DirectShow For Media Playback In Windows

Part I: Basics

By Chris Thompson (aka Lightman)
28 July 2000

Read This Disclaimer

Introduction

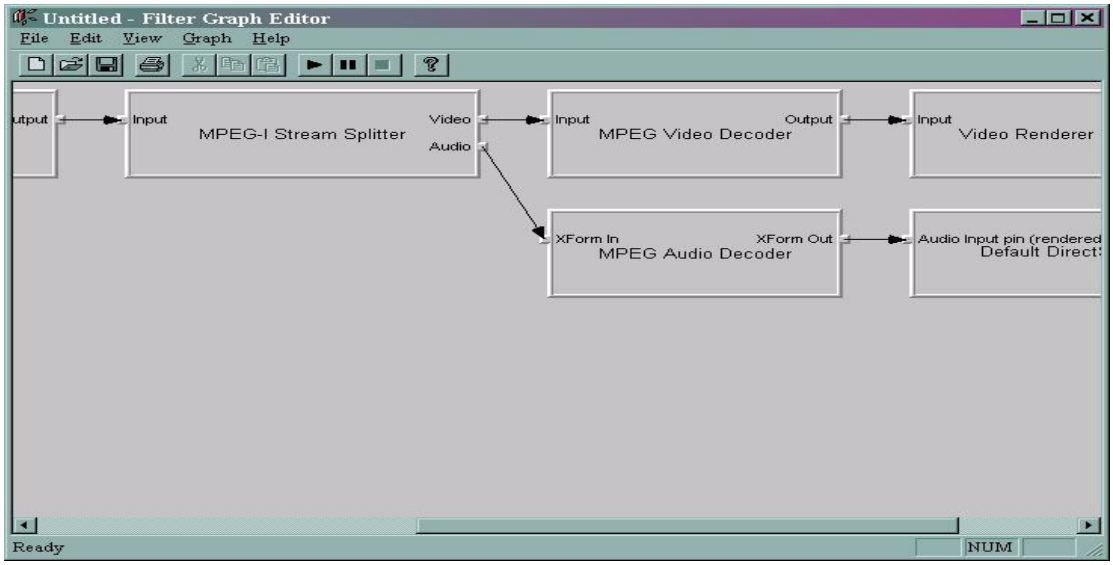
A few years ago, Microsoft introduced a media streaming layer on top of DirectX that was meant to handle pretty much any type of media you could throw at it, called DirectShow. It's included in DirectX Media, which also includes DirectAnimation (mostly web page stuff), DirectX Transform (more web page stuff), and the old Direct3D Retained Mode that used to be part of the standard DirectX collection. This tutorial will show you how to play back standard media types in your game or application using DirectShow from C++. Microsoft doesn't offer the DirectX Media SDK for download, so you either have to order a CD or look in the latest Platform SDK.

A Note On COM

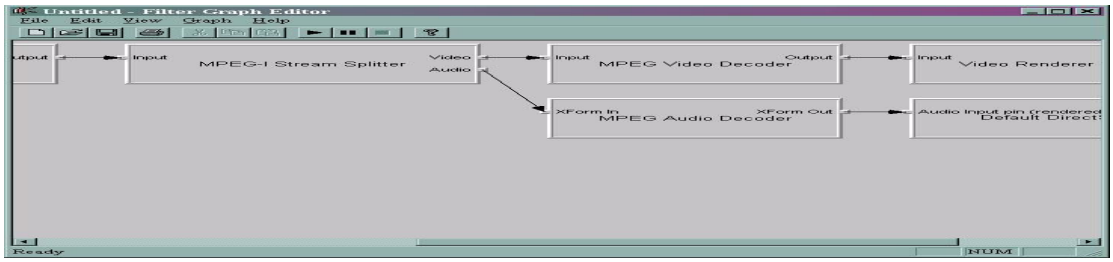
As with DirectX, the DirectShow API is accessed through COM interfaces, so I assume from the start that you have at least a basic knowledge of COM and how it works as far as getting pointers to the interfaces you need and releasing references correctly.

An Explanation Of DirectShow

DirectShow is set up with the ideas of a number "filters" joined together to create a "graph". Here's a visual representation of the graph an filters using a utility that comes with the DirectXMedia SDK, called GraphEdit:



Mike Hommel -
Steven Hugg -
Toby Jones -
Jacob McCosh -
Kurt Miller -
Jan Niestadt -
Tane Piper -
Muresan Robert -
Adam Robinson -
Henry Robinson -
Conor Stokes -
Jaap Suter -
Nicholas Vining -
Peter White -



Each box represents a filter. Arrows connecting boxes represent the output of one filter being passed to the input of another filter. Arrows also show the flow of data in the graph. GraphEdit is nice for those just getting started with DirectShow, as it gives a nice visual equivalent to what you'll be doing in software. GraphEdit also lets you drag filters around, connect them to other filters, and to run your final complete graph.

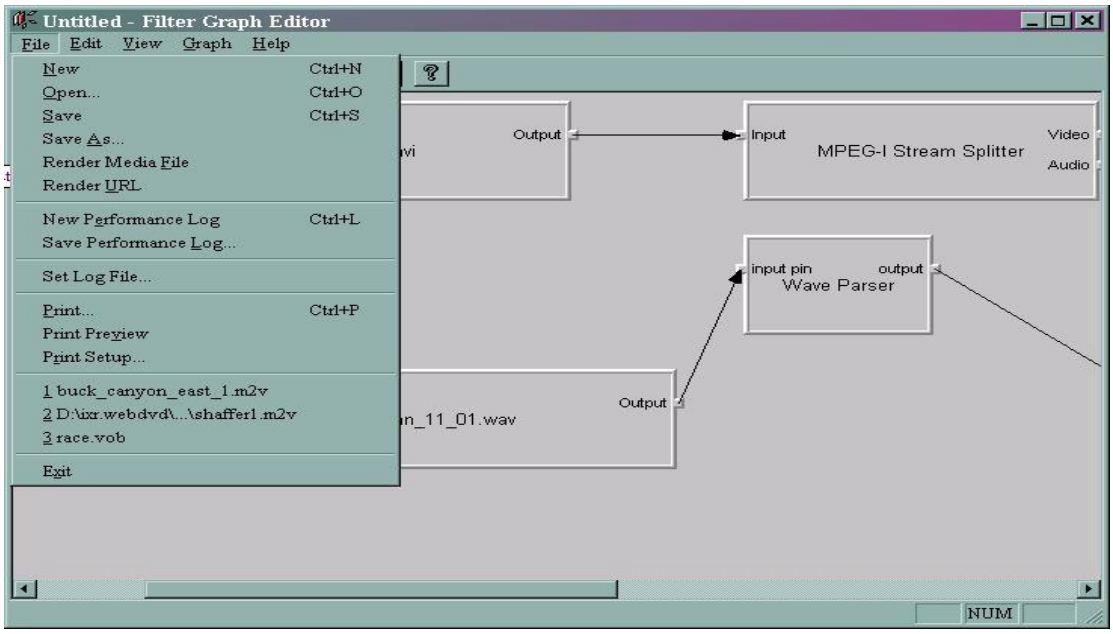
Each graph that is built follows certain guidelines. First off, there needs to be a source filter. This is the initial source of your data, be it a file, a URL for streaming media, or some device such as a firewire card hooked to a video camera. The output of the source filter is then run through any number of transform filters. Transform filters are any intermediate filters that take a certain type of input data, modify the data coming in, then pass the modified data to it's output. The final piece of a graph is a renderer filter. Renderer filters are the final destination of any data handled in a filter graph. Renderers can represent things such as a video window for displaying video on the screen, sound card for outputting sound, or a file writer for storing data to disk.

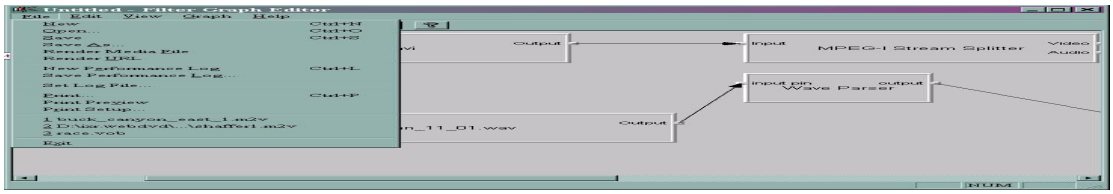
The way that filters are connected in a graph is through their "pins". Every filter, no matter what type, must have at least one pin to connect to other filters. When attempting to connect two filters, the pins on both the filters pass information back and forth to determine if the downstream filter (the one accepting data) can handle the data passed in by the upstream filter (the one sending data). If the pins successfully negotiate a data type they both know, a successful connection has been made between the 2 filters. As you can see in the image above, a filter is not restricted to a number of inputs or outputs, and many times a filter will require more than one input or output for the data being handled. For example, the MPEG-1 Stream Splitter filter needs to send the audio and video portions of MPEG-1 data to separate decoder filters.

DirectShow is distributed with a number of filters provided by Microsoft, including source filters, transform filters, and renderer filters. They provide us with a useful "File Source Filter" that we can use for reading in any type of file, transform filters capable of handling MPEG-1 video, AVI video, WAV audio, and other formats, and finally, renderer filters for outputting sound and video. For standard formats, the filters provided by Microsoft may be all you need for playback.

Building And Testing A Graph In GraphEdit

GraphEdit is a great tool that you can use to create and test a filter graph to play a media type that you plan on using in your program. I won't go into details on how to use GraphEdit, but it's an important that you know it and use it before you end up wasting time coding, only to realize that 2 certain filters don't agree on a data type. The first thing to do is to select "Render Media File" from the menu and select the media file that you wish to playback.





GraphEdit will attempt to build a graph that is capable of playing back the file you chose. If it succeeds, you can press the 'play' button on the toolbar to run your graph. If it fails, you should select "Graph->Insert Filters" from the menu, and look for any alternate filters that may work with your media type that you can try connecting into your graph. Sometimes the default decoder or renderer that GraphEdit places in the graph is not the desired one, or doesn't work, and on occasion you may find an alternate filter that will work better.

The most important thing is to find a combination of filters that successfully render your media, because if it doesn't work in GraphEdit, it's not going to work in your program.

A Simple Graph Built Using C++

The following code is an example in C++ that creates a graph for playing a WAV file, plays 10 seconds of it, then shuts it down and exits.

Download The Example Source Code (2k)

If you plan on compiling this, be sure you've installed the DirectX Media SDK, and included strmbasd.lib (debug) or strmbase.lib (retail) so it compiles correctly.

Run this program from the command line, passing in the file to play as an argument. Example: dshowtut1 test.wav

A Note On Threads

One important thing to realize when using DirectShow for media playback is that it uses separate threads to run the filter graph. Using Spy++, it shows that DShowTut1 has 10 threads when running. I can't account for all the threads, but I think what DirectShow will do is create 1 thread for the graph manager (IGraphBuilder), and a separate thread for each filter in the graph. This has it's good and bad points. Separate threads rendering your media means that your program can go back to doing other things while the media is playing. It also means that you have to free up time to let those threads run.

If you have a full framerate game running and feel like using DirectShow to render a wav file for background music, you probably will end up with choppy sound output, because your game loop will be running at a higher thread priority than the DirectShow threads, so Windows will take timeslices away from your game only when it feels it's absolutely necessary. Because of this, DirectShow might not be the best solution for certain applications. If your app can spare some time slices, say during cut scenes, or doesn't have a fast-as-it-can-go main loop, you need to free up your main thread to allow others to run. Threads is a pretty deep subject, so I won't dwell on it anymore than I already have. Just keep in mind that you need to free some time for the DirectShow threads to run, and if you get choppy sound/video output, hogging CPU time in the main thread is probably why.

Conclusion

This tutorial didn't delve too much into the depths of DirectShow, but you should now have at least a knowledge of the basic concepts of DirectShow. You should also now be able to see if it's possible for you to use DirectShow in your application by building and testing a graph using the GraphEdit tool. The DShowTut1.cpp sample code won't do much for anyone, I know, but you can see the C++ side of what you did with GraphEdit.

In the next tutorial, I'll explain more in depth the main interfaces of DirectShow and how to use them. There will also be alot more code for you to sink your teeth into.

Return To The Tutorial Index