ "extremes[0]␣<␣min_x_proj␣==␣" ≪ (extremes[0] < min_x_proj) ≪ endl ≪ flush;
    cout ≪ "extremes[1]␣>␣max_x_proj␣==␣" ≪ (extremes[1] > max_x_proj) ≪ endl ≪ flush;
    cout ≪ "extremes[2]␣<␣min_y_proj␣==␣" ≪ (extremes[2] < min_y_proj) ≪ endl ≪ flush;
    cout ≪ "extremes[3]␣>␣max_y_proj␣==␣" ≪ (extremes[3] > max_y_proj) ≪ endl ≪ flush;
  }
  if (extremes[0] < min_x_proj ∨ extremes[1] > max_x_proj ∨ extremes[2] < min_y_proj ∨ extremes[3] >
        max_y_proj) {
    if (do_warnings ≡ true) {
      cerr ≪ "WARNING!␣In␣Picture::output():\n" ≪ "Shape␣lies␣outside␣the␣limits␣for␣this␣\
          invocation␣of␣output().\n" ≪ "Suppressing␣output␣for␣this␣Shape*.\n" ≪ flush;
    }
    (**iter).suppress_output();
  }
  }    /* for */
  }    /* End of group. */
```

596. [LDF 2003.05.16.] Sorting can be performed in different ways, depending on the *sort_value* argument. This is explained in ⟨ Define comparison classes 498 ⟩.

[LDF 2002.09.18.] It's necessary to make sure that sorting is only performed if *elements* is non-empty. It could be empty now, if the error handling code above has removed all of the elements because *set_extremes*( ) returned *false* for all of them. We can't just return, because there might still be **Labels** on the **Picture**.

```
⟨ Define Picture functions 264 ⟩ +≡
  if (elements.size( ) > 0) {
    if (sort_value ≡ MIN_Z) sort(elements.begin( ), elements.end( ), Compare_minimum_z( ));
    else if (sort_value ≡ MAX_Z) sort(elements.begin( ), elements.end( ), Compare_maximum_z( ));
    else if (sort_value ≡ MEAN_Z) sort(elements.begin( ), elements.end( ), Compare_mean_z( ));
    if (DEBUG) {
      if (sort_value ≡ MIN_Z) cout ≪ "␣***␣MIN_Z␣sort.␣***\n";
      else if (sort_value ≡ MAX_Z) cout ≪ "␣***␣MAX_Z␣sort.␣***\n";
      else if (sort_value ≡ MEAN_Z) cout ≪ "␣***␣MEAN_Z␣sort.␣***\n";
      for (iter = elements.begin( ); iter ≠ elements.end( ); ++iter) {
        cout ≪ "Min␣z:␣" ≪ (**iter).get_minimum_z( ) ≪ endl ≪ "Max␣z:␣" ≪
            (**iter).get_maximum_z( ) ≪ endl ≪ "Mean␣z:␣" ≪ (**iter).get_mean_z( ) ≪ endl ≪ endl;
      }
      cout ≪ "****␣End␣of␣result␣of␣sort:\n\n" ≪ flush;
    }
    for (iter = elements.begin( ); iter ≠ elements.end( ); ++iter) {
      (**iter).output( );
      (**iter).unsuppress_output( );    /* [LDF 2002.09.18.] With a different Focus or different limiting
            values for the projection, this Shape might be projectable, so we reset do_output to true. If it
            can't be projected the next time, suppress_output( ) will be invoked again. */
    }
  }
```

597. Output the labels. LDF Undated. It is necessary to output the labels last because they might otherwise be drawn over by *fill*( ) or *filldraw*( ) commands.

[LDF 2002.09.17.] I'm not bothering to sort the labels so that the ones behind can be hidden by the ones in front. Labels should all be visible and are not put into perspective, so they shouldn't overlay one another.

[LDF 2002.04.25.] Added following conditional. 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.

```
⟨ Define Picture functions 264 ⟩ +≡
  if (do_labels ≡ true) {
    for (vector⟨Label *⟩::iterator i = labels.begin( ); i ≠ labels.end( ); ++i) {
        /* [LDF 2002.09.17.] Simplified the following code. Formerly, there was a conditional here that
          chose which version of Label::output( ) to call. I've removed the version without a Transform
          argument and invoke Transform::is_identity( ) in Label::output(const Focus &, const
          Transform &). [LDF 2002.04.25.] This applies transform to the Point Label::pt. */
      (**i).output(f, proj, factor, transform);
    }
  }    /* [LDF 2002.04.25.] Added following line. This fixes a bug. If I don't reset transform to identity,
        it will be applied again each time I output a Picture, which is not what I want. */
  transform.reset( );
  if (DEBUG) cout ≪ "Exiting␣Picture::output(const␣Focus&␣...).\n" ≪ flush;
  }
```

598. No Focus argument.

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

void Picture :: *output*(**const unsigned short** *proj*, **real** *factor*, **const unsigned short** *sort_value*, **const bool** *do_warnings*, **const real** *min_x_proj*, **const real** *max_x_proj*, **const real** *min_y_proj*, **const real** *max_y_proj*, **const real** *min_z_proj*, **const real** *max_z_proj*)

```
{
  output(default_focus, proj, factor, sort_value, do_warnings, min_x_proj, max_x_proj, min_y_proj,
      max_y_proj, min_z_proj, max_z_proj);
}
```

599. Focus.

600. Focus class definition. [LDF 2002.09.18.] Made **Focus** a **class** (it was formerly a **struct**). Added **char** *axis* data member. It indicates the axis to which *position* should be transformed to when it's put in standard position. This determines which plane the image is projected onto. If *axis* ≡ 'z' (the default), the image is projected onto the x-y plane, if *axis* ≡ 'x', the z-y plane, if 'y', the x-z plane.

At this time, I'm not adding the routines that will do this, which will entail changing *transform* and possibly *persp*, I'm just adding *axis*. I must also add a function for changing axis without changing any of the other data members. TO DO: Add these routines!!

[LDF 2002.09.14.] Added **Transform** *persp*. It's needed because I need to get the z value of the *world_coordinates* after the transformation that puts the **Focus** into standard position, but before the perspective transformation is performed. This z value can be used in an algorithm for surfaces hiding. If this were not the case, I could combine the transformations, because matrix multiplication is associative (it is not, however, commutative, except with special matrices).

[LDF 2002.09.11.] Added **class Focus** and the following constructors: The default constructor with no arguments, the one with two **Point** arguments and a **real** *argument*, and the one with seven **real** arguments.

format *Focus* *int*

⟨ Define **class Focus** 600 ⟩ ≡

```
class Focus {
  Point position;
  Point direction;
  Point up;
  real distance;
  real angle;
  char axis;
  Transform transform;
  Transform persp;
public: ⟨ Declare Focus functions 602 ⟩
};
```

This code is used in sections 633 and 634.

601. Constructors and setting functions. [LDF 2002.09.22.] TO DO: Check *magnitude* of *direction* − *position* and make sure it's non-zero!!

[LDF 2002.10.13.] The effect of using an *angle* ≠ 0 is similar to that of rotating a camera about an axis through its aperture and perpendicular to the surface of the lens. Because this is possible, it is necessary to indicate the upward direction of a projection. The **Point** *up* does this. It is determined in the constructors and setting functions by the vector *direction* − *position* and *angle*. *up* is first set to $(0, 1, 0)$ if *axis* ≡ 'x' or *axis* ≡ 'z', or $(1, 0, 0)$ if *axis* ≡ 'y'. If *angle* ≠ 0, *up* and *transform* are then rotated by −*angle*. Then, *up* is transformed by the inverse of *transform*, in order to put it in the correct location with respect to *position*. This location is "above" *position* by definition.

[LDF 2002.10.13.] Changed all of the constructors and setting functions except for the default constructor and the first non-default constructor. Now, all the others use the latter to create a **Focus** locally and use **Focus**::**operator**=( ), which I've defined today, to assign to *$*this$*. This eases maintenance and cuts down on the potential for error through inconsistencies in the different constructors and setting functions.

602. Default constructor. (No arguments).

⟨ Declare **Focus** functions 602 ⟩ ≡
  Focus( )
  { }

See also sections 604, 606, 609, 611, 613, 615, 617, 620, 621, 622, 623, 624, 625, 627, and 628.

This code is used in section 600.

603. real arguments. The first three **real** arguments are for the coordinates of the center of projection (the focus in the narrowest sense) (*position*), the fourth through the sixth are for the coordinates of the direction of view (*direction*), *dist* is for the distance of *position* to the projection plane (*distance*), *ang* is for the angle of rotation around the axis $\overrightarrow{pd}$ where p stands for *position* and d for *direction* (*angle*), and the **char** argument *ax* indicates the axis with which $\overrightarrow{pd}$ is to be aligned, and around which *up* is to be rotated (*axis*).

Log

[LDF 2003.07.04.] Now calling *persp.set_element*( ) instead of accessing the elements of *persp* directly. The latter is no longer possible, because **Focus** is no longer a **friend** of **Transform**.

---

604. Constructor.

⟨ Declare **Focus** functions 602 ⟩ +≡
  Focus(**const real** *pos_x*, **const real** *pos_y*, **const real** *pos_z*, **const real** *dir_x*, **const real** *dir_y*, **const real** *dir_z*, **const real** *dist*, **const real** *ang* = 0, **char** *ax* = 'z');

605.

⟨ Define **Focus** functions 605 ⟩ ≡

```
Focus::Focus(const real pos_x, const real pos_y, const real pos_z, const real dir_x, const real
        dir_y, const real dir_z, const real dist, const real ang, char ax)
  : distance(dist), angle(ang), axis(ax) {
    bool DEBUG = false;    /* true */
    if (DEBUG) cout << "Entering Focus() (7 real arguments).\n" << flush;
    axis = tolower(axis);
    if (axis ≠ 'x' ∧ axis ≠ 'y' ∧ axis ≠ 'z') {
      cerr << "WARNING! In Focus() (7 real arguments):\n" <<
          "axis argument has invalid value: " << axis << ". Using 'z'\n" << flush;
      axis = 'z';
    }
    position.set(pos_x, pos_y, pos_z);
    direction.set(dir_x, dir_y, dir_z);
#if 0
    transform.reset();
      /* [LDF 2002.12.10.] This doesn't seem to be necessary. I believe I added it while debugging. */
#endif
    if (DEBUG) {
      transform.show("transform before alignment.");
    }
    transform.align_with_axis(position, direction, axis);
    if (DEBUG) {
      transform.show("transform after alignment.");
      cout << "Enter <RETURN> to continue.\n\n" << flush;
    }
    Transform unalign_up = transform.inverse();    /* Use the positive y-axis for the "up" direction,
        if axis ≡ 'x' or 'z', and the positive x-axis if axis ≡ 'y'. */
    if (axis ≡ 'z' ∨ axis ≡ 'x') up.set(0, 1, 0);
    else up.set(1, 0, 0);
    if (angle ≠ 0) {
      if (axis ≡ 'z') up *= transform.rotate(0, 0, −angle);
      else if (axis ≡ 'x') up *= transform.rotate(−angle);
      else if (axis ≡ 'y') up *= transform.rotate(0, −angle);
      else {
        cerr << "ERROR! In Focus::Focus():\n" << "This can't happen! \
           axis has invalid value: " << axis << endl << "Rotating around z-axis.\n" <<
           "Enter <RETURN> to try to continue.\n" << flush;
        up *= transform.rotate(0, 0, −angle);
      }
      if (DEBUG) up.show("up after rotation");
    }
    up *= unalign_up;
    up.apply_transform();
    transform.shift(0, 0, −distance);
    persp.set_element(2, 2, 0);
    persp.set_element(2, 3, 1/distance);
    if (DEBUG) cout << "Exiting Focus() (7 real arguments).\n" << flush;
  }
```

See also sections 607, 610, 612, 614, 616, 618, 626, and 629.

This code is used in section 633.

606. Setting function. [LDF 2002.09.17.] Added this function.

⟨ Declare **Focus** functions 602 ⟩ +≡
void *set*(**const real** *pos_x*, **const real** *pos_y*, **const real** *pos_z*, **const real** *dir_x*, **const real** *dir_y*, **const real** *dir_z*, **const real** *dist*, **const real** *ang* = 0.0, **char** *ax* = 'z');

607.

⟨ Define **Focus** functions 605 ⟩ +≡
```
void Focus::set(const real pos_x, const real pos_y, const real pos_z, const real dir_x, const real
        dir_y, const real dir_z, const real dist, const real ang, char ax)
{
  Focus f(pos_x, pos_y, pos_z, dir_x, dir_y, dir_z, dist, ang, ax);
  *this = f;
}
```

608. Point arguments.

609. Constructor.

⟨ Declare **Focus** functions 602 ⟩ +≡
Focus(**const Point** &*pos*, **const Point** &*dir*, **const real** *dist*, **const real** *ang* = 0.0, **char** *ax* = 'z');

610.

⟨ Define **Focus** functions 605 ⟩ +≡
```
Focus::Focus(const Point &pos, const Point &dir, const real dist, const real ang, char ax)
{
  Focus f(pos.get_x(), pos.get_y(), pos.get_z(), dir.get_x(), dir.get_y(), dir.get_z(), dist, ang, ax);
  *this = f;
}
```

611. Setting function. [LDF 2002.09.17.] Added this function.

⟨ Declare **Focus** functions 602 ⟩ +≡
void *set*(**const Point** &*pos*, **const Point** &*dir*, **const real** *dist*, **const real** *ang* = 0.0, **char** *ax* = 'z');

612.

⟨ Define **Focus** functions 605 ⟩ +≡
```
void Focus::set(const Point &pos, const Point &dir, const real dist, const real ang, char ax)
{
  Focus f(pos.get_x(), pos.get_y(), pos.get_z(), dir.get_x(), dir.get_y(), dir.get_z(), dist, ang, ax);
  *this = f;
}
```

613. Assignment. [LDF 2002.10.13.] Added this function. Now using it in all but the first of the non-default constructors. This saves on duplicating code and reduces the probability of bugs that might arise from inconsistencies among the constructors and setting functions.

⟨ Declare **Focus** functions 602 ⟩ +≡
const Focus &**operator**=(**const Focus** &);

614.

⟨ Define **Focus** functions 605 ⟩ +≡

```
const Focus &Focus::operator=(const Focus &f)
{
  if (this ≡ &f)     /* [LDF 2002.10.13.] Prevent self-assignment. */
    return *this;
  position = f.position;
  direction = f.direction;
  up = f.up;
  distance = f.distance;
  angle = f.angle;
  axis = f.axis;
  transform = f.transform;
  persp = f.persp;
  return *this;
}
```

615. Reset angle. [LDF 2002.10.12.] Added this function.

⟨ Declare **Focus** functions 602 ⟩ +≡

```
void reset_angle(const real ang);
```

616.

⟨ Define **Focus** functions 605 ⟩ +≡

```
void Focus::reset_angle(const real ang)
{
  angle = ang;
  transform.reset();
  persp.reset();
  transform.align_with_axis(position, direction, axis);
  Transform unalign_up = transform.inverse();   /* Use the positive y-axis for the "up" direction,
      if axis ≡ 'x' or 'z', and the positive x-axis if axis ≡ 'y'. */
  if (axis ≡ 'z' ∨ axis ≡ 'x') up.set(0, 1, 0);
  else up.set(1, 0, 0);
  if (angle ≠ 0) {
    if (axis ≡ 'z') up *= transform.rotate(0, 0, −angle);
    else if (axis ≡ 'x') up *= transform.rotate(−angle);
    else if (axis ≡ 'y') up *= transform.rotate(0, −angle);
    else {
      cerr ≪ "ERROR!␣In␣Focus::Focus():\n" ≪ "This␣can't␣happen!␣\
          ␣axis␣has␣invalid␣value:␣" ≪ axis ≪ endl ≪ "Rotating␣around␣z-axis.\n" ≪
          "Enter␣<RETURN>␣to␣try␣to␣continue.\n" ≪ flush;
      up *= transform.rotate(0, 0, −angle);
    }
  }
  up *= unalign_up;
  up.apply_transform();
  transform.shift(0, 0, −distance);
  persp.set_element(2, 2, 0);
  persp.set_element(2, 3, 1/distance);
}
```

617. Show.

Log

[LDF 2002.09.17.] Added this function.
[LDF 2003.07.09.] Made the arguments **const**.

⟨ Declare **Focus** functions 602 ⟩ +≡

```
void show(const string text_str = "Focus:", const bool show_transforms = false) const;
```

618.

⟨ Define **Focus** functions 605 ⟩ +≡

```
void Focus::show(const string text_str, const bool show_transforms) const
{
  cout << text_str << endl;
  position.show("position:");
  direction.show("direction:");
  up.show("up:");
  cout << "distance == " << distance << ". " << "axis == " << axis << endl << flush;
  cout << "angle == " << angle << endl << flush;
  if (show_transforms == true) {
    transform.show("transform:");
    persp.show("persp:");
  }
  return;
}
```

619. Returning elements and information. [LDF 2002.09.18.] Added this section. The functions in this section are now necessary, since I've made **Focus** a **class** (it was formerly a **struct**), and the data members **private**.

620. Get position.

Log

[LDF 2002.09.18.] Added this function.

⟨ Declare **Focus** functions 602 ⟩ +≡

```
inline const Point &get_position() const
{
  return position;
}
```

621. Get direction.

Log

[LDF 2003.07.09.] Added this function.

⟨ Declare **Focus** functions 602 ⟩ +≡

```
inline const Point &get_direction() const
{
  return direction;
}
```

622. Get distance. [LDF 2002.09.18.] Added this function.

⟨ Declare **Focus** functions 602 ⟩ +≡

```
inline const real &get_distance() const
{
  return distance;
}
```

623. Get up. [LDF 2002.09.18.] Added this function.

⟨ Declare **Focus** functions 602 ⟩ +≡
inline const Point &*get_up*( ) **const**
{
return *up*;
}

624. Get transform.

Log

[LDF 2002.09.18.] Added this function.

⟨ Declare **Focus** functions 602 ⟩ +≡
inline const Transform &*get_transform*( ) **const**
{
return *transform*;
}

625. Get transform element.

Log

[LDF 2002.09.18.] Added this function.
[LDF 2003.07.04.] Made non-inline. It now calls **Transform** :: *get_element*( ) instead of accessing *transform.matrix* directly. This is no longer possible, because **Focus** is no longer a **friend** of **Transform**.
[LDF 2003.07.09.] Changed the **const unsigned int** arguments to **const unsigned short**.

⟨ Declare **Focus** functions 602 ⟩ +≡
real *get_transform_element*(**const unsigned short** *row*, **const unsigned short** *column*) **const**;

626.

⟨ Define **Focus** functions 605 ⟩ +≡
  real Focus :: *get_transform_element*(**const unsigned short** *row*, **const unsigned short** *column*) **const**
  {
    return *transform.get_element*(*row*, *column*);
  }

627. Get persp.

Log

[LDF 2002.09.18.] Added this function.

⟨ Declare **Focus** functions 602 ⟩ +≡
  inline const Transform &*get_persp*( ) **const**
  {
    return *persp*;
  }

628. Get persp element.

Log

[LDF 2002.09.18.] Added this function.
[LDF 2003.07.04.] Made non-inline. It now calls **Transform** :: *get_element*( ) instead of accessing *transform.matrix* directly. This is no longer possible, because **Focus** is no longer a **friend** of **Transform**.
[LDF 2003.07.09.] Changed the **const unsigned int** arguments to **const unsigned short**.

⟨ Declare **Focus** functions 602 ⟩ +≡
  real *get_persp_element*(**const unsigned short** *row*, **const unsigned short** *column*) **const**;

629.

⟨ Define **Focus** functions 605 ⟩ +≡
  real Focus :: *get_persp_element*(**const unsigned short** *row*, **const unsigned short** *column*) **const**
  {
    return *persp.get_element*(*row*, *column*);
  }

630. Global variables. [LDF 2002.12.08.] !! BUG. Commented-out, because of a bug involving transformations.
Default value. Can be changed.

⟨ Global variables 18 ⟩ +≡
  Focus *default_focus*(0, 10, −10, 0, 10, 0, 10);

631.

⟨ Declarations for the header file 21 ⟩ +≡
extern Focus *default_focus*;

632. Putting Point and Focus together.

633. This is what's compiled.

⟨ Include files 6 ⟩
⟨ Version control identifier 5 ⟩
⟨ Define **class Point** 309 ⟩
⟨ Declare non-member template functions for **Point** 336 ⟩
⟨ Define **Point** constructors 325 ⟩
⟨ Define **class Focus** 600 ⟩
⟨ Define comparison classes 498 ⟩
⟨ Define **static Point** data members 310 ⟩
⟨ Type definitions 15 ⟩
⟨ Global constants 22 ⟩
⟨ Global variables 18 ⟩
⟨ Define **Transform** functions 169 ⟩
⟨ Define **Point** functions 330 ⟩
⟨ Define non-member non-template functions for **Point** 481 ⟩
⟨ Define **bool_point** functions 314 ⟩
⟨ Define **bool_point_quadruple** functions 316 ⟩
⟨ Define **bool_real_point** functions 318 ⟩
⟨ Define **Focus** functions 605 ⟩
⟨ Define **Label** functions 514 ⟩
⟨ Define **Picture** functions 264 ⟩

634. This is what's written to `points.h`.

⟨ `points.h` 634 ⟩ ≡
⟨ Define **class Point** 309 ⟩
⟨ Define **class Focus** 600 ⟩
⟨ Define comparison classes 498 ⟩
⟨ Declare non-member template functions for **Point** 336 ⟩
⟨ Declare non-member non-template functions for **Point** 480 ⟩
⟨ Type definitions 15 ⟩
⟨ Declarations for the header file 21 ⟩

635. Line (`lines.web`). [LDF 2002.10.29.] **Lines** are not **Shapes**. They are used for performing vector operations. A **Line** is defined by a **Point** representing a position vector and a **Point** representing a direction vector.

[LDF 2003.06.03.] TO DO: Add setting functions.

Log

[LDF 2002.04.08.] Added this section. [LDF 2002.04.12.] Created this file. Removed the code for **Line** from `points.web` and put 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.

[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* = "$Id:␣lines.web,v␣1.4␣2004/01/12␣21:30:20␣lfinstol␣Exp␣$";

636. Include files.

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

637. Line struct definition.

format *Line Point*

⟨ Define **struct Line** 637 ⟩ ≡
struct Line {
public: **Point** *position*;
Point *direction*;
⟨ Declare **Line** constructors 639 ⟩
⟨ Declare **Line** functions 643 ⟩
};

This code is used in sections 657 and 658.

638. Constructors. [LDF 2002.10.29.] The constructors and assignment operator take **Point** arguments for *position* and *direction*. If you want to get the **Line** between two **Points**, use **Point** :: *get_line*( ).

Log

[LDF 2002.04.12.] It took me a while to figure out why I was having problems with **Lines**. The constructor was making the opposite assumption, namely, that it was supposed to calculate the **Line** from its arguments, rather than just taking them as they were. This caused a problem in **Plane** :: *intersection_line*( ).

---

639. Default constructor. This constructor takes two optional **Point** arguments. The default for the **Point** arguments is *origin*.

⟨ Declare **Line** constructors 639 ⟩ ≡
Line(**const Point** &*pos* = *origin*, **const Point** &*dir* = *origin*);

See also section 641.

This code is used in section 637.

640.

⟨ Define **Line** constructors 640 ⟩ ≡
Line :: **Line**(**const Point** &*pos*, **const Point** &*dir*)
: *position*(*pos*), *direction*(*dir*) {
position.*apply_transform*( );
direction.*apply_transform*( );
}

See also section 642.

This code is used in section 657.

641. Copy constructor. [LDF 2002.10.29.] Calling *apply_transform*( ) on *position* and *direction* is probably unnecessary, because it will already have been called on *l.position* and *l.direction* when *l* was declared or assigned to. But maybe some function has affected *l.position.transform* or *l.direction.transform*, so I'm doing it just to be sure.

⟨ Declare **Line** constructors 639 ⟩ +≡
Line(**const Line** &*l*);

642.

⟨ Define **Line** constructors 640 ⟩ +≡
Line :: **Line**(**const Line** &*l*)
: *position*(*l.position*), *direction*(*l.direction*) {
position.*apply_transform*( );
direction.*apply_transform*( );
}

643. Assignment.

⟨ Declare **Line** functions 643 ⟩ ≡
void operator=(**const Line** &*l*);

See also sections 646, 648, and 652.

This code is used in section 637.

644.

⟨ Define **Line** functions 644 ⟩ ≡

```
void Line::operator=(const Line &l)
{
  position = l.position;
  direction = l.direction;
}
```

See also sections 649, 650, 651, 653, and 978.

This code is used in sections 657 and 980.

645. Get Line. (**Point** function). LDF Undated. Declared in `points.web`. Must be defined here, because **Line** is an incomplete type there.

[LDF 2003.06.06.] *get_line*( ) returns a **Line** l corresponding to the line from $*this$ to p, where $l.position$ is a **Point** on the **Line**, and $l.direction$ is a direction vector. $l.position$ will be $*this$, and $l.direction$ will be $pt - *this$.

Log

[LDF 2003.06.06.] BUG FIX: Changed the call to **Line**( ), so that the argument for *direction* is $pt - *this$ instead of pt.

⟨ Define **Point** functions 330 ⟩ +≡

```
Line Point::get_line(const Point &pt) const
{
  Line l(*this, (pt - *this));
  return l;
}
```

646. Get Path. [LDF 2003.06.06.] Returns a linear **Path** consisting of two **Points**, and corresponding to the **Line**. Must be defined in `paths.web`, because **Path** is an incomplete type here.

Log

[LDF 2003.06.06.] Added this function.

⟨ Declare **Line** functions 643 ⟩ +≡

```
Path get_path(void) const;
```

647. Intersection. [LDF 2003.06.06.] Commented-out. This function doesn't work. Using a different version, that finds the intersection points of the traces of the lines on two or all of the major axes. TO DO: Fix it!

LDF Undated. Declared in `points.web`, but must be defined here, because **Line** is an incomplete type in `points.web`.

Log

[LDF 2002.04.12.] Moved this function definition here from `points.web` because it requires the use of **Lines**, and **Line** is an incomplete type there.

[LDF 2002.04.15.] Commented-out, because I'm having problems with it. Commented old version in `points.web` back in. I don't quite understand this, because it seemed to be working.

[LDF 2002.04.22.] Changed return value to **bool_point**, to correspond with the old version. This facilitates testing, since all I have to do is to comment-out whichever version I don't want to use, and uncomment-out the other one. Made a few changes in the function definition in order to be able to return a **bool_point**.

⟨ Define **Point** functions 330 ⟩ +≡

```
#if 1    /* 0 */
  bool_point Point::intersection_point(const Point &pp0, const Point &pp1, const Point
        &qq0, const Point &qq1)
  {    /* START HERE. DEBUGGING. [LDF 2003.06.24.] */
    bool DEBUG = false;    /* true */
    if (DEBUG) cout << "Entering Point::intersection_point()\n";
    Line l_p = pp0.get_line(pp1);
    Line l_q = qq0.get_line(qq1);
    if (DEBUG) {
      pp0.show("pp0");
      pp1.show("pp1");
      qq0.show("qq0");
      qq1.show("qq1");
      l_p.show("l_p");
      l_q.show("l_q");
    }
    bool_real_point brp = l_p.get_distance(l_q);
    if (DEBUG) {
      cout << "brp.b == " << brp.b << endl << flush;
      cout << "brp.r == " << brp.r << endl << flush;
      brp.pt.show("brp.pt");
    }
    if (brp.r ≠ 0 ∨ brp.b ≡ false ∨ brp.pt ≡ INVALID_POINT) return INVALID_BOOL_POINT;
    bool_point bp;
    bp.pt = brp.pt;
    bool_real br_p = brp.pt.is_on_segment(pp0, pp1);
    bool_real br_q = brp.pt.is_on_segment(qq0, qq1);
    if (DEBUG) {
      cout << "br_p.first == " << br_p.first << endl << flush;
      cout << "br_p.second == " << br_p.second << endl << flush;
      cout << "br_q.first == " << br_q.first << endl << flush;
      cout << "br_q.second == " << br_q.second << endl << flush;
    }
    bp.b = (br_p.first ∧ br_q.first);
    if (DEBUG) cout << "Exiting Point::intersection_point()\n";
    return bp;
  }
#endif
```

648. Get distance. [LDF 2002.04.22.] Renamed this *get_distance*( ) from *intersection_point*( ). The old version of **Point**::*intersection_point*( ), which I am currently using again, since the new version wasn't working, returns a **bool_point**, which is sensible. If I start using the commented-out version above again, I should have it return a **bool_point** too, instead of a **bool_real_point**. This will make it easier to switch back to the old version, if I have problems again.

[LDF 2003.06.11.] START HERE. TO DO: *get_distance*( ) may be working now, due to changes I've made to **Line** elsewhere. Read this through and see how it works. Then test. Also, check where it's used. [LDF 2003.06.03.] When I've fixed it, add description to `line.texi`.

Log

[LDF 2002.04.10.] Added this function.

```
⟨ Declare Line functions 643 ⟩ +≡
  bool_real_point get_distance(const Line &l) const;
```

649.

```
⟨ Define Line functions 644 ⟩ +≡
  bool_real_point Line::get_distance(const Line &l) const { bool DEBUG = false;    /* true */
      if (DEBUG) cout ≪ "Entering␣Line::get_distance()\n";
      bool_real_point brp;
      Point normal = direction.cross_product(l.direction);
      if (DEBUG) normal.show("normal␣after␣cross␣product.");
      Point normal_unit = normal.unit_vector();
      if (DEBUG) normal_unit.show("normal_unit.");
      if (normal_unit ≡ origin) {
        if (DEBUG) cout ≪ "Lines␣are␣parallel.\n" ≪ flush;
        brp.b = false;    /* No intersection. */
        brp.pt = INVALID_POINT;
        Point temp_pt(l.position);
        temp_pt −= position;
        temp_pt = temp_pt.cross_product(direction);
        brp.r = temp_pt.magnitude()/direction.magnitude();    /* [LDF 2002.10.29.] Distance. */
        if (DEBUG) {
          cout ≪ "distance␣==␣" ≪ brp.r ≪ endl ≪ flush;
        }
        if (DEBUG) cout ≪ "Exiting␣Line::get_distance()\n";
        return brp;
      }
      else {
      if (DEBUG) cout ≪ "Lines␣are␣not␣parallel.\n" ≪ flush;
      brp.r = fabs((l.position − position).dot_product(normal_unit));
      if (DEBUG) cout ≪ "distance␣==␣" ≪ brp.r ≪ endl ≪ flush;
```

650. Lines have an intersection.

Log

[LDF 2003.08.27.] Commented-out the declarations of v_x, v_y, and v_z, since they are not used. I haven't deleted them, in case I need them someday.

```
⟨ Define Line functions 644 ⟩ +≡
  if (brp.r < Point::epsilon()) {
    if (DEBUG) cout << "Lines␣have␣an␣intersection.\n" << flush;
    brp.r = 0;
    brp.b = true;
    real a_x = position.get_x();
    real a_y = position.get_y();
    real a_z = position.get_z();
    real b_x = l.position.get_x();
    real b_y = l.position.get_y();
    real b_z = l.position.get_z();
#if 0
    real v_x = direction.get_x();
    real v_y = direction.get_y();
    real v_z = direction.get_z();
#endif
    real w_x = l.direction.get_x();
    real w_y = l.direction.get_y();
    real w_z = l.direction.get_z();
    real u_x = normal.get_x();
    real u_y = normal.get_y();
    real u_z = normal.get_z();
    real t;
    if (u_z ≠ 0) {
      if (DEBUG) cout << "u_z␣(normal␣z)␣!=␣0\n";
      t = (((b_x − a_x) ∗ w_y) − ((b_y − a_y) ∗ w_x))/u_z;
    }
    else if (u_x ≠ 0) {
      if (DEBUG) cout << "u_x␣(normal␣x)␣!=␣0\n";
      t = (((b_y − a_y) ∗ w_z) − ((b_z − a_z) ∗ w_y))/u_x;
    }
    else if (u_y ≠ 0) {
      if (DEBUG) cout << "u_y␣(normal␣y)␣!=␣0\n";
      t = (((b_z − a_z) ∗ w_x) − ((b_x − a_x) ∗ w_z))/u_y;
    }
    else {
      cerr << "This␣can't␣happen!\n" << "In␣Line::get_distance().\n" <<
          "normal␣==␣origin.␣␣This␣case␣should␣have␣been␣" <<
          "caught␣above,␣so␣something␣is␣really␣wrong.\n" <<
          "Returning␣INVALID_BOOL_REAL_POINT.\n\n" << flush;
      return INVALID_BOOL_REAL_POINT;
    }
    if (DEBUG) cout << "t␣==␣" << t << endl << flush;
    brp.pt = direction;
    brp.pt ∗= t;
    brp.pt += position;
```

```
    if (DEBUG) {
      brp.pt.show("intersection␣point:");
    }
    if (DEBUG) cout ≪ "Exiting␣Line::get_distance()\n";
    return brp;
  }
```

651. Lines are not parallel, but do not intersect.

```
⟨ Define Line functions 644 ⟩ +≡
  else {
    if (DEBUG) cout ≪ "Lines␣are␣not␣parallel,␣but␣do␣not␣intersect.\n" ≪ flush;
    brp.b = false;
    brp.pt = INVALID_POINT;
    if (DEBUG) cout ≪ "Exiting␣Line::get_distance()\n";
    return brp;
  }
  } }
```

652. Show.

```
⟨ Declare Line functions 643 ⟩ +≡
  void show(string text = "");
```

653.

```
⟨ Define Line functions 644 ⟩ +≡
  void Line::show(string text)
  {
    if (text ≡ "") cout ≪ "Line:\n";
    else cout ≪ text ≪ endl;
    position.show("position:");
    direction.show("direction:");
  }
```

654. Global constants for Line.

```
⟨ Line global constants 654 ⟩ ≡
  extern const Line INVALID_LINE(INVALID_POINT, INVALID_POINT);
```

This code is used in section 657.

655.

```
⟨ Declarations for the header file 21 ⟩ +≡
  extern const Line INVALID_LINE;
```

656. Putting Line together.

657. This is what's compiled.

```
  ⟨ Include files 6 ⟩
  ⟨ Version control identifier 5 ⟩
  ⟨ Define struct Line 637 ⟩
  ⟨ Define Line constructors 640 ⟩
  ⟨ Line global constants 654 ⟩
  ⟨ Define Line functions 644 ⟩
  ⟨ Define Point functions 330 ⟩
```

658. This is what's written to `lines.h`.

⟨ `lines.h` 658 ⟩ ≡
  ⟨ Define **struct Line** 637 ⟩
  ⟨ Declarations for the header file 21 ⟩

659. Plane (`planes.web`). [LDF 2002.10.29.] **Planes** are not **Shapes**. They are used for performing vector operations. A **Plane** is defined by a **Point** representing a point on the plane, a **Point** representing the normal to the plane, and the distance of the plane from the origin.

Log

[LDF 2002.04.12.] Created this file. Removed the code for **Plane** from `points.web` and put 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.

[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* = `"$Id:␣planes.web,v␣1.4␣2004/01/12␣21:31:54␣lfinsto1␣Exp␣$"`;

660. Include files.

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

661. Plane struct definition.

  format *Plane Line*

⟨ Define **struct Plane** 661 ⟩ ≡
```
struct Plane {
public: Point normal;
  Point point;
  real distance;
  ⟨ Declare Plane functions 663 ⟩
};
```
This code is used in sections 694 and 695.

662. Constructors.

663. Default constructor. [LDF 2003.06.06.] Creates a degenerate **Plane** with *point*≡*normal*≡*origin*, and *distance*≡0. I could have made the **Plane** be equal to `INVALID_PLANE`, but there's probably no reason for doing so. A **Plane** constructed using this constructor will probably be set using the assignment operator or **Path** :: *get_plane*( ) immediately, or very soon after being declared.

Log

[LDF 2003.06.06.] Added this function.

⟨ Declare **Plane** functions 663 ⟩ ≡
 Plane(**void**);

See also sections 665, 667, 669, 672, 674, 677, 679, 684, 686, 687, and 689.
This code is used in section 661.

664.

⟨ Define **Plane** functions 664 ⟩ ≡
 Plane :: **Plane**(**void**)
 {
  normal = *point* = *origin*;
  distance = 0;
 }

See also sections 666, 668, 670, 673, 675, 678, 680, 685, 688, 690, and 966.
This code is used in sections 694 and 980.

665. Copy constructor.

Log

[LDF 2003.06.06.] Added this function.

⟨ Declare **Plane** functions 663 ⟩ +≡
 Plane(**const Plane** &*p*);

666.

⟨ Define **Plane** functions 664 ⟩ +≡
  Plane :: **Plane**(**const Plane** &*p*)
  {
    ∗*this* = *p*;
    return;
  }

667. Point arguments.

Log

[LDF 2003.06.03.] Changed this function. BUG FIX: *distance* is now calculated, instead of being passed as an argument. *normal* is now made a unit vector.

[LDF 2003.06.06.] Changed, so that if *point* or *normal* is equal to `INVALID_POINT`, the other one is also set to `INVALID_POINT`, and *distance* is set to `INVALID_REAL`.

[LDF 2003.06.06.] Arguments are no longer optional. I've made this change, because I've added a default constructor.

[LDF 2003.06.06.] Added conditional to test for case that *point*≡*normal*. In this case, a warning message is printed to standard error, they are both set to `INVALID_POINT`, and *distance* is set to `INVALID_REAL`.

[LDF 2003.06.24.] BUG FIX: Formerly, `INVALID_PLANE` was returned, if *point* ≡ *normal*. This has been changed, so that `INVALID_PLANE` is returned, if *normal* ≡ *origin*. There is, of course, no reason why *point* shouldn't be equal to *normal*.

⟨ Declare **Plane** functions 663 ⟩ +≡
  Plane(**const Point** &*p*, **const Point** &*n*);

668.

⟨ Define **Plane** functions 664 ⟩ +≡

```
Plane::Plane(const Point &p, const Point &n)
  : normal(n), point(p) {
    point.apply_transform();
    normal.apply_transform();
    if (point ≡ INVALID_POINT) {
      normal = INVALID_POINT;
      distance = INVALID_REAL;
      return;
    }
    else if (normal ≡ INVALID_POINT) {
      point = INVALID_POINT;
      distance = INVALID_REAL;
      return;
    }
    else if (normal ≡ origin) {
      cerr ≪ "WARNING!␣In␣Plane():\nnormal␣==␣origin.␣" ≪ "Plane␣is␣INVALID_PLANE.\n\n" ≪
          flush;
      point = INVALID_POINT;
      distance = INVALID_REAL;
      return;
    }
    normal.unit_vector(true);
    distance = −point.dot_product(normal);
    if (fabs(distance) < Point::epsilon()) distance = 0;
    return;
  }
```

669. Assignment.

Log

[LDF 2003.06.06.] Added this function.

⟨ Declare **Plane** functions 663 ⟩ +≡

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

670.

⟨ Define **Plane** functions 664 ⟩ +≡

```
const Plane &Plane::operator=(const Plane &p)
{
  if (this ≡ &p)      /* Make sure it's not self-assignment. [LDF 2003.06.06.] */
    return *this;
  point = p.point;
  normal = p.normal;
  distance = p.distance;
  return p;
}
```

671. Comparing Planes.

Log

[LDF 2003.06.06.] Added this section.

672. Equality.

Log

[LDF 2003.06.06.] Added this function.

⟨ Declare **Plane** functions 663 ⟩ +≡

```
bool operator≡(const Plane &p) const;
```

673.

⟨ Define **Plane** functions 664 ⟩ +≡

```
bool Plane::operator≡(const Plane &p) const
{
  return ((point ≡ p.point) ∧ (normal ≡ p.normal) ∧ (distance ≡ p.distance));
}
```

674. Inequality.

Log

[LDF 2003.06.06.] Added this function.

⟨ Declare **Plane** functions 663 ⟩ +≡

```
bool operator≠(const Plane &p) const;
```

675.

⟨ Define **Plane** functions 664 ⟩ +≡

```
bool Plane::operator≠(const Plane &p) const
{
  return ¬(operator≡(p));
}
```

676. Get distance.

677. Point argument. [LDF 2003.06.03.] This function returns a **real_short** r, where $r.first$ is the distance of the **Point** from the **Plane**. $r.first$ is always positive. $r.second$ can take on the following values:
0 If the **Point** lies in the **Plane**.
1 If it lies on the side of the **Plane** pointed at by the normal to the **Plane**, considered to be the "outside".
-1 If it lies on the side of the **Plane** *not* pointed at by the normal to the **Plane**, considered to be the "inside".

Log

[LDF 2003.06.03.] Changed the definition of this function. The old definition was incorrect. Also changed return type from **real** to **real_short**.

[LDF 2003.06.04.] BUG FIX: In the case that $r\_fabs <$ **Point** :: *epsilon*( ), now *r_fabs* is set to 0. Previously, r was, which was wrong, because *r_fabs* is returned, not r. Also, I now set *r_fabs* and s to 0 separately, because they are of different types. I don't believe any compiler would have trouble with this, but I think it's cleaner if they are assigned to separately.

⟨ Declare **Plane** functions 663 ⟩ +≡

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

678.

⟨ Define **Plane** functions 664 ⟩ +≡

```
real_short Plane::get_distance(const Point &p) const
{
  real r = (p − point).dot_product(normal);
  real r_fabs = fabs(r);
  signed short s;
  if (r_fabs < Point::epsilon()) {
    r_fabs = 0;
    s = 0;
  }
  else s = static_cast⟨signed short⟩(r/r_fabs);
  return real_short(r_fabs, s);
}
```

679. No argument. [LDF 2003.06.03.] This version of *get_distance*( ) returns the data member *distance* and its sign, i.e., the distance of *origin* to the **Plane**, and which side of the **Plane** it lies on. I'm not using *origin* as the default for an optional **Point** argument, because of problems that may arise, when I implement *user_coordinates* and *view_coordinates*.

Log

[LDF 2003.06.03.] Added this function.

⟨ Declare **Plane** functions 663 ⟩ +≡

```
real_short get_distance(void) const;
```

680.

⟨ Define **Plane** functions 664 ⟩ +≡

```
real_short Plane::get_distance(void) const
{
  real f = fabs(distance);
  signed short s = static_cast⟨signed short⟩(distance/f);
  return real_short(f, s);
}
```

681. Point is on Plane. [LDF 2003.06.04.] This function returns *true*, if the **Point** lies on the **Plane** *p*, otherwise *false*. Declared in `points.web`. Must be defined here, because **Plane** is an incomplete type in that file.

Log

[LDF 2003.06.04.] Added this function.

⟨ Define **Point** functions 330 ⟩ +≡

```
bool Point::is_on_plane(const Plane &p) const
{
  if (p.get_distance(*this).second ≡ 0) return true;
  else return false;
}
```

682. Intersection.

683. Intersection with a line.

684. Point arguments.

Log

[LDF 2003.06.03.] Added this function.

⟨ Declare **Plane** functions 663 ⟩ +≡

```
bool_point intersection_point(const Point &p0, const Point &p1) const;
```

685.

⟨ Define **Plane** functions 664 ⟩ +≡

```
bool_point Plane::intersection_point(const Point &p0, const Point &p1) const
{
  bool_point bp;
  real denominator = (p0 − p1).dot_product(normal);
  if (denominator ≡ 0)    /* !! TO DO: Handle cases: Path is in Plane, and Path is in a parallel
      Plane. [LDF 2003.06.03.]  */
  {
    cerr ≪ "ERROR!␣In␣Plane::intersection_point(const␣Point&,␣const␣Point&,):" ≪
        "denominator␣is␣0.␣␣Can't␣divide.␣" ≪ "Path␣is␣either␣in␣Plane,␣or␣parallel␣to\
        ␣Plane." ≪ endl ≪ "Haven't␣programmed␣these␣cases␣yet." ≪ endl ≪
        "Returning␣INVALID_BOOL_POINT." ≪ endl ≪ endl ≪ flush;
    return INVALID_BOOL_POINT;
  }
  real numerator = p0.dot_product(normal) + distance;
  bp.pt = p0 + ((numerator/denominator) ∗ (p1 − p0));
  bp.b = (bp.pt ≠ INVALID_POINT) ? true : false;
  return bp;
}
```

686. Path argument. [LDF 2003.06.03.] Defined in `paths.web`, because **Path** is an incomplete type in this file.

Log

[LDF 2003.06.03.] Added this function.

⟨ Declare **Plane** functions 663 ⟩ +≡

```
bool_point intersection_point(const Path &p) const;
```

687. Intersection of two Planes. [LDF 2002.10.29.] TO DO: Look up and explain!

Log

[LDF 2003.06.04.] Changed to **const**.

⟨ Declare **Plane** functions 663 ⟩ +≡

```
Line intersection_line(const Plane &pl) const;
```

688.

⟨ Define **Plane** functions 664 ⟩ +≡

```
Line Plane::intersection_line(const Plane &pl) const
{
  bool DEBUG = false;   /* true */
  if (DEBUG) cout << "Entering Plane::intersection_line()\n";
  Point pl_normal(pl.normal);
  Point direction_vector = normal.cross_product(pl_normal);   /* [LDF 2002.10.29.] Needed?? */
  if (DEBUG) direction_vector.show("direction_vector");
  if (direction_vector == origin) {
    cerr << "In Plane::intersection_line().\n" << "Planes are parallel\n";
    if (distance == pl.distance) {
        /* Ellipse::intersection_points(Ellipse &) calls this function to find out whether two Planes
          are coincident, so sometimes we don't want to see these messages. I may decide to add an
          argument bool silent to this function. ?? Add bool silent?? */
      cerr << "Planes are coincident.\n" << "Returning Line with origin as position \n" <<
          "and INVALID_POINT as direction.\n\n" << flush;
      return INVALID_LINE;
    }
    else {
      cerr << "Planes are not coincident.\n" << "Returning INVALID_LINE.\n\n" << flush;
      return INVALID_LINE;
    }
  }   /* Outer if. */
  if (DEBUG) cout << "Planes are not parallel\n" << flush;
      /* At least one of the x, y, or z components of direction_vector must be non-zero, otherwise, this
        function would have exited by now. */
  real x, y, z;
  real d = distance;
  real e = pl.distance;
  real nx = normal.get_x();
  real ny = normal.get_y();
  real nz = normal.get_z();
  real mx = pl.normal.get_x();
  real my = pl.normal.get_y();
  real mz = pl.normal.get_z();
  real vx = direction_vector.get_x();
  real vy = direction_vector.get_y();
  real vz = direction_vector.get_z();
  if (direction_vector.get_x() != 0) {
    x = 0;
    y = -1 * ((d * mz - e * nz)/vx);
    z = (d * my - e * ny)/vx;
  }
  else if (direction_vector.get_y() != 0) {
    x = (d * mz - e * nz)/vy;
    y = 0;
    z = -1 * ((d * mx - e * nx)/vy);
  }
  else {
    x = -1 * ((d * my - e * ny)/vz);
```

```
    y = (d * mx − e * nx)/vz;
    z = 0;
  }
  Point point_on_line(x, y, z);
  if (DEBUG) {
    point_on_line.show("point_on_line:");
    direction_vector.show("direction_vector:");
    getchar();
  }
  if (DEBUG) {
    cout << "Exiting␣Plane::intersection_line()\n";
    getchar();
  }
  return Line(point_on_line, direction_vector);
}
```

689. Show.

Log

[LDF 2003.06.06.] Minor change to the conditional that handles *text*.
[LDF 2003.06.06.] Made *show*( ) **const**.

⟨ Declare **Plane** functions 663 ⟩ +≡

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

690.

⟨ Define **Plane** functions 664 ⟩ +≡

```
void Plane::show(string text) const
{
  if (text ≡ "") text = "Plane:";
  cout << text << endl;
  if (*this ≡ INVALID_PLANE) {
    cout << "INVALID_PLANE.␣Can't␣show." << endl << endl << flush;
    return;
  }
  normal.show("normal:");
  point.show("point:");
  cout << "distance␣==␣" << distance << endl << endl << flush;
}
```

691. Global constants for Plane.

⟨ **Plane** global constants 691 ⟩ ≡

```
extern const Plane INVALID_PLANE(INVALID_POINT, INVALID_POINT);
```

This code is used in section 694.

692.

⟨ Declarations for the header file 21 ⟩ +≡

```
extern const Plane INVALID_PLANE;
```

693. Putting Plane together.

694. This is what's compiled.

⟨ Include files 6 ⟩
⟨ Version control identifier 5 ⟩
⟨ Define **struct Plane** 661 ⟩
⟨ **Plane** global constants 691 ⟩
⟨ Define **Plane** functions 664 ⟩
⟨ Define **Point** functions 330 ⟩

695. This is what's written to `planes.h`.

⟨ `planes.h` 695 ⟩ ≡
⟨ Define **struct Plane** 661 ⟩
⟨ Declarations for the header file 21 ⟩

696. Path (`paths.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.

---

format *Path* *Shape*

⟨ Version control identifier 5 ⟩ +≡
static string *rcs_id* = `"$Id:␣paths.web,v␣1.7␣2004/01/12␣21:30:51␣lfinsto1␣Exp␣$"`;

697. 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"
#include "points.h"
#include "lines.h"
#include "planes.h"
```

698. Path class definition.

Log

[LDF 2002.09.18.] Added *projective_extremes*. It contains the minimum and maximum values for x, y, and z of the **Points** in *points*. It's used in **Picture** :: *output*( ) for surface hiding.

[LDF 2002.4.8.] Added **static** variables for help lines and curves: *help_color*, *help_dash_pattern*, *do_help_lines*. The variables for help lines (or curves) are part of **Path**'s interface and can be set anywhere by anyone.

⟨ Define **class Path** 698 ⟩ ≡
```
class Path : protected Shape {
protected: bool line_switch;
  bool cycle_switch;
  bool on_free_store;
  bool do_output;    /* LDF 2002.09.18. Added. */
  signed short fill_draw_value;    /* Variables for drawing and filling. */
  const Color *draw_color;
  const Color *fill_color;
  string dashed;
```

```
    string pen;
    bool arrow;    /* LDF 2003.01.15. Added. Needed for drawarrow( ). */
    valarray⟨real⟩ projective_extremes;    /* LDF 2002.09.18. Added. */
    vector⟨Point *⟩ points;
    vector⟨string⟩ connectors;
  public: static const Color *help_color;
    static string help_dash_pattern;
    static bool do_help_lines;
    ⟨ Declare Path functions 700 ⟩
  };
```

This code is used in sections 980 and 981.

699. Static member variable definitions.

⟨ Define **static class Path** data members 699 ⟩ ≡

```
  const Color *Path::help_color = &Colors::red;
  string Path::help_dash_pattern = "evenly";
  bool Path::do_help_lines = true;
```

This code is used in section 980.

700. Assignment.

Log

[LDF 2002.10.23.] Now all of the data members of **class Path** are assigned to except for *on_free_store*. This has become necessary because of changes in **Solid**::*output*( ), where temporary **Paths** have to be created in order to sort them.

[LDF 2002.12.18.] Moved here. With the DEC compiler under Compaq Tru64 on the DEC Alpha computer, it worked to have this following the constructors. With the GNU C++ compiler (GCC) under GNU/Linux on the Intel i686 computer, it didn't: The copy constructor used the default assignment operator instead of this function, presumably because this function wasn't known at the time the copy constructor was compiled, *although it had been declared previously*! URGENT: Move assignment operators for the other *classes* before the constructors!

[LDF 2003.04.09.] ?? BUG FIX: Now resizing *projective_extremes*, if after setting it to *p.projective_extremes*, *projective_extremes.size*( ) ≡ 0. This prevents a Memory Fault error at run-time. I don't know why it should be necessary, though, since all of the constructors of **Path** and its derived classes resize *projective_extremes*; at least, I thought they did.

⟨ Declare **Path** functions 700 ⟩ ≡

```
  virtual Path &operator=(const Path &p);
```

See also sections 704, 707, 709, 712, 714, 717, 719, 721, 726, 728, 730, 732, 735, 738, 740, 743, 745, 747, 749, 751, 756, 762, 764, 766, 768, 771, 773, 776, 778, 781, 783, 785, 790, 792, 794, 797, 799, 801, 805, 810, 812, 818, 820, 827, 829, 836, 838, 844, 846, 849, 851, 855, 857, 863, 866, 868, 872, 874, 877, 879, 882, 884, 888, 889, 891, 893, 895, 897, 899, 909, 911, 914, 916, 918, 920, 921, 922, 923, 925, 932, 934, 936, 938, 940, 946, 952, 955, 959, 961, 964, and 976.

This code is used in section 698.

701.

⟨ Define **Path** functions 701 ⟩ ≡

```
Path &Path::operator=(const Path &p){
      if (this ≡ &p)     /* Make sure it's not self-assignment. */
        return *this;
      ⟨ Discard points and connectors 703 ⟩
      line_switch = p.line_switch;
      cycle_switch = p.cycle_switch;
      do_output = p.do_output;
      fill_draw_value = p.fill_draw_value;
      draw_color = p.draw_color;     /* [LDF 2002.10.23.] draw_color and fill_color point to the same
          Color as p.draw_color and p.fill_color. No memory allocation is performed. */
      fill_color = p.fill_color;
      dashed = p.dashed;
      pen = p.pen;
      projective_extremes = p.projective_extremes;     /* LDF 2002.09.18. Added this line. */
      if (projective_extremes.size() ≡ 0)     /* LDF 2003.04.09. Added this conditional. */
        projective_extremes.resize(6, 0);
      for (vector⟨Point *⟩::const_iterator p_iter = p.points.begin(); p_iter ≠ p.points.end(); p_iter++)
          points.push_back ( create_new < Point > (*p_iter) ) ;
      for (vector⟨string⟩::const_iterator c_iter = p.connectors.begin(); c_iter ≠ p.connectors.end();
              c_iter++) {
        connectors.push_back(*c_iter);
      }
      return *this; }
```

See also sections 705, 708, 710, 713, 715, 718, 720, 722, 727, 729, 731, 733, 736, 739, 741, 744, 746, 748, 750, 752, 757, 763, 765, 767, 769, 772, 774, 777, 779, 782, 784, 786, 791, 793, 795, 798, 800, 802, 806, 807, 808, 809, 811, 813, 814, 819, 821, 828, 830, 837, 839, 845, 847, 850, 852, 856, 858, 864, 867, 869, 873, 875, 878, 880, 883, 885, 886, 887, 890, 892, 894, 896, 898, 900, 901, 902, 903, 904, 905, 906, 910, 912, 915, 917, 919, 924, 926, 927, 928, 929, 930, 933, 935, 937, 941, 942, 943, 944, 947, 953, 956, 957, 958, 960, 962, 965, and 977.

This code is used in section 980.

702. Constructors and setting functions. Each constructor taking an argument has a corresponding function for setting an already existing **Path**.

Log

[LDF 2003.04.06.] BUG FIX: Now setting *dashed* = "", *pen* = "", and *arrow* = *false* in all constructors and setting functions. This fixed a problem I was having with **Icosahedron**: One of the **Reg_Polygons** was drawn with an arrow.

---

703. Discard points and connectors. This is useful in the setting functions.

⟨ Discard *points* and *connectors* 703 ⟩ ≡

```
if (points.size() > 0) {
    for (vector⟨Point *⟩::iterator iter = points.begin(); iter ≠ points.end(); ++iter) {
      delete *iter;
    }
    points.clear();
}
if (connectors.size() > 0) connectors.clear();
```

This code is used in sections 701, 710, 715, 720, and 729.

704. Default constructor. No arguments.

⟨ Declare **Path** functions 700 ⟩ +≡
```
Path();
```

705.

⟨ Define **Path** functions 701 ⟩ +≡
```
Path::Path()
{
  bool DEBUG = false;    /* true */
  if (DEBUG) cout << "Entering Path() (default version).\n" << flush;
  on_free_store = false;
  line_switch = false;
  cycle_switch = false;
  fill_draw_value = 0;
  dashed = "";    /* LDF 2003.04.06. Added these three lines. */
  pen = "";
  arrow = false;
  draw_color = 0;
  fill_color = 0;
  do_output = true;    /* LDF 2002.09.18. Added this line. */
  projective_extremes.resize(6,0);    /* LDF 2002.09.18. Added this line. */
  if (DEBUG) cout << "Exiting Path() (default version).\n" << flush;
  return;
}
```

706. Lines. [LDF 2002.10.15.] Lines in this sense are **Paths** containing two **Points** and the connector "--" They should not be confused with the **struct Line**, which is for vector operations (where the word "vector" is used in its mathematical sense).

707. Constructor.

⟨ Declare **Path** functions 700 ⟩ +≡
```
Path(const Point &p0, const Point &p1);
```

708.

⟨ Define **Path** functions 701 ⟩ +≡

```
Path::Path(const Point &p0, const Point &p1){ bool DEBUG = false;   /* true */
    if (DEBUG) cout << "Entering Path() (line version).\n" << flush;
    on_free_store = false;
    line_switch = true;
    cycle_switch = false;
    do_output = true;   /* LDF 2002.09.18. Added this line. */
    projective_extremes.resize(6, 0);   /* LDF 2002.09.18. Added this line. */
    fill_draw_value = 0;
    draw_color = 0;
    fill_color = 0;
    dashed = "";   /* LDF 2003.04.06. Added these three lines. */
    pen = "";
    arrow = false; points.push_back ( create_new < Point > (p0) ) ; points.push_back ( create_new <
        Point > (p1) ) ;
    connectors.push_back("--");
    if (DEBUG) cout << "Exiting Path() (line version).\n" << flush;
    return; }
```

709. Setting function.

⟨ Declare **Path** functions 700 ⟩ +≡

```
void set(const Point &p0, const Point &p1);
```

710.

⟨ Define **Path** functions 701 ⟩ +≡

```
void Path::set(const Point &p0, const Point &p1){ line_switch = true;
    cycle_switch = false;
    do_output = true;   /* LDF 2002.09.18. Added this line. */
    fill_draw_value = 0;
    draw_color = 0;
    fill_color = 0;
    dashed = "";   /* LDF 2003.04.06. Added these three lines. */
    pen = "";
    arrow = false;
    ⟨ Discard points and connectors 703 ⟩
    projective_extremes = 0;   /* LDF 2002.09.18. Added this line. */
    points.push_back ( create_new < Point > (p0) ) ; points.push_back ( create_new < Point > (p1) ) ;
    connectors.push_back("--");
    line_switch = true; }
```

711. Points and one type of connector. This constructor takes a variable number of **Point** ∗ arguments, but only allows one type of connector. The argument list must end with 0. If the order of the named arguments is reversed, the compiler can't resolve certain calls to **Path**( ). It couldn't resolve between **Path**(**bool** *cycle*, **string** *connector* ...) and **Path** (**Point** ∗*first_point_ptr* ...). I don't know why it should have had trouble, though, since pointers to **Points** are not **bools**.

[LDF 2002.4.6.] Probably it couldn't distinguish between a pointer and an **int** on the one hand and a **bool** and an **int** on the other. I hope that **bools** are more efficiently implemented than as *ints*, though!

[LDF 2002.10.29.] ?? I don't know why `cweave` needs instructions to put thin spaces after the "**bool**" in the declaration and definition below. Maybe it's because of the "...".

712. Constructor.

⟨ Declare **Path** functions 700 ⟩ +≡
  Path(**string** *connector*, **bool** *cycle* ...);

713.

⟨ Define **Path** functions 701 ⟩ +≡
  Path :: **Path**(**string** *connector*, **bool** *cycle* ...){ **bool** DEBUG = *false*;   /* *true* */
    if (DEBUG) *cout* ≪ "Entering␣Path()␣(connector,␣cycle,␣...).\n" ≪ *flush*;
    on_free_store = *false*;
    line_switch = *false*;
    cycle_switch = *cycle*;
    connectors.*push_back*(*connector*);
    do_output = *true*;   /* LDF 2002.09.18. Added this line. */
    projective_extremes.*resize*(6, 0);   /* LDF 2002.09.18. Added this line. */
    fill_draw_value = 0;
    draw_color = 0;
    fill_color = 0;
    dashed = "";   /* LDF 2003.04.06. Added these three lines. */
    pen = "";
    arrow = *false*;
    va_list *ap*;   /* For the variable length argument list. */
    va_start(*ap*, *cycle*);
    Point ∗*arg_ptr*; **while** ((*arg_ptr* = *va_arg*(*ap*, **Point** ∗)) ≠ **static_cast**⟨**Point** ∗⟩(0))
      points.*push_back* (*create_new* < **Point** > (*arg_ptr*)) ;
    va_end(*ap*);
    if (DEBUG) *cout* ≪ "Exiting␣Path()␣(connector,␣cycle,␣...).\n" ≪ *flush*;
    return; }

714. Setting function.

⟨ Declare **Path** functions 700 ⟩ +≡
  void *set*(**string** *connector*, **bool** *cycle* ...);

715.

⟨ Define **Path** functions 701 ⟩ +≡
void Path :: *set*(**string** *connector*, **bool** *cycle* ...){ *on_free_store* = *false*;
line_switch = *false*;
cycle_switch = *cycle*;
do_output = *true*; /* LDF 2002.09.18. Added this line. */
⟨ Discard *points* and *connectors* 703 ⟩
fill_draw_value = 0;
draw_color = 0;
fill_color = 0;
dashed = ""; /* LDF 2003.04.06. Added these three lines. */
pen = "";
arrow = *false*;
projective_extremes = 0; /* LDF 2002.09.18. Added this line. */
connectors.push_back(*connector*);
va_list *ap*; /* For the variable length argument list. */
va_start(*ap*, *cycle*);
Point **arg_ptr*; **while** ((*arg_ptr* = *va_arg*(*ap*, **Point** *)) ≠ **static_cast**⟨**Point** *⟩(0))
points.push_back (*create_new* < **Point** > (*arg_ptr*)) ;
va_end(*ap*); }

716. Variable number of Points and connectors. These functions takes a variable number of alternating **Point** * and connector arguments, starting with a **Point** *. The argument list must end with 0. We don't need an argument for whether it's a cycle or not, because if it is, it will have a connector at the end.

Log

[LDF 2002.10.29.] BUG FIX: No longer pushing *first_point_ptr* onto *points*. Copying it instead.

717. Constructor.

⟨ Declare **Path** functions 700 ⟩ +≡
Path (**Point** **first_point_ptr* ...) ;

718.

⟨ Define **Path** functions 701 ⟩ +≡

```
Path::Path ( Point *first_point_ptr ... ) { bool DEBUG = false;   /* true */
    if (DEBUG) cout ≪ "Entering␣Path()␣(Point*␣...).\n" ≪ flush;
    on_free_store = false;
    line_switch = false;
    cycle_switch = false;
    do_output = true;   /* LDF 2002.09.18. Added this line. */
    projective_extremes.resize(6, 0);   /* LDF 2002.09.18. Added this line. */
    fill_draw_value = 0;
    draw_color = 0;
    fill_color = 0;
    dashed = "";   /* LDF 2003.04.06. Added these three lines. */
    pen = "";
    arrow = false; points.push_back ( create_new < Point > (first_point_ptr) ) ;
    va_list ap;   /* For the variable length argument list. */
    va_start(ap, first_point_ptr);
    Point *point_ptr;
    char *connector_ptr;
    string connector_string; while ((connector_ptr = va_arg(ap, char *)) ≠ static_cast⟨char *⟩(0)) {
        connectors.push_back((connector_string = connector_ptr));
    if ((point_ptr = va_arg(ap, Point *)) ≡ static_cast⟨Point *⟩(0)) {
      cycle_switch = true;
      break;
    }
    points.push_back ( create_new < Point > (point_ptr) ) ; } va_end(ap);
    if (DEBUG) cout ≪ "Exiting␣Path()␣(Point*␣...).\n" ≪ flush;
    }
```

719. Setting function.

⟨ Declare **Path** functions 700 ⟩ +≡

```
void set ( Point *first_point_ptr ... ) ;
```

720.

⟨ Define **Path** functions 701 ⟩ +≡

```
void Path::set ( Point *first_point_ptr ... ) { on_free_store = false;
      line_switch = false;
      cycle_switch = false;
      do_output = true;    /* LDF 2002.09.18. Added this line. */
      ⟨ Discard points and connectors 703 ⟩
      projective_extremes = 0;    /* LDF 2002.09.18. Added this line. */
      fill_draw_value = 0;
      draw_color = 0;
      fill_color = 0;
      dashed = "";    /* LDF 2003.04.06. Added these three lines. */
      pen = "";
      arrow = false; points.push_back ( create_new < Point > (first_point_ptr) ) ;
      va_list ap;    /* For the variable length argument list. */
      va_start(ap, first_point_ptr);
      Point *point_ptr;
      char *connector_ptr;
      string connector_string; while ((connector_ptr = va_arg(ap, char *)) ≠ (char *) 0) {
            connectors.push_back((connector_string = connector_ptr));
      if ((point_ptr = va_arg(ap, Point *)) ≡ static_cast⟨Point *⟩(0)) {
         cycle_switch = true;
         break;
      }
      points.push_back ( create_new < Point > (point_ptr) ) ; } va_end(ap); }
```

721. Copy constructor. [LDF 2003.04.06.] ?? !! BUG: Got a memory fault when I tried to use this function. Haven't tested it yet. It worked to use the default constructor and then the assignment operator. Maybe it's not kosher to use "*this = p" in a copy constructor.

Log

[LDF 2002.10.15.] Rewrote this function. The old version caused a memory fault when I tried to use it. I've taken code from the default constructor and the assignment operator and put it here without bothering to see what was causing the problem. Probably the old version didn't account for changes I've made in other places, perhaps in the class definition.

[LDF 2002.11.03.] Rewrote this function. Now just using the assignment operator.

⟨ Declare **Path** functions 700 ⟩ +≡

```
Path(const Path &p);
```

722.

⟨ Define **Path** functions 701 ⟩ +≡

```
Path::Path(const Path &p)
{
  bool DEBUG = false;   /* true */
  if (DEBUG) cout << "Entering␣Path␣copy␣constructor.\n";
  *this = p;
  on_free_store = false;
  if (DEBUG) cout << "Exiting␣Path()␣(copy␣constructor).\n" << flush;
  return;
}
```

723. Pseudo-constructor for dynamic allocation.

724. Pointer argument.

Log

[LDF 2002.10.29.] Added argument **const Path** $*p$. If $p \neq 0$, the new **Path** is assigned to using the values from p.

[LDF 2003.12.30.] Replaced this version of **Path** :: *create_new_path*( ) with a specialization of **template**⟨**class** C⟩ $C*$*create_new*( ).

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

[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.

⟨ Declare non-member template functions for **Path** 724 ⟩ ≡

```
Path *create_new(const Path *p);
```

See also section 725.

This code is used in sections 980 and 981.

725. Reference argument.

Log

[LDF 2002.10.29.] Added this function.

[LDF 2003.12.30.] Replaced this version of **Path** :: *create_new_path*( ) with a specialization of **template**⟨**class** C⟩ $C*$*create_new*( ).

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

⟨ Declare non-member template functions for **Path** 724 ⟩ +≡

```
Path *create_new(const Path &p);
```

726. Destructor.

Log

[LDF 2003.08.27.] Made **virtual**, because GCC with the "`-Wall`" option issued the following warning: "'class Path' has virtual functions but non-virtual destructor".

⟨ Declare **Path** functions 700 ⟩ +≡

```
virtual ~Path();
```

727. !! Make sure to delete anything else that I allocate dynamically!

⟨ Define **Path** functions 701 ⟩ +≡

```
Path::~Path()
{
  bool DEBUG = false;    /* true */
  if (DEBUG) {
    cout << "Entering ~Path().\n" << flush;
    show("Path:");
    getchar();
  }
  for (vector<Point *>::iterator iter = points.begin(); iter != points.end(); iter++) {
    delete *iter;
  }
  points.clear();
    /* [LDF 2002.11.03.] This replaces a while loop in which pop_back() was used to empty points. */
  connectors.clear();    /* LDF 2002.11.03. Added. */
    /* LDF 2002.10.07. Added code for handling draw_color and fill_color. */
  if (draw_color != 0 && draw_color->is_on_free_store() == true) {
    if (DEBUG) cout << "Deleting draw_color\n";
    delete draw_color;
    draw_color = 0;
  }
  else if (DEBUG) {
    cout << "Not deleting draw_color\n";
  }
  if (fill_color != 0 && fill_color->is_on_free_store() == true) {
    if (DEBUG) cout << "Deleting fill_color\n";
    delete fill_color;
    fill_color = 0;
  }
  else if (DEBUG) {
    cout << "Not deleting fill_color\n";
  }
  if (DEBUG) {
    cout << "Exiting ~Path().\n" << flush;
    getchar();
  }
}
```

728. Clear. LDF Undated. This function is needed because it's a pure **virtual** function in **Shape**, and for getting rid of items in **Picture** :: *clear*( ).

[LDF 2002.10.07.] *clear*( ) is needed because it's called on the **Shapes** that are stored in **Pictures**, and I don't know of a way of overloading destructors. That is, in **Picture** :: *clear*( ), the actual types of the **Shapes** are unknown, so I can't call ~**Path**( ), ~**Circle**( ), or other destructors directly. But a named function such as *clear*( ) can serve the same purpose.

Log

[LDF 2002.10.07.] Added code for deallocating the memory allocated for *draw_color* and *fill_color*, if any.

?? I tried calling ~**Path**( ) inside **Path** : *clear*( ), but I got a memory fault. Don't know why. TO DO: Try to find out. However, this isn't urgent.

⟨ Declare **Path** functions 700 ⟩ +≡
```
virtual void clear();
```

729.

⟨ Define **Path** functions 701 ⟩ +≡
```
void Path::clear()
{
  bool DEBUG = false;  /* true */
  if (DEBUG) cout << "Entering Path::clear().\n";
  ⟨ Discard points and connectors 703 ⟩
    /* LDF 2002.10.07. Added code for handling draw_color and fill_color. */
  if (draw_color ≠ 0 ∧ draw_color→is_on_free_store() ≡ true) {
    if (DEBUG) cout << "Deleting draw_color\n";
    delete draw_color;
    draw_color = 0;
  }
  else if (DEBUG) {
    cout << "Not deleting draw_color\n";
  }
  if (fill_color ≠ 0 ∧ fill_color→is_on_free_store() ≡ true) {
    if (DEBUG) cout << "Deleting fill_color\n";
    delete fill_color;
    fill_color = 0;
  }
  else if (DEBUG) {
    cout << "Not deleting fill_color\n";
  }
  if (DEBUG) cout << "Exiting Path::clear().\n";
}
```

730. Get copy.

Log

[LDF 2002.11.03.] Made **virtual**. Changed **dynamic_cast**( ) to **static_cast**( ). This may not work.
[LDF 2003.01.29.] It seems to work. At least, I haven't had any problems with it.

⟨ Declare **Path** functions 700 ⟩ +≡
```
virtual Shape *get_copy() const;
```

731.

⟨ Define **Path** functions 701 ⟩ +≡

```
Shape *Path::get_copy() const{ Path *p = create_new < Path > (0);
    *p = *this;
    return static_cast⟨Shape *⟩(p); }
```

732. Set on free store.

Log

[LDF 2004.01.06.] Made non-inline.

⟨ Declare **Path** functions 700 ⟩ +≡

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

733.

⟨ Define **Path** functions 701 ⟩ +≡

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

734. Setting drawing and filling data.

735. Set fill_draw_value.

⟨ Declare **Path** functions 700 ⟩ +≡

```
virtual void set_fill_draw_value(const signed short s);
```

736.

⟨ Define **Path** functions 701 ⟩ +≡

```
void Path::set_fill_draw_value(const signed short s)
{
  fill_draw_value = s;
  return;
}
```

737. Set draw color.

738. Color version.

⟨ Declare **Path** functions 700 ⟩ +≡

```
virtual void set_draw_color(const Color &c);
```

739.

⟨ Define **Path** functions 701 ⟩ +≡

```
void Path::set_draw_color(const Color &c)
{
  draw_color = &c;
  return;
}
```

740. Color pointer version.

⟨ Declare **Path** functions 700 ⟩ +≡
  virtual void *set_draw_color*(**const Color** $*c$);

741.

⟨ Define **Path** functions 701 ⟩ +≡
  void Path :: *set_draw_color*(**const Color** $*c$)
  {
    if ($draw\_color \neq 0 \wedge draw\_color\rightarrow is\_on\_free\_store() \equiv true$) {
      delete *draw_color*;
    }
    $draw\_color = c$;
    return;
  }

742. Set fill color.

743. Color version.

⟨ Declare **Path** functions 700 ⟩ +≡
  virtual void *set_fill_color*(**const Color** $\&c$);

744.

⟨ Define **Path** functions 701 ⟩ +≡
  void Path :: *set_fill_color*(**const Color** $\&c$)
  {
    $fill\_color = \&c$;
    return;
  }

745. Color pointer version.

⟨ Declare **Path** functions 700 ⟩ +≡
  virtual void *set_fill_color*(**const Color** $*c$);

746.

⟨ Define **Path** functions 701 ⟩ +≡
  void Path :: *set_fill_color*(**const Color** $*c$)
  {
    if ($fill\_color \neq 0 \wedge fill\_color\rightarrow is\_on\_free\_store() \equiv true$) {
      delete *fill_color*;
    }
    $fill\_color = c$;
    return;
  }

747. Set dash pattern.

⟨ Declare **Path** functions 700 ⟩ +≡
  virtual void *set_dash_pattern*(**const string** $s =$ "");

748.

⟨ Define **Path** functions 701 ⟩ +≡
void Path :: *set_dash_pattern* (**const string** *s*)
{
dashed = *s*;
return;
}

749. Set pen.

⟨ Declare **Path** functions 700 ⟩ +≡
virtual void *set_pen* (**const string** *s* = "");

750.

⟨ Define **Path** functions 701 ⟩ +≡
void Path :: *set_pen* (**const string** *s*)
{
pen = *s*;
return;
}

751. Set connectors. [LDF 2003.02.08.] TO DO: Overload with a version taking a **vector**⟨**string**⟩ as its argument. arguments.

Log

[LDF 2003.02.08.] Added this function.

⟨ Declare **Path** functions 700 ⟩ +≡
virtual void *set_connectors* (**const string** *s* = "..");

752.

⟨ Define **Path** functions 701 ⟩ +≡
void Path :: *set_connectors* (**const string** *s*)
{
connectors.*clear*( );
connectors.*push_back*(*s*);
}

753. Transformations. [LDF 2002.11.03.] All of the transformations return a **Transform**, so that the same **Transform** can be applied to multiple objects by chaining expressions.

754. Affine transformations.

755. Rotation.

756. Rotation around the main axes.

⟨ Declare **Path** functions 700 ⟩ +≡
virtual Transform *rotate* (**const real** *x*, **const real** *y* = 0, **const real** *z* = 0);

757.

⟨ Define **Path** functions 701 ⟩ +≡

```
Transform Path::rotate(const real x, const real y, const real z)
{
  Transform t;
  t.rotate(x, y, z);
  return *this *= t;
}
```

758. Rotatation around an arbitrary axis.

759. Transform version. Declared in `transfor.web`. Must be defined here, because **Path** is an incomplete type there.

Log

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

[LDF 2003.07.06.] Changed, so that *is_linear*( ) is used, instead of *get_line_switch*( ).

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

```
Transform Transform::rotate(const Path &p, const real angle)
{
  if (¬p.is_linear()) {
    cerr ≪ "ERROR!␣In␣Transform::rotate(Path,␣real).\n" ≪
        "Path␣is␣not␣linear.␣␣Returning␣INVALID_TRANSFORM.\n\n";
    return INVALID_TRANSFORM;
  }
  Transform t;
  t.rotate(p.get_point(0), p.get_last_point(), angle);
  return (*this *= t);
}
```

760. Point version. Declared in `points.web`. Must be defined here, because **Path** is an incomplete type there.

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 **Point** functions 330 ⟩ +≡

```
Transform Point::rotate(const Path &p, const real angle)
{
  if (¬p.get_line_switch()) {
    cerr ≪ "ERROR!␣In␣Point::rotate(Path,␣real).\n" ≪
        "Path␣is␣not␣a␣line.␣␣Returning␣INVALID_TRANSFORM.\n\n";
    return INVALID_TRANSFORM;
  }
  Point pt0 = p.get_point(0);
  Point pt1 = p.get_point(1);
  return rotate(pt0, pt1, angle);
}
```

761. Path versions.

762. Point arguments.

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 **Path** functions 700 ⟩ +≡

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

763. TO DO: Change this, so that I use **operator**∗=(**Transform**) here and in the other transformation functions.

⟨ Define **Path** functions 701 ⟩ +≡

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

764. Path arguments.

Log

[LDF 2002.4.7.] Added default value for *angle* ≡ 180.
[LDF 2002.11.03.] Got rid of local **Points** *p0* and *p1*.
[LDF 2003.05.02.] Changed name of this function from *rotate_around*( ) to *rotate*( ). This function now overloads *rotate*( ) with three **real** arguments.
[LDF 2003.07.13.] Changed, so that *is_linear*( ) is used instead of checking the return value of *get_line_switch*( ). Also, *get_last_point*( ) passed as the second argument to *rotate*( ), instead of *get_point*(1).

```
⟨ Declare Path functions 700 ⟩ +≡
  Transform rotate(const Path &p, const real angle = 180);
```

765.

```
⟨ Define Path functions 701 ⟩ +≡
  Transform Path::rotate(const Path &p, const real angle)
  {
    if (¬p.is_linear()) {
      cerr ≪ "ERROR!␣In␣Path::rotate(Path,␣real).\n" ≪
          "Path␣is␣not␣a␣line.␣␣Returning␣INVALID_TRANSFORM.\n\n";
      return INVALID_TRANSFORM;
    }
    return rotate(p.get_point(0), p.get_last_point(), angle);
  }
```

766. Scale. [LDF 2002.12.20.] TO DO: Make all of the transformations **virtual**!

```
⟨ Declare Path functions 700 ⟩ +≡
  Transform scale(real x, real y = 1, real z = 1);
```

767.

```
⟨ Define Path functions 701 ⟩ +≡
  Transform Path::scale(real x, real y, real z)
  {
    Transform t;
    t.scale(x, y, z);
    return (*this *= t);
  }
```

768. Shear.

```
⟨ Declare Path functions 700 ⟩ +≡
  Transform shear(real xy, real xz = 0, real yx = 0, real yz = 0, real zx = 0, real zy = 0);
```

769.

⟨ Define **Path** functions 701 ⟩ +≡

```
Transform Path::shear(real xy, real xz, real yx, real yz, real zx, real zy)
{
  bool DEBUG = false;    /* true */
  if (DEBUG) cout << "Entering␣Path::shear().\n";
  Transform t;
  t.shear(xy, xz, yx, yz, zx, zy);
  if (DEBUG) cout << "Exiting␣Path::shear().\n";
  return (*this *= t);
}
```

770. Shift.

771. real arguments.

⟨ Declare **Path** functions 700 ⟩ +≡

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

772.

⟨ Define **Path** functions 701 ⟩ +≡

```
Transform Path::shift(real x, real y, real z)
{
  Transform t;
  t.shift(x, y, z);
  return (*this *= t);
}
```

773. Point argument.

⟨ Declare **Path** functions 700 ⟩ +≡

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

774.

⟨ Define **Path** functions 701 ⟩ +≡

```
Transform Path::shift(const Point &p)
{
  return shift(p.get_x(), p.get_y(), p.get_z());
}
```

775. Shift times.

[LDF 2003.01.19.] *shift_times*( ) returns **void**, because **Path** doesn't have a **Transform** data member, and there's no guarantee that all of the **Points** on *points* will have identical *transforms*.

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

Log

[LDF 2003.01.19.] Added this section.

776. real arguments.

Log

[LDF 2003.01.19.] Added this function.

⟨ Declare **Path** functions 700 ⟩ +≡
```
virtual void shift_times(real x, real y = 1, real z = 1);
```

777.

⟨ Define **Path** functions 701 ⟩ +≡
```
void Path::shift_times(real x, real y, real z)
{
  for (vector<Point *>::iterator iter = points.begin(); iter ≠ points.end(); ++iter)
    (**iter).shift_times(x, y, z);
  return;
}
```

778. Point argument.

Log

[LDF 2003.01.19.] Added this function.

⟨ Declare **Path** functions 700 ⟩ +≡
```
virtual void shift_times(const Point &p);
```

779.

⟨ Define **Path** functions 701 ⟩ +≡
```
void Path::shift_times(const Point &p)
{
  return shift_times(p.get_x(), p.get_y(), p.get_z());
}
```

780. Applying transformations.

781. Multiplying by a Transform.

⟨ Declare **Path** functions 700 ⟩ +≡
```
virtual Transform operator*=(const Transform &t);
```

782.

⟨ Define **Path** functions 701 ⟩ +≡
```
Transform Path::operator*=(const Transform &t)
{
  for (vector<Point *>::iterator iter = points.begin(); iter ≠ points.end(); iter++) (**iter) *= t;
  return t;
}
```

783. Applying transform to points.

⟨ Declare **Path** functions 700 ⟩ +≡
```
virtual void apply_transform();
```

784.

⟨ Define **Path** functions 701 ⟩ +≡

```
void Path::apply_transform()
{
  for (vector<Point *>::iterator iter = points.begin(); iter ≠ points.end(); iter++)
    (**iter).apply_transform();
  return;
}
```

785. Projection. [LDF 2002.12.20.] TO DO: Make this function **virtual**!

⟨ Declare **Path** functions 700 ⟩ +≡

```
bool project(const Focus &f, const unsigned short proj, real factor);
```

786.

⟨ Define **Path** functions 701 ⟩ +≡

```
bool Path::project(const Focus &f, const unsigned short proj, real factor)
{
  for (vector<Point *>::iterator iter = points.begin(); iter ≠ points.end(); iter++) {
    if (¬(**iter).project(f, proj, factor)) {
      cerr ≪ "ERROR!␣In␣Path::project():\n" ≪ "Point::project()␣returned␣false.\n" ≪
          "Returning␣false.\n\n" ≪ flush;
      return false;
    }
  }
  return true;
}
```

787. Functions for lines.

788. Alignment with an axis.

789. For lines.

790. No assignment. (Axis argument only). [LDF 2002.11.03.] This function returns the **Transform** that would transform **Path** such that it would come to lie on the major axis indicated by its argument (by default, the z-axis). It does not actually perform the transformation on the **Path**.

Log

[LDF 2002.11.03.] Changed **char** argument to **const char**.

⟨ Declare **Path** functions 700 ⟩ +≡

```
Transform align_with_axis(const char axis = 'z') const;
```

791.

⟨ Define **Path** functions 701 ⟩ +≡

```
Transform Path::align_with_axis(const char axis) const
{
  Transform t;
  if (¬get_line_switch()) {
    cerr ≪ "ERROR!␣In␣Path::align_with_axis().\n" ≪
        "Path␣is␣not␣a␣line.␣␣Returning␣INVALID_TRANSFORM.\n\n";
    return INVALID_TRANSFORM;
  }
  Point p0(*points[0]);
  Point p1(*points[1]);
  return t.align_with_axis(p0, p1, axis);
}
```

792. With assignment. [LDF 2002.11.03.] This function should never be called with the **bool** argument *assign* ≡ *false*. It won't cause any harm, though, since it will just call the **const** version above.

Log

[LDF 2002.11.03.] Added this function.

[LDF 2003.07.18.] Changed, so that *is_linear*( ) is used, rather than *get_line_switch*( ). Also, changed the way **Transform** *t* is set. The latter change was necessary, because GCC 3.3 couldn't compile this file the way it was before.

⟨ Declare **Path** functions 700 ⟩ +≡

```
Transform align_with_axis(bool assign, const char axis = 'z');
```

793.

⟨ Define **Path** functions 701 ⟩ +≡

```
Transform Path::align_with_axis(bool assign, const char axis)
{
  if (¬is_linear()) {
    cerr ≪ "ERROR!␣In␣Path::align_with_axis().\n" ≪
        "Path␣is␣not␣linear.␣␣Returning␣INVALID_TRANSFORM.\n\n";
    return INVALID_TRANSFORM;
  }
  Transform t;
  t.align_with_axis(get_point(0), get_last_point(), axis);
  if (assign ≡ false) {
    cerr ≪ "WARNING!␣In␣Path::align_with_axis():\n" ≪
        "Don't␣call␣this␣function␣with␣the␣\"assign\"␣" ≪
        "argument␣==␣false.\nIt␣won't␣cause␣any␣harm,␣though.\n" ≪ "Continuing.\n\n" ≪
        flush;
    return t;
  }
  return (*this *= t);
}
```

794. For non-lines. (**Point** and axis arguments). [LDF 2002.11.03.] This function finds the transformation that would align the line segment $\overrightarrow{p_0p_1}$ with the major axis indicated by the *axis* argument, and applies it to **this*. *p0* and *p1* are not changed.

Log

[LDF 2002.11.03.] Changed **Point** arguments to **const Point** & and **char** argument to **const char**.

⟨ Declare **Path** functions 700 ⟩ +≡
```
Transform align_with_axis(const Point &p0, const Point &p1, const char axis);
```

795.

⟨ Define **Path** functions 701 ⟩ +≡
```
Transform Path::align_with_axis(const Point &p0, const Point &p1, const char axis = 'z')
{
  Transform t;
  t.align_with_axis(p0, p1, axis);
  return (*this *= t);
}
```

796. Adding Points to Paths.

797. With assignment.

Log

[LDF 2002.4.6.] Added this function. Currently, it doesn't return a **Path**. If it turns out that it would be useful to return **this*, I can change it.

⟨ Declare **Path** functions 700 ⟩ +≡
```
void operator+=(const Point &pt);
```

798.

⟨ Define **Path** functions 701 ⟩ +≡
```
void Path::operator+=(const Point &pt){ points.push_back ( create_new < Point > (pt) ) ;
    return; }
```

799. Without assignment.

Log

[LDF 2002.4.6.] Added this function.

⟨ Declare **Path** functions 700 ⟩ +≡
```
Path operator+(const Point &pt) const;
```

800.

⟨ Define **Path** functions 701 ⟩ +≡
Path Path :: **operator**+(**const Point** &*pt*) **const**{ **Path** *pa*(∗*this*); *pa*.*points*.*push_back* (*create_new* < **Point** > (*pt*)) ;
return *pa*; }

801. Adding connectors to Paths.

Log

[LDF 2003.02.09.] Added this function.

⟨ Declare **Path** functions 700 ⟩ +≡
void operator+=(**const string** *s*);

802.

⟨ Define **Path** functions 701 ⟩ +≡
void Path :: **operator**+=(**const string** *s*)
{
connectors.*push_back*(*s*);
return;
}

803. Concatenating Paths.

804. Versions using "&".

805. With assignment. This function appends the **Path** argument *pa* to ∗*this*.

Log

[LDF 2002.4.6.] Added this function.
[LDF 2002.11.03.] Made non-**inline**.

⟨ Declare **Path** functions 700 ⟩ +≡
void operator&=(**const Path** &*pa*);

806.

⟨ Define **Path** functions 701 ⟩ +≡
void Path :: **operator**&=(**const Path** &*pa*){
if (*is_cycle*( ) ∨ *pa*.*is_cycle*( )) /∗ Return if either one of the **Paths** is a cycle. ∗/
{
cerr ≪ "ERROR!␣In␣Path::operator&(Path&).\n" ≪
"One␣of␣the␣Paths␣is␣a␣cycle.␣Can't␣concatenate.\n" ≪ "Returning␣*this.\n\n";
return;
}
string *last_connector*;

807. [LDF 2002.4.6.] If there isn't an explicit connector for every pair of **Points** in *this→points*, then we have to fill up *connectors* so that there are enough. Otherwise, the "&" will be at the wrong place. We don't have to worry about the connectors for *pa*.

⟨ Define **Path** functions 701 ⟩ +≡
if (*connectors.size*( ) ≡ 0) *last_connector* = "--";
else *last_connector* = *connectors.back*( );
while (*connectors.size*( ) < *points.size*( ) − 1) *connectors.push_back*(*last_connector*);

808. [LDF 2002.4.6.] If the **Paths** don't touch, they are joined using ".." instead of "&". This mimics the behavior of METAFONT.

Log

[LDF 2002.11.03.] Now using ∗(*points.back*( )) instead of *get_point*(*points.size*( ) − 1).

⟨ Define **Path** functions 701 ⟩ +≡
if (∗(*points.back*( )) ≠ *pa.get_point*(0)) {
cerr ≪ "ERROR!␣In␣Path::operator&(Path&)." ≪ "Paths␣don't␣touch.\n" ≪
"Using␣\"..\"␣to␣join␣them␣instead␣of␣\"&\".\n" ≪ *flush*;
connectors.push_back("..");
}
else *connectors.push_back*("&");
for (**vector**⟨**Point** ∗⟩ :: *const_iterator iter* = *pa.points.begin*( );
/∗ [LDF 2002.4.6.] Copy the **Points** in *pa* and put the copies onto *points*. ∗/
iter ≠ *pa.points.end*( ); *iter*++) *points.push_back* (*create_new* < **Point** > (∗*iter*)) ;

809. [LDF 2002.4.6.] Put the connectors from *pa* onto the new **Path**. Since they're **strings**, and not pointers, we don't have to copy them. I tested this to make sure it's true. I don't know how **strings** are implemented, but they seem to be handled like string literals.

⟨ Define **Path** functions 701 ⟩ +≡
for (**vector**⟨**string**⟩ :: *const_iterator iter* = *pa.connectors.begin*( ); *iter* ≠ *pa.connectors.end*( ); *iter*++) {
connectors.push_back(∗*iter*);
}
return; }

810. Without assignment.

Log

[LDF 2002.4.6.] Added this function. It behaves the way the operator "&" does in METAFONT.

⟨ Declare **Path** functions 700 ⟩ +≡
Path operator&(**const Path** &*pa*) **const**;

811.

⟨ Define **Path** functions 701 ⟩ +≡

```
Path Path::operator&(const Path &pa) const
{
  Path r(*this);
  r &= pa;
  return r;
}
```

812. Appending with a connector argument. [LDF 2002.4.7.] It would not have been possible to specify a connector if I'd defined this function as a binary operator, e.g., **operator**+=( ), so I've made it a named function. It can be useful when, for instance, rotation causes two **Points**, which should be identical, to differ by a small amount, like 1/10,000 in one coordinate. This has actually happened, which is why I've added this function. METAPOST can recover gracefully by using "`..`" instead of "&" to connect the paths, but it issues an error message and stops to wait for a response. Using this function can help to avoid such problems.

Log

[LDF 2002.4.7.] Added this function.

⟨ Declare **Path** functions 700 ⟩ +≡

```
Path append(const Path &pa, string connector = "--", bool assign = true);
```

813.

⟨ Define **Path** functions 701 ⟩ +≡

```
Path Path::append(const Path &pa, string connector, bool assign){ Path r(*this);
    string last_connector;   /* [LDF 2002.4.6.] If there isn't an explicit connector for every pair of
        Points in this→points, then we have to fill up connectors so that there are enough. Otherwise,
        the "&" will be at the wrong place. We don't have to worry about the connectors for pa. */
    if (r.connectors.size() ≡ 0) last_connector = "--";
    else last_connector = r.connectors.back();
    while (r.connectors.size() < r.points.size() − 1) r.connectors.push_back(last_connector);
    r.connectors.push_back(connector);
      /* [LDF 2002.4.6.] Copy the Points in pa and put the copies onto points. */
    for (vector⟨Point *⟩::const_iterator iter = pa.points.begin(); iter ≠ pa.points.end(); iter++)
        r.points.push_back ( create_new < Point > (*iter) ) ;
```

814. Put the connectors from *pa* onto the new **Path**. Since they're **strings**, and not pointers, we don't have to copy them. I tested this to make sure it's true. I don't know how **strings** are implemented, but they seem to be handled like string literals. [LDF 2002.4.6.]

⟨ Define **Path** functions 701 ⟩ +≡

```
for (vector⟨string⟩::const_iterator iter = pa.connectors.begin(); iter ≠ pa.connectors.end(); iter++) {
  r.connectors.push_back(*iter);
}
if (assign ≡ true) *this = r;
return r; }
```

815. Drawing and filling.

816. Draw.

817. Path versions.

818. Normal version.

Log

[LDF 2002.10.07.] Added code for handling *draw_color* and *fill_color*.

⟨ Declare **Path** functions 700 ⟩ +≡

```
virtual void draw(const Color &ddraw_color = *Colors::default_color, const string
    ddashed = "", const string ppen = "", Picture &picture = current_picture, bool aarrow = false)
    const;
```

819.

⟨ Define **Path** functions 701 ⟩ +≡

```
void Path::draw(const Color &ddraw_color, const string ddashed, const string ppen, Picture
        &picture, bool aarrow) const{ bool DEBUG = false;    /* true */
    if (DEBUG) cout << "Entering␣Path::draw().\n" << flush;
    if (points.size() ≡ 0)
      /* LDF 2002.09.27. Added this error handling code. If the Path is empty, don't draw it. */
    {
      cerr << "WARNING!␣In␣Path::draw():\n" << "Path␣doesn't␣contain␣any␣Points.\n" <<
          "Not␣doing␣anything.\n\n" << flush;
      return;
    }
    Path *p = create_new < Path > (*this);
    p→fill_draw_value = DRAW;
    p→arrow = aarrow;
    if (DEBUG)
      cout << "ddraw_color.get_use_name()␣==␣" << ddraw_color.get_use_name() << endl << flush;
    if (ddraw_color.get_use_name() ≡ false) {
    if (DEBUG) cout << "Allocating␣memory␣for␣Color.\n" << flush;
    Color *c = create_new < Color > (0);
    *c = ddraw_color;
    p→draw_color = c; }
    else {
      if (DEBUG) cout << "ddraw_color.get_name()␣==␣" << ddraw_color.get_name() << endl << flush;
      p→draw_color = &ddraw_color;
    }
    p→fill_color = Colors::background_color;
    p→dashed = ddashed;
    p→pen = ppen;
    picture += static_cast⟨Shape *⟩(p);
      /* LDF 2002.11.03. Changed dynamic_cast() to static_cast(). */
    if (DEBUG) cout << "Exiting␣Path::draw().\n" << flush;
    return; }
```

820. Picture argument first. [LDF 2002.09.17.] Added this function. It's convenient for when I want to pass a **Picture** argument. If I want to declare it **inline**, I must define it *within* the declaration of **class Path**. Otherwise, it causes a compiler error. I've decided to declare it non-**inline**, and hope that the compiler will inline it by itself.

⟨ Declare **Path** functions 700 ⟩ +≡
virtual void *draw*(**Picture** &*picture*, **const Color** &*ddraw_color* = ∗**Colors**::*default_color*, **string** *ddashed* = "", **string** *ppen* = "", **bool** *aarrow* = *false*) **const**;

821.

⟨ Define **Path** functions 701 ⟩ +≡
void Path::*draw*(**Picture** &*picture*, **const Color** &*ddraw_color*, **string** *ddashed*, **string** *ppen*, **bool** *aarrow*) **const**
{
draw(*ddraw_color*, *ddashed*, *ppen*, *picture*, *aarrow*);
}

822. Point versions. Declared in `points.web`, but must be defined here, because **Path** is an incomplete type here.

[LDF 2002.11.03.] Unlike the **Path** versions of *draw*( ), this function returns a **Path**. This can be useful, if you want to use the **Path** that it creates again for something else.

823. Normal version.

⟨ Define **Point** functions 330 ⟩ +≡
Path Point::*draw*(**const Point** &*p*, **const Color** &*ddraw_color*, **string** *ddashed*, **string** *ppen*, **Picture** &*picture*, **bool** *aarrow*) **const**
{
Path *pa*(∗*this*, *p*);
pa.*draw*(*ddraw_color*, *ddashed*, *ppen*, *picture*, *aarrow*);
return *pa*;
}

824. Picture argument first. [LDF 2002.09.17.] Added this function. It's convenient for when I want to pass a **Picture** argument.

⟨ Define **Point** functions 330 ⟩ +≡
Path Point::*draw*(**Picture** &*picture*, **const Point** &*p*, **const Color** &*ddraw_color*, **string** *ddashed*, **string** *ppen*, **bool** *aarrow*)
{
return *draw*(*p*, *ddraw_color*, *ddashed*, *ppen*, *picture*, *aarrow*);
}

825. Draw arrow.

826. Path versions.

827. Normal version.

Log

[LDF 2003.01.15.] Added this function.

⟨ Declare **Path** functions 700 ⟩ +≡
virtual void *drawarrow*(**const Color** &*ddraw_color* = ∗**Colors** :: *default_color*, **string** *ddashed* = "", **string** *ppen* = "", **Picture** &*picture* = *current_picture*) **const**;

828.

⟨ Define **Path** functions 701 ⟩ +≡
void Path :: *drawarrow*(**const Color** &*ddraw_color*, **string** *ddashed*, **string** *ppen*, **Picture** &*picture*) **const**
{
draw(*ddraw_color*, *ddashed*, *ppen*, *picture*, *true*);
}

829. Picture argument first.

Log

[LDF 2003.01.15.] Added this function.

⟨ Declare **Path** functions 700 ⟩ +≡
virtual void *drawarrow*(**Picture** &*picture*, **const Color** &*ddraw_color* = ∗**Colors** :: *default_color*, **string** *ddashed* = "", **string** *ppen* = "") **const**;

830.

⟨ Define **Path** functions 701 ⟩ +≡
void Path :: *drawarrow*(**Picture** &*picture*, **const Color** &*ddraw_color*, **string** *ddashed*, **string** *ppen*) **const**
{
draw(*picture*, *ddraw_color*, *ddashed*, *ppen*, *true*);
}

831. Point versions.

832. Normal version.

Log

[LDF 2003.01.15.] Added this function.
[LDF 2003.06.03.] Made *drawarrow*( ) **const**.

⟨ Define **Point** functions 330 ⟩ +≡
Path Point :: *drawarrow*(**const Point** &*p*, **const Color** &*ddraw_color*, **string** *ddashed*, **string** *ppen*, **Picture** &*picture*) **const**
{
Path *pa*(∗*this*, *p*);
pa.*drawarrow*(*ddraw_color*, *ddashed*, *ppen*, *picture*);
return *pa*;
}

833. Picture argument first.

Log

[LDF 2003.01.15.] Added this function.
[LDF 2003.06.03.] Made *drawarrow*( ) **const**.

⟨ Define **Point** functions 330 ⟩ +≡
```
Path Point::drawarrow(Picture &picture, const Point &p, const Color &ddraw_color, string
        ddashed, string ppen) const
{
  return drawarrow(p, ddraw_color, ddashed, ppen, picture);
}
```

834. Draw help.

Log

[LDF 2002.05.10.] Changed the way the default arguments are handled. The way it was didn't work for both versions, i.e., the **Path** version and the **Point** version.

[LDF 2002.4.8.] Added this section. !! It would be nice to do something to make sure that the help lines and curves are not drawn over by filling commands. Maybe it will be possible to take care of this when I implement the hidden surface algorithm in *output*( ). [LDF 2002.11.03.] I could have help lines outputted last, if I put them on a **vector** of their own.

[LDF 2003.07.13.] Made all versions of *draw_help*( ) **const**.

---

835. Path versions. [LDF 2002.12.20.] ?? Could these functions be **const**?

836. Normal version. [LDF 2002.4.8.] Added this function.

⟨ Declare **Path** functions 700 ⟩ +≡
```
void draw_help(const Color &ddraw_color = *help_color, string ddashed = help_dash_pattern, string
    ppen = "", Picture &picture = current_picture) const;
```

837.

⟨ Define **Path** functions 701 ⟩ +≡
```
void Path::draw_help(const Color &ddraw_color, string ddashed, string ppen, Picture &picture)
        const
{
  if (do_help_lines ≡ false) return;
  draw(ddraw_color, ddashed, ppen, picture);
}
```

838. Picture argument first.

⟨ Declare **Path** functions 700 ⟩ +≡
```
void draw_help(Picture &picture, const Color &ddraw_color = *help_color, string
    ddashed = help_dash_pattern, string ppen = "") const;
```

839.

⟨ Define **Path** functions 701 ⟩ +≡

void Path :: *draw_help* (**Picture** &*picture*, **const Color** &*ddraw_color*, **string** *ddashed*, **string** *ppen*) **const**
{
draw_help (*ddraw_color*, *ddashed*, *ppen*, *picture*);
}

840. Point versions.

841. Normal version. [LDF 2002.4.8.] Added this function. Declared in `points.web`, but must be defined here, because **Path** is an incomplete type here.

⟨ Define **Point** functions 330 ⟩ +≡

Path Point :: *draw_help* (**const Point** &*pt*, **const Color** &*ddraw_color*, **string** *ddashed*, **string** *ppen*, **Picture** &*picture*) **const**
{
Path *pa* (∗*this*, *pt*);
pa.*draw_help* (*ddraw_color*, *ddashed*, *ppen*, *picture*);
return *pa*;
}

842. Picture argument first. [LDF 2002.09.17.] Added this function. It's convenient for when I want to pass a **Picture** argument.

[LDF 2002.10.26.] Declared in `points.web`. Must be defined here, because **Path** is an incomplete type there.

⟨ Define **Point** functions 330 ⟩ +≡

Path Point :: *draw_help* (**Picture** &*picture*, **const Point** &*pt*, **const Color** &*ddraw_color*, **string** *ddashed*, **string** *ppen*) **const**
{
return *draw_help* (*pt*, *ddraw_color*, *ddashed*, *ppen*, *picture*);
}

843. Fill.

!! [LDF 2003.02.02.] Filling doesn't use a pen!! Change everywhere!!.

844. Normal version.

Log

[LDF 2002.10.07.] Added code for handling and *fill_color*.

⟨ Declare **Path** functions 700 ⟩ +≡

void *fill* (**const Color** &*ffill_color* = ∗**Colors** :: *default_color*, **Picture** &*picture* = *current_picture*) **const**;

845.

⟨ Define **Path** functions 701 ⟩ +≡

```
void Path::fill(const Color &ffill_color, Picture &picture) const { bool DEBUG = false;
      /* true */
    if (DEBUG) cout ≪ "Entering␣Path::fill().\n" ≪ flush;
    if (points.size() ≡ 0)
      /* LDF 2002.09.27. Added this error handling code. If the Path is empty, don't fill it. */
    {
      cerr ≪ "WARNING!␣In␣Path::fill():\n" ≪ "Path␣doesn't␣contain␣any␣Points.\n" ≪
          "Not␣doing␣anything.\n\n" ≪ flush;
      return;
    }
    Path *p = create_new < Path > (0);
    *p = *this;
    p→fill_draw_value = FILL;
    if (DEBUG)
      cout ≪ "ffill_color.get_use_name()␣==␣" ≪ ffill_color.get_use_name() ≪ endl ≪ flush;
    if (ffill_color.get_use_name() ≡ false) {
    if (DEBUG) cout ≪ "Allocating␣memory␣for␣Color.\n" ≪ flush;
    Color *c = create_new < Color > (0);
    *c = ffill_color;
    p→fill_color = c; }
    else {
      if (DEBUG) cout ≪ "ffill_color.get_name()␣==␣" ≪ ffill_color.get_name() ≪ endl ≪ flush;
      p→fill_color = &ffill_color;
    }
    p→pen = "";
    p→dashed = "";
    p→draw_color = Colors::background_color;
    picture += static_cast⟨Shape *⟩(p);
    if (DEBUG) cout ≪ "Exiting␣Path::fill().\n" ≪ flush;
    return; }
```

846. Picture argument first. [LDF 2002.09.17.] Added this function. It's convenient for when I want to pass a **Picture** argument.

⟨ Declare **Path** functions 700 ⟩ +≡

```
void fill(Picture &picture, const Color &ffill_color = *Colors::default_color);
```

847.

⟨ Define **Path** functions 701 ⟩ +≡
  void Path :: *fill*(**Picture** &*picture*, **const Color** &*ffill_color*)
  {
    fill(*ffill_color*, *picture*);
  }

848. Filldraw.

[LDF 2002.03.25] ([LDF 2002.11.03.] Revised the following text). At the present time, *filldraw*( ) differs from the **filldraw** command in METAFONT and METAPOST: In the default case, the outline is drawn in the default color (currently black) and the **Path** is filled with the background color (currently white by default). This makes it possible to hide objects that are behind the **Path** by using the painter's algorithm when rendering. If you want a **Path** to be filled with another color, you will have to use explicit arguments for *ddraw_color* and *ffill_color*. Either or both of these can be "", which causes the default color (currently black) to be used. Of course, plain *fill*( ) followed by plain *draw*( ) will produce the same result.

Log

[LDF 2003.07.16.] Made both versions **const**.

---

849. Normal version.

Log

[LDF 2002.10.07.] Added code for handling *draw_color* and *fill_color*.

⟨ Declare **Path** functions 700 ⟩ +≡
  void *filldraw*(**const Color** &*ddraw_color* = ∗**Colors** :: *default_color*, **const Color** &*ffill_color* = ∗**Colors** :: *background_color*, **string** *ddashed* = "", **string** *ppen* = "", **Picture** &*picture* = *current_picture*) **const**;

850.

⟨ Define **Path** functions 701 ⟩ +≡

```
void Path::filldraw(const Color &ddraw_color, const Color &ffill_color, string ddashed, string
        ppen, Picture &picture) const { bool DEBUG = false;   /* true */
    if (DEBUG) cout << "Entering Path::filldraw().\n" << flush;
    if (points.size() == 0)
      /* LDF 2002.09.27. Added this error handling code. If the Path is empty, don't filldraw it. */
    {
      cerr << "WARNING! In Path::filldraw():\n" << "Path doesn't contain any Points.\n" <<
          "Not doing anything.\n\n" << flush;
      return;
    }
    Path *p = create_new < Path > (0);
    *p = *this;
    p->fill_draw_value = FILLDRAW;
    if (DEBUG)
      cout << "ddraw_color.get_use_name() == " << ddraw_color.get_use_name() << endl << flush;
    if (DEBUG)
      cout << "ffill_color.get_use_name() == " << ffill_color.get_use_name() << endl << flush;
    if (ddraw_color.get_use_name() == false) {
    if (DEBUG) cout << "Allocating memory for Color.\n" << flush;
    Color *c = create_new < Color > (0);
    *c = ddraw_color;
    p->draw_color = c; }
    else {
      if (DEBUG) cout << "ddraw_color.get_name() == " << ddraw_color.get_name() << endl << flush;
      p->draw_color = &ddraw_color;
    }
    if (ffill_color.get_use_name() == false) {
    if (DEBUG) cout << "Allocating memory for Color.\n" << flush;
    Color *c = create_new < Color > (0);
    *c = ffill_color;
    p->fill_color = c; }
    else {
      if (DEBUG) cout << "ffill_color.get_name() == " << ffill_color.get_name() << endl << flush;
      p->fill_color = &ffill_color;
    }
    p->dashed = ddashed;
    p->pen = ppen;
    picture += static_cast<Shape *>(p);
    if (DEBUG) cout << "Exiting Path::filldraw().\n" << flush;
    return; }
```

851. Picture argument first. [LDF 2002.09.17.] Added this function. It's convenient for when I want to pass a **Picture** argument.

⟨ Declare **Path** functions 700 ⟩ +≡

```
void filldraw(Picture &picture, const Color &ddraw_color = *Colors::default_color, const Color
    &ffill_color = *Colors::background_color, string ddashed = "", string ppen = "") const;
```

852.

⟨ Define **Path** functions 701 ⟩ +≡

```
void Path::filldraw(Picture &picture, const Color &ddraw_color, const Color &ffill_color, string
        ddashed, string ppen) const
{
  filldraw(ddraw_color, ffill_color, ddashed, ppen, picture);
}
```

853. Undraw.

854. Path versions.

855. Normal version.

⟨ Declare **Path** functions 700 ⟩ +≡

```
void undraw(string ddashed = "", string ppen = "", Picture &picture = current_picture);
```

856.

⟨ Define **Path** functions 701 ⟩ +≡

```
void Path::undraw(string ddashed, string ppen, Picture &picture){
    if (points.size() ≡ 0)
      /* LDF 2002.09.27. Added this error handling code. If the Path is empty, don't undraw it. */
    {
      cerr ≪ "WARNING!␣In␣Path::undraw():\n" ≪ "Path␣doesn't␣contain␣any␣Points.\n" ≪
          "Not␣doing␣anything.\n\n" ≪ flush;
      return;
    }
    Path *p = create_new < Path > (this);
    p→fill_draw_value = UNDRAW;
    p→draw_color = 0;
    p→fill_color = 0;
    p→dashed = ddashed;
    p→pen = ppen;
    picture += static_cast⟨Shape *⟩(p); }
```

857. Picture argument first. [LDF 2002.09.17.] Added this function. It's convenient for when I want to pass a **Picture** argument.

⟨ Declare **Path** functions 700 ⟩ +≡

```
void undraw(Picture &picture, string ddashed = "", string ppen = "");
```

858.

⟨ Define **Path** functions 701 ⟩ +≡

```
void Path::undraw(Picture &picture, string ddashed, string ppen)
{
  undraw(ddashed, ppen, picture);
}
```

859. Point versions.

860. Normal version. This function is declared in `points.web`, but must be defined here, because **Path** 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**.

⟨ Define **Point** functions 330 ⟩ +≡
Path Point :: *undraw*(**const Point** &*pt*, **string** *ddashed*, **string** *ppen*, **Picture** &*picture*)
{
  Path *pa*(∗*this*, *pt*);
  pa.*undraw*(*ddashed*, *ppen*, *picture*);
  return *pa*;
}

861. Picture argument first.

Log

[LDF 2002.09.17.] Added this function. It's convenient for when I want to pass a **Picture** argument.
[LDF 2002.09.17.] Added this function. It's convenient for when I want to pass a **Picture** argument.

⟨ Define **Point** functions 330 ⟩ +≡
Path Point :: *undraw*(**Picture** &*picture*, **const Point** &*pt*, **string** *ddashed*, **string** *ppen*)
{
  return *undraw*(*pt*, *ddashed*, *ppen*, *picture*);
}

862. Unfill.

863. Normal version.

⟨ Declare **Path** functions 700 ⟩ +≡
void *unfill*(**Picture** &*picture* = *current_picture*);

864.

⟨ Define **Path** functions 701 ⟩ +≡

```
void Path::unfill(Picture &picture){
    if (points.size() ≡ 0)
      /* LDF 2002.09.27. Added this error handling code. If the Path is empty, don't unfill it. */
    {
      cerr ≪ "WARNING!␣In␣Path::unfill():\n" ≪ "Path␣doesn't␣contain␣any␣Points.\n" ≪
          "Not␣doing␣anything.\n\n" ≪ flush;
      return;
    }
    Path *p = create_new < Path > (this);
    p→fill_draw_value = UNFILL;
    p→draw_color = 0;
    p→fill_color = 0;
    p→dashed = "";
    p→pen = "";
    picture += static_cast⟨Shape *⟩(p); }
```

865. Unfilldraw.

866. Normal version.

⟨ Declare **Path** functions 700 ⟩ +≡

```
void unfilldraw(const Color &ddraw_color = *Colors::background_color, string ddashed = "", string
    ppen = "", Picture &picture = current_picture);
```

867.

⟨ Define **Path** functions 701 ⟩ +≡

```
void Path::unfilldraw(const Color &ddraw_color, string ddashed, string ppen, Picture &picture){
      bool DEBUG = false;    /* true */
   if (DEBUG) cout ≪ "Entering␣Path::unfilldraw().\n" ≪ flush;
   if (points.size() ≡ 0)
      /* LDF 2002.09.27. Added this error handling code. If the Path is empty, don't unfilldraw it. */
   {
      cerr ≪ "WARNING!␣In␣Path::unfilldraw():\n" ≪ "Path␣doesn't␣contain␣any␣Points.\n" ≪
          "Not␣doing␣anything.\n\n" ≪ flush;
      return;
   }
   Path *p = create_new < Path > (this);
   p→fill_draw_value = UNFILLDRAW;
      /* LDF 2002.10.07. Added code for handling draw_color and fill_color. Will get rid of this if I do
         actually change it to make it act more like unfilldraw in METAPOST. */
   if (ddraw_color.get_use_name() ≡ false) {
   if (DEBUG) cout ≪ "Allocating␣memory␣for␣Color.\n" ≪ flush;
   Color *c = create_new < Color > (0);
   *c = ddraw_color;
   p→draw_color = c; }
   else {
      if (DEBUG) cout ≪ "ddraw_color.get_name()␣==␣" ≪ ddraw_color.get_name() ≪ endl ≪ flush;
      p→draw_color = &ddraw_color;
   }
   p→fill_color = 0;
   p→dashed = ddashed;
   p→pen = ppen;
   picture += static_cast⟨Shape *⟩(p);
   if (DEBUG) cout ≪ "Exiting␣Path::unfilldraw().\n" ≪ flush;
   return; }
```

868. Picture argument first. [LDF 2002.09.17.] Added this function. It's convenient for when I want to pass a **Picture** argument.

⟨ Declare **Path** functions 700 ⟩ +≡

```
void unfilldraw(Picture &picture, const Color &ddraw_color = *Colors::background_color, string
      ddashed = "", string ppen = "");
```

869.

⟨ Define **Path** functions 701 ⟩ +≡
void Path :: *unfilldraw*(**Picture** &*picture*, **const Color** &*ddraw_color*, **string** *ddashed*, **string** *ppen*)
{
unfilldraw(*ddraw_color*, *ddashed*, *ppen*, *picture*);
}

870. Labelling.

871. Label.

Log

[LDF 2002.03.25.] Added argument *dot* and changed definition of *dotlabel*( ) below so that it just calls *label*( ).

[LDF 2003.04.01.] BUG FIX: Got rid of the first argument **unsigned int** *i*, and made the third argument **short** *text_short* the first argument. Formerly, the **Points** in **Paths** were always numbered starting from 0, because the argument *text_short* was passed to **Point** :: *label*( ), not *i*. Also changed the following versions of *label*( ) and *dotlabel*( ), that call this function.

872. Normal version.

Log

[LDF 2003.05.06.] Changed the conditional, where *text_short* is compared with `WORLD_VALUES`, `PROJ_VALUES`, etc. I had to change it, because I've added `WORLD_VALUES_X_Y`, etc. Now, the conditional tests for `VIEW_VALUES_X_Y`≤*text_short*≤`WORLD_VALUES`. Of course, this makes an assumption about the values that are used to signal that coordinate values should be used for the label, but I think it's worth it, to avoid testing *text_short* against each value individually.

[LDF 2003.07.09.] Made *position_string* and *dot* arguments **const**.

⟨ Declare **Path** functions 700 ⟩ +≡
void *label*(**short** *text_short* = 0, **const string** *position_string* = "top", **const bool** *dot* = *false*, **Picture** &*picture* = *current_picture*) **const**;

873.

⟨ Define **Path** functions 701 ⟩ +≡

```
void Path::label(short text_short, const string position_string, const bool dot, Picture &picture)
        const
{
  bool DEBUG = false;    /* true */
  if (DEBUG) cout << "Entering Path::label()" << "\n" << flush;
  if (Label::DO_LABELS == false) return;
  if (points.size() == 0)
    /* LDF 2002.09.27. Added this error handling code. If the Path is empty, don't label it. */
  {
    cerr << "WARNING! In Path::label():\n" << "Path doesn't contain any Points.\n" <<
        "Not doing anything.\n\n" << flush;
    return;
  }
  for (vector<Point *>::const_iterator iter = points.begin(); iter != points.end(); iter++) {
    (**iter).label(text_short, position_string, dot, picture);
    if (text_short <= Point::WORLD_VALUES && text_short >= Point::VIEW_VALUES)
        /* LDF 2003.05.06. Changed this conditional. */
      ;    /* Do nothing. */
    else ++text_short;
  }
  if (DEBUG) cout << "Exiting Path::label()" << "\n" << flush;
}
```

874. Picture argument first. [LDF 2002.09.17.] Added this function. It's convenient for when I want to pass a **Picture** argument.

[LDF 2003.07.09.] Made *position_string* and *dot* arguments **const**.

Log

⟨ Declare **Path** functions 700 ⟩ +≡

```
void label(Picture &picture, short text_short = 0, const string position_string = "top", const bool
    dot = false) const;
```

875.

⟨ Define **Path** functions 701 ⟩ +≡

```
void Path::label(Picture &picture, short text_short, const string position_string, const bool dot)
        const
{
  label(text_short, position_string, dot, picture);
}
```

876. Dotlabel.

877. Normal version.

Log

[LDF 2003.07.09.] Made *text_short* and *position_string* arguments **const**.

⟨ Declare **Path** functions 700 ⟩ +≡
void *dotlabel*(**const short** *text_short* = 0, **const string** *position_string* = "top", **Picture** &*picture* = *current_picture*) **const**;

878.

⟨ Define **Path** functions 701 ⟩ +≡
void Path :: *dotlabel*(**const short** *text_short*, **const string** *position_string*, **Picture** &*picture*) **const**
{
label(*text_short*, *position_string*, *true*, *picture*);
}

879. Picture argument first.

Log

[LDF 2002.09.17.] Added this function. It's convenient for when I want to pass a **Picture** argument.
[LDF 2003.07.09.] Made *text_short* and *position_string* arguments **const**.

⟨ Declare **Path** functions 700 ⟩ +≡
void *dotlabel*(**Picture** &*picture*, **const short** *text_short* = 0, **const string** *position_string* = "top") **const**;

880.

⟨ Define **Path** functions 701 ⟩ +≡
void Path :: *dotlabel*(**Picture** &*picture*, **const short** *text_short*, **const string** *position_string*) **const**
{
dotlabel(*text_short*, *position_string*, *picture*);
}

881. Outputting.

882. Extract. This is needed for outputting a **Picture**.
[LDF 2003.01.31.] ?? Do I need to call **Point** :: *project*( ) on the **Points** here and in **Path** :: *project*( )?

Log

[LDF 2002.09.17.] Added **const Focus** &*f* argument and error handling code. Now, if any of the **Points** on **vector**⟨**Point** ∗⟩ *points* cannot be projected onto the projection plane using the **Focus** *f*, the **Path** is not put onto the **vector**⟨**Shape** ∗⟩ **Picture** :: *elements*, and consequently never reaches **Picture** :: *output*( ) and **Path** :: *output*( ).
[LDF 2003.05.09.] Rewrote this function. It now calls **Point** :: *extract*( ) instead of calling *apply_transform*( ) and *project*( ) on the **Points** directly. This makes much more sense, since any changes to **Point** :: *extract*( ) would otherwise not have been applied to **Points** on **Paths**.

⟨ Declare **Path** functions 700 ⟩ +≡
vector⟨**Shape** ∗⟩ *extract*(**const Focus** &*f*, **const unsigned short** *proj*, **real** *factor*);

883.

⟨ Define **Path** functions 701 ⟩ +≡

```
vector⟨Shape *⟩ Path::extract(const Focus &f, const unsigned short proj, real factor)
{
  bool DEBUG = false;   /* true */
  vector⟨Shape *⟩ v;
  int i = 0;
  for (vector⟨Point *⟩::iterator iter = points.begin(); iter ≠ points.end(); iter++) {
    if (DEBUG) cout ≪ "Point␣" ≪ i++ ≪ ":" ≪ endl;
    v = (**iter).extract(f, proj, factor);
    if (DEBUG) cout ≪ endl;
    if (v.size() ≡ 0)   /* Point::extract() failed. LDF 2003.05.09. */
      return v;
  }
  vector⟨Shape *⟩ r;
  r.push_back(this);
  return r;
}
```

884. Set extremes. [LDF 2002.09.18.] *set_extremes*( ) doesn't check that the *projective_coordinates* in all of the **Points** on the **Path** are valid. This is done already in *project*( ) and *extract*( ), so I don't think it's necessary to repeat it here, since *extract*( ) (which invokes *project*( )), is called before *set_extremes*( ) in **Picture**::*output*( ). The latter is the only place where *set_extremes*( ) is invoked.

[LDF 2002.09.17.] Added this function.

⟨ Declare **Path** functions 700 ⟩ +≡

```
virtual bool set_extremes();
```

885.

⟨ Define **Path** functions 701 ⟩ +≡

```
bool Path::set_extremes(){ bool DEBUG = false;   /* true */
    if (DEBUG)
      cout ≪ "Entering␣Path::set_extremes()" ≪ endl ≪ flush;   /* If there are no Points on
          the Path, set all the elements of projective_extremes to INVALID_REAL and return false. */
    if (points.size() ≤ 0) {
      cerr ≪ "ERROR!␣In␣Path::set_extremes():\n" ≪ "points.size()␣<=␣0.␣" ≪
          "Setting␣extremes␣to␣INVALID_REAL␣and␣returning.\n" ≪ flush;
      projective_extremes = INVALID_REAL;
      return false;
    }
```

886. [LDF 2002.09.18.] Added this routine. Set the *minimum* values to `MAX_REAL` and the *maximum* values to −`MAX_REAL`. This way, any valid perspective coordinates will replace them on the first iteration of the **for** loop.

[LDF 2002.09.18.] I had some difficulty debugging this because instead of using −`MAX_REAL`, I defined and used `MIN_REAL` = **numeric_limits**⟨**real**⟩:: *min*( ). However, this isn't the negative **real** with the largest magnitude, but the smallest positive **real**.

⟨ Define **Path** functions 701 ⟩ +≡

```
projective_extremes[0] = MAX_REAL;    /* Minima. */
projective_extremes[2] = MAX_REAL;
projective_extremes[4] = MAX_REAL;
projective_extremes[1] = -MAX_REAL;    /* Maxima. */
projective_extremes[3] = -MAX_REAL;
projective_extremes[5] = -MAX_REAL;
for (vector<Point *>::iterator iter = points.begin(); iter != points.end(); ++iter) {
  if (DEBUG) {
    cout << "min x == " << projective_extremes[0] << endl;
    cout << "max x == " << projective_extremes[1] << endl;
    cout << "x world == " << (**iter).get_x() << endl;
    cout << "x persp == " << (**iter).get_x('p', false, false) << endl;
    cout << "min y == " << projective_extremes[2] << endl;
    cout << "max y == " << projective_extremes[3] << endl;
    cout << "y world == " << (**iter).get_y() << endl;
    cout << "y persp == " << (**iter).get_y('p', false, false) << endl;
    cout << "min z == " << projective_extremes[4] << endl;
    cout << "max z == " << projective_extremes[5] << endl;
    cout << "z world == " << (**iter).get_z() << endl;
    cout << "z persp == " << (**iter).get_z('p', false, false) << endl;
  }
  projective_extremes[0] = min(projective_extremes[0], (**iter).get_x('p', false, false));    /* Min x */
  projective_extremes[2] = min(projective_extremes[2], (**iter).get_y('p', false, false));    /* Min y */
  projective_extremes[4] = min(projective_extremes[4], (**iter).get_z('p', false, false));    /* Min z */
  projective_extremes[1] = max(projective_extremes[1], (**iter).get_x('p', false, false));    /* Max x */
  projective_extremes[3] = max(projective_extremes[3], (**iter).get_y('p', false, false));    /* Max y */
  projective_extremes[5] = max(projective_extremes[5], (**iter).get_z('p', false, false));    /* Max z */
}
if (DEBUG) {
  for (int i = 0; i < 6; i++)
    cout << "projective_extremes[" << i << "] == " << projective_extremes[i] << endl << flush;
}
```

887. [LDF 2002.09.18.] Added this error handling code. There is a remote chance that a valid **Point** could have a coordinate ≡ `MAX_REAL` or ±`MIN_REAL`, however, it is virtually impossible that it would be projectable. If it's the x or y-coordinate, it would probably lie outside the limits defined for the invocation of **Picture** :: *output*( ), and if it was the z, it would either be behind the **Focus** or so far away as to be practically invisible. I believe that this is the case, even though the z-coordinates are made smaller by applying the equation $z_p = z/(z + p)$.

```
⟨ Define Path functions 701 ⟩ +≡
  for (int i = 0; i < 6; i += 2) {
    if (projective_extremes[i] ≡ MAX_REAL      /* Minima */
    ∨ projective_extremes[i + 1] ≡ −MAX_REAL)      /* Maxima */
    {
      if (DEBUG) {
        cout ≪ "i␣==␣" ≪ i ≪ endl ≪ flush;
        cout ≪ "projective_extremes[" ≪ i ≪ "]␣==␣" ≪ projective_extremes[i] ≪ endl ≪ flush;
      }
      cerr ≪ "ERROR!␣In␣Path::set_extremes():\n" ≪ "maxima␣and␣minima␣could␣not␣be␣set␣prop\
          erly.␣" ≪ "Setting␣them␣all␣to␣INVALID_REAL␣and␣returning␣false.\n" ≪ flush;
      projective_extremes = INVALID_REAL;
      return false;
      break;
    }
  }
  if (DEBUG) cout ≪ "Exiting␣Path::set_extremes()" ≪ endl ≪ flush;
  return true; }
```

888. Get extremes. [LDF 2002.09.18.] Added this function. Any code that calls *get_extremes*( ) must ensure that *project*( ) has been invoked first.

```
⟨ Declare Path functions 700 ⟩ +≡
  virtual inline const valarray⟨real⟩ get_extremes( ) const
  {
    return projective_extremes;
  }
```

889. Get minimum z. [LDF 2003.05.16.] Added this function.

```
⟨ Declare Path functions 700 ⟩ +≡
  virtual real get_minimum_z( ) const;
```

890.

⟨ Define **Path** functions 701 ⟩ +≡

```
real Path::get_minimum_z() const
{
  bool DEBUG = false;   /* true */
  if (DEBUG) {
    cout << "Entering␣Path::get_minimum_z()" << endl << flush;
    cout << "projective_extremes[4]␣==␣" << projective_extremes[4] << endl << flush;
    cout << "Exiting␣Path::get_minimum_z()" << endl << flush;
  }
  return projective_extremes[4];
}
```

891. Get maximum z. [LDF 2002.09.17.] Added this function.

⟨ Declare **Path** functions 700 ⟩ +≡

```
virtual real get_maximum_z() const;
```

892.

⟨ Define **Path** functions 701 ⟩ +≡

```
real Path::get_maximum_z() const
{
  bool DEBUG = false;   /* true */
  if (DEBUG) {
    cout << "Entering␣Path::get_maximum_z()" << endl << flush;
    cout << "projective_extremes[5]␣==␣" << projective_extremes[5] << endl << flush;
    cout << "Exiting␣Path::get_maximum_z()" << endl << flush;
  }
  return projective_extremes[5];
}
```

893. Get mean z. [LDF 2003.05.16.] Added this function.

⟨ Declare **Path** functions 700 ⟩ +≡

```
virtual real get_mean_z() const;
```

894.

⟨ Define **Path** functions 701 ⟩ +≡

```
real Path::get_mean_z() const
{
  return ((projective_extremes[4] + projective_extremes[5])/2);
}
```

895. 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 **Path** functions 700 ⟩ +≡

```
virtual void suppress_output();
```

896.

⟨ Define **Path** functions 701 ⟩ +≡

```
void Path::suppress_output()
{
   do_output = false;
}
```

897. 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 **Path** functions 700 ⟩ +≡

```
virtual void unsuppress_output();
```

898.

⟨ Define **Path** functions 701 ⟩ +≡

```
void Path::unsuppress_output()
{
   do_output = true;
}
```

899. Output. [LDF 2002.09.17.] Removed error checking code to *extract*( ). Now *output*( ) assumes that all of the **Points** in *points* can be projected using *focus*. If they can't be, *extract*( ) will already have ensured that the **Path** is not on **vector**⟨**Shape** ∗⟩ *elements* in the **Picture**.

Log

[LDF 2003.01.15.] Added code for writing "drawarrow" to *out_stream*, if *arrow* ≡ *true*. This is for the *drawarrow*( ) functions for **Path** and **Point** that I've added today.

⟨ Declare **Path** functions 700 ⟩ +≡

```
virtual void output();
```

900.

⟨ Define **Path** functions 701 ⟩ +≡

```
void Path::output(){ bool DEBUG = false;    /* true */
      if (DEBUG) cout << "Entering Path::output()" << "\n" << flush;
      if (do_output == false) {
         if (DEBUG) cout << "In Path::output(): do_output == false. Returning.\n" << flush;
         return;
      }
```

901. [LDF 2002.09.27.] Added this error handling code. If the **Path** is empty, don't output it. This code should never be reached, because the case of a **Path** containing no **Points** should be caught in **Path** :: *draw* ( ) and the other drawing and filling commands. If it should reach *set_extremes* ( ) and **Picture** :: *output* ( ), which also shouldn't be possible, they would catch it, too.

⟨ Define **Path** functions 701 ⟩ +≡

```
if (points.size() ≡ 0) {
  cerr ≪ "THIS␣CAN'T␣HAPPEN!␣␣In␣Path::output():\n" ≪
      "This␣code␣should␣never␣be␣reached.\n" ≪ "However,␣I␣may␣be␣able␣to␣recover.␣␣" ≪
      "Type␣<RETURN>␣to␣continue.\n" ≪ flush;
  getchar();
  cerr ≪ "WARNING!␣In␣Path::output():\n" ≪ "Path␣doesn't␣contain␣any␣Points.\n" ≪
      "Not␣doing␣anything.\n\n" ≪ flush;
  return;
}
```

902.

```
⟨ Define Path functions 701 ⟩ +≡
  vector⟨Point *⟩::iterator point_iter = points.begin();
  vector⟨string⟩::iterator connector_iter = connectors.begin();
  string connector_string;
  if (connectors.size() > 0) connector_string = *connector_iter++;
  else connector_string = "--";
  if (fill_draw_value ≡ DRAW) {
    if (DEBUG) cout << "Drawing.\n" << flush;
    if (arrow ≡ true) out_stream << "drawarrow␣" << **point_iter++;
    else out_stream << "draw␣" << **point_iter++;
    ⟨ Output Path 907 ⟩   /* [LDF 2002.09.26.] Comparing pointers seems to work here. I think it
        should, but I wasn't sure that it really would. */
    if (draw_color ≠ Colors::default_color) {
      out_stream << "␣withcolor␣" << *draw_color;
    }
    if (dashed ≠ "") out_stream << "␣dashed␣" << dashed;
    if (pen ≠ "") out_stream << "␣withpen␣" << pen;
    out_stream << ";\n" << flush;
  }
  else if (fill_draw_value ≡ FILL) {
    if (DEBUG) cout << "Filling.\n" << flush;
    out_stream << "fill␣" << **point_iter++;
    ⟨ Output Path 907 ⟩
    if (fill_color ≠ Colors::default_color) out_stream << "␣withcolor␣" << *fill_color;
    out_stream << ";\n" << flush;
  }
  else if (fill_draw_value ≡ FILLDRAW) {
  if (DEBUG) cout << "Filldrawing.\n" << flush;
  if (draw_color ≡ fill_color) {
    out_stream << "filldraw␣" << **point_iter++;
    ⟨ Output Path 907 ⟩
    if (draw_color ≠ Colors::default_color) out_stream << "␣withcolor␣" << *draw_color;
    if (dashed ≠ "") out_stream << "␣dashed␣" << dashed;
    if (pen ≠ "") out_stream << "␣withpen␣" << pen;
    out_stream << ";\n" << flush;
  }
  else   /* We have two different colors, so we have to fill once and draw once. */
  { out_stream << "fill␣" << **point_iter++;
  ⟨ Output Path 907 ⟩
  out_stream << "␣withcolor␣" << *fill_color;
  out_stream << ";\n" << flush;
  point_iter = points.begin();
```

903.

Log

[LDF 2002.05.10.] Added the code in this section. It fixes a bug. If it's not done, then the correct connectors are not used when the **Path** is output the second time.

⟨ Define **Path** functions 701 ⟩ +≡
 connector_iter = *connectors.begin*( );
 if (*connectors.size*( ) > 0) *connector_string* = ∗*connector_iter* ++;
 else *connector_string* = "--";

904.

⟨ Define **Path** functions 701 ⟩ +≡
 if (*arrow* ≡ *true*) *out_stream* ≪ "drawarrow␣" ≪ ∗∗*point_iter* ++;
 else *out_stream* ≪ "draw␣" ≪ ∗∗*point_iter* ++;
 ⟨ Output **Path** 907 ⟩
 if (*draw_color* ≠ **Colors** :: *default_color*) *out_stream* ≪ "␣withcolor␣" ≪ ∗*draw_color*;
 if (*dashed* ≠ "") *out_stream* ≪ "␣dashed␣" ≪ *dashed*;
 if (*pen* ≠ "") *out_stream* ≪ "␣withpen␣" ≪ *pen*;
 out_stream ≪ ";\n" ≪ *flush*; } }
 else
  if (*fill_draw_value* ≡ UNDRAW) {
   if (DEBUG) *cout* ≪ "Undrawing.\n" ≪ *flush*;
   out_stream ≪ "undraw␣" ≪ ∗∗*point_iter* ++;
   ⟨ Output **Path** 907 ⟩
   if (*dashed* ≠ "") *out_stream* ≪ "␣dashed␣" ≪ *dashed*;
   if (*pen* ≠ "") *out_stream* ≪ "␣withpen␣" ≪ *pen*;
   out_stream ≪ ";\n" ≪ *flush*;
  }
  else if (*fill_draw_value* ≡ UNFILL) {
   if (DEBUG) *cout* ≪ "Unfilling.\n" ≪ *flush*;
   out_stream ≪ "unfill␣" ≪ ∗∗*point_iter* ++;
   ⟨ Output **Path** 907 ⟩
   out_stream ≪ ";\n" ≪ *flush*;
  }

905. Filldraw case.

Log

[LDF 2003.03.25.] Changed this section, so that the outline of the **Path** is drawn, if *draw_color* ≠ **Colors** :: *background_color*.

```
⟨ Define Path functions 701 ⟩ +≡
  else
    if (fill_draw_value ≡ UNFILLDRAW) {
      if (DEBUG) cout ≪ "Unfilldrawing.\n" ≪ flush;
      if (draw_color ≡ Colors :: background_color) {
        out_stream ≪ "unfilldraw␣" ≪ **point_iter++;
        ⟨ Output Path 907 ⟩
        if (dashed ≠ "") out_stream ≪ "␣dashed␣" ≪ dashed;
        if (pen ≠ "") out_stream ≪ "␣withpen␣" ≪ pen;
        out_stream ≪ ";\n" ≪ flush;
      }
      else {
        out_stream ≪ "unfill␣" ≪ **point_iter++;
        ⟨ Output Path 907 ⟩
        out_stream ≪ ";\n" ≪ flush;
        point_iter = points.begin();
        connector_iter = connectors.begin();
        if (connectors.size() > 0) connector_string = *connector_iter++;
        else connector_string = "--";
        out_stream ≪ "draw␣" ≪ **point_iter++;
        ⟨ Output Path 907 ⟩
        if (draw_color ≠ Colors :: default_color) out_stream ≪ "␣withcolor␣" ≪ *draw_color;
        if (dashed ≠ "") out_stream ≪ "␣dashed␣" ≪ dashed;
        if (pen ≠ "") out_stream ≪ "␣withpen␣" ≪ pen;
        out_stream ≪ ";\n" ≪ flush;
      }
    }   /* End of UNFILLDRAW case. [LDF 2003.03.25.] */
```

906. Default case. [LDF 2003.03.25.]

```
⟨ Define Path functions 701 ⟩ +≡
  else    /* Use DRAW as default. [LDF 2003.03.25.] */
  {
    cerr << "WARNING!␣Invalid␣|fill_draw_value|:␣" << fill_draw_value << ".␣Using␣\"draw\"\n" <<
        flush;
#if 0    /* !! Define a class for information on the run state. */
    if (¬Run_State :: non_stop) getchar( );
#endif
    if (arrow ≡ true) out_stream << "drawarrow␣" << **point_iter++;
    else out_stream << "draw␣" << **point_iter++;
    ⟨ Output Path 907 ⟩
    if (draw_color ≠ Colors :: default_color) out_stream << "␣withcolor␣" << *draw_color;
    if (dashed ≠ "") out_stream << "␣dashed␣" << dashed;
    if (pen ≠ "") out_stream << "␣withpen␣" << pen;
    out_stream << ";\n" << flush;
  }
  if (DEBUG) cout << "Exiting␣Path::output(Focus)" << "\n" << flush;
  return; }
```

907. When *fill_color* and *draw_color* are different, this will have to be performed twice, so I've made it a named section.

Log

[LDF 2002.11.03.] *counter* is now initially set to 2 instead of 1. This makes each line have at most two **Points**. Previously, the first line had 3 **Points** (if the **Path** had at least three **Points** on it).

[LDF 2002.12.20.] Using the manipulator "*fixed*" below. It solves the problem of **Points** being output in scientific format, which Metapost doesn't understand.

[LDF 2002.12.20.] I had to add preprocessor code for conditional compilation, because "*fixed*" is unknown to the GNU C++ Compiler. However, it doesn't need it in this case, since the problem only occurred when using the DEC C++ compiler on a DEC Alpha computer under Compaq Tru64.

```
⟨ Output Path 907 ⟩ ≡
  if (DEBUG) cout << "Entering␣Output␣|Path|.\n" << flush;
  for (unsigned short counter = 2; point_iter ≠ points.end( ); ) {
    out_stream << "␣" << connector_string << "␣" << **point_iter++;
      /* This breaks the line and indents after two points */
    if (counter ≡ 2 ∧ point_iter ≠ points.end( )) {
      out_stream << "\n␣␣";
      counter = 1;
    }
    else {
      ++counter;
    }
    if (connector_iter ≠ connectors.end( )) connector_string = *connector_iter++;
  }
  if (is_cycle( )) out_stream << "␣" << connector_string << "␣cycle";
  if (DEBUG) cout << "Exiting␣Output␣|Path|.\n" << flush;
```

This code is used in sections 902, 904, 905, and 906.

908. Showing.

909. Show.

Log

[LDF 2003.07.13.] Commented-out the line that prints *fill_draw_value* to *stdout*.

[LDF 2003.08.20.] Now printing *points.size*( ) and *connectors.size*( ) to *stdout*. If the latter is 0, a message is printed, that "--" will be used as the connector.

```
⟨ Declare Path functions 700 ⟩ +≡
  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;
```

910.

⟨ Define **Path** functions 701 ⟩ +≡

```
void Path::show(string text, char coords, const bool do_persp, const bool do_apply, Focus
        *f, const unsigned short proj, const real factor) const
{
    if (text ≡ "") text = "Path:";
    cout ≪ text ≪ endl;
#if 0
    cout ≪ "fill_draw_value␣==␣" ≪ fill_draw_value ≪ endl ≪ flush;
#endif
    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';
    }
    valarray⟨real⟩ v;
    v.resize(4, 0);    /* LDF 2002.12.13. Added this line. Needed for porting to Intel Linux (i686). */
    string connector_string;
    vector⟨string⟩::const_iterator connectors_iter = connectors.begin();
    cout ≪ "points.size()␣==␣" ≪ points.size() ≪ endl ≪ flush;
    cout ≪ "connectors.size()␣==␣" ≪ connectors.size() ≪ endl ≪ flush;
    if (connectors.size() ≡ 0) {
        cout ≪ "Using␣\"--\"␣as␣connector.\n" ≪ flush;
        connector_string = "--";
    }
    {
        int loop_ctr = 0;
        for (vector⟨Point *⟩::const_iterator points_iter = points.begin(); points_iter ≠ points.end();
                ++points_iter) {
            if (points_iter ≠ points.begin()) {
                cout ≪ "␣" ≪ connector_string ≪ "␣";
            }
            if (connectors_iter ≠ connectors.end()) connector_string = *connectors_iter++;
            if (loop_ctr ≡ 2)    /* Break each line after 2 Points. */
            {
                cout ≪ endl;
                loop_ctr = 0;
            }
            ++loop_ctr;
            v = (**points_iter).get_all_coords(coords, do_persp, do_apply, f, proj, factor);
            cout ≪ "(" ≪ v[0] ≪ ",␣" ≪ v[1] ≪ ",␣" ≪ v[2] ≪ ")";
        }
    }
    if (cycle_switch) {
        if (connectors_iter ≠ connectors.end()) connector_string = *connectors_iter++;
        cout ≪ "␣" ≪ connector_string ≪ "␣cycle;" ≪ endl;
```

```
    }
    else cout << ";\n";
  }
```

911. Show Colors.

⟨ Declare **Path** functions 700 ⟩ +≡

```
  void show_colors(bool stop = false);
```

912.

⟨ Define **Path** functions 701 ⟩ +≡

```
  void Path::show_colors(bool stop)
  {
    if (draw_color ≠ 0) draw_color->show("draw_color");
    else cout << "draw_color␣==␣0.\n";
    if (fill_color ≠ 0) fill_color->show("fill_color");
    else cout << "fill_color␣==␣0.\n";
    if (stop) getchar( );
  }
```

913. Returning elements and information.

914. Is on free store.

Log

[LDF 2004.01.06.] Made non-inline.

⟨ Declare **Path** functions 700 ⟩ +≡

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

915.

⟨ Define **Path** functions 701 ⟩ +≡

```
  bool Path::is_on_free_store( ) const
  {
    return on_free_store;
  }
```

916. Is planar. [LDF 2002.11.05.] *is_planar*( ) uses the return value of *get_normal*( ) to determine whether ∗*this* lies in a plane or not. If it does, *is_planar*( ) returns *true*, otherwise, it returns *false*. If ∗*this* is linear, *is_planar*( ) issues a warning and returns *true*.

Log

[LDF 2002.11.03.] Rewrote this function. It should now work for all **Paths**.
[LDF 2002.11.05.] Rewrote this function again. It now uses the new version of *get_normal*( ).
[LDF 2002.11.06.] Added optional **const bool** *verbose* and **string** *text* arguments for writing a message to the standard output.
[LDF 2003.08.14.] Made *verbose* non-**const**. Setting it to *true* if `VERBOSE_GLOBAL` is *true*. Added `VERBOSE_GLOBAL` to `pspglb.web` today.

⟨ Declare **Path** functions 700 ⟩ +≡

```
  virtual bool is_planar(bool verbose = false, string text = "") const;
```

917.

⟨ Define **Path** functions 701 ⟩ +≡

```
bool Path::is_planar(bool verbose, string text) const
{
  bool DEBUG = false;   /* true */
  if (VERBOSE_GLOBAL) verbose = true;
  if (DEBUG ∨ verbose) cout ≪ "Entering␣Path::is_planar().\n";
  Point p(get_normal());
  if (p ≡ INVALID_POINT) {
    if (DEBUG) cout ≪ "Exiting␣Path::is_planar().␣Returning␣false.\n\n" ≪ flush;
    if (verbose) {
      if (text ≡ "") text = "Path";
      cout ≪ text ≪ "␣is␣non-planar.\n\n";
    }
    return false;
  }
  else if (p ≡ origin) {
    cerr ≪ "WARNING!␣In␣Path::is_planar():\n" ≪ "Path␣is␣linear.␣␣Returning␣true.\n\n" ≪
        flush;
    if (verbose) {
      if (text ≡ "") text = "Path";
      cout ≪ text ≪ "␣is␣planar.\n\n";
    }
    return true;
  }
  else {
    if (verbose) {
      if (text ≡ "") text = "Path";
      cout ≪ text ≪ "␣is␣planar.\n\n";
    }
    if (DEBUG) cout ≪ "Exiting␣Path::is_planar().␣Returning␣true.\n\n" ≪ flush;
    return true;
  }
}
```

918. Is linear. [LDF 2003.04.09.] *is_linear*( ) first checks whether *line_switch* is *true*. If it is, it returns *true* right away. Otherwise, it uses the return value of *get_normal*( ) to determine whether ∗*this* is linear or not. If it is, *is_linear*( ) returns *true*, otherwise, it returns *false*.

Log

[LDF 2002.11.05.] Added this function.

[LDF 2002.11.06.] Added optional **const bool** *verbose* and **string** *text* arguments for writing a message to the standard output.

[LDF 2003.04.09.] Now checking whether *line_switch* is *true* before calling *get_normal*( ). !! If a **Path** whose *line_switch* ≡ *true* is modified such that it's no longer linear, the programmer must ensure that *line_switch* is set to *false*!

[LDF 2003.08.14.] Made *verbose* non-**const**. Setting it to *true* if VERBOSE_GLOBAL is *true*. Added VERBOSE_GLOBAL to pspglb.web today.

⟨ Declare **Path** functions 700 ⟩ +≡

```
bool is_linear(bool verbose = false, string text = "") const;
```

919.

⟨ Define **Path** functions 701 ⟩ +≡

```
bool Path::is_linear(bool verbose, string text) const
{
  bool DEBUG = false;    /* true */
  if (VERBOSE_GLOBAL) verbose = true;
  if (DEBUG ∨ verbose) cout ≪ "Entering␣Path::is_linear().\n";
  if (line_switch)    /* LDF 2003.04.09. Added this conditional. */
    return true;
  Point p(get_normal());
  if (p ≡ origin) {
    if (verbose) {
      if (text ≡ "") text = "Path";
      cout ≪ text ≪ "␣is␣linear.\n\n";
    }
    if (DEBUG) cout ≪ "Exiting␣Path::is_linear().␣" ≪ "Returning␣true.\n\n" ≪ flush;
    return true;
  }
  else {
    if (verbose) {
      if (text ≡ "") text = "Path";
      cout ≪ text ≪ "␣is␣non-linear.\n\n";
    }
    if (DEBUG) cout ≪ "Exiting␣Path::is_linear().␣" ≪ "Returning␣false.\n\n" ≪ flush;
    return false;
  }
}
```

920. Get line switch. [LDF 2002.11.03.] This function returns *true* for **Paths** that are created or set using two **Points** only, and no connectors, as arguments.

Log

[LDF 2002.11.03.] Renamed this function *get_line_switch*( ) from *is_line*( ). About to add *is_linear*( ), which will test whether all the **Points** are colinear or not.

⟨ Declare **Path** functions 700 ⟩ +≡

```
inline bool get_line_switch() const
{
  return line_switch;
}
```

921. Test for cycles.

⟨ Declare **Path** functions 700 ⟩ +≡

```
inline bool is_cycle() const
{
  return cycle_switch;
}
```

922. Size (number of points).

⟨ Declare **Path** functions 700 ⟩ +≡

```
inline int size()
{
  return points.size();
}
```

923. Slope. [LDF 2002.11.05.] *slope*( ) can only be used for linear **Paths**. It returns a **real** value representing the slope of the *trace* of a line on the major plane represented by the **char** arguments, or `INVALID_REAL`, if the **Path** is non-linear. For example, if $\overrightarrow{p_0p_1}$ is a **Path** p and $\overrightarrow{q_0q_1}$ is the trace of p on the x-y plane, then *p.slope*('x', 'y') returns a **real** m such that $m = (b - y)/x$ where b is the y-intercept of $\overrightarrow{q_0q_1}$ and x and y are the x and y-coordinates of points on $\overrightarrow{q_0q_1}$.

Log

[LDF 2002.11.05.] Changed this function, so that *is_linear*( ) is used instead of *get_line_switch*( ) (formerly "*is_line*( )"). Now, it can be used for all linear **Paths**, not just ones created using the constructor for lines. Also, it was commented-out.

⟨ Declare **Path** functions 700 ⟩ +≡

```
real slope(char a = 'x', char b = 'y');
```

924.

⟨ Define **Path** functions 701 ⟩ +≡

```
real Path::slope(char a, char b)
{
  if (¬is_linear()) {
    cerr ≪ "ERROR!␣In␣Path::slope().␣Path␣is␣not␣linear!\n" ≪
        "Returning␣INVALID_REAL\n" ≪ flush;
    return INVALID_REAL;
  }
  return points[1]→slope(*points[0], a, b);
}
```

925. Subpath. [LDF 2002.11.05.] *subpath*( ) returns a new **Path** using *points*[*start*] through *points*[*end* − 1] from *∗this*. If the optional **bool** argument *cycle* is used, then the new **Path** will be a cycle, whether *∗this* is or not. One optional connector argument can be used. If it is, it will be the only connector. Otherwise, the connectors from *∗this* are used.

[LDF 2002.11.05.] *start* must be $<$ *end*. It is not possible to have *start* $>$ *end*, even if *∗this* is a cycle.

Log

[LDF 2002.11.05.] Rewrote this function. Made *subpath*( ) itself and its arguments **const**. Added error handling code.

[LDF 2003.07.16.] Please note that *start* and *end* cannot be made **const**.

[LDF 2003.08.27.] Changed **int** *i* to **size_t** *i* in the **for** loops that compare *i* to *start* and *end*. The way it was before caused GCC with the "`-Wall`" option to issue a warning.

⟨ Declare **Path** functions 700 ⟩ +≡

```
Path subpath(size_t start, size_t end, const bool cycle = false, const string connector = "") const;
```

926.

⟨ Define **Path** functions 701 ⟩ +≡

```
Path Path::subpath(size_t start, size_t end, const bool cycle, const string connector) const{ bool
        DEBUG = false;    /* true */
    Path p;
```

927. [LDF 2002.11.05.] There is no "INVALID_PATH", so I return an empty one, if $start \geq end$. Since **operator**≡( ) currently doesn't exist, there's not much point in defining INVALID_PATH, since there's no way to compare another **Path** to it.

⟨ Define **Path** functions 701 ⟩ +≡

```
if (start ≥ end) {
  cerr ≪ "ERROR!␣In␣Path::subpath():\n" ≪ "The␣\"start\"␣argument␣is␣<␣the␣\"end\"\
      ␣argument.\n" ≪ "Returning␣empty␣Path.\n\n" ≪ flush;
  return p;
}
```

928. [LDF 2002.11.05.] More error handling. In these cases, it's possible to recover.

⟨ Define **Path** functions 701 ⟩ +≡

```
if (start > points.size() − 1) {
  cerr ≪ "ERROR!␣In␣Path::subpath():\n" ≪ "\"start\"␣argument␣is␣>␣points.size()␣-\
      ␣1.\n" ≪ "Will␣try␣to␣recover␣by␣setting␣start␣=␣0.\n\n" ≪ flush;
  start = 0;
}
if (end > points.size()) {
  cerr ≪ "ERROR!␣In␣Path::subpath():\n" ≪ "\"end\"␣argument␣is␣>␣points.size().\n" ≪
      "Will␣try␣to␣recover␣by␣setting␣end␣=␣points.size().\n\n" ≪ flush;
  end = points.size();
}
```

929. [LDF 2002.11.05.] If a *connector* argument is specified, all we have to do is put the appropriate **Points** from *points* onto *p.points*, put *connector* onto *p.connectors*, and return *p*.

⟨ Define **Path** functions 701 ⟩ +≡

```
if (connector ≠ "") { for (size_t i = start; i < end; i++) {
if (i ≥ points.size()) {
  cerr ≪ "ERROR!␣In␣Path::subpath():\n" ≪ "end␣argument␣>␣points.size().\n" ≪
      "Breaking␣out␣of␣loop.\n\n" ≪ flush;
  break;
}
p.points.push_back ( create_new < Point > (points[i]) ) ; } p.connectors.push_back(connector);
p.set_cycle(cycle);
return p; }
```

930. [LDF 2002.11.05.] If no *connector* argument is specified, then we have to get the appropriate connectors from **this*. This is slightly tricky, because *connectors* doesn't have to contain a *connector* for each pair of **Points** that is joined in a **Path**. So, first we must fill up *p.connectors* so that we can tell which ones to use.

[LDF 2002.11.05.] Actually, with the constructors that exist, there will either be only one connector or a connector for each pair of **Points** that need to be joined. However, it would be easy to write functions that add or remove connectors, so it's best to have this routine.

⟨ Define **Path** functions 701 ⟩ +≡

```
p = *this;
unsigned short a = points.size();
if (¬cycle) a −= 1;
int i;
string s = connectors.back();
for (i = connectors.size(); i < a; i++) {
  p.connectors.push_back(s);
}
if (DEBUG) cout ≪ "p.connectors.size()␣==␣" ≪ p.connectors.size() ≪ endl ≪ flush;
Path q; for (size_t i = start; i < end; i++) { q.points.push_back ( create_new < Point > (p.points[i])
    ) ;
if (i < p.connectors.size()) q.connectors.push_back(p.connectors[i]);
} q.set_cycle(cycle);
return q; }
```

931. Get point.

932. non-const version.

Log

[LDF 2002.11.05.] Made non-**inline**. Changed return value to **const Point** &.

[LDF 2003.11.28.] BUG FIX: Changed, so that *apply_transform*( ) is called on the **Point**. This entailed making this function non-**const**. Added **const** version below. This may actually be a bug, rather than a bug fix, depending on how this function is used elsewhere. However, I really think *apply_transform*( ) should be called.

⟨ Declare **Path** functions 700 ⟩ +≡

```
const Point &get_point(const unsigned short a);
```

933.

⟨ Define **Path** functions 701 ⟩ +≡

```
const Point &Path::get_point(const unsigned short a)
{
  if (a < points.size()) {
    (points[a])->apply_transform();
    return *points[a];
  }
  else {
    cerr << "ERROR!␣In␣Path::get_point():\n" << "Argument␣is␣>=␣size␣of␣Path.\n" <<
        "Returning␣INVALID_POINT.\n\n" << flush;
    return INVALID_POINT;
  }
}
```

934. const version.

Log

[LDF 2003.11.28.] Added this version.

⟨ Declare **Path** functions 700 ⟩ +≡

```
Point get_point(const unsigned short a) const;
```

935.

⟨ Define **Path** functions 701 ⟩ +≡

```
Point Path::get_point(const unsigned short a) const
{
  if (a < points.size()) {
    Point p = *(points[a]);
    p.apply_transform();
    return p;
  }
  else {
    cerr << "ERROR!␣In␣Path::get_point():\n" << "Argument␣is␣>=␣size␣of␣Path.\n" <<
        "Returning␣INVALID_POINT.\n\n" << flush;
    return INVALID_POINT;
  }
}
```

936. Get last point.

Log

[LDF 2002.05.10.] Added this function.
[LDF 2002.11.05.] Made non-**inline**. Changed return value to **const Point** &.

⟨ Declare **Path** functions 700 ⟩ +≡

```
const Point &get_last_point() const;
```

937.

⟨ Define **Path** functions 701 ⟩ +≡

```
const Point &Path::get_last_point() const
{
  if (points.size() ≠ 0) return *points[points.size() − 1];
  else {
    cerr ≪ "ERROR!␣In␣Path::get_last_point():\n" ≪ "Path␣is␣empty.\n" ≪
        "Returning␣INVALID_POINT.\n\n" ≪ flush;
    return INVALID_POINT;
  }
}
```

938. Get size.

⟨ Declare **Path** functions 700 ⟩ +≡

```
virtual inline size_t get_size() const
{
  return points.size();
}
```

939. Get normal.

940. Path version. [LDF 2002.11.05.] *get_normal*( ) returns a unit vector representing the normal to the plane of the **Path** $*this$, if $*this$ is planar. If the **Points** on $*this$ are colinear and there are no connectors that could make the **Path** non-linear, then *origin* ($(0,0,0)$) is returned. If the **Path** is neither planar nor linear, *get_normal*( ) returns `INVALID_POINT`.

[LDF 2002.11.05.]

- *get_normal*( ) first checks whether a **Path** contains no **Points** or only one **Point**. If so, *get_normal*( ) returns `INVALID_POINT`.
- Then it checks whether the **Path** has connnectors that might make the **Path** non-planar, even if the **Points** lie in a plane. If it does, it returns `INVALID_POINT`. Note that there is no guarantee that the connectors actually *will* make the **Path** non-planar.
- Then it checks whether the **Path** has only two **Points**. If it does, *get_normal*( ) returns the **Point** $(0,0,0)$, because the **Path** will be linear.
- Then it gets the cross product b_0 of $\overrightarrow{p_0p_1}$ and $\overrightarrow{p_0p_2}$, where p_0 and p_1 are the first and second **Points** on the **Path**, and p_2 is the next **Point** on the **Path** such that $b_0 \neq (0,0,0)$. If no **Points** on the **Path** fulfill this condition, then all of the **Points** are colinear, so *get_normal*( ) returns *origin*.
- If, however, $b_0 \neq (0,0,0)$ exists, then cross products b_x are calculated using $\overrightarrow{p_0p_1}$ and the direction vectors $\overrightarrow{p_0p_x}$ for the rest of the **Points** p_x on the **Path**. If and only if $b_x = b_0$, or $b_x = -b_0$, or $b_x = (0,0,0)$ for all b_x, then the **Path** is planar, and *get_normal*( ) returns $-b_0$ (see explanation of sign below). Otherwise, the **Path** is non-planar, and *get_normal*( ) returns `INVALID_POINT`.

[LDF 2003.06.04.] Reversing the sign of b_0 ensures that the normal will point in the direction of the positive y-axis, when a plane figure is created in the x-z plane, using one of the constructors taking a **Point** argument for the center, **real** arguments for the dimensions, and three **real** arguments for the rotation about the major axes. If non-zero arguments are used for rotation, the normal will be rotated accordingly. This direction considered to be "outside". In 3DLDF, the constructors generally generate **Points** moving about the figure in the counter-clockwise direction (as seen from a **Point** with a positive y-coordinate). However, according to Huw Jones, *Computer Graphics Through Key Mathematics*, p. 197, "outside" is considered to be the side of a plane, where the **Points** are meant to be traversed in the clockwise direction. !! Watch out for problems that may arise from this discrepancy!

Log

[LDF 2002.11.05.] Rewrote this function.

[LDF 2003.06.04.] Changed sign of the normal, when it's returned, in the cases where a proper normal is found (not `INVALID_POINT` or *origin*). See explanation above.

```
⟨ Declare Path functions 700 ⟩ +≡
  virtual Point get_normal( ) const;
```

941.

```
⟨ Define Path functions 701 ⟩ +≡
  Point Path::get_normal( ) const { bool DEBUG = false;    /* true */
      if (DEBUG) cout ≪ "Entering␣Path::get_normal().\n" ≪ flush;
      if (points.size( ) ≤ 0) {
        cerr ≪ "WARNING!␣In␣Path::get_normal():\n" ≪
            "Path␣is␣empty␣or␣contains␣only␣one␣Point.␣" ≪ "Returning␣INVALID_POINT.\n\n" ≪
            flush;
        return INVALID_POINT;
      }
```

942. [LDF 2002.11.05.] Connectors other than the ones in the conditional below could cause the **Path** to be non-linear or non-planar, even if the **Points** lie on a line or in a plane.

⟨ Define **Path** functions 701 ⟩ +≡

```
for (vector⟨string⟩::const_iterator iter = connectors.begin(); iter ≠ connectors.end(); ++iter) {
  if (¬(*iter ≡ ".." ∨ *iter ≡ "--" ∨ *iter ≡ "..." ∨ *iter ≡ "---")) {
    cerr ≪ "WARNING!␣In␣Path::get_normal():\n" ≪ "Connector␣may␣make␣Path␣non-linear␣or␣n\
        on-planar:␣" ≪ *iter ≪ endl ≪ "Returning␣INVALID_POINT.\n\n" ≪ flush;
    return INVALID_POINT;
  }
}
```

943. [LDF 2002.11.05.] Two points determine a line.

⟨ Define **Path** functions 701 ⟩ +≡

```
if (points.size() ≡ 2) {
  cerr ≪ "WARNING!␣In␣Path::get_normal():\n" ≪
      "Path␣has␣2␣Points.␣␣Returning␣origin.\n\n" ≪ flush;
  return origin;
}
```

944.

```
⟨ Define Path functions 701 ⟩ +≡
  vector⟨Point *⟩::const_iterator iter = points.begin();
  Point p0(**iter++);
  Point p1(**iter++);
  Point p2;
  Point a0(p1 − p0);
  Point a1;
  Point b0;
  while (b0 ≡ origin ∧ iter ≠ points.end()) {
    p2 = **iter++;
    a1 = p2 − p0;
    b0 = a0.cross_product(a1);
  }
  if (iter ≡ points.end() ∧ b0 ≡ origin) {
    if (DEBUG) cout ≪ "Exiting␣Path::get_normal().␣" ≪ "Points␣are␣all␣colinear.\n" ≪
          "Returning␣origin.\n\n" ≪ flush;
    return origin;
  }
  if (DEBUG) b0.show("b0");
  b0.unit_vector(true);
  if (iter ≡ points.end() ∧ b0 ≠ origin) {
    if (DEBUG) {
      cout ≪ "Exiting␣Path::get_normal().␣" ≪ "Points␣are␣all␣colinear␣except␣for␣one.\n" ≪
          "Returning␣normal.\n\n" ≪ flush;
    }
    return −b0;
  }
  Point b1;
  if (DEBUG) cout ≪ "Entering␣second␣while.\n" ≪ flush;
  while (iter ≠ points.end()) {
    p2 = **iter++;
    a1 = p2 − p0;
    b1 = a0.cross_product(a1);
    if (b1 ≠ origin)
        /* [LDF 2002.11.03.] This if merely prevents a warning from being issued by unit_vector(). */
      b1.unit_vector(true);
    if (DEBUG) b1.show("b1");
    if (¬(b1 ≡ origin ∨ b1 ≡ b0 ∨ b1 ≡ −b0)) {
      if (DEBUG)
        cout ≪ "Exiting␣Path::get_normal().␣" ≪ "Returning␣INVALID_POINT.\n\n" ≪ flush;
      return INVALID_POINT;
    }
  }
  if (DEBUG) cout ≪ "Exiting␣Path::get_normal().␣" ≪ "Returning␣normal.\n\n" ≪ flush;
  return −b0; }
```

945. Point version. **Point** :: *get_normal*( ) is declared `points.web`, but it must be defined here, because it calls **Path** :: *get_normal*( ). [LDF 2003.07.11.]

Log

[LDF 2003.07.11.] Added this function.

⟨ Define **Point** functions 330 ⟩ +≡
```
Point Point::get_normal(const Point &p, const Point &q) const
{
  Path r;
  r.set_connectors("--");
  r += *this;
  r += p;
  r += q;
  if (¬r.is_planar()) {
    cerr ≪ "ERROR!␣In␣Point::get_normal():\n" ≪
        "The␣Points␣do␣not␣determine␣a␣plane.\n" ≪ "Returning␣INVALID_POINT.\n\n" ≪ flush;
    return INVALID_POINT;
  }
  else return r.get_normal();
}
```

946. Get plane.

Log

[LDF 2002.11.05.] Rewrote this function to correspond to the new definition of *get_normal*( ).

⟨ Declare **Path** functions 700 ⟩ +≡
```
virtual Plane get_plane() const;
```

947.

⟨ Define **Path** functions 701 ⟩ +≡

```
Plane Path::get_plane() const
{
  Point normal(get_normal());
  if (normal ≡ INVALID_POINT ∨ normal ≡ origin) {
    cerr ≪ "WARNING!␣In␣Path::get_plane().\n" ≪
        "Path␣is␣not␣a␣Plane.␣Returning␣INVALID_PLANE.\n\n" ≪ flush;
    return INVALID_PLANE;
  }
  Point point(get_point(0));
  return Plane(point, normal);
}
```

948. Point lies within triangle. [LDF 2003.06.11.] Declared in `points.web`. Must be defined here, because **Path** is an incompletely defined type there.

Log

[LDF 2003.06.11.] Added this function.
[LDF 2003.06.24.] Removed the argument *test_points*. Now, planarity is always tested.
[LDF 2003.06.24.] BUG FIX: When the **Points** all lay in the x-z plane, or a plane parallel to it, *lambda_denominator* was 0. This caused *is_in_triangle*( ) to return *false*, even when ∗*this* did lie in the triangle. Now, if *lambda_denominator* or *mu_denominator* is equal to 0, the y and z-coordinates are exchanged, and *lambda_denominator* and *mu_denominator* are recalculated. If either of the new values is 0, the x and z-coordinates are exchanged (based on the original coordinate values), and *lambda_denominator* and *mu_denominator* are again recalculated. Only one exchange has been needed in the cases I've tested so far.
[LDF 2003.08.14.] Setting *verbose* to *true* if `VERBOSE_GLOBAL` is *true*. Added `VERBOSE_GLOBAL` to `pspglb.web` today.

⟨ Define **Point** functions 330 ⟩ +≡

```
bool Point::is_in_triangle(const Point &p0, const Point &p1, const Point &p2, bool verbose)
      const{ bool DEBUG = false;    /* true */
    if (VERBOSE_GLOBAL) verbose = true;
    Path q;
    q += p0;
    q += p1;
    q += p2;
    Plane q_pl = q.get_plane();
    if (q_pl ≡ INVALID_PLANE) {
      if (verbose) {
        cerr ≪ "WARNING!␣In␣Point::is_in_triangle():\n" ≪
            "The␣Point␣arguments␣do␣not␣determine␣a␣plane.\n" ≪ "Returning␣false.\n\n" ≪
            flush;
      }
      return false;
    }
    else if (¬is_on_plane(q_pl)) {
      if (verbose) {
```

```
      cerr ≪ "WARNING!␣In␣Point::is_in_triangle():\n" ≪
          "*this␣doesn't␣lie␣in␣the␣plane␣determined␣" ≪ "by␣the␣arguments.\n" ≪
          "Returning␣false.\n\n" ≪ flush;
    }
    return false;
  }
  Point t(*this);
  Point c(p0);
  Point d(p1);
  Point e(p2);
  t.apply_transform();
  c.apply_transform();
  d.apply_transform();
  e.apply_transform();
  if (DEBUG) {
    show("t:");
    c.show("c:");
    d.show("d:");
    e.show("e:");
  }
  real t_x = t.world_coordinates[0];
  real t_y = t.world_coordinates[1];
  real t_z = t.world_coordinates[2];
  real c_x = c.world_coordinates[0];
  real c_y = c.world_coordinates[1];
  real c_z = c.world_coordinates[2];
  real d_x = d.world_coordinates[0];
  real d_y = d.world_coordinates[1];
  real d_z = d.world_coordinates[2];
  real e_x = e.world_coordinates[0];
  real e_y = e.world_coordinates[1];
  real e_z = e.world_coordinates[2];
  real lambda_denominator = (((e_x − c_x) ∗ (d_y − c_y)) − ((e_y − c_y) ∗ (d_x − c_x)));
  real mu_denominator = ((e_x − c_x) ∗ (d_y − c_y)) − ((e_y − c_y) ∗ (d_x − c_x));
  bool exchange_y_z = false;
  bool exchange_x_z = false; if (lambda_denominator ≡ 0 ∨ mu_denominator ≡ 0) {
  if (DEBUG) cout ≪ "lambda_denominator␣or␣mu_denominator␣==␣0.␣" ≪
        "Exchanging␣y␣and␣z-coordinates.\n" ≪ flush;
  real temp;
  temp = t_y;
  t_y = t_z;
  t_z = temp;
  temp = c_y;
  c_y = c_z;
  c_z = temp;
  temp = d_y;
  d_y = d_z;
  d_z = temp;
  temp = e_y;
  e_y = e_z;
  e_z = temp;
```

```
lambda_denominator = (((e_x − c_x) ∗ (d_y − c_y)) − ((e_y − c_y) ∗ (d_x − c_x)));
mu_denominator = ((e_x − c_x) ∗ (d_y − c_y)) − ((e_y − c_y) ∗ (d_x − c_x));
if (¬(lambda_denominator ≡ 0 ∨ mu_denominator ≡ 0)) {
  if (DEBUG) cout ≪ "Exchanging␣y␣and␣z-coordinates␣worked.\n" ≪
        "lambda_denominator␣and␣mu_denominator␣are␣no␣longer␣0.\n" ≪ flush;
  exchange_y_z = true;
}
else {
if (DEBUG) cout ≪ "Exchanging␣y␣and␣z-coordinates␣didn't␣work.\n" ≪
      "Exchanging␣x␣and␣z-coordinates.\n" ≪ flush;
```

949. First, put things back the way they were. It's wasteful, but less confusing. [LDF 2003.06.24.]

⟨ Define **Point** functions 330 ⟩ +≡

```
temp = t_y;
t_y = t_z;
t_z = temp;
temp = c_y;
c_y = c_z;
c_z = temp;
temp = d_y;
d_y = d_z;
d_z = temp;
temp = e_y;
e_y = e_z;
e_z = temp;
```

950. Now, exchange the x and z-coordinates. [LDF 2003.06.24.]

⟨ Define **Point** functions 330 ⟩ +≡

```
temp = t_x;
t_x = t_z;
t_z = temp;
temp = c_x;
c_x = c_z;
c_z = temp;
temp = d_x;
d_x = d_z;
d_z = temp;
temp = e_x;
e_x = e_z;
e_z = temp;
lambda_denominator = (((e_x − c_x) ∗ (d_y − c_y)) − ((e_y − c_y) ∗ (d_x − c_x)));
mu_denominator = ((e_x − c_x) ∗ (d_y − c_y)) − ((e_y − c_y) ∗ (d_x − c_x));
if (¬(lambda_denominator ≡ 0 ∨ mu_denominator ≡ 0)) {
  if (DEBUG) cout ≪ "Exchanging␣x␣and␣z-coordinates␣worked.\n" ≪
        "lambda_denominator␣and␣mu_denominator␣are␣no␣longer␣0.\n" ≪ flush;
  exchange_x_z = true;
}
else {
  if (verbose ∨ DEBUG) {
    cerr ≪ "WARNING!␣In␣Point::is_in_triangle():\n" ≪
        "lambda_denominator␣or␣mu_denominator␣is␣0.\n" ≪ "Returning␣false.\n\n" ≪ flush;
  }
  return false;
}
} } real lambda = (((t_x − c_x) ∗ (d_y − c_y)) − ((t_y − c_y) ∗ (d_x − c_x)))/lambda_denominator;
real mu = −((((t_x − c_x) ∗ (e_y − c_y)) − ((t_y − c_y) ∗ (e_x − c_x)))/mu_denominator);
if (DEBUG) {
  cout ≪ "lambda␣==␣" ≪ lambda ≪ endl ≪ flush;
  cout ≪ "mu␣==␣" ≪ mu ≪ endl ≪ flush;
  cout ≪ "lambda␣+␣mu␣==␣" ≪ lambda + mu ≪ endl ≪ flush;
  cout ≪ "(lambda␣>=␣0␣&&␣mu␣>=␣0␣&&␣((lambda␣+␣mu)␣<=␣1))␣==␣" ≪ (lambda ≥ 0 ∧ mu ≥
      0 ∧ ((lambda + mu) ≤ 1)) ≪ endl ≪ flush;
}
bool b = (lambda ≥ 0 ∧ mu ≥ 0 ∧ ((lambda + mu) ≤ 1));
if (verbose) {
  cout ≪ "In␣Point::is_in_triangle:\n";
  if (b) cout ≪ "The␣Point␣lies␣within␣the␣triangle.␣" ≪ "Returning␣true.";
  else cout ≪ "The␣Point␣doesn't␣lie␣within␣the␣triangle.␣" ≪ "Returning␣false.";
  cout ≪ endl ≪ endl ≪ flush;
}
return b; }
```

951. Manipulating Paths.

952. Set cycle.

Log

[LDF 2002.4.7.] Changed, so that the argument **bool** *c* is *true* by default.

[LDF 2002.11.05.] Made **bool** c argument **const**.

⟨ Declare **Path** functions 700 ⟩ +≡
```
void set_cycle(const bool c = true);
```

953.

⟨ Define **Path** functions 701 ⟩ +≡
```
void Path::set_cycle(const bool c)
{
  cycle_switch = c;
}
```

954. Reverse.

955. With assignment.

Log

[LDF 2002.4.6.] Added this function.

[LDF 2003.07.16.] Added error handling code for the case that this function is called with *assign* ≡ *false*. I've now added a **const** version, so there's no need to call this version with *assign* ≡ *false*. If *assign* is *false*, the **const** version is called, so I could simplify the code in this version.

⟨ Declare **Path** functions 700 ⟩ +≡
```
Path reverse(bool assign);
```

956.

⟨ Define **Path** functions 701 ⟩ +≡
```
Path Path::reverse(bool assign){ bool DEBUG = false;   /* true */
    if (is_cycle())   /* Return *this if *this is a cycle. */
    {
      cerr << "ERROR!␣In␣Path::reverse().\n" << "*this␣is␣a␣cycle.␣Can't␣reverse.\n" <<
          "Returning␣*this.\n\n";
      return *this;
    }
    if (¬assign) {
      cerr << "WARNING!␣In␣Path::reverse(bool):\n" << "assign␣==␣false.␣Do\
          n't␣call␣this␣function␣" << "with␣false␣as␣its␣argument.\n" <<
          "Use␣reverse()␣without␣an␣argument␣instead.\n" << "Calling␣reverse(void).\n\n" <<
          flush;
      return reverse();
    }
```

957. [LDF 2002.4.6.] If there is more than one connector, but there isn't an explicit connector for every pair of **Points** in *points*, then we have to fill up *connectors* so that there is one for each pair of **Points**. Otherwise, the connectors and the **Points** won't match up properly when we reverse them.

⟨ Define **Path** functions 701 ⟩ +≡
```
if (connectors.size() > 1 ∧ connectors.size() ≠ points.size() − 1) {
  string last_connector;
  last_connector = connectors.back();
  while (connectors.size() < points.size() − 1) connectors.push_back(last_connector);
}
```

958. [LDF 2002.4.7.] If I don't explicitly refer to the **std** namespace here, this function is called, and since the arguments are different from the one used for this function, this causes an error at compile time.

⟨ Define **Path** functions 701 ⟩ +≡
```
if (DEBUG) cout ≪ "Reversing␣connectors␣and␣points.\n" ≪ flush;
std::reverse(connectors.begin(), connectors.end());
std::reverse(points.begin(), points.end());
if (DEBUG) {
  cout ≪ "Showing␣connectors:\n";
  for (vector⟨string⟩::iterator iter = connectors.begin(); iter ≠ connectors.end(); iter++)
    cout ≪ ∗iter ≪ endl;
  cout ≪ "Showing␣points:\n";
  for (vector⟨Point ∗⟩::iterator iter = points.begin(); iter ≠ points.end(); iter++) (∗∗iter).show();
  getchar();
}    /∗ if (DEBUG) ∗/
return ∗this; }
```

959. No assignment. This version merely copies *∗this* and calls *reverse*(*true*) on the copy, returning the return value of that function call.

Log

[LDF 2003.07.16.] Added this function.

⟨ Declare **Path** functions 700 ⟩ +≡
```
Path reverse(void) const;
```

960.

⟨ Define **Path** functions 701 ⟩ +≡
```
Path Path::reverse(void) const
{
  Path p = ∗this;
  return p.reverse(true);
}
```

961. Equality. TO DO: I'll need to make all connectors explicit in order to make this work. See **operator**&( ) for an example of how to make this work.

⟨ Declare **Path** functions 700 ⟩ +≡
```
#if 0
  virtual bool operator≡(Path &p);
#endif
```

962.

⟨ Define **Path** functions 701 ⟩ +≡
#if 0
  virtual bool Path :: **operator**≡(**Path** &*p*)
  { }
#endif

963. Intersection.

964. Intersection of two linear Paths. If ∗*this* is a line and the argument *pa* is a line, *intersection_point*( ) calls the version for **Points** in `points.web`.

Other kinds of **Paths** and other classes will need their own versions of this function.

I may have a problem with the constancy of ∗*this* and *pa*. If I do, just remove it.

Log

[LDF 2002.04.15.] Changed return value from **bool_real_point** to **bool_point**, since I've had to comment-out the version of **Point** :: *intersection_point*( ) that uses the **Line** version.

[LDF 2002.04.10.] Changed return type to **bool_real_point** to correspond with the same change to **Point** :: *intersection_point*( ).

[LDF 2003.07.04.] Added *trace* argument. Added conditional using *trace* to choose which version of **Point** :: *intersection_point*( ) should be called. Changed so that *is_linear*( ) is used instead of *get_line_switch*( ). Now using *get_last_point*( ) instead of ∗*points*[1].

⟨ Declare **Path** functions 700 ⟩ +≡
  bool_point *intersection_point*(**const Path** &*pa*, **const bool** *trace* = *false*) **const**;

965.

⟨ Define **Path** functions 701 ⟩ +≡

```
bool_point Path::intersection_point(const Path &pa, const bool trace) const
{
  if (is_linear() ∧ pa.is_linear()) {
    if (trace) return Point::intersection_point(*points[0], get_last_point(), *pa.points[0],
          pa.get_last_point(), trace);
    else return Point::intersection_point(*points[0], get_last_point(), *pa.points[0], pa.get_last_point(),
          trace);
  }
  else {
    cout ≪ "Haven't coded this case yet. " ≪ "Returning INVALID_BOOL_POINT.\n" ≪ flush;
    return INVALID_BOOL_POINT;
  }
}
```

966. Intersection of a linear Path with a Plane. [LDF 2003.06.03.] This function must be defined here, because **Path** is an incomplete type in `planes.web`.

Log

[LDF 2003.06.03.] Added this function.

⟨ Define **Plane** functions 664 ⟩ +≡

```
bool_point Plane::intersection_point(const Path &p) const
{
  if (p.is_linear()) return intersection_point(p.get_point(0), p.get_last_point());
  else {
    cerr ≪ "ERROR! In Plane::intersection_point(const Path&):" ≪ endl ≪
        "Path is not linear!  Returning INVALID_BOOL_POINT." ≪ endl ≪ endl ≪ flush;
    return INVALID_BOOL_POINT;
  }
}
```

967. Drawing axes. This function draws and labels arrows for the main axes at the origin. It can be helpful for determining whether the "*up*" direction is correct for a **Focus**.

[LDF 2003.04.01.] Sometimes placeholders are needed for the *dist* and position arguments. If *dist* is a number $x \leq 0$, then it's set to the default, currently 2.5. If a position argument (*pos_x*, *pos_y*, or *pos_z*) is `"d"`, it's set to the default.

Log

[LDF 2003.02.05.] Moved this function from `main.web` to here, so I can use it in my examples for the Texinfo documentation. Also, added additional arguments specifying the positions of labels and suppressing drawing the axes (and their labels).

[LDF 2003.04.01.] Added arguments for dash pattern (*ddashed*) and pen (*ppen*). Rearranged order of arguments. Also, got rid of the arguments *suppress_x*, *suppress_y*, and *suppress_z*. Now using the empty string (`""`) in the arguments *pos_x*, *pos_y*, and *pos_z* to indicate that the corresponding axes should be suppressed. Added error handling code that prints a warning to *stderr* if all axes are suppressed. ([LDF 2003.05.06.] Note that `""` will never be needed for labelling an axis, because putting the label on top of the **Point** would interfere with the arrow.)

[LDF 2003.04.01.] Added arguments *shift_x*, *shift_y*, and *shift_z* for adjusting the position of the labels. Note that the adjustment affects the position of the three-dimensional **Point** within the **Label**, *not* the two-dimensional projected point. Therefore, it's not possible to adjust the position of the **Label** precisely without changing the Metapost code. TO DO: Change *label*( ), so that it's possible to adjust the position of the points in the projection! This may open a can of worms, though, especially if the same code is used to generate drawings using different projections and/or different *Focuses*.

[LDF 2003.07.13.] Made *ddashed* and *ppen* **const** in both versions.

---

968. Length argument first.

⟨ Declare *draw_axes*( ) 968 ⟩ ≡
void *draw_axes*(**real** *dist* = 2.5, **string** *pos_x* = "bot", **string** *pos_y* = "lft", **string** *pos_z* = "bot", **const Color** &*ddraw_color* = ∗**Colors** :: *default_color*, **const string** *ddashed* = "", **const string** *ppen* = "", **const Point** &*shift_x* = *origin*, **const Point** &*shift_y* = *origin*, **const Point** &*shift_z* = *origin*, **Picture** &*picture* = *current_picture*);

See also section 973.

This code is used in section 981.

969.

⟨ Define *draw_axes*( ) 969 ⟩ ≡
void *draw_axes*(**real** *dist*, **string** *pos_x*, **string** *pos_y*, **string** *pos_z*, **const Color** &*ddraw_color*, **const string** *ddashed*, **const string** *ppen*, **const Point** &*shift_x*, **const Point** &*shift_y*, **const Point** &*shift_z*, **Picture** &*picture*){

See also sections 970, 971, 972, and 974.

This code is used in section 980.

970. Remember to change this if you change any of the defaults!

Log

[LDF 2003.04.01.] Added this section.

⟨ Define *draw_axes*( ) 969 ⟩ +≡
if (*dist* ≤ 0) *dist* = 2.5;
if (*pos_x* ≡ "d") *pos_x* = "bot";
if (*pos_y* ≡ "d") *pos_y* = "lft";
if (*pos_z* ≡ "d") *pos_z* = "bot";

971.

Log

[LDF 2003.04.01.] Added this error handling code.

⟨ Define *draw_axes*( ) 969 ⟩ +≡
if (*pos_x* ≡ "" ∧ *pos_y* ≡ "" ∧ *pos_z* ≡ "") {
cerr ≪ "WARNING!␣In␣draw_axes():" ≪ *endl* ≪ "All␣axes␣are␣suppressed.␣Returning." ≪ *endl* ≪ *endl* ≪ *flush*;
return;
}

972.

⟨ Define *draw_axes*( ) 969 ⟩ +≡

```
if (pos_x ≠ "") {
   Point x0(−dist);
   Point x1(dist);
   x0.drawarrow(x1, ddraw_color, ddashed, ppen, picture);
   x1 += shift_x;
   x1.label("x", pos_x, false, picture);
}
if (pos_y ≠ "") {
   Point y0(0, −dist);
   Point y1(0, dist);
   y0.drawarrow(y1, ddraw_color, ddashed, ppen, picture);
   y1 += shift_y;
   y1.label("y", pos_y, false, picture);
}
if (pos_z ≠ "") {
   Point z0(0, 0, −dist);
   Point z1(0, 0, dist);
   z0.drawarrow(z1, ddraw_color, ddashed, ppen, picture);
   z1 += shift_z;
   z1.label("z", pos_z, false, picture);
}
return; }
```

973. Color argument first.

Log

[LDF 2003.05.02.] Added this function.

⟨ Declare *draw_axes*( ) 968 ⟩ +≡

```
void draw_axes(const Color &ddraw_color, real dist = 2.5, string pos_x = "bot", string
      pos_y = "lft", string pos_z = "bot", const string ddashed = "", const string ppen = "", const
      Point &shift_x = origin, const Point &shift_y = origin, const Point &shift_z = origin, Picture
      &picture = current_picture);
```

974.

⟨ Define *draw_axes*( ) 969 ⟩ +≡

```
void draw_axes(const Color &ddraw_color, real dist, string pos_x, string pos_y, string pos_z, const
        string ddashed, const string ppen, const Point &shift_x, const Point &shift_y, const Point
        &shift_z, Picture &picture)
{
  draw_axes(dist, pos_x, pos_y, pos_z, ddraw_color, ddashed, ppen, shift_x, shift_y, shift_z, picture);
}
```

975. Paths and Lines.

Log

[LDF 2003.06.06.] Added this heading.

---

976. Get Line. Returns a **Line** corresponding to *∗this*, if *∗this* is linear. Otherwise, *get_line*( ) returns `INVALID_LINE`.

Log

[LDF 2003.06.06.] Added this function.

⟨ Declare **Path** functions 700 ⟩ +≡

```
Line get_line(void) const;
```

977.

⟨ Define **Path** functions 701 ⟩ +≡

```
Line Path::get_line(void) const
{
  if (is_linear()) return points.front()→get_line(*(points.back()));
  else {
    cerr ≪ "ERROR!␣In␣Path::get_line():\n" ≪ "Path␣is␣not␣linear.␣Returning␣INVALID_L\
        INE.\n\n" ≪ flush;
    return INVALID_LINE;
  }
}
```

978. Get Path. Declared in `lines.web`. Must be defined here, because **Path** is an incomplete type there.

Log

[LDF 2003.06.06.] Added this function.

⟨ Define **Line** functions 644 ⟩ +≡

```
Path Line::get_path(void) const
{
  Point p(position + direction);
  return Path(position, p);
}
```

979. Putting Path together.

980. This is what's compiled.

⟨ Include files 6 ⟩
⟨ Version control identifier 5 ⟩
⟨ Define **class Path** 698 ⟩
⟨ Define **static class Path** data members 699 ⟩
⟨ Define **Transform** functions 169 ⟩
⟨ Define **Point** functions 330 ⟩
⟨ Define **Plane** functions 664 ⟩
⟨ Define **Path** functions 701 ⟩
⟨ Define **Line** functions 644 ⟩
⟨ Define *draw_axes*( ) 969 ⟩
⟨ Declare non-member template functions for **Path** 724 ⟩

981. This is what's written to `paths.h`.

⟨ `paths.h` 981 ⟩ ≡
  ⟨ Define **class Path** 698 ⟩
  ⟨ Declare *draw_axes*( ) 968 ⟩
  ⟨ Declare non-member template functions for **Path** 724 ⟩

982. Curves (`curves.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.

---

format *Curve* *Path*

⟨ Version control identifier 5 ⟩ +≡
  static string *rcs_id* = `"$Id:␣curves.web,v␣1.5␣2004/01/12␣21:27:59␣lfinsto1␣Exp␣$"`;

983. Include files.

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

984. Regular closed plane curve.

Log

[LDF 2002.11.12.] Changed the name "*Regular_Closed_Plane_Curve*" to "**Reg_Cl_Plane_Curve**", because the former caused too many "Overfull boxes" when running `cweave`.

---

985. Reg_Cl_Plane_Curve class definition. A **Reg_Cl_Plane_Curve** is assumed to be closed, planar, convex, and have at least 3 points. The functions that create and modify **Reg_Cl_Plane_Curves** must ensure that these assumptions are correct!

[LDF 2002.11.05.] **Reg_Cl_Plane_Curve** is intended to be used as a base class. No objects of type **Reg_Cl_Plane_Curve** should be defined, however, it is not an abstract class, so it is possible to do so.

format *Reg_Cl_Plane_Curve* *Curve*

⟨ Define **class Reg_Cl_Plane_Curve** 985 ⟩ ≡
  class Reg_Cl_Plane_Curve : **public Path** {
  protected: **Point** *center*;
    unsigned short *number_of_points*;
  public: ⟨ Declare **Reg_Cl_Plane_Curve** functions 987 ⟩

};

This code is used in sections 1015 and 1016.

986. Returning elements and information. [LDF 2002.11.05.] The **virtual** functions in this section are meant to be overloaded by member functions of types derived from **Reg_Cl_Plane_Curve**.

Log

[LDF 2002.11.03.] Removed **Reg_Cl_Plane_Curve** :: *is_planar*( ).
A **Reg_Cl_Plane_Curve** can be manipulated into a non-planar state, so it's safer to use the **Path** version, which tests whether it's really planar or not.

---

987. Is quadratic.

⟨ Declare **Reg_Cl_Plane_Curve** functions 987 ⟩ ≡

```
inline virtual bool is_quadratic( ) const
{
  return false;
}
```

See also sections 988, 989, 990, 991, 992, 994, 997, 1008, 1011, 1013, and 1014.

This code is used in section 985.

988. Is cubic.

⟨ Declare **Reg_Cl_Plane_Curve** functions 987 ⟩ +≡

```
inline virtual bool is_cubic( ) const
{
  return false;
}
```

989. Is quartic.

⟨ Declare **Reg_Cl_Plane_Curve** functions 987 ⟩ +≡

```
inline virtual bool is_quartic( ) const
{
  return false;
}
```

990. Get coefficients.

⟨ Declare **Reg_Cl_Plane_Curve** functions 987 ⟩ +≡

```
inline virtual real_triple get_coefficients(real, real) const
{
  return real_triple(INVALID_REAL, INVALID_REAL, INVALID_REAL);
}
```

991. Solve. [LDF 2002.11.05.] This **virtual** function is meant to be overloaded by member functions of types derived from **Reg_Cl_Plane_Curve**.

⟨ Declare **Reg_Cl_Plane_Curve** functions 987 ⟩ +≡

```
inline virtual pair<real, real> solve(char, real) const
{
  return pair<real, real>(INVALID_REAL, INVALID_REAL);
}
```

992. Location of a point. *location*( ) returns a **signed short** indicating the location of its **Point** argument with respect to the **Reg_Cl_Plane_Curve**.

[LDF 2002.11.05.] TO DO: Currently, the programmer must ensure that a **Reg_Cl_Plane_Curve** is planar. It might be worthwhile to check that it really is by using **Path** :: *get_normal*( ), since some manipulations may cause a **Reg_Cl_Plane_Curve** to become non-planar.

[LDF 2002.11.05.] The number of **Points** in a **Reg_Cl_Plane_Curve** must be a multiple of 4, and that the **Point** *number_of_points* /4 must be at 90° to **Point** 0. Also, *ref_pt* can't be **Point** 0.

[LDF 2003.07.16.] **Reg_Cl_Plane_Curve** now has a data member named *center*. However, a **Reg_Cl_Plane_Curve** need not have a meaningful center. Usually, when an object of a class derived from **Reg_Cl_Plane_Curve** calls this function, its *center* will be passed as the *ref_pt* argument. However, this need not be the case.

TO DO: Check whether it will work if $pt0.x < 0$. I think it should.

[LDF 2003.06.14.] !! CHECK. Bug, when **Reg_Cl_Plane_Curve** is rotated about x and z-axes only.

The following values are returned if the **Point** is in the same plane as **this* and this function has worked properly:

-1 The **Point** lies outside the **Reg_Cl_Plane_Curve**.

0 The **Point** lies on the perimeter of the **Reg_Cl_Plane_Curve**.

1 The **Point** lies inside the perimeter of the **Reg_Cl_Plane_Curve**.

These values are returned in cases where errors have occurred:

-2 The **Point** is not in the same plane as the **Reg_Cl_Plane_Curve**.

-3 Something has gone terribly wrong.

-4 The normal to the **Reg_Cl_Plane_Curve** has 0 magnitude, i.e., the **Points** on the **Reg_Cl_Plane_Curve** are colinear.

-5 An error occurred in putting the **Reg_Cl_Plane_Curve** in one of the major planes.

-6 The **Reg_Cl_Plane_Curve** is non-planar.

Log

[LDF 2002.04.03.] Added and tested all cases. Seems to work properly.

[LDF 2002.11.12.] Added "`\relax`" after the arguments to "`\RV`" 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 `\RV` (which is currently "`\let`" to `\ARG`).

[LDF 2003.06.03.] Changed the line where **Plane** :: *get_distance*( ) is called below. It now returns a **real_short**, so ". *first*" has to be added, in order to get the **real** value.

[LDF 2003.06.13.] Changed *pt0.epsilon*( ) to **Point** :: *epsilon*( ).

[LDF 2003.06.14.] Added error handling code for the case that *get_plane*( ) fails.

[LDF 2003.06.14.] No longer taking absolute value of the **real** value *r0* returned by **Plane** :: *get_distance*( ), since it will always be positive, anyway. Comment at place below, where I made this change.

[LDF 2003.07.01.] Added argument *suppress_warnings*.

[LDF 2003.07.16.] Changed name of *center* argument to *ref_pt*, because I've made *center* a data member of **Reg_Cl_Plane_Curve**.

⟨ Declare **Reg_Cl_Plane_Curve** functions 987 ⟩ +≡

virtual signed short *location*(**Point** *ref_pt*, **Point** *pt0*, **const bool** *suppress_warnings* = *false*) **const**;

993.

⟨ Define **Reg_Cl_Plane_Curve** functions 993 ⟩ ≡

```
signed short Reg_Cl_Plane_Curve::location(Point ref_pt, Point pt0, const bool suppress_warnings)
        const
  {
    bool DEBUG = false;     /* true */
    if (DEBUG) {
      cout << "Entering Reg_Cl_Plane_Curve::location()\n";
      ref_pt.show("ref_pt");
      pt0.show("pt0");
    }
    unsigned short orientation;
    const unsigned short X_Y = 0;
    const unsigned short X_Z = 1;
    const unsigned short Z_Y = 2;
    Plane pl(get_plane());
    if (pl ≡ INVALID_PLANE)     /* LDF 2003.06.14. Added this error handling code. */
    {
      cerr << "ERROR! In Reg_Cl_Plane_Curve::location():" <<
          "The Reg_Cl_Plane_Curve is non-planar.\n" << "Returning -6\n\n" << flush;
      return −6;
    }
    real r0 = pl.get_distance(pt0).first;
    if (r0 > Point::epsilon())
      /* LDF 2003.06.14. Changed. r0 will always be positive, so I now longer take its absolute value. */
    {
      if (¬suppress_warnings) cerr << "WARNING! In Reg_Cl_Plane_Curve::location().\n" <<
            "Point is not in plane of regular closed plane curve.\n" <<
            "Returning -2.\n\n" << flush;
      return −2;
    }
    Reg_Cl_Plane_Curve copy(*this);
    if (ref_pt ≠ origin) pt0 *= ref_pt *= copy.shift(−ref_pt);     /* LDF 2002.11.05. Simplified. */
    Point copy_normal = copy.get_normal();
    if (DEBUG) copy_normal.show("copy_normal");
    if (copy_normal.magnitude() ≡ 0) {
      cerr << "ERROR! In Reg_Cl_Plane_Curve::location().\n" <<
          "Normal has no magnitude. Returning -4\n\n" << flush;
      return −4;
    }
    else if (fabs(copy_normal.get_x()) < Point::epsilon() ∧ fabs(copy_normal.get_y()) < Point::epsilon())
      {
      if (DEBUG) cout << "Regular closed plane curve is already in x-y plane.\n";
      orientation = X_Y;
    }
    else if (fabs(copy_normal.get_x()) < Point::epsilon() ∧ fabs(copy_normal.get_z()) < Point::epsilon())
      {
      if (DEBUG) cout << "Regular closed plane curve is already in x-z plane.\n";
      orientation = X_Z;
    }
```

```
else if (fabs(copy_normal.get_z()) < Point::epsilon() ∧ fabs(copy_normal.get_y()) < Point::epsilon())
  {
  if (DEBUG) cout ≪ "Regular␣closed␣plane␣curve␣is␣already␣in␣z-y␣plane.\n";
  orientation = Z_Y;
}
else {
  if (DEBUG) cout ≪ "Putting␣regular␣closed␣plane␣curve␣into␣x-z␣plane.\n";
  Transform t1;
  t1.align_with_axis(ref_pt, copy.get_point(0), 'x');
  copy *= ref_pt *= pt0 *= t1;
  Point pt_c4(copy.get_point(number_of_points/4));
  Transform t2;
  t2.align_with_axis(ref_pt, pt_c4, 'z');
  copy *= ref_pt *= pt0 *= t2;
  orientation = X_Z;
}
if (DEBUG) {
  cout ≪ "orientation␣==␣" ≪ orientation ≪ endl ≪ flush;
  copy.draw(Colors::blue);
  copy.show("copy");
  ref_pt.dotlabel("C");
  pt0.dotlabel("pt0");
  pt0.show("pt0");
}
Transform t3;
Point pt1 = copy.get_point(0);
pt1 -= ref_pt;
pt1.unit_vector(true);
Point x_axis(1, 0, 0);
Point z_axis(0, 0, 1);
if (orientation ≡ X_Y ∨ orientation ≡ X_Z) {
  if (pt1 ≠ x_axis ∧ pt1 ≠ −x_axis) {
    t3.align_with_axis(ref_pt, pt1, 'x');
    copy *= ref_pt *= pt0 *= t3;
  }
}
else if (orientation ≡ Z_Y) {
  if (pt1 ≠ z_axis ∧ pt1 ≠ −z_axis) {
    t3.align_with_axis(ref_pt, pt1, 'z');
    copy *= t3;
    ref_pt *= t3;
    pt0 *= t3;
  }
}
real_pair rr;
real pt0_v;    /* [LDF 2002.11.05.] Vertical. */
real pt0_h;    /* [LDF 2002.11.05.] Horizontal. */
if (orientation ≡ X_Y) {
  pt0_h = pt0.get_x();
  pt0_v = pt0.get_y();
```

```
}
else if (orientation ≡ X_Z) {
  pt0_h = pt0.get_x();
  pt0_v = pt0.get_z();
}
else if (orientation ≡ Z_Y) {
  pt0_h = pt0.get_z();
  pt0_v = pt0.get_y();
}
else {
  cerr ≪ "ERROR!␣In␣Reg_Cl_Plane_Curve::location().\n" ≪
      "orientation␣has␣invalid␣value:␣" ≪ orientation ≪ "\nReturning␣-5\n\n" ≪ flush;
  return −5;
}
rr = solve('v', pt0_h);    /* [LDF 2002.11.05.] TO DO: Explain. */
if (rr.first ≡ INVALID_REAL ∧ rr.second ≡ INVALID_REAL) {
  if (DEBUG) {
    cout ≪ "Point␣is␣outside␣regular␣closed␣plane␣curve.␣" ≪ "Returning␣-1.\n";
    cout ≪ "Exiting␣Reg_Cl_Plane_Curve::location()\n";
  }
  return −1;
}
else if ((fabs(fabs(pt0_v) − fabs(rr.first)) < Point::epsilon()) ∨ (fabs(fabs(pt0_v) − fabs(rr.second)) <
      Point::epsilon())) {
  if (DEBUG) {
    cout ≪ "Point␣is␣on␣regular␣closed␣plane␣curve.␣" ≪ "Returning␣0.\n" ≪
        "Exiting␣Reg_Cl_Plane_Curve::location()\n";
    getchar();
  }
  return 0;
}
else if (fabs(pt0_v) < fabs(rr.first)) {
  if (DEBUG) {
    cout ≪ "Point␣is␣inside␣regular␣closed␣plane␣curve.␣" ≪ "Returning␣1.\n";
    cout ≪ "Exiting␣Reg_Cl_Plane_Curve::location()\n";
    getchar();
  }
  return 1;
}
else if (fabs(pt0_v) > fabs(rr.first))
  /* This case should never occur, I believe. [LDF 2002.11.05.] Why not?? */
{
  if (DEBUG) {
    cout ≪ "Point␣is␣outside␣regular␣closed␣plane␣curve.␣" ≪ "Returning␣-1.\n" ≪
        "Exiting␣Reg_Cl_Plane_Curve::location()\n";
    getchar();
  }
  return −1;
}
else {
  cerr ≪ "ERROR!␣In␣Reg_Cl_Plane_Curve::location().\n" ≪
      "This␣can't␣happen!␣Returning␣-3.\n" ≪ flush;
```

```
      getchar();
      if (DEBUG) cout << "Exiting␣Reg_Cl_Plane_Curve::location()\n";
      return -3;
    }
  }
```

See also sections 995, 998, 999, 1000, 1001, 1002, 1003, 1004, 1005, 1006, 1007, 1009, and 1012.

This code is used in section 1015.

994. Angle point. [LDF 2003.01.05.] TO DO: Find out why this function isn't **const**!

```
⟨ Declare Reg_Cl_Plane_Curve functions 987 ⟩ +≡
  virtual Point angle_point(real angle);
```

995.

⟨ Define **Reg_Cl_Plane_Curve** functions 993 ⟩ +≡

```
Point Reg_Cl_Plane_Curve::angle_point(real angle)
{
  cerr << "ERROR!␣In␣Reg_Cl_Plane_Curve::angle_point().\n" <<
      "This␣virtual␣function␣doesn't␣have␣a␣real␣definition␣for␣" <<
      "ordinary␣Reg_Cl_Plane_Curves.\nReturning␣INVALID_POINT.\n\n" << flush;
  return INVALID_POINT;
}
```

996. Intersection points. [LDF 2002.11.05.] Intersection with a line. *intersection_points*( ) returns a **bool_point_pair**. The **bool** in a **bool_point** indicates whether the **Point** is on the line segment in question. If one of the **bools** is *false*, but the **Point** is not ≡ INVALID_POINT, then the *line* (as opposed to the segment) does intersect the **Reg_Cl_Plane_Curve**, and the **Point** indicates one of the intersection points. So, *intersection_points*( ) can be used whether you want to restrict the intersection points to ones that are actually on a particular line segment or not.

TO DO: [LDF 2002.04.12.] In the specializations, I should check whether the intersection points are on the curve in question. I should also write a version of this for **Lines**, where there's no test for whether the intersection points are on the segment, since there's no segment.

The versions of *intersection_points*( ) belonging to classes derived from **Reg_Cl_Plane_Curve** will most likely call on the functions described in this section, passing *center* as the *ref_pt* argument. However, this need not be the case, and **Reg_Cl_Plane_Curves** need not have a meaningful *center*. *ref_pt* in these functions merely refers to the **Point** which should be placed at the origin by the transformation. [LDF 2003.07.16.]

997. Point arguments.

Log

[LDF 2003.06.20.] Rewrote this function. The perpendicular and non-parallel, non-coplanar cases are handled in exactly the same way. In these cases, there can only be one intersection point. **Plane**::*intersection_point*( ) and **Reg_Cl_Plane_Curve**::*location*( ) are now used to find it, if it exists.

[LDF 2003.06.20.] The coplanar case was the one that was causing difficulty. The copy of ∗*this* is now always put into the x-z plane, even if it is in one of the major planes, or in a plane parallel to one of these. The advantage of this, is that it simplifies the code. The disadvantage is, that additional rotations reduce the accuracy of the calculation of the intersection points.

[LDF 2003.06.20.] **Transform**::*align_with_axis*( ) is no longer used for putting the copy of ∗*this* and the line into the x-z plane. It might be possible to use it, but I used **Point**::*angle*( ) while debugging, in order to see what was happening better. It might be possible to go back to using **Transform**::*align_with_axis*, but I don't see any advantage to doing so.

[LDF 2003.06.20.] I've tested this function for coplanar lines for planes with various orientations. I hope that it works properly for all planes now!

[LDF 2003.07.01.] Added *true* as *silent* argument to *unit_vector*( ) when I call it on *cross*. This prevents *unit_vector*( ) from issuing a warning message, when *cross* has magnitude 0, which occurs when *surface_vector* and *pt_vector* are colinear. Since this case is handled correctly, the warning messages are unnecessary and distracting.

[LDF 2003.07.01.] BUG FIX: Made changes to the way *on_segment* is used in the coplanar case. The way it was before handled certain cases wrong.

[LDF 2003.07.04.] Removed unreachable statement at end of function: **return** *bpp*. GCC didn't complain, but the DEC compiler issued a warning.

[LDF 2003.07.16.] Changed name of *center* argument to *ref_pt*, because I've made *center* a data member of **Reg_Cl_Plane_Curve**.

⟨ Declare **Reg_Cl_Plane_Curve** functions 987 ⟩ +≡

```
virtual bool_point_pair intersection_points(Point ref_pt, Point p0, Point p1) const;
```

998.

⟨ Define **Reg_Cl_Plane_Curve** functions 993 ⟩ +≡

```
bool_point_pair Reg_Cl_Plane_Curve::intersection_points(Point ref_pt, Point pt0, Point pt1)
        const{ bool DEBUG = false;    /* true */
    if (DEBUG)
      cout << "␣***␣Entering␣Reg_Cl_Plane_Curve::" << "intersection_points()\n" << flush;
    bool_point_pair bpp = INVALID_BOOL_POINT_PAIR;
    Plane pl = get_plane();
    if (DEBUG) pl.normal.show("pl.normal");
    Point surface_vector = (get_point(0) − ref_pt);
    surface_vector.unit_vector(true);
    Point pt_vector(pt1 − pt0);
    pt_vector.unit_vector(true);
    Point cross = surface_vector.cross_product(pt_vector);
    cross.unit_vector(true, true);
    if (DEBUG) {
      surface_vector.show("surface_vector");
      pt_vector.show("pt_vector");
      cross.show("cross");
      pl.normal.show("pl.normal");
    }
    short distance = pl.get_distance(pt0).second;
```

999. Degenerate cases, error handling.

⟨ Define **Reg_Cl_Plane_Curve** functions 993 ⟩ +≡

```
if (pt_vector ≡ INVALID_POINT ∨ pl.normal ≡ INVALID_POINT ∨ pt_vector ≡ origin ∨ pl.normal ≡ origin)
  {
  cerr << "ERROR!␣In␣Reg_Cl_Plane_Curve:intersection_points():\n" <<
      "Something␣is␣wrong␣with␣the␣normals:\n";
  pt_vector.show("pt_vector:");
  pl.normal.show("pl.normal");
  cerr << "Returning␣INVALID_BOOL_POINT_PAIR.\n\n" << flush;
  if (DEBUG) cout << "Exiting␣Polygon::intersection_points().\n\n" << flush;
  return INVALID_BOOL_POINT_PAIR;
}
```

1000. Parallel and coplanar cases.

⟨ Define **Reg_Cl_Plane_Curve** functions 993 ⟩ +≡

```
else if (surface_vector ≡ pt_vector ∨ surface_vector ≡ −pt_vector ∨ cross ≡ pl.normal ∨ cross ≡ −pl.normal)
    {
```

1001. Coplanar case.

⟨ Define **Reg_Cl_Plane_Curve** functions 993 ⟩ +≡

```
if (distance ≡ 0) {
if (DEBUG) cout ≪ "Line␣and␣Reg_Cl_Plane_Curve␣are␣coplanar.\n";
Transform t0;
Reg_Cl_Plane_Curve copy(*this);
Point curve_0 = copy.get_point(0);
if (DEBUG) {
  cout ≪ "After␣copying:" ≪ endl;
  show("this:");
  copy.show("copy:");
  curve_0.show("curve_0");
}
t0 *= curve_0 *= pt0 *= pt1 *= copy.shift(−ref_pt);
if (DEBUG) {
  cout ≪ "After␣shift:" ≪ endl;
  copy.show("copy:");
  curve_0.show("curve_0");
  pt0.show("pt0");
  pt1.show("pt1");
}
if (curve_0.get_x() < 0) {
  t0 *= curve_0 *= pt0 *= pt1 *= copy.rotate(0, 0, 180);
}
if (DEBUG) {
  cout ≪ "After␣rotating␣so␣curve_0␣has␣positive␣x:" ≪ endl;
  copy.show("copy:");
  curve_0.show("curve_0");
  pt0.show("pt0");
  pt1.show("pt1");
}
Point trace_x_z_0 = curve_0;
trace_x_z_0.shift(0, −(curve_0.get_y()));
Point x_axis_pt(1);
real ang = trace_x_z_0.angle(x_axis_pt);
if (DEBUG) {
  cout ≪ "ang␣==␣" ≪ ang ≪ endl ≪ flush;
}
if (ang ≠ 0) {
  if (curve_0.get_z() > 0) ang *= −1;
  t0 *= curve_0 *= pt0 *= pt1 *= copy.rotate(0, ang);
}
if (DEBUG) {
  cout ≪ "After␣rotating␣so␣the␣trace␣of␣curve_0␣is␣on␣x-axis:" ≪ endl;
  copy.show("copy:");
  curve_0.show("curve_0");
  pt0.show("pt0");
  pt1.show("pt1");
}
ang = curve_0.angle(x_axis_pt);
```

```
  if (ang ≠ 0) {
    if (curve_0.get_y() > 0) ang *= -1;
    t0 *= curve_0 *= pt0 *= pt1 *= copy.rotate(0, 0, ang);
  }
  if (DEBUG) {
    cout << "After␣rotating␣so␣curve_0␣is␣on␣x-axis:" << endl;
    copy.show("copy:");
    pt0.show("pt0");
    pt1.show("pt1");
  }
  Point curve_4 = copy.get_point(number_of_points/4);
  Point z_axis_pt(0, 0, 1);
  ang = curve_4.angle(z_axis_pt);
  if (DEBUG) {
    curve_4.show("curve_4");
    cout << "ang␣==␣" << ang << endl << flush;
  }
  if (ang ≠ 0) {
    if (curve_4.get_y() > 0) ang *= -1;
    t0 *= curve_4 *= pt0 *= pt1 *= copy.rotate(ang);
  }
  if (DEBUG) {
    cout << "After␣rotating␣so␣curve_4␣is␣on␣z-axis:" << endl;
    copy.show("copy:");
    curve_4.show("curve_4");
    pt0.show("pt0");
    pt1.show("pt1");
  }
  if (DEBUG) {
    copy.draw(Colors::blue);
    pt0.draw(pt1, Colors::black, "evenly");
    for (int i = 0; i < 16; i += 4) copy.get_point(i).dotlabel(i);
#if 0
    pt0.dotlabel("pt0");
    pt1.dotlabel("pt1");
    copy.get_center().dotlabel("copy␣center");
#endif
    draw_axes();
  }
  real pt0_h = pt0.get_x();
  real pt0_v = pt0.get_z();
  real pt1_h = pt1.get_x();
  real pt1_v = pt1.get_z();    /* pt1_v isn't used. Leaving it here, just in case. [LDF 2003.08.27.] */
  real Slope = pt1.slope(pt0, 'x', 'z');
  if (DEBUG) cout << "Slope␣==␣" << Slope << endl << flush;
  real_pair rr;    /* BEGIN */
```

1002. Slope is 0 (line is horizontal).

⟨ Define **Reg_Cl_Plane_Curve** functions 993 ⟩ +≡

```
if (Slope ≡ 0)     /* v is known, h is unknown. */
{
  if (DEBUG) {
    cout ≪ "Slope␣==␣0" ≪ endl ≪ flush;
  }
  rr = solve('h', pt0_v);
  if (rr.first ≠ INVALID_REAL) {
    bpp.first.pt.set(rr.first, 0, pt0_v);
  }
  else bpp.first.pt = INVALID_POINT;
  if (rr.second ≠ INVALID_REAL) {
    bpp.second.pt.set(rr.second, 0, pt0_v);
  }
  else bpp.second.pt = INVALID_POINT;
  if (DEBUG) {
    bpp.first.pt.show("bpp.first.pt");
    bpp.second.pt.show("bpp.second.pt");
  }
}     /* End Slope ≡ 0. */
```

1003. Slope is undefined (line is vertical).

⟨ Define **Reg_Cl_Plane_Curve** functions 993 ⟩ +≡

```
else
  if (Slope ≡ INVALID_REAL) {
    if (DEBUG) {
      cout ≪ "Line␣is␣vertical.\n";
    }
    rr = solve('v', pt1_h);
    if (rr.first ≠ INVALID_REAL) {
      bpp.first.pt.set(pt0_h, 0, rr.first);
    }
    else bpp.first.pt = INVALID_POINT;
    if (rr.second ≠ INVALID_REAL) {
      bpp.second.pt.set(pt0_h, 0, rr.second);
    }
    else bpp.second.pt = INVALID_POINT;
  }     /* End Slope ≡ INVALID_REAL. */
```

1004. Slope ∈ **real** is defined and ≠ 0.

⟨ Define **Reg_Cl_Plane_Curve** functions 993 ⟩ +≡

```
else {
  real v_intercept;
  v_intercept = pt0_v − (Slope ∗ pt0_h);
  if (DEBUG) cout ≪ "v_intercept␣==␣" ≪ v_intercept ≪ endl ≪ flush;
  real_triple coeffs = get_coefficients(Slope, v_intercept);    /∗ New h-values. ∗/
  if (is_quadratic()) {
    if (DEBUG) {
      cout ≪ "Solving␣quadratic.\n" ≪ flush;
    }
    rr = solve_quadratic(coeffs.first, coeffs.second, coeffs.third);
  }
  else {
    cout ≪ "Not␣a␣quadratic.␣" ≪ "Haven't␣programmed␣this␣case␣yet.\n" ≪ flush;
  }
  real v_coord;
  if (rr.first ≠ INVALID_REAL) {
    v_coord = (Slope ∗ rr.first) + v_intercept;
    bpp.first.pt.set(rr.first, 0, v_coord);
  }
  else bpp.first.pt = INVALID_POINT;
  if (rr.second ≠ INVALID_REAL) {
    v_coord = (Slope ∗ rr.second) + v_intercept;
    bpp.second.pt.set(rr.second, 0, v_coord);
  }
  else bpp.second.pt = INVALID_POINT;
}
```

1005. Common code for the "coplanar" case.

⟨ Define **Reg_Cl_Plane_Curve** functions 993 ⟩ +≡

```
bool_real on_segment;
if (bpp.first.pt ≡ INVALID_POINT) {
  on_segment.first = false;
  on_segment.second = INVALID_REAL;
}
else on_segment = bpp.first.pt.is_on_segment(pt0, pt1);
if (DEBUG) {
  cout ≪ "on_segment.first␣==␣" ≪ on_segment.first ≪ endl ≪ flush;
  cout ≪ "on_segment.second␣==␣" ≪ on_segment.second ≪ endl ≪ flush;
}
if (on_segment.first ≡ true) bpp.first.b = true;
else bpp.first.b = false;
Transform t_inverse;
t_inverse = t0.inverse();
if (bpp.first.pt ≠ INVALID_POINT) {
  if (DEBUG) cout ≪ "Transforming␣bpp.first.pt\n" ≪ flush;
  bpp.first.pt *= t_inverse;
}
else {
  if (DEBUG) cout ≪ "bpp.first.pt␣is␣invalid\n" ≪ flush;
}
if (bpp.second.pt ≡ INVALID_POINT) {
  on_segment.first = false;
  on_segment.second = INVALID_REAL;
}
else on_segment = bpp.second.pt.is_on_segment(pt0, pt1);
if (on_segment.first ≡ true) bpp.second.b = true;
else bpp.second.b = false;
if (DEBUG) {
  cout ≪ "on_segment.first␣==␣" ≪ on_segment.first ≪ endl ≪ flush;
  cout ≪ "on_segment.second␣==␣" ≪ on_segment.second ≪ endl ≪ flush;
}
if (bpp.second.pt ≠ INVALID_POINT) {
  if (DEBUG) cout ≪ "Transforming␣bpp.second.pt\n" ≪ flush;
  bpp.second.pt *= t_inverse;
}
else {
  bpp.second.pt = INVALID_POINT;
  if (DEBUG) cout ≪ "bpp.second.pt␣is␣invalid\n" ≪ flush;
}
if (DEBUG) {
  cout ≪ "rr.first␣==␣" ≪ rr.first ≪ endl ≪ flush;
  cout ≪ "rr.second␣==␣" ≪ rr.second ≪ endl ≪ flush;
  cout ≪ "bpp.first.b␣==␣" ≪ bpp.first.b ≪ endl ≪ flush;
  cout ≪ "bpp.second.b␣==␣" ≪ bpp.second.b ≪ endl ≪ flush;
  bpp.first.pt.show("bpp.first.pt");
  bpp.second.pt.show("bpp.second.pt");
}
return bpp; }   /* End of coplanar case. */
```

1006. Parallel case.

⟨ Define **Reg_Cl_Plane_Curve** functions 993 ⟩ +≡

```
else {
  cerr ≪ "WARNING!␣In␣Reg_Cl_Plane_Curve::intersection_points():\n" ≪
      "Line␣and␣Reg_Cl_Plane_Curve␣are␣in␣parallel␣planes.\n" ≪
      "No␣intersections.␣Returning␣INVALID_BOOL_POINT_PAIR." ≪ endl ≪ endl ≪ flush;
  return INVALID_BOOL_POINT_PAIR;
}
}     /* End of parallel and coplanar cases. */
```

1007. Perpendicular and non-coplanar cases. [LDF 2003.06.13.] These cases are handled in exactly the same way.

⟨ Define **Reg_Cl_Plane_Curve** functions 993 ⟩ +≡

```
else {
  if (pl.normal ≡ pt_vector ∨ pl.normal ≡ −pt_vector) {
    if (DEBUG)
      cout ≪ "The␣line␣is␣perpendicular␣to␣the␣" ≪ "Reg_Cl_Plane_Curve.\n" ≪ flush;
  }
  else {
    if (DEBUG)
      cout ≪ "The␣line␣and␣the␣Reg_Cl_Plane_Curve␣" ≪ "are␣non-coplanar.\n" ≪ flush;
  }
  bool_point bp = pl.intersection_point(pt0, pt1);
  if (DEBUG) {
    bp.pt.show("bp.pt");
  }
  short s = location(ref_pt, bp.pt);
  if (DEBUG) cout ≪ "location:␣s␣==␣" ≪ s ≪ endl ≪ flush;
  if (s > −1) {
    bpp.first.pt = bp.pt;
    bpp.first.b = bp.pt.is_on_segment(pt0, pt1).first;
    if (DEBUG) cout ≪ "On␣segment:␣==␣" ≪ bpp.first.b ≪ endl ≪ flush;
    return bpp;
  }
  else return INVALID_BOOL_POINT_PAIR;
}     /* End of "Perpendicular and non-coplanar cases". */
}
```

1008. Path arguments.

Log

[LDF 2003.06.20.] Added this function.

[LDF 2003.07.16.] Changed name of *center* argument to *ref_pt*, because I've made *center* a data member of **Reg_Cl_Plane_Curve**.

⟨ Declare **Reg_Cl_Plane_Curve** functions 987 ⟩ +≡

```
bool_point_pair intersection_points(const Point &ref_pt, const Path &p) const;
```

1009.

⟨ Define **Reg_Cl_Plane_Curve** functions 993 ⟩ +≡

```
bool_point_pair Reg_Cl_Plane_Curve::intersection_points(const Point &ref_pt, const Path &p)
        const
{
  if (¬p.is_linear()) {
    cerr ≪ "ERROR!␣In␣Reg_Cl_Plane_Curve::intersection_points():\n" ≪
        "Path␣argument␣p␣is␣non-linear.␣" ≪ "Returning␣INVALID_BOOL_POINT_PAIR.\n\n" ≪
        flush;
    return INVALID_BOOL_POINT_PAIR;
  }
  return intersection_points(ref_pt, p.get_point(0), p.get_last_point());
}
```

1010. Reg_Cl_Plane_Curve segments. The functions in this section require that the **Reg_Cl_Plane_Curve** have a meaningful *center*, in order to make it possible to rotate the segments. [LDF 2003.07.16.]

Log

[LDF 2003.07.16.] Added this section and its subsections, including the declarations and definitions of *segment*( ), *half* ( ), and *quarter*( ). They were formerly members of **Circle**.

1011. Segment. [LDF 2002.11.10.] *segment*( ) returns a subpath of the **Reg_Cl_Plane_Curve** representing a segment of ∗*this*.

int *factor* Determines how large a segment of the **Reg_Cl_Plane_Curve** is returned. *factor* must be > 1 and less than or equal to the number of points on the **Reg_Cl_Plane_Curve**.

real *angle* Optional, with 0 as the default. If*angle* is $\neq 0$, a **Point** is found in the direction of the normal to the **Reg_Cl_Plane_Curve** from the center of the **Reg_Cl_Plane_Curve**, and the segment is rotated around the center and this **Point**.

bool *closed* If *true*, the **Path** is made a "**cycle**" and the ends of the segment are joined by concatenating the curved **Path** with the straight line segment from its last to its first **Point** using the connector `"&"`.

[LDF 2003.07.27.] TO DO: Make arguments **const**, if possible. *angle* can't be, though. If *factor* ≡ *number_of_points*, return ∗*this*, cast to a **Path**, with warning.

Log

[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 2003.05.20.] Changed the way the last connector is set when *closed* ≡ *true*.

[LDF 2003.07.27.] Made **const**.

[LDF 2003.08.20.] BUG FIX: Added **unsigned short** *subpath_size*. Changed the way the subpath is created, when *closed* is *true*. Now concatenating the curved subpath with the straight line segment from the last to the first **Points** of the subpath using `"&"`.

⟨ Declare **Reg_Cl_Plane_Curve** functions 987 ⟩ +≡

```
Path segment(unsigned int factor, real angle = 0, bool closed = true) const;
```

1012.

⟨ Define **Reg_Cl_Plane_Curve** functions 993 ⟩ +≡

```
Path Reg_Cl_Plane_Curve::segment(unsigned int factor, real angle, bool closed) const
{
  Path p;
  if (factor ≤ 1 ∨ factor > number_of_points) {
    cerr ≪ "ERROR!␣In␣Reg_Cl_Plane_Curve::segment():\n" ≪
        "The␣argument␣factor␣has␣an␣invalid␣value:\n" ≪ factor ≪
        "\nReturning␣empty␣Path.\n\n";
    return p;
  }
  if (fabs(angle) > 360) {
    cerr ≪ "WARNING!␣In␣Reg_Cl_Plane_Curve::segment():\n" ≪
        "The␣argument␣angle␣is␣greater␣than␣360:␣" ≪ angle ≪ endl ≪
        "It␣will␣be␣reduced.\n\n" ≪ flush;
  }
  unsigned short subpath_size = (number_of_points/factor) + 1;
  p = subpath(0, subpath_size, false, "..");
  for (unsigned short i = 1; i < subpath_size − 1; ++i) p += "..";
  if (closed) {
    p += "&";
    p += p.get_last_point();
    p += "--";
    p += p.get_point(0);
    p += "&";
    p.set_cycle();
  }
  angle = fmod(angle, 360);
  if (angle ≠ 0) {
    Point normal = get_normal();
    normal.shift(center);
    p.rotate(center, normal, angle);
  }
  return p;
}
```

1013. Half. *half*( ) creates a curve using half of the points in *points* starting from point 0. If the argument *angle* is not zero, the resulting **Path** is rotated by that amount about a line from *center* in the direction of the normal to the **Reg_Cl_Plane_Curve**. If the argument *closed* is *true*, then the segment is closed and can be filled using *fill*( ) or *filldraw*( ).

⟨ Declare **Reg_Cl_Plane_Curve** functions 987 ⟩ +≡

```
inline Path half(real angle = 0, bool closed = true) const
{
  return segment(2, angle, closed);
}
```

1014. Quarter. *quarter*( ) creates a curve using a quarter of the points in *points* starting from point 0. If the argument *angle* is not zero, the resulting **Path** is rotated by that amount about a line from *center* in the direction of the normal to the **Reg_Cl_Plane_Curve**. If the argument *closed* is *true*, then the segment is closed and can be filled using *fill*( ) or *filldraw*( ).

⟨ Declare **Reg_Cl_Plane_Curve** functions 987 ⟩ +≡

```
inline Path quarter(real angle = 0, bool closed = true) const
{
  return segment(4, angle, closed);
}
```

1015. Putting Reg_Cl_Plane_Curve together.

This is what's compiled.

⟨ Include files 6 ⟩
⟨ Version control identifier 5 ⟩
⟨ Define **class Reg_Cl_Plane_Curve** 985 ⟩
⟨ Define **Reg_Cl_Plane_Curve** functions 993 ⟩

1016. This is what's written to `curves.h`.

⟨ `curves.h` 1016 ⟩ ≡
⟨ Define **class Reg_Cl_Plane_Curve** 985 ⟩

1017. Polygon (`polygons.web`).

Log

[LDF 2003.07.18.] Removed the transformation sections from **Reg_Polygon**, and made them members of **Polygon**. Also, removed the **Rectangle** versions in `rectangs.web`.

[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.

---

format *Polygon* *Path*

⟨ Version control identifier 5 ⟩ +≡
static string *rcs_id* = "`$Id:␣polygons.web,v␣1.4␣2004/01/12␣21:32:08␣lfinsto1␣Exp␣$`";

1018. 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"
#include "points.h"
#include "lines.h"
#include "planes.h"
#include "paths.h"
#include "curves.h"
```

1019. Polygon class definition. LDF Undated. **Polygon** is derived from **Path**. This makes sense, because a **Polygon** is really just a kind of **Path**. This way, we don't have to define the drawing and filling functions, or the transformations.

[LDF 2003.06.06.] **Polygon** is meant to be used primarily as a base class for more specialized types of polygons. Currently, **Reg_Polygon** and **Rectangle** are defined. I've added **Polygon** so that I can define intersection functions that will work for both **Reg_Polygon** and **Rectangle**.

Log

[LDF 2003.06.06.] Added **class Polygon**.

⟨ Define **class Polygon** 1019 ⟩ ≡
class Polygon : **public Path** {
protected: **Point** *center*;
public: ⟨ Declare **Polygon** functions 1022 ⟩
};

This code is used in sections 1097 and 1098.

1020. Returning elements and information.

1021. Get center.

Log

[LDF 2003.07.18.] Moved these functions from **Reg_Polygon** to **Polygon**. Also removed the **Rectangle** versions, since **Rectangle** inherits the **Polygon** versions.

1022. non-const version.

Log

[LDF 2002.04.24.] Added this function.
[LDF 2003.05.09.] Changed return value from **Point** & to **const Point** &.

⟨ Declare **Polygon** functions 1022 ⟩ ≡
  virtual const Point &*get_center*( );

See also sections 1024, 1028, 1037, 1039, 1045, 1047, 1050, 1052, 1054, 1056, 1059, 1061, 1064, and 1066.

This code is used in section 1019.

1023.

⟨ Define **Polygon** functions 1023 ⟩ ≡
```
const Point &Polygon::get_center()
{
  if (points.size() ≡ 0)   /* LDF 2002.09.27. Added this error handling code. If the Polygon is
      empty, don't return center. */
  {
    cerr ≪ "WARNING!␣In␣Polygon::get_center():\n" ≪ "Polygon␣doesn't␣con\
        tain␣any␣Points,␣" ≪ "so␣it␣presumably␣doesn't␣have␣a␣center.\n" ≪
        "Returning␣INVALID_POINT.\n\n" ≪ flush;
    return const_cast⟨Point &⟩(INVALID_POINT);
  }
  center.apply_transform();
  return center;
}
```

See also sections 1025, 1029, 1030, 1031, 1032, 1033, 1034, 1035, 1036, 1038, 1040, 1041, 1042, 1043, 1046, 1048, 1051, 1053, 1055, 1057, 1060, 1062, 1065, and 1067.

This code is used in section 1097.

1024. const version. [LDF 2002.09.27.] Note that this version returns a **Point** whereas the non-**const** version returns a **Point** &. That's because p is a local variable in this function and it would be an error to return a reference to it. [LDF 2002.04.24.] Added this function.

⟨ Declare **Polygon** functions 1022 ⟩ +≡
  Point *get_center*( ) **const**;

1025.

⟨ Define **Polygon** functions 1023 ⟩ +≡

```
Point Polygon::get_center() const
{
  if (points.size() ≡ 0)    /* LDF 2002.09.27. Added this error handling code. If the Polygon is
        empty, don't return center. */
  {
    cerr ≪ "WARNING!␣In␣Polygon::get_center():\n" ≪ "Polygon␣doesn't␣con\
        tain␣any␣Points,␣" ≪ "so␣it␣presumably␣doesn't␣have␣a␣center.\n" ≪
        "Returning␣INVALID_POINT.\n\n" ≪ flush;
    return const_cast⟨Point &⟩(INVALID_POINT);
  }
  Point p(center);
  p.apply_transform();
  return p;
}
```

1026. Intersections.

1027. Intersection with a line. [LDF 2003.06.13.] A line can intersect with a **Polygon** at two points at most.

1028. Point version.

Log

[LDF 2003.06.13.] Added this function.

[LDF 2003.06.17.] Minor change. Now using *get_point*(0) and *center* instead of *get_point*(0) and *get_last_point*( ) to generate *surface_vector*.

⟨ Declare **Polygon** functions 1022 ⟩ +≡

```
bool_point_pair intersection_points(const Point &pt0, const Point &pt1) const;
```

1029.

⟨ Define **Polygon** functions 1023 ⟩ +≡

```
bool_point_pair Polygon::intersection_points(const Point &pt0, const Point &pt1) const{ bool
        DEBUG = false;    /* true */
    if (DEBUG) cout << "Entering Polygon::intersection_points().\n" << flush;
    DEBUG = false;
    bool_point_pair bpp = INVALID_BOOL_POINT_PAIR;    /* The return value. [LDF 2003.06.13.] */
    Plane pl = get_plane();
    Point pt_vector(pt1 - pt0);
    Point surface_vector(get_point(0) - center);
    if (DEBUG) {
      pl.point.show("pl.point");
      pl.normal.show("pl.normal");
      pt0.show("pt0");
      pt1.show("pt1");
      pt_vector.show("pt_vector");
      surface_vector.show("surface_vector");
    }
    Point cross = surface_vector.cross_product(pt_vector);
    cross.unit_vector(true);
    if (DEBUG) {
      cross.show("cross");
      pl.normal.show("pl.normal");
    }
    short distance = pl.get_distance(pt0).second;
```

1030. Degenerate cases, error handling.

⟨ Define **Polygon** functions 1023 ⟩ +≡

```
if (pt_vector ≡ INVALID_POINT ∨ pl.normal ≡ INVALID_POINT ∨ pt_vector ≡ origin ∨ pl.normal ≡ origin)
  {
  cerr << "ERROR! In Polygon::intersection_points():\n" <<
      "Something is wrong with the normals:\n";
  pt_vector.show("pt_vector:");
  pl.normal.show("pl.normal");
  cerr << "Returning INVALID_BOOL_POINT_PAIR.\n\n" << flush;
  if (DEBUG) cout << "Exiting Polygon::intersection_points().\n\n" << flush;
  return INVALID_BOOL_POINT_PAIR;
}
```

1031. Parallel and coplanar cases.

Log

[LDF 2003.06.20.] Now checking *surface_vector* against *pt_vector* in the following conditional.

⟨ Define **Polygon** functions 1023 ⟩ +≡

```
else if (surface_vector ≡ pt_vector ∨ surface_vector ≡ −pt_vector ∨ cross ≡ pl.normal ∨ cross ≡ −pl.normal)
    {
```

1032. Coplanar case. [LDF 2003.06.13.] Only those intersection points that are on the line segments making of the **Polygon** are returned in *bpp*. If an intersection **Point** lies on both segments, the **bool** part of the **bool_point** will be *true*, otherwise *false*. If a **Point** lies on a line segment belonging to the **Polygon**, but not to the line segment $\overrightarrow{pt_0 pt_1}$, the **Point** will be put into the **bool_point**, but the **bool** will be *false*. The reason for this is, that the angles of the sides of the **Polygon** can cause intersection points to be found, that the user probably won't want.

```
⟨ Define Polygon functions 1023 ⟩ +≡
  if (distance ≡ 0) {
    if (DEBUG) cout ≪ "Line␣and␣Polygon␣are␣coplanar.\n";
    bool found = false;
    bool_point bp;
    Point q0;
    Point q1;
    if (DEBUG) show("this");
    for (vector⟨Point *⟩ :: const_iterator iter = points.begin(); iter ≠ points.end(); ++iter) {
      if ((iter + 1) ≡ points.end()) {
        if (DEBUG) cout ≪ "Doing␣last␣segment.\n";
        q0 = *(points.back());
        q1 = *(points.front());
      }
      else {
        if (DEBUG) cout ≪ "Doing␣normal␣segment.\n";
        q0 = **iter;
        q1 = **(iter + 1);
      }
      bp = Point :: intersection_point(pt0, pt1, q0, q1);
      if (bp.b)    /* Intersection point is on both segments. */
      {
        if (¬found) {
          bpp.first.b = true;
          bpp.first.pt = bp.pt;
          found = true;
          if (DEBUG) cout ≪ "Found␣first␣intersection.\n";
        }
        else {
          bpp.second.b = true;
          bpp.second.pt = bp.pt;
          if (DEBUG) cout ≪ "Found␣second␣intersection.␣Returning.\n";
          return bpp;
        }
      }
      else if (bp.pt ≠ INVALID_POINT ∧ bp.pt.is_on_segment(q0, q1).first) {
        if (¬found) {
          bpp.first.b = false;
          bpp.first.pt = bp.pt;
          found = true;
          if (DEBUG) cout ≪ "Found␣first␣intersection.\n";
        }
        else {
          bpp.second.b = false;
          bpp.second.pt = bp.pt;
```

```
          if (DEBUG) cout ≪ "Found␣second␣intersection.␣Returning.\n";
          return bpp;
        }
      }
      else continue;
    }    /* for */
    return bpp;
  }    /* End of coplanar case. */
```

1033. Parallel case.

```
⟨ Define Polygon functions 1023 ⟩ +≡
  else {
    cerr ≪ "WARNING!␣In␣Polygon::intersection_points():\n" ≪
        "Line␣and␣Polygon␣are␣in␣parallel␣planes.\n" ≪
        "No␣intersections.␣Returning␣INVALID_BOOL_POINT_PAIR." ≪ endl ≪ endl ≪ flush;
    return INVALID_BOOL_POINT_PAIR;
  }
  }    /* End of parallel and coplanar cases. */
```

1034. Perpendicular and non-coplanar cases. [LDF 2003.06.13.] These cases are handled in exactly the same way.

⟨ Define **Polygon** functions 1023 ⟩ +≡

```
else {
if (pl.normal ≡ pt_vector ∨ pl.normal ≡ −pt_vector) {
  if (DEBUG) cout ≪ "The␣line␣is␣perpendicular␣to␣the␣Polygon.\n" ≪ flush;
}
else {
  if (DEBUG) cout ≪ "The␣line␣and␣the␣Polygon␣are␣non-coplanar.\n" ≪ flush;
}
bool_point bp = pl.intersection_point(pt0, pt1);
if (DEBUG) {
  bp.pt.show("bp.pt");
}
if (bp.pt ≡ center) {
  if (DEBUG) {
    cout ≪ "bp.pt␣==␣center." ≪ endl ≪ flush;
  }
  bpp.first.pt = bp.pt;
  bpp.first.b = bp.pt.is_on_segment(pt0, pt1).first;
  return bpp;
}
Point r0;
Point r1;
if (DEBUG) {
  show("this:");
  center.show("center");
  cout ≪ "points.size()␣==␣" ≪ points.size() ≪ endl ≪ flush;
}
for (vector⟨Point *⟩::const_iterator iter = points.begin(); iter ≠ points.end(); ++iter) {
if ((iter + 1) ≡ points.end()) {
  r0 = *(points.back());
  r1 = *(points.front());
  if (DEBUG) {
    r0.show("r0");
    r1.show("r1");
  }
}
else {
  r0 = **iter;
  r1 = **(iter + 1);
}
if (bp.pt ≡ r0 ∨ bp.pt ≡ r1) {
  if (DEBUG) {
    if (bp.pt ≡ r0) cout ≪ "bp.pt␣==␣r0." ≪ endl ≪ flush;
    else if (bp.pt ≡ r1) cout ≪ "bp.pt␣==␣r1." ≪ endl ≪ flush;
  }
  bpp.first.pt = bp.pt;
  bpp.first.b = bp.pt.is_on_segment(pt0, pt1).first;
  return bpp;
}
```

1035. [LDF 2003.06.24.] `DEBUG` is passed as the *verbose* argument to *is_in_triangle*( ). So, if *intersection_points*( ) is being debugged, *is_in_triangle*( ) will print more information. However, `DEBUG` in *is_in_triangle*( ) will not be set to *true*.

⟨ Define **Polygon** functions 1023 ⟩ +≡

```
else
  if (bp.pt.is_in_triangle(center, r0, r1, DEBUG)) {
    if (DEBUG) cout ≪ "Intersection␣point␣is␣within␣triangle.\n";
    bpp.first.pt = bp.pt;
    bpp.first.b = bp.pt.is_on_segment(pt0, pt1).first;
    return bpp;
  }
} }     /* End of "Perpendicular and non-coplanar cases". */
```

1036. End of definition.

⟨ Define **Polygon** functions 1023 ⟩ +≡

```
if (DEBUG) cout ≪ "Exiting␣Polygon::intersection_points().\n\n" ≪ flush;
return bpp; }
```

1037. Path version.

⟨ Declare **Polygon** functions 1022 ⟩ +≡

```
bool_point_pair intersection_points(const Path &p) const;
```

1038.

⟨ Define **Polygon** functions 1023 ⟩ +≡

```
bool_point_pair Polygon::intersection_points(const Path &p) const
{
  if (¬p.is_linear()) {
    cerr ≪ "ERROR!␣In␣Polygon::intersection_points(const␣Path&␣p):\n" ≪
        "Path␣p␣is␣non-linear.␣␣Returning␣INVALID_BOOL_POINT_PAIR.\n\n" ≪ flush;
    return INVALID_BOOL_POINT_PAIR;
  }
  return intersection_points(p.get_point(0), p.get_last_point());
}
```

1039. Intersection with another Polygon. TO DO: Explain what this function does and how it works. [LDF 2003.06.29.]

TO DO: Find out where *unit_vector*( ) gets called, when this function is called, and try to pass *true* as its *silent* argument. [LDF 2003.07.16.]

Log

[LDF 2003.06.29.] Replaced the dummy definition of this function with a real one.

⟨ Declare **Polygon** functions 1022 ⟩ +≡

```
vector⟨Point⟩ intersection_points(const Polygon &r) const;
```

1040.

⟨ Define **Polygon** functions 1023 ⟩ +≡

```
vector⟨Point⟩ Polygon::intersection_points(const Polygon &r) const{ bool DEBUG = false;
      /* true */
    vector⟨Point⟩ v;
    Plane pl = get_plane();
    Plane r_pl = r.get_plane();
    if (DEBUG) {
      pl.normal.show("pl.normal");
      r_pl.normal.show("r_pl.normal");
      cout << "pl.distance␣==␣" << pl.distance << endl << flush;
      cout << "r_pl.distance␣==␣" << r_pl.distance << endl << flush;
    }
    real distance = fabs(fabs(pl.distance) − fabs(r_pl.distance));
    if (distance < Point::epsilon()) distance = 0;
    if (DEBUG) cout << "distance␣==␣" << distance << endl << flush;
    if (pl.normal ≡ r_pl.normal) {
```

1041. Coplanar case.

⟨ Define **Polygon** functions 1023 ⟩ +≡

```
if (distance ≡ 0) {
  if (DEBUG) cout << "Coplanar.\n";
  bool_point bp;
  Point *ptr;
  Point *r_ptr;
  for (vector⟨Point *⟩::const_iterator iter = points.begin(); iter ≠ points.end(); ++iter) {
    if ((iter + 1) ≡ points.end()) ptr = points.front();
    else ptr = *(iter + 1);
    for (vector⟨Point *⟩::const_iterator r_iter = r.points.begin(); r_iter ≠ r.points.end(); ++r_iter) {
      if ((r_iter + 1) ≡ r.points.end()) r_ptr = r.points.front();
      else r_ptr = *(r_iter + 1);
      bp = Point::intersection_point(**iter, *ptr, **r_iter, *r_ptr);
      if (bp.b) v.push_back(bp.pt);
    }   /* Inner for */
  }   /* Outer for */
  return v;
}
```

1042. Parallel case.

⟨ Define **Polygon** functions 1023 ⟩ +≡

```
else    /* Parallel. */
{
  cerr << "WARNING!␣␣In␣Polygon::intersection_points():\n" <<
      "The␣Polygons␣lie␣in␣parallel␣planes.␣" << "Returning␣empty␣vector<Point>.\n\n" <<
      flush;
  return v;
}
}
```

1043. Non-parallel, non-coplanar case. v will contain the intersection points of the **Line** l with $*this$ and r, if any. v can contain a maximum of four **Points** in this case. $v[0]$ and $v[1]$ will be the intersection points of the **Line** l with $*this$, and $v[2]$ and $v[3]$ the intersection points of l and r, if they exist. If any intersection point doesn't exist, `INVALID_POINT` will be stored in the corresponding element of v as a placeholder. [LDF 2003.06.29.]

The values in v provide information about the relative positions of the *Polygons*, e.g., whether they touch, whether lies within the perimeter of the other, etc. However, it's not possible to include this information in the return value, since the latter is merely a **vector⟨Point⟩**. The routine below may need to be put into another function in order to use this information. It may be of importance in breaking up *Polygons* and **Solids** for an improved surface hiding routine. [LDF 2003.06.29.]

⟨ Define **Polygon** functions 1023 ⟩ +≡

```
else {
  if (DEBUG) cout ≪ "Non-coplanar,␣non-parallel.\n";
  Line l = pl.intersection_line(r_pl);
  bool_point_pair bpp = intersection_points(l.position, (l.position + l.direction));
  v.push_back(bpp.first.pt);
  if (bpp.first.pt ≡ bpp.second.pt) {
    if (DEBUG) cout ≪ "bpp.first.pt␣and␣bpp.second.pt␣are␣equal␣for␣*this.\n";
    v.push_back(INVALID_POINT);
  }
  else v.push_back(bpp.second.pt);
  bpp = r.intersection_points(l.position, (l.position + l.direction));
  v.push_back(bpp.first.pt);
  if (bpp.first.pt ≡ bpp.second.pt) {
    if (DEBUG) cout ≪ "bpp.first.pt␣and␣bpp.second.pt␣are␣equal␣for␣r.\n";
    v.push_back(INVALID_POINT);
  }
  else v.push_back(bpp.second.pt);
  if (¬(v[0] ≠ INVALID_POINT ∨ v[1] ≠ INVALID_POINT ∨ v[2] ≠ INVALID_POINT ∨ v[3] ≠ INVALID_POINT))
    {
    if (DEBUG)
      cout ≪ "No␣intersection␣points␣found.␣␣" ≪ "Returning␣empty␣vector<Point>\n\n" ≪
          flush;
    v.clear();
    return v;
  }
  bool_real br[4];
  if (¬(v[2] ≡ INVALID_POINT ∨ v[3] ≡ INVALID_POINT)) {
    if (v[0] ≠ INVALID_POINT) br[0] = v[0].is_on_segment(v[2], v[3]);
    else {
      br[0].first = false;
      br[0].second = 0;
    }
    if (v[1] ≠ INVALID_POINT) br[1] = v[1].is_on_segment(v[2], v[3]);
    else {
      br[1].first = false;
      br[1].second = 0;
    }
  }
  if (¬(v[0] ≡ INVALID_POINT ∨ v[1] ≡ INVALID_POINT)) {
    if (v[2] ≠ INVALID_POINT) br[2] = v[2].is_on_segment(v[0], v[1]);
```

```
    else {
      br[2].first = false;
      br[2].second = 0;
    }
    if (v[3] ≠ INVALID_POINT) br[3] = v[3].is_on_segment(v[0], v[1]);
    else {
      br[3].first = false;
      br[3].second = 0;
    }
  }
  if (br[0].first ∧ br[1].first) {
    if (DEBUG) cout ≪ "The␣intersection␣of␣*this␣with␣l␣lies␣within␣" ≪
        "the␣intersection␣of␣r␣with␣l.\n";
  }
  else if (br[2].first ∧ br[3].first) {
    if (DEBUG) cout ≪ "The␣intersection␣of␣r␣with␣l␣lies␣within␣" ≪
        "the␣intersection␣of␣*this␣with␣l.\n";
  }
  else if (br[0].first ∨ br[1].first ∨ br[2].first ∨ br[3].first) {
    if (DEBUG) cout ≪ "The␣intersections␣of␣*this␣and␣r␣with␣l␣overlap␣partially.\n";
  }
  else {
    if (DEBUG) cout ≪ "The␣intersections␣of␣*this␣and␣r␣with␣l␣don't␣overlap␣at␣all.\n";
  }
  return v;
}   /* else. End of non-parallel, non-coplanar case. */
}
```

1044. Transformations.

Log

[LDF 2002.08.07.] Copied the entire "Transformations" section from `ellipses.web` and made the appropriate changes.

[LDF 2003.04.27.] The previous comment was out-of-date. I may have removed the transformation functions. At any rate, there were only a couple here. I have now copied the rest of them from `ellipses.web` and made the appropriate changes.

[LDF 2003.07.18.] [LDF 2003.07.18.] Moved "Transformations" section, including **operator**∗=(**const Transform** &) from **Reg_Polygon** to **Polygon**. Also removed the **Rectangle** versions in `rectangs.web`. The **Polygon** versions are now inherited by **Reg_Polygon** and **Rectangle**.

1045. Applying a transformation.

Log

[LDF 2002.11.06.] Now calling **Path** :: **operator**∗=( ) instead of looping through *points*. This way, if I change **Path** :: **operator**∗=( ), the change will automatically be reflected here.

⟨ Declare **Polygon** functions 1022 ⟩ +≡

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

1046.

⟨ Define **Polygon** functions 1023 ⟩ +≡
  Transform Polygon::**operator**∗=(**const Transform** &t)
  {
    Path::**operator**∗=(t);
    return (*center* ∗= t);
  }

1047. Rotatation around the main axes.

⟨ Declare **Polygon** functions 1022 ⟩ +≡
  virtual Transform *rotate*(**const real** x, **const real** $y = 0$, **const real** $z = 0$);

1048.

⟨ Define **Polygon** functions 1023 ⟩ +≡
  Transform Polygon::*rotate*(**const real** x, **const real** y, **const real** z)
  {
    Transform t;
    t.*rotate*(x, y, z);
    return (∗*this* ∗= t);
  }

1049. Rotate around an arbitrary axis.

1050. Point arguments.

Log

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

⟨ Declare **Polygon** functions 1022 ⟩ +≡
  virtual Transform *rotate*(**const Point** &*p0*, **const Point** &*p1*, **const real** *angle* = 180);

1051.

⟨ Define **Polygon** functions 1023 ⟩ +≡
  Transform Polygon::*rotate*(**const Point** &*p0*, **const Point** &*p1*, **const real** *angle*)
  {
    Transform t;
    t.*rotate*(*p0*, *p1*, *angle*);
    return (∗*this* ∗= t);
  }

1052. Path argument.

Log

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

⟨ Declare **Polygon** functions 1022 ⟩ +≡
  virtual Transform *rotate*(**const Path** &p, **const real** *angle* = 180);

1053.

⟨ Define **Polygon** functions 1023 ⟩ +≡

```
Transform Polygon::rotate(const Path &p, const real angle)
{
  if (¬p.is_linear()) {
    cerr ≪ "ERROR!␣In␣Ellipse::rotate(Path,␣real).\n" ≪
        "Path␣is␣not␣a␣line.␣␣Returning␣INVALID_TRANSFORM.\n\n";
    return INVALID_TRANSFORM;
  }
  return rotate(p.get_point(0), p.get_last_point(), angle);
}
```

1054. Scale.

⟨ Declare **Polygon** functions 1022 ⟩ +≡

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

1055.

⟨ Define **Polygon** functions 1023 ⟩ +≡

```
Transform Polygon::scale(real x, real y, real z)
{
  Transform t;
  t.scale(x, y, z);
  return (*this *= t);
}
```

1056. Shear.

⟨ Declare **Polygon** functions 1022 ⟩ +≡

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

1057.

⟨ Define **Polygon** functions 1023 ⟩ +≡

```
Transform Polygon::shear(real xy, real xz, real yx, real yz, real zx, real zy)
{
  Transform t;
  t.shear(xy, xz, yx, yz, zx, zy);
  return (*this *= t);
}
```

1058. Shift.

1059. real arguments.

⟨ Declare **Polygon** functions 1022 ⟩ +≡

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

1060.

⟨ Define **Polygon** functions 1023 ⟩ +≡

Transform Polygon :: *shift*(**real** x, **real** y, **real** z)
{
  Transform t;
  t.*shift*(x, y, z);
  return (∗*this* ∗= t);
}

1061. Point argument.

⟨ Declare **Polygon** functions 1022 ⟩ +≡

virtual Transform *shift*(**const Point** &p);

1062.

⟨ Define **Polygon** functions 1023 ⟩ +≡

Transform Polygon :: *shift*(**const Point** &p)
{
  return *shift*(p.*get_x*( ), p.*get_y*( ), p.*get_z*( ));
}

1063. Shift times.

1064. real arguments.

⟨ Declare **Polygon** functions 1022 ⟩ +≡

virtual void *shift_times*(**real** x, **real** $y = 1$, **real** $z = 1$);

1065.

⟨ Define **Polygon** functions 1023 ⟩ +≡

void Polygon :: *shift_times*(**real** x, **real** y, **real** z)
{
  Path :: *shift_times*(x, y, z);
  center.*shift_times*(x, y, z);
  return;
}

1066. Point argument.

⟨ Declare **Polygon** functions 1022 ⟩ +≡

virtual void *shift_times*(**const Point** &p);

1067.

⟨ Define **Polygon** functions 1023 ⟩ +≡

```
void Polygon::shift_times(const Point &p)
{
  return shift_times(p.get_x(), p.get_y(), p.get_z());
}
```

1068. Reg_Polygon (polygons.web). [LDF 2003.04.15.] TO DO: It will be necessary to supply **Reg_Polygon** with a complete set of transformation functions, so that *center* will be transformed along with the **Points** pointed to by the pointers on *points*. Some are present already, but not all.

[LDF 2003.04.15.] TO DO: Add *in_circle*( ), *out_circle*( ). Align a line from *center* in the direction of a normal with the y-axis. Use the inverse of the **Transform** to transform the **Circle**.

format *Reg_Polygon* *Polygon*

1069. Reg_Polygon class definition. **Reg_Polygon** is derived from **Polygon**.

Log

[LDF 2003.04.15.] Changed, so that **Reg_Polygon** is derived from **Path**. Previously, it was derived from **Reg_Cl_Plane_Curve**.

[LDF 2003.04.27.] Changed **protected** data members to **private**. They no longer need to be **protected**, because **Rectangle** is no longer derived from **Reg_Polygon**.

[LDF 2003.06.06.] Changed, so that **Reg_Polygon** is derived from **Polygon**, which I've just added above.

⟨ Define **class Reg_Polygon** 1069 ⟩ ≡

```
class Reg_Polygon : public Polygon {
  real internal_angle;
  real radius;
  unsigned short sides;
  bool on_free_store;
public: ⟨ Declare Reg_Polygon functions 1070 ⟩
};
```

This code is used in sections 1097 and 1098.

1070. Assignment.

Log

[LDF 2002.12.18.] Moved here. With the DEC compiler under Compaq Tru64 on the DEC Alpha computer, it worked to have the assignment operators following the constructors. With the GNU C++ compiler (GCC) under GNU/Linux on the Intel i686 computer, it didn't. See **Path**::**operator**=( ) in paths.web for more information.

⟨ Declare **Reg_Polygon** functions 1070 ⟩ ≡

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

See also sections 1073, 1076, 1079, 1087, 1089, 1091, 1092, 1093, 1095, and 1096.

This code is used in section 1069.

1071.

⟨ Define **Reg_Polygon** functions 1071 ⟩ ≡

```
const Reg_Polygon &Reg_Polygon::operator=(const Reg_Polygon &p)
{
  clear();
  Path::operator=(p);
  internal_angle = p.internal_angle;
  radius = p.radius;
  sides = p.sides;
  center = p.center;
  return *this;
}
```

See also sections 1074, 1077, 1078, 1080, 1081, 1307, 1309, 1310, 1311, 1313, and 1314.

This code is used in sections 1097 and 1315.

1072. Constructors and setting functions.

1073. Default constructor. No arguments.

⟨ Declare **Reg_Polygon** functions 1070 ⟩ +≡

```
Reg_Polygon();
```

1074.

⟨ Define **Reg_Polygon** functions 1071 ⟩ +≡

```
Reg_Polygon::Reg_Polygon()
{
  on_free_store = false;
  line_switch = false;
  cycle_switch = true;
  projective_extremes.resize(6, 0);   /* LDF 2003.04.09. Added this line. */
}
```

1075. Center, sides, diameter, and angles.

1076. Constructor.

⟨ Declare **Reg_Polygon** functions 1070 ⟩ +≡

```
Reg_Polygon(const Point &ccenter, const unsigned short ssides, const real ddiameter, const
    real angle_x = 0, const real angle_y = 0, const real angle_z = 0);
```

1077.

Log

[LDF 2003.08.27.] Reversed the order of the initializations following ":", because GCC with the "--Wall" option issued the following warning:
"'**Reg_Polygon**::*sides*' will be initialized after '**real Reg_Polygon**::*radius*'".

⟨ Define **Reg_Polygon** functions 1071 ⟩ +≡

```
Reg_Polygon::Reg_Polygon(const Point &ccenter, const unsigned short ssides, const real
        ddiameter, const real angle_x, const real angle_y, const real angle_z): radius(ddiameter/2),
        sides(ssides) { bool DEBUG = false;   /* true */
    if (DEBUG) cout << "Entering Reg_Polygon::Reg_Polygon() " <<
        "(center, sides, diameter, angles).\n" << flush;
    center = ccenter, on_free_store = false;
    internal_angle = 360.0/sides;
    cycle_switch = true;
    projective_extremes.resize(6, 0);   /* LDF 2003.04.09. Added this line. */
    center.apply_transform();
```

1078. For regular polygons with an even number of sides, we rotate them so that a flat side is at the "top" (in the direction of the positive z-axis, if *angle_x*, *angle_y*, and *angle_z* are all 0).

⟨ Define **Reg_Polygon** functions 1071 ⟩ +≡

```
for (int i = 0; i < sides; i++) { Point *vertex = create_new < Point > (0);
vertex->set(0, 0, radius);
if (sides % 2 ≡ 0) vertex->rotate(0, internal_angle/2, 0);
if (i > 0)   /* [LDF 2002.11.06.] Only rotate if the angle ≠ 0, i.e., don't rotate the first time. */
  vertex->rotate(0, i * internal_angle, 0);
if (angle_x ≠ 0 ∨ angle_y ≠ 0 ∨ angle_z ≠ 0)
    /* Rotation around the x-axis, y-axis, and z-axis, if applicable. */
  vertex->rotate(angle_x, angle_y, angle_z);
vertex->shift(center);   /* Put in position around center. */
points.push_back(vertex); }
if (DEBUG)
  cout << "Exiting Reg_Polygon::Reg_Polygon() " << "(center, sides, diameter, angles).\n" <<
      flush;
return; }
```

1079. Setting function.

⟨ Declare **Reg_Polygon** functions 1070 ⟩ +≡

```
void set(const Point &ccenter, const unsigned short ssides, const real ddiameter, const real
    angle_x = 0, const real angle_y = 0, const real angle_z = 0);
```

1080. ?? [LDF 2002.10.07.] See below.

⟨ Define **Reg_Polygon** functions 1071 ⟩ +≡

```
void Reg_Polygon::set(const Point &ccenter, const unsigned short ssides, const real
        ddiameter, const real angle_x, const real angle_y, const real angle_z){ bool DEBUG = false;
      /* true */
    if (DEBUG)
      cout << "Entering Reg_Polygon::set() " << "(center, sides, diameter, angles).\n" <<
          flush;
```

1081. ?? [LDF 2002.10.07.] At exactly this place, **Path** :: **Path**( ) (the default version with no arguments) is invoked. When *set*( ) exits, ~**Path**( ) is called on the empty **Path**. When DEBUG ≡ *true*, the following message is printed before ~**Path**( ) is entered. I don't know why **Path**( ) is invoked and this bothers me a bit. However, it's destroyed cleanly, so I don't have to worry about leakage.

⟨ Define **Reg_Polygon** functions 1071 ⟩ +≡
  Reg_Polygon *p*(*ccenter*, *ssides*, *ddiameter*, *angle_x*, *angle_y*, *angle_z*);
  ∗*this* = *p*;
  if (DEBUG)
    cout ≪ "Exiting␣Reg_Polygon::set()␣" ≪ "(center,␣sides,␣diameter,␣angles).\n" ≪ *flush*;
  return; }

1082. Pseudo-constructor for dynamic allocation.

1083. Pointer argument.

Log

[LDF 2002.11.06.] Added optional **Reg_Polygon** pointer argument. Made non-**inline**.
[LDF 2003.12.30.] Replaced **Reg_Polygon** :: *create_new_reg_polygon*( ) with a specialization of **template**⟨**class** *C*⟩ *C*∗*create_new*( ) for **Reg_Polygon**.

⟨ Declare non-member template functions for **Reg_Polygon** 1083 ⟩ ≡
  Reg_Polygon ∗*create_new*(**const Reg_Polygon** ∗*r*);

See also section 1084.
This code is used in sections 1097 and 1098.

1084. Reference argument.

Log

[LDF 2002.11.06.] Added this function.
[LDF 2003.12.30.] Replaced **Reg_Polygon** :: *create_new_reg_polygon*( ) with a specialization of **template**⟨**class** *C*⟩ *C*∗*create_new*( ) for **Reg_Polygon**.

⟨ Declare non-member template functions for **Reg_Polygon** 1083 ⟩ +≡
  Reg_Polygon ∗*create_new*(**const Reg_Polygon** &*r*);

1085. Destructor. [LDF 2002.10.09.] Removed the destructor. **Path** :: ~**Path**( ) or **Path** :: *clear*( ) should be used instead, unless I add dynamically allocated data members to **Reg_Polygon** (rather than **Path**).

1086. Returning elements and information.

Log

[LDF 2002.11.03.] Removed **Reg_Polygon** :: *is_planar*( ). A **Reg_Polygon** can be manipulated into a non-planar state, so it's safer to use the **Path** version, which tests whether it's really planar or not.

1087. Get radius.

Log

[LDF 2003.06.13.] Added this function.

⟨ Declare **Reg_Polygon** functions 1070 ⟩ +≡
```
inline real get_radius( ) const
{
  return radius;
}
```

1088. Circles. [LDF 2003.06.13.] The functions in this section are all defined in `circles.web`, because **Circle** is an incomplete type in this file.

Log

[LDF 2003.06.13.] Added this section.

1089. Enclosed circle.

Log

[LDF 2003.06.13.] Added this function.

⟨ Declare **Reg_Polygon** functions 1070 ⟩ +≡
```
Circle in_circle( ) const;
```

1090. Draw enclosed circle.

1091. Normal version.

⟨ Declare **Reg_Polygon** functions 1070 ⟩ +≡
```
Circle draw_in_circle(const Color &ddraw_color = *Colors::default_color, const string
    ddashed = "", const string ppen = "", Picture &picture = current_picture) const;
```

1092. Picture argument first.

Log

[LDF 2003.07.04.] Removed default argument for *picture*. Having one made it impossible for the compiler to resove calls to *draw_in_circle*( ) with no arguments.

⟨ Declare **Reg_Polygon** functions 1070 ⟩ +≡
```
Circle draw_in_circle(Picture &picture, const Color &ddraw_color = *Colors::default_color, const
    string ddashed = "", const string ppen = "") const;
```

1093. Surrounding circle.

⟨ Declare **Reg_Polygon** functions 1070 ⟩ +≡
```
Circle out_circle( ) const;
```

1094. Draw surrounding circle.

1095. Normal version.

⟨ Declare **Reg_Polygon** functions 1070 ⟩ +≡
```
Circle draw_out_circle(const Color &ddraw_color = *Colors::default_color, const string
    ddashed = "", const string ppen = "", Picture &picture = current_picture) const;
```

1096. Picture argument first.

Log

[LDF 2003.07.04.] Removed default argument for *picture*. Having one made it impossible for the compiler to resove calls to *draw_out_circle*( ) with no arguments.

⟨ Declare **Reg_Polygon** functions 1070 ⟩ +≡
Circle *draw_out_circle*(**Picture** &*picture*, **const Color** &*ddraw_color* = ∗**Colors** :: *default_color*, **const string** *ddashed* = "", **const string** *ppen* = "") **const**;

1097. Putting polygons together. This is what's compiled.

⟨ Include files 6 ⟩
⟨ Version control identifier 5 ⟩
⟨ Define **class Polygon** 1019 ⟩
⟨ Define **class Reg_Polygon** 1069 ⟩
⟨ Define **Reg_Polygon** functions 1071 ⟩
⟨ Define **Polygon** functions 1023 ⟩
⟨ Declare non-member template functions for **Reg_Polygon** 1083 ⟩

1098. This is what's written to `polygons.h`.

⟨ `polygons.h` 1098 ⟩ ≡
  ⟨ Define **class Polygon** 1019 ⟩
  ⟨ Define **class Reg_Polygon** 1069 ⟩
  ⟨ Declare non-member template functions for **Reg_Polygon** 1083 ⟩

1099. Rectangle (`rectangles.web`).

Log

[LDF 2003.07.18.] Removed the "Transformations" section, including **operator**∗=(**const Transform** &). Also moved the **Reg_Polygon** versions to **Polygon** in `polygons.web`. These are now inherited by **Rectangle**.

[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.

---

format *Rectangle* *Reg_Polygon*

⟨ Version control identifier 5 ⟩ +≡
  static string *rcs_id* = `"$Id:␣rectangs.web,v␣1.5␣2004/01/12␣21:32:24␣lfinsto1␣Exp␣$"`;

1100. 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"
#include "points.h"
#include "lines.h"
#include "planes.h"
#include "paths.h"
#include "curves.h"
#include "polygons.h"
```

1101. Rectangle class definition. [LDF 2003.07.18.] TO DO: *axis_h* and *axis_v* are not recalculated when a **Rectangle** is transformed. I should do something about this.

⟨ Define **class Rectangle** 1101 ⟩ ≡
```
class Rectangle : public Polygon {
   real axis_h;
   real axis_v;
   bool on_free_store;
public: ⟨ Declare Rectangle functions 1103 ⟩
};
```
This code is used in sections 1139 and 1140.

1102. Constructors and setting functions.

1103. Default constructor. No arguments.

⟨ Declare **Rectangle** functions 1103 ⟩ ≡
  Rectangle( );

See also sections 1106, 1108, 1111, 1113, 1119, 1122, 1125, 1127, 1130, 1132, 1135, 1136, 1137, and 1138.

This code is used in section 1101.

1104.

⟨ Define **Rectangle** functions 1104 ⟩ ≡
  Rectangle :: **Rectangle**( )
  {
    on_free_store = *false*;
    line_switch = *false*;
    cycle_switch = *true*;
  }

See also sections 1107, 1109, 1112, 1114, 1120, 1123, 1126, 1128, 1131, 1133, 1267, 1268, 1269, and 1270.

This code is used in sections 1139 and 1271.

1105. Center, lengths, and angles. [LDF 2002.11.06.] The following constructor and setting function create the **Rectangle** in the x-z plane and then rotate according to the arguments *angle_x*, *angle_y*, and *angle_z*, if at least one of them is non-zero.

1106. Constructor.

Log

[LDF 2002.11.06.] Made **real** arguments **const**.

[LDF 2003.07.18.] BUG FIX: Now, *axis_h* and *axis_v* are no longer divided by 2, when I initialize *axis_h_half* and *axis_v_half*. I mistakenly used /= instead of /.

⟨ Declare **Rectangle** functions 1103 ⟩ +≡
  Rectangle(**const Point** &*ccenter*, **const real** *aaxis_h*, **const real** *aaxis_v*, **const real** *angle_x* = 0, **const real** *angle_y* = 0, **const real** *angle_z* = 0);

1107.

⟨ Define **Rectangle** functions 1104 ⟩ +≡

```
Rectangle::Rectangle(const Point &ccenter, const real aaxis_h, const real aaxis_v, const
        real angle_x, const real angle_y, const real angle_z): axis_h(aaxis_h), axis_v(aaxis_v) {
        on_free_store = false;
    line_switch = false;
    cycle_switch = true;
    center = ccenter;
    center.apply_transform();
    real axis_h_half = axis_h/2;
    real axis_v_half = axis_v/2;
    Point bot_lft(−axis_h_half, 0, −axis_v_half);
    Point bot_rt(axis_h_half, 0, −axis_v_half);
    Point top_lft(−axis_h_half, 0, axis_v_half);
    Point top_rt(axis_h_half, 0, axis_v_half);
    if (angle_x ≠ 0 ∨ angle_y ≠ 0 ∨ angle_z ≠ 0)    /* Rotation around the x-axis, y-axis, and z-axis. */
    {
      bot_lft.rotate(angle_x, angle_y, angle_z);
      bot_rt.rotate(angle_x, angle_y, angle_z);
      top_lft.rotate(angle_x, angle_y, angle_z);
      top_rt.rotate(angle_x, angle_y, angle_z);
    }    /* Put around center. */
    bot_lft.shift(center);
    bot_rt.shift(center);
    top_lft.shift(center);
    top_rt.shift(center); for (int i = 0; i < 4; i++) points.push_back ( create_new < Point > (0) ) ;
    *points[0] = bot_lft;
    *points[1] = bot_rt;
    *points[2] = top_rt;
    *points[3] = top_lft; }
```

1108. Setting function.

Log

[LDF 2002.11.06.] Made **real** arguments **const**.

⟨ Declare **Rectangle** functions 1103 ⟩ +≡

```
void set(const Point &ccenter, const real aaxis_h, const real aaxis_v, const real angle_x = 0, const
    real angle_y = 0, const real angle_z = 0);
```

1109.

⟨ Define **Rectangle** functions 1104 ⟩ +≡

```
void Rectangle::set(const Point &ccenter, const real aaxis_h, const real aaxis_v, const real
        angle_x, const real angle_y, const real angle_z)
{
  Rectangle r(ccenter, aaxis_h, aaxis_v, angle_x, angle_y, angle_z);
  *this = r;
}
```

1110. Four Points. The **Point** arguments must be so ordered, that they are contiguous in the resulting **Rectangle**.

1111. Constructor.

⟨ Declare **Rectangle** functions 1103 ⟩ +≡
Rectangle(**const Point** &*pt0*, **const Point** &*pt1*, **const Point** &*pt2*, **const Point** &*pt3*);

1112.

⟨ Define **Rectangle** functions 1104 ⟩ +≡
Rectangle::**Rectangle**(**const Point** &*pt0*, **const Point** &*pt1*, **const Point** &*pt2*, **const Point** &*pt3*){ **Point** *pt4* = *pt1* − *pt0*; /* Check that the **Point** arguments are coplanar. */
Point *pt5* = *pt2* − *pt0*;
Point *pt6* = *pt3* − *pt0*;
Point *pt7* = *pt4*.*cross_product*(*pt5*);
pt7.*unit_vector*(*true*);
Point *pt8* = *pt4*.*cross_product*(*pt6*);
pt8.*unit_vector*(*true*); **if** (*pt7* ≡ *pt8* ∨ *pt7* ≡ −*pt8*) /* If they are, create a **Rectangle**. */
{ *on_free_store* = *false*;
line_switch = *false*;
cycle_switch = *true*;
center = *pt0*.*mediate*(*pt2*);
axis_h = (*pt1* − *pt0*).*magnitude*( );
axis_v = (*pt2* − *pt1*).*magnitude*( ); *points*.*push_back* (*create_new* < **Point** > (*pt0*)) ; *points*.*push_back* (*create_new* < **Point** > (*pt1*)) ; *points*.*push_back* (*create_new* < **Point** > (*pt2*)) ; *points*.*push_back* (*create_new* < **Point** > (*pt3*)) ;
connectors.*push_back*("--"); }
else {
cerr ≪ "ERROR!␣In␣Rectangle()␣with␣four␣Point␣arguments.\n" ≪ "Points␣are␣not␣coplanar.␣␣Returning.\n\n" ≪ *flush*;
}
return; }

1113. Setting function.

⟨ Declare **Rectangle** functions 1103 ⟩ +≡
void *set*(**const Point** &*pt0*, **const Point** &*pt1*, **const Point** &*pt2*, **const Point** &*pt3*);

1114.

⟨ Define **Rectangle** functions 1104 ⟩ +≡

void Rectangle :: *set*(**const Point** &*pt0*, **const Point** &*pt1*, **const Point** &*pt2*, **const Point** &*pt3*)
{
  Rectangle *r*(*pt0*, *pt1*, *pt2*, *pt3*);
  ∗*this* = *r*;
}

1115. Pseudo-constructor for dynamic allocation.

1116. Pointer argument.

Log

[LDF 2003.12.30.] Replaced **Rectangle** :: *create_new_rectangle*( ) with a specialization of **template**⟨**class** *C*⟩ *C*∗*create_new*( ) for **Rectangle**.

⟨ Declare non-member template functions for **Rectangle** 1116 ⟩ ≡

Rectangle ∗*create_new*(**const Rectangle** ∗*c*);

See also section 1117.

This code is used in sections 1139 and 1140.

1117. Reference argument.

Log

[LDF 2003.12.30.] Replaced **Rectangle** :: *create_new_rectangle*( ) with a specialization of **template**⟨**class** *C*⟩ *C*∗*create_new*( ) for **Rectangle**.

⟨ Declare non-member template functions for **Rectangle** 1116 ⟩ +≡

Rectangle ∗*create_new*(**const Rectangle** &*c*);

1118. Destructor. [LDF 2002.10.09.] Removed the destructor. **Path** :: ~**Path**( ) or **Path** :: *clear*( ) should be used instead, unless I add dynamically allocated data members to **Rectangle** (rather than **Reg_Polygon** or **Path**).

1119. Assignment.

Log

[LDF 2002.11.06.] Changed return value from **void** to **const Rectangle** &.

⟨ Declare **Rectangle** functions 1103 ⟩ +≡

const Rectangle &**operator**=(**const Rectangle** &*c*);

1120. !! Remember to put anything specific to **Rectangles** in here!

⟨ Define **Rectangle** functions 1104 ⟩ +≡

```
const Rectangle &Rectangle::operator=(const Rectangle &c)
{
  clear();
  Path::operator=(c);
  center = c.center;
  axis_h = c.axis_h;
  axis_v = c.axis_v;
  return *this;
}
```

1121. Returning Elements and information.

Log

[LDF 2003.04.15.] Added this section. It's become necessary, since I'm deriving **Rectangle** from **Path** now, and not from **Reg_Polygon**.

1122. Is rectangular. *is_rectangular*( ) tests whether a **Rectangle** is rectangular. It first tests if the **Rectangle** is planar. Then it creates vectors from the points on the **Rectangle**, and checks their angles to one another. If they are within **Point** :: *epsilon*( ) (exclusive) of 180° in one case, and 90° in the other two, *is_rectangular*( ) returns 1, otherwise 0. [LDF 2003.12.02.]

Log

[LDF 2003.11.28.] Added this function.
[LDF 2003.12.02.] Added test of planarity at beginning of function.
[LDF 2003.12.09.] Now using *cross_product*( ) to test for parallelity of the sides. TO DO: Add **Path** :: *is_parallel*( ) and a version for **Points**.

⟨ Declare **Rectangle** functions 1103 ⟩ +≡

```
bool is_rectangular() const;
```

1123.

⟨ Define **Rectangle** functions 1104 ⟩ +≡

bool Rectangle :: *is_rectangular* ( ) **const**
{
  if ($\neg$*is_planar* ( )) **return** *false*;
  Point a = (*get_point* (1) − *get_point* (0));
  Point b = (*get_point* (2) − *get_point* (3));
  Point c = (*get_point* (3) − *get_point* (0));
  Point d = (*get_point* (2) − *get_point* (1));
  return (a.*cross_product* (b) ≡ *origin* ∧ c.*cross_product* (d) ≡ *origin* ∧ *fabs* (*fabs* (a.*angle* (d)) − 90) < **Point** :: *epsilon* ( ));
}

1124. Returning Points.

Log

[LDF 2002.11.06.] Got rid of *get_center* ( ). It's not needed, since **Reg_Polygon** :: *get_center* ( ) does the trick.

[LDF 2003.04.15.] Added *get_center* ( ) again, since I'm no longer deriving **Rectangle** from **Reg_Polygon**, but from **Path**.

[LDF 2003.07.18.] Got rid of *get_center* ( ) again, because **Rectangle** is now derived from **Polygon**, and I've moved the **Reg_Polygon** versions to **Polygon**.

1125. Corner. The argument c should be in the range 0 <= c <= 3.

⟨ Declare **Rectangle** functions 1103 ⟩ +≡

Point *corner* (**unsigned short** c);

1126.

⟨ Define **Rectangle** functions 1104 ⟩ +≡

Point Rectangle :: *corner* (**unsigned short** c)
{
  if ($c > 3$) {
    cerr ≪ "ERROR:␣Rectangles␣have␣4␣corners,␣" ≪ "numbered␣0␣through␣3.\nReturning␣INVALID_POINT.\n" ≪ *flush*;
    return INVALID_POINT;
  }
  return ∗*points* [c];
}

1127. Get Mid-point. The argument c should be in the range 0 <= c <= 3.

Log

[LDF 2002.11.06.] Changed this function so that it uses *mediate* ( ).

[LDF 2003.05.09.] Renamed this function *get_mid_point* ( ). Formerly, it was called *mid_point* ( ).

[LDF 2003.07.18.] Made **const**.

⟨ Declare **Rectangle** functions 1103 ⟩ +≡

Point *get_mid_point* (**unsigned short** c) **const**;

1128.

⟨ Define **Rectangle** functions 1104 ⟩ +≡

```
Point Rectangle::get_mid_point(unsigned short c) const
{
  if (c > 3) {
    cerr << "ERROR:␣Rectangles␣have␣4␣mid_points,␣" <<
        "numbered␣0␣through␣3.\nReturning␣INVALID_POINT.\n" << flush;
    return INVALID_POINT;
  }
  Point p0;
  Point p1;
  Point p2;
  p0 = *points[c];
  p1 = (c < 3) ? *points[c + 1] : *points[0];
  return p0.mediate(p1);
}
```

1129. Getting axes. [LDF 2003.07.18.] TO DO: *axis_h* and *axis_v* are not recalculated when a **Rectangle** is transformed. I should do something about this.

Log

[LDF 2003.07.18.] Added this section.

1130. Get axis_h.

⟨ Declare **Rectangle** functions 1103 ⟩ +≡

```
real get_axis_h() const;
```

1131.

⟨ Define **Rectangle** functions 1104 ⟩ +≡

```
real Rectangle::get_axis_h() const
{
  return axis_h;
}
```

1132. Get axis_v.

⟨ Declare **Rectangle** functions 1103 ⟩ +≡

```
real get_axis_v() const;
```

1133.

⟨ Define **Rectangle** functions 1104 ⟩ +≡
  real Rectangle :: *get_axis_v*( ) **const**
  {
    return *axis_v*;
  }

1134. Ellipses.

Log

[LDF 2003.07.18.] Added this section. These functions must be defined in `ellipses.web`, because **Ellipse** is an incomplete type in this file.

1135. Surrounding Ellipse.

Log

[LDF 2003.07.18.] Added this function.

⟨ Declare **Rectangle** functions 1103 ⟩ +≡
  Ellipse *out_ellipse*( ) **const**;

1136. Enclosed Ellipse.

Log

[LDF 2003.07.18.] Added this function.

⟨ Declare **Rectangle** functions 1103 ⟩ +≡
  Ellipse *in_ellipse*( ) **const**;

1137. Draw surrounding Ellipse.

Log

[LDF 2003.07.18.] Added this function.

⟨ Declare **Rectangle** functions 1103 ⟩ +≡
  Ellipse *draw_out_ellipse*(**const Color** &*ddraw_color* = ∗**Colors** :: *default_color*, **string**
    ddashed = "", **string** *ppen* = "", **Picture** &*picture* = *current_picture*) **const**;

1138. Draw enclosed Ellipse.

Log

[LDF 2003.07.18.] Added this function.

⟨ Declare **Rectangle** functions 1103 ⟩ +≡
  Ellipse *draw_in_ellipse*(**const Color** &*ddraw_color* = ∗**Colors** :: *default_color*, **string**
    ddashed = "", **string** *ppen* = "", **Picture** &*picture* = *current_picture*) **const**;

1139. Putting Rectangle together. This is what's compiled.

⟨ Include files 6 ⟩
⟨ Version control identifier 5 ⟩

⟨ Define **class Rectangle** 1101 ⟩
⟨ Define **Rectangle** functions 1104 ⟩
⟨ Declare non-member template functions for **Rectangle** 1116 ⟩

1140. This is what's written to `rectangs.h`.

⟨ `rectangs.h` 1140 ⟩ ≡
  ⟨ Define **class Rectangle** 1101 ⟩
  ⟨ Declare non-member template functions for **Rectangle** 1116 ⟩

1141. Ellipse (`ellipses.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.

---

format *Ellipse* *Path*

⟨ Version control identifier 5 ⟩ +≡
  static string *rcs_id* = "$Id:␣ellipses.web,v␣1.7␣2004/01/14␣17:56:19␣lfinsto1␣Exp␣$";

1142. 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"
#include "points.h"
#include "lines.h"
#include "planes.h"
#include "paths.h"
#include "curves.h"
#include "polygons.h"
#include "rectangs.h"
```

1143. Ellipse class definition.

Log

[LDF 2003.07.25.] Added *focus0*, *focus1*, and *linear_eccentricity*.

[LDF 2003.07.27.] Added *numerical_eccentricity*.

⟨ Define **class Ellipse** 1143 ⟩ ≡
```
class Ellipse : public Reg_Cl_Plane_Curve {
protected: Point focus0;
  Point focus1;
  real linear_eccentricity;
  real numerical_eccentricity;
  real axis_h;
  real axis_v;
  static unsigned short DEFAULT_NUMBER_OF_POINTS;
```

public: ⟨ Declare **Ellipse** functions 1146 ⟩
};

This code is used in sections 1271 and 1272.

1144. Static data members.

⟨ Define **static Ellipse** data members 1144 ⟩ ≡
unsigned short Ellipse::DEFAULT_NUMBER_OF_POINTS = 16;
/* [LDF 2002.11.06.] Must be a multiple of 4. */

This code is used in section 1271.

1145. Constructors.

1146. Default constructor. No arguments.

⟨ Declare **Ellipse** functions 1146 ⟩ ≡
Ellipse( );

See also sections 1149, 1151, 1157, 1160, 1162, 1164, 1166, 1167, 1169, 1171, 1174, 1177, 1179, 1182, 1184, 1186, 1188, 1192, 1194, 1197, 1199, 1201, 1203, 1205, 1208, 1210, 1214, 1231, 1233, 1235, 1237, 1239, 1242, 1244, 1247, 1249, 1252, 1254, 1257, 1259, 1261, and 1263.

This code is used in section 1143.

1147.

⟨ Define **Ellipse** functions 1147 ⟩ ≡
Ellipse::**Ellipse**( )
{
on_free_store = *false*;
line_switch = *false*;
cycle_switch = *true*;
}

See also sections 1150, 1152, 1158, 1161, 1165, 1168, 1170, 1172, 1173, 1175, 1178, 1180, 1183, 1185, 1187, 1189, 1193, 1195, 1198, 1200, 1202, 1204, 1206, 1209, 1211, 1215, 1216, 1217, 1218, 1219, 1220, 1221, 1222, 1223, 1224, 1226, 1227, 1232, 1234, 1236, 1238, 1240, 1243, 1245, 1248, 1250, 1253, 1255, 1258, 1260, 1262, and 1264.

This code is used in section 1271.

1148. Center, lengths, and angles of rotation.

1149. Constructor. The ellipse is always generated in the x-z plane with the center at the origin. Then it is rotated about the main axes according to the values of the angle arguments and shifted to *center*.

Log

[LDF 2002.11.06.] Made **real** arguments **const**.
[LDF 2003.07.25.] Added code for setting *focus0*, *focus1*, and *linear_eccentricity*.
[LDF 2003.07.27.] Added code for setting *numerical_eccentricity*.

⟨ Declare **Ellipse** functions 1146 ⟩ +≡
Ellipse(**const Point** &*ccenter*, **const real** *aaxis_h*, **const real** *aaxis_v*, **const real** *angle_x* = 0, **const real** *angle_y* = 0, **const real** *angle_z* = 0, **const unsigned short** *nnumber_of_points* = DEFAULT_NUMBER_OF_POINTS);

1150.

⟨ Define **Ellipse** functions 1147 ⟩ +≡

```
Ellipse::Ellipse(const Point &ccenter, const real aaxis_h, const real aaxis_v, const real
        angle_x, const real angle_y, const real angle_z, const unsigned short nnumber_of_points):
        axis_h(aaxis_h), axis_v(aaxis_v) { center = ccenter;
    center.apply_transform();
    focus0 = origin;
    focus1 = origin;
    real axis_h_half = axis_h/2;
    real axis_v_half = axis_v/2;
    if (axis_h ≥ axis_v) {
      linear_eccentricity = sqrt((axis_h_half * axis_h_half) − (axis_v_half * axis_v_half));
      focus0.shift(−linear_eccentricity);
      focus1.shift(linear_eccentricity);
      numerical_eccentricity = linear_eccentricity/axis_h_half;
    }
    else {
      linear_eccentricity = sqrt((axis_v_half * axis_v_half) − (axis_h_half * axis_h_half));
      focus0.shift(0, 0, −linear_eccentricity);
      focus1.shift(0, 0, linear_eccentricity);
      numerical_eccentricity = linear_eccentricity/axis_v_half;
    }
    if (nnumber_of_points % 4 ≠ 0) {
      cerr ≪ "WARNING!␣In␣Ellipse():␣Invalid␣value␣for␣number_of_points:" ≪
          nnumber_of_points ≪ "." ≪ endl ≪ "Using␣default␣instead:␣" ≪
          DEFAULT_NUMBER_OF_POINTS ≪ endl ≪ flush;
      number_of_points = DEFAULT_NUMBER_OF_POINTS;
    }
    else number_of_points = nnumber_of_points;
    on_free_store = false;
    line_switch = false;
    cycle_switch = true;
    connectors.push_back("..");
    Transform t;
    if (angle_x ≠ 0 ∨ angle_y ≠ 0 ∨ angle_z ≠ 0) t.rotate(angle_x, angle_y, angle_z);
    t.shift(center);
    focus0 *= focus1 *= t;
    real curr_angle;
    real curr_x;
    real curr_z; for (int i = 0; i < number_of_points; i++)
      /* LDF 2002.11.06. Modified this code. */
    { curr_angle = 2 * i * PI/number_of_points;
    curr_x = axis_h/2 * cos(curr_angle);
    curr_z = axis_v/2 * sin(curr_angle); points.push_back ( create_new < Point > (0) ) ;
    points.back()→set(curr_x, 0, curr_z);
    *(points.back()) *= t;
      /* Rotate *(points.back()) around the x, y, and z-axes and shift it to center. */
    } }
```

1151. Setting function.

Log

[LDF 2003.03.01.] Added this function.

⟨ Declare **Ellipse** functions 1146 ⟩ +≡
void *set*(**const Point** &*ccenter*, **const real** *aaxis_h*, **const real** *aaxis_v*, **const real** *angle_x* = 0, **const real** *angle_y* = 0, **const real** *angle_z* = 0, **const unsigned short** *nnumber_of_points* = `DEFAULT_NUMBER_OF_POINTS`);

1152.

⟨ Define **Ellipse** functions 1147 ⟩ +≡
void Ellipse :: *set*(**const Point** &*ccenter*, **const real** *aaxis_h*, **const real** *aaxis_v*, **const real** *angle_x*, **const real** *angle_y*, **const real** *angle_z*, **const unsigned short** *nnumber_of_points*)
{
Ellipse *e*(*ccenter*, *aaxis_h*, *aaxis_v*, *angle_x*, *angle_y*, *angle_z*, *nnumber_of_points*);
∗*this* = *e*;
}

1153. Pseudo-constructor for dynamic allocation.

1154. Pointer argument.

Log

[LDF 2002.11.06.] Added optional **const Ellipse** ∗ argument.
[LDF 2003.12.30.] Replaced **Ellipse** :: *create_new_ellipse*( ) with a specialization of **template**⟨**class** *C*⟩ *C*∗*create_new*( ) for **Ellipse**.

⟨ Declare non-member template functions for **Ellipse** 1154 ⟩ ≡
Ellipse ∗*create_new*(**const Ellipse** ∗*c*);

See also section 1155.

This code is used in sections 1271 and 1272.

1155. Reference argument.

Log

[LDF 2002.11.06.] Added this function.
[LDF 2003.12.30.] Replaced **Ellipse** :: *create_new_ellipse*( ) with a specialization of **template**⟨**class** C⟩ C∗*create_new*( ) for **Ellipse**.

⟨ Declare non-member template functions for **Ellipse** 1154 ⟩ +≡
  Ellipse ∗*create_new*(**const Ellipse** &*c*);

1156. Destructor. [LDF 2002.10.09.] Removed the destructor. **Path** :: ∼**Path**( ) or **Path** :: *clear*( ) should be used instead, unless I add dynamically allocated data members to **Ellipse** (rather than **Path**).

1157. Assignment.

Log

[LDF 2002.11.06.] Added error handling code to prevent self-assignment.
[LDF 2002.11.10.] Changed return value to **Ellipse** &.

⟨ Declare **Ellipse** functions 1146 ⟩ +≡
  Ellipse &**operator**=(**const Ellipse** &*e*);

1158. !! Remember to put anything specific to **Ellipses** in here!

⟨ Define **Ellipse** functions 1147 ⟩ +≡
```
Ellipse &Ellipse::operator=(const Ellipse &e)
{
  if (this ≡ &e)      /* [LDF 2002.11.06.] Make sure it's not self-assignment. */
    return *this;
  Path::operator=(e);
  center = e.center;      /* Ellipse members. */
  axis_h = e.axis_h;
  axis_v = e.axis_v;
  focus0 = e.focus0;
  focus1 = e.focus1;
  number_of_points = e.number_of_points;      /* Reg_Cl_Plane_Curve members. */
  return *this;
}
```

1159. Labelling.

1160. Label.

⟨ Declare **Ellipse** functions 1146 ⟩ +≡
```
void label(const string pos = "top", const bool dot = false, Picture &picture = current_picture)
    const;
```

1161.

⟨ Define **Ellipse** functions 1147 ⟩ +≡

```
void Ellipse::label(const string pos, const bool dot, Picture &picture) const
{
  if (Label::DO_LABELS ≡ false) {
    return;
  }
  string s;
  char c = 'a';
  for (vector⟨Point *⟩::const_iterator iter = points.begin(); iter ≠ points.end(); iter++) {
    s = c;
    (**iter).label(s, pos, dot, picture);
    c++;
  }
}
```

1162. Dotlabel.

Log

[LDF 2002.11.06.] Changed this function so that it just calls **Ellipse**::*label*( ) with *dot* = *true*. Made it **inline**.

⟨ Declare **Ellipse** functions 1146 ⟩ +≡

```
inline void dotlabel(string pos = "top", Picture &picture = current_picture) const
{
  label(pos, true, picture);
}
```

1163. Returning elements and information.

1164. Is elliptical. [LDF 2003.07.20.] *is_elliptical*( ) first checks whether $*this$ is planar by calling *get_normal*( ). If the latter function returns `INVALID_POINT`, then *is_elliptical*( ) returns *false*. Otherwise, it makes a copy of $*this$ called e, puts e into the x-z plane, and rotates it, so that $*(e.points[0])$ lies on the x-axis. Then, it plugs the x and z-coordinates of the **Points** on e into the ellipse equation, i.e., $x^2/a^2 + z^2/b^2 = 1$, where a is half of the horizontal axis of the ellipse, and b is half of the vertical axis. Let $r = x^2/a^2 + z^2/b^2$ and ϵ stand for the return value of **Point**::*epsilon*( ). If $|r - 1| > \epsilon$ for any of the **Points** on e, *is_elliptical*( ) returns *false*, otherwise it returns *true*.

Log

[LDF 2003.07.20.] Added this function.
[LDF 2003.07.25.] Now checking *normal* ≡ `INVALID_POINT`. If it is, *is_elliptical*( ) returns *false*.

⟨ Declare **Ellipse** functions 1146 ⟩ +≡

```
bool is_elliptical() const;
```

1165.

⟨ Define **Ellipse** functions 1147 ⟩ +≡

```
bool Ellipse::is_elliptical() const
{
  bool DEBUG = false;    /* true */
  Point normal = get_normal();
  if (normal ≡ INVALID_POINT) {
    if (DEBUG)
      cerr ≪ "In␣Ellipse::is_elliptical():␣" ≪ "get_normal()␣returned␣INVALID_POINT." ≪
          endl ≪ "*this␣is␣non-planar.␣Returning␣false." ≪ endl ≪ endl ≪ flush;
    return false;
  }
  Ellipse e = *this;
  normal.shift(center);
  Transform t;
  t.align_with_axis(center, normal, 'y');
  e.do_transform(t);
  if (DEBUG) e.show("e␣after␣alignment:");
  Point x_axis_pt(1);
  Point p0 = e.get_point(0);
  real ang = p0.angle(x_axis_pt);
  t.reset();
  t.rotate(0, ang);
  p0 *= e.do_transform(t);
  p0.unit_vector(true);
  if (p0 ≠ x_axis_pt) {
    t.reset();
    t.rotate(0, −2 * ang);
    e.do_transform(t);
  }
  p0 = e.get_point(0);
  p0.unit_vector(true);
  if (p0 ≠ x_axis_pt) {
    if (DEBUG)
      cerr ≪ "ERROR!␣In␣is_elliptical():\n" ≪ "Rotation␣failed.\nReturning␣false\n\n" ≪
          flush;
    return false;
  }
  if (DEBUG) e.show("e␣after␣rotation␣in␣x-z␣plane:");
  real x;
  real z;
  real a = (e.get_point(0) − e.get_center()).magnitude();
  real b = (e.get_point(number_of_points/4) − e.get_center()).magnitude();
  if (DEBUG) {
    cout ≪ "a␣==␣" ≪ a ≪ endl ≪ flush;
    cout ≪ "b␣==␣" ≪ b ≪ endl ≪ flush;
  }
  real r;
  int i = 0;
```

```
for (vector⟨Point *⟩::const_iterator iter = e.points.begin(); iter ≠ e.points.end(); ++iter) {
  x = (**iter).get_x();
  z = (**iter).get_z();
  if (DEBUG) {
    cout ≪ "Point␣" ≪ i ≪ ":␣" ≪ endl ≪ flush;
    (**iter).show("iter:");
    cout ≪ "x␣==␣" ≪ x ≪ endl ≪ flush;
    cout ≪ "z␣==␣" ≪ z ≪ endl ≪ flush;
  }
  r = ((x * x)/(a * a)) + ((z * z)/(b * b));
  if (DEBUG) cout ≪ "r␣==␣" ≪ r ≪ endl ≪ flush;
  if (fabs(r − 1) > Point::epsilon()) {
    if (DEBUG) cout ≪ "Point␣" ≪ i ≪ "␣doesn't␣satisfy␣ellipse␣equation.\n" ≪
          "Returning␣false.\n\n" ≪ flush;
    return false;
  }
  ++i;
}
if (DEBUG)
  cout ≪ "Exiting␣Ellipse::is_elliptical().␣␣Returning␣true." ≪ endl ≪ endl ≪ flush;
return true;
}
```

1166. Is quadratic.

⟨ Declare **Ellipse** functions 1146 ⟩ +≡

```
inline bool is_quadratic() const
{
  return true;
}
```

1167. Is cubic.

Log

[LDF 2003.07.27.] Made **virtual** and non-inline.

⟨ Declare **Ellipse** functions 1146 ⟩ +≡

```
virtual bool is_cubic() const;
```

1168.

⟨ Define **Ellipse** functions 1147 ⟩ +≡
  bool Ellipse :: *is_cubic*( ) **const**
  {
    return *false*;
  }

1169. Is quartic.

Log

[LDF 2003.07.27.] Made **virtual** and non-inline.

⟨ Declare **Ellipse** functions 1146 ⟩ +≡
  virtual bool *is_quartic*( ) **const**;

1170.

⟨ Define **Ellipse** functions 1147 ⟩ +≡
  bool Ellipse :: *is_quartic*( ) **const**
  {
    return *false*;
  }

1171. Solve. [LDF 2002.11.06.] *solve*( ) assumes that the **Ellipse** lies in a major plane with its center at the origin. Code that calls it must ensure that these conditions are fulfilled. *solve*( ) returns the two possible values for either the horizontal or the vertical coordinate.

TO DO: Read through, and then explain this function.

Log

[LDF 2003.07.20.] Now using *get_axis_v*( ) and *get_axis_h*( ), instead of accessing *axis_h* and *axis_v* directly.
[LDF 2003.07.25.] Removed some commented-out code, and an explanatory comment.

⟨ Declare **Ellipse** functions 1146 ⟩ +≡
  real_pair *solve*(**char** *axis_unknown*, **real** *known*) **const**;

1172.

⟨ Define **Ellipse** functions 1147 ⟩ +≡

```
real_pair Ellipse::solve(char axis_unknown, real known) const { real radius_known;
    real radius_unknown;
    axis_unknown = tolower(axis_unknown);
    real_pair r;
    if (axis_unknown ≡ 'h') {
      radius_known = get_axis_v( )/2;
      radius_unknown = get_axis_h( )/2;
    }
    else if (axis_unknown ≡ 'v') {
      radius_known = get_axis_h( )/2;
      radius_unknown = get_axis_v( )/2;
    }
    else {
      cerr ≪ "ERROR!␣In␣Ellipse::solve().\n" ≪ "Invalid␣character␣for␣axis_unknown:␣" ≪
          axis_unknown ≪ "\nReturning␣INVALID_REAL_PAIR.\n\n" ≪ flush;
      return INVALID_REAL_PAIR;
    }
    if (fabs(known) > radius_known) {
      return INVALID_REAL_PAIR;
    }
```

1173. The equation for an ellipse in the x-y plane with its center at the origin is

$$x^2/a^2 + y^2/b^2 = 1$$

where a is half the horizontal axis and b is half the vertical axis. Therefore,

$$y = \sqrt{(1 - x^2/a^2) \times b^2}$$

and

$$x = \sqrt{(1 - y^2/b^2) \times a^2}.$$

⟨ Define **Ellipse** functions 1147 ⟩ +≡
 r.first = *sqrt*((1−((*known* ∗ *known*)/(*radius_known* ∗ *radius_known*))) ∗ (*radius_unknown* ∗ *radius_unknown*));
 r.second = −*r.first*;
 return *r*; }

1174. Get coefficients. This is used for getting the coefficients of the quadratic equation that results from replacing y with $mx + b$ from the line equation, where m is the slope and b the intercept with the vertical axis, (whichever that might be in a particular case; it needn't be the y-axis) in the equation for the ellipse

$$x^2/\alpha^2 + y^2/\beta^2 = 1$$

namely !! START HERE. Check this. I think the x in the coefficient for x^2 is wrong.

$$x^2/\alpha^2 + (mx + b)^2/\beta^2 - 1 = 0 \equiv (\beta^2 x + \alpha^2 m^2)x^2 + 2\alpha^2 bmx + (\alpha^2 b^2 - \alpha^2\beta^2) = 0.$$

The coefficients are returned in the **struct real_triple** in the order one would expect: *r.first* is the coefficient of x^2, *r.second* of x and *r.third* of the constant term (x^0).

Log

[LDF 2003.07.20.] Now using *get_axis_v*( ) and *get_axis_h*( ), instead of accessing *axis_h* and *axis_v* directly.
[LDF 2003.07.27.] Corrected a typo in the math mode material showing the coefficients.

⟨ Declare **Ellipse** functions 1146 ⟩ +≡
 real_triple *get_coefficients*(**real** *Slope*, **real** *v_intercept*) **const**;

1175.

⟨ Define **Ellipse** functions 1147 ⟩ +≡

```
real_triple Ellipse::get_coefficients(real Slope, real v_intercept) const
{
  real_triple r;
  real ax_h = get_axis_h();
  real ax_v = get_axis_v();
  r.first = ((ax_v/2) * (ax_v/2)) + (((ax_h/2) * (ax_h/2)) * (Slope * Slope));   /* a */
  r.second = 2 * Slope * v_intercept * ((ax_h/2) * (ax_h/2));   /* b */
  r.third = (((ax_h/2) * (ax_h/2)) * (v_intercept * v_intercept)) - (((ax_v/2) * (ax_v/2)) * ((ax_h/2) * (ax_h/2)));
     /* c */
  return r;
}
```

1176. Get center.

1177. Non-const version.

Log

[LDF 2002.11.10.] Made this function **virtual** and non-**inline**.
[LDF 2003.05.09.] Changed return value from **Point** & to **const Point** &.

⟨ Declare **Ellipse** functions 1146 ⟩ +≡
virtual const Point &*get_center*( );

1178.

⟨ Define **Ellipse** functions 1147 ⟩ +≡
const Point &**Ellipse**::*get_center*( )
{
center.*apply_transform*( );
return *center*;
}

1179. const version.

Log

[LDF 2002.11.10.] Made this function **virtual** and non-**inline**.

⟨ Declare **Ellipse** functions 1146 ⟩ +≡
virtual Point *get_center*( ) **const**;

1180.

⟨ Define **Ellipse** functions 1147 ⟩ +≡
Point Ellipse::*get_center*( ) **const**
{
Point *p*(*center*);
p.*apply_transform*( );
return *p*;
}

1181. Get focus.

1182. Non-const version.

Log

[LDF 2003.07.25.] Added this function.

⟨ Declare **Ellipse** functions 1146 ⟩ +≡
const Point &*get_focus*(**const unsigned short** *s*);

1183.

⟨ Define **Ellipse** functions 1147 ⟩ +≡

```
const Point &Ellipse::get_focus(const unsigned short s)
{
  if (s ≡ 0) {
    focus0.apply_transform();
    return focus0;
  }
  else if (s ≡ 1) {
    focus1.apply_transform();
    return focus1;
  }
  else {
    cerr ≪ "ERROR!␣In␣|Ellipse::get_focus()|:\n" ≪ "Invalid␣argument:␣" ≪ s ≪ endl ≪
        "Valid␣arguments␣are␣0␣and␣1.\n" ≪ "Returning␣INVALID_POINT.\n\n" ≪ flush;
    return INVALID_POINT;
  }
}
```

1184. const version.

Log

[LDF 2003.07.25.] Added this function.

⟨ Declare **Ellipse** functions 1146 ⟩ +≡

```
Point get_focus(const unsigned short s) const;
```

1185.

⟨ Define **Ellipse** functions 1147 ⟩ +≡

```
Point Ellipse::get_focus(const unsigned short s) const
{
  Point p;
  if (s ≡ 0) {
    p = focus0;
    p.apply_transform();
    return p;
  }
  else if (s ≡ 1) {
    p = focus1;
    p.apply_transform();
    return p;
  }
  else {
    cerr ≪ "ERROR!␣In␣|Ellipse::get_focus()|:\n" ≪ "Invalid␣argument:␣" ≪ s ≪ endl ≪
        "Valid␣arguments␣are␣0␣and␣1.\n" ≪ "Returning␣INVALID_POINT.\n\n" ≪ flush;
    return INVALID_POINT;
  }
}
```

1186. Get linear eccentricity.

Log

[LDF 2003.07.25.] Added this function.

⟨ Declare **Ellipse** functions 1146 ⟩ +≡

```
real get_linear_eccentricity() const;
```

1187.

⟨ Define **Ellipse** functions 1147 ⟩ +≡

```
real Ellipse::get_linear_eccentricity() const
{
  return linear_eccentricity;
}
```

1188. Get numerical eccentricity.

Log

[LDF 2003.07.27.] Added this function.

⟨ Declare **Ellipse** functions 1146 ⟩ +≡

```
real get_numerical_eccentricity() const;
```

1189.

⟨ Define **Ellipse** functions 1147 ⟩ +≡

```
real Ellipse::get_numerical_eccentricity() const
{
  return numerical_eccentricity;
}
```

1190. Get axes.

Log

[LDF 2003.07.20.] Rewrote the **const** versions of the functions in this section, and added non-**const** versions. All of them now check whether *∗this* is still elliptical using *is_ellipse*( ). If it is, the value *axis_v* or *axis_h* should have is recalculated, and this value is returned. In the non-**const** versions, *axis_v* or *axis_h* is reset to the new value. If *∗this* is no longer elliptical, the function returns `INVALID_REAL`, and *axis_h* or *axis_v* is set to `INVALID_REAL` in the non-**const** versions.

[LDF 2003.07.20.] *axis_h* and *axis_v* are updated by the transformation functions, and these are presumably the only ones that could cause an **Ellipse** to become non-elliptical. So, checking and recalculating them here is probably redundant. However, this may change, so it's safer to do this here.

[LDF 2003.07.25.] BUG FIX: *axis_h* and *axis_v* were too small by half. Now multiplying by 2 in all versions of *get_axis_h*( ) and *get_axis_v*( ).

1191. Get vertical axis.

1192. const version.

⟨ Declare **Ellipse** functions 1146 ⟩ +≡

```
real get_axis_v() const;
```

1193.

⟨ Define **Ellipse** functions 1147 ⟩ +≡

```
real Ellipse::get_axis_v() const
{
  if (is_elliptical()) return (2 * (get_point(number_of_points/4) - get_center()).magnitude());
  else return INVALID_REAL;
}
```

1194. Non-const version.

⟨ Declare **Ellipse** functions 1146 ⟩ +≡

```
real get_axis_v();
```

1195.

⟨ Define **Ellipse** functions 1147 ⟩ +≡

```
real Ellipse::get_axis_v()
{
  if (is_elliptical()) axis_v = ((get_point(number_of_points/4) - get_center()).magnitude() * 2);
  else axis_v = INVALID_REAL;
  return axis_v;
}
```

1196. Get horizontal axis.

1197. const version.

⟨ Declare **Ellipse** functions 1146 ⟩ +≡
real *get_axis_h*( ) **const**;

1198.

⟨ Define **Ellipse** functions 1147 ⟩ +≡

```
real Ellipse::get_axis_h() const
{
  if (is_elliptical()) return ((get_point(0) - get_center()).magnitude() * 2);
  else return INVALID_REAL;
}
```

1199. Non-const version.

⟨ Declare **Ellipse** functions 1146 ⟩ +≡
real *get_axis_h*( );

1200.

⟨ Define **Ellipse** functions 1147 ⟩ +≡

```
real Ellipse::get_axis_h()
{
  if (is_elliptical()) axis_h = (get_point(0) - get_center()).magnitude() * 2;
  else axis_h = INVALID_REAL;
  return axis_h;
}
```

1201. Angle point. *angle_point*( ) returns a point on the ellipse given an angle. Effectively, *point*[0] is rotated about the center in the plane of the ellipse and the intersection of the ray from the center through *point*[0] and the ellipse is returned.

[LDF 2003.07.27.] TO DO: Try to get the rotation to always go in the direction I would like.

Log

[LDF 2003.07.01.] BUG FIX: Now returning *bpp.first.pt* if it's not equal to INVALID_POINT. Otherwise, *bpp.second.pt* is returned. The latter may be a valid **Point**, or INVALID_POINT. Before, INVALID_POINT was returned if *bpp.first.b* and *bpp.second.b* were *false*, but this is the case, if the intersection points didn't lie on the line segment between *center* and *pt0*.

BUG FIX: Now checking to make sure that the intersection point lies in the proper direction. Now that the intersection point doesn't have to be on the line segment, it's necessary to check this.

[LDF 2003.07.20.] Now using *get_axis_v*( ) and *get_axis_h*( ), instead of accessing *axis_h* and *axis_v* directly.

[LDF 2003.07.27.] Made **const**. Made *angle* argument **const**. Now using *get_center*( ) instead of *center*.

⟨ Declare **Ellipse** functions 1146 ⟩ +≡
Point *angle_point*(**const real** *angle*) **const**;

1202.

⟨ Define **Ellipse** functions 1147 ⟩ +≡

```
Point Ellipse::angle_point(const real angle) const
{
  Point Center = get_center();
  Point normal = get_normal();
  normal.shift(Center);
  Point pt0 = get_point(0);
  pt0 -= Center;
  pt0.unit_vector(true);
  pt0 *= max(get_axis_h(), get_axis_v())/2;
    /* [LDF 2002.11.06.] pt0 will either lie on the perimeter of the Ellipse or beyond it. */
  pt0.shift(Center);
  pt0.rotate(Center, normal, angle);
  bool_point_pair bpp = intersection_points(Center, pt0);
  Point direction_line(pt0 - Center);
  direction_line.unit_vector(true);
  Point direction_pt;
  if (bpp.first.pt ≠ INVALID_POINT) {
    direction_pt = bpp.first.pt;
    direction_pt -= Center;
    direction_pt.unit_vector(true);
    if (direction_pt ≡ direction_line) return bpp.first.pt;
  }
  return bpp.second.pt;
}
```

1203. Equality. TO DO: I'll need to define **Path**::**operator**≡( ) in order to be able to define this function.

⟨ Declare **Ellipse** functions 1146 ⟩ +≡

```
#if 0
  virtual bool operator≡(const Ellipse &e);
#endif
```

1204.

⟨ Define **Ellipse** functions 1147 ⟩ +≡
#if 0
  virtual bool Ellipse :: **operator**≡(**const Ellipse** &*e*)
  { }
#endif

1205. Location of a point. [LDF 2003.07.25.] This function overloads **Reg_Cl_Plane_Curve** :: *location*( ). It's simpler, because it doesn't need to copy the **Ellipse** and transform the copy into a major plane. Nor does it require the use of *solve*( ).

[LDF 2003.07.25.] If the **Point** argument P lies in the same plane as ∗*this*, *location*( ) compares the sum of the distances of P from the foci to 2 times the maximum of *axis_h* and *axis_v*.

[LDF 2003.07.25.] Let m stand for (P − *focus0*).*magnitude*( ) + (P − *focus0*).*magnitude*( ), d for 2 ∗ *max*(*axis_h*, *axis_v*), and ϵ for **Point** :: *epsilon*( ). The return values are as follows:

0 $|m - d| < \epsilon$. P lies on the perimeter of the **Ellipse**.

-1 $m > d$. P lies outside the **Ellipse**.

1 $m < d$. P lies inside the perimeter of the **Ellipse**.

-2 P is not in the same plane as the **Ellipse**.

-3 The **Ellipse** is non-elliptical.

Log

[LDF 2003.07.25.] Added this function.

⟨ Declare **Ellipse** functions 1146 ⟩ +≡
  virtual signed short *location*(**const Point** &*p*) **const**;

1206.

⟨ Define **Ellipse** functions 1147 ⟩ +≡

```
signed short Ellipse::location(const Point &p) const
{
  bool DEBUG = false;    /* true */
  real ax_h = get_axis_h();
  real ax_v = get_axis_v();
  if (ax_h ≡ INVALID_REAL ∨ ax_v ≡ INVALID_REAL) {
    cerr ≪ "ERROR!␣In␣Ellipse::location():\n" ≪
        "Ellipse␣is␣non-elliptical.␣Returning␣-3.\n\n" ≪ flush;
    return −3;
  }
  if (¬p.is_on_plane(get_plane())) {
    cerr ≪ "WARNING!␣In␣Ellipse::location():\n" ≪
        "Point␣doesn't␣lie␣in␣plane␣of␣Ellipse.\n" ≪ "Returning␣-2.\n\n" ≪ flush;
    return −2;
  }
  real max_ax = max(ax_h, ax_v);
  Point q = p − get_focus(0);
  real mag = q.magnitude();
  q = p − get_focus(1);
  mag += q.magnitude();
  if (fabs(mag − max_ax) < Point::epsilon()) {
    if (DEBUG) cout ≪ "Point␣lies␣on␣perimeter␣of␣|Ellipse|.\n";
    return 0;
  }
  else if (mag > max_ax) {
    if (DEBUG) cout ≪ "Point␣lies␣outside␣of␣perimeter␣of␣|Ellipse|.\n";
    return −1;
  }
  else {
    if (DEBUG) cout ≪ "Point␣lies␣inside␣of␣perimeter␣of␣|Ellipse|.\n";
    return 1;
  }
}
```

1207. Intersection points.

1208. Point arguments.

Log

[LDF 2003.07.27.] Made the arguments **const Point** &.

⟨ Declare **Ellipse** functions 1146 ⟩ +≡

```
virtual bool_point_pair intersection_points(const Point &p0, const Point &p1) const;
```

1209.

⟨ Define **Ellipse** functions 1147 ⟩ +≡
```
bool_point_pair Ellipse::intersection_points(const Point &pt0, const Point &pt1) const
{
  return Reg_Cl_Plane_Curve::intersection_points(center, pt0, pt1);
}
```

1210. Path argument. This function just checks to be sure that **Path** p is a line, extracts the **Points**, and calls the version with **Point** arguments, returning the latter's return value.

Log

[LDF 2003.07.09.] Made **virtual**.

[LDF 2003.07.27.] Made argument p a **const Path** &. Changed, so that *is_linear*( ) is used, and *get_last_point*( ) rather than *get_point*(1).

⟨ Declare **Ellipse** functions 1146 ⟩ +≡
```
virtual bool_point_pair intersection_points(const Path &p) const;
```

1211.

⟨ Define **Ellipse** functions 1147 ⟩ +≡
```
bool_point_pair Ellipse::intersection_points(const Path &p) const
{
  if (¬p.is_linear()) {
    cerr ≪ "ERROR!␣In␣Ellipse::intersection_points(const␣Path&):\n" ≪
        "Path␣argument␣is␣non-linear.\n" ≪ "Haven't␣programmed␣this␣case␣yet.\n" ≪
        "Returning␣INVALID_BOOL_POINT_PAIR." ≪ endl ≪ endl ≪ flush;
    return INVALID_BOOL_POINT_PAIR;
  }
  return intersection_points(p.get_point(0), p.get_last_point());
}
```

1212. Ellipse argument. TO DO: Read through and explain. [LDF 2002.11.06.]

The *step* argument is used in the case that the **Ellipses** have different centers and/or axis orientation. It is the number of degrees of rotation performed while the algorithm is searching for an intersection. The default is 3, which should work as long as the **Ellipses** don't differ too much in size. [LDF 2004.01.12.]

If the *verbose* argument is *true*, information about the intersection points is printed to standard output. [LDF 2003.07.01.]

If the **Ellipses** are coplanar, the intersection points of the perimeters of the **Ellipses** are returned. If the planes of the **Ellipses** are perpendicular or skew, the intersection line of the planes is found. Then, the intersection points of the this line with the **Ellipses**, if they exist.

TO DO: [LDF 2003.07.20.] The following code found only one intersection:

```
#if 0
  Ellipse t(origin, 5, 4, 90);
  Circle c(origin, 3, 90);
  c.shift(3);
  c.rotate(0, 0, 30);
  bool_point_quadruple bpq = t.intersection_points(c);
#endif
```

1213. When c was rotated by 15° or 45°, *intersection_points*( ) found both intersections. Try to find out why! I want to write a **Circle** version with an **Ellipse** argument, and vice versa. If I do, I may not have to worry about this problem.

1214. [LDF 2003.07.20.]

Log

[LDF 2002.04.12.] Actually, it looks as if the equations Dr. Schwardmann gave me won't do the trick. They assume both ellipses are centered about the origin. When I tried to have Mathematica solve the equations for $x^2/a^2 + y^2/b^2 = (x - m)^2/c^2 + (y - n)^2/d^2$, Mathematica produced solutions that took up over 35,000 lines of text!

[LDF 2002.04.12.] I've rewritten the other cases (parallel, perpendicular, and skew) in this function completely, and it seems like it wasn't really worth the effort. The new version does however make use of **Planes** and **Lines**, which is not a bad thing, and it checks whether the intersection **Points** are on the **Ellipse**, so it wasn't a total loss.

[LDF 2002.04.12.] I've written to Dr. Schwardmann and asked him about this problem. It may just be my ignorance, or perhaps I've overlooked something simple. I'm not too hopeful, however, so I'll probably have to implement the numerical solution I'd started, after all.

[LDF 2002.04.12.] Removed old definition. Dr. Ulrich Schwardmann of the Gesellschaft für wissenschaftliche Datenverarbeitung, Göttingen, Germany showed me how to use Mathematica's "Solve" command to solve the systems of equations that describe the intersections between two ellipses. Therefore, I've removed the old definition of this function, which wasn't complete, anyway, and I'll put in a new definition soon.

[LDF 2002.04.14.] About to start work on implementing the coplanar case again. The equations produced by Mathematica will work for the case that the ellipses have the same center. I will catch the case that they are congruent and have the same center and the case that they are non-congruent and have the same center before going on to implement a numerical solution for the case that they have different centers.

[LDF 2002.04.14.] Changed argument from **const Ellipse** & to **Ellipse**, since I was having to copy it, anyway.

[LDF 2003.07.01.] Added *verbose* argument.

[LDF 2003.07.01.] Changed "perpendicular and skew case". Debugged coplanar case, where the **Ellipses** don't have the same centers and/or axis orientation.

[LDF 2003.07.06.] Made a minor change to the conditional that determines whether the **Ellipses** are coplanar or not.

[LDF 2003.08.14.] Setting *verbose* to *true* if `VERBOSE_GLOBAL` is *true*. Added `VERBOSE_GLOBAL` to `pspglb.web` today.

[LDF 2004.01.12.] Added **real** *step* argument, with 3 as its default value.

⟨ Declare **Ellipse** functions 1146 ⟩ +≡
virtual bool_point_quadruple *intersection_points*(**Ellipse** e, **const real** *step* = 3, **bool** *verbose* = *false*) **const**;

1215.

⟨ Define **Ellipse** functions 1147 ⟩ +≡

```
bool_point_quadruple Ellipse::intersection_points(Ellipse e, const real step, bool verbose) const{
        bool DEBUG = false;      /* true */
    if (VERBOSE_GLOBAL) verbose = true;
    if (DEBUG ∨ verbose)
      cout ≪ "Entering␣Ellipse::intersection_points(Ellipse␣e,␣const␣bool␣verbose)" ≪
          endl;
    bool_point_quadruple bpq = INVALID_BOOL_POINT_QUADRUPLE;
    Plane this_plane = this→get_plane( );
    Plane e_plane = e.get_plane( );
    if (DEBUG) {
      this_plane.show("this_plane:");
      e_plane.show("e_plane:");
    }
    if (this_plane.normal ≡ e_plane.normal ∨ this_plane.normal ≡ −e_plane.normal) {
```

1216. Coplanar case.

Log

[LDF 2003.07.20.] Now using *get_axis_v*( ) and *get_axis_h*( ), instead of accessing *axis_h* and *axis_v* directly.

⟨ Define **Ellipse** functions 1147 ⟩ +≡

```
  if (fabs(fabs(this_plane.distance) − fabs(e_plane.distance)) < Point::epsilon( )) {
  if (DEBUG) {
    cout ≪ "Ellipses␣are␣coplanar.\n" ≪ flush;
  }
  real ax_h = get_axis_h( );
  real ax_v = get_axis_v( );
```

1217. Congruent, same location and axis orientation. [LDF 2002.04.15.] Added check for the axis orientation. If the axes are not all vertical or horizontal, then this fails (floating exception). This means that the circular case will not be caught here, but I plan to program a specialization for **Circles**, so this shouldn't cause too much of a problem.

[LDF 2002.04.14.] Added this section.

This routine only works if the axis orientation of the **Ellipses** is the same, or rotated 90° or 180° about an axis through the center in the direction of the normal to the plane of the **Ellipse**, or in the opposite direction. The axis orientation is tested on the basis of the vector from the center to *points*[0]. If the axis orientation is the same, or rotated 180° about the above-mentioned axis, then *axis_h* of $*this$ must equal *axis_h* of e and *axis_v* of $*this$ must equal *axis_v* of e. If the axis orientation is rotated at an angle of $\pm 90°$, then *axis_h* of $*this$ must equal *axis_v* of e and *axis_v* of $*this$ must equal *axis_h* of e.

The maximum number of intersection points of two non-congruent ellipses is four, so checking five points eliminates the possibility that we could accidentally choose intersection points on two non-congruent ellipses and mistakenly conclude that they are congruent.

```
⟨ Define Ellipse functions 1147 ⟩ +≡
  Point this_axis_orientation(get_point(0));
  this_axis_orientation −= get_center( );
  this_axis_orientation.unit_vector(true);
  Point e_axis_orientation(e.get_point(0));
  e_axis_orientation −= e.get_center( );
  e_axis_orientation.unit_vector(true);
  if (DEBUG) {
    this_axis_orientation.show("this_axis_orientation");
    e_axis_orientation.show("e_axis_orientation");
  }
  Point normal_point(this_plane.normal);
  normal_point.unit_vector(true);
  if (DEBUG) normal_point.show("normal_point");
  Point e_axis_orientation_rotated(e_axis_orientation);
```

1218. [LDF 2002.09.26.] START HERE. Added the **if** condition. I think this should be done, but I should check to be sure. Apparently I haven't programmed the case that the centers aren't the same, but it would be easy enough to do, I think.

⟨ Define **Ellipse** functions 1147 ⟩ +≡

```
if (e_axis_orientation ≠ this_axis_orientation) e_axis_orientation_rotated.rotate(origin, normal_point, 90);
if (DEBUG) e_axis_orientation_rotated.show("e_axis_orientation");
if (this→center ≡ e.center ∧ ((this_axis_orientation ≡ e_axis_orientation ∨ this_axis_orientation ≡
        −e_axis_orientation) ∨ (this_axis_orientation ≡ e_axis_orientation_rotated ∨ this_axis_orientation ≡
        −e_axis_orientation_rotated))) {
  if (DEBUG) cout ≪ "Centers␣and␣axis␣orientation␣are␣the␣same.\n" ≪ flush;
      /* [LDF 2002.04.14.] Pick the maximum of axis_h and axis_v and multiply it by 1.5. We'll use a
         line segment of this length to find intersection points with the two Ellipses. Using this length
         guarantees we'll find them. Actually, max(axis_h, axis_v) ought to do the trick. */
  Point pt0;
  if (ax_h ≥ ax_v) pt0 = get_point(0);
  else pt0 = get_point(number_of_points/4);
  pt0 −= center;
  pt0 *= 1.5;
  pt0.shift(center);
  if (DEBUG) {
    this_plane.normal.show("normal");
    center.show("center");
  }    /* [LDF 2002.04.14.] We'll rotate pt0 around the normal to the plane of the ellipse from center,
          i.e., the line segment from center to pt1. */
  Point pt1(this_plane.normal);
  pt1.shift(center);
  if (DEBUG) pt1.show("pt1");    /* [LDF 2002.04.14.] pt2 is the intersection of the line from center
          to pt0 with *this, and pt3 is the intersection of the same line with e. */
  Point pt2;
  Point pt3;
  bool_point_pair bpp_this;
  bool_point_pair bpp_e;
  bool congruent_flag = true;    /* [LDF 2002.04.14.] We'll find intersection points for five values
      of pt0. If one set of intersection points are not the same, this means the we Ellipses are not
      congruent and in the same location, so we break out of the loop. */
  for (int i = 0; i < 5; i++) {
    if (DEBUG) cout ≪ "i␣==␣" ≪ i ≪ endl ≪ flush;
    if (i ≠ 0) pt0.rotate(center, pt1, 30);
    if (DEBUG) pt0.dotlabel("0");
    bpp_this = intersection_points(center, pt0);
    bpp_e = e.intersection_points(center, pt0);
    if (bpp_this.first.b ≡ true) {
      if (DEBUG) cout ≪ "first␣is␣an␣intersection␣point␣(this).\n";
      pt2 = bpp_this.first.pt;
    }
    else if (bpp_this.second.b ≡ true) {
      if (DEBUG) cout ≪ "second␣is␣an␣intersection␣point␣(this).\n";
      pt2 = bpp_this.second.pt;
    }
    else {
```

```
        cerr ≪ "In␣Ellipse::intersection_points(Ellipse).\n" ≪
            "This␣can't␣happen␣(this)!\n" ≪ "Will␣try␣to␣continue.\n\n" ≪ flush;
        pt2 = INVALID_POINT;
      }
      if (bpp_e.first.b ≡ true) {
        if (DEBUG) cout ≪ "first␣is␣an␣intersection␣point␣(e).\n";
        pt3 = bpp_e.first.pt;
      }
      else if (bpp_e.second.b ≡ true) {
        if (DEBUG) cout ≪ "second␣is␣an␣intersection␣point␣(e).\n";
        pt3 = bpp_e.second.pt;
      }
      else {
        cerr ≪ "In␣Ellipse::intersection_points(Ellipse).\n" ≪
            "This␣can't␣happen␣(e)!\nWill␣try␣to␣continue.\n\n" ≪ flush;
        pt3 = INVALID_POINT;
      }
      if (DEBUG) {
        pt2.show("2");
        pt2.dotlabel("2");
        pt3.show("3");
        pt3.dotlabel("3", "bot");
      }
      if (pt2 ≡ INVALID_POINT ∨ pt3 ≡ INVALID_POINT ∨ pt2 ≠ pt3) {
        if (DEBUG) cout ≪ "Ellipses␣are␣not␣congruent.␣Breaking.\n";
        congruent_flag = false;
        break;
      }
      else continue;
    }   /* for */
    if (DEBUG) {
      cout ≪ "congruent_flag␣==␣" ≪ congruent_flag ≪ endl ≪ flush;
    }
    if (congruent_flag ≡ true) {
      cerr ≪ "WARNING!␣In␣Ellipse::intersection_points(Ellipse).\n" ≪
          "Ellipses␣are␣congruent␣and␣in␣the␣same␣location.\n" ≪
          "Returning␣INVALID_BOOL_POINT_QUADRUPLE.\n\n" ≪ flush;
      return INVALID_BOOL_POINT_QUADRUPLE;
    }
    else if (DEBUG) {
      cout ≪ "All␣five␣points␣are␣not␣on␣both␣ellipses.\n" ≪ flush;
    }
  }   /* End of test of congruency and same location. */
      /* [LDF 2002.09.26.] START HERE. Programm this case!! */
  else if (DEBUG) {
    cout ≪ "The␣centers␣are␣different,␣or␣the␣␣axis␣orientation␣" ≪
        "is␣different,␣or␣both.␣" ≪ "Haven't␣programmed␣this␣case␣yet.\n";
  }
  Ellipse copy(*this);
  Point copy_center(copy.get_center());
  Transform t;
  Transform t_inverse;
```

1219. Shift to origin (if necessary).

⟨ Define **Ellipse** functions 1147 ⟩ +≡

```
if (copy_center ≠ origin) {
  if (DEBUG) {
    cout ≪ "Shifting␣copy␣to␣origin.\n" ≪ flush;
  }
  t.shift(−copy_center.get_x(), −copy_center.get_y(), −copy_center.get_z());
  copy *= t;
  copy_center *= t;
  e *= t;
}
else if (DEBUG) {
  cout ≪ "copy␣is␣already␣at␣origin.\n" ≪ flush;
}
```

1220. Get coordinates of normal.

⟨ Define **Ellipse** functions 1147 ⟩ +≡

```
real normal_x = this_plane.normal.get_x();
real normal_y = this_plane.normal.get_y();
real normal_z = this_plane.normal.get_z();
```

1221. Determine the orientation of the ellipse and rotate, if it's not already in a plane parallel to a major plane. Rotating the **Ellipse** can cause inaccuracies in the coordinate values, so if the ellipse is already in a major plane, (i.e., one perpendicular to a major axis), we leave it where it is.

Log

[LDF 2003.08.27.] Commented-out the declaration of OTHER, because it's never used. I haven't deleted it, just in case.

```
⟨ Define Ellipse functions 1147 ⟩ +≡
  unsigned short orientation;
  const unsigned short X_Y = 0;
  const unsigned short X_Z = 1;
  const unsigned short Z_Y = 2;
#if 0
  const unsigned short OTHER = 3;
#endif
  if (normal_x ≡ 0 ∧ normal_y ≡ 0 ∧ normal_z ≡ 0) {
    cerr ≪ "ERROR!␣In␣Ellipse::intersection_points():\n" ≪ "Normal␣==␣0.␣" ≪
        "Returning␣INVALID_BOOL_POINT_QUADRUPLE\n\n" ≪ flush;
    return INVALID_BOOL_POINT_QUADRUPLE;
  }
  else if (normal_x ≡ 0 ∧ normal_y ≡ 0)    /* Ellipse lies in a plane parallel to x-y plane. */
  {
    if (DEBUG) cout ≪ "Ellipse␣lies␣in␣a␣plane␣parallel␣to␣x-y␣plane\n" ≪ flush;
    orientation = X_Y;
  }
  else if (normal_x ≡ 0 ∧ normal_z ≡ 0)    /* Ellipse lies in a plane parallel to x-z plane. */
  {
    if (DEBUG) cout ≪ "Ellipse␣lies␣in␣a␣plane␣parallel␣to␣x-z␣plane\n" ≪ flush;
    orientation = X_Z;
  }
  else if (normal_z ≡ 0 ∧ normal_y ≡ 0)    /* Ellipse lies in a plane parallel to z-y plane. */
  {
    if (DEBUG) cout ≪ "Ellipse␣lies␣in␣a␣plane␣parallel␣to␣z-y␣plane\n" ≪ flush;
    orientation = Z_Y;
  }
  else    /* Ellipse doesn't lie in a plane parallel to a major plane. */
  {
    if (DEBUG) cout ≪ "Ellipse␣doesn't␣lie␣in␣a␣plane␣parallel␣to␣a␣major␣plane.\n" ≪ flush;
        /* Put it in x-z plane. */
    if (DEBUG) {
      copy_center.dotlabel("c");
    }
    Point ellipse_pt0 = copy.get_point(0);
    Transform t0;
    t0.align_with_axis(copy_center, ellipse_pt0, 'x');
    copy *= t0;
    copy_center *= t0;
    ellipse_pt0 *= t0;
    e *= t0;
    if (DEBUG) {
```

```
      ellipse_pt0.show("ellipse_pt0");
      ellipse_pt0.dotlabel("0");
    }
    Point ellipse_pt4 = copy.get_point(number_of_points/4);
    if (DEBUG) ellipse_pt4.show("ellipse_pt4");
    Transform t1;
    t1.align_with_axis(copy_center, ellipse_pt4, 'z');
    copy *= t1;
    copy_center *= t1;
    e *= t1;
    ellipse_pt4 *= t1;
    t *= t0;
    t *= t1;
    orientation = X_Z;
  }
  t_inverse = t.inverse();
  Point e_center = e.get_center();
  if (DEBUG) {
    copy_center.show("copy_center");
    e_center.show("e_center");
  }
```

1222. Ellipses have the same center and orientation. If they do, then there is an algebraic solution we can apply to find the intersection points.

⟨ Define **Ellipse** functions 1147 ⟩ +≡

```
Point copy_axis_orientation(copy.get_point(0));
copy_axis_orientation −= copy_center;
copy_axis_orientation.unit_vector(true);
e_axis_orientation = e.get_point(0);
e_axis_orientation −= e_center;
e_axis_orientation.unit_vector(true); if ((e_center ≡ origin ∧ copy_center ≡
    origin) ∧ ((copy_axis_orientation ≡ e_axis_orientation ∨ copy_axis_orientation ≡
    −e_axis_orientation) ∨ (copy_axis_orientation ≡ e_axis_orientation_rotated ∨ copy_axis_orientation ≡
    −e_axis_orientation_rotated))) {
if (DEBUG) cout ≪ "Both␣ellipses␣have␣the␣same␣center␣and␣axis␣orientation.\n";
Point pt20;
Point pt21;
Point pt22;
Point pt23;
real x;
real y;
real a = ax_h/2.0;
real b = ax_v/2.0;
real c = e.get_axis_h()/2.0;
real d = e.get_axis_v()/2.0;
if (DEBUG) {
  cout ≪ "a␣==␣" ≪ a ≪ endl ≪ flush;
  cout ≪ "b␣==␣" ≪ b ≪ endl ≪ flush;
  cout ≪ "c␣==␣" ≪ c ≪ endl ≪ flush;
  cout ≪ "d␣==␣" ≪ d ≪ endl ≪ flush;
}
real aa = (a ∗ a);
real bb = (b ∗ b);
real cc = (c ∗ c);
real dd = (d ∗ d);
if (DEBUG) {
  cout ≪ "aa␣==␣" ≪ aa ≪ endl ≪ flush;
  cout ≪ "bb␣==␣" ≪ bb ≪ endl ≪ flush;
  cout ≪ "cc␣==␣" ≪ cc ≪ endl ≪ flush;
  cout ≪ "dd␣==␣" ≪ dd ≪ endl ≪ flush;
}
real denominator;
real numerator;
if (DEBUG) cout ≪ "x␣coordinate.\n";
denominator = (aa − ((bb ∗ cc)/dd)) ∗ dd;
numerator = bb ∗ (aa − cc);
if (DEBUG) {
  cout ≪ "numerator␣==␣" ≪ numerator ≪ endl ≪ flush;
  cout ≪ "denominator␣==␣" ≪ denominator ≪ endl ≪ flush;
}
if (denominator ≡ 0) {
  if (DEBUG) cout ≪ "x␣==␣INVALID_REAL.\n";
```

```
  x = INVALID_REAL;
}
else {
  try {
    x = −(c ∗ sqrt(1 − (numerator/denominator)));
  }
  catch(...)
  {
    x = INVALID_REAL;
    if (DEBUG) cout ≪ "x␣==␣INVALID_REAL\n";
  }
}
if (DEBUG) {
  cout ≪ "x␣==␣" ≪ x ≪ endl ≪ flush;
  cout ≪ "y␣coordinate:\n";
}
numerator = b ∗ sqrt(fabs(aa − cc));
denominator = sqrt(fabs(aa − ((bb ∗ cc)/dd)));
if (DEBUG) {
  cout ≪ "numerator␣==␣" ≪ numerator ≪ endl ≪ flush;
  cout ≪ "denominator␣==␣" ≪ denominator ≪ endl ≪ flush;
}
if (denominator ≡ 0) {
  if (DEBUG) cout ≪ "y␣==␣INVALID_REAL.\n";
  y = INVALID_REAL;
}
else {
  try {
    y = −(numerator/denominator);
  }
  catch(...)
  {
    y = INVALID_REAL;
    if (DEBUG) cout ≪ "y␣==␣INVALID_REAL\n";
  }
}
if (DEBUG) {
  cout ≪ "y␣==␣" ≪ y ≪ endl ≪ flush;
}
```

1223. The ellipses can intersect at no points, two points, or four points.

- If they do not intersect, then one of the ellipses must be inside the other.
- If they intersect at two points, then either $x = 0$ or $y = 0$ (but not both).
- Otherwise, they intersect at four points.

⟨ Define **Ellipse** functions 1147 ⟩ +≡

```
if (x ≡ INVALID_REAL ∨ y ≡ INVALID_REAL) {
  cerr ≪ "WARNING!␣In␣Ellipse::intersection_points(Ellipse).\n" ≪
      "Ellipses␣don't␣intersect.␣" ≪ "Returning␣INVALID_BOOL_POINT_QUADRUPLE.\n\n";
  return INVALID_BOOL_POINT_QUADRUPLE;
}
else if (orientation ≡ X_Y) {
  pt20.set(x, y, 0);
  pt21.set(−x, −y, 0);
  if (y ≠ 0 ∧ x ≠ 0) {
    pt22.set(x, −y, 0);
    pt23.set(−x, y, 0);
  }
  else {
    pt22 = INVALID_POINT;
    pt23 = INVALID_POINT;
  }
}
else if (orientation ≡ X_Z) {
  pt20.set(x, 0, y);
  pt21.set(−x, 0, −y);
  if (y ≠ 0 ∧ x ≠ 0) {
    pt22.set(x, 0, −y);
    pt23.set(−x, 0, y);
  }
  else {
    pt22 = INVALID_POINT;
    pt23 = INVALID_POINT;
  }
}
else if (orientation ≡ Z_Y) {
  pt20.set(0, y, x);
  pt21.set(0, −y, −x);
  if (x ≠ 0 ∧ y ≠ 0) {
    pt22.set(0, y, −x);
    pt23.set(0, −y, x);
  }
  else {
    pt22 = INVALID_POINT;
    pt23 = INVALID_POINT;
  }
}
else {
  cerr ≪ "ERROR!␣In␣Ellipse::intersection_points(Ellipse).\n" ≪
      "This␣can't␣happen!␣orientation␣has␣invalid␣value:␣" ≪ orientation ≪
      endl ≪ "Will␣to␣try␣to␣continue.\n\n" ≪ flush;
}
if (DEBUG) {
```

```
    pt20.show("pt20");
    pt20.dotlabel("20");
    pt21.show("pt21");
    pt21.dotlabel("21");
    pt22.show("pt22");
    pt22.dotlabel("22");
    pt23.show("pt23");
    pt23.dotlabel("23");
  }
  signed short ss_copy;
  signed short ss_e;
  if (pt20 ≠ INVALID_POINT) {
    ss_copy = copy.location(pt20);
    if (DEBUG) cout ≪ "ss_copy␣==␣" ≪ ss_copy ≪ endl ≪ flush;
    ss_e = e.location(pt20);
    if (DEBUG) cout ≪ "ss_e␣==␣" ≪ ss_e ≪ endl ≪ flush;
  }
  if (ss_copy ≡ 0 ∧ ss_e ≡ 0) {
    pt20 *= t_inverse;
    pt21 *= t_inverse;
    bpq.first.b = true;
    bpq.first.pt = pt20;
    bpq.second.b = true;
    bpq.second.pt = pt21;
    if (pt22 ≠ INVALID_POINT) {
      if (DEBUG) cout ≪ "Ellipses␣intersect␣at␣four␣points.\n";
      pt22 *= t_inverse;
      bpq.third.b = true;
      bpq.third.pt = pt22;
    }
    else {
      if (DEBUG) cout ≪ "Ellipses␣intersect␣at␣two␣points.\n";
    }
    if (pt23 ≠ INVALID_POINT) {
      pt23 *= t_inverse;
      bpq.fourth.b = true;
      bpq.fourth.pt = pt23;
    }
    return bpq;
  }   /* if */
  else {
    cerr ≪ "WARNING!␣In␣Ellipse::intersection_points(Ellipse).\n";
    ss_e = e.location(copy.get_point(0));
    if (ss_e ≡ 1) cerr ≪ "*this␣is␣inside␣of␣e.\n";
    else if (ss_e ≡ −1) cerr ≪ "e␣is␣inside␣of␣*this.\n";
    else {
      cerr ≪ "This␣can't␣happen!␣Invalid␣value␣for␣ss_e:␣" ≪ ss_e ≪ endl;
    }
    cerr ≪ "No␣intersection.\n" ≪ "Returning␣INVALID_BOOL_POINT_QUADRUPLE.\n\n";
    return INVALID_BOOL_POINT_QUADRUPLE;
  }
}   /* End of "Ellipses have the same center and axis orientation". */
```

1224. Ellipses have different centers and/or axis orientation. There is no simple algebraic solution for this case, so I have to implement a numerical one here.

⟨ Define **Ellipse** functions 1147 ⟩ +≡

```
else {
  if (DEBUG) cout << "The␣two␣ellipses␣don't␣have␣the␣same␣center␣and/or␣" <<
      "the␣axis␣orientation␣is␣different.\n";
  Point curr_point;
  signed short curr_location;
  signed short location_switch;
  unsigned short intersection_ctr = 0;
  real save_angle;
  real in_angle;
  real out_angle;
  real test_angle;
  if (DEBUG) {
    e.show("e:");
    copy.show("copy:");
  }
  cout << "In␣Ellipse::intersection_points():\n" <<
      "Searching␣for␣Ellipse␣intersections.␣" << "This␣can␣take␣some␣time...\n\n" << flush;
  for (real i = 0; i < 360; i += step) {
    if (DEBUG) cout << "i␣==␣" << i << endl << flush;
    curr_point = copy.angle_point(i);
    if (DEBUG) curr_point.show("curr_point");
    curr_location = e.location(curr_point);
    if (DEBUG) cout << "curr_location␣==␣" << curr_location << endl << flush;
    if (curr_location ≡ 0) {
      if (DEBUG) {
        cout << "Found␣an␣intersection␣point!\n";
        curr_point.show("curr_point:");
      }
      ⟨ Handle intersection point 1225 ⟩
      i += .5;
      save_angle = i;
      curr_point = copy.angle_point(i);
      location_switch = e.location(curr_point);
      continue;
    }   /* if. End of "Found an intersection point". */
    else if (i ≡ 0) {
      location_switch = curr_location;
      save_angle = 0;
      continue;
    }
    else if (curr_location ≡ 1) {
      if (DEBUG) cout << "Point␣is␣inside␣e.\n";
    }
    else if (curr_location ≡ −1) {
      if (DEBUG) cout << "Point␣is␣outside␣e.\n";
    }
    else {
```

```
      cerr ≪ "ERROR!␣In␣Ellipse::intersection_points():\n" ≪
          "This␣can't␣happen!␣curr_location␣has␣bad␣value:␣" ≪ curr_location ≪
          "␣" ≪ "Returning␣INVALID_BOOL_POINT_QUADRUPLE\n\n" ≪ flush;
      return INVALID_BOOL_POINT_QUADRUPLE;
    }
    if (curr_location ≠ location_switch) {
      if (DEBUG) {
        cout ≪ "Found␣a␣transition!\n";
        cout ≪ "i␣==␣" ≪ i ≪ endl ≪ flush;
        cout ≪ "save_angle␣==␣" ≪ save_angle ≪ endl ≪ flush;
        cout ≪ "curr_location␣==␣" ≪ curr_location ≪ endl ≪ flush;
        cout ≪ "location_switch␣==␣" ≪ location_switch ≪ endl ≪ flush;
      }
      if (location_switch ≡ 1) {
        in_angle = save_angle;
        out_angle = i;
      }
      else {
        in_angle = i;
        out_angle = save_angle;
      }
      while (true) {
        test_angle = (in_angle + out_angle)/2;
        curr_point = copy.angle_point(test_angle);
        curr_location = e.location(curr_point);
        if (curr_location ≡ 0) {
          if (DEBUG) {
            cout ≪ "Found␣an␣intersection␣point!\n";
            curr_point.show("curr_point:");
          }
          ⟨ Handle intersection point 1225 ⟩
          i = floor(test_angle);
          i += .5;
          if (i < test_angle) i += .5;
          save_angle = i;
          curr_point = copy.angle_point(i);
          location_switch = e.location(curr_point);
          break;
        }
        else if (curr_location ≡ 1) {
          in_angle = test_angle;
          continue;
        }
        else if (curr_location ≡ −1) {
          out_angle = test_angle;
          continue;
        }
        else {
          cerr ≪ "ERROR!␣In␣Ellipse::intersection_points(Ellipse).\n" ≪
              "This␣can't␣happen!␣Invalid␣value␣for␣curr_location:␣" ≪ curr_location ≪
              "\nReturning␣INVALID_BOOL_POINT_QUADRUPLE.\n\n" ≪ flush;
          return INVALID_BOOL_POINT_QUADRUPLE;
```

```
          }
        }    /* while */
      }    /* if */
    }    /* for */
    cout ≪ "Finished␣searching␣for␣Ellipse␣intersections." ≪ endl ≪ endl ≪ flush;
  }    /* End of “Ellipses have different centers and/or axis orientation”. */
  }    /* End of coplanar case. */
```

1225. Handle intersection point.

⟨ Handle intersection point 1225 ⟩ ≡

```
  {
    ++intersection_ctr;
    if (DEBUG) {
      cout ≪ "intersection_ctr␣==␣" ≪ intersection_ctr ≪ endl ≪ flush;
    }
    if (intersection_ctr ≡ 1) {
      bpq.first.b = true;
      bpq.first.pt = curr_point;
      bpq.first.pt *= t_inverse;
    }
    else if (intersection_ctr ≡ 2) {
      bpq.second.b = true;
      bpq.second.pt = curr_point;
      bpq.second.pt *= t_inverse;
    }
    else if (intersection_ctr ≡ 3) {
      bpq.third.b = true;
      bpq.third.pt = curr_point;
      bpq.third.pt *= t_inverse;
    }
    else if (intersection_ctr ≡ 4) {
      bpq.fourth.b = true;
      bpq.fourth.pt = curr_point;
      bpq.fourth.pt *= t_inverse;
      if (DEBUG ∨ verbose) {
        cout ≪ "In␣Ellipse::intersection_points(Ellipse␣e,␣const␣bool␣verbose)" ≪
            "Found␣fourth␣intersection␣point.\n" ≪ "Returning␣bpq.\n\n";
      }
      return bpq;
    }
    else {
      cerr ≪ "ERROR!␣In␣Ellipse::intersection_points(Ellipse).\n" ≪
          "This␣can't␣happen!␣␣Invalid␣value␣for␣" ≪ "intersection_ctr:␣" ≪
          intersection_ctr ≪ ".\nReturning␣INVALID_BOOL_POINT_QUADRUPLE.\n\n";
      return INVALID_BOOL_POINT_QUADRUPLE;
    }
  }
```

This code is used in section 1224.

1226. Parallel and non-coplanar case.

⟨ Define **Ellipse** functions 1147 ⟩ +≡

```
else {
  cerr ≪ "WARNING!␣In␣Ellipse::intersection_points(Ellipse).\n" ≪
      "Ellipses␣are␣in␣parallel␣planes,␣" ≪ "so␣they␣don't␣intersect.\n" ≪
      "Returning␣INVALID_BOOL_POINT_QUADRUPLE.\n\n";
  return INVALID_BOOL_POINT_QUADRUPLE;
}
}
```

1227. Perpendicular and skew cases. These cases are handled in exactly the same way.

```
⟨ Define Ellipse functions 1147 ⟩ +≡
  else {
    if (DEBUG) cout ≪ "Ellipses␣are␣in␣perpendicular␣or␣skew␣planes.\n" ≪ flush;
    Line isect_line(this_plane.intersection_line(e_plane));
    if (DEBUG) isect_line.show("isect_line:");
    Point pt0(isect_line.direction);
    pt0.shift(isect_line.position);
    bool_point_pair bpp = intersection_points(isect_line.position, pt0);
    if (DEBUG) {
      cout ≪ "bpp.first.b␣==␣" ≪ bpp.first.b ≪ endl ≪ flush;
      bpp.first.pt.show("bpp␣first␣point:");
      cout ≪ "bpp.second.b␣==␣" ≪ bpp.second.b ≪ endl ≪ flush;
      bpp.second.pt.show("bpp␣second␣point:");
    }
    bpq.first.pt = bpp.first.pt;
    bpq.first.b = (bpp.first.pt ≡ INVALID_POINT) ? false : true;
    bpq.second.pt = bpp.second.pt;
    bpq.second.b = (bpp.second.pt ≡ INVALID_POINT) ? false : true;
    bpp = e.intersection_points(isect_line.position, pt0);
    bpq.third.pt = bpp.first.pt;
    bpq.third.b = (bpp.first.pt ≡ INVALID_POINT) ? false : true;
    bpq.fourth.pt = bpp.second.pt;
    bpq.fourth.b = (bpp.second.pt ≡ INVALID_POINT) ? false : true;
    signed short s_t;
    signed short s_e;
    bool temp_bool;
    string temp_string;
    if (bpq.first.b) {
      s_t = location(bpq.first.pt);
      s_e = e.location(bpq.first.pt);
      temp_bool = bpq.first.b;
      temp_string = "First";
      ⟨ Check intersection point locations 1228 ⟩
    }
    else if (verbose) cout ≪ "First␣intersection␣point␣is␣INVALID_POINT.\n";
    if (bpq.second.b) {
      s_t = location(bpq.second.pt);
      s_e = e.location(bpq.second.pt);
      temp_bool = bpq.second.b;
      temp_string = "Second";
      ⟨ Check intersection point locations 1228 ⟩
    }
    else if (verbose) cout ≪ "Second␣intersection␣point␣is␣INVALID_POINT.\n";
    if (bpq.third.b) {
      s_t = location(bpq.third.pt);
      s_e = e.location(bpq.third.pt);
      temp_bool = bpq.third.b;
      temp_string = "Third";
      ⟨ Check intersection point locations 1228 ⟩
```

```
    }
    else if (verbose) cout << "Third intersection point is INVALID_POINT.\n";
    if (bpq.fourth.b) {
      s_t = location(bpq.fourth.pt);
      s_e = e.location(bpq.fourth.pt);
      temp_bool = bpq.fourth.b;
      temp_string = "Fourth";
      ⟨ Check intersection point locations 1228 ⟩
    }
    else if (verbose) cout << "Fourth intersection point is INVALID_POINT.\n";
  }    /* else. End of perpendicular and skew cases. */
  if (DEBUG ∨ verbose) cout << "Exiting Ellipse::intersection_points(Ellipse)\n" << flush;
  return bpq; }
```

1228. Check intersection point locations. This is used in the "Perpendicular and Skew Cases" of *intersection_points*(**Ellipse** *e*, **const bool** *verbose*). [LDF 2003.07.01.]

Log

[LDF 2003.07.01.] Added this section.

```
⟨ Check intersection point locations 1228 ⟩ ≡
  if (temp_bool) {
    if (verbose) {
      if (s_t ≡ 0 ∧ s_e ≡ 0)
        cout << temp_string << " point lies on the perimeter of both ellipses.\n";
      else if (s_t ≡ 0) cout << temp_string << " point lies on the perimeter of *this.\n";
      else if (s_t ≡ −1) cout << temp_string << " point lies outside *this.\n";
      else if (s_t ≡ 1) cout << temp_string << " point lies inside *this.\n";
      else if (s_t ≡ −2) cout << temp_string << " point doesn't lie in the plane of *this.\n";
      if (s_e ≡ 0 ∧ s_t ≠ 0) cout << temp_string << " point lies on the perimeter of e.\n";
      if (s_e ≡ −1) cout << temp_string << " point lies outside e.\n";
      else if (s_e ≡ 1) cout << temp_string << " point lies inside e.\n";
      else if (s_e ≡ −2) cout << temp_string << " point doesn't lie in the plane of e.\n";
    }
  }
```

This code is used in section 1227.

1229. Transformations.

Log

[LDF 2003.04.27.] Finished adding the transformation functions. I already had *shift*( ), now I have the rest of them.

1230. Performing a transformation.

1231. Do transform. [LDF 2003.07.20.] Performs a transformation on $*this$.

If $check \equiv true$, *is_elliptical*( ) is called on $*this$ following the transformation. If the latter causes $*this$ to become non-elliptical, *axis_h*, *axis_v*, *linear_eccentricity*, and *numerical_eccentricity* are set to `INVALID_REAL`, and a warning is issued to *stderr*. *center*, *focus0*, and *focus1* are not set to `INVALID_POINT`. They may no longer really be the center and foci of the (non-elliptical) **Ellipse**, but they may have some use for the programmer and/or user.

If $check \equiv true$, and the transformation does not cause $*this$ to become non-elliptical, *axis_h* and *axis_v* are recalculated.

Log

[LDF 2003.07.20.] Added this function. It makes it possible to perform a transformation on an **Ellipse**, optionally calling *is_elliptical*( ). It is called in *is_elliptical*( ) with $check \equiv false$. This prevents **operator**$*$=( ) and *is_elliptical*( ) from calling each other *ad infinitum*.

[LDF 2003.07.25.] Added code for recalculating *linear_eccentricity*, *focus0* and *focus1*.

[LDF 2003.07.25.] BUG FIX: *axis_h* and *axis_v* were too small by half. Now multiplying by 2.

[LDF 2003.07.25.] Added code for recalculating *numerical_eccentricity*.

⟨ Declare **Ellipse** functions 1146 ⟩ +≡

virtual Transform *do_transform*(**const Transform** &*t*, **bool** *check* = *false*);

1232.

⟨ Define **Ellipse** functions 1147 ⟩ +≡

```
Transform Ellipse::do_transform(const Transform &t, bool check)
{
  bool DEBUG = false;   /* true */
  focus0 *= focus1 *= center *= Path::operator*=(t);
  real old_axis_h = axis_h;
  real old_axis_v = axis_v;
  if (check) {
    if (is_elliptical()) {
      Point c = get_center();
      axis_h = (get_point(0) - c).magnitude() * 2;
      axis_v = (get_point(number_of_points/4) - c).magnitude() * 2;
      real axis_h_half = axis_h/2;
      real axis_v_half = axis_v/2;
      if (fabs(axis_h - old_axis_h) > Point::epsilon() ∨ fabs(axis_v - old_axis_v) > Point::epsilon()) {
        if (DEBUG) cout ≪ "Recalculating␣linear_eccentricity,␣" ≪
            "numerical_eccentricity␣and␣foci.\n" ≪ flush;
        if (axis_h ≥ axis_v) {
          linear_eccentricity = sqrt((axis_h_half * axis_h_half) - (axis_v_half * axis_v_half));
          numerical_eccentricity = linear_eccentricity/axis_h_half;
          focus1 = focus0 = (get_point(0) - get_center());
        }
        else {
          linear_eccentricity = sqrt((axis_v_half * axis_v_half) - (axis_h_half * axis_h_half));
          numerical_eccentricity = linear_eccentricity/axis_v_half;
          focus1 = focus0 = (get_point(number_of_points/4) - get_center());
        }
        focus0.unit_vector(true);
        focus1.unit_vector(true);
        focus0 *= -linear_eccentricity;
        focus1 *= linear_eccentricity;
        focus1 *= focus0.shift(get_center());
      }
      else if (DEBUG)
        cout ≪ "axis_h␣and␣axis_v␣are␣unchanged.\n" ≪ "Not␣recalculating␣foci.\n" ≪ flush;
    }   /* if (is_elliptical()) */
    else {
      cerr ≪ "WARNING!␣In␣Ellipse::do_transform(const␣Transform&):\n" ≪
          "This␣transformation␣has␣made␣*this␣non-elliptical!" ≪ endl ≪ endl ≪ flush;
      axis_h = axis_v = linear_eccentricity = INVALID_REAL;
      numerical_eccentricity = INVALID_REAL;
    }
  }
  return t;
}
```

1233. Operator.

Log

[LDF 2002.04.12.] Added this section.
[LDF 2003.07.20.] Changed, so that it calls *do_transform*( ) with *check* ≡ *true*, so that *is_elliptical*( ) is called.

⟨ Declare **Ellipse** functions 1146 ⟩ +≡
virtual Transform operator∗=(**const Transform** &*t*);

1234.

⟨ Define **Ellipse** functions 1147 ⟩ +≡
Transform Ellipse::**operator**∗=(**const Transform** &*t*)
{
 return *do_transform*(*t*, *true*);
}

1235. Rotation around the main axes.

Log

[LDF 2003.07.25.] Changed, so that *do_transform*( ) is called with *check* ≡ *false*. Rotation can neither change the lengths of *axis_h* or *axis_v*, nor make an **Ellipse** non-elliptical, so there's no need to check ∗*this* after rotation.

⟨ Declare **Ellipse** functions 1146 ⟩ +≡
virtual Transform *rotate*(**const real** x, **const real** $y = 0$, **const real** $z = 0$);

1236.

⟨ Define **Ellipse** functions 1147 ⟩ +≡
Transform Ellipse::*rotate*(**const real** x, **const real** y, **const real** z)
{
 Transform t;
 $t.rotate(x, y, z)$;
 return *do_transform*(*t*, *false*);
}

1237. Scale.

Log

[LDF 2003.07.20.] Added check for whether ∗*this* is still elliptical after the scaling operation.

⟨ Declare **Ellipse** functions 1146 ⟩ +≡
virtual Transform *scale*(**real** x, **real** $y = 1$, **real** $z = 1$);

1238.

⟨ Define **Ellipse** functions 1147 ⟩ +≡

```
Transform Ellipse::scale(real x, real y, real z)
{
  Transform t;
  t.scale(x, y, z);
  return (*this *= t);
}
```

1239. Shear.

Log

[LDF 2003.07.20.] Added check for whether **this* is still elliptical after the shearing operation.

⟨ Declare **Ellipse** functions 1146 ⟩ +≡

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

1240.

⟨ Define **Ellipse** functions 1147 ⟩ +≡

```
Transform Ellipse::shear(real xy, real xz, real yx, real yz, real zx, real zy)
{
  Transform t;
  t.shear(xy, xz, yx, yz, zx, zy);
  return (*this *= t);
}
```

1241. Shift.

1242. real arguments.

Log

[LDF 2003.07.25.] Changed, so that *do_transform*( ) is called with *check* ≡ *false*. Shifting can neither change the lengths of *axis_h* or *axis_v*, nor make an **Ellipse** non-elliptical, so there's no need to check **this* after shifting.

⟨ Declare **Ellipse** functions 1146 ⟩ +≡

```
virtual Transform shift(real xx, real yy = 0, real zz = 0);
```

1243.

⟨ Define **Ellipse** functions 1147 ⟩ +≡

```
Transform Ellipse::shift(real xx, real yy, real zz)
{
  Transform t;
  t.shift(xx, yy, zz);
  return do_transform(t, false);
}
```

1244. Point argument.

⟨ Declare **Ellipse** functions 1146 ⟩ +≡

```
virtual Transform shift(const Point &p);
```

1245.

⟨ Define **Ellipse** functions 1147 ⟩ +≡

```
Transform Ellipse::shift(const Point &p)
{
  return shift(p.get_x(), p.get_y(), p.get_z());
}
```

1246. Shift times.

1247. real arguments.

Log

[LDF 2003.07.25.] Now performing *shift_times*( ) on *focus0* and *focus1* , too.

⟨ Declare **Ellipse** functions 1146 ⟩ +≡

```
virtual void shift_times(real x, real y = 1, real z = 1);
```

1248.

⟨ Define **Ellipse** functions 1147 ⟩ +≡

```
void Ellipse::shift_times(real x, real y, real z)
{
  Path::shift_times(x, y, z);
  focus1.shift_times(x, y, z);
  focus0.shift_times(x, y, z);
  center.shift_times(x, y, z);
  return;
}
```

1249. Point argument.

⟨ Declare **Ellipse** functions 1146 ⟩ +≡

```
virtual void shift_times(const Point &p);
```

1250.

⟨ Define **Ellipse** functions 1147 ⟩ +≡

```
void Ellipse::shift_times(const Point &p)
{
  return shift_times(p.get_x(), p.get_y(), p.get_z());
}
```

1251. Rotatation around an arbitrary axis.

1252. Point arguments.

Log

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

[LDF 2003.07.25.] Changed, so that *do_transform*( ) is called with *check* ≡ *false*. Rotation can neither change the lengths of *axis_h* or *axis_v*, nor make an **Ellipse** non-elliptical, so there's no need to check ∗*this* after rotation.

⟨ Declare **Ellipse** functions 1146 ⟩ +≡
virtual Transform *rotate*(**const Point** &*p0*, **const Point** &*p1*, **const real** *angle* = 180);

1253.

⟨ Define **Ellipse** functions 1147 ⟩ +≡
Transform Ellipse::*rotate*(**const Point** &*p0*, **const Point** &*p1*, **const real** *angle*)
{
  Transform *t*;
  t.*rotate*(*p0*, *p1*, *angle*);
  return *do_transform*(*t*, *false*);
}

1254. Path arguments.

Log

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

⟨ Declare **Ellipse** functions 1146 ⟩ +≡
virtual Transform *rotate*(**const Path** &*p*, **const real** *angle* = 180);

1255.

Log

[LDF 2003.04.27.] Changed *get_point*(1) to *get_last_point*( ).

⟨ Define **Ellipse** functions 1147 ⟩ +≡

```
Transform Ellipse::rotate(const Path &p, const real angle)
{
  if (¬p.is_linear()) {
    cerr ≪ "ERROR!␣In␣Ellipse::rotate(Path,␣real).\n" ≪
        "Path␣is␣not␣a␣line.␣␣Returning␣INVALID_TRANSFORM.\n\n";
    return INVALID_TRANSFORM;
  }
  return rotate(p.get_point(0), p.get_last_point(), angle);
}
```

1256. Rectangles.

1257. Surrounding rectangle.

Log

[LDF 2003.07.18.] Made **const**.

⟨ Declare **Ellipse** functions 1146 ⟩ +≡

```
Rectangle out_rectangle() const;
```

1258.

⟨ Define **Ellipse** functions 1147 ⟩ +≡
```
Rectangle Ellipse::out_rectangle() const
{
    Point C(get_center());
    Point pt0(get_point(0));
    Point pt1(get_point(number_of_points/2));
    Path pa0(pt0, pt1);
    Point pt2(get_point(number_of_points/4));
    Point pt3(get_point(3 * number_of_points/4));
    Point pt4(pt1);
    pt4 -= C;
    Point pt5(pt2);
    pt5.shift(pt4);
    Point pt6(pt3);
    pt6.shift(pt4);
    Point pt7(pt0);
    pt7 -= C;
    Point pt8(pt2);
    pt8.shift(pt7);
    Point pt9(pt3);
    pt9.shift(pt7);
#if 0
    pt0.dotlabel("0");
    pt1.dotlabel("1");
    pt2.dotlabel("2");
    pt3.dotlabel("3");
    pt5.dotlabel("5");
    pt6.dotlabel("6");
    pt8.dotlabel("8");
    pt9.dotlabel("9");
#endif
    Rectangle r(pt6, pt9, pt8, pt5);
    return r;
}
```

1259. Inscribed rectangle.

Log

[LDF 2003.07.18.] Made **const**.

⟨ Declare **Ellipse** functions 1146 ⟩ +≡
```
Rectangle in_rectangle() const;
```

1260.

⟨ Define **Ellipse** functions 1147 ⟩ +≡

```
Rectangle Ellipse::in_rectangle() const
{
  Rectangle r0 = out_rectangle();
  bool_point_pair bpp0 = intersection_points(r0.get_point(0), r0.get_point(2));
  bool_point_pair bpp1 = intersection_points(r0.get_point(1), r0.get_point(3));
  if (bpp0.first.pt ≡ INVALID_POINT ∨ bpp0.second.pt ≡ INVALID_POINT ∨ bpp1.first.pt ≡
        INVALID_POINT ∨ bpp1.second.pt ≡ INVALID_POINT) {
    cerr ≪ "Intersection␣didn't␣work.\n" ≪ "Returning␣empty␣rectangle.\n\n" ≪ flush;
    Rectangle r2;
    return r2;
  }
  Rectangle r1(bpp0.second.pt, bpp1.first.pt, bpp0.first.pt, bpp1.second.pt);
  return r1;
}
```

1261. Draw surrounding rectangle.

Log

[LDF 2003.07.01.] Changed the return value from **void** to **Rectangle**. Now the surrounding **Rectangle** is returned.

[LDF 2003.07.18.] Made **const**.

⟨ Declare **Ellipse** functions 1146 ⟩ +≡

```
Rectangle draw_out_rectangle(const Color &ddraw_color = *Colors::default_color, string
    ddashed = "", string ppen = "", Picture &picture = current_picture) const;
```

1262.

⟨ Define **Ellipse** functions 1147 ⟩ +≡

```
Rectangle Ellipse::draw_out_rectangle(const Color &ddraw_color, string ddashed, string
      ppen, Picture &picture) const
{
  Rectangle r = out_rectangle();
  r.draw(ddraw_color, ddashed, ppen, picture);
  return r;
}
```

1263. Draw inscribed rectangle.

Log

[LDF 2003.07.01.] Changed the return value from **void** to **Rectangle**. Now the inscribed **Rectangle** is returned.

[LDF 2003.07.18.] Made **const**.

⟨ Declare **Ellipse** functions 1146 ⟩ +≡

```
Rectangle draw_in_rectangle(const Color &ddraw_color = *Colors::default_color, string
    ddashed = "", string ppen = "", Picture &picture = current_picture) const;
```

1264.

⟨ Define **Ellipse** functions 1147 ⟩ +≡

Rectangle Ellipse:: *draw_in_rectangle*(**const Color** &*ddraw_color*, **string** *ddashed*, **string** *ppen*, **Picture** &*picture*) **const**
{
  Rectangle r = *in_rectangle*( );
  r.draw(*ddraw_color*, *ddashed*, *ppen*, *picture*);
  return r;
}

1265. Rectangle functions. [LDF 2003.07.18.] TO DO: Add *undraw_in_ellipse*( ), *fill_out_ellipse*( ), etc. Also, I should add versions with the **Picture** argument first.

Log

[LDF 2003.07.18.] Added this section. These functions are declared in `rectangs.web`, but must be defined here, because **Ellipse** is an incomplete type there.

1266. Ellipses.

1267. Surrounding Ellipse.

Log

[LDF 2003.07.18.] Added this function.

⟨ Define **Rectangle** functions 1104 ⟩ +≡
```
Ellipse Rectangle::out_ellipse() const
{
  Point C = get_center();
  Point p0 = get_point(0);
  Point M = get_mid_point(1);
  Point normal = get_normal();
  normal.shift(C);
  real out_distance = (p0 - C).magnitude();
  Transform t;
  real h_length = (get_point(1) - p0).magnitude();
  real v_length = (get_point(3) - p0).magnitude();
  p0 *= M *= C *= t.align_with_axis(C, normal, 'y');
  Point x_axis_pt(1);
  real angle = M.angle(x_axis_pt);
  p0 *= M *= t.rotate(0, angle);
  if (M.unit_vector() ≠ x_axis_pt) {
    cerr << "WARNING! In |Rectangle::in_ellipse()|:\n" << "M is not (1, 0, 0)!\n";
    M.show("M:");
      /* I'd rather output this to stderr, but I don't have a way to do this yet. [LDF 2003.07.18.] */
    cout << endl << flush;
  }
  Ellipse e(origin, h_length, v_length);
  bool_point_pair bpp = e.intersection_points(origin, p0);
  real in_distance;
  if (bpp.first.b) {
    in_distance = bpp.first.pt.magnitude();
  }
  else if (bpp.second.b) {
    in_distance = bpp.second.pt.magnitude();
  }
  else {
    cerr << "ERROR! In |Rectangle::out_ellipse()|:\n" <<
        "Couldn't find intersection point.\n" << "Returning empty Ellipse.\n\n" << flush;
    Ellipse r;
    return r;
  }
  real scale_value = out_distance/in_distance;
  e.scale(scale_value, 0, scale_value);
  e *= t.inverse();
  return e;
}
```

1268. Enclosed Ellipse.

Log

[LDF 2003.07.18.] Added this function.

⟨ Define **Rectangle** functions 1104 ⟩ +≡

```
Ellipse Rectangle::in_ellipse() const
{
  Point C = get_center();
  Point M = get_mid_point(1);
  Point normal = get_normal();
  normal.shift(C);
  Transform t;
  real h_length = (get_point(1) − get_point(0)).magnitude();
  real v_length = (get_point(3) − get_point(0)).magnitude();
  M *= C *= t.align_with_axis(C, normal, 'y');
  Point x_axis_pt(1);
  real angle = M.angle(x_axis_pt);
  M *= t.rotate(0, angle);
  if (M.unit_vector() ≠ x_axis_pt) {
    cerr ≪ "WARNING!␣In␣|Rectangle::in_ellipse()|:\n" ≪ "M␣is␣not␣(1,␣0,␣0)!\n";
    M.show("M:");
      /* I'd rather output this to stderr, but I don't have a way to do this yet. [LDF 2003.07.18.] */
    cout ≪ endl ≪ flush;
  }
  Ellipse e(origin, h_length, v_length);
  e *= t.inverse();
  return e;
}
```

1269. Draw surrounding Ellipse. [LDF 2003.07.18.] TO DO: Add version with **Picture** argument first.

Log

[LDF 2003.07.18.] Added this function.

⟨ Define **Rectangle** functions 1104 ⟩ +≡

```
Ellipse Rectangle::draw_out_ellipse(const Color &ddraw_color, string ddashed, string ppen, Picture
        &picture) const
{
  Ellipse e = out_ellipse();
  e.draw(ddraw_color, ddashed, ppen, picture);
  return e;
}
```

1270. Draw enclosed Ellipse. [LDF 2003.07.18.] TO DO: Add version with **Picture** argument first.

Log

[LDF 2003.07.18.] Added this function.

⟨ Define **Rectangle** functions 1104 ⟩ +≡
Ellipse Rectangle :: *draw_in_ellipse* (**const Color** &*ddraw_color*, **string** *ddashed*, **string** *ppen*, **Picture** &*picture*) **const**
{
Ellipse $e = in\_ellipse()$;
e.draw (*ddraw_color*, *ddashed*, *ppen*, *picture*);
return e;
}

1271. Putting Ellipse together. This is what's compiled.

⟨ Include files 6 ⟩
⟨ Version control identifier 5 ⟩
⟨ Define **class Ellipse** 1143 ⟩
⟨ Define **static Ellipse** data members 1144 ⟩
⟨ Define **Ellipse** functions 1147 ⟩
⟨ Define **Rectangle** functions 1104 ⟩
⟨ Declare non-member template functions for **Ellipse** 1154 ⟩

1272. This is what's written to `ellipses.h`.

⟨ `ellipses.h` 1272 ⟩ ≡
  ⟨ Define **class Ellipse** 1143 ⟩
  ⟨ Declare non-member template functions for **Ellipse** 1154 ⟩

1273. Circle (`circles.web`). It won't be possible to make circles recede to the central vanishing point. !! Get quote from book!!

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 2003.12.01.] Put the version control identifiers back into the release versions, because I've put them in their own RCS repository.

---

format *Circle* *Shape*

⟨ Version control identifier 5 ⟩ +≡
  static string *rcs_id* = `"$Id:␣circles.web,v␣1.8␣2004/01/12␣21:27:22␣lfinsto1␣Exp␣$"`;

1274. 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"
#include "points.h"
#include "lines.h"
#include "planes.h"
#include "paths.h"
#include "curves.h"
#include "polygons.h"
#include "rectangs.h"
#include "ellipses.h"
```

1275. Circle class definition.

1276.

⟨ Define **class Circle** 1276 ⟩ ≡
  class Circle : **public Ellipse** {
    real *radius*;
  public: ⟨ Declare **Circle** functions 1278 ⟩
  };

This code is used in sections 1315 and 1316.

1277. Constructors and setting functions.

1278. Default constructor. No arguments.

⟨ Declare **Circle** functions 1278 ⟩ ≡
  Circle( );

See also sections 1281, 1283, 1290, 1292, 1295, 1297, 1298, 1300, 1302, and 1304.

This code is used in section 1276.

1279.

⟨ Define **Circle** functions 1279 ⟩ ≡
  Circle :: **Circle**( )
  {
    on_free_store = *false*;
    line_switch = *false*;
    cycle_switch = *true*;
  }

See also sections 1282, 1284, 1291, 1293, 1296, 1301, 1303, and 1305.

This code is used in section 1315.

1280. Center, diameters and angles.

1281. Constructor.

⟨ Declare **Circle** functions 1278 ⟩ +≡
  Circle(**const Point** &*ccenter*, **const real** *ddiameter*, **const real** *angle_x* = 0, **const real** *angle_y* = 0,
      const real *angle_z* = 0, **const unsigned short** *nnumber_of_points* = `DEFAULT_NUMBER_OF_POINTS`);

1282.

⟨ Define **Circle** functions 1279 ⟩ +≡
  Circle :: **Circle**(**const Point** &*ccenter*, **const real** *ddiameter*, **const real** *angle_x*, **const real**
      angle_y, **const real** *angle_z*, **const unsigned short** *nnumber_of_points*)
  : *radius*(*ddiameter*/2) {
    on_free_store = *false*;
    line_switch = *false*;
    cycle_switch = *true*;
    center = *ccenter*;
    center.*apply_transform*( );
    axis_h = *axis_v* = *ddiameter*;
    number_of_points = *nnumber_of_points*;
    Ellipse *e*(*center*, *axis_h*, *axis_v*, *angle_x*, *angle_y*, *angle_z*, *number_of_points*);
    **this* = *e*;
  }

1283. Setting function.

Log

[LDF 2003.05.06.] Added the argument *nnumber_of_points*. Without it, this setting function didn't match the constructor above.

⟨ Declare **Circle** functions 1278 ⟩ +≡
void *set*(**const Point** &*ccenter*, **const real** *ddiameter*, **const real** *angle_x* = 0, **const real** *angle_y* = 0, **const real** *angle_z* = 0, **const unsigned short** *nnumber_of_points* = DEFAULT_NUMBER_OF_POINTS);

1284.

⟨ Define **Circle** functions 1279 ⟩ +≡

void Circle::*set*(**const Point** &*ccenter*, **const real** *ddiameter*, **const real** *angle_x*, **const real** *angle_y*, **const real** *angle_z*, **const unsigned short** *nnumber_of_points*)
{
  Circle *c*(*ccenter*, *ddiameter*, *angle_x*, *angle_y*, *angle_z*, *nnumber_of_points*);
  ∗*this* = *c*;
  return;
}

1285. Pseudo-constructor for dynamic allocation.

1286. Pointer argument.

Log

[LDF 2003.07.27.] Made non-inline. Made argument *p* **const Circle** ∗.

[LDF 2003.08.10.] Removed redefinition of the default argument **const Circle** ∗*p* = 0 from the definition. The DEC compiler complained, but GCC didn't.

[LDF 2003.12.30.] Replaced **Circle**::*create_new_circle*( ) with a specialization of **template**⟨**class** *C*⟩ *C*∗*create_new*( ) for **Circle**.

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

[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.

⟨ Declare non-member template functions for **Circle** 1286 ⟩ ≡

Circle ∗*create_new*(**const Circle** ∗*c*);

See also section 1287.

This code is used in sections 1315 and 1316.

1287. Reference argument.

Log

[LDF 2003.07.27.] Made non-inline.

[LDF 2003.12.30.] Replaced **Circle**::*create_new_circle*( ) with a specialization of **template**⟨**class** *C*⟩ *C*∗*create_new*( ) for **Circle**.

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

⟨ Declare non-member template functions for **Circle** 1286 ⟩ +≡

Circle ∗*create_new*(**const Circle** &*c*);

1288. Destructor. [LDF 2002.10.09.] Removed the destructor. **Path**::∼**Path**( ) or **Path**::*clear*( ) should be used instead, unless I add dynamically allocated data members to **Circle** (rather than **Ellipse** or **Path**).

1289. Assignment.

1290. Circle argument. This function returns a reference to ∗*this*, which can be used for further assignment.

Log

[LDF 2002.11.10.] Changed and simplified this function. It now uses **Ellipse**::**operator**=( ).

⟨ Declare **Circle** functions 1278 ⟩ +≡
```
Circle &operator=(const Circle &c);
```

1291.

⟨ Define **Circle** functions 1279 ⟩ +≡
```
Circle &Circle::operator=(const Circle &c)
{
  radius = c.radius;
  Ellipse::operator=(c);
  return *this;
}
```

1292. Ellipse argument. This function returns a reference to *∗this*, which can be used for further assignment.

Log

[LDF 2002.11.10.] Changed and simplified this function. It now uses **Ellipse** :: **operator**=( ).

[LDF 2003.08.14.] Added code for handling the case that *e.axis_v* and *e.axis_h* differ by a small amount, possible due to imprecision (see below).

⟨ Declare **Circle** functions 1278 ⟩ +≡
```
Circle &operator=(const Ellipse &e);
```

1293. If $e.axis\_v \neq e.axis\_h$, it's quite possible that the difference is negligible, and the result of imprecision resulting from the representation of floating point numbers, or calculations performed on them. Therefore, we compare the absolute value of their difference with **Point** :: *epsilon*( ) instead of checking whether they're equal. [LDF 2003.08.14.]

It's also possible that one of them has an integer value, i.e., it has only zeroes following the decimal point, and the other deviates by a small amount. In this case, we want to set *radius* to half of the former, because it's probably the correct value. So, if $e\_axis\_v \equiv floor(e\_axis\_v)$, we set *radius* to $e\_axis\_v/2$. Otherwise, we set *radius* to *e_axis_h* without further ado. If $e\_axis\_h \equiv floor(e\_axis\_h)$, so much the better, and if it isn't, neither it nor *e_axis_v* has an integer value, so it doesn't matter which one we use to set *radius*. [LDF 2003.08.14.]

```
⟨ Define Circle functions 1279 ⟩ +≡
  Circle &Circle::operator=(const Ellipse &e)
  {
    real e_axis_v = e.get_axis_v( );
    real e_axis_h = e.get_axis_h( );
    if (e_axis_v ≠ e_axis_h) {
      if (fabs(e_axis_v − e_axis_h) > Point::epsilon( )) {
        cerr ≪ "ERROR!␣In␣Circle::operator=(Ellipse).\n" ≪ "Ellipse␣has␣unequal␣axes.␣" ≪
            "Can't␣perform␣assignment.␣Returning.\n\n";
        return ∗this;
      }
      else if (e_axis_v ≡ floor(e_axis_v)) radius = e_axis_v/2.0;
      else radius = e_axis_h/2.0;
    }
    else radius = e.get_axis_v( )/2.0;
    Ellipse::operator=(e);
    return ∗this;
  }
```

1294. Returning elements and information.

1295. Is circular. Tests whether ∗*this* is circular. Operations such as *scale*( ) and *shear*( ) can cause **Circles** to become non-circular. [LDF 2003.07.25.]

is_circular( ) first tests whether ∗*this* is planar using *is_planar*( ). If it's not, *false* is returned. Otherwise, the **Point** p is set to ∗(*points*[0]), and **Point** $c =$ *get_center*( ) is is subtracted from p. *p.magnitude*( ) is then stored in **real** *mag0*. [LDF 2003.07.25.]

Then, p is set to each of the other **Points** on the **Circle** in turn, c is subtracted from p, and *p.magnitude*( ) is stored in **real** *mag*. If the absolute value of the difference $mag - mag0$ is greater than **Point** :: *epsilon*( ), *is_circular*( ) immediated returns *false*. If $fabs(mag - mag0) \leq$ **Point** :: *epsilon*( ) for all of the **Points** ∗(*points*[n]) for $n > 0$, *is_circular*( ) returns *true*. [LDF 2003.07.25.]

Log

[LDF 2003.07.25.] Added this function.

```
⟨ Declare Circle functions 1278 ⟩ +≡
  bool is_circular(void) const;
```

1296.

⟨ Define **Circle** functions 1279 ⟩ +≡

```
bool Circle::is_circular(void) const
{
  bool DEBUG = true;    /* false */
  if (DEBUG) cout << "Entering Circle::is_circular().\n";
  if (¬is_planar()) {
    if (DEBUG)
      cerr << "In Circle::is_circular(): " << "*this is non-planar. Returning false." <<
          endl << endl << flush;
    return false;
  }
  Point p;
  Point c = get_center();
  real mag0;
  real mag;
  vector⟨Point *⟩::const_iterator iter = points.begin();
  p = **iter++ − c;
  mag0 = p.magnitude();
  for (int i = 1; iter ≠ points.end(); ++iter) {
    p = **iter − c;
    mag = p.magnitude();
    if (fabs(mag − mag0) > Point::epsilon()) {
      if (DEBUG) cerr << "Point " << i << " doesn't lie on Circle.\n" <<
          "Exiting Circle::is_circular().\n" << "Returning false.\n\n" << flush;
      return false;
    }
    ++i;
  }
  if (DEBUG) cout << "Exiting Circle::is_circular(). Returning true.\n" << flush;
  return true;
}
```

1297. Get radius.

Log

[LDF 2002.05.10.] Added this function.

⟨ Declare **Circle** functions 1278 ⟩ +≡

```
inline real get_radius()
{
  return radius;
}
```

1298. Get diameter. [LDF 2002.05.10.] Added this function.

⟨ Declare **Circle** functions 1278 ⟩ +≡

```
inline real get_diameter()
{
  return (2 * radius);
}
```

1299. Intersections. Neither GCC nor the DEC compiler could resolve a call to *intersection_points*( ) with **Point** arguments to **Ellipse** :: *intersection_points*( ), after a **Circle** version with a **Circle** argument had been declared. TO DO: I didn't think this would happen, so I should probably review the rules governing resolution of calls to functions on objects of derived classes. [LDF 2003.07.09.]

Therefore, I've added **Circle** versions of this function, with **Point** and **Path** arguments, that simply call the **Ellipse** versions, and return their return values. This solves the problem. [LDF 2003.07.09.]

The program executed correctly under Linux, after I recompiled with GCC. However, under Tru65, the program caused a "Memory fault" error. After I removed the object files, and recompiled (with the DEC compiler), the problem disappeared. [LDF 2003.07.09.]

[LDF 2003.07.18.] TO DO: Add **Circle** :: *intersection_points*(**const Ellipse** &) and **Ellipse** :: *intersection_points*(**const Circle** &).

1300. Point argument.

Log

[LDF 2003.07.09.] Added this function.

⟨ Declare **Circle** functions 1278 ⟩ +≡
virtual bool_point_pair *intersection_points*(**const Point** &*pt0*, **const Point** &*pt1*) **const**;

1301.

⟨ Define **Circle** functions 1279 ⟩ +≡
bool_point_pair Circle :: *intersection_points*(**const Point** &*pt0*, **const Point** &*pt1*) **const**
{
return Ellipse :: *intersection_points*(*pt0*, *pt1*);
}

1302. Path argument.

Log

[LDF 2003.07.09.] Added this function.

⟨ Declare **Circle** functions 1278 ⟩ +≡
virtual bool_point_pair *intersection_points*(**const Path** &*p*) **const**;

1303.

⟨ Define **Circle** functions 1279 ⟩ +≡
bool_point_pair Circle :: *intersection_points*(**const Path** &*p*) **const**
{
return Ellipse :: *intersection_points*(*p*);
}

1304. Circle argument.

Log

[LDF 2003.07.20.] Wrote the definition of this function. Tested all cases. It should probably be tested more thoroughly.

[LDF 2003.08.14.] Made *verbose* argument non-**const**. Setting *verbose* to *true* if `VERBOSE_GLOBAL` is *true*. Added `VERBOSE_GLOBAL` to `pspglb.web` today.

[LDF 2003.08.27.] Removed the declaration **real** *c_radius* = *c.radius*, since *c_radius* was never used.

⟨ Declare **Circle** functions 1278 ⟩ +≡
virtual bool_point_quadruple *intersection_points*(**const Circle** &*c*, **bool** *verbose* = *false*) **const**;

1305.

⟨ Define **Circle** functions 1279 ⟩ +≡

```
bool_point_quadruple Circle::intersection_points(const Circle &c, bool verbose) const
{
  bool DEBUG = false;    /* true */
  if (VERBOSE_GLOBAL) verbose = true;
  if (DEBUG ∨ verbose) cout ≪ "**********************\n" ≪
      "\nEntering␣Circle::intersection_points(const␣Circle&)\n\n";
  bool_point_quadruple bpq = INVALID_BOOL_POINT_QUADRUPLE;
  Plane this_plane = get_plane();
  Plane c_plane = c.get_plane();
  if (¬(this_plane.normal ≡ c_plane.normal ∨ this_plane.normal ≡ −c_plane.normal)) {
    if (verbose ∨ DEBUG) cout ≪ "Circles␣are␣non-coplanar.␣" ≪
        "Calling␣Ellipse::intersection_points()\n\n" ≪ flush;
    bpq = Ellipse::intersection_points(c, verbose);
    if (DEBUG)
      cout ≪ "Exiting␣Circle::intersection_points(const␣Circle&)" ≪ endl ≪ endl ≪ flush;
    return bpq;
  }
  else {
    if (verbose ∨ DEBUG) cout ≪ "Circles␣are␣coplanar.\n";
    real dist = (c.center − center).magnitude();
    if (DEBUG) cout ≪ "dist␣==␣" ≪ dist ≪ endl ≪ flush;
    if (dist ≡ 0) {
      if (verbose ∨ DEBUG) cout ≪ "Circles␣have␣same␣center.\n";
      if (radius ≡ c.radius) {
        if (verbose ∨ DEBUG) {
          cout ≪ "Circles␣are␣congruent.␣" ≪ "Returning␣INVALID_BOOL_POINT_QUADRUPLE.\
              \n\n" ≪ flush;
        }
        if (DEBUG) cout ≪ "\nExiting␣Circle::intersection_points(const␣Circle&)." ≪
              endl ≪ endl ≪ flush;
        return INVALID_BOOL_POINT_QUADRUPLE;
      }
      else if (radius > c.radius) {
        if (verbose ∨ DEBUG) {
          cout ≪ "*this␣and␣c␣lie␣outside␣of␣each␣other.␣No␣intersections." ≪ endl ≪
              "Returning␣INVALID_BOOL_POINT_QUADRUPLE.\n\n" ≪ flush;
        }
        if (DEBUG) cout ≪ "\nExiting␣Circle::intersection_points(const␣Circle&)." ≪
              endl ≪ endl ≪ flush;
        return INVALID_BOOL_POINT_QUADRUPLE;
      }
      else if (radius < c.radius) {
        if (verbose ∨ DEBUG) {
          cout ≪ "*this␣lies␣inside␣c.␣No␣intersections." ≪ endl ≪
              "Returning␣INVALID_BOOL_POINT_QUADRUPLE.\n\n" ≪ flush;
        }
        if (DEBUG) cout ≪ "\nExiting␣Circle::intersection_points(const␣Circle&)." ≪
              endl ≪ endl ≪ flush;
        return INVALID_BOOL_POINT_QUADRUPLE;
```

```
    }
    else {
      cerr ≪ "ERROR!␣In␣" ≪ "|Circle::intersection_points(const␣Circle&)|" ≪ endl ≪
          "This␣can't␣happen!" ≪ "radius␣and/or␣c.radius␣have␣invalid␣values:\n" ≪
          "radius␣=␣" ≪ radius ≪ endl ≪ "c.radius␣=␣" ≪ c.radius ≪ endl ≪
          "Returning␣INVALID_BOOL_POINT_QUADRUPLE.\n\n" ≪ flush;
      if (DEBUG) cout ≪ "\nExiting␣Circle::intersection_points(const␣Circle&)." ≪
            endl ≪ endl ≪ flush;
      return INVALID_BOOL_POINT_QUADRUPLE;
    }
  }   /* if (dist ≡ 0) */
  else if (dist > (radius + c.radius)) {
    if (verbose ∨ DEBUG)
      cout ≪ "*this␣and␣c␣lie␣outside␣of␣each␣other.␣" ≪ "No␣intersections.\n" ≪
          "Returning␣INVALID_BOOL_POINT_QUADRUPLE.\n\n" ≪ flush;
    if (DEBUG) cout ≪ "\nExiting␣Circle::intersection_points(const␣Circle&)." ≪ endl ≪
          endl ≪ flush;
    return INVALID_BOOL_POINT_QUADRUPLE;
  }
  else if (dist ≡ (radius + c.radius)) {
    if (verbose ∨ DEBUG) cout ≪ "this␣and␣c␣have␣a␣tangent␣on␣the␣outside␣" ≪
          "(one␣intersection␣point).\n";
    bpq.first.pt.set(c.center − center);
    bpq.first.pt.unit_vector(true);
    bpq.first.pt *= radius;
    bpq.first.pt.shift(center);
    bpq.first.b = true;
    if (DEBUG) cout ≪ "\nExiting␣Circle::intersection_points(const␣Circle&)." ≪ endl ≪
          endl ≪ flush;
    return bpq;
  }
  else if (dist ≡ max(radius, c.radius) − min(radius, c.radius)) {
    if (verbose ∨ DEBUG) cout ≪ "*this␣and␣c␣have␣a␣tangent␣on␣the␣inside␣" ≪
          "(one␣intersection␣point).\n";
    if (radius > c.radius) {
      if (verbose ∨ DEBUG) cout ≪ "c␣lies␣within␣*this.\n";
      bpq.first.pt.set(c.center − center);
    }
    else {
      if (verbose ∨ DEBUG) cout ≪ "*this␣lies␣within␣c.\n";
      bpq.first.pt.set(center − c.center);
    }
    bpq.first.pt.unit_vector(true);
    bpq.first.pt *= radius;
    bpq.first.pt.shift(center);
    bpq.first.b = true;
    if (DEBUG) cout ≪ "\nExiting␣Circle::intersection_points(const␣Circle&)." ≪ endl ≪
          endl ≪ flush;
    return bpq;
  }
  else if (dist < (radius + c.radius)) {
    if (dist > max(radius, c.radius) − min(radius, c.radius)) {
```

```
      if (verbose ∨ DEBUG) cout ≪ "t␣and␣c␣have␣2␣intersections.\n";
      real a = radius;
      real bb = c.radius * c.radius;
      real beta = 2 * atan(sqrt((bb − ((dist − a) * (dist − a)))/(((dist + a) * (dist + a)) − bb)));
      beta *= 180/PI;
      if (verbose ∨ DEBUG) cout ≪ "beta␣==␣" ≪ beta ≪ endl ≪ flush;
      Point P(c.center − center);
      P.unit_vector(true);
      P *= radius;
      P.shift(center);
      Point normal = get_normal( );
      normal.shift(center);
      if (DEBUG) {
        P.dotlabel("P");
        P.show("P");
        normal.show("normal");
      }
      bpq.first.b = true;
      bpq.first.pt = P;
      bpq.first.pt.rotate(center, normal, beta);
      if (DEBUG) bpq.first.pt.show("bpq.first.pt");
      bpq.second.b = true;
      bpq.second.pt = P;
      bpq.second.pt.rotate(center, normal, −beta);
      if (DEBUG) bpq.second.pt.show("bpq.second.pt");
      if (DEBUG) cout ≪ "\nExiting␣Circle::intersection_points(const␣Circle&)." ≪
            endl ≪ endl ≪ flush;
      return bpq;
    }
    else if (radius > c.radius) {
      if (verbose ∨ DEBUG)
        cout ≪ "c␣lies␣within␣*this,␣different␣centers␣" ≪ "(no␣intersections).\n" ≪
            "Returning␣INVALID_BOOL_POINT_QUADRUPLE.\n";
      if (DEBUG) cout ≪ "\nExiting␣Circle::intersection_points(const␣Circle&)." ≪
            endl ≪ endl ≪ flush;
      return INVALID_BOOL_POINT_QUADRUPLE;
    }
    else {
      if (verbose ∨ DEBUG)
        cout ≪ "*this␣lies␣within␣c,␣different␣centers␣" ≪ "(no␣intersections).\n" ≪
            "Returning␣INVALID_BOOL_POINT_QUADRUPLE.\n";
      if (DEBUG) cout ≪ "\nExiting␣Circle::intersection_points(const␣Circle&)." ≪
            endl ≪ endl ≪ flush;
      return INVALID_BOOL_POINT_QUADRUPLE;
    }
  }    /* else if (dist < (radius + c.radius)) */
  else {
    cerr ≪ "ERROR!␣In␣Circle::intersection_points(const␣Circle&):" ≪
        "This␣can't␣happen!\nThis␣case␣isn't␣accounted␣for.\n" ≪
        "Returning␣INVALID_BOOL_POINT_QUADRUPLE." ≪ endl ≪ flush;
```

```
        if (DEBUG) cout << "\nExiting Circle::intersection_points(const Circle&)." << endl <<
              endl << flush;
        return INVALID_BOOL_POINT_QUADRUPLE;
      }
    }    /* else (Coplanar case). */
  }
```

1306. Reg_Polygon functions. [LDF 2003.06.13.] The functions in this section are declared in `polygons.web`. They must be defined here, because **Circle** is an incomplete type there.

Log

[LDF 2003.06.13.] Added this section.

1307. Enclosed circle.

Log

[LDF 2003.06.13.] Added this function.
[LDF 2003.12.09.] Changed call to **Point**::*mediate*( ) below. It's now a member function.

⟨ Define **Reg_Polygon** functions 1071 ⟩ +≡

```
  Circle Reg_Polygon::in_circle( ) const
  {
    Circle c;
    if (¬is_planar( )) {
      cerr << "ERROR! In Reg_Polygon::in_circle():\n" << "Reg_Polygon is non-planar.\n" <<
          "Returning empty Circle.\n\n" << flush;
      return c;
    }
    if (points.size( ) < 3) {
      cerr << "ERROR! In Reg_Polygon::in_circle():\n" <<
          "Reg_Polygon has less than 3 Points.\n" << "Returning empty Circle.\n\n" << flush;
      return c;
    }
    Point mid_pt = points[0]→mediate(*points[1]);
    mid_pt −= center;
    real r = mid_pt.magnitude( );
    c.set(origin, 2 * r);
    Point normal = get_normal( );
    normal.shift(center);
    Transform t;
    t.align_with_axis(center, normal, 'y');
    c *= t.inverse( );
    return c;
  }
```

1308. Draw enclosed circle.

1309. Normal version.

⟨ Define **Reg_Polygon** functions 1071 ⟩ +≡

```
Circle Reg_Polygon::draw_in_circle(const Color &ddraw_color, const string ddashed, const string
        ppen, Picture &picture) const
{
  Circle c = in_circle();
  c.draw(ddraw_color, ddashed, ppen, picture);
  return c;
}
```

1310. Picture argument first.

⟨ Define **Reg_Polygon** functions 1071 ⟩ +≡

```
Circle Reg_Polygon::draw_in_circle(Picture &picture, const Color &ddraw_color, const string
        ddashed, const string ppen) const
{
  return draw_in_circle(ddraw_color, ddashed, ppen, picture);
}
```

1311. Surrounding circle.

Log

[LDF 2003.06.13.] Added this function.

⟨ Define **Reg_Polygon** functions 1071 ⟩ +≡

```
Circle Reg_Polygon::out_circle() const
{
  Circle c;
  if (¬is_planar()) {
    cerr ≪ "ERROR!␣In␣Reg_Polygon::out_circle():\n" ≪ "Reg_Polygon␣is␣non-planar.\n" ≪
        "Returning␣empty␣Circle.\n\n" ≪ flush;
    return c;
  }
  if (points.size() < 3) {
    cerr ≪ "ERROR!␣In␣Reg_Polygon::out_circle():\n" ≪
        "Reg_Polygon␣has␣less␣than␣3␣Points.\n" ≪ "Returning␣empty␣Circle.\n\n" ≪ flush;
    return c;
  }
  Point normal = get_normal();
  normal.shift(center);
  c.set(origin, 2 * radius);
  Transform t;
  t.align_with_axis(center, normal, 'y');
  c *= t.inverse();
  return c;
}
```

1312. Draw surrounding circle.

1313. Normal version.

Log

[LDF 2003.06.13.] Added this function.

⟨ Define **Reg_Polygon** functions 1071 ⟩ +≡
Circle Reg_Polygon :: *draw_out_circle*(**const Color** &*ddraw_color*, **const string** *ddashed*, **const string** *ppen*, **Picture** &*picture*) **const**
{
  Circle $c =$ *out_circle*( );
  c.draw(*ddraw_color*, *ddashed*, *ppen*, *picture*);
  return c;
}

1314. Picture argument first.

Log

[LDF 2003.06.13.] Added this function.

⟨ Define **Reg_Polygon** functions 1071 ⟩ +≡
Circle Reg_Polygon :: *draw_out_circle*(**Picture** &*picture*, **const Color** &*ddraw_color*, **const string** *ddashed*, **const string** *ppen*) **const**
{
  return *draw_out_circle*(*ddraw_color*, *ddashed*, *ppen*, *picture*);
}

1315. Putting Circle together. This is what's compiled.

⟨ Include files 6 ⟩
⟨ Version control identifier 5 ⟩
⟨ Define **class Circle** 1276 ⟩
⟨ Define **Circle** functions 1279 ⟩
⟨ Define **Reg_Polygon** functions 1071 ⟩
⟨ Declare non-member template functions for **Circle** 1286 ⟩

1316. This is what's written to `circles.h`.

⟨ `circles.h` 1316 ⟩ ≡
⟨ Define **class Circle** 1276 ⟩
⟨ Declare non-member template functions for **Circle** 1286 ⟩

1317. Patterns (`patterns.web`). [LDF 2002.09.21.] NOTE: When you add a new `.web` file and move code to it by copying it from another `.web` file, remember to change the name of the header file that it writes. Otherwise, this can cause problems and it's not obvious what's caused them.

Log

[LDF 2002.09.21.] Started using this file again. Moved *hex_pattern_1* ( ) here. Made the appropriate changes to `cmplprsp.web` and `main.web`.

[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* = "$Id:␣patterns.web,v␣1.4␣2004/01/12␣21:31:19␣lfinsto1␣Exp␣$";

1318. Include files.

⟨ Include files 6 ⟩ +≡
```
#include "loader.h"
#include "pspglb.h"
#include "io.h"
#include "colors.h"
#include "transfor.h"
#include "shapes.h"
#include "pictures.h"
#include "points.h"
#include "lines.h"
#include "planes.h"
#include "paths.h"
#include "curves.h"
#include "polygons.h"
#include "rectangs.h"
#include "ellipses.h"
#include "circles.h"
```

1319. Plane tesselations.

1320. Hexagonal tesselation 1.

[LDF 2003.08.10.] TO DO: Change, so that it's possible to put the pattern into a rectangular area.

Log

[LDF 2002.09.22.] Added arguments. *hex_pattern_1* ( ) can now be used to make a pattern with up to three nested hexagons which can be drawn and filled. Each hexagon has its own argument for the draw color, the fill color and the pen to be used. *hex_pattern_1* ( ) does not have arguments for dash patterns, but they could be added, if necessary. If the argument for the diameter of the middle or the inner hexagon is 0, then that hexagon is not drawn or filled. In this case, the arguments for the draw and fill colors are ignored for that hexagon. The default pen for the outer hexagon is thicker than the pens for the other two (.5mm and .3mm, respectively).

[LDF 2002.09.20.] Rewrote this function. It now works with the new *project*( ) function. *hex_pattern_1* ( ) makes a "honeycomb" pattern on the x-z plane using a single hexagon (i.e., there aren't nested hexagons yet, as in the old version). In the next version, I plan to add arguments for optionally putting two smaller hexagons inside the large one, and for filling and unfilling.

This version contains arguments for the drawing command, including a **Picture** argument, so that *hex_pattern_1* ( ) need not be put onto *current_picture*.

⟨ Declare Pattern functions 1320 ⟩ ≡

```
unsigned int hex_pattern_1(real diameter_outer = 5, real diameter_middle = 0, real
      diameter_inner = 0, unsigned short first_row = 5, unsigned short double_rows = 10, unsigned
      short row_shift = 2,
      /* Arguments for the drawing and filling commands. */
   Color draw_color_outer = *Colors::default_color,    /* Outer */
   Color fill_color_outer = *Colors::background_color, Color draw_color_middle = *Colors::default_color,
     /* Middle */
   Color fill_color_middle = *Colors::background_color, Color draw_color_inner = *Colors::default_color,
     /* Inner */
   Color fill_color_inner = *Colors::background_color, string pen_outer = "pencircle␣scaled␣.5mm",
      string pen_middle = "pencircle␣scaled␣.3mm", string pen_inner = "pencircle␣scaled␣.3mm",
      Picture &picture = current_picture, unsigned int max_hexagons = 1000);
```

See also sections 1323 and 1326.

This code is used in sections 1329 and 1330.

1321.

⟨ Define Pattern functions 1321 ⟩ ≡
unsigned int *hex_pattern_1* (**real** *diameter_outer*, **real** *diameter_middle*, **real** *diameter_inner*, **unsigned short** *first_row*, **unsigned short** *double_rows*, **unsigned short** *row_shift*,
/* Arguments for the drawing and filling commands. */
Color *draw_color_outer*, /* Outer */
Color *fill_color_outer*, **Color** *draw_color_middle*, /* Middle */
Color *fill_color_middle*, **Color** *draw_color_inner*, /* Inner */
Color *fill_color_inner*, **string** *pen_outer*, **string** *pen_middle*, **string** *pen_inner*, **Picture** &*picture*, **unsigned int** *max_hexagons*)
{
bool DEBUG = *false*; /* *true* */
if (DEBUG) *cout* ≪ "Entering␣hex_pattern_1().\n" ≪ *flush*;
if (*first_row* < 1) {
cerr ≪ "ERROR!␣In␣hex_pattern_1():\n" ≪ "first_row␣has␣invalid␣value:␣" ≪ *first_row* ≪ "It␣must␣be␣strictly␣positive.\n" ≪ "Taking␣absolute␣value.\n" ≪ *flush*;
first_row = *abs*(*first_row*);
cerr ≪ "Now␣first_row␣==␣" ≪ *first_row* ≪ "." ≪ *endl* ≪ *flush*;
}
if (*first_row* > 25) /* [LDF 2002.09.22.] This can't be **else if** because the preceding condition might have produced a value > 25. */
{
cerr ≪ "ERROR!␣In␣hex_pattern_1():\n" ≪ "first_row␣has␣invalid␣value:␣" ≪ *first_row* ≪ "It␣can␣be␣at␣most␣25.\n" ≪ "Setting␣first_row␣to␣25.\n" ≪ *flush*;
first_row = 25;
}
if (*fill_color_outer* ≡ ∗**Colors** :: *default_color*)
/* LDF 2002.09.24. Changed this, because I'm now using **class Color** instead of **strings**. Now it looks like I'm going to have to use "∗**Colors** :: *background_color*" as a placeholder. I'm going to have to check to see what the consequences of this change are here. !! [LDF 2002.09.22.] This is necessary, because **Path** :: *fill*( ) interprets """" as "black". In *hex_pattern_1*( ), it may be necessary to have a placeholder for a *fill_color*, and it's better to be able to use """" than to have to type "*background*". */
fill_color_outer = ∗**Colors** :: *background_color*;
if (*fill_color_middle* ≡ ∗**Colors** :: *default_color*) *fill_color_middle* = ∗**Colors** :: *background_color*;
if (*fill_color_inner* ≡ ∗**Colors** :: *default_color*) *fill_color_inner* = ∗**Colors** :: *background_color*;

bool *do_middle*; /* [LDF 2002.09.22.] Having *do_middle* and *do_inner* is a convenience, so I don't have to check whether *diameter_middle* and *diameter_inner* are ≡ 0 below, which wouldn't be as clearly understandable. */
bool *do_inner*;

do_middle = (*diameter_middle* ≡ 0) ? *false* : *true*;
do_inner = (*diameter_inner* ≡ 0) ? *false* : *true*;

Point *pt0*; /* origin. */
Reg_Polygon *p_outer*(*pt0*, 6, *diameter_outer*);
Reg_Polygon *p_middle*;
Reg_Polygon *p_inner*; /* [LDF 2002.09.22.] The middle and inner hexagons are only set (and used) if *do_middle* and/or *do_inner* are true. */

if (*do_middle*) *p_middle.set*(*pt0*, 6, *diameter_middle*);
if (*do_inner*) *p_inner.set*(*pt0*, 6, *diameter_inner*);

Reg_Polygon *p_outer_copy*; /* These **Reg_Polygons** are used for copying. */

```
    Reg_Polygon p_middle_copy;
    Reg_Polygon p_inner_copy;
#if 0
    if (DEBUG) p_outer.dotlabel();
#endif
    real right_shift = p_outer.get_point(4).get_x() + p_outer.get_point(3).get_x() −
        p_outer.get_point(1).get_x() − p_outer.get_point(2).get_x();
    real left_shift = −right_shift;   /* Center the first row on the origin. */
    Transform t;
    t.shift((first_row/2) * left_shift);
    if (first_row % 2 ≡ 0) t.shift(.5 * right_shift);
    p_outer *= t;
    if (do_middle) p_middle *= t;
    if (do_inner) p_inner *= t;
    Transform offset;
      /* [LDF 2002.09.22.] offset is for moving the hexagons to the second row in the double row, which is
         offset with respect to the first (and will contain one more set of nested hexagons). */
    offset.shift(p_outer.get_point(4) − p_outer.get_point(2));
    Transform move_back;   /* [LDF 2002.09.22.] move_back is for moving in the direction of the
         positive z-axis before starting the next double row. */
    move_back.shift(p_outer.get_point(5) − p_outer.get_point(3));
      /* [LDF 2002.09.22.] The number of sets of hexagons in the rows differs, so we use i_min and i_max
         for controlling the for loop that shifts and draws and fills the hexagons. */
    signed short i_min = 0;
    unsigned short i_max = first_row;
      /* [LDF 2002.09.20.] I could just use first_row instead of declaring a new variable, but the name
         i_max makes more sense in the loop, so I think it's worth doing for the sake of clarity. */
    short i, j, k;
    unsigned int hexagon_ctr = 0;   /* Each time a hexagon is drawn, hexagon_ctr is incremented.
         hexagon_ctr is the return value of this function (hex_pattern_1()). */
    for (k = 0; k < double_rows; ++k)   /* k is the number of double lines. [LDF 2002.09.22.] This loop
         takes care of moving back in the direction of the positive z-axis. */
    {   /* [LDF 2002.09.20.] If 0 is passed as the row_shift argument, don't do any shifting. Otherwise,
           every row_shift rows, increase the number of hexagons in the rows by 2. The rows remain
           centered around the z-axis. row_shift applies to the double rows. The offset row which is drawn
           when j ≡ 1 already has one hexagon more than the first row drawn, so when row_shift ≡ 1, the
           effect is that each single row is one hexagon longer than the last. This makes the edges recede
           diagonally. */
      if (k ≠ 0 ∧ row_shift ≠ 0 ∧ k % row_shift ≡ 0) {
        −−i_min;
        ++i_max;
        t = p_outer.shift(left_shift);
        if (do_middle) p_middle *= t;
        if (do_inner) p_inner *= t;
      }
      for (j = 0; j < 2; ++j)   /* This loop makes the second line in each set of double lines. */
      {
        p_outer_copy = p_outer;
        if (do_middle) p_middle_copy = p_middle;
        if (do_inner) p_inner_copy = p_inner;
```

```
        if (j ≡ 1) {
          p_outer_copy *= offset;
          if (do_middle) p_middle_copy *= offset;
          if (do_inner) p_inner_copy *= offset;
          --i_min;
          t = p_outer_copy.shift(left_shift);
          if (do_middle) p_middle_copy *= t;
          if (do_inner) p_inner_copy *= t;
        }
        for (i = i_min; i < i_max; ++i)
          /* [LDF 2002.09.22.] This loop draws and/or fills the horizontal rows. */
        {
          if (fill_color_outer ≡ *Colors::background_color)
            p_outer_copy.draw(draw_color_outer, "", pen_outer, picture);
          else if (draw_color_outer ≡ fill_color_outer) p_outer_copy.fill(fill_color_outer, picture);
          else p_outer_copy.filldraw(draw_color_outer, fill_color_outer, "", pen_outer, picture);
          if (do_middle) {
            if (fill_color_middle ≡ fill_color_outer)
              p_middle_copy.draw(draw_color_middle, "", pen_middle, picture);
            else if (draw_color_middle ≡ fill_color_middle) p_middle_copy.fill(fill_color_middle, picture);
            else p_middle_copy.filldraw(draw_color_middle, fill_color_middle, "", pen_middle, picture);
          }
          if (do_inner) {
            if (fill_color_inner ≡ fill_color_middle)
              p_inner_copy.draw(draw_color_inner, "", pen_inner, picture);
            else if (draw_color_inner ≡ fill_color_middle) p_inner_copy.fill(fill_color_inner, picture);
            else p_inner_copy.filldraw(draw_color_inner, fill_color_inner, "", pen_inner, picture);
          }
          ++hexagon_ctr;
          if (hexagon_ctr ≥ max_hexagons) {
            cerr << "ERROR! In hex_pattern_1():\n" << "Too many sets of hexagons: " <<
                hexagon_ctr << ".  Returning." << endl << flush;
            return hexagon_ctr;
          }
          t = p_outer_copy.shift(right_shift);
          if (do_middle) p_middle_copy *= t;
          if (do_inner) p_inner_copy *= t;
        }
      }
      p_outer *= move_back;
      if (do_middle) p_middle *= move_back;
      if (do_inner) p_inner *= move_back;
      ++i_min;
    }
    if (DEBUG) cout << "Exiting hex_pattern_1().\n" << flush;
    return hexagon_ctr;
  }
```

See also sections 1324 and 1327.

This code is used in section 1329.

1322. patterns.

1323. Epicycloid pattern 1. [LDF 2003.02.11.] This functions works well for outer **Circles** with radii that are divisors (with no remainder) of the radius of the inner **Circle**. Each outer **Circle** is rolled around the inner **Circle** once only. If the radius of the outer **Circle** is a divisor of the inner **Circle**, the end of the epicycloid will meet up with the beginning. If the radius of the outer **Circle** is not a divisor of the inner **Circle**, it won't. See *epicycloid_pattern_3*( ) below, for a pattern that works well for outer **Circles**, whose radii are not divisors of the radius of the inner **Circle**. !! START HERE. TO DO: Start from beginning of **Color** ∗ vector, if I get to the end.

Log

[LDF 2003.02.09.] Added this function.
[LDF 2003.08.27.] Removed the declaration **const Color** ∗*curr_color*, since *curr_color* was never used.

⟨ Declare Pattern functions 1320 ⟩ +≡
unsigned int *epicycloid_pattern_1* (**real** *diameter_inner*, **real** *diameter_outer_start*, **real** *diameter_outer_end*, **real** *step*, **unsigned int** *offsets*, **vector**⟨**const Color** ∗⟩ *colors* = **Colors** :: *default_color_vector*, **int** *arc_divisions* = 72);

1324.

```
⟨ Define Pattern functions 1321 ⟩ +≡
  unsigned int epicycloid_pattern_1(real diameter_inner, real diameter_outer_start, real
        diameter_outer_end, real step, unsigned int offsets, vector⟨const Color *⟩ colors, int
        arc_divisions)
  {
    bool DEBUG = false;     /* true */
    if (diameter_inner < diameter_outer_start) {
      cerr ≪ "WARNING!␣diameter_inner␣<␣diameter_outer!\n" ≪
          "This␣is␣likely␣to␣lead␣to␣strange␣results.\n" ≪ "Continuing.\n\n" ≪ flush;
    }
    using namespace Colors;
    unsigned int spiral_counter = 0;
    real radius_outer;
    real phi;
    Path spiral;
    spiral += "..";
    real radius_inner = diameter_inner/2;
    Circle inner_circle(origin, diameter_inner);
    inner_circle.draw();
    if (DEBUG) inner_circle.get_center().dotlabel("inner_circle", "rt");
    Circle outer_circle;
    Point outer_circle_center;
    Point normal;
    Point p0;
    real theta = 360.0/arc_divisions;
    Circle temp_circle;
    Point p2;
    Point temp_circle_center;
    Point temp_circle_normal;
    if (offsets < 1) {
      cerr ≪ "WARNING!␣offsets␣has␣invalid␣value:␣" ≪ offsets ≪ endl ≪
          "offsets␣must␣be␣>=␣1.␣␣Setting␣to␣1.\n\n" ≪ flush;
      offsets = 1;
    }
    real diameter_outer;
    vector⟨const Color *⟩::iterator iter = colors.begin();
    for (diameter_outer = diameter_outer_start; diameter_outer ≥ diameter_outer_end;
          diameter_outer −= step) {
      if (iter ≠ colors.end() − 1) ++iter;
      radius_outer = diameter_outer/2;
      outer_circle_center.set(0, 0, radius_inner + radius_outer);
      normal = outer_circle_center;
      normal.shift(0, 1);
      p0.set(0, 0, radius_inner + diameter_outer);
      outer_circle.set(outer_circle_center, diameter_outer);
      if (colors.size() ≡ 0) colors.push_back(default_color);
      for (unsigned int i = 0; i < offsets; ++i) {
        if (i ≠ 0) outer_circle_center *= normal *= p0 *= outer_circle.rotate(0, 360.0/offsets);
        else {
```

```
        if (DEBUG) {
            p0.dotlabel("p0");
            outer_circle.draw(blue);
        }
    }
    if (DEBUG) {
        p0.dotlabel("p0", "bot");
        outer_circle.draw(green, "evenly");
    }
    spiral += p0;
    phi = theta * radius_inner/radius_outer;
    temp_circle = outer_circle;
    temp_circle_center = outer_circle_center;
    temp_circle_normal = normal;
    p2 = p0;
    for (int j = 1; j <= arc_divisions; ++j) {
        p2 *= temp_circle_normal *= temp_circle_center *= temp_circle.rotate(0, theta);
        p2.rotate(temp_circle_center, temp_circle_normal, phi);
        spiral += p2;
        if (DEBUG) {
            p2.dotlabel("p2", "lft");
            temp_circle.draw(black, "evenly");
        }
    }
    spiral.draw(**iter);
    ++spiral_counter;
    spiral.clear();
    spiral += "..";
   }
  }
  return spiral_counter;
}
```

1325. Epicycloid pattern 2. [LDF 2003.02.11.] This pattern should be like *epicycloid_pattern_1*( ), except that the offsets are not made by rotating the outer **Circle** around the center of the inner **Circle**, but by rotating the **Point** used for tracing the epicycloid about the center of the outer **Circle**.

1326. Epicycloid pattern 3. [LDF 2003.02.11.] This function works well for outer **Circles** with radii that are not even divisors of the radius of the inner **Circle**.

Log

[LDF 2003.02.11.] Added this function.

[LDF 2003.08.27.] Removed the declaration **real** *radius_ratio* = *radius_outer*/*radius_inner*, since *radius_ratio* was never used.

⟨ Declare Pattern functions 1320 ⟩ +≡

```
unsigned int epicycloid_pattern_3(real diameter_inner, real diameter_outer, vector<const Color *>
    colors = Colors::default_color_vector, unsigned int limit = 100, int arc_divisions = 72);
```

1327.

⟨ Define Pattern functions 1321 ⟩ +≡

```
unsigned int epicycloid_pattern_3(real diameter_inner, real diameter_outer, vector⟨const Color *⟩
        colors, unsigned int limit, int arc_divisions)
{
  using namespace Colors;
  bool DEBUG = false;    /* true */
  vector⟨const Color *⟩::iterator color_iter = colors.begin( );
  real radius_outer = diameter_outer/2;
  real radius_inner = diameter_inner/2;
  real theta = 360.0/arc_divisions;
  real phi = theta * radius_inner/radius_outer;
  real theta_total = 0;
  Circle inner_circle(origin, diameter_inner);
  inner_circle.draw( );
  Point outer_circle_center(0, 0, radius_inner + radius_outer);
  Circle outer_circle(outer_circle_center, diameter_outer);
  outer_circle.draw( );
  Point normal(outer_circle_center);
  normal.shift(0, 1);
  Point p0(0, 0, radius_inner + diameter_outer);
  p0.dotlabel("p0");
  Path spiral;
  spiral += "..";
  spiral += p0;
  Point start_pt(p0);
  unsigned int spiral_counter = 1;
  unsigned int iter_ctr = 0;
  while (true) {
    if (theta_total ≥ 360) {
      cout ≪ "theta_total␣==␣" ≪ theta_total ≪ endl ≪ "Reducing␣theta_total␣to␣" ≪
          (theta_total − 360) ≪ endl ≪ flush;
      theta_total −= 360;
      ++iter_ctr;
      spiral.draw(**color_iter++);
      spiral.clear( );
      spiral += "..";
      if (color_iter ≡ colors.end( )) color_iter = colors.begin( );
    }
    if (iter_ctr > limit) {
      cout ≪ "Exceeded␣limit.␣␣iter_ctr␣==␣" ≪ iter_ctr ≪ endl ≪ "Breaking.\n\n" ≪ flush;
      break;
    }
    else if (iter_ctr > 0 ∧ (fmod((iter_ctr * radius_outer), radius_inner) ≡ 0)) {
      cout ≪ "Came␣out␣even.\n" ≪ "iter_ctr␣==␣" ≪ iter_ctr ≪ endl ≪
          "fmod((iter_ctr␣*␣radius_outer),␣radius_inner)␣==␣" ≪ fmod((iter_ctr * radius_outer),
          radius_inner) ≪ endl ≪ "Breaking.\n\n" ≪ flush;
      break;
    }
```

```
        outer_circle_center *= normal *= p0 *= outer_circle.rotate(0, theta);
        p0.rotate(outer_circle_center, normal, phi);
        theta_total += theta;
        if (DEBUG) {
            outer_circle.draw(black, "evenly");
            p0.dotlabel("p0");
        }
        spiral += p0;
        ++spiral_counter;
    }
    return spiral_counter;
}
```

1328. Putting patterns together.

1329. This is what's compiled.

⟨ Include files 6 ⟩
⟨ Version control identifier 5 ⟩
⟨ Declare Pattern functions 1320 ⟩
⟨ Define Pattern functions 1321 ⟩

1330. This is what's written to `patterns.h`.

⟨ `patterns.h` 1330 ⟩ ≡
⟨ Declare Pattern functions 1320 ⟩

1331. Solid (`solids.web`). [LDF 2002.11.12.] TO DO: Add *get_center*( ). Must set *center* in the **Polyhedra** before this is useful.

Log

[LDF 2002.09.29.] Created this file.

[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* = `"$Id:␣solids.web,v␣1.5␣2004/01/12␣21:33:23␣lfinsto1␣Exp␣$"`;

1332. 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"
#include "points.h"
#include "lines.h"
#include "planes.h"
#include "paths.h"
#include "curves.h"
#include "polygons.h"
#include "rectangs.h"
#include "ellipses.h"
#include "circles.h"
```

1333. Solid class definition.

Log

[LDF 2002.09.30.] Added the data members *circles*, *ellipses*, *polygons*, and *paths*. On the one hand, this is wasteful, since most if not all **Solids** will contain only one kind of **Path**; on the other hand, it's an advantage to be able to have the drawing and filling functions be members of **Solid**, since they don't have to know what kind of a **Path** a **Path** is, in order to draw or fill it. This way, I don't have to define the drawing and filling functions for *Sphere*, *Ellipsoid*, **Polyhedron**, etc.

[LDF 2002.10.01.] Added the data member *projective_extremes*.

[LDF 2003.04.11.] Added the **static const** data members `PATH`, `CIRCLE`, `ELLIPSE`, `REG_POLYGON`, and `RECTANGLE`. Currently, their only use is as arguments to *get_shape_ptr*( ) and *get_shape_center*( ).

[LDF 2003.04.11.] Renamed *polygons* to *reg_polygons*. This is in case I decide to make it possible to have irregular polygons. In this case, I may define a **class Polygon** and derived **Reg_Polygon** from it.

⟨ Define **class Solid** 1333 ⟩ ≡
```
class Solid : public Shape {
protected: bool on_free_store;
  Point center;
  bool do_output;    /* LDF 2002.10.01. Added. */
  vector<Circle *> circles;
  vector<Ellipse *> ellipses;
  vector<Path *> paths;
  vector<Rectangle *> rectangles;
  vector<Reg_Polygon *> reg_polygons;
  valarray<real> projective_extremes;
public: static const unsigned short CIRCLE;
  static const unsigned short ELLIPSE;
  static const unsigned short PATH;
  static const unsigned short RECTANGLE;
  static const unsigned short REG_POLYGON;
  ⟨ Declare Solid functions 1336 ⟩
};
```
This code is used in sections 1441 and 1442.

1334. Define static const Solid data members.

Log

[LDF 2003.04.11.] Added this section.

⟨ Define **static const Solid** data members 1334 ⟩ ≡
```
const unsigned short Solid::CIRCLE = 0;
const unsigned short Solid::ELLIPSE = 1;
const unsigned short Solid::PATH = 2;
const unsigned short Solid::RECTANGLE = 3;
const unsigned short Solid::REG_POLYGON = 4;
```
This code is used in section 1441.

1335. Constructors.

1336. Default constructor. (No arguments.) [LDF 2002.10.02.] **Solid** will not normally be used in user code, since it is intended to be a base class only. Therefore, objects of type **Solid** will not normally be declared as automatic variables and there will be no **static** global **Solids**. However, *create_new* < **Solid** > ( ) is used in the drawing and filling functions, in order to put **Solids** onto **Pictures**. If we didn't have a constructor, *projective_extremes* wouldn't initially have the right size and *on_free_store* and *do_output* would both be *false* (assuming the compiler set s the initial values of uninitialized **bools** to *false*). None of this would really matter, because presumably an assignment would follow immediately, which would take care of everything, but there's no harm in making sure.

⟨ Declare **Solid** functions 1336 ⟩ ≡
```
Solid();
```
See also sections 1338, 1343, 1345, 1348, 1350, 1353, 1356, 1358, 1360, 1362, 1364, 1366, 1369, 1371, 1373, 1375, 1377, 1379, 1381, 1383, 1386, 1388, 1390, 1392, 1395, 1397, 1399, 1401, 1404, 1406, 1408, 1409, 1411, 1413, 1415, 1417, 1419, 1423, 1426, 1429, 1432, 1435, and 1438.

This code is used in section 1333.

1337.

⟨ Define **Solid** functions 1337 ⟩ ≡
  Solid :: **Solid**( )
  {
    on_free_store = *false*;
    do_output = *true*;
    projective_extremes.*resize*(6, 0);
  }

See also sections 1339, 1344, 1346, 1347, 1349, 1351, 1354, 1357, 1359, 1361, 1363, 1365, 1367, 1370, 1372, 1374, 1376, 1378, 1380, 1382, 1384, 1387, 1389, 1391, 1393, 1396, 1398, 1400, 1402, 1405, 1407, 1410, 1412, 1414, 1416, 1418, 1420, 1424, 1427, 1430, 1433, 1436, and 1439.

This code is used in section 1441.

1338. Copy constructor. [LDF 2002.10.02.]

⟨ Declare **Solid** functions 1336 ⟩ +≡
  Solid(**const Solid** &*s*);

1339.

⟨ Define **Solid** functions 1337 ⟩ +≡
Solid :: **Solid**(**const Solid** &*s*){ *on_free_store* = *false*;
do_output = *true*;
projective_extremes.*resize*(6, 0); **for** (**vector**⟨**Path** ∗⟩ :: *const_iterator iter* = *s*.*paths*.*begin*( );
iter ≠ *s*.*paths*.*end*( ); ++*iter*) { *paths*.*push_back* (*create_new* < **Path** > (0)) ;
∗(*paths*.*back*( )) = ∗∗*iter*; } **for** (**vector**⟨**Circle** ∗⟩ :: *const_iterator iter* = *s*.*circles*.*begin*( );
iter ≠ *s*.*circles*.*end*( ); ++*iter*) { *circles*.*push_back* (*create_new* < **Circle** > (0)) ;
∗(*circles*.*back*( )) = ∗∗*iter*; } **for** (**vector**⟨**Ellipse** ∗⟩ :: *const_iterator iter* = *s*.*ellipses*.*begin*( );
iter ≠ *s*.*ellipses*.*end*( ); ++*iter*) { *ellipses*.*push_back* (*create_new* < **Ellipse** > (0)) ;
∗(*ellipses*.*back*( )) = ∗∗*iter*; } **for** (**vector**⟨**Reg_Polygon**
∗⟩ :: *const_iterator iter* = *s*.*reg_polygons*.*begin*( ); *iter* ≠ *s*.*reg_polygons*.*end*( );
++*iter*) { *reg_polygons*.*push_back* (*create_new* < **Reg_Polygon** > (0)) ;
∗(*reg_polygons*.*back*( )) = ∗∗*iter*; } **for** (**vector**⟨**Rectangle**
∗⟩ :: *const_iterator iter* = *s*.*rectangles*.*begin*( ); *iter* ≠ *s*.*rectangles*.*end*( ); ++*iter*) {
rectangles.*push_back* (*create_new* < **Rectangle** > (0)) ;
∗(*rectangles*.*back*( )) = ∗∗*iter*; } }

1340. Pseudo-constructor for dynamic allocation.

1341. Pointer argument.

Log

[LDF 2003.12.30.] Replaced **Solid** :: *create_new_solid*( ) with a specialization of **template**⟨**class** *C*⟩ *C*∗*create_new*( ) for **Solid**. The argument is now **const**.

⟨ Declare non-member template functions for **Solid** 1341 ⟩ ≡
Solid ∗*create_new*(**const Solid** ∗*c*);

See also section 1342.

This code is used in sections 1441 and 1442.

1342. Reference argument.

Log

LDF 2003.12.30. Added this function.

⟨ Declare non-member template functions for **Solid** 1341 ⟩ +≡
Solid ∗*create_new*(**const Solid** &*c*);

1343. 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 Solid' has virtual functions but non-virtual destructor".

⟨ Declare **Solid** functions 1336 ⟩ +≡
virtual ∼**Solid**( );

1344.

⟨ Define **Solid** functions 1337 ⟩ +≡

```
Solid::~Solid()
{ }
```

1345. Assignment.

⟨ Declare **Solid** functions 1336 ⟩ +≡

```
virtual const Solid &operator=(const Solid &s);
```

1346.

⟨ Define **Solid** functions 1337 ⟩ +≡

```
const Solid &Solid::operator=(const Solid &s){ bool DEBUG = false;     /* true */
    if (DEBUG) cout << "Entering Solid::operator=().\n";
    if (this ≡ &s)     /* Make sure it's not self-assignment. */
      return *this;
    center = s.center;     /* LDF 2002.10.06. Added this line, because center is now a member of
        Solid. */     /* [LDF 2002.10.02.] First, call the destructor on all of the elements of paths,
        circles, ellipses, reg_polygons, and rectangles, because they've been allocated dynamically. Then
        clear out the vectors. */
    if (paths.size() > 0) {
      for (vector<Path *>::iterator iter = paths.begin(); iter ≠ paths.end(); ++iter) (**iter).clear();
      paths.clear();
    }
    if (circles.size() > 0) {
      for (vector<Circle *>::iterator iter = circles.begin(); iter ≠ circles.end(); ++iter)
        (**iter).clear();
      circles.clear();
    }
    if (ellipses.size() > 0) {
      for (vector<Ellipse *>::iterator iter = ellipses.begin(); iter ≠ ellipses.end(); ++iter)
        (**iter).clear();
      ellipses.clear();
    }
    if (reg_polygons.size() > 0) {
      for (vector<Reg_Polygon *>::iterator iter = reg_polygons.begin(); iter ≠ reg_polygons.end();
          ++iter) (**iter).clear();
      reg_polygons.clear();
    }
    if (rectangles.size() > 0) {
      for (vector<Rectangle *>::iterator iter = rectangles.begin(); iter ≠ rectangles.end(); ++iter)
        (**iter).clear();
      rectangles.clear();
    }
```

1347. Now, create new **Path**, **Circle**, **Ellipse**, **Reg_Polygon**, and **Rectangle** pointers, allocate memory for them, assign values to the objects they point to from *s*, and push them onto the appropriate vectors. [LDF 2002.10.02.]

⟨ Define **Solid** functions 1337 ⟩ +≡
```
Path *p; for (vector⟨Path *⟩::const_iterator iter = s.paths.begin(); iter ≠ s.paths.end(); ++iter) {
    p = create_new < Path > (0);
  paths.push_back(p);
  *(paths.back()) = **iter; } for (vector⟨Circle *⟩::const_iterator iter = s.circles.begin();
    iter ≠ s.circles.end(); ++iter) { circles.push_back ( create_new < Circle > (0) ) ;
  *(circles.back()) = **iter; } for (vector⟨Ellipse *⟩::const_iterator iter = s.ellipses.begin();
    iter ≠ s.ellipses.end(); ++iter) { ellipses.push_back ( create_new < Ellipse > (0) ) ;
  *(ellipses.back()) = **iter; } for (vector⟨Reg_Polygon *⟩::const_iterator iter = s.reg_polygons.begin();
    iter ≠ s.reg_polygons.end(); ++iter) { reg_polygons.push_back ( create_new < Reg_Polygon > (0)
    ) ;
  *(reg_polygons.back()) = **iter; } for (vector⟨Rectangle *⟩::const_iterator iter = s.rectangles.begin();
    iter ≠ s.rectangles.end(); ++iter) { rectangles.push_back ( create_new < Rectangle > (0) ) ;
  *(rectangles.back()) = **iter; } projective_extremes = 0;    /* For output. */
  do_output = true;
  if (DEBUG) {
    cout ≪ "paths.size()␣==␣" ≪ paths.size() ≪ endl ≪ flush;
    cout ≪ "circles.size()␣==␣" ≪ circles.size() ≪ endl ≪ flush;
    cout ≪ "ellipses.size()␣==␣" ≪ ellipses.size() ≪ endl ≪ flush;
    cout ≪ "reg_polygons.size()␣==␣" ≪ reg_polygons.size() ≪ endl ≪ flush;
    cout ≪ "rectangles.size()␣==␣" ≪ rectangles.size() ≪ endl ≪ flush;
  }
  if (DEBUG) cout ≪ "Exiting␣Solid::operator=().\n";
  return *this; }
```

1348. Copying.

Log

[LDF 2003.05.06.] BUG FIX: Changed *s* from **Shape** * to **Solid** *. I noticed this bug when I tried to copy a **Picture** containing a **Cuboid**, and the copy contained a single empty **Shape** * on *shapes*.

⟨ Declare **Solid** functions 1336 ⟩ +≡
```
virtual Shape *get_copy() const;
```

1349.

⟨ Define **Solid** functions 1337 ⟩ +≡
```
Shape *Solid::get_copy() const{ Solid *s = create_new < Solid > (0);
    *s = *this;
    return dynamic_cast⟨Shape *⟩(s); }
```

1350. Set on free store.

Log

[LDF 2004.01.06.] Made non-inline.

⟨ Declare **Solid** functions 1336 ⟩ +≡
```
virtual bool set_on_free_store(bool b = true);
```

1351.

⟨ Define **Solid** functions 1337 ⟩ +≡
```
bool Solid::set_on_free_store(bool b)
{
  on_free_store = b;
  return b;
}
```

1352. Returning elements and information. [LDF 2003.04.11.] The functions *get_shape_ptr*( ), *get_circle_ptr*( ), *get_ellipse_ptr*( ), *get_path_ptr*( ), *get_rectangle_ptr*( ), and *get_reg_polygon_ptr*( ) all return **const** pointers to **Shape**, **Circle**, **Ellipse**, etc. Therefore, they must be invoked in such a way, that the **const** qualifier is not discarded. For example, following **Dodecahedron** *d*(*origin*, 5);, two ways of invoking *get_reg_polygon_ptr*( ) are: **const Reg_Polygon** **ptr* = *d.get_reg_polygon_ptr*(5); and **Reg_Polygon** *A* = **d.get_reg_polygon_ptr*(5);.

Log

[LDF 2003.05.09.] Changed the names of *get_shape*( ), *get_circle*( ), *get_ellipse*( ), *get_path*( ), *get_rectangle*( ), and *get_reg_polygon*( ) to *get_shape_ptr*( ), *get_circle_ptr*( ), *get_ellipse_ptr*( ), *get_path_ptr*( ), *get_rectangle_ptr*( ), and *get_reg_polygon_ptr*( ). The names without "*_ptr*" were confusing, because they didn't make clear that the functions returned pointers.

1353. Get center.

Log

[LDF 2003.05.06.] Added this function.
[LDF 2003.08.10.] Made this function **const**.

⟨ Declare **Solid** functions 1336 ⟩ +≡
```
virtual const Point &get_center() const;
```

1354.

⟨ Define **Solid** functions 1337 ⟩ +≡
```
const Point &Solid::get_center() const
{
  return center;
}
```

1355. Getting Shapes.

Log

[LDF 2003.04.30.] Changed the functions *get_circle_ptr*( ), *get_ellipse_ptr*( ), *get_path_ptr*( ), *get_rectangle_ptr*( ), and *get_reg_polygon_ptr*( ). They no longer use *get_shape_ptr*( ). There's no good reason for casting pointers from one type to another. I rather doubt that *get_shape_ptr*( ) is needed, anyway.

1356. Get Shape pointer. [LDF 2003.05.30.] This function copies one of the objects on one of the vectors of **Shape** $*$ belonging to the **Solid**, and returns a pointer to **Shape** that points to the copy. Currently, a **Solid** contains the vectors *circles*, *ellipses*, *paths*, *rectangles*, and *reg_polygons*. The argument *shape_type* indicates which vector should be accessed. Normally, the corresponding **public static const** data members `CIRCLE`, `ELLIPSE`, `PATH`, `RECTANGLE`, or `REG_POLYGON` should be passed as the *shape_type* argument, e.g., **Circle** $*$*c_ptr* = **static_cast**⟨**Circle** $*$⟩(*get_shape_ptr*(**Solid** :: `CIRCLE`, 3)).

[LDF 2003.04.30.] This function was mainly intended for use in the functions *get_circle_ptr*( ), *get_ellipse_ptr*( ), etc., and was not intended for use in user code. I now doubt whether this function is needed at all, especially since it is no longer used in the functions mentioned above.

Log

[LDF 2003.04.11.] Added this function.

[LDF 2003.04.30.] Now using *get_copy*( ) instead of **static_cast**⟨**const Shape** $*$⟩( ). The way it was caused compilation errors under Tru64 (DEC ALPHA).

[LDF 2003.05.30.] Changed return value to **Shape** $*$ from **const Shape** $*$. The way it was before caused "Memory fault" errors at run-time.

⟨ Declare **Solid** functions 1336 ⟩ +≡

virtual Shape $*$*get_shape_ptr*(**const unsigned short** *shape_type*, **const unsigned short** *s*) **const**;

1357.

⟨ Define **Solid** functions 1337 ⟩ +≡

```
Shape *Solid::get_shape_ptr(const unsigned short shape_type, const unsigned short s) const
{
  bool DEBUG = false;   /* true */
  if (DEBUG) {
    cout << "Entering Solid::get_shape_ptr().\n" << flush;
  }
  if (shape_type ≡ CIRCLE) {
    if (s < circles.size()) {
      return circles[s]→get_copy();
    }
    else {
      cerr << "ERROR! In Solid::get_shape_ptr():\n" << "s (" << s << ") > circles.size() (" <<
          circles.size() << ")\nReturning a null pointer (0).\n\n" << flush;
      return static_cast⟨Shape *⟩(0);
    }
  }
  else if (shape_type ≡ ELLIPSE) {
    if (s < ellipses.size()) {
      return ellipses[s]→get_copy();
    }
    else {
      cerr << "ERROR! In Solid::get_shape_ptr():\n" << "s (" <<
          s << ") > ellipses.size() (" << ellipses.size() <<
          ")\nReturning a null pointer (0).\n\n" << flush;
      return static_cast⟨Shape *⟩(0);
    }
  }
  else if (shape_type ≡ PATH) {
    if (s < paths.size()) {
      return paths[s]→get_copy();
    }
    else {
      cerr << "ERROR! In Solid::get_shape_ptr():\n" << "s (" << s << ") > paths.size() (" <<
          paths.size() << ")\nReturning a null pointer (0).\n\n" << flush;
      return static_cast⟨Shape *⟩(0);
    }
  }
  else if (shape_type ≡ RECTANGLE) {
    if (s < rectangles.size()) {
      return rectangles[s]→get_copy();
    }
    else {
      cerr << "ERROR! In Solid::get_shape_ptr():\n" << "s (" <<
          s << ") > rectangles.size() (" << rectangles.size() <<
          ")\nReturning a null pointer (0).\n\n" << flush;
      return static_cast⟨Shape *⟩(0);
    }
  }
  else if (shape_type ≡ REG_POLYGON) {
    if (s < reg_polygons.size()) {
```

```
      return reg_polygons[s]→get_copy();
    }
    else {
      cerr ≪ "ERROR!␣In␣Solid::get_shape_ptr():\n" ≪ "s␣(" ≪
          s ≪ ")␣>␣reg_polygons.size()␣(" ≪ reg_polygons.size() ≪
          ")\nReturning␣a␣null␣pointer␣(0).\n\n" ≪ flush;
      return static_cast⟨Shape *⟩(0);
    }
  }
  else {
    cerr ≪ "ERROR!␣In␣Solid::get_shape_ptr():\n" ≪ "Invalid␣value␣for␣shape_type:␣" ≪
        shape_type ≪ endl ≪ "Returning␣a␣null␣pointer␣(0).\n\n" ≪ flush;
    return static_cast⟨Shape *⟩(0);
  }
}
```

1358. Get Circle pointer.

Log

[LDF 2003.04.11.] Added this function.
[LDF 2003.04.30.] Changed this function, so that it no longer uses *get_shape_ptr*( ).

⟨ Declare **Solid** functions 1336 ⟩ +≡

```
virtual const Circle *get_circle_ptr(const unsigned short s) const;
```

1359.

⟨ Define **Solid** functions 1337 ⟩ +≡

```
const Circle *Solid::get_circle_ptr(const unsigned short s) const
{
  if (circles.size() > s) {
    return circles[s];
  }
  else {
    return static_cast⟨const Circle *⟩(0);
  }
}
```

1360. Get Ellipse pointer.

Log

[LDF 2003.04.11.] Added this function.
[LDF 2003.04.30.] Changed this function, so that it no longer uses *get_shape_ptr*( ).

⟨ Declare **Solid** functions 1336 ⟩ +≡

```
virtual const Ellipse *get_ellipse_ptr(const unsigned short s) const;
```

1361.

⟨ Define **Solid** functions 1337 ⟩ +≡
```
const Ellipse *Solid::get_ellipse_ptr(const unsigned short s) const
{
  if (ellipses.size() > s) {
    return ellipses[s];
  }
  else {
    return static_cast<const Ellipse *>(0);
  }
}
```

1362. Get Path pointer.

Log

[LDF 2003.04.11.] Added this function.
[LDF 2003.04.30.] Changed this function, so that it no longer uses *get_shape_ptr*( ).

⟨ Declare **Solid** functions 1336 ⟩ +≡
```
virtual const Path *get_path_ptr(const unsigned short s) const;
```

1363.

⟨ Define **Solid** functions 1337 ⟩ +≡
```
const Path *Solid::get_path_ptr(const unsigned short s) const
{
  if (paths.size() > s) {
    return paths[s];
  }
  else {
    return static_cast<const Path *>(0);
  }
}
```

1364. Get Rectangle pointer.

Log

[LDF 2003.04.11.] Added this function.
[LDF 2003.04.30.] Changed this function, so that it no longer uses *get_shape_ptr*( ).

⟨ Declare **Solid** functions 1336 ⟩ +≡
```
virtual const Rectangle *get_rectangle_ptr(const unsigned short s) const;
```

1365.

⟨ Define **Solid** functions 1337 ⟩ +≡

```
const Rectangle *Solid::get_rectangle_ptr(const unsigned short s) const
{
  if (rectangles.size() > s) {
    return rectangles[s];
  }
  else {
    return static_cast<const Rectangle *>(0);
  }
}
```

1366. Get Reg_Polygon pointer.

Log

[LDF 2003.04.11.] Added this function.
[LDF 2003.04.30.] Changed this function, so that it no longer uses *get_shape_ptr*( ).

⟨ Declare **Solid** functions 1336 ⟩ +≡

```
virtual const Reg_Polygon *get_reg_polygon_ptr(const unsigned short s) const;
```

1367.

⟨ Define **Solid** functions 1337 ⟩ +≡

```
const Reg_Polygon *Solid::get_reg_polygon_ptr(const unsigned short s) const
{
  if (reg_polygons.size() > s) {
    return reg_polygons[s];
  }
  else {
    return static_cast<const Reg_Polygon *>(0);
  }
}
```

1368. Getting Shape centers. [LDF 2003.04.30.] TO DO: I think it might be possible to code the functions in this section more succinctly.

Log

[LDF 2003.04.11.] Added this section.

1369. Get Shape center. This function returns the center of the **Circle**, **Ellipse**, **Rectangle**, or **Reg_Polygon** number *s* in *circles*, *ellipses*, *rectangles*, or *reg_polygons*, respectively. If *s* is larger than ⟨*vector*⟩. *size*( ), an error message is issued and `INVALID_POINT` is returned.

One of the following **public static const** data members of **Solid** can (and probably should) be used as the *shape_type* argument: `CIRCLE`, `ELLIPSE`, `RECTANGLE`, and `REG_POLYGON`.

!! Note that this function will have to be changed, if new vectors of **Shape** pointers are added to **class Solid**!

Log

[LDF 2002.10.16.] Added this function.

[LDF 2003.04.09.] Moved this function from **Polyhedron** to **Solid**.

[LDF 2003.04.11.] Changed this function from *get_polygon_center*( ) to *get_shape_center*( ). Added the **char** argument *shape_type* to indicate whether it should return the center of a **Circle**, **Ellipse**, **Rectangle**, or **Reg_Polygon**.

[LDF 2003.04.11.] Changed the *shape_type* argument from **char** to **const unsigned short**.

⟨ Declare **Solid** functions 1336 ⟩ +≡
virtual const Point &*get_shape_center*(**const unsigned short** *shape_type*, **const unsigned short** *s*) **const**;

1370.

```
⟨ Define Solid functions 1337 ⟩ +≡
  const Point &Solid::get_shape_center(const unsigned short shape_type, const unsigned short s)
        const
  {
    if (shape_type ≡ CIRCLE) {
      if (s < circles.size()) return circles[s]→get_center();
      else {
        cerr ≪ "ERROR!␣In␣Solid::get_shape_center():\n" ≪ "s␣(" ≪ s ≪
            ")␣>␣circles.size()␣(" ≪ circles.size() ≪ ")\nReturning␣INVALID_POINT.\n\n" ≪
            flush;
        return INVALID_POINT;
      }
    }
    else if (shape_type ≡ ELLIPSE) {
      if (s < ellipses.size()) return ellipses[s]→get_center();
      else {
        cerr ≪ "ERROR!␣In␣Solid::get_shape_center():\n" ≪ "s␣(" ≪ s ≪
            ")␣>␣ellipses.size()␣(" ≪ ellipses.size() ≪ ")\nReturning␣INVALID_POINT.\n\n" ≪
            flush;
        return INVALID_POINT;
      }
    }
    else if (shape_type ≡ RECTANGLE) {
      if (s < rectangles.size()) return rectangles[s]→get_center();
      else {
        cerr ≪ "ERROR!␣In␣Solid::get_shape_center():\n" ≪ "s␣(" ≪
            s ≪ ")␣>␣rectangles.size()␣(" ≪ rectangles.size() ≪
            ")\nReturning␣INVALID_POINT.\n\n" ≪ flush;
        return INVALID_POINT;
      }
    }
    else if (shape_type ≡ REG_POLYGON) {
      if (s < reg_polygons.size()) return reg_polygons[s]→get_center();
      else {
        cerr ≪ "ERROR!␣In␣Solid::get_shape_center():\n" ≪ "s␣(" ≪
            s ≪ ")␣>␣reg_polygons.size()␣(" ≪ reg_polygons.size() ≪
            ")\nReturning␣INVALID_POINT.\n\n" ≪ flush;
        return INVALID_POINT;
      }
    }
    else {
      cerr ≪ "ERROR!␣In␣Solid::get_shape_center():\n" ≪
          "Invalid␣argument␣for␣shape_type:␣" ≪ shape_type ≪ endl ≪
          "Returning␣INVALID_POINT.\n\n" ≪ flush;
      return INVALID_POINT;
    }
  }
```

1371. Get Circle center.

Log

[LDF 2003.04.11.] Added this function.

⟨ Declare **Solid** functions 1336 ⟩ +≡
virtual const Point &*get_circle_center*(**const unsigned short** *s*) **const**;

1372.

⟨ Define **Solid** functions 1337 ⟩ +≡
const Point &**Solid**::*get_circle_center*(**const unsigned short** *s*) **const**
{
return *get_shape_center*(CIRCLE, *s*);
}

1373. Get Ellipse center.

Log

[LDF 2003.04.11.] Added this function.

⟨ Declare **Solid** functions 1336 ⟩ +≡
virtual const Point &*get_ellipse_center*(**const unsigned short** *s*) **const**;

1374.

⟨ Define **Solid** functions 1337 ⟩ +≡
const Point &**Solid**::*get_ellipse_center*(**const unsigned short** *s*) **const**
{
return *get_shape_center*(ELLIPSE, *s*);
}

1375. Get Rectangle center.

Log

[LDF 2003.04.11.] Added this function.

⟨ Declare **Solid** functions 1336 ⟩ +≡
virtual const Point &*get_rectangle_center*(**const unsigned short** *s*) **const**;

1376.

⟨ Define **Solid** functions 1337 ⟩ +≡
const Point &**Solid**::*get_rectangle_center*(**const unsigned short** *s*) **const**
{
return *get_shape_center*(RECTANGLE, *s*);
}

1377. Get Reg_Polygon center.

Log

[LDF 2003.04.11.] Added this function.

⟨ Declare **Solid** functions 1336 ⟩ +≡
virtual const Point &*get_reg_polygon_center*(**const unsigned short** *s*) **const**;

1378.

⟨ Define **Solid** functions 1337 ⟩ +≡
const Point &**Solid**::*get_reg_polygon_center*(**const unsigned short** *s*) **const**
{
return *get_shape_center*(`REG_POLYGON`, *s*);
}

1379. Is on free store.

⟨ Declare **Solid** functions 1336 ⟩ +≡
virtual bool *is_on_free_store*( ) **const**;

1380.

⟨ Define **Solid** functions 1337 ⟩ +≡
bool Solid::*is_on_free_store*( ) **const**
{
bool $b = true$;
return *b*;
}

1381. Show.

⟨ Declare **Solid** functions 1336 ⟩ +≡
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* = **Projections**::`PERSP`, **const real** *factor* = 1) **const**;

1382.

⟨ Define **Solid** functions 1337 ⟩ +≡

```
void Solid::show(string text, char coords, const bool do_persp, const bool do_apply, Focus
        *f, const unsigned short proj, const real factor) const
{
  if (text ≡ "") text = "Solid:";
  cout ≪ text ≪ endl;
  cout ≪ "on_free_store␣==␣" ≪ on_free_store ≪ endl ≪ flush;
  stringstream g;
  int i;
  if (paths.size() > 0) {
    cout ≪ "Showing␣paths.\n";
    i = 0;
    for (vector⟨Path *⟩::const_iterator iter = paths.begin(); iter ≠ paths.end(); ++iter) {
      g ≪ "Path␣" ≪ i++ ≪ ":";
      (**iter).show(g.str(), coords, do_persp, do_apply, f, proj, factor);
      g.str("");
    }
  }
  else cout ≪ "paths␣is␣empty.\n";
  if (circles.size() > 0) {
    cout ≪ "Showing␣circles.\n";
    i = 0;
    for (vector⟨Circle *⟩::const_iterator iter = circles.begin(); iter ≠ circles.end(); ++iter) {
      g ≪ "Circle␣" ≪ i++ ≪ ":";
      (**iter).show(g.str(), coords, do_persp, do_apply, f, proj, factor);
      g.str("");
    }
  }
  else cout ≪ "circles␣is␣empty.\n";
  if (ellipses.size() > 0) {
    cout ≪ "Showing␣ellipses.\n";
    i = 0;
    for (vector⟨Ellipse *⟩::const_iterator iter = ellipses.begin(); iter ≠ ellipses.end(); ++iter) {
      g ≪ "Ellipse␣" ≪ i++ ≪ ":";
      (**iter).show(g.str(), coords, do_persp, do_apply, f, proj, factor);
      g.str("");
    }
  }
  else cout ≪ "ellipses␣is␣empty.\n";
  if (reg_polygons.size() > 0) {
    cout ≪ "Showing␣reg_polygons.\n";
    i = 0;
    for (vector⟨Reg_Polygon *⟩::const_iterator iter = reg_polygons.begin(); iter ≠ reg_polygons.end();
          ++iter) {
      g ≪ "Polygon␣" ≪ i++ ≪ ":";
      (**iter).show(g.str(), coords, do_persp, do_apply, f, proj, factor);
      g.str("");
    }
  }
  else cout ≪ "reg_polygons␣is␣empty.\n";
```

```
    if (rectangles.size() > 0) {
      cout << "Showing rectangles.\n";
      i = 0;
      for (vector<Rectangle *>::const_iterator iter = rectangles.begin(); iter ≠ rectangles.end();
            ++iter) {
        g << "Rectangle " << i++ << ":";
        (**iter).show(g.str(), coords, do_persp, do_apply, f, proj, factor);
        g.str("");
      }
    }
    else cout << "rectangles is empty.\n";
    cout << endl << flush;
    return;
  }
```

1383. Clear. [LDF 2002.10.07.] Replaced dummy definition with a real one. Now, *clear*( ) is called for all of the objects in the **Solid**.

```
⟨ Declare Solid functions 1336 ⟩ +≡
  virtual void clear();
```

1384.

```
⟨ Define Solid functions 1337 ⟩ +≡
  void Solid::clear()
  {
    bool DEBUG = false;    /* true */
    if (DEBUG) cout << "Entering Solid::clear().\n" << flush;
    for (vector<Path *>::iterator iter = paths.begin(); iter ≠ paths.end(); ++iter) (**iter).clear();
    paths.clear();
    for (vector<Circle *>::iterator iter = circles.begin(); iter ≠ circles.end(); ++iter) (**iter).clear();
    circles.clear();
    for (vector<Ellipse *>::iterator iter = ellipses.begin(); iter ≠ ellipses.end(); ++iter)
      (**iter).clear();
    ellipses.clear();
    for (vector<Reg_Polygon *>::iterator iter = reg_polygons.begin(); iter ≠ reg_polygons.end();
          ++iter) (**iter).clear();
    reg_polygons.clear();
    for (vector<Rectangle *>::iterator iter = rectangles.begin(); iter ≠ rectangles.end(); ++iter)
      (**iter).clear();
    rectangles.clear();
    if (DEBUG) cout << "Exiting Solid::clear().\n" << flush;
    return;
  }
```

1385. Transformations.

1386. Multiplying by a Transform.

```
⟨ Declare Solid functions 1336 ⟩ +≡
  virtual Transform operator*=(const Transform &t);
```

1387.

⟨ Define **Solid** functions 1337 ⟩ +≡

```
Transform Solid::operator*=(const Transform &t)
{
  center *= t;
  for (vector<Path *>::iterator iter = paths.begin(); iter != paths.end(); ++iter) **iter *= t;
  for (vector<Ellipse *>::iterator iter = ellipses.begin(); iter != ellipses.end(); ++iter) **iter *= t;
  for (vector<Circle *>::iterator iter = circles.begin(); iter != circles.end(); ++iter) **iter *= t;
  for (vector<Reg_Polygon *>::iterator iter = reg_polygons.begin(); iter != reg_polygons.end();
        ++iter) **iter *= t;
  for (vector<Rectangle *>::iterator iter = rectangles.begin(); iter != rectangles.end(); ++iter)
    **iter *= t;
  return t;
}
```

1388. Applying a transformation.

Log

[LDF 2003.01.05.] Added this function. It's now needed because I've made *apply_transform*( ) a pure **virtual** function in **class Shape**. BUG FIX: I've done this in an attempt to fix a bug in **Picture** :: *output*( ), where the **Points** on a **Path** were not transformed when I used "**Transform** *t*; *current_picture* ∗= *t*".

⟨ Declare **Solid** functions 1336 ⟩ +≡

```
virtual void apply_transform(void);
```

1389.

⟨ Define **Solid** functions 1337 ⟩ +≡

```
void Solid::apply_transform(void)
{
  center.apply_transform();
  for (vector<Path *>::iterator iter = paths.begin(); iter != paths.end(); ++iter)
    (**iter).apply_transform();
  for (vector<Ellipse *>::iterator iter = ellipses.begin(); iter != ellipses.end(); ++iter)
    (**iter).apply_transform();
  for (vector<Circle *>::iterator iter = circles.begin(); iter != circles.end(); ++iter)
    (**iter).apply_transform();
  for (vector<Reg_Polygon *>::iterator iter = reg_polygons.begin(); iter != reg_polygons.end();
        ++iter) (**iter).apply_transform();
  for (vector<Rectangle *>::iterator iter = rectangles.begin(); iter != rectangles.end(); ++iter)
    (**iter).apply_transform();
}
```

1390. Scale.

⟨ Declare **Solid** functions 1336 ⟩ +≡

```
virtual Transform scale(real xx, real yy = 0, real zz = 0);
```

1391.

⟨ Define **Solid** functions 1337 ⟩ +≡

```
Transform Solid::scale(real xx, real yy, real zz)
{
  Transform t;
  t.scale(xx, yy, zz);
  *this *= t;
  return t;
}
```

1392. Shear.

⟨ Declare **Solid** functions 1336 ⟩ +≡

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

1393.

⟨ Define **Solid** functions 1337 ⟩ +≡

```
Transform Solid::shear(real xy, real xz, real yx, real yz, real zx, real zy)
{
  Transform t;
  t.shear(xy, xz, yx, yz, zx, zy);
  *this *= t;
  return t;
}
```

1394. Shift.

1395. real arguments.

⟨ Declare **Solid** functions 1336 ⟩ +≡

```
virtual Transform shift(real xx, real yy = 0, real zz = 0);
```

1396.

⟨ Define **Solid** functions 1337 ⟩ +≡

```
Transform Solid::shift(real xx, real yy, real zz)
{
  Transform t;
  t.shift(xx, yy, zz);
  *this *= t;
  return t;
}
```

1397. Point argument.

⟨ Declare **Solid** functions 1336 ⟩ +≡

```
virtual Transform shift(const Point &pt);
```

1398.

⟨ Define **Solid** functions 1337 ⟩ +≡

```
Transform Solid::shift(const Point &pt)
{
  Transform t;
  t.shift(pt);
  *this *= t;
  return t;
}
```

1399. Rotatation around the main axes.

⟨ Declare **Solid** functions 1336 ⟩ +≡

```
virtual Transform rotate(const real xx, const real yy = 0, const real zz = 0);
```

1400.

⟨ Define **Solid** functions 1337 ⟩ +≡

```
Transform Solid::rotate(const real xx, const real yy, const real zz)
{
  Transform t;
  t.rotate(xx, yy, zz);
  *this *= t;
  return t;
}
```

1401. Rotatation around an arbitrary axis.

Log

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

⟨ Declare **Solid** functions 1336 ⟩ +≡

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

1402.

⟨ Define **Solid** functions 1337 ⟩ +≡

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

1403. Outputting.

1404. Extract.

⟨ Declare **Solid** functions 1336 ⟩ +≡

```
virtual vector<Shape *> extract(const Focus &f, const unsigned short proj, real factor);
```

1405.

⟨ Define **Solid** functions 1337 ⟩ +≡

```
vector⟨Shape ∗⟩ Solid::extract(const Focus &f, const unsigned short proj, real factor)
{
  vector⟨Shape ∗⟩ v;
  for (vector⟨Path ∗⟩::iterator iter = paths.begin(); iter ≠ paths.end(); ++iter) {
    (∗∗iter).apply_transform();
    if (¬(∗∗iter).project(f, proj, factor)) {
      cerr ≪ "WARNING!␣In␣Solid::extract():\n" ≪ "Path␣cannot␣be␣projected.␣" ≪
          "Returning␣empty␣vector␣<Shape*>.\n" ≪ flush;
      return v;
      break;
    }
  }
  for (vector⟨Ellipse ∗⟩::iterator iter = ellipses.begin(); iter ≠ ellipses.end(); ++iter) {
    (∗∗iter).apply_transform();
    if (¬(∗∗iter).project(f, proj, factor)) {
      cerr ≪ "WARNING!␣In␣Solid::extract():\n" ≪ "Ellipse␣cannot␣be␣projected.␣" ≪
          "Returning␣empty␣vector␣<Shape*>.\n" ≪ flush;
      return v;
      break;
    }
  }
  for (vector⟨Circle ∗⟩::iterator iter = circles.begin(); iter ≠ circles.end(); ++iter) {
    (∗∗iter).apply_transform();
    if (¬(∗∗iter).project(f, proj, factor)) {
      cerr ≪ "WARNING!␣In␣Solid::extract():\n" ≪ "Circle␣cannot␣be␣projected.␣" ≪
          "Returning␣empty␣vector␣<Shape*>.\n" ≪ flush;
      return v;
      break;
    }
  }
  for (vector⟨Reg_Polygon ∗⟩::iterator iter = reg_polygons.begin(); iter ≠ reg_polygons.end();
        ++iter) {
    (∗∗iter).apply_transform();
    if (¬(∗∗iter).project(f, proj, factor)) {
      cerr ≪ "WARNING!␣In␣Solid::extract():\n" ≪ "Polygon␣cannot␣be␣projected.␣" ≪
          "Returning␣empty␣vector␣<Shape*>.\n" ≪ flush;
      return v;
      break;
    }
  }
  for (vector⟨Rectangle ∗⟩::iterator iter = rectangles.begin(); iter ≠ rectangles.end(); ++iter) {
    (∗∗iter).apply_transform();
    if (¬(∗∗iter).project(f, proj, factor)) {
      cerr ≪ "WARNING!␣In␣Solid::extract():\n" ≪ "Rectangle␣cannot␣be␣projected.␣" ≪
          "Returning␣empty␣vector␣<Shape*>.\n" ≪ flush;
      return v;
      break;
    }
  }
  v.push_back(this);
```

```
    return v;
  }
```

1406. Set extremes.

⟨ Declare **Solid** functions 1336 ⟩ +≡
  virtual bool *set_extremes*( );

1407.

⟨ Define **Solid** functions 1337 ⟩ +≡

```
bool Solid::set_extremes()
{
  bool DEBUG = false;    /* true */
  if (DEBUG) cout << "Entering Solid::set_extremes()" << "\n" << flush;
  valarray<real> v;
  v.resize(6, 0);    /* LDF 2002.12.13. Added. Needed for compiling under GNU/Linux using GCC on
      the Intel i686 computer gwdu101.gwdg.de. */
  for (vector<Path *>::iterator iter = paths.begin(); iter != paths.end(); ++iter) {
    if (!(**iter).set_extremes()) {
      cerr << "ERROR! In Solid::set_extremes():\n" <<
          "Path::set_extremes() returned false. " << "Returning false.\n\n" << flush;
      return false;
    }
    v = (**iter).get_extremes();
    for (int i = 0; i < 3; ++i)    /* Minima. */
    {
      projective_extremes[i] = min(projective_extremes[i], v[i]);
    }
    for (int i = 3; i < 6; ++i)    /* Maxima. */
    {
      projective_extremes[i] = max(projective_extremes[i], v[i]);
    }
  }
  for (vector<Ellipse *>::iterator iter = ellipses.begin(); iter != ellipses.end(); ++iter) {
    if (!(**iter).set_extremes()) {
      cerr << "ERROR! In Solid::set_extremes():\n" <<
          "Path::set_extremes() returned false. " << "Returning false.\n\n" << flush;
      return false;
    }
    v = (**iter).get_extremes();
    for (int i = 0; i < 3; ++i)    /* Minima. */
    {
      projective_extremes[i] = min(projective_extremes[i], v[i]);
    }
    for (int i = 3; i < 6; ++i)    /* Maxima. */
    {
      projective_extremes[i] = max(projective_extremes[i], v[i]);
    }
  }
  for (vector<Circle *>::iterator iter = circles.begin(); iter != circles.end(); ++iter) {
    if (!(**iter).set_extremes()) {
      cerr << "ERROR! In Solid::set_extremes():\n" <<
          "Path::set_extremes() returned false. " << "Returning false.\n\n" << flush;
      return false;
    }
    v = (**iter).get_extremes();
    for (int i = 0; i < 3; ++i)    /* Minima. */
    {
      projective_extremes[i] = min(projective_extremes[i], v[i]);
```

```
      }
      for (int i = 3; i < 6; ++i)    /* Maxima. */
      {
        projective_extremes[i] = max(projective_extremes[i], v[i]);
      }
    }
    for (vector<Reg_Polygon *>::iterator iter = reg_polygons.begin(); iter ≠ reg_polygons.end();
          ++iter) {
      if (¬(**iter).set_extremes()) {
        cerr ≪ "ERROR!␣In␣Solid::set_extremes():\n" ≪
            "Path::set_extremes()␣returned␣false.␣" ≪ "Returning␣false.\n\n" ≪ flush;
        return false;
      }
      v = (**iter).get_extremes();
      for (int i = 0; i < 3; ++i)    /* Minima. */
      {
        projective_extremes[i] = min(projective_extremes[i], v[i]);
      }
      for (int i = 3; i < 6; ++i)    /* Maxima. */
      {
        projective_extremes[i] = max(projective_extremes[i], v[i]);
      }
    }
    for (vector<Rectangle *>::iterator iter = rectangles.begin(); iter ≠ rectangles.end(); ++iter) {
      if (¬(**iter).set_extremes()) {
        cerr ≪ "ERROR!␣In␣Solid::set_extremes():\n" ≪
            "Path::set_extremes()␣returned␣false.␣" ≪ "Returning␣false.\n\n" ≪ flush;
        return false;
      }
      v = (**iter).get_extremes();
      for (int i = 0; i < 3; ++i)    /* Minima. */
      {
        projective_extremes[i] = min(projective_extremes[i], v[i]);
      }
      for (int i = 3; i < 6; ++i)    /* Maxima. */
      {
        projective_extremes[i] = max(projective_extremes[i], v[i]);
      }
    }
    if (DEBUG) cout ≪ "Exiting␣Solid::set_extremes()" ≪ "\n" ≪ flush;
    return true;
  }
```

1408. Get extremes.

⟨ Declare **Solid** functions 1336 ⟩ +≡

```
  inline virtual const valarray<real> get_extremes() const
  {
    return projective_extremes;
  }
```

1409. Get minimum z.

⟨ Declare **Solid** functions 1336 ⟩ +≡

virtual real *get_minimum_z*( ) **const**;

1410.

⟨ Define **Solid** functions 1337 ⟩ +≡
```
real Solid::get_minimum_z() const
{
  bool DEBUG = false;    /* true */
  if (DEBUG) {
    cout << "Entering Solid::get_minimum_z()" << endl << flush;
    cout << "projective_extremes[4] == " << projective_extremes[4] << endl << flush;
    cout << "Exiting Solid::get_minimum_z()" << endl << flush;
  }
  return projective_extremes[4];
}
```

1411. Get maximum z.

⟨ Declare **Solid** functions 1336 ⟩ +≡
virtual real *get_maximum_z*( ) **const**;

1412.

⟨ Define **Solid** functions 1337 ⟩ +≡
```
real Solid::get_maximum_z() const
{
  bool DEBUG = false;    /* true */
  if (DEBUG) {
    cout << "Entering Solid::get_maximum_z()" << endl << flush;
    cout << "projective_extremes[5] == " << projective_extremes[5] << endl << flush;
    cout << "Exiting Solid::get_maximum_z()" << endl << flush;
  }
  return projective_extremes[5];
}
```

1413. Get mean z. [LDF 2003.05.16.] Added this function.

⟨ Declare **Solid** functions 1336 ⟩ +≡
virtual real *get_mean_z*( ) **const**;

1414.

⟨ Define **Solid** functions 1337 ⟩ +≡
```
real Solid::get_mean_z() const
{
  return ((projective_extremes[4] + projective_extremes[5])/2);
}
```

1415. Suppress output.

⟨ Declare **Solid** functions 1336 ⟩ +≡
virtual void *suppress_output*( );

1416.

⟨ Define **Solid** functions 1337 ⟩ +≡
  void Solid :: *suppress_output*( )
  {
    do_output = *false*;
    return;
  }

1417. Unsuppress output.

⟨ Declare **Solid** functions 1336 ⟩ +≡
  virtual void *unsuppress_output*( );

1418.

⟨ Define **Solid** functions 1337 ⟩ +≡
  void Solid :: *unsuppress_output*( )
  {
    do_output = *true*;
    return;
  }

1419. Output. [LDF 2002.10.02.] In **Picture** :: *output*( ), *shapes* is sorted according to the values in *projective_extremes* for each **Shape**. However, it's possible (and even likely) that the individual **Paths** in a **Solid** are not ordered in such a way that they will be output in the correct order. Therefore, I declare a **vector**⟨**Shape** ∗⟩ *s* and put the **Paths** from *paths*, *circles*, *ellipses*, *reg_polygons*, and *rectangles* onto it. Then I sort *s* and call *output*( ) for each **Shape**. Currently, *output*( ) will resolve to **Path** :: *output*( ), because *output*( ) hasn't been overloaded for **Circle**, **Ellipse**, **Reg_Polygon**, or **Rectangle** (and probably won't be).

The invocation of *push_back*( ) in each of the four loops depends on the fact that **Path** :: *extract*( ) returns a **vector** containing only one element. That's why I use *front*( ). There is no operator or function for concatenating **vectors**, at least I couldn't find one in Stroustrup. TO DO: Get reference!

⟨ Declare **Solid** functions 1336 ⟩ +≡
  virtual void *output*( );

1420.

⟨ Define **Solid** functions 1337 ⟩ +≡

```
void Solid::output()
{
  bool DEBUG = false;   /* true */
  if (DEBUG) cout << "Entering␣Solid::output().\n";
  vector<Shape *> s;
  for (vector<Path *>::iterator iter = paths.begin(); iter ≠ paths.end(); ++iter)
    s.push_back((**iter).get_copy());
  for (vector<Circle *>::iterator iter = circles.begin(); iter ≠ circles.end(); ++iter)
    s.push_back((**iter).get_copy());
  for (vector<Ellipse *>::iterator iter = ellipses.begin(); iter ≠ ellipses.end(); ++iter)
    s.push_back((**iter).get_copy());
  for (vector<Reg_Polygon *>::iterator iter = reg_polygons.begin(); iter ≠ reg_polygons.end();
        ++iter) s.push_back((**iter).get_copy());
  for (vector<Rectangle *>::iterator iter = rectangles.begin(); iter ≠ rectangles.end(); ++iter)
    s.push_back((**iter).get_copy());
  sort(s.begin(), s.end(), Compare_maximum_z());
  for (vector<Shape *>::iterator iter = s.begin(); iter ≠ s.end(); ++iter) {
    (**iter).output();
    delete (*iter);
  }
  if (DEBUG) cout << "Exiting␣Solid::output().\n";
}
```

1421. Drawing and filling.

1422. Process vectors for *draw*(). [LDF 2002.10.09.] Added this section. The same things are done to each of the **Shape** ∗ vectors *paths*, *ellipses*, *circles*, *reg_polygons*, and *rectangles*, so I've put the code in this named section. Each time it it's used, *iter* is an iterator for a different vector.

⟨ Process vectors for *draw*( ) 1422 ⟩ ≡

```
{
if (c_iter ≠ v.end()) {
  color_ptr = *c_iter++;
}
(**iter).set_fill_draw_value(DRAW);    /* LDF 2002.10.09. Added code for handling draw_color. */
if (DEBUG) {
  cout << "color_ptr->get_use_name()␣==␣" << color_ptr→get_use_name() << endl << flush;
}
if (color_ptr→get_use_name() ≡ false) {
if (DEBUG) cout << "Allocating␣memory␣for␣Color.\n" << flush;
Color *c = create_new < Color > (0);
*c = *color_ptr;
(**iter).set_draw_color(c); }
else {
  if (DEBUG) cout << "color_ptr->get_name()␣==␣" << color_ptr→get_name() << endl << flush;
  (**iter).set_draw_color(color_ptr);
}
(**iter).set_fill_color(static_cast⟨Color *⟩(0));
(**iter).set_dash_pattern(ddashed);
(**iter).set_pen(ppen); }
```

This code is used in section 1424.

1423. Draw.

⟨ Declare **Solid** functions 1336 ⟩ +≡

```
virtual void draw(const vector⟨const Color *⟩ v = Colors::default_color_vector, const string
    ddashed = "", const string ppen = "", Picture &picture = current_picture) const;
```

1424.

⟨ Define **Solid** functions 1337 ⟩ +≡

```
void Solid::draw(const vector<const Color *> v, const string ddashed, const string ppen, Picture
        &picture) const { bool DEBUG = false;      /* true */
    if (DEBUG) cout << "Entering Solid::draw():" << "\n" << flush;
    Solid *s = create_new < Solid > (0);
    *s = *this;
    const Color *color_ptr = Colors::default_color;
    vector<const Color *>::const_iterator c_iter = v.begin();
    for (vector<Path *>::const_iterator iter = s->paths.begin(); iter != s->paths.end(); ++iter) {
      ⟨ Process vectors for draw() 1422 ⟩
    }
    c_iter = v.begin();
    for (vector<Circle *>::const_iterator iter = s->circles.begin(); iter != s->circles.end(); ++iter) {
      ⟨ Process vectors for draw() 1422 ⟩
    }
    c_iter = v.begin();
    for (vector<Ellipse *>::const_iterator iter = s->ellipses.begin(); iter != s->ellipses.end(); ++iter) {
      ⟨ Process vectors for draw() 1422 ⟩
    }      /* for */
    c_iter = v.begin();
    for (vector<Reg_Polygon *>::const_iterator iter = s->reg_polygons.begin();
            iter != s->reg_polygons.end(); ++iter) {
      ⟨ Process vectors for draw() 1422 ⟩
    }
    for (vector<Rectangle *>::const_iterator iter = s->rectangles.begin(); iter != s->rectangles.end();
            ++iter) {
      ⟨ Process vectors for draw() 1422 ⟩
    }
    picture += dynamic_cast<Shape *>(s);
    if (DEBUG) {
      cout << "Exiting Solid::draw():" << "\n" << flush;
    }
    }
```

1425. Process vectors for *fill*(). [LDF 2002.10.09.] Added this section. The same things are done to each of the **Shape** ∗ vectors *paths*, *ellipses*, *circles*, *reg_polygons*, and *rectangles*, so I've put the code in this named section. Each time it it's used, *iter* is an iterator for a different vector.

Log

[LDF 2003.08.10.] Now setting pen to "", because I've removed the pen argument from *fill*( ).

```
⟨ Process vectors for fill( ) 1425 ⟩ ≡
  {
  if (c_iter ≠ v.end( )) {
    color_ptr = ∗c_iter ++;
  }
  (∗∗iter).set_fill_draw_value(FILL);
    /∗ LDF 2002.10.09. Added code for handling draw_color and fill_color. ∗/
  if (DEBUG) {
    cout ≪ "color_ptr->get_use_name()␣==␣" ≪ color_ptr→get_use_name( ) ≪ endl ≪ flush;
  }
  if (color_ptr→get_use_name( ) ≡ false) {
  if (DEBUG) cout ≪ "Allocating␣memory␣for␣Color.\n" ≪ flush;
  Color ∗c = create_new < Color > (0);
  ∗c = ∗color_ptr;
  (∗∗iter).set_fill_color(c); }
  else {
    if (DEBUG) cout ≪ "color_ptr->get_name()␣==␣" ≪ color_ptr→get_name( ) ≪ endl ≪ flush;
    (∗∗iter).set_fill_color(color_ptr);
  }
  (∗∗iter).set_draw_color(static_cast⟨Color ∗⟩(0));
  (∗∗iter).set_dash_pattern("");
  (∗∗iter).set_pen(""); }
```

This code is used in section 1427.

1426. Fill.

Log

[LDF 2003.08.10.] Removed pen argument, since filling doesn't use a pen.

```
⟨ Declare Solid functions 1336 ⟩ +≡
  virtual void fill(const vector⟨const Color ∗⟩ v = Colors::default_color_vector, Picture
      &picture = current_picture) const;
```

1427.

⟨ Define **Solid** functions 1337 ⟩ +≡

```
void Solid::fill(const vector<const Color *> v, Picture &picture) const { bool DEBUG = false;
      /* true */
    if (DEBUG) cout << "Entering Solid::fill():" << "\n" << flush;
    Solid *s = create_new < Solid > (0);
    *s = *this;
    const Color *color_ptr = Colors::default_color;
    vector<const Color *>::const_iterator c_iter = v.begin();
    for (vector<Path *>::const_iterator iter = s->paths.begin(); iter != s->paths.end(); ++iter) {
      ⟨ Process vectors for fill() 1425 ⟩
    }
    c_iter = v.begin();
    for (vector<Circle *>::const_iterator iter = s->circles.begin(); iter != s->circles.end(); ++iter) {
      ⟨ Process vectors for fill() 1425 ⟩
    }
    c_iter = v.begin();
    for (vector<Ellipse *>::const_iterator iter = s->ellipses.begin(); iter != s->ellipses.end(); ++iter) {
      ⟨ Process vectors for fill() 1425 ⟩
    }
    c_iter = v.begin();
    for (vector<Reg_Polygon *>::const_iterator iter = s->reg_polygons.begin();
          iter != s->reg_polygons.end(); ++iter) {
      ⟨ Process vectors for fill() 1425 ⟩
    }
    c_iter = v.begin();
    for (vector<Rectangle *>::const_iterator iter = s->rectangles.begin(); iter != s->rectangles.end();
          ++iter) {
      ⟨ Process vectors for fill() 1425 ⟩
    }
    picture += dynamic_cast<Shape *>(s);
    if (DEBUG) {
      cout << "Exiting Solid::fill():" << "\n" << flush;
    }
    }
```

1428. Process vectors for *filldraw*(). [LDF 2002.10.09.] Added this section. The same things are done to each of the **Shape** ∗ vectors *paths*, *ellipses*, *circles*, *reg_polygons*, and *rectangles*, so I've put the code in this named section. Each time it it's used, *iter* is an iterator for a different vector.

⟨ Process vectors for *filldraw* ( ) 1428 ⟩ ≡

```
{
if (draw_color_iter ≠ draw_colors.end()) {
  draw_color_ptr = ∗draw_color_iter++;
}
if (fill_color_iter ≠ fill_colors.end()) {
  fill_color_ptr = ∗fill_color_iter++;
}
(∗∗iter).set_fill_draw_value(FILLDRAW);
if (DEBUG) {
  cout ≪ "draw_color_ptr->get_use_name()␣==␣" ≪ draw_color_ptr→get_use_name() ≪ endl ≪ flush;
}
if (draw_color_ptr→get_use_name() ≡ false) {
if (DEBUG) cout ≪ "Allocating␣memory␣for␣Color.\n" ≪ flush;
Color ∗c = create_new < Color > (0);
∗c = ∗draw_color_ptr;
(∗∗iter).set_draw_color(c); }
else {
  if (DEBUG)
    cout ≪ "draw_color_ptr->get_name()␣==␣" ≪ draw_color_ptr→get_name() ≪ endl ≪ flush;
  (∗∗iter).set_draw_color(draw_color_ptr);
}
if (DEBUG) {
  cout ≪ "fill_color_ptr->get_use_name()␣==␣" ≪ fill_color_ptr→get_use_name() ≪ endl ≪ flush;
}
if (fill_color_ptr→get_use_name() ≡ false) {
if (DEBUG) cout ≪ "Allocating␣memory␣for␣Color.\n" ≪ flush;
Color ∗c = create_new < Color > (0);
∗c = ∗fill_color_ptr;
(∗∗iter).set_fill_color(c); }
else {
  if (DEBUG)
    cout ≪ "fill_color_ptr->get_name()␣==␣" ≪ fill_color_ptr→get_name() ≪ endl ≪ flush;
  (∗∗iter).set_fill_color(fill_color_ptr);
}
(∗∗iter).set_pen(ppen);
(∗∗iter).set_dash_pattern(ddashed); }
```

This code is used in section 1430.

1429. Filldraw.

⟨ Declare **Solid** functions 1336 ⟩ +≡

```
virtual void filldraw(const vector⟨const Color ∗⟩ draw_colors = Colors::default_color_vector, const
    vector⟨const Color ∗⟩ fill_colors = Colors::background_color_vector, const string
    ddashed = "", const string ppen = "", Picture &picture = current_picture) const;
```

1430.

⟨ Define **Solid** functions 1337 ⟩ +≡

```
void Solid::filldraw(const vector<const Color *> draw_colors, const vector<const Color *>
        fill_colors, const string ddashed, const string ppen, Picture &picture) const { bool
        DEBUG = false;   /* true */
    if (DEBUG) cout << "Entering␣Solid::filldraw():" << "\n" << flush;
    Solid *s = create_new < Solid > (0);
    *s = *this;
    const Color *draw_color_ptr;
    const Color *fill_color_ptr;
    vector<const Color *>::const_iterator draw_color_iter = draw_colors.begin();
    vector<const Color *>::const_iterator fill_color_iter = fill_colors.begin();
    for (vector<Path *>::const_iterator iter = s->paths.begin(); iter ≠ s->paths.end(); ++iter) {
      ⟨ Process vectors for filldraw() 1428 ⟩
    }
    draw_color_iter = draw_colors.begin();
    fill_color_iter = fill_colors.begin();
    for (vector<Circle *>::const_iterator iter = s->circles.begin(); iter ≠ s->circles.end(); ++iter) {
      ⟨ Process vectors for filldraw() 1428 ⟩
    }
    draw_color_iter = draw_colors.begin();
    fill_color_iter = fill_colors.begin();
    for (vector<Ellipse *>::const_iterator iter = s->ellipses.begin(); iter ≠ s->ellipses.end(); ++iter) {
      ⟨ Process vectors for filldraw() 1428 ⟩
    }
    draw_color_iter = draw_colors.begin();
    fill_color_iter = fill_colors.begin();
    for (vector<Reg_Polygon *>::const_iterator iter = s->reg_polygons.begin();
          iter ≠ s->reg_polygons.end(); ++iter) {
      ⟨ Process vectors for filldraw() 1428 ⟩
    }
    for (vector<Rectangle *>::const_iterator iter = s->rectangles.begin(); iter ≠ s->rectangles.end();
          ++iter) {
      ⟨ Process vectors for filldraw() 1428 ⟩
    }
    picture += dynamic_cast<Shape *>(s);
    if (DEBUG) {
      cout << "Exiting␣Solid::filldraw():" << "\n" << flush;
    }
    }
```

1431. Process vectors for *undraw*(). [LDF 2002.10.09.] Added this section. The same things are done to each of the **Shape** ∗ vectors *paths*, *ellipses*, *circles*, *reg_polygons*, and *rectangles*, so I've put the code in this named section. Each time it it's used, *iter* is an iterator for a different vector.

```
⟨ Process vectors for undraw( ) 1431 ⟩ ≡
  {
    (∗∗iter).set_fill_draw_value(UNDRAW);
    (∗∗iter).set_draw_color(static_cast⟨Color ∗⟩(0));
    (∗∗iter).set_fill_color(static_cast⟨Color ∗⟩(0));
    (∗∗iter).set_dash_pattern(ddashed);
    (∗∗iter).set_pen(ppen);
  }
```

This code is used in section 1433.

1432. Undraw.

```
⟨ Declare Solid functions 1336 ⟩ +≡
  virtual void undraw(const string ddashed = "", const string ppen = "", Picture
      &picture = current_picture) const;
```

1433.

```
⟨ Define Solid functions 1337 ⟩ +≡
  void Solid::undraw(const string ddashed, const string ppen, Picture &picture) const{ bool
        DEBUG = false;    /∗ true ∗/
      if (DEBUG) cout ≪ "Entering␣Solid::undraw():" ≪ "\n" ≪ flush;
      Solid ∗s = create_new < Solid > (0);
      ∗s = ∗this;
      for (vector⟨Path ∗⟩::const_iterator iter = s→paths.begin( ); iter ≠ s→paths.end( ); ++iter) {
        ⟨ Process vectors for undraw( ) 1431 ⟩
      }
      for (vector⟨Circle ∗⟩::const_iterator iter = s→circles.begin( ); iter ≠ s→circles.end( ); ++iter) {
        ⟨ Process vectors for undraw( ) 1431 ⟩
      }
      for (vector⟨Ellipse ∗⟩::const_iterator iter = s→ellipses.begin( ); iter ≠ s→ellipses.end( ); ++iter) {
        ⟨ Process vectors for undraw( ) 1431 ⟩
      }
      for (vector⟨Reg_Polygon ∗⟩::const_iterator iter = s→reg_polygons.begin( );
            iter ≠ s→reg_polygons.end( ); ++iter) {
        ⟨ Process vectors for undraw( ) 1431 ⟩
      }
      for (vector⟨Rectangle ∗⟩::const_iterator iter = s→rectangles.begin( ); iter ≠ s→rectangles.end( );
            ++iter) {
        ⟨ Process vectors for undraw( ) 1431 ⟩
      }
      picture += dynamic_cast⟨Shape ∗⟩(s);
      if (DEBUG) {
        cout ≪ "Exiting␣Solid::undraw():" ≪ "\n" ≪ flush;
      }
      }
```

1434. Process vectors for *unfill*().

Log

[LDF 2002.10.09.] Added this section. The same things are done to each of the **Shape** ∗ vectors *paths*, *ellipses*, *circles*, *reg_polygons*, and *rectangles*, so I've put the code in this named section. Each time it it's used, *iter* is an iterator for a different vector.

[LDF 2003.08.10.] Now setting pen to `""`, since I've removed the pen argument to *unfill*( ).

⟨ Process vectors for *unfill*( ) 1434 ⟩ ≡
```
{
  (**iter).set_fill_draw_value(UNFILL);
  (**iter).set_draw_color(static_cast<Color *>(0));
  (**iter).set_fill_color(static_cast<Color *>(0));
  (**iter).set_dash_pattern("");
  (**iter).set_pen("");
}
```
This code is used in section 1436.

1435. Unfill.

Log

[LDF 2003.08.10.] Removed the pen argument, since unfilling doesn't use a pen.

⟨ Declare **Solid** functions 1336 ⟩ +≡
```
virtual void unfill(Picture &picture = current_picture) const;
```

1436.

⟨ Define **Solid** functions 1337 ⟩ +≡
```
void Solid::unfill(Picture &picture) const{ bool DEBUG = false;   /* true */
    if (DEBUG) cout << "Entering␣Solid::unfill():" << "\n" << flush;
    Solid *s = create_new < Solid > (0);
    *s = *this;
    for (vector⟨Path *⟩::const_iterator iter = s→paths.begin(); iter ≠ s→paths.end(); ++iter) {
      ⟨ Process vectors for unfill() 1434 ⟩
    }
    for (vector⟨Circle *⟩::const_iterator iter = s→circles.begin(); iter ≠ s→circles.end(); ++iter) {
      ⟨ Process vectors for unfill() 1434 ⟩
    }
    for (vector⟨Ellipse *⟩::const_iterator iter = s→ellipses.begin(); iter ≠ s→ellipses.end(); ++iter) {
      ⟨ Process vectors for unfill() 1434 ⟩
    }
    for (vector⟨Reg_Polygon *⟩::const_iterator iter = s→reg_polygons.begin();
            iter ≠ s→reg_polygons.end(); ++iter) {
      ⟨ Process vectors for unfill() 1434 ⟩
    }
    for (vector⟨Rectangle *⟩::const_iterator iter = s→rectangles.begin(); iter ≠ s→rectangles.end();
            ++iter) {
      ⟨ Process vectors for unfill() 1434 ⟩
    }
    picture += dynamic_cast⟨Shape *⟩(s);
    if (DEBUG) {
      cout << "Exiting␣Solid::unfill():" << "\n" << flush;
    }
    }
```

1437. Process vectors for *unfilldraw*(). [LDF 2002.10.09.] Added this section. The same things are done to each of the **Shape** ∗ vectors *paths*, *ellipses*, *circles*, *reg_polygons*, and *rectangles*, so I've put the code in this named section. Each time it it's used, *iter* is an iterator for a different vector.

⟨ Process vectors for *unfilldraw*( ) 1437 ⟩ ≡
```
  {
    (**iter).set_fill_draw_value(UNFILLDRAW);
    (**iter).set_draw_color(static_cast⟨Color *⟩(0));
    (**iter).set_fill_color(static_cast⟨Color *⟩(0));
    (**iter).set_dash_pattern(ddashed);
    (**iter).set_pen(ppen);
  }
```
This code is used in section 1439.

1438. Unfilldraw. [LDF 2002.10.09.] Unlike **Path** :: *unfilldraw*( ), **Solid** :: *unfilldraw*( ) behaves like METAPOST's *unfilldraw* command, i.e., it unfills and undraws. I intend to change **Path** :: *unfilldraw*( ) so that it also behaves this way.

!! [LDF 2002.10.09.] Check this: the correct code is written to *out_stream*, but after *filldraw*( ) and *unfill_draw*( ), the outline is visible, but no lines inside the outline. TO DO: Check what **filldraw** and **unfilldraw** mean in METAPOST and METAFONT.

⟨ Declare **Solid** functions 1336 ⟩ +≡
```
  virtual void unfilldraw(const string ddashed = "", const string ppen = "", Picture
      &picture = current_picture) const;
```

1439.

⟨ Define **Solid** functions 1337 ⟩ +≡

void Solid :: *unfilldraw*(**const string** *ddashed*, **const string** *ppen*, **Picture** &*picture*) **const** { **bool** DEBUG = *false*; /* *true* */

if (DEBUG) *cout* ≪ "Entering␣Solid::unfilldraw():" ≪ "\n" ≪ *flush*;
Solid ∗*s* = *create_new* < **Solid** > (0);
∗*s* = ∗*this*;
for (**vector**⟨**Path** ∗⟩ :: *const_iterator iter* = *s*→*paths*.*begin*( ); *iter* ≠ *s*→*paths*.*end*( ); ++*iter*) {
⟨ Process vectors for *unfilldraw*( ) 1437 ⟩
} /* **for** */
for (**vector**⟨**Circle** ∗⟩ :: *const_iterator iter* = *s*→*circles*.*begin*( ); *iter* ≠ *s*→*circles*.*end*( ); ++*iter*) {
⟨ Process vectors for *unfilldraw*( ) 1437 ⟩
}
for (**vector**⟨**Ellipse** ∗⟩ :: *const_iterator iter* = *s*→*ellipses*.*begin*( ); *iter* ≠ *s*→*ellipses*.*end*( ); ++*iter*) {
⟨ Process vectors for *unfilldraw*( ) 1437 ⟩
}
for (**vector**⟨**Reg_Polygon** ∗⟩ :: *const_iterator iter* = *s*→*reg_polygons*.*begin*( ); *iter* ≠ *s*→*reg_polygons*.*end*( ); ++*iter*) {
⟨ Process vectors for *unfilldraw*( ) 1437 ⟩
}
for (**vector**⟨**Rectangle** ∗⟩ :: *const_iterator iter* = *s*→*rectangles*.*begin*( ); *iter* ≠ *s*→*rectangles*.*end*( ); ++*iter*) {
⟨ Process vectors for *unfilldraw*( ) 1437 ⟩
}
picture += **dynamic_cast**⟨**Shape** ∗⟩(*s*);
if (DEBUG) {
cout ≪ "Exiting␣Solid::unfilldraw():" ≪ "\n" ≪ *flush*;
}
}

1440. Putting Solid together.

1441. This is what's compiled.

⟨ Include files 6 ⟩
⟨ Version control identifier 5 ⟩
⟨ Define **class Solid** 1333 ⟩
⟨ Define **static const Solid** data members 1334 ⟩
⟨ Define **Solid** functions 1337 ⟩
⟨ Declare non-member template functions for **Solid** 1341 ⟩

1442. This is what's written to `solids.h`.

⟨ `solids.h` 1442 ⟩ ≡
⟨ Define **class Solid** 1333 ⟩
⟨ Declare non-member template functions for **Solid** 1341 ⟩

1443. Solid_Faced (`solfaced.web`).

Log

[LDF 2002.09.26.] Created this file.

[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: solfaced.web,v 1.4 2004/01/12 21:33:15 lfinsto1 Exp $";
```

1444. Include files.

⟨ Include files 6 ⟩ +≡
```
#include "loader.h"
#include "pspglb.h"
#include "io.h"
#include "colors.h"
#include "transfor.h"
#include "shapes.h"
#include "pictures.h"
#include "points.h"
#include "lines.h"
#include "planes.h"
#include "paths.h"
#include "curves.h"
#include "polygons.h"
#include "rectangs.h"
#include "ellipses.h"
#include "circles.h"
#include "solids.h"
```

1445. Solid_Faced class definition.

⟨ Define **class Solid_Faced** 1445 ⟩ ≡
```
class Solid_Faced : public Solid {
protected: unsigned short faces;
   unsigned short vertices;
   unsigned short edges;
public: ⟨ Declare Solid_Faced functions 1446 ⟩
};
```
This code is used in sections 1449 and 1450.

1446.

⟨ Declare **Solid_Faced** functions 1446 ⟩ ≡

This code is used in section 1445.

1447.

⟨ Define **Solid_Faced** functions 1447 ⟩ ≡

This code is used in section 1449.

1448. Putting Solid_Faced together.

1449. This is what's compiled.

⟨ Include files 6 ⟩
⟨ Version control identifier 5 ⟩
⟨ Define **class Solid_Faced** 1445 ⟩
⟨ Define **Solid_Faced** functions 1447 ⟩

1450. This is what's written to `solfaced.h`.

⟨ `solfaced.h` 1450 ⟩ ≡
⟨ Define **class Solid_Faced** 1445 ⟩

1451. Cuboid (`cuboid.web`).

Log

[LDF 2002.04.22.] Created this file. When I've found out what the English word is for "Quader", I'll change it globally.

[LDF 2002.04.22.] **Cuboid** is the first three-dimensional object I've defined. I've just quickly put it together for use in a drawing. Ultimately, I'd like to derive it from **Shape**, which will require defining versions of all the pure **virtual** functions in **Shape**.

[LDF 2002.04.23.] Changed *Quader* to **Cuboid**. Haven't changed name of file, because this is more complicated, because of RCS (the source code control system).

[LDF 2002.05.03.] Changed the name of this file from `quader.web` to `cuboid.web`. This means that if you need to compare this file with revisions earlier than the initial version of this file, you'll have to check revisions of `quader.web`.

[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.

---

format *Cuboid* *Solid*

⟨ Version control identifier 5 ⟩ +≡
static string *rcs_id* = `"$Id:␣cuboid.web,v␣1.6␣2004/01/12␣21:27:51␣lfinsto1␣Exp␣$"`;

1452. 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"
#include "points.h"
#include "lines.h"
#include "planes.h"
#include "paths.h"
#include "curves.h"
#include "polygons.h"
#include "rectangs.h"
#include "ellipses.h"
#include "circles.h"
#include "solids.h"
#include "solfaced.h"
```

1453. Cuboid class definition.

Log

[LDF 2002.04.22.] Added this **class** declaration.
[LDF 2003.08.10.] Removed *dihedral_angle*.

⟨ Define **class Cuboid** 1453 ⟩ ≡
```
class Cuboid : public Solid_Faced {
protected: real height;
  real width;
  real depth;
public: ⟨ Declare Cuboid functions 1455 ⟩
};
```
This code is used in sections 1468 and 1469.

1454. Constructors and setting functions.

1455. Default constructor. No arguments. [LDF 2002.04.22.] Added this function.

⟨ Declare **Cuboid** functions 1455 ⟩ ≡
```
Cuboid();
```
See also sections 1457, 1459, 1464, and 1466.

This code is used in section 1453.

1456.

⟨ Define **Cuboid** functions 1456 ⟩ ≡
```
Cuboid::Cuboid()
{
  on_free_store = false;
  do_output = true;
  projective_extremes.resize(6, 0);
  faces = 6;
  vertices = 8;
  edges = 12;
}
```
See also sections 1458, 1460, 1465, and 1467.

This code is used in section 1468.

1457. Copy constructor. [LDF 2002.05.03.] Added this function.

⟨ Declare **Cuboid** functions 1455 ⟩ +≡
```
Cuboid(const Cuboid &c);
```

1458.

⟨ Define **Cuboid** functions 1456 ⟩ +≡
Cuboid :: **Cuboid**(**const Cuboid** &c){ *on_free_store* = *false*;
do_output = *true*;
projective_extremes.*resize*(6, 0);
faces = 6;
vertices = 8;
edges = 12; **for** (**vector**⟨**Rectangle** ∗⟩ :: *const_iterator iter* = *c.rectangles*.*begin*( );
iter ≠ *c.rectangles*.*end*( ); *iter*++) { *rectangles*.*push_back* (*create_new* < **Rectangle** > (0)) ;
∗(*rectangles*.*back*( )) = ∗∗*iter*; } }

1459. Center, height, width, depth, and angles. [LDF 2002.10.06.] Added this constructor.

⟨ Declare **Cuboid** functions 1455 ⟩ +≡
Cuboid(**const Point** &c, **const real** h, **const real** w, **const real** d, **const real** $x = 0$, **const real** $y = 0$, **const real** $z = 0$);

1460.

⟨ Define **Cuboid** functions 1456 ⟩ +≡

Cuboid :: **Cuboid**(**const Point** &*c*, **const real** *h*, **const real** *w*, **const real** *d*, **const real** *x*, **const real** *y*, **const real** *z*): *height*(*h*), *width*(*w*), *depth*(*d*) { **bool** DEBUG = *false*; /* *true* */

on_free_store = *false*;
do_output = *true*;
projective_extremes.*resize*(6, 0);
center = *c*;
faces = 6;
vertices = 8;
edges = 12;
Point *pts*[9];
pts[1].*shift*(−.5 ∗ *width*, −.5 ∗ *height*, −.5 ∗ *depth*);
pts[2].*shift*(.5 ∗ *width*, −.5 ∗ *height*, −.5 ∗ *depth*);
pts[3].*shift*(.5 ∗ *width*, .5 ∗ *height*, −.5 ∗ *depth*);
pts[4].*shift*(−.5 ∗ *width*, .5 ∗ *height*, −.5 ∗ *depth*);
pts[5].*shift*(−.5 ∗ *width*, −.5 ∗ *height*, .5 ∗ *depth*);
pts[6].*shift*(.5 ∗ *width*, −.5 ∗ *height*, .5 ∗ *depth*);
pts[7].*shift*(.5 ∗ *width*, .5 ∗ *height*, .5 ∗ *depth*);
pts[8].*shift*(−.5 ∗ *width*, .5 ∗ *height*, .5 ∗ *depth*); **for** (**int** *i* = 0; *i* < 6; *i*++) { *rectangles*.*push_back* (*create_new* < **Rectangle** > (0)) ; } *rectangles*[0]→*set*(*pts*[1], *pts*[2], *pts*[3], *pts*[4]);
/* front */
rectangles[1]→*set*(*pts*[5], *pts*[6], *pts*[7], *pts*[8]); /* back */
rectangles[2]→*set*(*pts*[1], *pts*[4], *pts*[8], *pts*[5]); /* left */
rectangles[3]→*set*(*pts*[2], *pts*[6], *pts*[7], *pts*[3]); /* right */
rectangles[4]→*set*(*pts*[3], *pts*[7], *pts*[8], *pts*[4]); /* top */
rectangles[5]→*set*(*pts*[1], *pts*[2], *pts*[6], *pts*[5]); /* bottom */
rotate(*x*, *y*, *z*);
shift(*c*);
if (DEBUG)
  for (**int** *i* = 1; *i* < 9; *i*++) *pts*[*i*].*dotlabel*(*i*);
}

1461. Pseudo-constructor for dynamic allocation.

1462. Pointer argument.

Log

[LDF 2002.04.22.] Added this function.

[LDF 2003.12.30.] Replaced **Cuboid** :: *create_new_cuboid*( ) with a specialization of **template**⟨**class** *C*⟩ *C*∗*create_new*( ) for **Cuboid**. The argument is now **const**.

⟨ Declare non-member template functions for **Cuboid** 1462 ⟩ ≡

Cuboid ∗*create_new*(**const Cuboid** ∗*c*);

See also section 1463.

This code is used in sections 1468 and 1469.

1463. Reference argument.

Log

LDF 2003.12.30. Added this function.

⟨ Declare non-member template functions for **Cuboid** 1462 ⟩ +≡
Cuboid ∗*create_new*(**const Cuboid** &*c*);

1464. Destructor. !! Make sure to delete anything else that I allocate dynamically!

⟨ Declare **Cuboid** functions 1455 ⟩ +≡
∼**Cuboid**( );

1465.

⟨ Define **Cuboid** functions 1456 ⟩ +≡

```
Cuboid :: ~Cuboid( )
{
  for (vector⟨Rectangle *⟩ :: iterator iter = rectangles.begin( ); iter ≠ rectangles.end( ); iter++) {
    delete *iter;
  }
  rectangles.clear( );
}
```

1466. Assignment.

⟨ Declare **Cuboid** functions 1455 ⟩ +≡
void operator=(**const Cuboid** &*c*);

1467.

⟨ Define **Cuboid** functions 1456 ⟩ +≡

```
void Cuboid :: operator=(const Cuboid &c)
{
  this→Solid :: operator=(c);
  height = c.height;
  width = c.width;
  depth = c.depth;
}
```

1468. Putting Cuboid together. This is what's compiled.

⟨ Include files 6 ⟩
⟨ Version control identifier 5 ⟩
⟨ Define **class Cuboid** 1453 ⟩
⟨ Define **Cuboid** functions 1456 ⟩
⟨ Declare non-member template functions for **Cuboid** 1462 ⟩

1469. This is what's written to `cuboid.h`.

⟨ `cuboid.h` 1469 ⟩ ≡
⟨ Define **class Cuboid** 1453 ⟩
⟨ Declare non-member template functions for **Cuboid** 1462 ⟩

1470. Polyhedra (`polyhedra.web`). [LDF 2002.11.12.] TO DO: Set *center* for **Dodecahedron**, **Icosahedron**, and *Trunc_Octahedron*. Find out how to set it for **Tetrahedron**.

[LDF 2002.11.12.] TO DO: Add assignment operators for **Polyhedra**! The individual types will need there own, but they can call **Polyhedron** :: **operator**=( ).

Log

[LDF 2002.09.26.] Created this file.

[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* = `"$Id:␣polyhed.web,v␣1.5␣2004/01/12␣21:32:19␣lfinsto1␣Exp␣$"`;

1471. 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"
#include "points.h"
#include "lines.h"
#include "planes.h"
#include "paths.h"
#include "curves.h"
#include "polygons.h"
#include "rectangs.h"
#include "ellipses.h"
#include "circles.h"
#include "solids.h"
#include "solfaced.h"
```

1472. Polyhedron class definition. [LDF 2002.10.06.] **Polyhedron** is meant to be used only as a base class, so there's no need for constructors or setting functions.

TO DO: [LDF 2003.08.15.] If I add any functions, I should add an explanation to "`@node Polyhedron Getstart`" in `DOCUMENTATION/gssolfig.texi` about abstract or non-abstract base classes.

Log

[LDF 2002.11.08.] Got rid of pure **virtual** function **Polyhedron** :: *get_net*( ). I've made it **static** in the classes derived from **Polyhedron**, which makes more sense. **virtual** functions must be non-**static**.

---

format *Polyhedron* *Reg_Polygon*

⟨ Define **class Polyhedron** 1472 ⟩ ≡

```
class Polyhedron : public Solid_Faced {
protected: unsigned short number_of_polygon_types;
  real face_radius;
  real edge_radius;
  real vertex_radius;
public: ⟨ Declare Polyhedron functions 1473 ⟩
};
```

This code is used in sections 1534 and 1535.

1473. Intersection. !! [LDF 2003.04.15.] START HERE. This function doesn't work yet.

Log

[LDF 2003.04.09.] Added this section, and the function it contains.

[LDF 2003.04.15.] Commented-out this function, since I need to get **Reg_Polygon** :: *intersection_points* ( ) working first.

⟨ Declare **Polyhedron** functions 1473 ⟩ ≡

```
#if 0
  virtual vector⟨Point⟩ intersection_points(const Reg_Polygon &r) const;
#endif
```

This code is used in section 1472.

1474.

⟨ Define **Polyhedron** functions 1474 ⟩ ≡

```
#if 0
  vector⟨Point⟩ Polyhedron::intersection_points(const Reg_Polygon &r) const
  {
    vector⟨Point⟩ v;
    vector⟨Point⟩ w;
    for (vector⟨Reg_Polygon *⟩::const_iterator iter0 = reg_polygons.begin( ); iter0 ≠ reg_polygons.end( );
          ++iter0) {
      for (vector⟨Reg_Polygon *⟩::const_iterator iter1 = iter0 + 1; iter1 ≠ reg_polygons.end( );
            ++iter1) {
        v = (**iter0).intersection_points(**iter1);
        cout ≪ "v.size()␣==␣" ≪ v.size( ) ≪ endl ≪ flush;
        for (vector⟨Point⟩::iterator pt_iter = v.begin( ); pt_iter ≠ v.end( ); ++pt_iter)
          w.push_back(*pt_iter);
      }
    }
    return w;
  }
#endif
```

This code is used in section 1534.

1475. Regular Platonic Polyhedra.

1476. Tetrahedron.

Log

[LDF 2002.11.12.] Added this section.

1477. Tetrahedron class definition.

Log

[LDF 2002.11.12.] Added this section.

format *Tetrahedron* *Polyhedron*

⟨ Define **class Tetrahedron** 1477 ⟩ ≡

```
  class Tetrahedron : public Polyhedron {
  protected: static const real dihedral_angle;      /* In radians! */
    real triangle_radius;
  public: ⟨ Declare Tetrahedron functions 1480 ⟩
  };
```

This code is used in sections 1534 and 1535.

1478. Define static const Tetrahedron data members.

Log

[LDF 2002.11.12.] Added this section.

⟨ Define **static const Tetrahedron** data members 1478 ⟩ ≡

```
  const real Tetrahedron::dihedral_angle = PI * (70 + 32/60.0)/180.0;
```

This code is used in section 1534.

1479. Constructors and setting functions.

1480. Default constructor. (No arguments.)

Log

[LDF 2002.11.12.] Added this function.

⟨ Declare **Tetrahedron** functions 1480 ⟩ ≡

```
Tetrahedron();
```

See also sections 1483, 1487, 1489, and 1491.

This code is used in section 1477.

1481.

⟨ Define **Tetrahedron** functions 1481 ⟩ ≡

```
Tetrahedron::Tetrahedron()
{
  on_free_store = false;   /* from Solid. */
  do_output = true;
  faces = 4;   /* from Solid_Faced. */
  vertices = 4;
  edges = 6;
  center = INVALID_POINT;   /* from Polyhedron. */
  number_of_polygon_types = 1;
  face_radius = edge_radius = vertex_radius = INVALID_REAL;
  triangle_radius = INVALID_REAL;   /* From Tetrahedron. */
}
```

See also sections 1484, 1485, 1486, 1488, 1490, and 1492.

This code is used in section 1534.

1482. Center, diameter of triangle, and angles.

1483. Constructor.

Log

[LDF 2002.11.12.] Added this function.
[LDF 2003.04.27.] Got this function to work, at least in a rudimentary way.
[LDF 2002.08.12.] Rewrote this function. It now works properly.

⟨ Declare **Tetrahedron** functions 1480 ⟩ +≡

```
Tetrahedron(const Point &p, const real diameter_of_triangle, real angle_x = 0, real angle_y = 0, real
    angle_z = 0);
```

1484.

⟨ Define **Tetrahedron** functions 1481 ⟩ +≡

```
  Tetrahedron::Tetrahedron(const Point &p, const real triangle_diameter, real angle_x, real
          angle_y, real angle_z){ bool DEBUG = true;    /* false */
      on_free_store = false;    /* from Solid. */
      do_output = true;
      faces = 4;    /* from Solid_Faced. */
      vertices = 4;
      edges = 6;
      number_of_polygon_types = 1;
#if 0    /* START HERE. TO DO: Must calculate these! */
      face_radius = 0;
      edge_radius = 0;
      vertex_radius = 0;
#endif
      triangle_radius = triangle_diameter/2.0;
      reg_polygons = get_net(triangle_diameter);
      real angle = 180 − (dihedral_angle * 180/PI);
      Point pts[11];
      int i;
      for (i = 0; i < 3; ++i) pts[i] = reg_polygons[0]→get_point(i);
      reg_polygons[1]→rotate(pts[0], pts[1], angle);
      reg_polygons[2]→rotate(pts[2], pts[0], angle);
      reg_polygons[3]→rotate(pts[1], pts[2], −angle);
#if 0
      for (i = 0; i < 3; ++i) pts[i].label(i, "");
#endif
      for (i = 3; i < 7; ++i) {
        pts[i] = reg_polygons[i − 3]→get_center( );
#if 0
        pts[i].label(i, "");
#endif
      }
      pts[7] = reg_polygons[3]→get_point(0);
#if 0
      pts[7].label(7, "");
#endif
      pts[8] = pts[0].mediate(pts[1]);
      pts[9] = pts[1].mediate(pts[2]);
      pts[10] = pts[2].mediate(pts[0]);
#if 0
      for (i = 8; i < 11; i++) pts[i].label(i);
#endif
      using namespace Colors;
#if 0
      pts[0].draw(pts[6], blue);
      pts[1].draw(pts[5], red);
      pts[2].draw(pts[4], green);
      pts[3].draw(pts[7], orange);
#endif
```

1485. *center* is the intersection point of the line segments from the vertices of ∗(*reg_polygons*[0]) to the centers of the opposite faces. *distance* is the distance along one of these line segments to the intersection point divided by the length of the entire line segment. [LDF 2002.08.12.]

Since this ratio should be the same for all *Tetrahedra*, there's no need to recalculate it each time a **Tetrahedron** is constructed. In addition, intersections can't always be found, because of inaccuracies caused by rotating the triangles. [LDF 2002.08.12.]

Therefore, I've calculated distance using the commented-out code below, and now simply use the value I found. [LDF 2002.08.12.]

⟨ Define **Tetrahedron** functions 1481 ⟩ +≡

```
  real distance = 0.7499788995574951171875;
#if 0
  Point P0 = Point::intersection_point(pts[0], pts[6], pts[1], pts[5]).pt;
  P0.show("P0␣=␣0,␣6,␣1,␣5:");
  Point::intersection_point(pts[0], pts[6], pts[2], pts[4]).pt.show("0,␣6,␣2,␣4:");
  Point::intersection_point(pts[0], pts[6], pts[3], pts[7]).pt.show("0,␣6,␣3,␣7:");
  Point::intersection_point(pts[1], pts[5], pts[2], pts[4]).pt.show("1,␣5,␣2,␣4:");
  Point::intersection_point(pts[1], pts[5], pts[3], pts[7]).pt.show("1,␣5,␣3,␣7:");
  Point::intersection_point(pts[2], pts[4], pts[3], pts[7]).pt.show("2,␣4,␣3,␣7:");
  Point P1 = pts[5] − pts[1];
  Point P2 = P0 − pts[1];
  P1.show("P1");
  cout ≪ "␣␣P1.magnitude()␣==␣" ≪ P1.magnitude() ≪ endl ≪ flush;
  P2.show("P2");
  cout ≪ "P2.magnitude()␣==␣" ≪ P2.magnitude() ≪ endl ≪ flush;
  distance = (P2.magnitude()/P1.magnitude());
  cout.precision(25);
  cout ≪ "distance␣==␣" ≪ distance ≪ endl ≪ flush;
  cout.precision(6);
#endif
```

1486.

⟨ Define **Tetrahedron** functions 1481 ⟩ +≡

```
  center = pts[1].mediate(pts[5], distance);
  for (i = 0; i < 4; ++i) reg_polygons[i]→shift(−center);
  center.shift(−center);
  if (angle_x ≠ 0 ∨ angle_y ≠ 0 ∨ angle_z ≠ 0) {
    for (i = 0; i < 4; ++i) reg_polygons[i]→rotate(angle_x, angle_y, angle_z);
  }
  if (p ≠ origin) {
    center = p;
    for (i = 0; i < 4; ++i) reg_polygons[i]→shift(p);
  }
  return; }
```

1487. Setting function. [LDF 2002.11.12.] !! This works, but it fails to assign to the data members of **Tetrahedron** that are defined in its own **class** declaration. That's because neither **Tetrahedron** nor **Polyhedron** has an assignment operator yet. TO DO: Write assignment operators for **Polyhedra**!

Log

[LDF 2002.11.12.] Added this function.

⟨ Declare **Tetrahedron** functions 1480 ⟩ +≡
void *set*(**const Point** &*p*, **const real** *diameter_of_triangle*, **real** *angle_x* = 0, **real** *angle_y* = 0, **real** *angle_z* = 0);

1488.

⟨ Define **Tetrahedron** functions 1481 ⟩ +≡
void Tetrahedron :: *set*(**const Point** &*p*, **const real** *triangle_diameter*, **real** *angle_x*, **real** *angle_y*, **real** *angle_z*)
{
Tetrahedron *t*(*p*, *triangle_diameter*, *angle_x*, *angle_y*, *angle_z*);
∗*this* = *t*;
return;
}

1489. Get net. [LDF 2002.11.12.] Unlike the *get_net*( ) functions for some of the other **Polyhedra**, this function has no "**bool** *do_half*" argument. It doesn't pay for a **Tetrahedron**.

Log

[LDF 2002.11.12.] Added this function.
[LDF 2002.08.12.] Removed *center_0* argument.

⟨ Declare **Tetrahedron** functions 1480 ⟩ +≡
static vector⟨**Reg_Polygon** ∗⟩ *get_net*(**const real** *triangle_diameter*);

1490.

⟨ Define **Tetrahedron** functions 1481 ⟩ +≡

```
vector<Reg_Polygon *> Tetrahedron::get_net(const real triangle_diameter){ vector<Reg_Polygon
        *> triangles;
    int i; for (i = 0; i < 4; i++) triangles.push_back ( create_new < Reg_Polygon > (0) ) ;
    triangles[0]->set(origin, 3, triangle_diameter, 0, 180);
    triangles[1]->set(origin, 3, triangle_diameter);
    Point pts[6];
    for (i = 0; i < 3; ++i) {
      pts[i] = triangles[0]->get_point(i);
    }
    for (i = 3; i < 6; ++i) {
      pts[i] = triangles[1]->get_point(i - 3);
    }
    *triangles[2] = *triangles[3] = *triangles[1];
    triangles[1]->shift(pts[0] - pts[4]);
    triangles[2]->shift(pts[0] - pts[5]);
    triangles[3]->shift(pts[2] - pts[4]);
    return triangles; }
```

1491. Draw net. [LDF 2002.11.12.] As of this date it's necessary to rotate the triangles into the x-y plane, because **Point**::*intersection_point*( ) has a bug that I discovered when I tried to call it on **Points** in the x-z plane. It's not so terrible, because as of this date it's necessary to put the **Picture** in the x-y plane in order to use the parallel projection. The latter currently only works for the x-y plane. TO DO: Fix the bug and get parallel projection onto other major planes to work!

Log

[LDF 2002.11.12.] Added this function.

⟨ Declare **Tetrahedron** functions 1480 ⟩ +≡

```
static void draw_net(const real triangle_diameter, bool make_tabs = true);
```

1492.

```
⟨ Define Tetrahedron functions 1481 ⟩ +≡
  void Tetrahedron::draw_net(const real triangle_diameter, bool make_tabs)
  {
    vector⟨Reg_Polygon *⟩ v = get_net(triangle_diameter);
    int i;
    for (i = 0; i < 4; i++) {
      v[i]→rotate(90);
      v[i]→draw( );
      v[i]→get_center( ).label(i, "");
    }
    if (¬make_tabs) return;
    Point pts[32];
    pts[0] = v[2]→get_point(1);
    pts[1] = v[2]→get_point(2);
    pts[2] = v[1]→get_point(2);
    pts[3] = v[1]→get_point(0);
    pts[4] = v[3]→get_point(0);
    pts[5] = v[3]→get_point(1);
    pts[6] = pts[0].mediate(pts[5], .075);
    pts[7] = pts[5].mediate(pts[0], .075);
    pts[8] = pts[6];
    pts[9] = pts[7];
    pts[8] *= pts[9].shift(0, 0, 1);
    pts[10] = pts[0];
    pts[11] = pts[5];
    pts[10].rotate(pts[6], pts[8], −110);
    pts[11].rotate(pts[7], pts[9], 110);
    pts[10] = pts[6].mediate(pts[10], 1.5);
    pts[11] = pts[7].mediate(pts[11], 1.5);
#if 0
    for (i = 0; i < 8; i++) pts[i].dotlabel(i);
    pts[10].dotlabel(10);
    pts[11].dotlabel(11);
#endif
    Path p[6];
    p[0].set("--", true, &pts[6], &pts[10], &pts[11], &pts[7], 0);
    p[0].draw( );
    pts[12] = pts[6].mediate(pts[7]);
    pts[13] = pts[10].mediate(pts[11]);
#if 0
    pts[12].dotlabel(12);
    pts[13].dotlabel(13);
#endif
    p[1].set(pts[12], pts[13]);
#if 0
    p[1].draw_help(*Colors::help_color, "");
#endif
    pts[14] = pts[12].mediate(pts[13]);
#if 0
    pts[14].dotlabel(14);
```

```
#endif
    pts[15] = pts[6].mediate(pts[7], .25);
#if 0
    pts[15].dotlabel(15);
#endif
    pts[16] = pts[14];
    pts[16].shift(pts[15] − pts[12]);
#if 0
    pts[16].dotlabel(16);
#endif
    bool_point bp = Point::intersection_point(pts[14], pts[16], pts[6], pts[10]);
    pts[17] = bp.pt;
#if 0
    pts[17].dotlabel(17);
#endif
    bp = Point::intersection_point(pts[14], pts[16], pts[7], pts[11]);
    pts[18] = bp.pt;
#if 0
    pts[18].dotlabel(18);
#endif
    p[2].set(pts[17], pts[18]);
#if 0
    p[2].draw_help(*Colors::help_color, "");
#endif
    for (i = 1; i < 16; ++i) {
      pts[19] = pts[17].mediate(pts[18], i/16.0);
      pts[19].drawdot(*Colors::default_color, "pencircle␣scaled␣.5mm");
    }
    pts[20] = pts[17];
    pts[21] = pts[18];
    p[3] = p[0];
    Transform t;
    t.shift(pts[4] − pts[5]);
    t.rotate(pts[4], pts[5]);
    pts[20] *= pts[21] *= p[3] *= t;
    p[3].draw();
    for (i = 1; i < 16; ++i) {
      pts[19] = pts[20].mediate(pts[21], i/16.0);
      pts[19].drawdot(*Colors::default_color, "pencircle␣scaled␣.5mm");
    }
    t.reset();
    t.rotate(pts[4], pts[1]);
    pts[20] *= pts[21] *= p[3] *= t;
    p[3].draw();
    for (i = 1; i < 16; ++i) {
      pts[19] = pts[20].mediate(pts[21], i/16.0);
      pts[19].drawdot(*Colors::default_color, "pencircle␣scaled␣.5mm");
    }
    pts[20] = pts[17];
    pts[21] = pts[18];
    p[3] = p[0];
```

```
    pts[20] *= pts[21] *= p[3] *= t;
    p[3].draw();
    for (i = 1; i < 16; ++i) {
      pts[19] = pts[20].mediate(pts[21], i/16.0);
      pts[19].drawdot(*Colors::default_color, "pencircle␣scaled␣.5mm");
    }
    t.reset();
    pts[22] = v[2]->get_center();
    pts[23] = pts[22];
    pts[23].shift(0, 0, 1);
    t.rotate(pts[22], pts[23], 120);
    pts[20] = pts[17];
    pts[21] = pts[18];
    p[3] = p[0];
    pts[20] *= pts[21] *= p[3] *= t;
    p[3].draw();
    for (i = 1; i < 16; ++i) {
      pts[19] = pts[20].mediate(pts[21], i/16.0);
      pts[19].drawdot(*Colors::default_color, "pencircle␣scaled␣.5mm");
    }
    t.rotate(pts[4], pts[1]);
    t.rotate(pts[0], pts[2]);
    pts[20] = pts[17];
    pts[21] = pts[18];
    p[3] = p[0];
    pts[20] *= pts[21] *= p[3] *= t;
    p[3].draw();
    for (i = 1; i < 16; ++i) {
      pts[19] = pts[20].mediate(pts[21], i/16.0);
      pts[19].drawdot(*Colors::default_color, "pencircle␣scaled␣.5mm");
    }
  }
```

1493. Dodecahedron.

1494. Dodecahedron class definition.

format *Dodecahedron* *Polyhedron*

⟨ Define **class Dodecahedron** 1494 ⟩ ≡

```
  class Dodecahedron : public Polyhedron {
  protected: static const real dihedral_angle;    /* In radians! */
    real pentagon_radius;
  public: ⟨ Declare Dodecahedron functions 1497 ⟩
  };
```

This code is used in sections 1534 and 1535.

1495. Define static const Dodecahedron data members.

Log

[LDF 2003.07.18.] Now passing "2.0" instead of "2.0" as the argument to *atan*( ). GCC 3.3 couldn't compile this file, the way it was before.

⟨ Define **static const Dodecahedron** data members 1495 ⟩ ≡
const real Dodecahedron :: *dihedral_angle* = PI − *atan*(2.0);

This code is used in section 1534.

1496. Constructors and setting functions.

1497. Default constructor. (No arguments.) [LDF 2002.09.29.] TO DO: I should set the data members of other classes to `INVALID_POINT`, `INVALID_REAL`, etc., in the default constructors, too.

⟨ Declare **Dodecahedron** functions 1497 ⟩ ≡
Dodecahedron( );

See also sections 1500, 1503, and 1505.

This code is used in section 1494.

1498.

⟨ Define **Dodecahedron** functions 1498 ⟩ ≡

```
Dodecahedron::Dodecahedron()
{
  on_free_store = false;      /* from Solid. */
  do_output = true;
  faces = 12;      /* from Solid_Faced. */
  vertices = 20;
  edges = 30;
  center = INVALID_POINT;      /* from Polyhedron. */
  number_of_polygon_types = 1;
  face_radius = edge_radius = vertex_radius = INVALID_REAL;
  pentagon_radius = INVALID_REAL;      /* From Dodecahedron. */
}
```

See also sections 1501, 1502, 1504, and 1506.

This code is used in section 1534.

1499. Center, diameter of pentagon, and angles.

1500. Constructor. [LDF 2003.08.10.] TO DO: Check the way I specify the rotations. If its not the way I think it should be, check **Transform** :: *align_with_axis*( ) and any other functions that are involved. This may be a long-term project.

Log

[LDF 2002.10.16.] Added shift to *center* and rotation.

[LDF 2003.08.10.] Rewrote this function. It had suddenly stopped working properly, probably because of changes I made to **Transform** :: *align_with_axis*( ). I'm still not entirely happy with the way I've had to specify the rotations, see the "TO DO" note of this date, above.

⟨ Declare **Dodecahedron** functions 1497 ⟩ +≡
Dodecahedron(**const Point** &*p*, **const real** *pentagon_diameter*, **real** *angle_x* = 0, **real** *angle_y* = 0, **real** *angle_z* = 0);

1501.

⟨ Define **Dodecahedron** functions 1498 ⟩ +≡

```
Dodecahedron::Dodecahedron(const Point &p, const real pentagon_diameter, real angle_x, real
        angle_y, real angle_z){ bool DEBUG = false;    /* true */
    if (DEBUG) cout << "Entering␣Dodecahedron::Dodecahedron().\n" << flush;
    on_free_store = false;    /* from Solid. */
    do_output = true;
    faces = 12;    /* from Solid_Faced. */
    vertices = 20;
    edges = 30;
    number_of_polygon_types = 1;
#if 0    /* START HERE. TO DO: Must calculate these! */
    face_radius = 0;
    edge_radius = 0;
    vertex_radius = 0;
#endif
    pentagon_radius = pentagon_diameter/2.0;
    reg_polygons = get_net(pentagon_diameter, true);
    Point pts[8];
    int i = 0;
    for (i = 0; i < 5; ++i) {
      pts[i] = reg_polygons[0]→get_point(i);
    }
    real angle = 180 − (dihedral_angle ∗ 180.0/PI);
```

1502. [LDF 2003.08.10.] Check this, as noted above.

```
⟨ Define Dodecahedron functions 1498 ⟩ +≡
  reg_polygons[1]→rotate(pts[0], pts[1], angle);
  reg_polygons[2]→rotate(pts[4], pts[0], angle);
  reg_polygons[3]→rotate(pts[3], pts[4], angle);
  reg_polygons[4]→rotate(pts[3], pts[2], angle);
  reg_polygons[5]→rotate(pts[1], pts[2], angle);
#if 0
  using namespace Colors;
  vector⟨const Color *⟩ col_vec;
  col_vec.push_back(&black);
  col_vec.push_back(&red);
  col_vec.push_back(&green);
  col_vec.push_back(&blue);
  col_vec.push_back(&cyan);
  col_vec.push_back(&magenta);
  col_vec.push_back(&orange);
#endif
  if (DEBUG) {
    i = 0;
    for (vector⟨Reg_Polygon *⟩::iterator iter = reg_polygons.begin(); iter ≠ reg_polygons.end();
          ++iter) {
#if 0
      (**iter).draw(*(col_vec[i]));
#endif
      ++i;
    }
    reg_polygons[0]→draw();
  }
  for (i = 0; i < 6; ++i) { reg_polygons.push_back ( create_new < Reg_Polygon > (0) ) ;
  *reg_polygons.back() = *reg_polygons[i];
  reg_polygons.back()→rotate(180); } pts[5] = reg_polygons[1]→get_point(4);
  pts[6] = reg_polygons[10]→get_point(2);
  if (DEBUG) {
    pts[5].dotlabel("$p_5$");
    pts[6].dotlabel("$p_6$");
  }
  pts[7] = pts[5] − pts[6];
  for (i = 6; i < 12; ++i) reg_polygons[i]→shift(pts[7]);
#if 0
  if (DEBUG) {
    for (i = 6; i < 12; ++i) reg_polygons[i]→draw(*(col_vec[i − 6]));
  }
#endif
  Point center_0 = reg_polygons[0]→get_center();
  Point center_6 = reg_polygons[6]→get_center();
  center = center_0.mediate(center_6);
  Transform t = center.shift(−center);
  if (angle_x ≠ 0 ∨ angle_y ≠ 0 ∨ angle_z ≠ 0) t.rotate(angle_x, angle_y, angle_z);
  if (p ≠ origin) center *= t.shift(p);
```

for (**vector**⟨**Reg_Polygon** ∗⟩ :: *iterator iter* = *reg_polygons.begin*( ); *iter* ≠ *reg_polygons.end*( ); ++*iter*)
  (∗∗*iter*) ∗= *t*;
if (DEBUG) *cout* ≪ "Exiting␣Dodecahedron::Dodecahedron().\n" ≪ *flush*;
return; }

1503. Get net. [LDF 2002.09.29.] Changed this function. TO DO. I've removed *rotate*( ) and now rotate the pentagons in the x-z plane only in order to avoid having the y-coordinates be off by small amounts. This might not happen with *rotate*( ) once I implement the routine in Salomon (!! Get reference!) for calculating sine and cosine using integers.

const Point &*center_0* is only ever used when *do_half* ≡ *false*, that is, when the net isn't just being generated for use by a constructor. If *get_net*( ) *is* called by a constructor, the net is always made with the center of pentagon 0 at the origin. Even if *center_0* is used, the net is always generated in a plane parallel to the x-z plane.

Log

[LDF 2002.08.12.] Removed *center_0* argument.

⟨ Declare **Dodecahedron** functions 1497 ⟩ +≡
  static vector⟨**Reg_Polygon** ∗⟩ *get_net*(**const real** *pentagon_diameter*, **bool** *do_half* = *false*);

1504.

⟨ Define **Dodecahedron** functions 1498 ⟩ +≡

```
vector⟨Reg_Polygon *⟩ Dodecahedron::get_net(const real pentagon_diameter, bool do_half){ int i;
    vector⟨Reg_Polygon *⟩ pents; for (i = 0; i < 6; ++i) pents.push_back ( create_new <
        Reg_Polygon > (0) ) ;
    pents[0]→set(origin, 5, pentagon_diameter, 0, 180);    /* The middle pentagon. */
    Point pts[10];
    for (i = 0; i < 5; i++) {
      pts[i] = pents[0]→get_point(i);
    }
    *pents[1] = *pents[0];
    pents[1]→rotate(0, 36);
    *pents[5] = *pents[4] = *pents[3] = *pents[2] = *pents[1];
    for (i = 5; i < 10; i++) pts[i] = pents[1]→get_point(i − 5);
    pents[1]→shift(pts[0] − pts[8]);
    pents[2]→shift(pts[0] − pts[6]);
    pents[3]→shift(pts[4] − pts[5]);
    pents[4]→shift(pts[3] − pts[9]);
    pents[5]→shift(pts[1] − pts[9]);
    if (do_half ≡ true)
      /* [LDF 2002.09.29.] We only need half of the net, if we're using it to make a polyhedron. In that
         case, we copy and transform the half we've already got and the copy on top of the first half. */
    {
      for (i = 0; i < 6; ++i) {
        for (int j = 0; j < 5; j++)
          if (pents[i]→get_point(j).get_y( ) ≠ 0) {
            cerr ≪ "ERROR!␣In␣Dodecahedron::get_net():\n" ≪ "y-coordinate␣!=␣0!\n" ≪
                "You'd␣better␣fix␣this!\n\n" ≪ flush;
          }
      }
      return pents;
    }
    for (i = 6; i < 12; i++) { pents.push_back ( create_new < Reg_Polygon > (0) ) ;
    *pents[i] = *pents[i − 6];
    pents[i]→rotate(0, 180); } pts[0] = pents[11]→get_point(0);
    pts[1] = pents[5]→get_point(1);
    for (i = 6; i < 12; ++i) pents[i]→shift(pts[1] − pts[0]);
    for (i = 0; i < 12; ++i) {
      for (int j = 0; j < 5; j++)
        if (pents[i]→get_point(j).get_y( ) ≠ 0) {
          cerr ≪ "ERROR!␣In␣Dodecahedron::get_net():\n" ≪ "y-coordinate␣!=␣0!\n" ≪
              "You'd␣better␣fix␣this!\n\n" ≪ flush;
        }
    }
    return pents; }
```

1505. Draw net. [LDF 2002.11.10.] This function is for drawing the net of a **Dodecahedron**. Normally, this will be done in order to make a cardboard model, which will require tabs for gluing the pentagons together. If no tabs are desired, passing *false* as the *make_tabs* argument will suppress the tabs.

[LDF 2002.11.10.] TO DO: The arrays **Point** *pts* and **Path** *p* have too many members. In working on this function, I ended up getting rid of some of the members of these arrays after I'd already used members following them. I should go through and reassign the numbers, so that no members are skipped.

[LDF 2002.11.10.] TO DO: Add the usual arguments for drawing and filling commands.

[LDF 2002.11.10.] TO DO: *portrait* doesn't work right. Fix it! !! KLUDGE: *portrait* is set to *false* at the beginning of this function and a warning is issued.

Log

[LDF 2002.11.10.] Added this function.

[LDF 2002.11.10.] Tried to get output in portrait format to work, but it doesn't yet.

⟨ Declare **Dodecahedron** functions 1497 ⟩ +≡

static void *draw_net*(**const real** *pentagon_diameter*, **bool** *portrait* = *true*, **bool** *make_tabs* = *true*);

1506.

```
⟨ Define Dodecahedron functions 1498 ⟩ +≡
  void Dodecahedron::draw_net(const real pentagon_diameter, bool portrait, bool make_tabs)
  {
    vector⟨Reg_Polygon *⟩ v = get_net(pentagon_diameter);
    for (vector⟨Reg_Polygon *⟩::iterator iter = v.begin(); iter ≠ v.end(); ++iter) {
      if (portrait) (**iter).rotate(0, 90);
#if 0
      (**iter).rotate(90);
#endif
    }
    Point p[32];
    int i;
    for (i = 0; i < 5; i++) {
      p[i] = v[3]→get_point(i);
#if 0
      p[i].dotlabel(i);
#endif
    }    /* START HERE. */
    i = 0;
    for (vector⟨Reg_Polygon *⟩::iterator iter = v.begin(); iter ≠ v.end(); ++iter) {
      (**iter).draw();
      (**iter).get_center().label(i++, "");
    }
    if (¬make_tabs) return;
    return;
  }
```

1507. Icosahedron.

1508. Icosahedron class definition.

```
⟨ Define class Icosahedron 1508 ⟩ ≡
  class Icosahedron : public Polyhedron {
  protected: static const real dihedral_angle;    /* In radians! */
    real triangle_radius;
  public: ⟨ Declare Icosahedron functions 1511 ⟩
  };
```

This code is used in sections 1534 and 1535.

1509. Define static const Icosahedron data members.

```
⟨ Define static const Icosahedron data members 1509 ⟩ ≡
  const real Icosahedron::dihedral_angle = PI − asin(2.0/3.0);
```

This code is used in section 1534.

1510. Constructors and setting functions.

1511. Default constructor. (No arguments.)

```
⟨ Declare Icosahedron functions 1511 ⟩ ≡
  Icosahedron();
```

See also sections 1514, 1516, and 1518.

This code is used in section 1508.

1512.

⟨ Define **Icosahedron** functions 1512 ⟩ ≡

```
Icosahedron::Icosahedron()
{
  on_free_store = false;   /* from Solid. */
  do_output = true;
  faces = 20;   /* from Solid_Faced. */
  vertices = 12;
  edges = 30;
  center = INVALID_POINT;   /* from Polyhedron. */
  number_of_polygon_types = 1;
  face_radius = edge_radius = vertex_radius = INVALID_REAL;
  triangle_radius = INVALID_REAL;
}
```

See also sections 1515, 1517, and 1519.
This code is used in section 1534.

1513. Center, diameter of triangle, and angles.

1514. Constructor. [LDF 2002.12.18.] !! START HERE. Must fix this constructor. There's a problem with rotation. I've already done so for the **Dodecahedron** constructor.

Log

[LDF 2002.10.16.] Defined this function.

⟨ Declare **Icosahedron** functions 1511 ⟩ +≡

```
Icosahedron(const Point &p, const real diameter_of_triangle, real angle_x = 0, real angle_y = 0, real angle_z = 0);
```

1515.

⟨ Define **Icosahedron** functions 1512 ⟩ +≡

```
Icosahedron::Icosahedron(const Point &p, const real triangle_diameter, real angle_x, real
        angle_y, real angle_z){ bool DEBUG = false;    /* true */
    if (DEBUG) cout << "Entering␣Icosahedron::Icosahedron().\n" << flush;
    on_free_store = false;    /* from Solid. */
    do_output = true;
    faces = 20;    /* from Solid_Faced. */
    vertices = 12;
    edges = 30;
    number_of_polygon_types = 1;
#if 0    /* START HERE. TO DO: Must calculate these! */
    face_radius = 0;
    edge_radius = 0;
    vertex_radius = 0;
#endif
    triangle_radius = triangle_diameter/2.0;
    int i; reg_polygons.push_back ( create_new < Reg_Polygon > (0) ) ;
    reg_polygons.front()→set(origin, 3, triangle_diameter);
    Point pts[7];
    for (i = 0; i < 3; ++i) pts[i] = reg_polygons.front()→get_point(i);
    for (i = 1; i < 6; ++i) { reg_polygons.push_back ( create_new <
        Reg_Polygon > (reg_polygons.front()) ) ; } reg_polygons[1]→shift(pts[2] − pts[1]);
    reg_polygons[2]→shift(pts[1] − pts[2]);
    reg_polygons[3]→shift(pts[0] − pts[2]);
    reg_polygons[4]→shift(pts[0] − pts[1]);
    if (DEBUG) origin.label("0", "");
    reg_polygons[5]→rotate(0, 180);
    reg_polygons[5]→shift(pts[1] − reg_polygons[5]→get_point(0)); for (i = 6; i < 10; ++i) {
        reg_polygons.push_back ( create_new < Reg_Polygon > (reg_polygons[5]) ) ; }
        reg_polygons[6]→shift(pts[2] − pts[1]);
    reg_polygons[7]→shift(pts[1] − pts[0]);
    reg_polygons[8]→shift(pts[2] − pts[0]);
    reg_polygons[9]→shift(pts[2] − pts[0]);
    reg_polygons[9]→shift(pts[2] − pts[1]);
    pts[3] = reg_polygons[2]→get_point(1);
    pts[4] = reg_polygons[2]→get_point(0);
    pts[5] = reg_polygons[1]→get_point(2);
    pts[6] = reg_polygons[1]→get_point(0);
    real angle = 180.0 − (dihedral_angle * 180.0/PI);
    *reg_polygons[9] *= *reg_polygons[8] *= reg_polygons[7]→rotate(pts[3], pts[5], −angle);
    *reg_polygons[3] *= reg_polygons[4]→rotate(pts[4], pts[6], angle);
    *reg_polygons[2] *= reg_polygons[7]→rotate(pts[1], pts[4], −angle);
    *reg_polygons[7] *= *reg_polygons[2] *= *reg_polygons[3] *= reg_polygons[5]→rotate(pts[1], pts[0],
        −angle);
    *reg_polygons[1] *= reg_polygons[9]→rotate(pts[2], pts[6], angle);
    *reg_polygons[9] *= *reg_polygons[1] *= *reg_polygons[6] *= reg_polygons[4]→rotate(pts[2],
        pts[0], angle); for (i = 10; i < 20; ++i) { reg_polygons.push_back ( create_new <
        Reg_Polygon > (reg_polygons[i − 10]) ) ;
    reg_polygons.back()→rotate(180);
```

```
    reg_polygons.back()→shift(0, 12); }
    if (DEBUG) {
      reg_polygons[4]→dotlabel();
      reg_polygons[19]→dotlabel();
    }
    real y_shift = reg_polygons[4]→get_point(0).get_y() − reg_polygons[19]→get_point(0).get_y();
    for (i = 10; i < 20; ++i) reg_polygons[i]→shift(0, y_shift);
    center = reg_polygons[0]→get_center().mediate(reg_polygons[10]→get_center());
    for (i = 0; i < 20; ++i) reg_polygons[i]→shift(−center);
    center = origin;
    if (angle_x ≠ 0 ∨ angle_y ≠ 0 ∨ angle_z ≠ 0)
      for (vector⟨Reg_Polygon *⟩::iterator iter = reg_polygons.begin(); iter ≠ reg_polygons.end();
            ++iter) (**iter).rotate(angle_x, angle_y, angle_z);
    if (p ≠ origin) {
      for (vector⟨Reg_Polygon *⟩::iterator iter = reg_polygons.begin(); iter ≠ reg_polygons.end();
            ++iter) {
        (**iter).shift(p − center);
      }
      center = p;
    }
    if (DEBUG) {
      cout ≪ "Exiting␣Icosahedron::Icosahedron().\n" ≪ flush;
    }
    return; }
```

1516. Get net.

Log

[LDF 2002.11.10.] BUG FIX: Removed erroneous code that created 6 extra triangles.

[LDF 2002.08.12.] Removed *center_0* argument.

[LDF 2003.08.27.] Added **size_t** *triangles_size* and added **size_t** *i* to a **for** loop, where it's compared to *triangles_size*. This occurs in debugging code.

⟨ Declare **Icosahedron** functions 1511 ⟩ +≡

```
  static vector⟨Reg_Polygon *⟩ get_net(const real triangle_diameter, bool do_half = false);
```

1517.

⟨ Define **Icosahedron** functions 1512 ⟩ +≡

```
vector<Reg_Polygon *> Icosahedron::get_net(const real triangle_diameter, bool do_half){ bool
        DEBUG = false;     /* true */
    vector<Reg_Polygon *> triangles;
    int i; for (i = 0; i < 10; ++i) triangles.push_back ( create_new < Reg_Polygon > (0) ) ;
      /* The bottom left triangle. */
    triangles[0]->set(origin, 3, triangle_diameter, 0, 180);
    Point pts[4];
    pts[0] = triangles[0]->get_point(0);
    pts[1] = triangles[0]->get_point(1);
    pts[2] = triangles[0]->get_point(2);
    *triangles[1] = *triangles[0];
    triangles[1]->rotate(0, 180);
    triangles[1]->shift(pts[2] - triangles[1]->get_point(1));
    pts[3] = triangles[1]->get_point(0);
    *triangles[2] = *triangles[0];
    triangles[2]->shift(pts[3] - pts[2]);
    *triangles[3] = *triangles[1];
    triangles[3]->shift(pts[3] - pts[2]);
    for (i = 0; i < 4; ++i) {
      *triangles[4 + i] = *triangles[i];
      triangles[4 + i]->shift(pts[1] - pts[2]);
    }
    if (DEBUG)
      for (i = 0; i < 4; ++i) pts[i].dotlabel(i);
    if (do_half ≡ true) {
      *triangles[8] = *triangles[4];
      *triangles[9] = *triangles[5];
      *triangles[8] *= triangles[9]->shift(pts[1] - pts[2]);
      for (i = 0; i < 10; ++i) {
        for (int j = 0; j < 3; j++)
          if (triangles[i]->get_point(j).get_y() ≠ 0) {
            cerr << "ERROR!␣In␣Icosahedron::get_net():\n" << "y-coordinate␣!=␣0!\n" <<
                "You'd␣better␣fix␣this!\n\n" << flush;
          }
      }
      return triangles;
    }     /* Do the second half. */
    if (DEBUG) cout << "Doing␣the␣second␣half.\n";
    for (i = 0; i < 10; ++i) triangles.push_back ( create_new < Reg_Polygon > (0) ) ;
    for (int j = 8; j ≤ 16; j += 4)
      for (i = 0; i < 4; ++i) {
        *triangles[j + i] = *triangles[j - 4 + i];
        triangles[j + i]->shift(pts[1] - pts[2]);
      }
    if (DEBUG) {
      size_t triangles_size = triangles.size();
      for (size_t i = 0; i < triangles_size; ++i)
        if (triangles[i]->size() > 0) triangles[i]->get_center().label(i, "");
    }
```

```
for (i = 0; i < 20; ++i) {
  for (int j = 0; j < 3; j++)
    if (triangles[i]→get_point(j).get_y() ≠ 0) {
      cerr ≪ "ERROR!␣In␣Icosahedron::get_net():\n" ≪ "y-coordinate␣!=␣0!\n" ≪
          "You'd␣better␣fix␣this!\n\n" ≪ flush;
    }
}
return triangles; }
```

1518. Draw net. TO DO: Add parallel projections onto planes other than the x-y plane.

Log

[LDF 2002.11.10.] Added this function. *portrait* works, unlike **Dodecahedron**::*draw_net*()*polyhed.web* (as of this date).

[LDF 2002.08.12.] Changed, so that net is drawn in x-z plane.

[LDF 2002.08.12.] This function now returns before the code for making the tabs can be executed, because it doesn't work yet. TO DO: Write code for tabs.

⟨ Declare **Icosahedron** functions 1511 ⟩ +≡

```
static void draw_net(const real triangle_diameter, bool portrait = true, bool make_tabs = true);
```

1519.

⟨ Define **Icosahedron** functions 1512 ⟩ +≡

```
void Icosahedron::draw_net(const real triangle_diameter, bool portrait, bool make_tabs)
{
    vector⟨Reg_Polygon *⟩ v = get_net(triangle_diameter);
    int i = 0;
    for (vector⟨Reg_Polygon *⟩::iterator iter = v.begin(); iter ≠ v.end(); ++iter) {
        if (portrait) (**iter).rotate(0, 90);
        (**iter).get_center().label(i++, "");
        (**iter).draw();
    }
    return;    /* Delete, when I start writing code for tabs. [LDF 2002.08.12.] */
    if (¬make_tabs) return;
    Path p[11];
    Point pts[11];
#if 0
    v[0]→dotlabel();
#endif
    pts[0] = v[0]→get_point(0);
    pts[1] = v[0]→get_point(1);
    pts[2] = v[0]→get_point(2);
    pts[3] = pts[0].mediate(pts[1], .1);
    pts[4] = pts[1].mediate(pts[0], .1);
#if 0
    pts[3].dotlabel(2);
    pts[4].dotlabel(3);
#endif
    pts[5] = pts[0].mediate(pts[3], .1);
    pts[6] = pts[1].mediate(pts[4], .1);
    pts[7] = pts[3];
    pts[7].shift(0, 0, 1);
    pts[8] = pts[4];
    pts[8].shift(0, 0, 1);
    pts[5].rotate(pts[3], pts[7], 90);
    pts[6].rotate(pts[8], pts[4], 90);
#if 0
    pts[5].dotlabel(4);
    pts[6].dotlabel(5);
#endif
    pts[9] = pts[5].mediate(pts[6], .1);
    pts[10] = pts[6].mediate(pts[5], .1);
#if 0
    pts[9].dotlabel(8);
    pts[10].dotlabel(9);
#endif
    p[0].set("--", true, &pts[3], &pts[9], &pts[10], &pts[4], 0);
    p[1] = p[2] = p[3] = p[4] = p[5] = p[0];
    p[1].shift(v[4]→get_point(1) − pts[1]);
    p[2].shift(v[8]→get_point(1) − pts[1]);
    p[3].shift(v[12]→get_point(1) − pts[1]);
    p[4].shift(v[16]→get_point(1) − pts[1]);
```

```
    p[5].shift(v[18]→get_point(1) − pts[1]);
    p[6] = p[0];
    p[6].rotate(0, 0, 240);
    p[6].rotate(pts[0], pts[2]);
#if 0
    v[3]→dotlabel();
#endif
    p[7] = p[8] = p[9] = p[10] = p[6];
    p[6].shift(v[3]→get_point(0) − pts[2]);
    p[7].shift(v[7]→get_point(0) − pts[2]);
    p[8].shift(v[11]→get_point(0) − pts[2]);
    p[9].shift(v[15]→get_point(0) − pts[2]);
    p[10].shift(v[19]→get_point(0) − pts[2]);
    for (i = 0; i < 11; i++) p[i].draw();
    return;
  }
```

1520. Semi-Regular Archimedean Polyhedra.

1521. Truncated Octahedron.

1522. Trunc_Octahedron class definition.

⟨ Define **class Trunc_Octahedron** 1522 ⟩ ≡

```
  class Trunc_Octahedron : public Polyhedron {
  protected: static const real angle_hex_square;   /* In radians! */
    static const real angle_hex_hex;   /* In radians! */
    real hexagon_radius;
  public: ⟨ Declare Trunc_Octahedron functions 1525 ⟩
  };
```

This code is used in sections 1534 and 1535.

1523. Define static const Trunc_Octahedron data members.

⟨ Define **static const Trunc_Octahedron** data members 1523 ⟩ ≡

```
  const real Trunc_Octahedron::angle_hex_square = (125 + (16.0/60.0)) * (PI/180.0);
  const real Trunc_Octahedron::angle_hex_hex = (109 + (28.0/60.0)) * (PI/180.0);
```

This code is used in section 1534.

1524. Constructors and setting functions.

Log

[LDF 2003.04.15.] Commented-out the constructors and *get_net*( ). They made use of the fact, which is no longer true, that **Rectangle** was formerly derived from **Reg_Polygon**. Now that **Rectangle** is derived from **Path**, some of the code in these functions doesn't work. TO DO: Fix these functions!

1525. Default constructor. (No arguments.)

⟨ Declare **Trunc_Octahedron** functions 1525 ⟩ ≡

```
  Trunc_Octahedron();
```

See also sections 1528 and 1530.

This code is used in section 1522.

1526.

⟨ Define **Trunc_Octahedron** functions 1526 ⟩ ≡

Trunc_Octahedron :: **Trunc_Octahedron**( )
{
on_free_store = *false*; /* from **Solid**. */
do_output = *true*;
faces = 14; /* from **Solid_Faced**. [LDF 2002.10.29.] Truncated octahedrons consist of 6 squares and 8 hexagons. */
vertices = 24;
edges = 36;
center = `INVALID_POINT`; /* from **Polyhedron**. */
number_of_polygon_types = 2;
face_radius = *edge_radius* = *vertex_radius* = `INVALID_REAL`;
hexagon_radius = `INVALID_REAL`;
}

See also sections 1529, 1531, and 1532.

This code is used in section 1534.

1527. Center, diameter of hexagon, and angles.

1528. Constructor.

Log

[LDF 2002.11.08.] Added this function.

⟨ Declare **Trunc_Octahedron** functions 1525 ⟩ +≡

#**if** 0
Trunc_Octahedron(**const Point** &*p*, **const real** *diameter_of_hexagon*, **real** *angle_x* = 0, **real** *angle_y* = 0, **real** *angle_z* = 0);
#**endif**

1529.

```
⟨ Define Trunc_Octahedron functions 1526 ⟩ +≡
#if 0
  Trunc_Octahedron::Trunc_Octahedron(const Point &p, const real hexagon_diameter, real
        angle_x, real angle_y, real angle_z){ bool DEBUG = false;   /* true */
    if (DEBUG) cout ≪ "Entering␣Trunc_Octahedron::Trunc_Octahedron().\n" ≪ flush;
    center = p;
    on_free_store = false;   /* from Solid. */
    do_output = true;   /* START HERE. TO DO: Must calculate these! */
    face_radius = 0;
    edge_radius = 0;
    vertex_radius = 0;
    faces = 14;   /* from Solid_Faced. [LDF 2002.10.29.] Truncated octahedrons consist of 6 squares
        and 8 hexagons. */
    vertices = 24;
    edges = 36;
    number_of_polygon_types = 2;
    hexagon_radius = hexagon_diameter/2.0;
    reg_polygons = get_net(hexagon_diameter, true);
    Point pts[24];
    int i;
    for (i = 0; i < 6; i++) pts[i] = reg_polygons[0]→get_point(i);
#if 0
    pts[0].dotlabel(0);
    pts[1].dotlabel(1, "lft");
    pts[2].dotlabel(2, "bot");
    pts[3].dotlabel(3, "bot");
    pts[4].dotlabel(4, "rt");
    pts[5].dotlabel(5);
#endif
    real angle_h_h = 180.0 − (angle_hex_hex ∗ 180.0/PI);
    reg_polygons[1]→rotate(pts[4], pts[5], angle_h_h);
    reg_polygons[2]→rotate(pts[3], pts[2], −angle_h_h);
    reg_polygons[3]→rotate(pts[0], pts[1], angle_h_h);
    real angle_h_s = 180.0 − (angle_hex_square ∗ 180.0/PI);
    reg_polygons[4]→rotate(pts[0], pts[5], angle_h_s);
    reg_polygons[5]→rotate(pts[1], pts[2], angle_h_s);
    reg_polygons[6]→rotate(pts[3], pts[4], angle_h_s);
      /* [LDF 2002.11.07.] Do something about these comments! */   /* 7 */
    reg_polygons.push_back ( create_new < Reg_Polygon > (reg_polygons[0]) ) ;   /* 8 */
    reg_polygons.push_back ( create_new < Reg_Polygon > (reg_polygons[1]) ) ;   /* 9 */
    reg_polygons.push_back ( create_new < Reg_Polygon > (reg_polygons[2]) ) ;   /* 10 */
    reg_polygons.push_back ( create_new < Reg_Polygon > (reg_polygons[3]) ) ;   /* 11 */
    reg_polygons.push_back ( create_new < Reg_Polygon > (reg_polygons[4]) ) ;   /* 12 */
    reg_polygons.push_back ( create_new < Reg_Polygon > (reg_polygons[5]) ) ;   /* 13 */
    reg_polygons.push_back ( create_new < Reg_Polygon > (reg_polygons[6]) ) ;
    Transform t;
    t.rotate(180);
    t.shift(0, 15);
    for (i = 7; i < 14; i++) ∗reg_polygons[i] ∗= t;
```

```
    t.reset();
    t.shift(reg_polygons[2]→get_point(2) − reg_polygons[11]→get_point(3));
    for (i = 7; i < 14; i++) *reg_polygons[i] *= t;
    i = 0;
    for (vector⟨Reg_Polygon *⟩::iterator iter = reg_polygons.begin(); iter ≠ reg_polygons.end();
            ++iter) {
      if (center ≠ origin) (**iter).shift(center);
    }
    if (DEBUG) {
      cout ≪ "Exiting␣Trunc_Octahedron::Trunc_Octahedron()." ≪ endl ≪ endl ≪ flush;
    }
    return; }
#endif
```

1530. Get net.

Log

[LDF 2002.11.08.] Added this function.
[LDF 2002.08.12.] Removed *center_0* argument.

⟨ Declare **Trunc_Octahedron** functions 1525 ⟩ +≡

```
#if 0
  static vector⟨Reg_Polygon *⟩ get_net(const real hexagon_diameter, bool do_half = false);
#endif
```

1531.

⟨ Define **Trunc_Octahedron** functions 1526 ⟩ +≡

```
#if 0
  vector⟨Reg_Polygon *⟩ Trunc_Octahedron::get_net(const real hexagon_diameter, bool do_half){
        bool DEBUG = false;    /* true */
     vector⟨Reg_Polygon *⟩ reg_polygons;
     int i; reg_polygons.push_back ( create_new < Reg_Polygon > (0) ) ;
     reg_polygons[0]→set(origin, 6, hexagon_diameter);
     Point pts[24];
     for (i = 0; i < 6; i++) pts[i] = reg_polygons[0]→get_point(i);
     reg_polygons.push_back ( create_new < Reg_Polygon > (reg_polygons[0]) ) ;
     reg_polygons[1]→shift(pts[4] − pts[2]); reg_polygons.push_back ( create_new <
        Reg_Polygon > (reg_polygons[0]) ) ;
     reg_polygons[2]→shift(pts[3] − pts[5]); reg_polygons.push_back ( create_new <
        Reg_Polygon > (reg_polygons[0]) ) ;
     reg_polygons[3]→shift(pts[1] − pts[3]);
     real side_length = pts[5].get_x( ) − pts[0].get_x( );
     pts[6] = pts[0];
     pts[6].shift(0, 0, side_length);
     pts[7] = pts[5];
     pts[7].shift(0, 0, side_length);
     Rectangle r(pts[0], pts[5], pts[7], pts[6]); reg_polygons.push_back ( create_new < Reg_Polygon > (r)
        ) ;
     pts[8] = pts[2];
     pts[9] = pts[1];
     pts[9].shift(0, 1);
     pts[8].rotate(pts[9], pts[1], −90);
     pts[10] = pts[8];
     pts[10].shift(pts[2] − pts[1]);
     r.set(pts[10], pts[2], pts[1], pts[8]); reg_polygons.push_back ( create_new < Reg_Polygon > (r) ) ;
     pts[11] = pts[3];
     pts[12] = pts[4];
     pts[12].shift(0, 1);
     pts[11].rotate(pts[4], pts[12], 90);
     pts[13] = pts[11];
     pts[13].shift(pts[3] − pts[4]);
     r.set(pts[11], pts[4], pts[3], pts[13]); reg_polygons.push_back ( create_new < Reg_Polygon > (r) ) ;
     if (do_half) return reg_polygons;
#endif
```

1532. [LDF 2002.11.08.] If we just want the net, *reg_polygons*[5] and *reg_polygons*[6] must be changed, because I've made the net a bit differently from the way it's done in Cundy. Get reference!! Page 104. I made two of the squares slanted, in order to avoid having to rotate them twice.

⟨ Define **Trunc_Octahedron** functions 1526 ⟩ +≡

```
#if 0
  *reg_polygons[6] = *reg_polygons[5] = *reg_polygons[4];
  pts[14] = reg_polygons[4]->get_point(2);
  pts[15] = reg_polygons[4]->get_point(3);
  reg_polygons[5]->shift(pts[1] - pts[14]);
  reg_polygons[6]->shift(pts[4] - pts[15]); reg_polygons.push_back ( create_new <
      Reg_Polygon > (reg_polygons[0]) ) ;    /* 7 */
  reg_polygons.push_back ( create_new < Reg_Polygon > (reg_polygons[4]) ) ;    /* 8 */
  *reg_polygons[7] *= reg_polygons[8]->shift(pts[1] + pts[2] - pts[3] - pts[4]); reg_polygons.push_back (
      create_new < Reg_Polygon > (reg_polygons[0]) ) ;    /* 9 */
  reg_polygons[9]->shift(pts[4] + pts[0] - (2 * pts[2])); reg_polygons.push_back ( create_new <
      Reg_Polygon > (reg_polygons[4]) ) ;    /* 10 */
  reg_polygons.push_back ( create_new < Reg_Polygon > (reg_polygons[0]) ) ;    /* 11 */
  reg_polygons.push_back ( create_new < Reg_Polygon > (reg_polygons[1]) ) ;    /* 12 */
  reg_polygons.push_back ( create_new < Reg_Polygon > (reg_polygons[6]) ) ;    /* 13 */
  *reg_polygons[10] *= *reg_polygons[11] *= *reg_polygons[12] *=
      reg_polygons[13]->shift(pts[4] + pts[3] - pts[2] - pts[1]);
  return reg_polygons; }
#endif
```

1533. Putting polyhedra together.

1534. This is what's compiled.

⟨ Include files 6 ⟩
⟨ Version control identifier 5 ⟩
⟨ Define **class Polyhedron** 1472 ⟩
⟨ Define **class Tetrahedron** 1477 ⟩
⟨ Define **static const Tetrahedron** data members 1478 ⟩
⟨ Define **class Dodecahedron** 1494 ⟩
⟨ Define **static const Dodecahedron** data members 1495 ⟩
⟨ Define **class Icosahedron** 1508 ⟩
⟨ Define **static const Icosahedron** data members 1509 ⟩
⟨ Define **class Trunc_Octahedron** 1522 ⟩
⟨ Define **static const Trunc_Octahedron** data members 1523 ⟩
⟨ Define **Polyhedron** functions 1474 ⟩
⟨ Define **Tetrahedron** functions 1481 ⟩
⟨ Define **Dodecahedron** functions 1498 ⟩
⟨ Define **Icosahedron** functions 1512 ⟩
⟨ Define **Trunc_Octahedron** functions 1526 ⟩

1535. This is what's written to `polyhed.h`.

⟨ `polyhed.h` 1535 ⟩ ≡
⟨ Define **class Polyhedron** 1472 ⟩
⟨ Define **class Tetrahedron** 1477 ⟩
⟨ Define **class Dodecahedron** 1494 ⟩
⟨ Define **class Icosahedron** 1508 ⟩
⟨ Define **class Trunc_Octahedron** 1522 ⟩

1536. Parsing (`parser.web`).

Log

Removed the code from this file. I plan to use Bison for making the parser. [LDF 2003.08.25.]

[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* = `"$Id:␣parser.web,v␣1.4␣2004/01/12␣21:30:44␣lfinsto1␣Exp␣$"`;

1537. Include files. `map.h` is currently not needed, but I plan to use it for the input routine. [LDF 2004.01.06.]

⟨ Include files 6 ⟩ +≡
```
#include "loader.h"
#include "pspglb.h"
#include "io.h"
#include "colors.h"
#include "transfor.h"
#include "shapes.h"
#include "pictures.h"
#include "points.h"
#include "lines.h"
#include "planes.h"
#include "paths.h"
#include "curves.h"
#include "polygons.h"
#include "rectangs.h"
#include "ellipses.h"
#include "circles.h"
#include "patterns.h"
#include "solids.h"
#include "solfaced.h"
#include "cuboid.h"
#include "polyhed.h"
#include "utility.h"
```

1538. Parse.

⟨ Declare parser functions 1538 ⟩ ≡
This code is used in section 1542.

1539.

⟨ Define parser functions 1539 ⟩ ≡

This code is used in section 1541.

1540. Putting the parser together.

1541. This is what's compiled.

⟨ Include files 6 ⟩
⟨ Version control identifier 5 ⟩
⟨ Define parser functions 1539 ⟩

1542. This is what's written to `parser.h`.

⟨ `parser.h` 1542 ⟩ ≡
  ⟨ Declare parser functions 1538 ⟩

1543. Main (`main.web`).

Log

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

[LDF 2003.08.29.] Moved `getopt.h` from `loader.web` to here, because it's only used here. TO DO: Move the rest of the include commands to the files where they're needed, **and get rid of** `loader.web`.

[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 2003.12.01.] Put the version control identifiers back into the release versions, because I've put them in their own RCS repository.

⟨ Version control identifier 5 ⟩ +≡
  static string *rcs_id* = "$Id:␣main.web,v␣1.12␣2004/01/12␣21:30:34␣lfinsto1␣Exp␣$";

1544. Include files. `getopt.h` is included for processing the command line options. [LDF 2003.08.14.]

⟨ Include files 6 ⟩ +≡
```
#include "loader.h"
#include <bitset>
#ifdef __GNUC__
#include <getopt.h>
#endif
#include "pspglb.h"
#include "creatnew.h"
#include "gsltmplt.h"
#include "io.h"
#include "colors.h"
#include "transfor.h"
#include "shapes.h"
#include "pictures.h"
#include "points.h"
#include "lines.h"
#include "planes.h"
#include "paths.h"
#include "curves.h"
#include "polygons.h"
#include "rectangs.h"
#include "ellipses.h"
#include "circles.h"
#include "patterns.h"
#include "solids.h"
#include "solfaced.h"
#include "cuboid.h"
#include "polyhed.h"
#include "utility.h"
#include "parser.h"
#include "examples.h"
```

1545. Get input. I plan to use Flex and Bison to handle scanning and parsing input. I haven't started work on this yet. [LDF 2003.08.20.]

⟨ Get input 1545 ⟩ ≡

1546. Actions in main.

⟨ Actions in main 1546 ⟩ ≡

```
using namespace Colors;
using namespace Projections; if (ldf_real_float) MAX_REAL = System::get_second_largest <
    float > (FLT_MAX, false); else if (ldf_real_double) MAX_REAL = System::get_second_largest <
    double > (DBL_MAX, false);
MAX_REAL_SQRT = sqrt(MAX_REAL);
vector<const Color *> v;
v.push_back(&red);
v.push_back(&green);
v.push_back(&blue);
v.push_back(&cyan);
v.push_back(&yellow);
#if 0
v.push_back(&magenta);
v.push_back(&orange);
v.push_back(&violet);
v.push_back(&yellow_green);
v.push_back(&green_yellow);
v.push_back(&blue_violet);
v.push_back(&red);
v.push_back(&green);
v.push_back(&blue);
v.push_back(&cyan);
v.push_back(&yellow);
v.push_back(&magenta);
v.push_back(&orange);
v.push_back(&violet);
v.push_back(&yellow_green);
v.push_back(&green_yellow);
v.push_back(&blue_violet);
v.push_back(&violet_red);
#endif
```

See also section 1547.

This code is used in section 1557.

1547. Your code here!

⟨ Actions in main 1546 ⟩ +≡
beginfig(1);
Point p;
Point $q(1, 1, 1)$;
Circle $c(p, 2, 90)$;
c.filldraw( );
p.draw(q);
current_picture.output( );
endfig( );

1548. Process command line options. This section includes one of (currently) two other sections, one for the GCC/Linux version and one for the DEC version. The section to be included is chosen by testing whether preprocessor macros are defined or not. Put another way, the command line option processing code is conditionally compiled. [LDF 2003.08.14.]

The problem is that, unlike GCC, the DEC C++ compiler doesn't support long command line options, So I have to implement the command line option processing code separately for each version. [LDF 2003.08.14.]

Log

[LDF 2003.08.14.] Added this section.

⟨ Process command line options 1548 ⟩ ≡
#ifdef `__GNUC__`
⟨ GCC command line option processing 1549 ⟩
#else
#ifdef `__DECCXX`
⟨ DEC command line option processing 1551 ⟩
#endif
#endif

This code is used in section 1555.

1549. GCC version of command line processing.

Log

[LDF 2003.08.14.] Added this section.
[LDF 2003.08.14.] Added code for handling the "`--silent`" option, including the constant `SILENT_INDEX`.

⟨ GCC command line option processing 1549 ⟩ ≡

```
{
  bool DEBUG = false;    /* true */
  int option_ctr;
  int digit_optind = 0;
  const unsigned short HELP_INDEX = 0;
  const unsigned short SILENT_INDEX = 1;
  const unsigned short VERBOSE_INDEX = 2;
  const unsigned short VERSION_INDEX = 3;
  static struct option long_options[] = {{"help", 0, 0, 0}, {"silent", 0, 0, 0}, {"verbose", 0, 0, 0},
      {"version", 0, 0, 0}, {0, 0, 0, 0}};
  int option_index = 0;
  int this_option_optind = optind ? optind : 1;
  while (1) {
    option_ctr = getopt_long_only(argc, argv, "hv", long_options, &option_index);
    if (DEBUG) {
      cout << "option_ctr == " << option_ctr << endl << flush;
      cout << "option_index == " << option_index << endl << flush;
      cout << "optarg == " << optarg << endl << flush;
    }
    if (option_ctr == -1) {
      if (DEBUG) cout << "No more options." << endl << endl << flush;
      break;
    }
    if (option_ctr == 0) {
      if (DEBUG) {
        cout << "option " << long_options[option_index].name;
        if (optarg) cout << " with arg " << optarg;
        cout << endl;
      }
      if (option_index == HELP_INDEX) {
        cout << "3DLDF Version " << VERSION_3DLDF << ". " << COPYRIGHT_3DLDF <<
            endl << "Valid options for 3DLDF are:" << endl <<
            "--help:      Prints this message and exits " << "with return value 0." <<
            endl << endl << "--silent:    Suppresses some output to standard output" <<
            endl << "             and standard error when 3dldf is run." << endl <<
            endl << "--verbose:   Causes status information to be printed " <<
            "to standard output" << endl << "             when 3dldf is run." <<
            endl << endl << "--version:   Prints the version number of 3DLDF" << endl <<
            "             to standard output and " << "exits with return value 0." <<
            endl << endl << flush;
        if (DEBUG) cout << "Exiting with return value 0." << endl << flush;
        exit(0);
      }
      else if (option_index == SILENT_INDEX) {
        if (DEBUG) cout << "Setting SILENT_GLOBAL to true." << endl;
```

```
        SILENT_GLOBAL = true;
      }
      else if (option_index ≡ VERBOSE_INDEX) {
        if (DEBUG) cout ≪ "Setting␣VERBOSE_GLOBAL␣to␣true." ≪ endl;
        VERBOSE_GLOBAL = true;
      }
      else if (option_index ≡ VERSION_INDEX) {
        cout ≪ "3DLDF␣Version␣" ≪ VERSION_3DLDF ≪ ".␣" ≪ COPYRIGHT_3DLDF ≪ endl ≪ flush;
        if (DEBUG) cout ≪ "Exiting␣with␣return␣value␣0." ≪ endl ≪ flush;
        exit(0);
      }
      else {
        cerr ≪ "This␣can't␣happen!␣" ≪ "option_index␣has␣invalid␣value:␣" ≪ option_index ≪
            endl ≪ "Will␣try␣to␣continue." ≪ endl ≪ endl ≪ flush;
      }
    }
    else if (option_ctr ≡ '?') {
      cerr ≪ "getopt_long()␣returned␣ambiguous␣match.␣Breaking." ≪ endl ≪ endl ≪ flush;
      break;
    }
    else {
      cerr ≪ "getopt_long()␣returned␣invalid␣option." ≪ endl ≪ flush;
    }
    if (DEBUG) cout ≪ "********************\n\n";
  }   /* while */
  if (optind < argc) {
    cout ≪ "non-option␣ARGV-elements:␣";
    while (optind < argc) cout ≪ argv[optind++] ≪ endl;
    cout ≪ endl ≪ flush;
  }
  if (DEBUG) {
    cout ≪ "Exiting.␣(Debugging␣command␣line␣option␣processing.)" ≪ endl ≪ endl ≪ flush;
    exit(0);
  }
}   /* End of group. */
```

This code is used in section 1548.

1550. DEC version.

1551. This section doesn't contain any code yet.

Log

[LDF 2003.08.14.] Added this section.

⟨ DEC command line option processing 1551 ⟩ ≡ /* Do nothing. */

This code is used in section 1548.

1552. Print version, copyright, and license information. The version, copyright, and license information is printed to standard output when 3DLDF is run, unless the "`--silent`" option was used. The code for this differs for the GCC 2.95/Linux version on the one hand, and the other versions (currently, GCC 3.3/Linux and DEC) on the other. The reason for this is, that GCC 2.95 doesn't handle stream formatting in the same way as the others. I assume that the others adhere to the standard and that GCC 2.95 doesn't, but I haven't checked this. At any rate, the non-GCC 2.95 version corresponds to what Stroustrup describes in *The C++ Programming Language*.

Log

[LDF 2003.08.14.] Added this section.

```
⟨ Print version, copyright, and license information 1552 ⟩ ≡
  if (¬SILENT_GLOBAL) {
#ifdef LDF_GCC_2_95
    ⟨ GCC 2.95 print version, copyright, and license information 1553 ⟩
#else
    ⟨ GCC 3.3 and DEC print version, copyright, and license information 1554 ⟩
#endif
  }
```

This code is used in section 1556.

1553. GCC 2.95 version.

Log

[LDF 2003.08.14.] Added this section.

```
⟨ GCC 2.95 print version, copyright, and license information 1553 ⟩ ≡
  cout.setf(ios::fixed, ios::floatfield);
  cout.precision(1);
  cout ≪ "3DLDF␣Version␣" ≪ VERSION_3DLDF ≪ "." ≪ endl ≪ COPYRIGHT_3DLDF ≪ endl ≪
      DISCLAIMER_3DLDF ≪ endl ≪ endl ≪ flush;
  cout.setf(ios::fmtflags(0), ios::floatfield);    /* Reset to defaults. [LDF 2003.08.14.] */
  cout.precision(6);
```

This code is used in section 1552.

1554. GCC 3.3 and DEC version.

Log

[LDF 2003.08.14.] Added this section.

⟨ GCC 3.3 and DEC print version, copyright, and license information 1554 ⟩ ≡
cout.precision(1);
cout ≪ "3DLDF␣Version␣" ≪ VERSION_3DLDF ≪ "." ≪ *endl* ≪ COPYRIGHT_3DLDF ≪ *endl* ≪ DISCLAIMER_3DLDF ≪ *endl* ≪ *endl* ≪ *flush*;
cout.precision(6);

This code is used in section 1552.

1555. Main itself.

⟨ Main 1555 ⟩ ≡
int *main*(**int** *argc*, **char** ∗*argv*[ ]){ ⟨ Process command line options 1548 ⟩;

See also sections 1556 and 1557.

This code is used in section 1558.

1556.

Log

[LDF 2003.08.29.] Changed the string that's passed to *initialize_io*( ) as the name of the input file. It's a dummy name, since I've changed *initialize_io*( ) today, so that no *in_stream* isn't opened. Also, commented-out the line where *in_stream* is closed.

[LDF 2003.11.28.] Removed TEX text above, that referred to the precision used when printing out VERSION_3DLDF. This is no longer relevant, because VERSION_3DLDF is now a **string** rather than a **real**.

⟨ Main 1555 ⟩ +≡
⟨ Print version, copyright, and license information 1552 ⟩;

1557.

```
⟨ Main 1555 ⟩ +≡
  initialize_io("3DLDFin.ldf", "3DLDFput.mp", "3DLDFput.tex", argv[0]);
  Color::initialize_colors();
  ⟨ Actions in main 1546 ⟩;
  write_footers();
#if 0
  in_stream.close();
#endif
  out_stream.close();
  tex_stream.close();
  if (¬SILENT_GLOBAL) {
    cout ≪ "Exiting␣3DLDF␣Version␣" ≪ VERSION_3DLDF ≪ ".\n\n" ≪ flush;
  }
  return (0); }
```

1558. Putting Main together. This is what's compiled.

```
⟨ Include files 6 ⟩
⟨ Version control identifier 5 ⟩
⟨ Main 1555 ⟩
```

1559. Appendices.

1560. References.

Cundy, H. Martyn and A.P. Rollet. *Mathematical Models.* Oxford 1961. Oxford University Press. Unfortunately out of print.

Finston, Laurence D. *3DLDF User and Reference Manual.* Göttingen 2003.

Fischer, Gerd. *Ebene algebraische Kurven.* Vieweg Studium. Aufbaukurs Mathematik. Friedr. Vieweg Sohn Verlagsgesellschaft mbH. Braunschweig/Wiesbaden 1994.

Harbison, Samuel P., and Guy L. Steele Jr. *C, A Reference Manual.* Prentice Hall. Englewood Cliffs, New Jersey 1995. ISBN 0-13-326232-4 {Case}.
ISBN 0-13-326224-3 {Paperback}.

Hobby, John D. *A User's Manual for MetaPost.* ATT Bell Laboratories. Murray Hill, NJ. No date.

Jones, Huw. *Computer Graphics through Key Mathematics.* Springer-Verlag London Limited 2001. ISBN 1-85233-422-3.

Knuth, Donald Ervin. *Metafont: The Program.* Computers and Typesetting; D. Addison Wesley Publishing Company, Inc. Reading, Massachusetts 1986.
ISBN 0-201-13438-1.

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

Knuth, Donald Ervin. *TeX: The Program.* Computers and Typesetting; B. Addison Wesley Publishing Company, Inc. Reading, Massachusetts 1986.
ISBN 0-201-13437-3.

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

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

Salomon, David. *Computer Graphics and Geometric Modeling.* Berlin 1999. Springer-Verlag. ISBN: 0-387-98682-0.

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

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

Stroustrup, Bjarne. *The Design and Evolution of* C++. Reading, Massachusetts 1994. Addison-Wesley Publishing Company.
ISBN 0-201-54330-3.

1561. GNU Free Documentation License. The GNU Free Documentation License, which follows, applies to this document.

GNU Free Documentation License
Version 1.2, November 2002

A “Secondary Section” is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The “Invariant Sections” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The “Cover Texts” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A “Transparent” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not “Transparent” is called “Opaque”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The “Title Page” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

A section “Entitled XYZ” means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as “Acknowledgements”, “Dedications”, “Endorsements”, or “History”.) To “Preserve the Title” of such a section when you modify the Document means that it remains a section “Entitled XYZ” according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission. B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement. C. State on the Title page the name of the publisher of the Modified Version, as the publisher. D. Preserve all the copyright notices of the Document. E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices. F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below. G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice. H. Include an unaltered copy of this License. I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence. J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission. K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein. L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles. M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version. N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section. O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

⟨ GNU Free Documentation License 1561 ⟩ ≡ /* This section contains no C++ code. */

This code is cited in section 1.

1562. GNU General Public License. The GNU General Public License, which follows, applies to the program 3DLDF described in this document.

GNU GENERAL PUBLIC LICENSE
Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software–to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License.

If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO

LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

⟨one line to give the program's name and a brief idea of what it does.⟩
Copyright (C) ⟨year⟩ ⟨name of author⟩

This program 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 2 of the License, or (at your option) any later version.

This program 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 this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

Gnomovision version 69, Copyright © year name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type 'show w'.
This is free software, and you are welcome to redistribute it under certain conditions; type 'show c' for details.

The hypothetical commands 'show w' and 'show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than 'show w' and 'show c'; they could even be mouse-clicks or menu items–whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the program 'Gnomovision' (which makes passes at compilers) written by James Hacker.

⟨signature of Ty Coon⟩, 1 April 1989 Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

⟨ GNU General Public License 1562 ⟩ ≡ /* This section contains no C++ code. */

This code is cited in section 1.

1563. Index. [LDF 2002.10.09.] The way CWEAVE handles indexing is not ideal for C++. It doesn't index identifiers that include non-alphanumeric characters, so that neither **Path** :: *draw* ( ) nor **operator** ≪ ( ) are indexed automatically. Nor is there any indication of whether an identifier refers to a variable or a function.

I have added indexing commands in the source files for operators and class member functions. However, the alphabetization routine is naïve and doesn't ignore the characters of the TEX macros that I use for formatting the index entries, so the order of the entries is a bit peculiar. For example, `\cfunc{z}` would come before `\ofunc{a}` (`\cfunc` is for class functions, and `\ofunc` is for operators that aren't members of a class).

First come the index entries which start with "??" and "!!". These are followed by the non-operator class member functions for all of the classes. Then come the class member operators for all of the classes, followed by the non-class operators. Currently, I'm only putting index entries in by hand where the class member functions and operators are declared.

[LDF 2002.10.11.] Another problem is that "*operator&*" and "*operator&=*" use the italic ampersand. It would be possible to fix this, but slightly tricky. TO DO: Fix this!