money doesn't matter to me.

features
Development News -
Development Tutorials -
Development Links -
Industry Interviews -
Computer Book Reviews -
Programming Contest -
FC Developer Desktops -

columns
Currently Active:
Ask MidNight - Q&A -
Theory & Practice -
DirtyPunk's Column -
3D Geometry Primer -
Harmless Algorithms -

Complete / Closed:
Art Of Demomaking -
Fountain Of Knowledge -
Network Programming -
Portal Engine Series -
Cool, It Works! -

forums
Message Center -
#flipCode IRC Channel -

hosted
gollum.flipcode.com -
freeside.flipcode.com -
jratcliff.flipcode.com -
epicboy.flipcode.com -
fourth.flipcode.com -
tentacle.flipcode.com -
orbit.flipcode.com -
ghoul.flipcode.com -
polygone.flipcode.com -
pixelpracht.flipcode.com
frog.flipcode.com -

site info
General Information -
Advertising Information -
News Archives -

tech files
Tech File Information -

Anis Ahmad -
Jacco Bikker -
Jeroen Bouwens -
Lionel Brits -
Phil Carlisle -
Alex Champandard -
Roy Jacobs -
Alexander Herz -
Victor Ho -
Luke Hodorowicz -
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 -

DirectShow For Media Playback In Windows

Part III: Customizing Graphs

By Chris Thompson (aka Lightman)
13 September 2000

tutorial Flipcode

Read This Disclaimer

Introduction

This installment of the DirectShow tutorial series will cover customizing filter graphs. There are many reasons why we might need to customize a filter graph. We may want to add a transform filter that will EQ our audio output, or add a color correction filter to our video output. Sometimes GraphBuilder just doesn't add the filter we want, so we need to fiddle with things a bit to get the desired filter graph. This is also good for learning what actually goes on during filter graph creation: how filters are added to a filter graph, how 2 filters are connected, etc.

More Interfaces

Here's a rundown of the interfaces we will be using. More in depth descriptions can be found in the DirectShow documentation.

- IEnumFilters - interface implemented by GraphBuilder that lets us enumerate all the filters in a filter graph
- IBaseFilter - interface implemented by all DirectShow filters
- IEnumPins - interface implemented by filters to let us enumerate all their pins
- IPin - interface implemented by all pins in a filter

In order to make changes to a filter graph, we need to know these interfaces. You'll see that connecting two filters is done at the pin level using the IPin interface. This means that we need to be able to get the IPin interface of any pins belonging to any filters in the filter graph.

There are a few different ways to build a suitable filter graph for any purpose. The first way is automatically build graphs using IGraphBUilder::RenderFile(), like we've done so far. Another way we can do it is by instantiating every filter we need and connecting them all one by one. It's not the easiest way, but possible, and a good learning exercise. The best way, though, is to let GraphBuilder do as much of the work as possible, while manually modifying the filter graph for the desired final graph. I'll call these semi-automatically created graphs. Let's look at each one separately.

Automatically Built Graphs

This is the way we've built all filter graphs so far. Quick Recap:

```
IGraphBuilder*  g_pGraphBuilder;        // already created

void BuildGraph(char* file)
{
        WCHAR*  wfile;
        int     length;

        length = strlen(file)+1;
        wfile = new WCHAR[length];
        MultiByteToWideChar(CP_ACP, 0, file, -1, wfile, length);
        g_pGraphBuilder->RenderFile(wfile, NULL);       // graphbuilder creates entire graph for us
}
```

Manually Built Graphs

Now let's say now that we have used GraphEdit to build a graph for a .WAV file, so we've seen the completed graph and know exactly which filters we need and how they are to be connected. We can build the filter graph from the ground up by adding filters one at a time. Here's a quick look at what the filter graph should be:

<file source filter>--<wave parser>--<sound renderer>

Remember that in COM we have interfaces, indicated by 'IID_xxx', and specific implementations of that interface, indicated by 'CLSID_xxx'. We need to know a class identifier (CLSID_xxx) for each filter in order to call CoCreateInstance to create that filter. Most of the common source and renderer filter CLSID's are listed in the uuids.h header. We can find CLSID_FileSource and CLSID_AudioRender in there (or CLSID_DSoundRender for the DirectSound renderer). Now how do we find a CLSID for the 'Wave Parser' filter that decodes the stream? Easiest way is to search through the registry for the string "Wave Parser" and

use the CLSID that you find. There are other ways, like using IFilterMapper and IEnumRegFilters to get a list of registered filters, but for this I've just looked it up.

First, let's create the filters we know we need.

```
// our own definition of Wave Parser based on CLSID we found in registry
#define INITGUID        // have to define this so next line actually creates GUID structure
DEFINE_GUID(CLSID_WaveParser,
0xD51BD5A1, 0x7548, 0x11CF, 0xA5, 0x20, 0x00, 0x80, 0xC7, 0x7E, 0xF5, 0x8A);

IGraphBuilder* g_pGraphBuilder; // assume graph builder already created

IBaseFilter*    g_pSource;
IBaseFilter*    g_pWaveParser;
IBaseFilter*    g_pSoundRenderer;

void CreateFilters()
{
        CoCreateInstance(CLSID_FileSource, NULL, CLSCTX_INPROC_SERVER, IID_IBaseFilter, (void**)&g_pSource);
        CoCreateInstance(CLSID_WaveParser, NULL, CLSCTX_INPROC_SERVER, IID_IBaseFilter, (void**)&g_pWaveParser);
        CoCreateInstance(CLSID_DSoundRender, NULL, CLSCTX_INPROC_SERVER, IID_IBaseFilter, (void**)&g_pSoundRenderer);

        g_pGraphBuilder->AddFilter(g_pSource, NULL);
        g_pGraphBuilder->AddFilter(g_pWaveParser, NULL);
        g_pGraphBuilder->AddFliter(g_pSoundRender, NULL);
}
```

Notice I just used the DEFINE_GUID macro to define the CLSID (which is a GUID) for the wave parser. A GUID is just a structure, so all we need is to create it and put the right values in.

Now that we have the filters we want in our filter graph, we need to connect them. Connections between filters are done at the pin level, so we need to find the right pins we want to connect on each filter. There's a few different ways we can connect pins. IFilterGraph provides a ConnectDirect() method where we can specify a media type to use for the connection, and the 2 pins to be connected. This is fine if you know the media type that is going to be used, and you are confident that the 2 filters will be happy connecting to each other. A better way is to use IGraphBuilder::Connect(), which tries to connect 2 pins directly, and if that fails it tries to find combinations of filters that will go in between the 2 pins to complete the connection. This is the safest way, so it's what we'll use.

We use the IEnumPins interface to enumerate all the pins for the filters we want to connect. Since filters can have any number of pins, we need to pick the right ones to connect. We can use some of our extra knowledge we got from looking at the completed graph in GraphEdit. We know that the source filter and render filter both only have 1 pin, so no problem with pins there. We also know that the Wave Parser only has 1 input pin and 1 output pin. To make sure we have the pin we want, all we have to do is check the pin direction with IPin::QueryPinInfo().

```
// Assume here we've created the filters and added them to our graph.
// There normally would be more error checking in here, but that's
// something you can figure out.

// assume all of these are created and filters have been added to graph
IGraphBuilder*  g_pGraphBuilder;
IBaseFilter*    g_pSource;
IBaseFilter*    g_pWaveParser;
IBaseFilter*    g_pSoundRenderer;

void ConnectFilters()
{
        IEnumPins*      EnumPins;
        IPin*           OutPin;
        IPin*           InPin;
        ULONG           fetched;
        PIN_INFO        pinfo;

        // find source output
        g_pSource->EnumPins(&EnumPins);
        EnumPins->Reset();
        EnumPins->Next(1, &OutPin, &fetched);   // only 1 pin for source, so we know this is the one we need
        EnumPins->Release();

        // find wave parser input
        g_pWaveParser->EnumPins(&EnumPins);
        EnumPins->Reset();
        EnumPins->Next(1, &InPin, &fetched);
        InPin->QueryPinInfo(&pinfo);
        pinfo.pFilter->Release();               // make sure you release the returned IBaseFilter interface
        if (pinfo.dir == PINDIR_OUTPUT)         // check if we have wrong pin (not input pin)
        {
                InPin->Release();
                EnumPins->Next(1, &InPin, &fetched);            // if so, get next pin
        }

        // connect --
        g_pGraphBuilder->Connect(OutPin, InPin);
        InPin->Release();
        OutPin->Release();

        // find wave parser output
        EnumPins->Reset();
        EnumPins->Next(1, &OutPin, &fetched);
        OutPin->QueryPinInfo(&pinfo);
        pinfo.pFilter->Release();
        if (pinfo.dir == PINDIR_INPUT)
        {
                OutPin->Release();
                EnumPins->Next(1, &OutPin, &fetched);
        }
        EnumPins->Release();

        // find renderer input
        g_pSoundRenderer->EnumPins(&EnumPins);
        EnumPins->Reset();
        EnumPins->Next(1, &InPin, &fetched);    // renderer has only 1 pin, so this is the pin we need
        EnumPins->Release();

        // connect --
        g_pGraphBuilder->Connect(OutPin, InPin);
        InPin->Release();
        OutPin->Release();
}
```

Now we should have a fully functional filter graph exactly like the one created by calling IGraphBuilder::RenderFile(). This is alot of work for us compared to automatically generated graphs, so we normally don't use this approach. It's still good to see

this and know it is possible to do. Now let's look at a more practical way to have control over our graph while letting GraphBuilder handle all the tedious work.

Semi-Automatically Built Graphs

When I say semi-automatically built graphs, this can mean a few things. We could have a filter graph that is automatically generated, but we insert or change some filters. We could also create a source filter, add it to the graph (connect to some transform filters if we need to), and let graphbuilder automatically create the rest of the graph. Finally, we could add some specific filters we want to use to the graph, and let graphbuilder render the entire graph and it will try to use the filters we've created. I'll quickly go over each one now.

Let's say we're still just playing a .WAV file. But this time, we want to run it through a transform filter to apply an effect to it. We'll use the Gargle filter since it's a sample provided with the DShow SDK. Note that sample filters need to be built and registered before they can be used, so make sure you build the Gargle sample and run 'regvsr32' on it before you try to create a Gargle filter. Our filter graph will look like so:

<source filters>--<wave parser>--<gargle effect>--<sound renderer>

We let graphbuilder create the default graph (without gargle filter). Then we enumerate filters and look for the renderer filter. We know that uncompressed sound is passed into the sound renderer from the filter it's connected to, and that the gargle effect input pin accepts uncompressed sound data also. So we can just slip the gargle filter in ahead of the renderer. For transform filters, this is a good way to insert them. You may want to insert an effect into a graph for mp3's, or AIFF files, and you can be pretty confident that the renderer's input is always going to be a format that the effect filter can take as input.

We can find the render filter either by looking for it by name, or by checking number of output pins. The only filter in a completed graph to have no output pins should be the renderer. In order to make our code work with different renderers (remember there are at least 2 different audio renderers that come with DirectShow), we don't want to look for it by name. So we'll find it by checking for output pins. Once we have found the renderer filter, we'll connect the gargle filters pins.

Here's the code to do this:

```
IGraphBuilder*  g_pGraphBuilder;        // assume already created

void BuildGraph()
{
        IEnumFilters*   EnumFilters;
        IBaseFilter*    Renderer;
        IBaseFilter*    Gargle;
        IEnumPins*      EnumPins;
        bool            FoundRenderer = false;
        IPin*           InPin;  // renderer input
        IPin*           OutPin; // decoder or other filter output;
        IPin*           GargleIn;
        IPin*           GargeOut;
        ULONG           fetched;
        PIN_INFO        pinfo;
        int             numoutputpins = 0;

        g_pGraphBuilder->RenderFile(L"blah.wav");       // 'L' macro makes it WCHAR like we need it to be
        g_pGraphBuilder->EnumFilters(&EnumFilters);
        EnumFilters->Reset();
        while (FoundRenderer == false)
        {
                EnumFilters->Next(1, &Renderer, &fetched);      // get next filter
                Renderer->EnumPins(&EnumPins);
                EnumPins->Reset();
                numoutputpins = 0;
                while (EnumPins->Next(1, &InPin, &fetched) == S_OK)
                {
                        InPin->QueryPinInfo(&pinfo);
                        pinfo.pFilter->Release();
                        InPin->Release();
                        if (pinfo.dir == PINDIR_OUTPUT)
                        {
                                numoutputpins++;
                                break;          // we can jump out if we found an output pin
                        }

                }
                EnumPins->Release();
                if (numoutputpins == 0)
                        FoundRenderer = true;
                else
                        Renderer->Release();
        }
        EnumFilters->Release();

        // Find renderer input
        Filter->EnumPins(&EnumPins);
        EnumPins->Reset();
        EnumPins->Next(1, &InPin, &fetched);    // first one is only one
        EnumPins->Release();

        // Find ouput pin on filter it is connected to
        Pin->ConnectedTo(&OutPin);

        // Disconnect the filters - note that we have to call Disconnect for both pins
        g_pGraphBuilder->Disconnect(InPin);
        g_pGraphBuilder->Disconnect(OutPin);

        // Create Gargle filter - CLSID for gargle filter is defined in garguids.h in sample source
        CoCreateInstance(CLSID_Gargle, NULL, CLSCTX_INPROC_SERVER, IID_IBaseFilter, (void**)&Gargle);
        g_pGraphBuilder->AddFilter(Gargle, NULL);

        // get it's pins
        Gargle->EnumPins(&EnumPins);
        EnumPins->Reset();
        EnumPins->Next(1, &GargleIn, &fetched);
        GargleIn->QueryPinInfo(&pinfo);
        pinfo.pFilter->Release();
        if (pinfo.dir = PINDIR_OUTPUT)
```

```
        {
                GargleOut = GargleIn;
                EnumPins->Next(1, &GargleIn, &fetched);
        }
        else
                EnumPins->Next(1, &GargleOut, &fetched);

        // now connect pins
        g_pGraphBuilder->Connect(OutPin, GargleIn);
        g_pGraphBuilder->Connect(GargleOut, InPin);

}
```

Well that was long and messy. At least we now have the graph the way we want it, and it's independant of which decoder and renderer are being used. Notice that in order to get the output pin of the filter that was previously connected to the renderer, we just call the renderer input pin's IPin::ConnectedTo().

The second way we can create semi-automatically built graphs makes use of some things we know about how graphbuilder creates graphs. We know that while GraphBuilder is attempting to create a graph, it tries to connect the current filter its working with, to any filter already in the graph before it starts adding new filters to the graph. This means that we can create a Gargle filter, add it to the graph, and call IGraphBuilder::RenderFile(), and graphbuilder will automatically stick the Gargle filter in the first place it will connect. Here's basically how we do this:

```
IGraphBuilder*  g_pGraphBuilder;        // already created

void Buildgraph()
{
        IBaseFilter*    Gargle;

        CoCreateInstance(CLSID_Gargle, NULL, CLSCTX_INPROC_SERVER, IID_IBaseFilter, (void**)&Gargle);
        g_pGraphBuilder->AddFilter(Gargle, NULL);
        g_pGraphBuilder->RenderFile(L"blah.wav");
}
```

That's it. You can do this same thing in GraphEdit to see that it works. Some filters, for 1 reason or another, will not automatically be inserted. The best way to ensure that a filter ends up where you want it is to add it to the graph first, then enumerate it's pins and check if they are connected. If not, then go through the method of insertimg it manually before the renderer as we did earlier.

The last way to create a semi-automatically generated graph is to create the first filter (or filters) and render an output pin. Let's say this time we want to play a .WAV file from a website. Instead of the regular 'File Source (Async)' filter, we need the 'File Source (URL)'. What we need to do is create the source filter explicitly, then we can call IGraphBuilder::Render() on it's output pin. We're not restricted to calling Render() on source filters. Any time we have a partial graph (a source filter connected to any number of transform filters), we can call Render() on the unconnected output pin of the last filter in line, and graphbuilder will attempt to complete the graph for us. Here's the way that would go:

```
// our own definition of 'File Source (URL)' based on CLSID we found in registry
#define INITGUID        // have to define this so next line actually creates GUID structure
DEFINE_GUID(CLSID_FileSourceURL,
0xE436EBB6, 0x524F, 0x11CE, 0x9F, 0x53, 0x00, 0x20, 0xAF, 0x0B, 0xA7, 0x70);

IGraphBuilder*  g_pGraphBuilder;        // already created


void BuildGraph()
{
        IBaseFilter*            Source;
        IFileSourceFilter*      FileSource;
        IEnumPins*              EnumPins;
        IPin*                   Pin;

        CoCreateInstance(CLSID_FileSourceURL, NULL, CLSCTX_INPROC_SERVER, IID_IBaseFilter, (void**)&Source);
        g_pGraphBuilder->AddFilter(Source, NULL);
        Source->QueryInterface(IID_IFileSourceFilter, (void**)&Source);
        FileSource->Load("http://www.asite.com/blah.wav");      // set file to play
        FileSource->Release();
        Source->EnumPins(&EnumPins);
        EnumPins->Reset();
        EnumPins->Next(1, &Pin, &fetched);
        EnumPins->Release();
        g_pGraphBuilder->Render(Pin);   // graphbuilder completes graph
        Pin->Release();
        Source->Release();
}
```

Conclusion

Using these techniques, you should now be able build any filter graph you want. In future tutorials, we'll look at using IDDrawExclModeVideo to get DirectShow to play in a window owned by our app, and also look at Multimedia streaming using DirectShow. Multimedia streaming is very useful for games since it basically streams media in just like a file stream, while running it through a filter graph on the way. I'll also be covering building custom filters sometime too, so that you can make your own transforms or use custom media formats.

Return To The Tutorial Index

The views expressed in this document are those of *the author*, not neccesarily anyone else related to flipCode. This document may not be reproduced in any way without explicit permission from *flipCode* and the author.

Any and all trademarks used belong to their respective owners. Please read our Terms, Conditions, and Privacy information.
This site is optimized for at least 1024x768 resolution (hi-color) viewing with a browser that supports style sheets.