dxflib

Programmer's Guide

Author: Andrew Mustun

Release: March 2005

Contents

Chapter 1

Overview

dxflib is a C++ library for reading and writing DXF files. When reading DXF files, dxflib parses the file and calls functions that you define in your own C++ class for adding entities, layers, ..

Please note that dxflib does not store any entities or other information for you. It only passes the supported entities and other objects found in the DXF file to your C++ class.

Using dxflib to read DXF files doesn't require any knowlege of the DXF format. However, it's still an advantage to know the basics about entities, attributes, layers and blocks. To write DXF files with dxflib you definitely need an idea of how a DXF file is organized.

dxflib does not depend on any exotic other libraries, just the standard C / C++ libraries.

Chapter 2

Compiling dxflib

Unix / Linux

Under Unix and Linux Systems, compiling dxflib is a simple standard procedure:

```
./configure
make
```

This creates the file './lib/dxflib.a' . To compile a dynamic version of dxflib, run 'make shared' instead of 'make' . This will create the file './lib/libdxf.so.2.0.x.x' and the link './lib/libdxf.so' that points to './lib/libdxf.so.2.0.x.x' .

Instead of running 'make install', you can also copy the header files into a header directory of your choice and copy the library files into a library directory of your choice.

Windows

There are many different ways to compile dxflib under Windows. You can use Microsoft's Visual C++ compiler, Borland's C++ command line tools, gcc and many other compilers.
For this manual, only the process of compiling dxflib using cygwin [CYGWIN] and the gcc compiler from the MinGW32 package is shown:

```
./configure
MinGW32-make
```

Chapter 3

Reading DXF Files

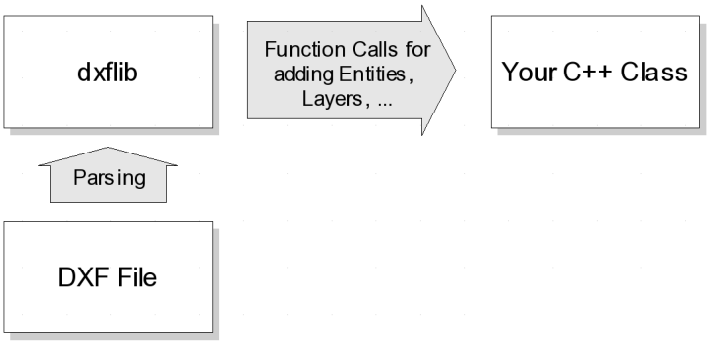

Figure 1: dxflib parses DXF files and calls functions in your class. In those functions you can for example add the entities to a vector or list of entities.

Implementing the Creation Interface

Your C++ class that handles DXF entities has to be derived from DL_CreationInterface or DL_CreationAdapter. In most cases DL_CreationAdapter is more convenient because it doesn't force you to implement all functions.

```
class MyDxfFilter : public DL_CreationAdapter {
    virtual void addLine(const DL_LineData& d);
    ...
}
```

The implementation of the functions in your class will typically add the entities to a container of entities or use the information in another way.

```
void MyDxfFilter::addLine(const DL_LineData& d) {
    std::cout << "Line: " << d.x1 << "/" << d.y1
    << " " << d.x2 << "/" << d.y2 << std::endl;
}
```

When reading a DXF file you simply pass on a reference to an object of your class to the parser.

```
MyDxfFilter f;
DL_Dxf dxf;
if (!dxf.in("drawing.dxf", &f)) {
    std::cerr << "drawing.dxf could not be opened.\n";
}
```

Chapter 4

Writing DXF Files

To write a DXF file, you need to wrap the entities, layers, blocks, .. you have into the wrapper classes of dxflib. Since dxflib does not store any entities, you need to iterate through your entities and call the write functions for each of them. Please note that you have to stick to the exact order in which you call the write functions of dxflib. Otherwise your DXF file will not be standard conform.

Creating the Writer Object

To create a DXF writer object you need to specify the file name as well as the DXF version you want to produce. At the time of writing only two DXF versions were supported: R12 and DXF 2000/2002. The dxflib codes for DXF version R12 is DL_Codes::AC1009 and for DXF 2000/2002 DL_Codes::AC1015.

There are two APIs you will need to write a DXF file. The API in DL_WriterA offers low level functions to write basic key/value tuples on which a DXF file is based. Creating a valid DXF file using only these functions would be very difficult and inconvenient. Therefore, there is a higher level API in the DL_Dxf class which allows you to write for example a whole line without knowing the key/value tuples that are needed for it.

The following code creates and opens a file for a DXF 2000/2002 drawing:

```
DL_Dxf dxf;
DL_Codes::version exportVersion = DL_Codes::AC1015;
DL_WriterA* dw = dxf.out("myfile.dxf", exportVersion);
if (dw==NULL) {
    printf("Cannot open file 'myfile.dxf' \
            for writing.");
    // abort function e.g. with return
}
```

Writing the DXF Header

Opening the DXF Header

The DXF header contains information about the DXF version. It has to be written before anything else with

```
dxf.writeHeader(*dw);
```

The following list shows how a DXF header typically looks like:

```
999
dxflib 2.0.4.8
  0
SECTION
  2
HEADER
  9
$ACADVER
  1
AC1015
  9
$HANDSEED
  5
FFFF
```

As you can see, the writeHeader() function does not close the header. This is because you might want to store a set of variables into it. If you have to store variables, you have to do it now. If not, proceed with "Closing the Header".

Storing Additional Variables

Variables in the DXF header are used to store meta data for the drawing contained in the file. For a description of all supported variables, please refer to the DXF documentation [DXF].

The following code snippet shows examples for storing variables of different types. You can store as many variables as you need but you have to stick to the supported variable names and types in order to create a valid DXF file.

```
// int variable:
dw->dxfString(9, "$INSUNITS");
dw->dxfInt(70, 4);
// real (double, float) variable:
dw->dxfString(9, "$DIMEXE");
dw->dxfReal(40, 1.25);
// string variable:
dw->dxfString(9, "$TEXTSTYLE");
dw->dxfString(7, "Standard");
// vector variable:
dw->dxfString(9, "$LIMMIN");
dw->dxfReal(10, 0.0);
dw->dxfReal(20, 0.0);
```

Closing the Header

Use the following code to close the DXF header (end the current section):

```
dw->sectionEnd();
```

Writing the Tables Section

Opening the Tables Section

The tables section of a DXF file contains some tables defining viewports, linestyles, layers, etc.

Open the tables section with the function:

```
dw->sectionTables();
```

Writing the Viewports

Viewports are not directly supported by dxflib. However, they still need to be there in a valid DXF file. You can write the standard viewports using the function:

```
dxf.writeVPort(*dw);
```

Writing the Linetypes

Only linetypes that are actually used need to be defined in the DXF file. For simplification, you might want to store all linetypes supported by dxflib as shown below.

```
dw->tableLineTypes(25);
dxf.writeLineType(*dw, DL_LineTypeData("BYBLOCK", 0));
dxf.writeLineType(*dw, DL_LineTypeData("BYLAYER", 0));
dxf.writeLineType(*dw,
    DL_LineTypeData("CONTINUOUS", 0));
dxf.writeLineType(*dw,
    DL_LineTypeData("ACAD_ISO02W100", 0));
dxf.writeLineType(*dw,
    DL_LineTypeData("ACAD_ISO03W100", 0));
dxf.writeLineType(*dw,
    DL_LineTypeData("ACAD_ISO04W100", 0));
dxf.writeLineType(*dw,
    DL_LineTypeData("ACAD_ISO05W100", 0));
dxf.writeLineType(*dw, DL_LineTypeData("BORDER", 0));
dxf.writeLineType(*dw, DL_LineTypeData("BORDER2", 0));
dxf.writeLineType(*dw, DL_LineTypeData("BORDERX2", 0));
dxf.writeLineType(*dw, DL_LineTypeData("CENTER", 0));
dxf.writeLineType(*dw, DL_LineTypeData("CENTER2", 0));
dxf.writeLineType(*dw, DL_LineTypeData("CENTERX2", 0));
dxf.writeLineType(*dw, DL_LineTypeData("DASHDOT", 0));
dxf.writeLineType(*dw, DL_LineTypeData("DASHDOT2", 0));
dxf.writeLineType(*dw,
    DL_LineTypeData("DASHDOTX2", 0));
dxf.writeLineType(*dw, DL_LineTypeData("DASHED", 0));
dxf.writeLineType(*dw, DL_LineTypeData("DASHED2", 0));
dxf.writeLineType(*dw, DL_LineTypeData("DASHEDX2", 0));
dxf.writeLineType(*dw, DL_LineTypeData("DIVIDE", 0));
dxf.writeLineType(*dw, DL_LineTypeData("DIVIDE2", 0));
dxf.writeLineType(*dw,
    DL_LineTypeData("DIVIDEX2", 0));
dxf.writeLineType(*dw, DL_LineTypeData("DOT", 0));
dxf.writeLineType(*dw, DL_LineTypeData("DOT2", 0));
dxf.writeLineType(*dw, DL_LineTypeData("DOTX2", 0));
dw->tableEnd();
```

Writing the Layers

Layers are a substantial part of most DXF drawings. All layers that are used in the drawing need to be defined in this table section. The following example code writes three layers with names "0", "mainlayer" and "anotherlayer" to the DXF file. Note that before writing the layers, you need to specify how many layers there are in total. Layer "0" is the default layer. It cannot be omitted.

```
int numberOfLayers = 3;
dw->tableLayers(numberOfLayers);

dxf.writeLayer(*dw,
    DL_LayerData("0", 0),
    DL_Attributes(
        std::string(""),      // leave empty
        DL_Codes::black,      // default color
        100,                  // default width
        "CONTINUOUS"));       // default line style

dxf.writeLayer(*dw,
    DL_LayerData("mainlayer", 0),
    DL_Attributes(
        std::string(""),
        DL_Codes::red,
        100,
        "CONTINUOUS"));

dxf.writeLayer(*dw,
    DL_LayerData("anotherlayer", 0),
    DL_Attributes(
        std::string(""),
        DL_Codes::black,
        100,
        "CONTINUOUS"));

dw->tableEnd();
```

The default line width is given in 1/100mm. The color enum in namespace DL_Codes defines the most common colors.

Writing Various Other Tables

These tables are also needed. For more information, please refer to the DXF documentation [DXF].

```
dxf.writeStyle(*dw);
dxf.writeView(*dw);
dxf.writeUcs(*dw);

dw->tableAppid(1);
dw->tableAppidEntry(0x12);
dw->dxfString(2, "ACAD");
dw->dxfInt(70, 0);
dw->tableEnd();
```

Writing Dimension Styles

Dimension Styles define the look of dimensions.

```
dxf.writeDimStyle(*dw,
    arrowSize,
    extensionLineExtension,
    extensionLineOffset,
    dimensionGap,
    dimensionTextSize);
```

Writing Block Records

Block records define the names of available blocks in the DXF file. The following example declares the existence of two blocks with names "myblock1" and "myblock2". Note that the first call is also needed. It opens the blocks table and writes some standard blocks that might be required by the DXF version.

```
dxf.writeBlockRecord(*dw);
dxf.writeBlockRecord(*dw, "myblock1");
dxf.writeBlockRecord(*dw, "myblock2");
dw->tableEnd();
```

Ending the Tables Section

```
dw->sectionEnd();
```

Writing the Blocks Section

The blocks section defines the entities of each block.

```
dw->sectionBlocks();

dxf.writeBlock(*dw,
    DL_BlockData("*Model_Space", 0, 0.0, 0.0, 0.0));
dxf.writeEndBlock(*dw, "*Model_Space");

dxf.writeBlock(*dw,
    DL_BlockData("*Paper_Space", 0, 0.0, 0.0, 0.0));
dxf.writeEndBlock(*dw, "*Paper_Space");

dxf.writeBlock(*dw,
    DL_BlockData("*Paper_Space0", 0, 0.0, 0.0, 0.0));
dxf.writeEndBlock(*dw, "*Paper_Space0");

dxf.writeBlock(*dw,
    DL_BlockData("myblock1", 0, 0.0, 0.0, 0.0));
// ...
// write block entities e.g. with dxf.writeLine(), ..
// ...
dxf.writeEndBlock(*dw, "myblock1");

dxf.writeBlock(*dw,
    DL_BlockData("myblock2", 0, 0.0, 0.0, 0.0));
// ...
// write block entities e.g. with dxf.writeLine(), ..
// ...
dxf.writeEndBlock(*dw, "myblock2");

dw->sectionEnd();
```

Writing the Entities Section

The entities section defines the entities of the drawing. The two entities in the following example use the attributes of their layer (256 = color by layer, -1 = line width by layer, "BYLAYER" = line style by layer).

```
dw->sectionEntities();

// write all your entities..
dxf.writePoint(
    *dw,
    DL_PointData(10.0,
                 45.0,
                  0.0),
    DL_Attributes("mainlayer", 256, -1, "BYLAYER"));

dxf.writeLine(
    *dw,
    DL_LineData(25.0,   // start point
                30.0,
                 0.0,
               100.0,   // end point
               120.0,
                 0.0),
    DL_Attributes("mainlayer", 256, -1, "BYLAYER"));

dw->sectionEnd();
```

Writing the Objects Section

```
dxf.writeObjects(*dw);
dxf.writeObjectsEnd(*dw);
```

Ending and Closing the File

```
dw->dxfEOF();
dw->close();
delete dw;
```

Chapter 5

Appendix

Bibliography

[DXF]
http://www.autodesk.com/techpubs/autocad/acad2000/dxf
Autodesk DXF Reference

[CYGWIN]
http://www.cygwin.com
cygwin - a Linux-like environment for Windows.

Chapter 6

Index