Plug your Cam - extending Gem the modular way

IOhannes m zmölnig
Institute of Electronic Music and Acoustics
University of Music and Dramatic Arts
Graz, Austria
zmoelnig@iem.at

ABSTRACT

In this paper we present the new plugin infrastructure in the upcoming release of Gem, that aims to cleanse the core library of superfluous dependencies.

The motivation for externalizing these parts of Gem into a plugin system, was mainly driven by two aspects: lowering binary dependencies and providing a uniform interface to various objectclasses across platforms.

So far, the main image input/output objectclasses (live video acquisition ([pix_video]), film footage acquisition ([pix_record]), video output ([pix_record]) and still image acquisition/output ([pix_image], [pix_buffer], ...)) have been switched to the new plugin infrastructure.

Keywords

Pure data, Gem, plugins, C++

1. MOTIVATION

Traditionally Gem (Graphics Environment for Multimedia) is built as a monolithic library of Pd objects, with linked in (that is: fixed) support for various system specific features, like image acquisition or output methods.

With Gem's (slow but steady) growth over the years, this has become more and more of a burden.

The motivation for externalizing these parts of Gem into a plugin system, was mainly driven by two aspects: lowering dependencies and providing a uniform interface across platforms.

Lowering dependencies aims at easing installation of binary packages (und thus the maintenance of such packages). For instance on linux, Gem (0.92) can be compiled with support for five different movie reading libraries, some of them being (partially) patent encumbered, outdated or otherwise hard to obtain. In order to support the widest range of film footage, one is tempted to link against all possible libraries. However, this also means that the end user has to install these libraries first in order to make use of Gem, because the Pd (or rather: the operating system) will refuse to load Gem if one of those libraries is missing. Even if they are not interested in video playback at all! (The alternative to making the end user install a number of libraries, is to provide them in the release, which bloats the package, eventually introducing legal problems.)

From the user's point of view, different backends provide different possibilities to interact with e.g. an image acquisition device. What's worse, different devices can have different features, the user might want to control. (E.g. a webcam could provide a means to control pan/tilt/zoom, whereas a video capture card might allow to switch between different inputs). In the past this divergence has led to incompatible implementations of e.g. the [pix_video] object, leading to patches that are not portable across operating systems.

2. DESIGN

As a solution for the above mentioned problems, the idea was born to remove as many external dependencies from the core Gem code base, and move them into an abstraction layer, that can be maintained separately. The design pattern that can be used for such a task, is the use of plugins, that encapsulate the specific code to interface with a given infrastructure/library. Plugins would allow to add (or remove) interoperability with frameworks from the core of Gem, without having to recompile Gem. Instead it would be enough, to simply add (or remove) a binary plugin file to Gem's search path, which would then be searched for automatically and loaded as needed.

What's better, if the user has a plugin installed on their machine, that depends on a library they have not installed (because they just copied the plugin file), then Gem will continue to work! Only the plugin will fail to load, which might be no show stopper.

2.1 plugin naming scheme

To ease the automatic loading of plugins, Gem it was decided to create a special naming scheme for the plugins, and simply load any file conforming to this naming scheme.

The naming scheme is as follows:
`gem_<pluginclass><NAME>.<extension>`

Teh `gem_` prefix obviously indicates that the given file is a plugin for Gem. The *<extension>* is the system's native extension for files that can be dynamically loaded at runtime. On W32, this would be `dll`, whereas on linux and OSX (and other UN*X like operating systems), the extension would be `so`.

The *<pluginclass>* is meant as a (human readable) filter, to allow to distinguish between different types of plugins, so only those plugins will be loaded that are actually needed to accomplish a certain task. This name usually corresponds to the name of the objectclass that is primarily associated with a given plugin.

For instance, plugins for live video capture (as used by the `[pix_video]` objectclass), would use *video* as plugin class name, whereas plugins to open movie files (e.g. `[pix_film]`), would use *film*.

Finally, the *NAME* part of the scheme, can be freely chosen by the plugin author, and is mainly a means to allow multiple plugin files live in the same directory. For readabilities sake, the *NAME* should reflect on the infrastructure the plugin interoperates with, e.g. a plugin for interfacing the *video4linux2* API to capture live video, would aptly be named *gem_ videoV4L2.so*.

2.2 properties

Different multimedia frameworks offer different ways to interact with the media. What's worse, different media streams might offer different ways to interact with them.

For instance, any framework offering the capabilities to capture video from a live source, will (hopefully) offer a way to grab a video frame. However, some frameworks might offer this frame in different qualities, e.g. simple black/white images of low quality (but at higher framerates) or high quality colour images. There might be the possibility to change the framerate of the capture device. Some industry standard cameras (and some webcams as well) might offer means to modify some properties of the camera's optics (e.g. exposure time), or even a way to manipulate features not related to the grabbing process at all (e.g. Pan/Tilt/Zoom of a motorized camera). Other devices will never be able to offer these properties (e.g. what does "focal length" mean to a frame grabber card?)

Older versions of Gem only offered a limited set of properties that could be manipulated and where available on most of these devices. This included e.g. the frame size or the video standard when grabbing video (the latter being targeted mainly at analog frame grabber cards, and not settable on most webcams).

If a given framework would expose more properties and one of those was actually needed for a project, a developer might eventually add a Pd-message to the Pd objectclass in order to be able to manipulate it. Unfortunately this led to a divergence of available methods for different implementations: e.g. `[pix_videoDarwin]` would have a method to set the "saturation" of the grabbed frame, whereas `[pix_video]` on linux would lack such a method.

This basically made patches non cross-platform.

In order to overcome this problem, a more dynamic but standardized way to manipulate such properties was introduced with the plugins: each plugin can be queried which properties it supports at runtime. This allows for both backend specific and device specific properties (e.g. even using the same backend one might get different properties, depending on the device currently opened: a framegrabber card might have different properties as a webcam). The names are not standardized, but given that they are symbolic names, it is hoped that they make sense to the user, so they know which knob they need to turn in order to acchieve whatever they want.

Property values can be of various types (strings, floats, enumerations), and querying the properties will give cues on which type is expected. This allows to build dynamic dialogs for a given interface from within Pd.

Setting properties uses key-value messages, e.g. "set exposure 0.5" would set the *exposure* level of a device to *0.5*. If the plugin does not know what to do with the given property, it will simply ignore it.

In some situations it is important to be able to set multiple properties at once. E.g. when a media stream needs to be restarted whenever the user wants to change the dimension or the framerate or the whitebalance, and restarting the media stream would take (say) 10 seconds, then it would be nice to be able to set dimension, framerate and whitebalance in one go rather than to have to restart the stream 3 times in a row. This can be done by a set of compound messages, that wait for a final "doit" message that actually triggers the properties to take effect. e.g. the messages `[setProps FrameRate 60, setProps AutoWhiteBalance 1, applyProps(` would remember that "FrameRate" should be "60", and "AutoWhiteBalance" should be turned on, but only then send apply these properties to the actual device.

3. MODULES

So far, plugins have been implemented for image acquisition and output.

3.1 Still image Acquisition

The *imageloader* plugin class, can be used to load still images, like JPEG or TIFF images. Given the simplicity of the task, the interface for this plugin is straightforward, and consists basically of a single function: `bool load(filename, &Image, &Properties)` This will try to load *filename* and store the result in the *Image* structure. The *Properties* could be used by the loader to report back additional information about the image, like tags encoded into the image file. Currently no plugin will actually report any properties, but this might be added later.

Currently these plugins are implemented for loading images:

imageSGI loads SGI raw images

imageJPEG loads JPEG images using libjpeg

imageTIFF loads TIFF images using libtiff

imageMAGICK loads any image supported by ImageMagick

imageQT loads any image supported by QuickTime

When loading an image, the host is supposed to call each plugin's *load()* function until the first one succeeds (and returns `true`).

3.2 Still image Output

Similar to the *imageloader* plugin class, there is the *imagesaver* plugin class for saving images. The API is a bit more complicated, since several APIs might be able to save a given image, but some might do better than others. Therefore a simple scoring system is introduced, where plugins can rate themselves, how well they will save the image: `float estimateSave(&Image, filename, mimetype, Properties)` If the plugin can store the *Image* using the given *mimetype*, it will assign itself (say) "100" points. For every property in the *Properties* list which it will respect when saving the image, it will gain another point. The plugin that returns the most points, is then chosen to save the image: `bool save(&Image, filename, mimetype, Properties)` If it fails (and returns `false`), the plugin with the next highest score is tried, and so on.

Most plugins available for image saving, are also available for image storing, with the exception of *imageSGI*.

3.3 Film Footage Acquisition

Reading movie files (series of prerecorded images as opposed to a single still image as above), requires the *film* plugin class. Compared to the still image loader, the API is broken into multiple steps for opening (and closing) the file, as well as for retrieving frames.

- `bool open(URI)`
- `void close(void)`
- `pixBlock* getFrame(void)`
- `int changeImage(frameNum)`

The host is supposed to call the *open()* function of each plugin until it finds one that is able to actually decode the given URI (which might be a file or a webcast URL). The thus opened stream, can then be retrieved using *getFrame()* calls. With media that supports it, one can do random frame access using the *changeImage()* method.

filmDarwin read movies using QuickTime (Apple only)

filmQT read movies using QuickTime (Apple and W32)

filmAVI read movies using the (deprecated) Video-For-Windows API

filmDS read movies using DirectShow

filmGMERLIN decode movies with the meta-decoder "gmerlin-avdecoder" (which internally uses FFMPEG, libAVIPLAY and other libraries)

filmAVIPLAY read movies using libAVIPLAY

filmMPEG1 an outdated MPEG2 decoder

filmMPEG3 read movies using libMPEG3 (for reading MPEG-2 movies)

filmQT4L decode movies using the Quicktime-For-Linux library

3.4 Live Video Capturing

Video capturing is done in the *video* plugin class that basically consists of the following API:

- `bool setDevice(devicename)`
- `bool open(Properties)`
- `void close(void)`
- `bool start(void)`
- `bool stop(void)`
- `pixBlock *getFrame(void)`
- `void setProperties(Properties)`

When a new device is to be opened, all plugins' *setDevice()* method will be called (which doesn't do anythin, but stores the device to be opened). Then each plugin's *open()* method is called, which tries to open the previously given device with the given properties. If it succeeds it returns `true`, if not, the next plugin is tried. The first plugin to succeed is then used by calling the *start()* method, which will turn grabbing on (grabbing can be turned on/off multiple times while a device is opened in many APIs, hence this two step process). Once *start()* succeeded, the host can call *getFrame()* to retrieve the newest frame. For changing properties while running, the *setProperties()* method is used, which gives a list or properties to be changed atomically. The plugin is expected to restart grabbing (if needed) itself.

videoDarwin using QuickTime

videoVFW using the (deprecated) Video-for-Windows API

videoDS using DirectShow

videoV4L using the (deprecated) video4linux(1) API

videoV4L2 using the video4linux2 API

videoDC1394 for grabbing IIDC cameras (industrial firewire video protocol)

videoDV4L for grabbing from consumer DV cams

videoSGI using SGI's grabbing facilities

videoUNICAP using Unicap

videoAVT using Allied Vision Technoligies GigE cameras (proprietary)

videoHALCON using MVTec's HALCON API (proprietary)

videoPYLON using Basler's PYLON API (proprietary)

4. FUTURE THOUGHTS

The main problem with the plugins as is, is that they are implemented in C++. Given that C++ does name wrangling of function names, this means, that one cannot use a plugin compiled with one compiler inside a Gem that is compiled with a different compiler. This problem is most prominent on the W32 platform, where several compilers (the Gnu g++ and Microsoft's Visual C++) are in prominent use.

A possibly workaround for this solution would be to provide a C API to write these plugins.