Using Multiple Graphs in DirectShow

Geraint Davies

Introduction & Overview

DirectShow uses a graph of connected filters. Each filter in the graph has the same state (stopped, paused or running) and the same clock, and the data flows directly from filter to filter. While this works well for many tasks, there are a number of tasks for which it is not suited. Applications need to start and stop some filters independently of others, and to switch connections between filters dynamically. In this article, and the accompanying code, I want to demonstrate how to solve some of these problems by using multiple, related graphs of filters.

First, some background. Why is it so complicated to change a graph while it's running? The design of DirectShow is intended to make filters as lightweight as possible, to encourage developers to break down large monolithic processing engines into several separate components with no loss of efficiency. This meant allowing each filter to call directly to interfaces on other filters and not insisting on a thread switch for each filter. The result is that a single thread created by one filter can call down through several other filters and block in a number of places. This has created a low-overhead design, but sadly it has also resulted in much of the complexity of the DirectShow architecture. Only by following the strict rules of DirectShow behaviour can deadlocks be avoided.

All the filters in a graph must be in the same state. But the graph itself is a very lightweight object: there is very little additional cost to having the same filters divided between several different graphs. A set of custom filters can then feed the data between graphs, outside of the DirectShow framework, and the graphs can be started and stopped independently. You can do this very straightforwardly without sharing allocators or threads between the graphs: the data is copied into queues in one graph, and then picked up and delivered on a separate thread in a separate graph. However, as well as the copy and thread switch, this scheme can often mean conversion between formats since dynamic format changes cannot be passed on easily.

For my examples, I've used a framework that allows data to be passed directly between graphs. The GMFBridge tool ("*GDCL Multigraph Framework*") allows you to build separate source and render graphs that can be connected together, and when connected, data is passed efficiently from source to render. I want to show how this can be used to solve a few common DirectShow tasks. I'm hoping that in the process of explaining it, I will also be able to shed some light on some of the darker corners of DirectShow.

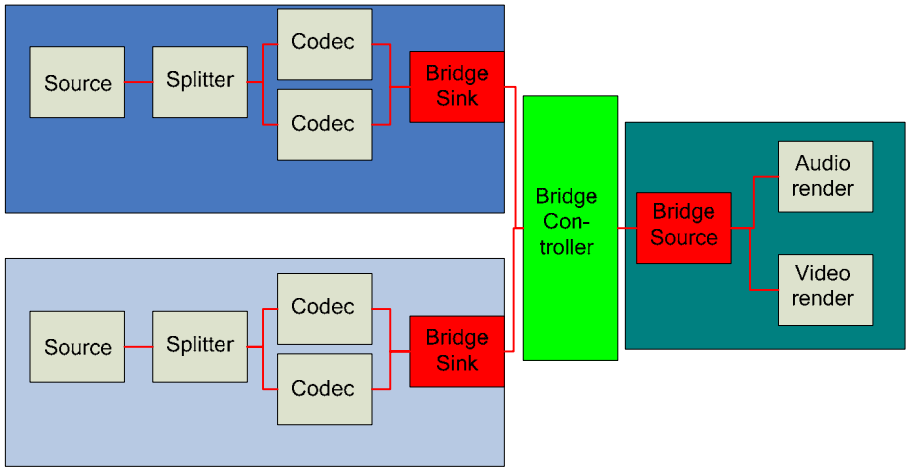
Full source code to the GMFBridge tool and the two example applications is available from http://www.gdcl.co.uk/articles. This version is based on version 1.0.0.8, which includes dynamic changes to video dimensions, and revised input locking.

I'm extremely grateful for the reviews, changes and suggestions provided by Iain Downs, Phil Webster, Thore Karlsen and Stephen Estrop.

Using GMFBridge

To use the GMFBridge architecture, you create a BridgeController, specifying whether you have a video and/or audio stream. You can then create a number of graphs; the controller will insert either sink filters or feeder filters into your graphs. The sink filters consume data at the end of a graph: these are *source graphs*. The feeder filters feed that data into another graph: these are *render graphs*. The controller can then connect one sink filter to one feeder filter at once. The controller can operate in either suspend mode or discard mode: this controls what happens when a source graph is running but not connected: the data received at the sink can be held, causing the graph to suspend, or it can be discarded so that the graph can continue running.

You will get confused about the names; sorry about that! At the downstream end of a source graph, there is a sink filter (class BridgeSink). At the beginning of a render graph, there is a feeder filter that acts as a source filter for that graph; this is the BridgeSource class. The diagram below shows two source graphs and a render graph joined by the Bridge Controller (in green) and the Bridge Sink and Bridge Source filters (coloured red).



Continuous Preview During Capture

I want to solve two different tasks using the GMFBridge toolkit. The simpler of my two tasks is this: how can I keep showing a preview stream from my video capture device, while starting and stopping capture into different capture files?

If you are capturing video to an AVI file using the AVI Multiplexor and File Writer filters, the file is not finalized until the graph is stopped. So if you want to close one file and start another, you need to stop and restart the graph – but since the preview is performed by the same graph, that will cause a visible glitch in the preview window.

So why not put the AVI Mux and File Writer in a separate graph? We can then stop, rebuild and restart those two components whenever we need a new file, without affecting the preview graph. All we need is a way to get the data from the capture output into the new graph. This is done using the GMFBridge custom sink and feeder filters.

The GMFPreview sample app creates a normal video preview graph: a video input device is selected and its preview pin is rendered. Then a BridgeController is created, and it inserts a sink filter into the graph, and the capture pin is fed to the sink filter. Once this is set up, the preview graph is started. Whenever the user wants to capture a segment of video, a capture graph can be built and the controller will bridge the preview graph to the capture graph. When the file is to be closed, the bridge is disconnected and the capture

graph is cleaned up. The controller is set in *discard mode*: this means that when the bridge is not active, data received from the capture pin is discarded so that preview graph continues to run. As soon as the bridge is connected, data from the capture pin will be fed into the capture graph to the AVI Mux and then File Writer.

Seamless Clip Player

My second task is to play a sequence of separate video clips as a single movie, allowing the user to seek about within the movie, so that the breaks between clips are not apparent. The GMFPlay application does this by having a separate graph for each source clip and a single graph containing the renderers. It can then switch seamlessly between source clips as necessary.

This requires a few extra features: the controller signals the app when the source graph reaches the end, so that the app can specify which source clip should come next. Dynamic type changes are passed upstream, so that type changes requested by the video renderer can be implemented in the decoder in another graph. And the allocator is shared, so that buffers produced by the decoders can be fed directly to the renderers when connected.

The controller is in *suspend mode*; this means that when a source graph is not connected, the data is blocked at the sink, ready to play back as soon as it is bridged to the render graph.

The sink filter input pins are configured to accept only uncompressed data. This is to ensure that the compressor is always in the source graph, so clips in different formats can be linked, provided that the same decompressed format can be output. If the clips have different dimensions, the player will attempt to switch the video media type at the video renderer using ReceiveConnection. In the current player, there is somewhat less flexibility in changing audio properties between clips, particularly as regards buffering requirements.

Basic Operations

The Controller

The central component of the bridge is the BridgeController. This COM object is created by the application to manage the connection between graphs, and it in turn creates and manages the other components: the sink and feeder filters, and a BridgeStream object for each pin-to-pin connection.

The controller's IGMFBridgeController interface provides the application with the following methods:

AddStream	Creates a BridgeStream object for one audio or video stream, and specifies options for that stream. The sink and feeder filters will have a pin for each stream.
SetBufferMinimum	The application can gain more time to perform the change-over from one graph to another by increasing the buffering in the source filter using this method, at a cost of increased latency.
SetNotify	Allows the application to receive end-of-segment notifications via a posted Windows message. The end of segment condition is received on worker threads in the sink filter's pins, and this method transfers the processing to the application's main thread.
InsertSinkFilter	Creates a sink filter and inserts it into a graph. The sink filter has an input pin linked to the video BridgeStream, and/or an input pin linked to the audio BridgeStream.

InsertSourceFilter	Creates a feeder filter and inserts it into a graph. The filter has output pins corresponding to each BridgeStream that the controller manages (audio and/or video). The media type negotiation of the output pins needs to refer back (via the BridgeStreams) to the sink filter's pins, so this method requires a graph containing a sink filter to be specified as well.
CreateSourceGraph	Inserts a sink filter into the graph, and then builds a playback graph for the file specified, connecting the decoder outputs to the sink filter's pins. The graph is built using AddSourceFilter and Connect: this is to ensure that only the streams that are required are rendered.
CreateRenderGraph	This method is given a fully-built source graph, and an empty render graph. It inserts a source filter with an output pin for each of the sink filter's input pins, and it renders each of these streams.
BridgeGraphs	Makes a connection between a sink filter (in a source graph) and a feeder filter (in a render graph), disconnecting any previous connection. It can be called with two NULL parameters to disconnect the current bridge. It can also be called with just a sink filter parameter, to enable media type negotiation when building a render graph.
NoMoreSegments	Indicates that the app has no further source graphs to attach, so the EndOfStream signals are passed on to the render graph. The end of segment Notify calls are passed to the application when the EndOfStream calls arrive at the sink filter. But at this point, there will still be some unplayed data queued in the render graph. The *NoMoreSegments* call passes the EndOfStream onto the render graph, so the app can wait for EC_COMPLETE from the render graph before stopping playback.
GetSegmentTime	Returns the current time in the playback graph, as an offset from the beginning of the current source graph. Essentially, returns the current position of the renderers, given in terms of the source graph. The demo application uses this to update its slider position.

The controller has a BridgeStream for each stream to be connected – video and/or audio. The BridgeStream object manages the connection between an input pin in one graph, and an output pin in another graph. It passes calls in both directions to manage media type and allocator negotiation and data delivery.

Media Type Negotiation

When building the graphs, the first thing the pins need to solve is to negotiate a suitable media type. The input pins agree a media type when the source graph is built. But this media type needs to be acceptable to the filters used in the render graph, and to other source graphs that are to be used with the same bridge, and of course the source graph building must be completed before these other graphs have been created.

So the sink input pin negotiates a media type, and when the first render graph is built this can be renegotiated; once that render graph is built, the type is fixed and subsequent source graphs must be able to negotiate the same type.

The steps in the basic type-negotiation process are as follows:

1. When configuring the controller, the application can specify the range of media types to be accepted in the source graph. This can be *uncompressed*, *AVI Mux-compatible* and *any*. The SinkInputPin accepts any types that meet this specification. For example, if *uncompressed* is specified, the sink input pin will reject any media types that it does not recognise as RGB, YUV or PCM audio.

2. When a render graph is being built, the GetMediaType enumeration of preferred output types comes from the output pin upstream of the Sink, and this same output pin is called to QueryAccept a type before connection.

3. If the filter is in suspend mode, then a custom redirecting allocator is used at the sink input pin (this is discussed further below). In this mode, a dynamic media type change can be initiated by the sink to switch to a new media type. So the render output pin is allowed to negotiate a different type provided the upstream output pin's QueryAccept accepts it. This is essential for negotiating efficient surface formats with a video renderer.

4. In discard mode, this type of dynamic change is not possible, so the feeder output pin will only support the type that the sink input pin has already negotiated. This mode is typically used with a capture graph, where the bridged stream is being fed to a multiplexor, and hence the *AVI Mux Compatible* option is sufficient to ensure that the correct type is chosen. It is possible to change the sink input's media type with a *Reconnect* call, but the code currently does not do this.

5. Once the render graph has been built, the bridge media type is fixed. Further graphs after this point will only be allowed to negotiate the type that is set in the bridge. However, in suspend mode, dynamic media type changes originated by (for example) the video renderer are still possible.

Starting with version 1.0.0.5, the GMFBridge code supports clips in different formats. We allow dynamic changes between audio PCM formats, since the renderer can normally handle a dynamic switch like this. If the clips have different video dimensions, we may be able to switch dynamically to that type. This adds an additional complexity that is discussed below.

To switch media types, we use dynamic type changes in the same way that the video renderer does. This means attaching the new media type to a sample returned from GetBuffer. We can only do this if we implement the allocator ourselves, so that the GetBuffer method is implemented in our code. In discard mode, we don't use a custom allocator and so we cannot use dynamic type changes in discard mode. Discard mode is normally used in a video capture environment where downstream-originated dynamic type changes are not particularly important.

When the render graphs are used for video and audio rendering, it's best to build the render graph immediately after the first source graph is built. Then subsequent source graphs can be verified for compatibility with the source and render graphs already built. This is how the GMFPlay demo application behaves.

The GMFPreview application, on the other hand, runs in *discard* mode and the render graph contains the multiplexor and file writer. In this case the correct media type is ensured by the *eMuxInputs* option, which ensures that the sink input only accepts types that are known to be acceptable to the AVI Multiplexor filter.

Allocators and Type Changes

The simplest way to build a bridge like this would be to copy the data received at the sink input into some buffer outside the DirectShow architecture, and then copy it back into *IMediaSample* objects in the feeder source filter on a separate thread. Obviously this

unnecessary copy is somewhat inefficient, but it is much simpler and would work fine for tasks such as the preview/capture application.

What makes this approach fail is the fact that the video renderer initiates dynamic media type changes. These are attached to empty buffers that are given to the decoder to fill. If a type change occurred, any data that was already decoded would need either conversion to the new format or discarding. Of course the conversion may be very simple (bottom-up to top-down, for example), and if you are happy with this restriction, you can bridge multiple graphs with much lower coupling between graphs and far fewer worries about thread synchronisation.

As an aside, the VMR (*video mixing renderer*) does not switch pixel formats once the format has been established. So in this way it is less of a problem than the older renderer, which would switch types readily. However, even the VMR switches types on the first buffer, and may change types when running if (for example), the target rectangle changes.

In any case, the GMFBridge tool goes the whole hog:

- The BridgeAllocator class implements the *IMemAllocator* interface by redirecting the calls to the allocator used in the render graph, when connected.
- The decoder calls the BridgeAllocator's GetBuffer method.
- This blocks until the graph is connected across the bridge, and then passes the call to the SourceOutput pin's allocator (which will typically come from the renderer).
- The BridgeAllocator will watch out for dynamic type changes, which might not otherwise be apparent to the bridge filters, since they are typically cleared from the IMediaSample object before it is passed back downstream.

This means that a thread coming from upstream of the decoder, in the source graph, may be executing code inside a filter in the render graph, and thus thread synchronisation is something that needs to be managed with great care.

The output pins in the render graph ask for the same buffer size and count that were requested in the source graph, except for two cases:

- A video output pin connected to the video renderer may negotiate a different image format to that originally agreed in the source graph. This will have a different buffer size, and so in this case the BridgeSourceOutput pin needs to request buffers of the correct size for the new format. The source graph will be switched to the new format by a dynamic type change on the first buffer.
- You can increase the buffering at the bridge so that there is more time to handle the switch from one graph to another. In this case, the output pin uses the same buffer size but increases the number of buffers requested.

For video capture applications such as the GMFPreview example, we want the source graph to continue when the bridge is not connected. This means that the allocator used at the sink input must function correctly on its own, so in this case we use a standard allocator. We can also accept an upstream allocator in this case. One consequence of this which I've mentioned above is that we can't perform dynamic type changes in this case, but in applications such as video capture, the types are fixed at the source rather than the renderer so this is not as important.

Changing Video Dimensions with ReceiveConnection

For version 1.0.0.5 of the sample code, I have added support for clips of differing dimensions. This adds an extra wrinkle or two to the already complicated type negotiation.

For an upstream filter to switch types when delivering to the video renderer, it is not enough to attach a new media type to the buffer. The buffer size needs to be changed to fit the size of the new video frame, and the buffer allocator is usually controlled by the video renderer. To support this, the video renderer allows the upstream pin to call ReceiveConnection (without disconnecting first). If the type is acceptable and there are no buffers outstanding, the video renderer will switch to the new format. To ensure that the type is acceptable, we keep the same pixel format, and just change the video dimensions. In the GMFBridge code, this switch takes place at the first GetBuffer for the new clip.

Threads and Locking

In general terms, there are two types of thread in DirectShow. Worker threads are created by filters, and they perform the processing and delivery of data downstream, calling methods on the filters downstream and sometimes blocking until the clock reaches a specified time or the graph state changes. State-change threads originate in the application or the filter graph manager, and they instruct the filters to change state or to perform seek operations.

The biggest issue with threads is to ensure that these types of thread interoperate correctly. So for example, when the state change thread is stopping a filter, we need to make sure that the worker threads exit correctly, without deadlocking if the worker threads are blocked. This is helped to some extent by the filter graph manager; a transform filter does not need to worry about worker threads being blocked in a downstream filter because the filter graph manager always stops filters in order, starting downstream and working up to the source. The same behaviour at start time prevents the first samples being discarded if the source is started before the renderer.

Since we have two separate graphs, with worker threads calling across from one graph into the next, we cannot assume that the downstream filters have been started or stopped in the right order. So when we are stopping, the BridgeSinkInput pin must make sure that any worker thread is no longer in the render graph. When we are starting, the BridgeSourceOutput pin must make sure that samples delivered early are held until the graph is active.

BridgeSinkInput locking

At the sink input pin in the source graph, we need to make sure that no thread from upstream of us is active in the downstream graph when we stop. When we are not using the proxy allocator, this is just a question of managing the Receive method. But when we are using the proxy allocator, there is a further problem: if the decoder has got a buffer from the downstream graph's allocator, but has not yet delivered it, then it is not safe to stop. We manage this by holding a semaphore from the GetBuffer call until the end of the Receive call; however, not all GetBuffer calls result in a Receive, so we must watch for discarded buffers and unlock correctly.

Early versions of the software used locking in GetBuffer and Receive to manage this. However, this was prone to error, for example when a filter called GetBuffer multiple times and then delivered all the buffers in a single ReceiveMultiple call. From version 1.0.0.8 of the software, the locking is done using a proxy sample. The proxy allocator allocates an IMediaSample object that is simply a wrapper for the IMediaSample object obtained from the downstream allocator; however, it also holds a lock on the semaphore, which is unlocked when the proxy sample is released.

So the rules for the sink input are:

- The delivery semaphore (the input pin's m_hsemDelivery member) is held whenever the upstream filter holds a buffer from the allocator (from GetBuffer until the buffer is released),

- The delivery semaphore is held in any connect or disconnect operation. Note that when the semaphore is held for a connect or disconnect operation, this is regarded as an exclusive lock and the m_nDeliveryLocks count will be 0.

- When the proxy allocator is not used, the delivery semaphore is not held in GetBuffer, since this does not enter the downstream graph; instead it is locked at the beginning of Receive: this is done by obtaining a proxy sample from the proxy allocator, just to manage the semaphore in a common manner.

- An event handle in the proxy allocator (m_evNonBlocking) is reset when the GetBuffer calls should be suspended: this is only when the allocator is committed and active, but not connected to the downstream graph. In other cases, this event is set so a wait on the event will complete immediately.

BridgeSourceOutput locking

In the source output pin (in the render graph, downstream of the bridge) , there is a timing window because the upstream filter may be started first, which would never happen in a single DirectShow graph. In this case, data may arrive before the pin is fully active. So there is a timing window when the filter is marked as active, but the pin is not ready – for example, it may not have the queue prepared to receive samples.

So we need to make sure that early delivery of samples is suspended until we are ready if the upstream graph is started first. However, we still need to allow the graph to stop correctly without blocking indefinitely in Send. This is managed with a combination of a semaphore and a Boolean flag, following these rules:

- The semaphore (the output pin's m_hsemActive member) is locked by the pin when it is inactive and only released when the filter goes into paused or running state.

- The pin's delivery method (*Send*) acquires the semaphore before delivering downstream, and releases it afterwards. This ensures that early delivery blocks until the pin is ready.

- A Boolean flag m_bActive (protected by m_csActive) is set to true immediately the filter starts going active, and set to false immediately the filter starts stopping. The pin's delivery method returns immediately without acquiring the semaphore if the m_bActive flag is false.