Developing Krita Plugins

Table of Contents

[2] Note: we really, really should find a way to enable Krita to keep a file open and only read data on a as-needed basis, instead of copying the entire contents to the internal paint device representation. But that would mean datamanager backends that know about tiff files and so on, and is not currently implemented. It would be ideal if some file filters could implement a class provisionally named KisFileDataManager, create an object of that instance with the current file and

1. Introduction

Krita is infinitely extensible with plugins. Tools, filters, large chunks of the user interface and even colorspaces are plugins. In fact, Krita recognizes these six types of plugins:

- colorspaces — these define the channels that constitute a single pixel
- tools — anything that is done with a mouse or tablet input device
- paint operations — pluggable painting effects for tools
- image filters — change all pixels, or just the selected pixels in a layer
- viewplugins — extend Krita's user interface with new dialog boxes, palettes and operations
- import/export filters — read and write all kinds of image formats

Krita itself consists of three layered libraries and a directory with some common support classes: kritacolor, kritaimage and kritaui. Within Krita, objects can by identified by a KisID, that is the combination of a unique untranslated string (used when saving, for instance) and a translated string for GUI purposes.

A word on compatibility: Krita is still in development. From Krita 1.5 to 1.6 not many API changes are expected, but there may be some. From Krita 1.6 to 2.0 we will move from Qt3 to Qt4, from KDE3 to KDE4, from autmake to cmake: many changes are to be expected. If you develop a plugin for Krita and choose to do so in Krita's subversion repository, chances are excellent that we'll help you porting. These changes may also render parts of this document out of date. Always check with the latest api documentation or the header files installed on your system.

1.1. KritaColor

The first library is kritacolor. This library loads the colorspace plugins.

- A **colorspace plugin** should implement the KisColorSpace abstract class or, if the basic capabilities of the new colorspace will be implemented by lcms (http://www.littlecms.com/), extend KisAbstractColorSpace. The kritacolor library could be used from other applications and does not depend on KOffice.

1.2. KritaImage

The libkritaimage library loads the filter and paintop plugins and is responsible for working with image data: changing pixels, compositing and painting. Brushes, palettes, gradients and

patterns are also loaded by libkritaimage. It is our stated goal to make libkritaimage independent of KOffice, but we currently share the gradient loading code with KOffice.

It is not easy at the moment to add new types of resources such as brushes, palettes, gradients or patterns to Krita. (Adding new brushes, palettes, gradients and patterns is easy, of course.) Krita follows the guidelines of the Create project (http://create.freedesktop.org/) for these. Adding support for photoshops brush file format needs libkritaimage hacking; adding more gimp brush data files not.

KritaImage loads the following types of plugins:

- **Krita filters** must extend and implement the abstract class KisFilter, KisFilterConfiguration and possibly KisFilterConfigurationWidget. An example of a filter is Unsharp Mask.

Paint operations or paintops are the set of operations painting tools suchs as freehand or circle have access to. Examples of paintops are pen, airbrush or eraser. Paintops should extend the KisPaintop base class. Examples of new paintops could be a chalk brush, an oilpaint brush or a complex programmable brush.

1.3. KritaUI

The libkritaui library loads the tool and viewplugins. This library is a KOffice Part, but also contains a number of widgets that are useful for graphics applications. Maybe we will have to split this library in kritapart and kritaui in the 2.0 release. For now, script writers are not given access to this library and plugin writers are only allowed to use this library when writing tools or viewplugins. KritaUI loads the following types of plugins:

- **Tools** are derived from KisTool or one of the specialized tool base classes such as KisToolPaint, KisToolNonPaint or KisToolFreehand. A new tool could be a foreground object selection tool. Painting tools (and that includes tools that paint on the selection) can use any paintop to determine the way pixels are changed.
- **Viewplugins** are ordinary kparts that use kxmlgui to insinuate themselves into Krita's user interface. Menu options, dialogs, toolbars -- any kind of user interface extension can be a viewplugin. In fact, important functionality like Krita's scripting support is written as a viewplugin.

1.4. Import/Export filters

Import/Export filters are KOffice filters, subclasses of KoFilter. Filters read and write image

data in any of the myriad image formats in existence. And example of a new Krita import/export filter could be a PDF filter. Filters are loaded by the KOffice libraries.

2. Creating plugins

Plugins are written in C++ and can use all of KDE and Qt and the Krita developer API. Only viewplugins should use the KOffice API. Don't worry: Krita's API's are quite clear and rather extensively documented (for free software) and coding your first filter is really easy.

If you do not want to use C++, you can write scripts in Python or Ruby; that is a different thing altogether, though, and you cannot currently write tools, colorspaces, paintops or import/export filters as scripts.

Krita plugins use KDE's parts mechanism for loading, so the parts documentation at http://developer.kde.org is relevant here, too.

Your distribution should have either installed the relevant header files with Krita itself, or might have split the header files into either a KOffice dev or a Krita dev package. You can find the api documentation for Krita's public API at http://koffice.org/developer/apidocs/krita/html/.

2.1. Automake (and CMake)

KDE 3.x and thus KOffice 1.5 and 1.6 use automake; KDE 4.0 and KOffice 2.0 use cmake. This tutorial describes the automake way of creating plugins. If I have not updated this manual when we release KOffice 2.0, please remind me to do so.

Plugins are kde modules and should be tagged as such in their Makefile.am. Filters, tools, paintops, colorspaces and import/export filters need .desktop files; viewplugins need a KXMLGui **pluginname.rc** file in addition. The easiest way to get started is to checkout the krita-plugins project from the koffice subversion repository and use it as the basis for your own project. We intend to prepare a skeleton krita plugin pack for KDevelop, but haven't had the time to do so yet.

2.1.1. Makefile.am

Let's look at the skeleton for a plugin module. First, the Makefile.am. This is what KDE uses to generate the makefile that builds your plugin:

```
kde_services_DATA = kritaLIBRARYNAME.desktop

INCLUDES = $(all_includes)

kritaLIBRARYNAME_la_SOURCES = sourcefile1.cc sourcefile2.cc
```

```
kde_module_LTLIBRARIES = kritaLIBRARYNAME.la
noinst_HEADERS = header1.h header2.h

kritaLIBRARYNAME_la_LDFLAGS = $(all_libraries) -module $(KDE_PLUGIN)
kritaLIBRARY_la_LIBADD = -lkritacommon

kritaextensioncolorsfilterse_la_METASOURCES = AUTO
```

This is the makefile for a filter plugin. Replace LIBRARYNAME with the name of your work, and you're set.

If your plugin is a viewplugin, you will likely also install a .rc file with entries for menubars and toolbars. Likewise, you may need to install cursors and icons. That's all done through the ordinary KDE Makefile.am magic incantantions:

```
kritarcdir = $(kde_datadir)/krita/kritaplugins
kritarc_DATA = LIBRARYNAME.rc
EXTRA_DIST = $(kritarc_DATA)

kritapics_DATA = \
    bla.png \
    bla_cursor.png
kritapicsdir = $(kde_datadir)/krita/pics
```

2.1.2. Desktop files

The .desktop file announces the type of plugin:

```
[Desktop Entry]
Encoding=UTF-8
Icon=
Name=User-visible Name
ServiceTypes=Krita/Filter
Type=Service
X-KDE-Library=kritaLIBRARYNAME
X-KDE-Version=2
```

Possible ServiceTypes are:

- Krita/Filter
- Krita/Paintop
- Krita/ViewPlugin
- Krita/Tool
- Krita/ColorSpace

File import and export filters use the generic KOffice filter framework and need to be discussed separately.

2.1.3. Boilerplate

You also need a bit of boilerplate code that is called by the KDE part framework to instantiate the plugin — a header file and an implementation file.

a header file:

```
#ifndef TOOL_STAR_H_
#define TOOL_STAR_H_

#include <kparts/plugin.h>

/**
```

```
 * A module that provides a star tool.
 */
class ToolStar : public KParts::Plugin
{
    Q_OBJECT
public:
    ToolStar(QObject *parent, const char *name, const QStringList &);
    virtual ~ToolStar();

};

#endif // TOOL_STAR_H_
```

And an implementation file:

```
#include <kinstance.h>
#include <kgenericfactory.h>

#include <kis_tool_registry.h>

#include "tool_star.h"
#include "kis_tool_star.h"


typedef KGenericFactory<ToolStar> ToolStarFactory;
K_EXPORT_COMPONENT_FACTORY( kritatoolstar, ToolStarFactory( "krita" ) )


ToolStar::ToolStar(QObject *parent, const char *name, const QStringList &)
    : KParts::Plugin(parent, name)
{
    setInstance(ToolStarFactory::instance());
    if ( parent->inherits("KisToolRegistry") )
    {
        KisToolRegistry * r = dynamic_cast<KisToolRegistry*>( parent );
        r -> add(new KisToolStarFactory());
    }

}

ToolStar::~ToolStar()
{
}

#include "tool_star.moc"
```

2.1.4. Registries

Tools are loaded by the tool registry and register themselves with the tool registry. Plugins like tools, filters and paintops are loaded only once: view plugins are loaded for every view that is created. Note that we register factories, generally speaking. For instance, with tools a new instance of a tool is created for every pointer (mouse, stylus, eraser) for every few. And a new paintop is created whenever a tool gets a mouse-down event.

Filters call the filter registry:

```
    if (parent->inherits("KisFilterRegistry")) {
        KisFilterRegistry * manager = dynamic_cast<KisFilterRegistry *>(parent);
        manager->add(new KisFilterInvert());
    }
```

Paintops the paintop registry:

```
    if ( parent->inherits("KisPaintOpRegistry") ) {
            KisPaintOpRegistry * r = dynamic_cast<KisPaintOpRegistry*>(parent);
            r -> add ( new KisSmearyOpFactory );
    }
```

Colorspaces the colorspace registry (with some complications):

```
    if ( parent->inherits("KisColorSpaceFactoryRegistry") ) {
      KisColorSpaceFactoryRegistry * f = dynamic_cast<KisColorSpaceFactoryRegistry*>(parent);

        KisProfile *defProfile = new KisProfile(cmsCreate_sRGBProfile());
        f->addProfile(defProfile);

        KisColorSpaceFactory * csFactory = new KisRgbColorSpaceFactory();
        f->add(csFactory);

        KisColorSpace * colorSpaceRGBA = new KisRgbColorSpace(f, 0);
        KisHistogramProducerFactoryRegistry::instance() -> add(
                new KisBasicHistogramProducerFactory<KisBasicU8HistogramProducer>
                (KisID("RGB8HISTO", i18n("RGB8 Histogram")), colorSpaceRGBA) );
    }
```

View plugins don't have to register themselves, *and* they get access to a KisView object:

```
    if ( parent->inherits("KisView") )
    {
        setInstance(ShearImageFactory::instance());
        setXMLFile(locate("data","kritaplugins/shearimage.rc"), true);

        (void) new KAction(i18n("&Shear Image..."), 0, 0, this, SLOT(slotShearImage()),
actionCollection(), "shearimage");
        (void) new KAction(i18n("&Shear Layer..."), 0, 0, this, SLOT(slotShearLayer()),
actionCollection(), "shearlayer");

        m_view = (KisView*) parent;
    }
```

Remember that this means that a view plugin will be created for every view the user creates: splitting a view means loading all view plugins again.

2.1.5. Plugin versioning

Krita 1.5 loads plugins with the `X-KDE-Version=2` set in the .desktop file. Krita 1.6 plugins will probably be binary incompatible with 1.5 plugins and will need the version number 3. Krita 2.0 plugins will need the version number 3. Yes, this is not entirely logical.

3. Colorspaces

Colorspaces implement the KisColorSpace pure virtual class. There are two types of colorspaces: those that can use lcms for transformations between colorspaces, and those that are too weird for lcms to handle. Examples of the first are cmyk, rgb, yuv. An example of the latter is watercolor or wet & sticky. Colorspaces that use lcms can be derived from KisAbstractColorSpace, or of one of the base classes that are specialized for a certain number of bits per channel.

Implementing a colorspace is pretty easy. The general principle is that colorspaces work on a simple array of bytes. The interpretation of these bytes is up to the colorspace. For instance, a pixel in 16-bit GrayA consists of four bytes: two bytes for the gray value and two bytes for the

alpha value. You're free to use a struct to work with the memory layout of a pixel in your colorspace implementation, but that representation is not exported. The only way the rest of Krita can know what channels and types of channels your colorspace pixels consist of is through the KisChannelInfo class.

Filters and paintops make use of the rich set of methods offered by KisColorSpace to do their work. In many cases, the default implementation in KisAbstractColorSpace will work, but more slowly than a custom implementation in your own colorspace because KisAbstractColorSpace will convert all pixels to 16-bit L*a*b and back.

3.1. KisChannelInfo

(http://websvn.kde.org/trunk/koffice/krita/kritacolor/kis_channelinfo.h)

This class defines the channels that make up a single pixel in a particular colorspace. A channel has the following important characteristics:

- a name for display in the user interface
- a position: the byte where the bytes representing this channel start in the pixel.
- a type: color, alpha, substance or substrate. Color is plain color, alpha is see-throughishness, substance is a representation of amount of pigment or things like that, substrate is the representation of the canvas. (Note that this may be refactored at the drop of a hat.)
- a valuetype: byte, short, integer, float — or other.
- size: the number of bytes this channel takes
- color: a QColor representation of this channel for user interface visualization, for instance in histograms.
- an abbreviaton for use in the GUI when there’s not much space

3.2. KisCompositeOp

As per original (XXX: link) Porter-Duff, there are many ways of combining pixels to get a new color. The KisCompositeOp class defines most of them: this set is not easily extensible except by hacking the kritacolor library.

A colorspace plugin can support any subset of these possible composition operations, but the set must always include "OVER" (same as "NORMAL") and "COPY". The rest are more or less optional, although more is better, of course.

3.3. KisColorSpace

The methods in the KisColorSpace pure virtual classs can be divided into a number of groups: conversion, identification and manipulation.

All classes must be able to convert a pixel from and to 8 bit RGB (i.e., a QColor), and preferably also to and from 16 bit L*a*b. Additionally, there is a method to convert to any colorspace from the current colorspace.

Colorspaces are described by the KisChannelInfo vector, number of channels, number of bytes in a single pixel, whether it supports High Dynamic Range images and more.

Manipulation is for instance the combining of two pixels in a new pixel: bitBlt, darkening or convolving of pixels.

Please consult the api documentation (XXX: link) for a full description of all methods you need to implement in a colorspace.

KisAbstractColorSpace implements many of the virtual methods of KisColorSpace using functions from the lcms library. On top of KisAbstractColorSpace there are base colorspace classes for 8 and 16 bit integer and 16 and 32 bit float colorspaces that define common operations to move between bit depths.

4. Filters

Filters are plugins that examine the pixels in a layer and them make changes to them. Although Krita uses an efficient tiled memory backend to store pixels, filter writers do not have to bother with that. When writing a filter plugin for the Java imaging api, Photoshop or the Gimp you need to take care of tile edges and "cobble" tiles together: krita hides that implementation detail[1].

Krita uses iterators to read and write pixel values. Alternatively, you can read a block of pixels into a memory buffer, mess with it and then write it back as a block. But that is not necessarily more efficient, it may even be slower than using the iterators; it may just be more convenient. See the API documentation: XXX.

Krita images are composed of layers, of which there are currently four kinds: paint layers,

[1] Note that it is theoretically easy to replace the current tile image data storage backend with another backend, but that backens are not true plugins at the moment, for performance reasons.