FreeType 2 - D Language Interface

Introduction

FreeType 2 is an excellent font utility library available to the C language developer. This valuable library has been ported to many platforms and has proved to be the tool of choice in many open and closed source projects. FreeType is, therefore, a worthy addition to the D language programming arsenal. It so happens that easy interfacing with C is one of D's specialties. This D language binding to the FreeType library offers a ready solution for the D programmer. The following discussion provides an overview of the D language interface to FreeType 2.

Using FreeType the Dynamic Way

D sports some practical language features that are designed simplify the development process. D's FreeType 2 Binding makes use of one of these: it is called a static constructor. The static constructor is a way to initialize modules prior to execution of a D program's main() function. The D FreeType interface, in particular, uses this feature to load a shared library and then initialize a list of function pointers to their respective shared library symbols. There is no need for the developer to worry about these details however, other than to understand what is happening when they import the freetype interface into their program; all freetype symbols are available to the D programmer. When the program is compiled and executed, the static constructor is executed first guaranteeing that all symbols used in the project are valid before the D main() function executes. The loader library linked with the project takes care of this.

Setting up the Environment

In order to use the freetype bindings, it is advisable that you make some changes to your dmd compiler settings. First, create an "import" directory in your dmd path like this: *.\dmd\import* (forward slashes for Linux). Place the D freetype modules in a new *freetype* directory in *import.* The file structure should look like this:

\dmd\import\freetype\ft.d
\dmd\import\freetype\def.d
\dmd\import\freetype\loader.d

Next add *\dmd\import* to your sc.ini (or dmd.conf for linux) include path. You should do this by adding a semicolon to the current path plus the designated directory above.

With the above completed, you should only have to place an *import freetype.ft;* statement in your project source to make use of the freetype bindings. Dmd will now know where to look for the bindings when you compile your project.

Overview of the Dynamic Library Loader

This interface depends on the presence of a freetype shared library on the target operating system. On Windows, it is assumed that the library is called *freetype.dll*; and on Linux, the loader searches for the file named *libfreetype.so*. This project is still a work in progress so very little is done to verify that these library names point to the correct version of freetype. But library presence or absence detection is straightforward and automatic. If the library names exist on the file system path or the library loader path, then they will be loaded as expected.

On Windows, this package includes a specially built version of freetype.dll that includes all freetype 2 modules and functions compiled in by default. Thus, the library loading mechanism is guaranteed to load all symbols correctly as long as the freetype.dll is in the correct path; thus the windows version of the interface, if used with the included freetype.dll, results in the best chances of valid operation of the freetype library.

On Linux, there is no guarantee that the latest libfreetype.so (1.2.10 is assumed with this interface) has been installed on certain distributions. On some linux systems this may result in some symbols failing to load. Future versions of this library should provide a check that allows the programmer to determine which freetype modules or functions are available for use. At present, the library loader merely outputs the symbol names that failed to load and the number of failed symbols. The library still allows the valid symbols to be used. On the other hand, attempting to use the function pointers to the failed symbol names will result in a program crash. Despite this forewarning, current tests on an up to date Gentoo Linux distribution revealed no errors in the symbol loading mechanism. All symbols loaded as expected indicating that the freetype system on this Linux system appears to default to a full freetype build.

Differences between the D Interface and the C library

While D and C share many similarities, there still exist several conflicts that demand slight variation in the library interface implementation. These conflicts do not affect the operation of the interface as a whole. They only influence how the programmer uses some freetype features from D. This section discusses how D deals with differences in freetype macro definitions, structures, and enumerations.

Firstly, D does not support C macro definitions. Some of these, as defined in the C freetype library, have no existing counterpart in the D interface. Where possible, some macro equivalents can be implemented as D functions. The current D interface implements very few, since most are not strictly required in FreeType. More may be added as the need arises.

Secondly, many of the C structure names have been simplified in there D form since D has a much simpler and more logical way of declaring structure types. This should have no influence on the interface, but may, at some point, confuse those who look at the D definitions and find no exact equivalent of the C structure names. Most D structure names do not use the *typedef* attributes like there C counterparts. Instead they directly define the structure type and its pointer on separate lines.

Lastly, in both C and D, enumerations are commonly used to neatly organize constant definitions according to application or function. When translating these to D, it appears that a fully qualified name is required in places where C only required the individual enumerated constant. This means that D requires more typing to perform the same task than C. The advantage in D's favor is clarity. The disadvantage is a lack of direct equivalence with the C version. The solution is to remove all enumerated type names, and make all D enumerated types anonymous. This produces the same result as the C version and may be the preferred route to take in future improvements to the library. For now, the current system will remain.

Conversion Process and Technical Details

This project was an attempt to acquire a complete interface to the C FreeType 2 library for D. As such, it involved a fairly comprehensive conversion task. Of especial relevance was the extreme use of macros contained in the C freetype project source, perhaps most infamous of the challenges faced by the D developer who confronts the C to D conversion process. In this case, the conversion approach followed this pattern:

- Preprocess FreeType C source with a macro preprocessor; result: macro clean source with function and data type declarations.
- Take preprocessor output and hand manipulate in preparation for python script consumption.
- Run a specialized (for parsing) python script on augmented source; result: D function declarations for exported symbols in properly arranged format for dynamic loader system.
- Take all preprocessor output and gather by hand all structures, enumerations, and function pointers and quick convert to D (only minor changes necessary).
- Join the output of the last two steps and add the dynamic loader code.
- Recompile the C FreeType Library as a dll using the dmc compiler system.

Tools used:

Mcpp – a free and extremely useful standard C/C++ macro preprocessor for linux and windows.
Python 2.4 – for scripting support and custom conversion scripts.
Dmc – to create the windows freetype.dll for the best chances for compatibility with dmd code.

Files Included in Project

These are the files included in this project package:

.\freetype.dll	dmc compiled freetype dll for win32 with all freetype 2 modules included
.\freetype	Place files in this directory in .\dmd\import path
.\freetype\ft.d	FreeType function declarations and static constructor code
.\freetype\def.d	FreeType types, structures, enumerations and constants
.\freetype\loader.d	Module that handles loading the freetype dynamic library
.\examples	Contains test programs to verify working state of FreeType interface
.\examples\example1.d	Render text test
.\examples\ftest.d	Simple loader test
.\examples\testname.d	Load font and list font glyph names
.\examples\aircut.ttf	Sample font for use with test programs
.\examples\tahoma.ttf	Sample font
.\examples\trebuchet.ttf	Sample font

Have fun!

For further information or questions, contact me:

John J. Reimer
terminal.node@gmail.com
or JJR – dsource.org