

GStreamer

John M McIntosh

<http://www.smalltalkconsulting.com>

Maintainer, Squeak Macintosh VM

Sophie Authoring Tool, storage & media subsystem code.

In progress

Squeak VM iPhone Port.

Squeak VM Macintosh OS-X (10.5+) re-engineering
Macintosh Hydra support

GStreamer

<http://www.gstreamer.net/>

GStreamer is a library for constructing of graphs of media-handling components. The use cases it covers range from simple Ogg/Vorbis playback, audio/video streaming to complex audio (mixing) and video (non-linear editing) processing.

It's a mesh network of data flow objects. Obvious uses are read a media file, decode and play the resulting audio and video to your audio/video hardware.

Non-obvious uses are using the network aware media get element to stream data into the file system put element thus enabling the downloading of internet files to the local file system.

Codec elements are broken into three major groups, Base, Good, Ugly. Where each category defines issues with stability and licensing (or lack there of). All work and testing was done with 'Base' which is approved for OLPC

GStreamer

<http://www.squeaksource.com/GStreamer.html>

The GStreamer plugin for Unix systems was written via funding from Viewpoints Research Institute, Inc. to replace the Squeak Mpeg plugin, thus providing an patent un-encumbered media player for the OLPC. The target media is OGG, but one in theory could load the MPEG or other media codec on an OLPC enabling the playback of other forms of media.

Platform Status:

Windows,	in progress via Qwaq
Macintosh OS-X,	need unix expert to build libraries
OLPC,	available
Linux,	available (in theory)

GStreamer

The plugin and supporting Smalltalk code is licensed via

- * Created by John M McIntosh on 3/10/08.
- * Copyright 2008 Corporate Smalltalk Consulting Ltd.
- * <http://www.smalltalkconsulting.com> All rights reserved.
- * Written for Viewpoints Research Institute <http://www.vpri.org/>
- * <http://www.opensource.org/licenses/mit-license.php>

GStreamer

SmartSyntaxInterpreterPlugin subclass: #GStreamerPlugin

About 100 primitive calls to support **a subset** of the GStream C api

Written for 32 bit systems, it could be altered for 64 bit systems, volunteers can submit code for that. The GStreamerPlugin class contains 99% of the SLANG code, there is a small percentage of C code to support the ability of Squeak to push data into a fake GStreamer element source, and to pull data from a fake element sink.

The Smalltalk code base.

SUnits for every method publicly exposed and each primitive call, plus various test scenarios

GStreamer

The architecture is subclassed off of GStreamerObject

The screenshot shows a 'Hierarchy Browser: GStreamerObject' window. The left pane lists the class hierarchy starting from 'ProtoObject' and 'Object', with 'GStreamerObject' highlighted. The middle pane shows methods like 'accessing', 'finalize', 'get value', 'set value', 'initialize-release', 'state', 'system primitives', and 'testing'. The right pane lists methods such as 'actAsExecutor', 'finalize', 'getKeyValueBoolean:', 'getKeyValueDouble:', 'getKeyValueFloat:', 'getKeyValueLongLong:', 'getKeyValueLong:', 'getKeyValuePointer:', 'getKeyValueString:', 'getKeyValueULongLong:', 'getKeyValueULong:', 'getSimplifiedState', 'getSmalltalkObject:', 'getState:', 'handle', 'handle:', 'isHandleSane', 'noCheckHandle', 'noCheckHandle:', 'objectUnref', and various 'primobjectget...' methods. Below the panes are buttons for 'browse', 'senders', 'implementors', 'versions', 'inheritance', 'hierarchy', 'inst vars', and 'class'. The bottom section shows 'Object subclass: #GStreamerObject' with instance and class variable names, pool dictionaries, and category. It also includes a note about 'handle' and 'Register' and a copyright notice for John M McIntosh.

```
Object subclass: #GStreamerObject
  instanceVariableNames: 'handle'
  classVariableNames: 'Registry'
  poolDictionaries: ''
  category: 'GStreamer-Base'
```

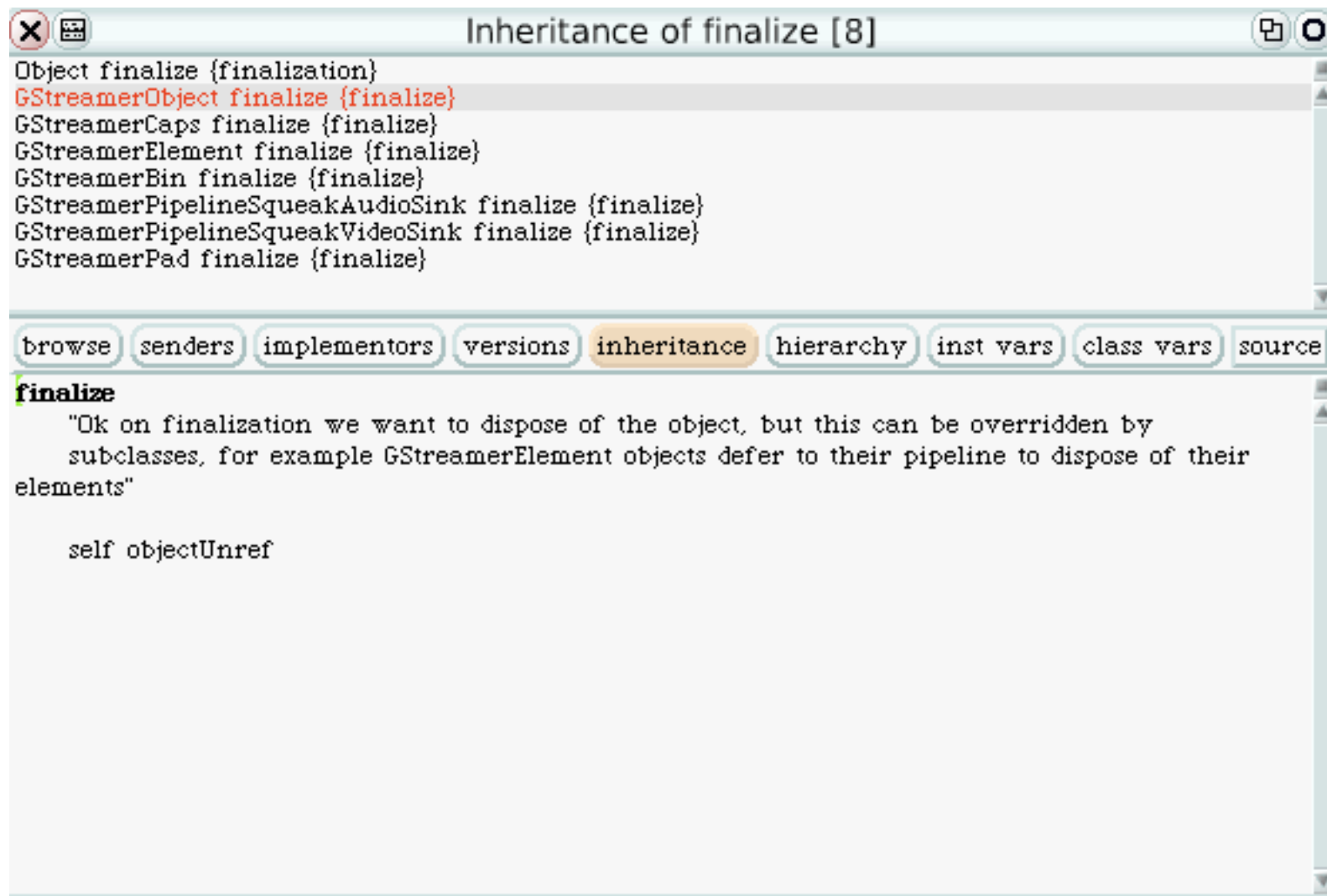
handle is a memory address for a GStreamer Object
Register is the Weak Registry to enable our ability to deal with GCed Objects

* Created by John M McIntosh on 3/10/08.
* Copyright 2008 Corporate Smalltalk Consulting Ltd. <http://www.smalltalkconsulting.com> All rights reserved.
* Written for Viewpoints Research Institute <http://www.vpri.org/>
* <http://www.opensource.org/licenses/mit-license.php>

GStreamer

GStreamer supports finalization/GC on the C side so we have to be careful on the Smalltalk side that GC finalize does not trigger a double C memory free call. So we have painfully created an inheritance tree on the Smalltalk side, since freeing a pipeline for example frees elements, but not say explicit dynamically created pads.

Complex rules, but seems correctly implemented?



```
Inheritance of finalize [8]
Object finalize {finalization}
GStreamerObject finalize {finalize}
GStreamerCaps finalize {finalize}
GStreamerElement finalize {finalize}
GStreamerBin finalize {finalize}
GStreamerPipelineSqueakAudioSink finalize {finalize}
GStreamerPipelineSqueakVideoSink finalize {finalize}
GStreamerPad finalize {finalize}
```

browse senders implementors versions inheritance hierarchy inst vars class vars source

finalize
"Ok on finalization we want to dispose of the object, but this can be overridden by subclasses, for example GStreamerElement objects defer to their pipeline to dispose of their elements"

self objectUnref

GStreamer

A element is either a source or sink, or both. You instantiate a GStreamerElement (or subclass), this element then is tied to a instance of the C code GStreamerElement subclass.

"Make a Volume control element"

"This element modifies the data stream to adjust the sound volume"

```
volume := GStreamerElement elementFactoryMake: 'volume' name: 'volume'.  
self should: [volume isHandleSane]. "check C handle for sanity"
```

You then create a GStreamerPipeline or Bin which will contain the elements, and then you connect the items by indicating how the individual Pads on each object should connect to the objects before and after in the pipeline. In most cases plugins have static Pads, but some have dynamic Pads which are only created based on the data is flowing.

Pad linkup can be automatic, but in some cases it requires more complex commands due to choices that need to be made. You can ask a Pad for it's type, it may answer that multiple kinds of MIME Types are supported.

GStreamer

For example, let's playback a tone from a tone generator.

First we need an audio source, there is a source data node that produces tones. 'audiotestsrc'

```
audiotestsrc := GStreamerElement elementFactoryMake: 'audiotestsrc' name: 'source'.
```

"Note how I give an alias name 'source' so I can find it via name in the pipeline later"

We then need a converter which is responsible for changing one form of audio to another, this then converts the audio format coming from audiotestsrc to the target sound player. Recall that there are many basic audio formats, integer, fixed, or float data, compression etc.

```
audioconvert := GStreamerElement elementFactoryMake: 'audioconvert' name: 'convert'.
```

In this case the audioconvert element looks at both sides, asking each Pad what it can work with, then resolves either a common agreement for direct pass thru or does data conversion between the two elements. Finally we add a converter to manage playback Volume

```
volume := GStreamerElement elementFactoryMake: 'volume' name: 'volume'.
```

GStreamer

Finally we create the sink, which talks to the platform's ALSA hardware

```
audiosink := GStreamerElement elementFactoryMake: 'alsasink' name: 'sink'.
```

Now create the pipeline, and add all the elements

```
pipeLine := GStreamerPipeline name: 'my-pipeline'.
```

```
result := pipeLine addElement: audiotestsrc.
```

```
result := pipeLine addElement: audioconvert.
```

```
result := pipeLine addElement: volume.
```

```
result := pipeLine addElement: audiosink.
```

Obviously we could have a helper method to make this a bit less wordy.

Later when we release the pipeline it will release all the elements automatically.

GStreamer

Now link up the elements

```
result := GStreamerSystem default linkElementSrc: audiotestsrc toDest: audioconvert.
```

```
result := GStreamerSystem default linkElementSrc: audioconvert toDest: volume.
```

```
result := GStreamerSystem default linkElementSrc: volume toDest: audiosink.
```

Once a pipeline is created it has a number of states, this also applies to individual items in a pipeline, they are #pending, #null, #ready, #paused, #playing.

```
pipeLine setStateTo: #playing.
```

```
(Delay forSeconds: 1) wait.
```

```
pipeLine setStateTo: #paused.
```

```
pipeLine setStateTo: #null.
```

```
pipeLine release.
```

In the above example after building the pipeline the state will be #null, so I send it #playing which starts the pipeline and starts the elements. We delay for one second, then I send #paused to stop the pipeline, then I send it #null to change it to a releasable state, then I send release to the pipeline.

GStreamer

More examples: making a number of seconds of test audio

>>**makeTestDataAudio**

```
| pipeLine filesink audiotestsrc |
```

```
pipeLine := GStreamerPipeline
```

```
createPipeLineCalled: 'foo' thenBuildAndLinkElements: #('audiotestsrc' 'filesink')
```

```
initialize: [:p | ].
```

>>**thenBuildAndLinkElements:initialize:**

Takes a list of element names to build, stick in the pipeline and link together, along with a block that is evaluated with :p being the created pipeline so you can initialize it.

```
filesink := pipeLine findElementCalled: 'filesink'.
```

```
audiotestsrc := pipeLine findElementCalled: 'audiotestsrc'.
```

Remember I said we can find elements by their aliases?

GStreamer

*filesink setKey: 'location' toStringValue: self dailyTempFilePath,'Audio'.
audiotestsrc setKey: 'num-buffers' toLongValue: 1000.*

setKey:toStringValue:

There are a large number of methods to get or set meta-data on an element, in this case we set the location attribute for the file sink to a particular file/directory path, and we ask the audio test source element for only a certain number of buffers '*num-buffers*'. *Cough* remember C is a data typed language!

See the Unix cmd **gst-inspect** *element*

very helpful in describing what named properties an element has.

GStreamer

Now set the pipeline playing which generates the 1000 buffers of data.

```
pipeLine setStateTo: #playing.  
lastMessage := pipeLine getBus waitUntilErrorOrMessage: 'eos' uptoMilliseconds: 5000.  
pipeLine release.
```

The **waitUntilErrorOrMessage:uptoMilliseconds:** is a helper method which looks at data on the pipeline communication bus and returns if we see the 'eos' End of Stream message, or an error message, or if 5000 milliseconds pass. Normally we would expect the eos

Technically the code above is incorrect since I should be looking for the returned message of 'eos' or 'error' / nil which means the timer ran to the limit.

GStreamer

Now let's look at a much more complex item, you want to playback an OGG file from the internet to your audio hardware. For this example **the unix command** would look like so:

```
gst-launch gnomevfsrc location='http://www.gutenberg.org/files/20000/ogg/20000-47.ogg' ! oggdemux name=demux
demux. ! vorbisdec ! audioconvert ! osxaudiosink
```

```
Setting pipeline to PAUSED ...
Pipeline is PREROLLING ...
Pipeline is PREROLLED ...
Setting pipeline to PLAYING ...
New clock: GstAudioSinkClock
"Do a Cmd-C to stop the play back"
```

```
^Ccaught interrupt -- handling interrupt.
Interrupt: Stopping pipeline ...
Execution ended after 4884110000 ns.
Setting pipeline to PAUSED ...
Setting pipeline to READY ...
Setting pipeline to NULL ...
FREEING pipeline ...
```

In typical unix fashion you will find there are multiple ways to code '*gst-launch*' requests to perform a desired action. In this case we ask the *gnomevfsrc* (gnome virtual file system src element) to read data from <http://www.gutenberg.org/files/20000/ogg/20000-47.ogg> , this is fed to the *oggdemux* which demuxs the data into audio and video streams. We only feed the audio side into the *vorbisdec* element which is the vorbis audio decoder, which we then convert from the resulting audio format to the format needed for the os-x audio hardware.

GStreamer

In Squeak we require a few more lines.

>>setupoggAudioNormalFromHttp: aURIString

```
| oggdemux pipeLine result audiosink audioconvert vorbisdec vorbisdecSinkPad vorbisdecSinkPadCaps  
vorbisdecSinkPadCapsString gnomevfssrc |
```

"make all the elements"

```
gnomevfssrc := GStreamerElement elementFactoryMake: 'gnomevfssrc' name: 'gnomevfssrc'.  
oggdemux := GStreamerElement elementFactoryMake: 'oggdemux' name: 'oggdemux'.  
vorbisdec := GStreamerElement elementFactoryMake: 'vorbisdec' name: 'vorbisdec'.  
audiosink := GStreamerElement elementFactoryMake: self audiosink name: 'audiosink'.  
audioconvert := GStreamerElement elementFactoryMake: 'audioconvert' name: 'audioconvert'.
```

"Supply the URI"

```
gnomevfssrc setKey: 'location' toStringValue:  
    'http://www.gutenberg.org/files/20000/ogg/20000-47.ogg'.
```

GStreamer

"Setup the pipeline"

```
pipeLine := GStreamerPipeline name: 'my-pipeline'.
```

```
result := pipeLine addElement: gnomevfsrc.
```

```
result := pipeLine addElement: oggdemux.
```

```
result := GStreamerSystem default linkElementSrc: gnomevfsrc toDest: oggdemux.
```

```
result := pipeLine addElement: vorbisdec.
```

```
result := pipeLine addElement: audioconvert.
```

```
result := pipeLine addElement: audiosink.
```

"Start linking elements together"

```
result := GStreamerSystem default linkElementSrc: vorbisdec toDest: audioconvert.
```

```
result := GStreamerSystem default linkElementSrc: audioconvert toDest: audiosink.
```

GStreamer

"Now lets setup the dynamic pad link via a C callback that is implemented by the Plugin"

```
vorbisdecSinkPad := vorbisdec requestStaticPadByName: 'sink'.
```

```
oggdemux requestCallbackForSignal: 'pad-added' useArray: (Array with:  
vorbisdecSinkPad).
```

The **requestCallbackForSignal: useArray:** is a bit of magic that enables the *'pad-added'* message that flows on the bus to be seen and acted upon by a 'C' callback procedure since the vorbisdec element only creates the src pad we need for audio once it decodes the fact there is audio data. On the *'pad-added'* message we then connect the waiting sink pad to the dynamically created src pad. If we were doing audio **and** video we would have coded

```
oggdemux requestCallbackForSignal: 'pad-added' useArray:  
    (Array with: vorbisdecSinkPad with: theoradecSinkPad).
```

So on the *'pad-added'* callback the data type that is added maps to either of the Sink Pads

GStreamer

Time, where in this Time based Media am I?

seekToTimeInSeconds: seconds

seekToTimeInSecondsEnsurePlaying: seconds

seekToTime: nanoSeconds

You can send a request in seconds or nano-seconds to the pipeline, say you want to start playback at 5.0 seconds, the pipeline then will then ask it's elements to seek to 5 seconds.

or ***seekToPercent:*** or ***seekSimpleFormat:flags:cur:*** to seek on some other time based value.

This all combines in a **GStreamerMoviePlayerMorph**, **GStreamerMoviePlayerMorph**, **GStreamerPlayer** set of classes that are a clone of the original MP3 Morphic player written by John Maloney.

GStreamer

Squeak as a Sink or Src? Either for Audio or Video

Normally we just play the audio directly to the hardware playback element, why direct OGG audio to squeak to have it play with bits then give to audio hardware

See **oggHookupToSqueakAudioViaSqueakSink** versus **oggHookupToSqueakAudioAndVideo**

For Video well we need to see the bits.

See class **GStreamerElementForSqueakSinkVideo**

and

testPipeCheckPadHookupToSqueakAudioAndVideo

But a UNIX X11 guru could alter the logic to embed X11 window into squeak window and bypass Squeak VM for video generation. Pending action on OLPC.

The interface uses the 'fakeSink' element which calls an indicated procedure when data (video/audio) arrives for the element. We have a Smalltalk Class and a primitive api that setups the GStreamer element to call back to the Squeak plugin to queue the audio or video data then signal a semaphore. The supporting Smalltalk Code which is waiting on the semaphore then wakes up, and pulls the data from the plugin for the video or audio. We use both audioconvert or ffmpegcolorspace to convert the data from it's native form into a known format for Squeak to interpreter, versus having Squeak do that work.

GStreamer

'fakeSrc' where Squeak is a source for video/audio is left as an exercise for the reader.

There is primitive code support, and Smalltalk code framework to support this, but no working example.

GStreamer

playbin and playbin2

Factory Details:

Long name: Player Bin 2

Class: Generic/Bin/Player

Description: Autoplug and play media from an uri

Author(s): Wim Taymans <wim.taymans@gmail.com>

Rank: none (o)

Element that is a pre-built Bin that handles all the playback details.

Could use to playback audio/video in squeak versus current hand built pipeline.

Did not work on OLPC

Squeak code example, left as an exercise for the reader!