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 -

# DirectShow For Media Playback In Windows

Part II: DirectShow In C++

By Chris Thompson (aka Lightman)
12 August 2000

Read This Disclaimer

## Introduction

In the last part of this tutorial we learned the basic concepts behind DirectShow, but we didn't go into too much actual code. Now we will have a look at the standard interfaces used in DirectShow a little more in depth.

## The Standard Interfaces

The basic DirectX facilities, such as DirectDraw, DirectSound, and Direct3D, for the most part are accessed through one interface, IDirectDraw, IDirectSound, and IDirect3D respectively. DirectShow is a little more complex than this. In a typical application using DirectShow, you may have 3 or more interfaces you need in order to control your filter graph. Here's a list of the most common interfaces, which we will be using:

**IGraphBuilder** - This is the most important interface. It provides facilities to create a graph for a specific file, and to add/remove filters individually from the graph.

**IMediaControl** - This interface provides us with methods to start and stop the filter graph.

**IMediaEventEx** - This interface is used for setting up notifications that will be sent to the app when things happen in the graph (reach end of media, buffer underruns, etc.)

**IMediaPosition** - This interface will let us seek to certain points in our media, let us set playback rates and other useful bits.

## Getting Ready To Play A File

First thing we do is create the interfaces we described above:

```
IGraphBuilder*  g_pGraphBuilder;
IMediaControl*  g_pMediaControl;
IMediaEventEx*  g_pMediaEvent;
IMediaPosition* g_pMediaPosition;

int InitDirectShow()
{
        HRESULT hr;

        hr = CoCreateInstance(CLSID_FilterGraph, NULL, CLSCTX_INPROC_SERVER,
        IID_IGraphBuilder,(void**)&g_pGraphBuilder);

        if (FAILED(hr))
                return -1;
```

```
        g_pGraphBuilder->QueryInterface(IID_IMediaControl, (void**)&g_pMediaControl);
        g_pGraphBuilder->QueryInterface(IID_IMediaEvent, (void**)&g_pMediaEvent);
        g_pGraphBuilder->QueryInterface(IID_IMediaPosition, (void**)g_pMediaPosition);
        return 0;
}
```

Once we have these interfaces, we need to set up notifications. It is possible to not have notifications set up, but you have much more control if you do. Maybe you need to do something right when the media ends, such as go from a cut-scene to playing the actual game. You can poll for events if you like, but you have to be sure to free up enough CPU time to let the DirectShow threads run. For now, we'll just be using notifications.

You can set up notifications in 2 ways. The first way is by event triggers. You can ask for an event handle that will be triggered when an event occurs, which you can use in a number of ways. You can use MsgWaitForMultipleObjects in your main loop instead of GetMessage, or you can create a thread that calls WaitForSingleObject that will wake when the event is triggered. The second way we can set up notifications is by window messages. You can set a window as the recipient of a window message when an event occurs. This is the way it is done most of the time and the way we will use for our simple media player.

```
#define WM_GRAPHEVENT   WM_USER         // define a custom window message for graph events
HWND    g_AppWindow;                    // applications main window

void SetNotifications()
{
        g_pMediaEvent->SetNotifyWindow((OAHWND)g_AppWindow, WM_GRAPHEVENT, 0);
        g_pMediaEvent->SetNotifyFlags(0);       // turn on notifications
}
```

In our WindowProc we add some code to handle the notifications:

```
LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
        switch (uMsg)
        {
        case WM_GRAPHEVENT:
                OnGraphEvent();         // handles events
                break;
        default:
                break;
        }
        return DefWindowProc(hwnd, uMsg, lParam, wParam);
}
```

Once we receive a graph event, we must do a few things. DirectShow stores some data for events internally, so that memory must be freed once we are done reacting to the event. Also, a window message is sent only once when the event queue goes from being empty to having messages in it, so we must handle all pending events. Here is the code that would get called from the above WindowProc:

```
void OnGraphEvent()
{
        long EventCode, Param1, Param2;
        while (g_pMediaEvent->GetEvent(&EventCode, &Param1, &Param2, 0) != E_ABORT)
        {
                switch (EventCode)
                {
                // react to specific events here
                default:
                        break;
                }
                g_pMediaEvent->FreeEventParams(EventCode, Param1, Param2);
        }
}
```

Now we're pretty much ready for playback. Once we know what file we wish to play, we ask IGraphBuilder to create a suitable filter graph for it:

```
int CreateGraph(char* filename)
{
        int     length;         // length of filename
        WCHAR*  wfilename;      // where we store WCHAR version of filename

        length = strlen(filename)+1;
        wfilename = new WCHAR[length];
        MultiByteToWideChar(CP_ACP, 0, filename, -1, wfilename, length);
        if (FAILED(g_pGraphBuilder->RenderFile(wfilename, NULL))
                return -1;
        else
                return 0;
}
```

Note that RenderFile only accepts WCHAR strings, so we have to convert if we are using ANSI char strings. If GraphBuilder successfully creates a agraph for our media, we can start playing it like so:

```
g_pMediaControl->Run();
```

## Cleaning Up

The standard practice is to release all interfaces, including IGraphBuilder, when you are done playing your media. We can do this with the following:

```
// convenient macro for releasing interfaces
#define HELPER_RELEASE(x)    if (x != NULL) \
                             { \
                                 x->Release(); \
                                 x = NULL; \
                             }

void CleanUpDirectShow()
{
    HELPER_RELEASE(g_pMediaPosition);
    HELPER_RELEASE(g_pMediaEvent);
    HELPER_RELEASE(g_pMediaControl);
    HELPER_RELEASE(g_pGraphBuilder);
}
```

The HELPER_RELEASE macro is just a real easy way to check if an interface has already been released (or failed to be created) so we don't have exceptions popping up.

Usually you will release all these interfaces as soon as you are done playing your media. I know it seems like a waste if you are going to be using DirectShow again later in your app, but it's the easiest way to get rid of all filters and reset DirectShow. There are ways to get around this, but we'll maybe touch on that in a future tutorial section. For now we'll rebuild the graph for each media file.

## Simple Media Player

Now we need to put this all together into a simple media player. Here we have a small app that will play most standard media types, such as WAV files, MIDI files, AVI files, MPEG files.

dshowtut2.zip (26k)

This program will let you select a media file to open, then it will try to render a filter graph for it. There are Play and Stop commands to control playback, as well as a Looping option.

## Graph States

When the graph is in a running state, it stays in the running state even when all data has been passed through the graph. When the EC_COMPLETE notification is sent, all data has passed through all filters and they are now waiting for more data. If you look at event handling code in our media player, you will see this:

```
switch (EventCode)
{
        case EC_COMPLETE:
                // here when media is completely done playing
                if (!g_Looping)
                        g_pMediaControl->Stop();
                g_pMediaPosition->put_CurrentPosition(0);   // reset to beginning
                break;
        default:
                break;
}
```

Note that we don't need to call IMediaControl::Run() again after setting the position back to the beginning of the media in order to loop it. Also, we need to explicitly call IMediaControl::Stop() when we hit EC_COMPLETE. This isn't necessary, but it helps us to better keep DirectShow under control.

## Letting GraphBuilder Create A Graph

For the media player, we let GraphBuilder have control over creation of the graph, but it is good to know what is going on in the background. One helpful way to see how GraphBuilder goes about finding an appropriate filter graph is to call IGraphBuilder::SetLogFile(). This will tell GraphBuilder to create a log of the steps it is taking in it's attempt to create a filter graph

for the media. The code is simply:

```
HANDLE  logfile;
char    logfilename[MAX_PATH];

// create win32 file handle
sprintf(logfilename, "%s\\graph.log", g_ExecuteDir);
LogFile = CreateFile(logfilename, GENERIC_WRITE, 0, NULL, CREATE_ALWAYS,
FILE_ATTRIBUTE_NORMAL, NULL);

g_pGraphBuilder->SetLogFile(logfile);
```

There's one problem though, and that is the log file is created wherever the media file is located. The media player program uses Win32's GetOpenFileName() to choose a file, which I guess changes the directory of execution also. To get around this, I just saved the directory of execution at program initialization and recalled it later.

If you look at the log file that GraphBuilder creates (all you need to do is run the app and choose a file) you will see lots of details about creating filters, trying to connect pins on filters, etc. The way GraphBuilder goes about creating a graph is like this:

- Create source filter: This is the easiest part. There's a "File Source" filter that is created for local files.

- Try attaching the output pin of the source filter to any filter already in graph: GraphBuilder goes one by one through all the filters that are already in the existing filter graph, looks for an input pin on it, and tries to connect the source output to it. You'll notice that in the log file created by our media player, no other filters exist besides the source, so it tries to connect to itself, which of course fails (source filters don't even have an input pin). If a connection is made, then GraphBuilder repeats this step using the output pin of the newly connected filter, otherwise, it moves on to the next step.

- If no connection was made, create new filters and try them: If GraphBuilder couldn't make a connection, it creates a new filter, adds it to the graph, and tries to connect the source output to it. How it decides which filter to create is something I don't know. Possible could do with media types, but i'm not sure. If a connection is made, it continues back at the last step, trying to connect the output pin of the new filter to all filters already in the graph.

- If the current filter has no output pin, then an appropriate filter graph has been successfully built.

This works well for most standard media types that you wish to play. If you want to use transform filters or filters to handle special media types, you will have to create part or all of the graph by hand. We'll look at how to do that in a future tutorial.

## Conclusion

Now you should be a little more familiar with controlling DirectShow from C++ and understand some of what goes on in the background. Next time we'll look at creating custom filter graphs by hand how to get video to play in your own window.

One last thing I forgot to mention. In order to run the programs in these tutorials you must have the DirectShow runtime installed. If you have the SDK installed then you're all set, but if not, you must download the runtime from Micrsoft (unless another program you use has already installed it, such as the newer Microsoft Media Player).

*Return To The Tutorial Index*