

## DirectShow and C#: The final frontier ;-)

I've mentioned before that we won't write DirectShow filters in C#. You would have to be pretty brave to tackle such an undertaking.

[UPDATE: I've described a way to write DirectShow filters in C# in my blog entry of November 3th, 2005]

On the other

hand, we have seen, that by using DirectShowLib (or a custom library), you can exploit much more of DirectShow's functionalities than you would have been able with only the Automation-compatible interfaces provided to a language like VB.

In this tutorial, we'll look at a DirectShow interface, ISampleGrabber, and see that sometimes the functionalities provided by a filter can be coded relatively easily in C# through this interface. As the title says, this is probably the final frontier that can be reached with C#. Beyond that, you'll have to return to standard COM and C++ programming for your DirectShow needs, but it is surprising how much you can achieve with just a little C#. Again a zip file with the code is provided.

The ISampleGrabber interface provides a way for an application to access the data that flow through the filter graph as it processes the different streams that make up an application. This interface is accessible from C# and we can manipulate the bits to create different effects.

This tutorial implement the functionality provided by the EZRgb24 filter sample. Instead of writing a filter, we'll use the ISampleGrabber interface to produce the same effects.

First, we'll make our form implements the ISampleGrabberCB interface:

```
{geshibot lang="csharp"}
public class Form1 : Form, ISampleGrabberCB
{/geshibot}
```

The ISampleGrabberCB interface has only two methods that can be called when the filter graph has a sample (typically a frame in a video application) to hand over to the SampleGrabber filter which is the DirectShow filter that exposes the ISampleGrabber interface. That is: in order to use the ISampleGrabber in an application, we create a filter graph then add the DirectShow filter SampleGrabber (through which we access its ISampleGrabber interface). Then, we have to configure the interface for our task. So, let's do that:

```
{geshibot lang="csharp"}
// variable declared in Form1
ISampleGrabber sb = null;
....
// then in BuildGraph, we have
sb = (ISampleGrabber)new SampleGrabber();
ConfigSampleGrabber( sb );
gb.AddFilter( (IBaseFilter)sb, "SampleGrabber" );
{/geshibot}
```

We add a variable member to our form then we define a SampleGrabber object and access its ISampleGrabber interface. We call our ConfigSampleGrabber method and add the filter to the graph. Configuring the SampleGrabber consists in defining the media type we want it to handle and calling the method SetMediaType. We follow this by setting the method that will be called for every samples (using the method SetCallback).

```
{geshibot lang="csharp"}
AMMediaType media;
// set the media type
media = new AMMediaType();
media.majorType = MediaType.Video;
media.subType = MediaSubType.RGB24;
media.formatType = FormatType.VideoInfo;
// that's the call to the ISampleGrabber interface
sb.SetMediaType( media );
DsUtils.FreeAMMediaType(media);
media = null;
// we use BufferCB (i.e. 1) because we couldn't make SampleCB work
sb.SetCallback( this, 1 );
{/geshibot}
```

The SetCallback method takes an object of type ISampleGrabberCB for its first argument. Since our form will implement this interface, we can use this as our argument. The second argument is 0 or 1 to indicate which one of SampleCB or BufferCB the DirectShow runtime should call to process the sample.

As pointed out in the comment, we use the ISampleGrabberCB method BufferCB in our call to SetCallback because we haven't succeeded in using SampleCB. The SampleCB is a little more general than BufferCB so the approach described in this tutorial would be more versatile if we could use SampleCB. After having called ConfigSampleGrabber, we call RenderFile to build the rest of the graph using Intelligent Connect (the doc says not to assume this call works as expected but our conditions on the media types accepted by our instance of the SampleGrabber filter are very common and the chances of Intelligent Connect building the wrong graph are small). Then the code that follows (in BuildGraph) should be familiar by now:

```
{geshibot lang="csharp"}
bv = (IBasicVideo)gb;
bv.GetVideoSize( out videoWidth, out videoHeight );
GetStride();
vw = (IVideoWindow)gb;
vw.put_Owner( (IntPtr)this.Handle );
vw.put_WindowStyle( WindowStyle.Child |
    WindowStyle.ClipSiblings |
    WindowStyle.ClipChildren );
Rectangle rc = this.ClientRectangle;
vw.SetWindowPosition( 0, 0, rc.Right, rc.Bottom );
{/geshibot}
```

We get the video size then we call the method GetStride to retrieve the stride of the sample.

```
{geshibot lang="csharp"}
// excerpt from GetStride()
// GetConnectedMediaType retrieve the media type for a sample
sb.GetConnectedMediaType( media );
```

```
// save the stride
VideoInfoHeader videoInfoHeader = (VideoInfoHeader)
    Marshal.PtrToStructure( media.formatPtr, typeof(VideoInfoHeader) );
videoStride = videoWidth * (videoInfoHeader.BmiHeader.BitCount / 8);
{/geshibot}
```

First, we call `GetConnectedMediaType` to retrieve the media type of the sample we'll process. Then we get the `VideoInfoHeader` by calling `Marshal.PtrToStructure` with the value `formatPtr` from the media type. The stride is deduced from the value tucked away in the `BitmapInfoHeader` member of the `VideoInfoHeader` structure. Once this is done, `BuildGraph` just calls `Run` to run the graph and process the samples in our callback which starts like:

```
{geshibot lang="csharp"}
unsafe int ISampleGrabberCB.BufferCB( double SampleTime, IntPtr pBuffer, int BufferLen )
{
    byte *b = (byte *)pBuffer;
    int x, y;
    switch( effectRequested )
    {
    ....
{/geshibot}
```

First, we use "unsafe" code because we want to directly access the bits of our sample, we cast the `pBuffer` argument to a pointer to a byte and make a switch on the effect requested. The code for each effect is a little different and we'll only look at the xor'ing of the bits that make up the bitmap.

```
{geshibot lang="csharp"}
case Effects.Xor :
for ( x = 1; x <= videoHeight; x++)
{
    for ( y = 0; y < videoStride; y++)
    {
        *b ^= 0xff;
        b++;
    }
    b = (byte *)pBuffer;
    b += (x * videoStride);
}
break;
{/geshibot}
```

For each column and row of pixels, we "xor" the bits of each color using the "and" operator and the value "0xff", then move to the next color. At the end of a row, we move to the next row of pixels color by using the `videoStride` value.

The menu items, under the "Options" menu, are just used to set the variable `effectRequested` which define which pixels processing loop will be executed in our callback.

The ISampleGrabber interface provides a way to peek into the stream of data and modify it "on the fly". This is a very powerful mechanism and, when the restrictions imposed by this interface are respected, is very easy to access from C# (compare this tutorial code with the C++ code for the EzRbg24 filter sample in the SDK). The ISampleGrabber has limitations and there is a "GrabberSample" filter in the samples section of the SDK that improved on the ISampleGrabber interface but this sample filter never made it into an "official DirectShow supported" filter. There is an informative document entitled "grabber\_text" in that sample directory that explains many details about writing filters (in C++) in general.