

## OPENMEDIALIB

### Purpose:

This document provides details on the openmedialib (OML) design and implementation.

### Introduction:

OML is designed to provide a standardised wrapper mechanism for multimedia libraries (such as ffmpeg's avformat, quicktime, directshow and others).

Each library is typically wrapped by two interfaces – an *input* and a *store*. [perhaps a note to the effect that input == decode, store == encode]

An input implementation typically generates *frame* objects on request, where a frame provides an *image* and/or a collection of *audio* samples. It typically reports a duration and allows seeking.

A further type, known as a *filter*, is essentially a frame modifier which is connected to 1 or more inputs and behaves as an input itself. It can be used to provide image and/or audio effects.

A *filter graph* consists of a fully connected mix of *filters* and *inputs*.

A store implementation receives frame objects and does something with them – for example, it might encode them, or analyse them, or provide playout functionality.

Additionally, OML provides utility functionality for general image, audio and time manipulations.

### Dependencies:

OML is dependent on boost, openimagelib (OIL) and openpluginlib (OPL).

All input, store and filter objects are introduced as *plugins* – some core, open source (typically, LGPL compliant) plugins are provided in the core OLIB distribution, but subsequent plugins can be developed independently and are not license restricted (though, obviously, distribution of those components is restricted by the licensing involved).

### Usage:

OML provides a C++ interface which can be used directly. There is also a higher level python binding provided which allows simplified/rapid development of applications and test cases.

For the purposes of this document, most sample code relating to usage will be in python.

Sample code relating to development of additional plugins will be expressed in C++.

## Terminology:

*image* – an object which conforms to the OIL *image\_type* interface. Typically, images provided by OML are in the native *colour space* of the source material (codec or usage dependent).

*colour space* – in OIL parlance, a colour space stipulates the make up of the image, both in terms of the sample space used (RGB, YUV/YCbCr etc.) and the packing of the samples (hence, a colour space is RGB, BGR, RGBA, YUV420P etc.).

*audio* – an object which conforms to the OML *audio\_type* interface. At the point of writing, all audio must be coerced to the 16 bit signed integers at the input phase. It is described by the frequency (being the number of samples per channel per second), the number of channels and the number of samples in each channel.

*frame* – an object which provides an image and/or audio. When both are provided, the number of audio samples in successive frame should correlate to a pattern dictated by the frame rate (see utility functionality below).

*frames per second (fps)* or *frame rate* – all inputs must report a valid fps expressed as integer numerator:denominator pairs (for example, PAL 25 fps should be reported as 25:1 and NTSC should report 30000:1001) and provide a frame accurate seek interface where applicable.

*variable frame rates* – some video formats provide support for variable frame rates – in those cases, images are not delivered at fixed durations. It is incumbent on the input implementation to map these to a fixed frame rate (typically by employing a duplicate and drop mechanism for images).

*sample* – depends on the context – it can refer to either discrete audio or image measurements, for example, the value of a colour component at a particular offset in an image.

*sample aspect ratio (sar)* – typically, images coming from video sources do not consist of 'square' pixels – often, they are rectangular, which means that the display resolution has a different width to the resolution provided. As with frame rates, the sample aspect ratio is expressed as a numerator:denominator pair.

*aspect ratio* – the aspect ratio is typically determined by dividing the width by the height in the square pixel scenario (1:1 being the common sar for still images).

*input* – an object which conforms to the OML input interface – it's primary purpose is to provide frames.

*pull/push* – the general model for the input and store interaction is defined as pull and push respectively – applications pull frames from inputs, and push to the store.

*rpn* – Reverse Polish Notation – this is described in more detail below.

High Level Use:

NB: The following relies on code which is currently placed in test/openmedialib/python

The following snippet is a python example that provides a transcoder from an avi file to an mpeg file:

```
import bootstrap
import openpluginlib
import openmedialib

openpluginlib.init( "" )
input = openmedialib.create_input( "input.avi" )
frame = input.fetch( )
store = openmedialib.create_store( "output.mpg", frame )
if store.init( ):
    i = 1
    while i < input.get_frames( ):
        if not store.push( frame ):
            print "failed to push frame"
            break
        input.seek( i, False )
        frame = input.fetch( )
        i += 1
    store.complete( )
```

Note that the the general interface to our input is simply via 'fetch' and 'seek' methods, and a store has a 'push' method.

A more interesting example is in the generation of a filter graph – this example simply composites the video given as an input on to a black background such that the video itself doesn't get cropped:

```
import bootstrap
import openpluginlib
import openmedialib
import stacks
import common

openpluginlib.init( "" )
project = stacks.project( )
rpn = stacks.stack( project )
rpn.push_args( [ "colour:" ] )
rpn.push_args( sys.argv )
rpn.push_args( [ "filter:composite", "@0.out=@1.out" ] )
input = rpn.pop( )
common.play( input )
```

In this example, we introduce two new OML classes – the 'project' and the 'stack'.

The stack provides a non-ambiguous grammar for the construction of complex filter graphs – it is responsible for acquiring and connecting OML nodes. The theory behind this and more examples of use are shown in the following sections.

The project provides a 'pool pattern' (signified here by the withdraw and deposit methods) – this object is responsible for pooling the OML nodes.

Also note that `common.play` is a convenience function which is just a reiteration of the fetch/push loop in the previous example – it attempts to locate a video playout surface (`sdl`, `glew` or `caca` are supported) and an audio playout component (currently, this is restricted to `openal`).

## Reverse Polish Notation

As a race, humans are conditioned to accept certain complexities via a heady mix of social engineering and indoctrination. Mostly, the effects of this are beneficial, but sometimes they lead us to introducing complex solutions where simpler ones exist.

One example of this is in our traditional notation for arithmetic calculations – we all immediately understand:

$$1 + 2$$

We are taught the rules of association, so that for most of us above a certain age, the following is unambiguous:

$$1 + 2 * 3$$

We know that the  $*$  has a higher order of precedence, so the result is  $1 + (2 * 3)$ , not  $(1 + 2) * 3$ . In fact, we need the parenthesis to differentiate between those cases, and in truth, it would be more desirable to enforce the parenthesis usage rather than having implied rules, but humans, particularly intelligent ones, tend to laziness.

Like humans, a computer needs to be taught the associative rules in much the same we are, and thus, the complex conditioning continues into the field of computers.

There are alternative notations which don't have the same ambiguity – Reverse Polish Notation (or RPN) is one such approach.

RPN essentially involves a stack of number and operators, so our initial  $1 + 2$ , becomes:

$$1 2 +$$

When parsing, we start on the right and pop the entry – a  $+$  operator logically requires 2 numbers, so it in turn pops two numbers (being 2 and 1), adds them together and pushes the result on the stack.

Similarly,  $1 + (2 * 3)$  becomes:

$$1 2 3 * +$$

The  $+$  requires 2 numbers, the first pop results in  $*$ , which in turn requires two numbers (2 and 3), leaving 6 as the first operand of the  $+$  and 1 is retrieved for the second, thus we get 7.

Our  $(1 + 2) * 3$  becomes:

$$1 2 + 3 *$$

Thus, there is no ambiguity and no parenthesis are required.

As a final example, consider a maths function like a sine.

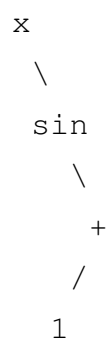
Our traditional arithmetic model gives us expressions like:

$$\sin(x) + 1$$

Which we can express in RPN as:

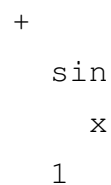
$$x \sin 1 +$$

It should be clear by now that the in-fix expression which we use is just one way to represent any particular expression. In fact, many permutations of the representation are possible – for example, we could represent the above in a manner which is similar to a filter graph (with the result being the outer most right hand node):



In other words, mapping a filter graph to a stack is trivial, just as going from a stack to a filter graph is also trivial – we can also go to an unambiguous infix representation (to avoid the 'educational' requirement we could simply enforce potentially redundant parenthesis in the output).

For consistency, the document adopts a graph representation such that the top left operator is the one from which the application will use and all others will be pulled on demand from that operator:



This is essentially a mirror of the previous graph minus the connectors (and is generated by a simple walk of the generated filter graph).

## Applying RPN to Video/Audio

Arithmetic operations and filter graphs are very similar.

To illustrate this, we will start with a fairly simple graph which simply applies a filter to a video:

```
<filter>  
  <input>
```

The first line here is the node which the application reads from – its first (and in this case, only) slot is connected to the <input> node.

It should be clear that this is similar to our sine case in the previous section and thus, our rpn specification is simply:

```
<input> <filter>
```

This isn't particularly interesting in itself, but if we take a binary operator like a 'composite' (which is analogous with a +, \* or / operator), we end up with a more interesting graph:

```
<composite>  
  <background>  
  <foreground>
```

which maps (rather unsurprisingly) to:

```
<background> <foreground> <composite>
```

Or in combination with a filter on the foreground:

```
<composite>  
  <background>  
  <filter>  
  <foreground>
```

we get:

```
<background> <foreground> <filter> <composite>
```

These examples should show that RPN does indeed provide an unambiguous grammar for filter graph construction, but it would be understandable if there is still a 'so what?' question hanging – so far, all we have focused on is ensuring that RPN is non-restrictive, but we haven't touched on the advantages (other than the associative rule aspect).

Well, the advantages are subtle – for example, in order to construct our graph, we simply need push and pop methods:

```
stack.push( background_info )
stack.push( foreground_info )
stack.push( filter_info )
stack.push( composite_info )
return stack.pop( )
```

NB: the pushes on to the stack can be strings which describe the operator required (ie: a filename or the name of filter [preceded with 'filter:']) or a preconstructed oml operator or a partially constructed filter graph or a modifier for the current operator [strings which take the form of name=value] or a callback specification.

This is comparable to a more specifically crafted bit of code:

```
composite = create( composite_info )
foreground = create( foreground_info )
background = create( background_info )
filter = create( filter_info )
composite.connect( background, 0 )
composite.connect( filter, 1 )
filter.connect( foreground, 0 )
return composite
```

Obviously, there are more lines of code to carry out the same task, and that complexity will increase as the number of filters and inputs are increased. To be fair, you can reduce the lines of code, but you would not be able to reduce the number of function calls or variable assignment (at least, not by much, and not without making the code even more brittle and dense).

Whilst the loc count is debatable, the larger issue lies in the fact that the code itself is brittle – changing a connection in a non-trivial graph is a non-trivial operation (as an exercise, try extending both to include a second filter between the foreground and the composite and compare the lines of code added and modified – it should be immediately obvious which one 'wins').

The following sections detail some of the filters which are currently available.

## Temporal Modifications

If the filter graph is constructed with a single input, the resultant duration of the filter graph is clearly the duration of the input.

However, if you wish to trim a video, you can use the 'temporal' filter as follows:

```
<input> filter:temporal in=100 out=200
```

This of course assumes that the input is seekable, but of course, if an input isn't seekable, it's not much use in a video editing environment :-).

Temporal also provides a mechanism to do on the fly frame rate conformance:

```
<input> filter:temporal in=100 out=200 fps_num=25 fps_den=1
```

NB: this is unimplemented at this point. The in/out properties are currently expressed in terms of the source frame rate, but a property modifier will allow this to be specified as the output frame rate.

## Upstream Properties

A more complex problem related to length lies in the previously demonstrated composite filter.

Here we have:

```
colour: <input> filter:composite
```

The duration here is rather arbitrarily selected as the duration of the 0<sup>th</sup> slot – this being the colour.

We can enforce a duration of the colour: input by specifying it's out point:

```
colour: out=1000 <input> filter:composite
```

Depending on the requirements of the filter graph, this might be fine, but typically, it would be nice to force the background duration to be that of the foreground. In order to do that, we would typically need to know the duration of the foreground. Taking a naïve approach, we can do the following:

```
rpn.push( "colour:" )
rpn.push( input_info )
input = rpn.pop( )
rpn.push( "out=" + str( input.get_frames( ) ) )
rpn.push( input )
```

This is workable, but inelegant.

OML's RPN provides an alternative mechanism for property retrieval – if we consider our graph as:

```
filter:composite
  colour:
    <foreground>
```

To the composite, the colour: is connected on slot 0 and the foreground on slot 1. As we've mentioned above, the colour: input has a property called 'out' which we can specify to control the duration of the colour. Further to that, all rpn popped and connected nodes expose an 'rpn\_length' property. In this particular case, we can force the colours out to equal the foreground's rpn\_length, using:

```
colour: <foreground> filter:composite @0.out=@1.rpn\_length
```

Note that this particular example only works with colour and it relies entirely on the fact that the out property is applicable to that node, and that there are no filters associated to the colour – if there were a filter, then we would need to extend our indexing to dig deeper into the graph:

```
colour: filter:something <foreground> filter:composite @0.0.out=@1.rpn\_lengththis assigns
```

the value of the length attribute on the input element to the out attribute on colour

## Playlists

If you want to play a collection of video (or audio) out in sequence, then you can use the playlist filter.

An example of this would be:

```
<video1> <video2> filter:playlist slots=2
```

which would provide a graph like:

```
filter:playlist  
  <video1>  
  <video2>
```

The duration is the sum of the duration of the inputs.

Note that no frame rate, image size, audio frequency, colour space etc conformance is stipulated here – whether mixed formats is usable relies entirely on the capabilities of the recipient. You can enforce conformance by attaching the requisite filters to each of the inputs (or, perhaps on the playlist itself? needs to be checked).

## Stack Manipulations

This is a small deviation for the filter descriptions since it defines a collection of operations which you can carry out on the RPN stack.

These operations are inspired by the standard collection provided by Forth, but deviate somewhat since we aren't quite so constrained when operations occur – since we're creating filter graphs, we can break up the graph on a whim (see Decapitation as an example).

None of these are strictly necessary since they can all be carried out via push and pop operations with the exception of clone and decap since they need to walk through a filter graph (see walk operations below).

Swapping operations:

```
<input1> <input2> swap == <input2> <input1>
<input1> <input2> <input3> <input4> 2swap == <input3> <input4> <input1> <input2>
```

Rotations:

```
<input1> <input2> <input3> rot == <input2> <input3> <input1>
```

Dropping:

```
<input1> <input2> drop == <input1>
<input1> <input2> 2drop ==
```

Duplicating:

```
<input1> dup == <input1> <input1>
```

NB: This creates a second reference to the same input – it might not be what you want.

Cloning:

```
<input1> clone == <input1> <input1'>
```

NB: dup creates a reference, clone clones the entire input (hence, if you only want the same frame in both eventual uses, use dup – if you want to seek to different parts, use clone).

TODO: Check safety issues related to dup – it might be necessary to recreate the frame objects before updating...

Over:

```
<input1> <input2> over == <input1> <input2> <input1'>
```

NB: this invokes clone rather than dup.

Nip:

<input1> <input2> nip == <input2>

Tuck:

<input1> <input2> tuck == <input1> <input2> <input1'>

Assignment

<input> blah ! <something> blah == <something> <input>

Decapitation:

<input1> <input2> filter:composite decap == <input1> <input2>

NB: This is, of course, impossible in arithmetic terms (ie: 1 2 + decap is nonsensical – you can't go back from 3 to 1 2), but for our purposes, it is both possible and probably desirable.

Debug:

<...> debug == <...>

This outputs a 'stack dump' – it shows all the contents of the current stack (equivalent to the Forth's .s word).

Render:

<...> render == <..>

Another debug operation - this 'draws' a graph of the top of stack input.

## Aspect Ratio Considerations

Throughout the document so far, we have used the by now familiar:

```
colour: <input> filter:composite @0.out=@1.rpn_length
```

stack entries, but other than stating that it's an example of a binary operator no explanation has been offered as to what it really does, or what you can do with it.

This is however, possibly one of the most critical structure in the field of video editing and encoding. To truly understand what it's all about, we need to step back into the dirty laundry of the video world...

The problem is that video images dimensions typically don't conform to pixel sizes on a computer monitor – since computer monitors typically consist of square pixels – hence, if you take a JPEG from a digital camera and its size is 640x480, to fully display it, you would need precisely 640x480 of screen real estate.

In the world of video, the rules change – typically, pixels are rectangular. And the dimensions of that pixel rectangular change need to be applied to each 'sample' in order for them to be displayed correctly. This additional information is known as 'sample aspect ratio' or 'sar'. In OML, sar consists of a pair of numbers and is annotated as num:den (or numerator and denominator). These numbers dictate the ratio of width to height of each sample.

Hence, the JPEG from the digital camera has a sar of 1:1. Note that it also has an 'aspect ratio' of 4:3, being the ratio of width to height.

NB: To allow for non-square pixel monitors, there really should be a 'display aspect ratio' or 'dar' num:den pair – for the purposes of simplification, we will consider non-square dar's as a specific problem of the user of the frame.

So, what values of sar's are used?

Unfortunately, that is entirely dependent on the video input. But for the purposes of this document, we will focus on some specific cases. These are the image dimensions and sar for DV flavours:

```
DV PAL 4:3 – 720x576 @ 59:54  
DV PAL 16:9 – 720x576 @ 118:81  
DV NTSC 4:3 – 720x480 @ 10:11  
DV NTSC 16:9 – 720x480 @ 40:33
```

And yes, those are correct – 4:3 and 16:9 have precisely the same number of samples per image – the difference is purely in the way that they are captured and presented. In the wide screen case, the samples are simply stretched horizontally.

So how do we use this information?

Let's assume that our only requirement at this point is simply to display the image on a computer monitor – in order to do that, we need to convert the dimensions to a 1:1 image.

The following equation can be used:

$$\text{realwidth} = ( \text{height} * \text{width} * \text{sar\_num} ) / ( \text{height} * \text{sar\_den} )$$

Note that we scale horizontally, so the height given is the height used.

This gives us some interesting results, and shows another flaw in the system. First of all, the PAL DV 4:3 figures give us:

$$\text{realwidth} = ( 576 * 720 * 59 ) / ( 576 * 54 ) = 786.6666666 = \text{approx } 787$$

This is at odds with what we might expect from the 4:3 description – the 4:3 there should denote the ratio between width and height, therefore, we would expect that the sar usage and the following would match:

$$\text{readlwidth} = \text{height} * 4 / 3 = 768$$

The reasons behind this are twofold – first up, the sar is actually an approximation – in this case it's closer to 1094:1000 which gives us:

$$\text{realwidth} = ( 576 * 720 * 1094 ) / ( 576 * 1000 ) = 787.68 = \text{approx } 788$$

Now we now have a 20 pixel discrepancy. And oddly, that is now exactly correct which brings us to the second point - a TV screen does not show the outer left 10 pixels on both sides – hence, only the central 768 are visible.

Similar rules and approximations apply to the other examples above.

NB: OML ignores the pixel discrepancy and ends up with an ever so slightly distorted image. It could be easily corrected if the sar's were more accurately provided, so the view taken is simply that – correction is made from the outside.

How does all of this relate to the original colour: <input> composite?

All frames in a video need to conform to a particular resolution and sample aspect ratio. For example, if you want to generate a PAL DV stream, all images must be scaled to 720x576 @ 59:54.

So, if we take our 640x480 1:1 JPEG, we obviously need to scale it – but if we just blindly scale to 720x576, it's going to be completely wrong. Ultimately, we want the 640x480 image which has a 4:3 width/height ratio to fill the central 768x576 pixels of the TV – right?

In our examples thus far, preserving aspect ratio is essentially what the composite filter does.

Providing the target resolution and sar is the role of the colour: (or in fact, any other background which is used to replace it).

The colour: input has properties which allow these settings to be specified – namely, width, height, sar\_num and sar\_den. These default to the PAL 720x576 59:54 settings. It also has

properties to allow the specification of the colour itself – obviously, in this case, we want black which is also the default.

To calculate the dimensions of the foreground such that it fits vertically within the dimensions of the image, we apply the following:

$$w = ( bg\_h * fg\_w * fg\_sar\_num * bg\_sar\_den ) / ( fg\_h * fg\_sar\_den * bg\_sar\_num )$$
$$h = bg\_h$$

Hence in our 640x480 1:1 foreground on to a 720x576 59:54 background, we get:

$$w = ( 576 * 640 * 1 * 54 ) / ( 480 * 1 * 59 ) = 702.9152 = \text{approx } 703$$
$$h = 576$$

This mode of compositing is known as 'pillarbox'.

To calculate the dimensions of the foreground such that it fits vertically within the dimensions of the image, we apply the following:

$$w = bg\_w$$
$$h = ( bg\_w * fg\_h * fg\_sar\_den * bg\_sar\_num ) / ( fg\_w * fg\_sar\_num * bg\_sar\_den )$$

Again, using our 640x480 1:1 foreground on to a 720x576 59:54 background, we get:

$$w = 720$$
$$h = ( 720 * 480 * 1 * 59 ) / ( 640 * 1 * 54 ) = 590$$

This compositing is known as 'letterbox'.

The default behaviour of the composite is to rescale the foreground such that it is placed centred on to the background such that it's cropped neither vertically or horizontally. This is simply done by calculating one of the above and ensuring the computed value is less than or equal to the corresponding dimension of the background – if it's greater, then use the other computation. Finally, the image is centred and composited.

OML dubs this mode of compositing 'fill'.

The 'mode' of compositing is also a property of the composite filter – mode can take the values of full, letterbox, pillarbox, native and distort. The last two operations preserve the dimensions of the original [and hence, crop or contain padding where necessary] and distort the foreground to fill the background respectively. Any value other than those cause it to behave as a distort.

References:

<http://www.bbc.co.uk/commissioning/tvbranding/picturesize.shtml>  
<http://lipas.uwasa.fi/~f76998/video/conversion/>

## The Encoding Filter Graph

While image scaling and aspect ratio are crucial considerations from the encoding cycle, that is only part of the problem of encoding.

A couple of other factors play their part in the process – frame rate and audio sampling also need to be considered.

OML provides some additional filters for these and the following general form of a filter graph is used to encode any number of clips to fixed frame rate:

```
composite
  colour:
  deinterlace
  playlist
  wrapper
    temporal
    resampler
    <input1>
  wrapper
    temporal
    resampler
    <input2>
```

NB: wrapper is a temporary filter which carries out frame rate conversion via a drop and dupe approach – proper pulldown filters can be implemented.

Note that this 'all in one' graph isn't strictly necessary, and might not be ideal in all cases – in its favour it has the following advantage - by constructing one graph, there is a known number of frames for the entire content, and progress can easily be shown for the whole.

From a resource utilisation point of view, it might be better to consider creating a graph for each clip and thus, progress can be shown for each clip (or if the durations of the inputs are known in advance, progress can be computed for the whole). As shown in a later section, the breakdown of large multitrack projects works in much the same way.

As should be clear from the details previously, the full graph can be constructed using the following pseudo python code which assumes in/out points are correctly provided:

```
rpn.push_args( [ "colour:", "fps_num=25", "fps_num=1", "width=720", "height=576",
"sar_num=59", "sar_den=54" ] )
count = 0
for v in videos:
    rpn.push_args( [ v.file ] )
    rpn.push_args( [ "filter:resampler", "frequency=48000", "channels=2" ] )
    rpn.push_args( [ "filter:temporal", "in=%i" % v.in, "out=%i" % v.out ] )
    rpn.push_args( [ "filter:wrapper", "fps_num=25", "fps_num=1" ] )
    count += 1
rpn.push_args( [ "filter:playlist", "slots=%i" % count ] )
rpn.push_args( [ "filter:deinterlace", "filter:composite", "@0.out=@1.rpn_length" ] )
```

Note the frame rate properties are stipulated at multiple points in the graph – the net result is that each video file is 'normalised' individually and independently before being composited on to the single background node.

This should also bring up a fundamental point about the rpn logic – although it has its roots firmly in the Forth language, it makes absolutely no attempt to replicate Forth's flow control logic – this is provided by the hosting language (in this case, Python).

Also note – this graph is built 'blind' – we don't know and don't care if it consists entirely of files which contain both audio and video or not, and the same graph can be used to output audio or video only (without the additional processing requirements of the superfluous component).

Caveat: the last paragraph has revealed a bug in the current encoder – mixtures of inputs with missing components are going to cause a failure...

In terms of use, the following essentially reiterates the loop from the opening hello world example but shows where the process request would be placed:

```
input = rpn.pop( )
input.set_process_flags( openmedialib.audio | openmedialib.image )
frame = input.fetch( )
store = openmedialib.create_store( "output.mpg", frame )
if store.init( ):
    i = 1
    while i < input.get_frames( ):
        if not store.push( frame ): break
        input.seek( i, False )
        frame = input.fetch( )
        i += 1
    store.complete( )
```

## Threading

A subgraph can be threaded using the threader filter. For example:

```
<input> filter:chroma filter:threader
```

Once activated, the threader carries out a readahead operation.

TODO: Explain this and provide the activation/deactivation logic...

## Putting it All Together

It is acknowledged that the above is rather abstract, so let's continue by making it a bit more concrete. We'll take a traditional video editor as our application and model it into the RPN filter graph system:

```
track 1: +-----+                                     +-----+
          | input1                                     |input3   |
          +-----+                                     +-----+
          | filter                                     |filter   |
          +-----+                                     +-----+

track 2:           +-----+
                  | input2                             |
                  +-----+
```

We interpret it as 4 filter graphs:

```
/--- FG0 ---//--- FG1 ---//----- FG2 -----//----- FG3 -----/
```

We also typically composite the results on to a background or canvas (typically, this is just a black image – you can picture this as being introduced as track 0 and it'll stretch for the duration of the longest track [track 1 in this case]).

Internally, we hook up our graphs as follows:

FG0: bg input1 filter:filter filter:composite

FG1: bg input1 filter:filter input2 filter:composite filter:composite

FG2: bg input2 filter:composite

FG3: bg input3 filter:filter filter:composite

Notes:

- 1) the FGs need to be constructed on demand (ie: when you cross from one section to the next)
- 2) in order to satisfy 1, we need to be able to construct and deconstruct the graphs to reuse the same objects
- 3) the objects in the multitrack are not the physical nodes - they're proxies which allow the identification of the node

Interfaces:

input\_type:

```
// Property object
pcos::property_container properties( )

// Basic information
virtual const openpluginlib::wstring get_uri( ) const = 0;
virtual const openpluginlib::wstring get_mime_type( ) const = 0;
virtual bool has_video( ) const = 0;
virtual bool has_audio( ) const = 0;

// Audio/Visual
virtual void get_fps( int &num, int &den ) const = 0;
virtual int get_frames( ) const = 0;
virtual bool is_seekable( ) const = 0;

// Visual
virtual void get_sar( int &num, int &den ) const = 0;
virtual int get_video_streams( ) const = 0;
virtual int get_width( ) const = 0;
virtual int get_height( ) const = 0;

// Audio
virtual int get_audio_streams( ) const = 0;

// Set video and audio streams
virtual bool set_video_stream( const int ) { return false; }
virtual bool set_audio_stream( const int ) { return false; }

// Default number of frames
virtual double fps( ) const
virtual double get_sar( ) const

virtual double get_duration( )
void set_process_flags( int flags )
int get_process_flags( )

// Default seek functionality
virtual void seek( const int position, const bool relative = false )
virtual void seek_time( const double time, const bool relative = false )
virtual int get_position( ) const

// Virtual frame fetch method
virtual frame_type_ptr fetch( ) = 0;

// Allow the app to refresh a cached frame
virtual void refresh_cache( frame_type_ptr ) { }
```

store\_type:

```
// Property object
pcos::property_container properties( )

// Initialise method
virtual bool init( )

// Push a frame to the store
virtual bool push( frame_type_ptr ) = 0;

// Tell the store to flush all pending frames, while returning the
// one that was pending. The default implementation applies when
// the store doesn't queue frames.
virtual frame_type_ptr flush( )

// Playout all queued frames and return when done. The default
// implementation applies when the store doesn't queue frames.
virtual void complete( )

// Allows the store to dictate when it is running empty (ie: any
// realtime store such as an audio player or device feed needs more
// frames in order to provide smooth playout)
virtual bool empty( )
```

filter\_type: