DDX SDK

Delphi DirectX Plugin (DirectShow Filter) SDK

Author: Trevor Magnusson
Eddress: trevor@cloneensemble.com
Website: www.cloneensemble.com

History

After writing some VST plugins in Delphi, I wanted to port them to DirectX. Since I could not find any Delphi DirectX SDKs on the web, I decided to develop my own. I purchased Microsoft Visual C++ Version 6, and started off with the Cakewalk DirectX wizard. My approach was to write an "empty" plugin, that would link to a Delphi DLL (via a simple, old-style, procedural interface). Having delved down into C++, I could seal it off and spend the rest of the time working in Delphi.

Credits

The DDX helper DLL was created using the Cakewalk DirectX wizard, and I received some assistance from Jesse Jost at Cakewalk. Matthjis Hebly provided some valuable testing and feedback.

Copyright and distribution

Disclaimer

This SDK has been developed using Delphi version 5. There are no guarantees that it will work in other versions.

Overview

The DDX SDK lets you write DirectX filters using Delphi, without having to mess around with DirectShow and COM. This documentation assumes the following:

- You are fluent in Delphi.
- You know about audio processing algorithms. (If you've written a VST plugin in Delphi, you're in a good position.)
- You don't want to mess around with COM or C++.

There are some limitations on the types of plugins that you can write using this SDK, because to make things simple I had to make some compromises.
With the DDX SDK, you can write filters that can:

- Process IEEE 32-bit floating point and/or 16-bit integer PCM data.
- Process any combination of mono/stereo input/output (you choose which combinations to accept, eg you can force mono->stereo if you like).
- Have up to 16 parameters.

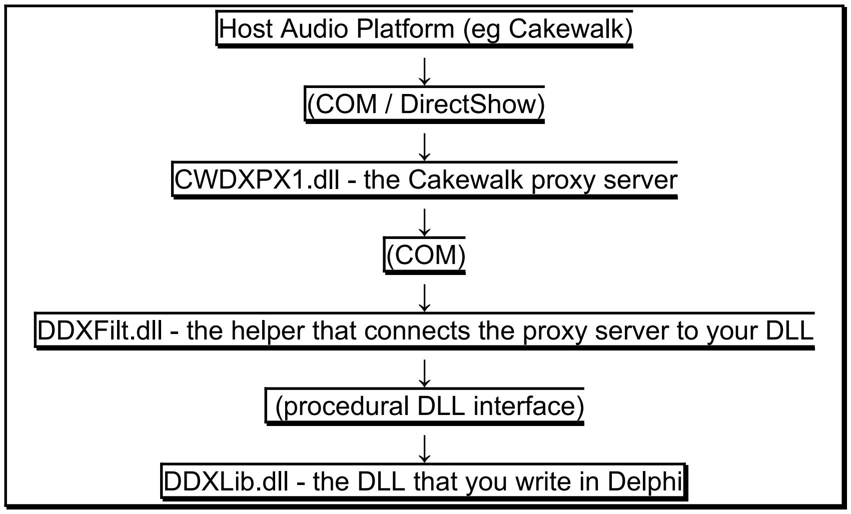
However, this SDK does **not** support:

- In-place buffers

- Multi-pass filtering
- Asynchronous (ie overlapping-blocks) filtering

The advantage of in-place filtering is speed, but on today's PCs the difference is not significant. If you want to build the equivalent of an in-place filter, make sure that you support only mono-mono and stereo-stereo modes, do a Delphi `move` command to copy the input buffer to the output in one go, and then process the output buffer. The `move` procedure is very fast, and this approach will halve the number of "pointer-step" operations - you will only step through the output buffer.
If you are seriously interested in multi-pass or asynchronous plugins, you really should consider learning C++, since you will have to work at the advanced end of town.

Here is how the final plugin will hang together:


Host Audio Platform (eg Cakewalk)
(COM / DirectShow)
CWDXPX1.dll - the Cakewalk proxy server
(COM)
DDXFilt.dll - the helper that connects the proxy server to your DLL
(procedural DLL interface)
DDXLib.dll - the DLL that you write in Delphi

Before we step through the tutorial, we need to establish some very important rules:

Rule #1

You **MUST** create and use two fresh GUIDs for **every** plugin project.
The included little app, NewGUIDs.exe, can do this for you. It possible, run this app on a PC that has a network adapter card installed (ie, take NewGUIDs.exe to your office PC, run it there, and email the GUIDs to your home PC).
Under **NO** circumstances should a new project use GUIDs that have been used in another project.
This is **THE most** important rule for using this SDK.
If you break this rule, you'll not only mess up your own system, but that of **everybody else** who installs your plugins. You will be **very unpopular** if this happens.
(If you don't know what GUID are, they are unique identifiers associated with your plugin and its editor form, and once Windows has been told about them, it uses these GUIDs as a sort of handle to those classes. The important thing is that they are unique.)

Rule #2

You will be creating a DLL for each plugin you write. The name of that DLL **must** be DDXLib.dll (Delphi DirectX Library). That means that your project file must be called DDCLib.dpr. That's right - **ALL** of your plugins must have the **SAME FILENAME**, and it must be DDXLib. Obviously, you will have to use a different folder for each one, to keep them separate.
Don’t panic – we are not talking about the name of the plugin that the user will see, just the name of the DLL file that you will be building – they are two different things.

Tutorial

Let's look at all the files we've unzipped, and some suggested locations.
First, create a folder for DDX common files. This folder will have to be in the search path of all your projects.

C:\DDX\COMMON

This folder should contain:

DDXBase.PAS	the base classes and interface calls. You will need to use this unit in your own Plugin units, and in all your DDXLib.DPR files.
DDXLib.INC	an include file that exports all the interface calls. You will need to include it in all your DDXLib.DPR files.

C:\DDX\LIB

This folder should contain the two helper DLLs - you will need to copy them to the output folder for each of your projects, but this is a good central place to store the "original" copies:

CWDXPX1.DLL	the Cakewalk proxy server DLL.
DDXFilt.DLL	the helper DLL that connects your DDXLib.DLL files with the Cakewalk proxy server. It is actually this DLL that is "installed", but it will call your Delphi DLL to do all the actual work.

C:\DDX\DDXDemo

This folder is for the sample plugin, which we shall step through directly. You will need a separate folder for each one of your plugin projects. These folders must contain:

DDXDemo.pas/dfm	the Delphi Pascal source for this particular plugin. For more advanced plugins, you may have several source files, one of which would have the editor form. The names of these files don't matter.
CWDXPX1.DLL	copied over from C:\DDX\LIB.
DDXFilt.DLL	copied over from C:\DDX\LIB. (This assumes the output directory of your project is left blank, or set to the same as the source files - C:\ DDX\ DDXDemo in this case).
DDXLib.DPR	Your project source file - it must be called this name.

DDXDemo
Let's step through the sample plugin provided.

(1) Let's look at the DDXLib.DPR. As you can see, it is very small. It must contain the following:
- A delaration that DDXLib is a library
- A reference to DDXBASE in its **uses** clause,
- A reference to your main unit(s) in its **uses** clause,
- A resource directive,
- An include directive for the file DDXLib.inc.
- An empty begin..end block.

(2) Our "main unit" happens to be called DDXDemo, and here is a .PAS and a .DFM file. Now let's have a look at the .PAS file.
- You must have DDXBase in your uses clause,
```
uses … DDXBase;
```
- You must declare some constants:
```
const
  EffectName     = 'DDXDemo';
  EffectDesc     = 'Delphi DirectX Demo';
  EffectHelpFile = 'DDXDemo.hlp';
  PropertiesName = ' DDX_Demo_Settings';
  EffectGUID     = '{D1600122-381B-11D5-A607-444553540000}';
  EffectGUID     = '{D1600122-381B-11D5-A607-444553540000}';
```
  Remember - when you create your own new plugins, you must use two freshly-created GUIDs here!!! Use the included app NewGUIDs to generate them for you, then simply copy and paste them in.
- You'll need a declaration of the editor form, which is partly managed by Delphi when you create the form unit. This will have a reference to a base plugin class.
```
type
  TDDXDemoForm = class (TForm)
    …
  private
    fPlugin : TDelphiDirectXPluginBase;
    …
  end;
```

- You must declare a class which descends from TDirectXPluginBase,

```
  type
    TDDXDemoPlugin = class (TDelphiDirectXPluginBase)
```

- We override the following methods:

```
  constructor create; override;
```

  (a) to enforce the number of parameters (zero)
  (b) to allow you to specify the channel modes and sample types you are prepared to support.
  (c) and of course, you need to specify default starting up values for your effect's parameters
  (d) and allocate any buffers / working variables etc.

  Why to we need to override the constructor? Virtual constructors are one of the "tricky" features of Delphi, but we definitely need it here. Later we will be storing a *reference* to this descended class in a reference variable declared as the ancestor's class reference, so if we don’t have a virtual constructor, the ancestor class will be created instead of ours!

```
  destructor destroy; override;
```

  to free any buffers etc you have created

```
  procedure CompleteConnectEditor; override;
```

  This gets called after the effect class have been connected to the editor form, but the editor form is not yet connected to the effect. So we override this to make our editor form aware of the effect.

```
  procedure DisplaySetParameter (
              aParamNum : integer;
              aValue    : single); override;
```

  This gets called after the system has modified one of the parameters, and we override it so that we can update the associated control on our editor form.

```
  function Transform (
             aInBuffer  : pointer;
             aInSamples : integer;
             aOutBuffer : pointer;
             aOutSamples: integer;
             aProcessed : pInteger)
                        : integer; override;
```

  This is where the audio work gets done.

  Important note: there are some curious points to be made about the relationship between `aInSamples` and `aProcessed`.

  cmMonoToMono: `aInSamples` is the number of **mono** samples, which means that as each time you increment your counter, you increment the buffer pointer. When you are finished, you should copy `aInSamples` into `aProcessed`.

  cmStereoToToStereo: `aInSamples` is the number of **stereo** samples, which means that for each time you increment your counter, you must increment your buffer **twice** – once for the left value, once for the right. Once again, when you are finished, you should copy `aInSamples` into `aProcessed`.

  cmMonoToStereo: `aInSamples` is the number of mono samples, and as you read each sample, you will be writing **two** samples to the output buffer. The curious point is that `aProcessed` requires the number of single samples – left and right. This means that you must use `2 * InSamples`.

  cmStereoToMono: I have experienced problems with this mode, some platforms do not seem to support it.

- There are some other key methods that you would probably need to override. In this demo project, we won’t do it, but let’s look at some of the important ones anyway:

```
  function SetInputFormat (aFormat : PWAVEFORMATEX) : integer; override;
  function SetOutputFormat (aFormat : PWAVEFORMATEX) : integer; override;
```

  These Set...Format calls are the first time you will know about the number of input and output channels, the sample format and the sample rate. If you need to do some precomputation based on these items, you might override these calls. Of course, if you just need to use these values as they are, they will be in place by the time Transform is called.

```
function SetParameter (
           aParamNum : integer;
           aValue    : single)
                     : integer; override;
```

Called by the C++ plugin to set a parameter. If you need to do some pre-computation based on parameters, you might override this.

- While we're about it, let's look at some of the key properties available:

```
property ParamArray [ndx: integer] : single;
```

An array from 0..15 of your plugin parameters.

```
property SampleType    : tSampleType;
```

The actual sample type we are dealing with. This will not be known until the host platform has called SetInputFormat or SetOutputFormat.

```
property InputChannels : integer;
property OutputChannels: integer;
```

The number of input and output channels. In your overridden constructor you can specify which combinations of input and output channels you are prepared to handle. The host platform then decides what to use, and calls SetInputFormat and SetOutputFormat to inform you of that decision.

```
property SampleRate    : integer;
```

The sample rate, in samplers per second. Once again, you have to wait until SetInputFormat or SetOutputFormat to find out what the value is.

- OK, back to the demo project. We add some named properties for our parameters. This is optional, you can always reference the plugin parameters with the ParamArray property [0..15], but it's easier to understand code that has them by name, and Delphi provides a cool way for doing this. In our demo plugin, we use:

```
property Volume: single index 0 read GetParam write SetParam;
property Pan   : single index 1 read GetParam write SetParam;
```

NB: in the unlikely event that you need more than 16 parameters (you're writing a *graphic equalizer*?) you'll just have to get by with the technique of rolling two parameters into a single value. For instance, if you have two parameters *a* and *b*, each of which is 0..1, your storage rule might be to convert *a* to an integer from 0 to 1000 (use round(*a**1000)), and then restrict *b* to 0.0 to 0.99999 (if *a*=1 then…). Then you can safely add them together and store them in a single number. To separate them again use trunc(*n*)/1000 to get *a* and frac(*n*) to get *b*. To nail it home, if a is 0.123 and b is 0.456, your storage parameter would be 123.456.

- We declare two more (typed) constants:

```
const
  DelphiPluginClass     : TDelphiPluginClassRef     = TDDXDemoPlugin
  DelphiEditorFormClass : TDelphiEditorFormClassRef = TDDXDemoForm;
```

- In the implementation section, we implement all the methods we have overridden, and some events for our editor form. We won't include that code in this doco, please see the source file for details.
- Finally, in the initialization section, we call RegisterEffect, so that the system will know about the GUIDs, effect names, and overridden classes we have declared:

```
RegisterEffect (
  EffectName,
  EffectDesc,
  EffectHelpFile,
  PropertiesName,
  EffectGUID,
  PropertiesGUID,
  DelphiPluginClass,
  DelphiEditorFormClass);
```

(3) A point to make about the .DFM file – the form. The visible property of the form (in the object inspector) should be false. The reason for this is that before loading the plugin, the host application will create a temporary instance of the editor form, to discover its size. If your form has the visible property set to true, you will get some annoying flashing on the screen.

(4) That's it! After compiling the project, we simply need to register (install) the effect.

Installation - during development

After you have compiled your DLL, you should organise things so that you have a folder that contains DDXLib.DLL, DDXFilt.DLL and CWDXPX1.DLL. Then from a command shell, while sitting in that directory, you should run:

```
REGSVR32.exe DDXFilt.dll
```

You only need to do this once per effect, as long as the files remain in that directory, you can recompile as many times as you need to, and the changes will take effect without needing to register again.

If you need to unregister an effect, you should download the free DXMAN utility from www.analogx.com. This is a very useful utility, and you should mention it to anyone you distribute your plugins to.

MakeInst - the easy way to distribute your effect

When distributing your plugin, you can simply include the instructions above, and your users can go through that procedure. But that is not very slick, and some users might have problems following the instructions.

The MakeInst utility can be used to construct a single .EXE file for the user to double-click on, which will lead them through a simple wizard and perform all this for them.

MakeInst.exe operates in two modes:

Build mode

To run MakeInst in build mode, you must first construct a WinZip file (if you don't have WinZip, you can find a demo version on the web, the only limitation of which is that it makes you press an extra button on startup). This WinZip file must contain:

DDXLib.DLL	your plugin DLL
DDXFilt.DLL	the DDX helper DLL
CWDXPX1.DLL	the Cakewalk helper DDL
{filename}.BMP	(optional) a picture, ideally 300*150 pixels, which will be displayed as the opening page in the install wizard - perhaps a logo for your plugin? The actual filename doesn't matter - so long as it's a bitmap.
ReadMe.TXT	(optional) a text file describing the plugin, with licensing information, acknowledgements etc. This will be displayed as the second page in the wizard.

You then run the following:

```
MakeInst {zipname} {NewExeName}
```

where

{ZipName}	is the name of the zip file you have constructed,
{NewExeName}	is the name of the .EXE file that will be created - perhaps something helpful that includes the word "Install" and the name of your plugin - InstallDDXDemo.EXE if we were distributing this demo plugin.

This will create an installer program that you can distribute. You should zip up that exe file if you want people to download it from a web page, because although it already contains your zipped up data, the executable part is not compressed.

Install mode

Install mode is simply what happens when you run the new installer program that you have created in build mode. It steps the user through an install wizard that displays your logo bitmap, then your readme.txt, then a "Choose path to install files into" page, after which the installation process is complete.

Installation - advanced

These notes will be of interest if you do not use MakeInst for your installation.

You must implement a setup and installation procedure that does three things. You can use the included specially-designed utility MakeInst, the popular InstallShield, some other third-party utility, or you can write your own. Whatever the case, it must do the following:

(1) Choose a directory (folder) that the files will live in. Since we have already established that all plugins will have the same name, we must have a separate directory for each plugin.

(2) Copy your DDXLib.dll and the two "helper" dlls (DDXFilt.dll and CWDXPX.dll) to this directory.

(3) Run "REGSVR32.exe /s DDXFilt.dll" (no quotes). You should find REGSVR32.exe in the Windows System directory (the exact location will depend on the OS, NT is usually different to 9x). If you are doing this with a batch file, make sure you are "in" the directory where the files have been copied. If you are doing it with a call to CreateProcess, you should specify the full pathname (ie prepend the directory) of DDXFilt.dll.
NB: please note that we are installing DDXFilt.DLL - this is not the DLL that you have written in Delphi, it is one of the distributed helper DLLs - and the one that must be registered.

Source code

The C++ source code for CWDXPX1.DLL is held by Cakewalk and is not available.
The C++ source code for DDXFilt.DLL is available upon request, however I do not wish to provide user support.
The Delphi source code for MAKEINST.EXE is available upon request, however to compile it you will need to buy Turbo Power Abbrevia for Delphi 5, and I do not wish to provide user support.
The Delphi source code for NEWGUIDS.EXE is available upon request, and it is so basic that I don't think user support will be required!