Styles in Qt and KDE: A new approach

Eduardo Madeira Fleury
openBossa / INdT
Recife, PE - Brazil
eduardo.fleury@openbossa.org

ABSTRACT
This paper complements the homonymous talk presented at Akademy 2010 in Tampere, Finland.

It documents the research done by openBossa[1] in Recife, Brazil and Qt Development Frameworks[2] in Oslo, Norway in the first semester of 2010.

This study proposes a new approach for styling graphic user interfaces written in Qt, compares that to the existing solution and evaluates the relevance of integrating styling and the new QtQuick[3] technology.

Categories and Subject Descriptors
H.5.2 [**Information Interfaces and Presentation**]: User Interfaces

General Terms
GUI Styling, Primitive-Graph, Element Caching, Declarative GUI

Keywords
KDE, Akademy, Qt, Styles, QtQuick, C++

1. INTRODUCTION
Qt 4.0 was released in 2005, since then a lot of changes have been introduced. Among those, the QGraphicsView canvas was introduced in 2006, then one year later with Qt 4.4 we had the introduction of alien widgets, last year it was time for the Animation Framework and finally in 4.7 we will have QtQuick, a set of tools for designing interfaces in a declarative way.

However, even after all these changes, Qt still relies on a widget set based on the old QWidgets. With the increasing popularity of canvas-based *GUIs* (Graphical User Interfaces) the Qt community started discussing whether it was time to port old widgets to QGraphicsView, or maybe to QtQuick. That raised important questions regarding how to reduce code duplication and how to meet the requirements of modern interfaces, one of these being the ability of customization.

To gather more information on the subject, we at openBossa together with Marius Bugge Monsen and other friends at Qt Development Frameworks started a research project[4] on what could be the next generation of Qt widgets and Qt *styles*, the latter being the subject of this paper.

1.1 What are Styles
In the development of GUI applications, it is often desired to split the code that actually paints the interface apart from UI logic code. In some contexts widgets like buttons, labels and text fields are expected to act as small black boxes. In other situations however, we must go deep into the painting details of each of these building blocks to change, for instance, the font used or the background colour of a button.

That separation helps keeping the code organized and also allows for the painting code to be replaceable without affecting other parts of the application. The latter is a requirement when the interface has to be theme-able or has to look like it belongs to different platforms, for instance KDE, Gnome, MacOS, Windows or MeeGo.

To meet those requirements many application frameworks create modular classes to handle the painting recipes for each widget, these are called *Style* classes.

1.2 Objective
This article will examine the current style implementation in Qt and KDE, identify a few shortcomings, propose a solution based on our ongoing research and then evaluate the integration of this solution with upcoming technologies like Qt Quick.

1.3 Styles in Qt and KDE
In Qt the style concept is implemented by the `QStyle` family of classes. Being a cross-platform framework, Qt developers wanted Qt applications to look like native applications anywhere they run. For instance, the same application should appear with rounded blue buttons when running in Mac, with rectangular grey buttons in KDE or Windows and so forth.

To accomplish that, Qt provides different `QStyle` classes

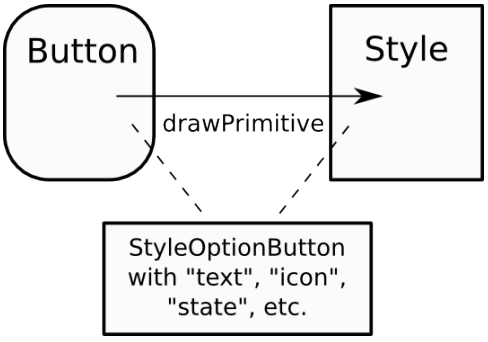

Figure 1: StyleOption structure used to move data between widget and style

for each platform. The most important part of those is the `drawPrimitive` method, that is responsible for drawing small parts of the different widgets, for instance the the background of a Button or the down arrow of a scrollbar. It is in each implementation of `drawPrimitive` that `QPainter` is called do draw lines, polygons, texts and pixmaps to compose the looks of each widget.

The widgets on the other hand do not know how to paint themselves. Instead they use their `paint` method to call the current style so the actual painting happens in a platform-specific way.

In KDE the approach is similar. The class `KStyle` is a subclass of `QStyle` and while some features are added, the core behaviour is similar.

Another concept important to understanding how styles work in Qt is the role played by `QStyleOption`. This structure holds information regarding the current widget status and is provided to the style at paint time. It is through this mechanism that the current style knows whether it should paint a sunken or raised background for a button. Or which text to write for a given label object.

Figure 1 shows the `QStyleOption` data structure being used to share data between a Button widget and the current Style.

2. WHAT CAN BE IMPROVED

Our study identified two major drawbacks in the current styling system used by Qt and KDE. Below we go into each of them and in the next section propose a new solution for the subject.

2.1 Procedural Painting

In the work flow described, the painting of a widget is done by a series of imperative calls to methods, or procedures, of the current style. That is called *Procedural Painting*.

That means widgets are implemented in terms of a single objects on the screen. Back in the 90's that was a requirement. Each single widget on the screen was visible to the window system as a separate window, that represented a huge overhead and caused painting artifacts such as flickering.

However this is no longer true. Since Qt 4.4 widgets are represented as surfaces internal to Qt, also known as alien windows[5]. Therefore the effort of reducing the number of widgets on the screen is no longer of great relevance.

On the other hand, with the increasing popularity of animated interfaces, and the quest for high frame rates, even in mobile devices, the cost of repainting the screen became a bottleneck.

The collateral effect of reducing the number of widgets on the screen is that the painting routines for each of them become rather complex. For instance, the painting of a single button usually requires the painting of a background (sunken or raised), a label, a highlight hint and also some decoration.

This complexity means once a small change happens to a widget, the style procedures for each of these primitives need to be completed. That can be quite an expensive operation when painting of vectorial images or text glyphs is involved.

2.2 Ability to customize look and feel

Another issue with the current styling system is the extent to which widgets can have their look and feel customized. With the current approach it is possible to change the way each primitive is painted by the style, for instance, by changing the palette of colours, a text font or the images used.

On the other hand, styles are not allowed to decide which primitives are used to paint each widget. That becomes a problem when designers ask for deeper changes in a screen component.

For instance, suppose we are porting Qt to a platform where buttons must show some effect around the mouse pointer when hovered. With current implementation chances are we do not have within the style, the information or the hooks required to paint that. To proper implement that we would need to make the core widget aware of that decoration and change the style/widget interface. And suddenly it seems not appropriate to change widget core to fulfil a requirement with regard to how the widget looks.

An even tighter restriction exists when we try to customize the behaviour of a widget, ie. the way it feels to use it. The problem here is that events happening on the widget are not visible to the style, that means all event handling must be done, and most probably hardcoded, inside the widget event handlers. If a designer wants that to be different on a platform or style basis, there is not much that can be done.

The most straightforward example is the way click events are handled in touch-based UIs. In these environments it is recommended widgets have large interaction areas, maybe larger than the areas actually painted by them. The rationale here is to make it easier for users to click small buttons with large fingers. How could the Qt style used in mobile phones customize that? The same restrictions appear when styles are required to change the way widgets respond to different gestures or other kind of events.

In other words, while the current styling mechanism can change imagery, colours and shapes, we can not go as far as saying it is possible to really customize the look and feel of

widgets through it.

3. PROPOSED SOLUTION

What should styles be like in the future? Our study was based on three main requirements, namely:

- Be canvas agnostic. The solution proposed had to be easily used by the existing canvas as well as any upcoming ones. The concept should rely only on high level canvas definitions that we can assume will not cease to exist.
- Provide an alternative to procedural painting.
- Be flexible enough to provide style creators with freedom to implement the look and feel they need in different situations.

To explain the architecture used I will first go through the key decisions we made and then wrap it up with a the general overview.

3.1 Painting Primitives

Procedural painting was no longer an option, instead it should support defining widgets in terms of *Primitive-Trees*. The idea is that complex widgets should not paint themselves, ie. should not implement the `paint()` method. Instead those will act as the parent of simpler, paint-enabled parts, the *primitives*.

So, while in the old approach `Button::paint()` was responsible for painting a raised/sunken background, a label and also an icon, now it will not paint at all. Instead, every button widget will have a set of primitives beneath it, in this case a "background", a "text" and an "icon".

Figure 2 shows the painting of a Button widget using the former procedural method (at the top) and the proposed one (at the bottom). Note that in the older approach Button makes three calls to Style every time it is painted. In the new one it just acts as the parent for three primitives, one for its background, one for the icon and a last one for its label.

Each of these primitives could be part of a standard set provided by Qt plus a few tailored to a widget or an application. One of the main advantages of this approach is that it becomes easy for the canvas to keep pixmap caches of such primitives. With that, a change in one of them does not trigger repaints of the others. Instead just a composition of pixmaps is required, which can accelerated in hardware and is way cheaper than drawing complex primitives like those that draw texts.

Recapitulation of first concept, widgets do not draw themselves, instead they play the parent role for primitives that do so.

3.2 Populating widgets

The painting method had been decided but still there were important decisions to be made. The next one was regarding

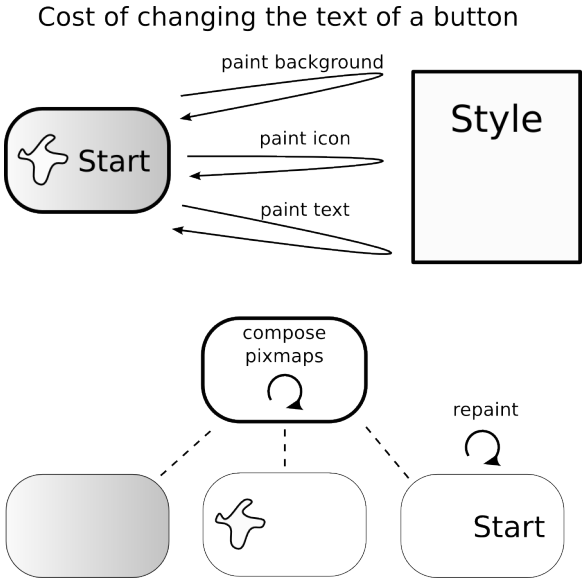

Figure 2: Difference between procedural and primitive-graph painting

who would be responsible for creating such primitives, in other words, who would *populate* the empty widgets?

When customization is not important widgets can simply populate themselves. However this means the widget will always create the same primitives, that does not fit the idea of styling and was disregarded as an option.

In our situation though it makes sense for the style classes to take that responsibility. Nothing can grant more flexibility to a style than allowing it to populate each widget with whichever primitives it needs to achieve the target looks.

Second concept in one phrase, styles when given an empty widget can create primitives and hook them on it.

3.3 Property Binding

Other important part of the style machinery is the communication between the style and the widget it is painting or populating. The style needs information about the widget state to paint it. Take a button as an example, the style could paint its background raised or sunken, it needs data to make that decision.

In the old implementation widgets provide a `QStyleOption` structure with state data every time it knows a paint method is required. This was possible because the widget did its own event handling and had a good understanding of the way it was painted. So using the same button example, when the widget is clicked it knows about that and also knows its background needs a repaint, then it calls the appropriate method from the current style.

Respecting the second concept, our research widgets do not know about their primitives, that means they do not know who to update whenever something happens. Still using the button example, the widget in this case does not even know whether it has a background image that needs to be updated

on click. It may be transparent, it may have a small light indicator or whatever says the style.

One possible solution here was to make the style responsible for updating the primitives as well. In that case, each time something important happened at the widget it would inform the style who in turn would change the visibility, colour or any other property of the relevant primitive.

However that approach would add too much widget-related code inside the style. It is one thing to ask styles to populate widgets once, but is quite another to require them to keep updating each of these primitives whenever something changes. It would be way too much different code, from different contexts in one single class, what in practice leads to huge and clumsy switch/case blocks.

The chosen solution came from another example of primitive-based programming: QtQuick[3]. In QtQuick the interfaces are entirely built from primitives like Rectangle, Image, Text and so forth. The context is different since these blocks are put together by the GUI designer using the QML language, it is not about a style creating them. But nevertheless the requirement of primitives needing to be aware of general UI state exists and it was solved in an interesting way.

One of the most important features QtQuick introduce is *Property Binding*[6]. This is the ability of hooking, or binding, the value of one property to the value of a general QtScript expression. For instance, the colour of a Rectangle primitive could change when the button it represents is clicked. To respect that, the "colour" property is bound an appropriate expression, for instance `button.pressed ? 'Red' : 'Blue'`. The most interesting part here is that whenever the result of this expression changes, the property bound to it is immediately updated.

We see this declarative way of defining properties to be easier to understand and to generate clearer code then the imperative switch/case block in the `Style::updatePrimitive()` idea. With some machinery in place to allow for these bindings to happen in C++, it is now up to the style to bind the primitives it creates to the appropriate value sources, one time only, and then let it go.

Figure 3 shows what happens when text is changed in a Label widget. The differences in data flow can be seen when the Style is used to update primitive or when data binding is in place.

That said, the third concept is communication between primitives and widgets is done using Property Binding configured by the style.

3.4 Event handling

With those concepts in place, style creators have pretty much freedom to ensure widgets look the way they need. But what about the way it feels like to use them? How could a style change the size of a button click area? Or how could a style make a list become kinetic? What about gestures?

In traditional implementation all event handling is done by

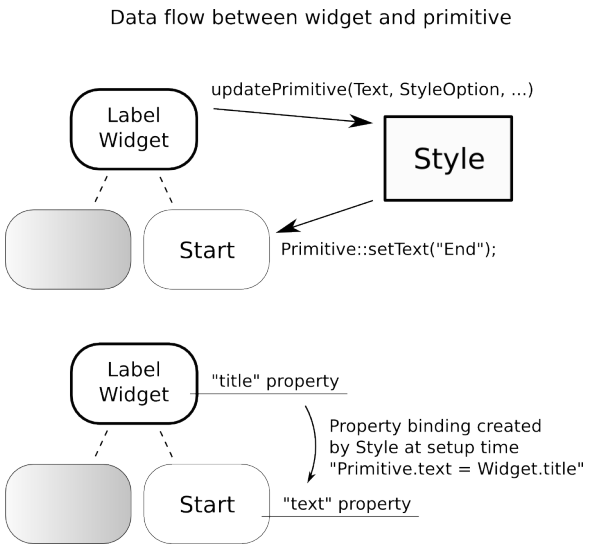

Figure 3: Property binding versus updating primitives through style

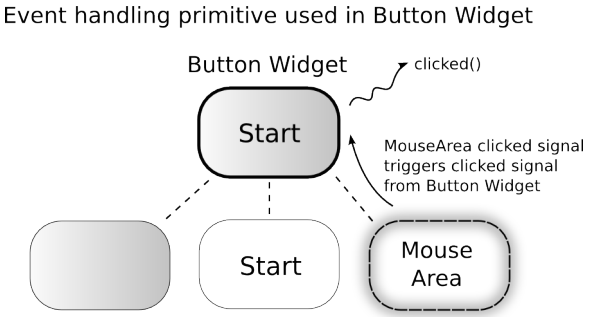

Figure 4: MouseArea, an event handling primitive used in Button Widget

the widgets internally, styles play no important role on that. To add flexibility to event handling as well as painting, we extended the concepts above by creating *Event handling primitives*.

This concept is also present in QtQuick and consists in using special primitives that do not paint themselves, instead they are sensible to different user interaction events like mouse movement, keyboard pressing or multitouch gestures.

Styles can then populate widgets with event handling primitives as well as painting ones and bind properties and signals among them.

Figure 4 shows how an hypothetical MouseArea primitive could be used to implement behaviour of a Button Widget. In this example the primitive is able to grab mouse events and emits a signal when clicked. As this signal is meant to be public, the style connects that to the `clicked()` signal of the widget. In other situations that primitive could be replaced by one sensible to the keyboard or to multitouch gestures, for instance.

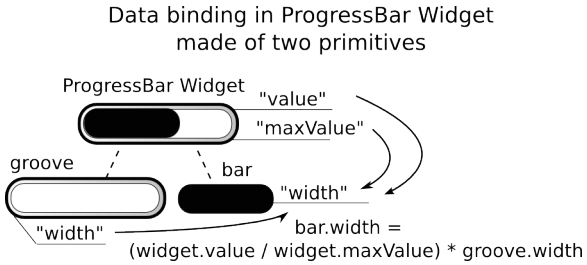

Figure 5: Track of a ProgressBar widget have its width bounded to an expression of other primitives

The fourth concept: events are handled by special primitives, also created by the style.

3.5 Public widget features

This new styling system has the flexibility needed to change the way widgets look like and behave. However to which extent can that modify the expected semantics of a widget? For instance, is there a tipping point where a heavily modified button is not perceived as a button anymore? If so, where is it?

If there is one set of rules styles must respect it is the list of features a widget must provide, ie. its public API. From the point of view of application development, buttons are expected to emit `clicked()` signals once in a while, labels are supposed to show some text, scrollbars must provide a getter for the current value and so forth. How can the style respect these requirements?

The first part of our solution was to ensure the public functionality was exported as properties or signals. Meaning for instance that labels have `text` or `font` properties, progress bars implement `value` while buttons emit `clicked()` signals.

That ready, once styles have access to the widgets they need to populate, it is their call how to bind widget and primitive properties together. With proper bindings in place the primitives and widget state will remain synchronized and consistent.

Figure 5 shows a ProgressBar widget with public properties "value" and "maxValue" that are used in an expression to define the "width" of the painted part of the progress bar.

Widget exports properties which must be bound to the primitives, that is the fifth and last concept.

The whole new styling system is based on these five high level ideas, once they have been well understood the implementation will come naturally. It is important to note here that while the experiments have been made on top of QGraphicsView, the only Qt concepts required here are QObject properties and bindings. This was taken into account throughout the discussion process to ensure the solution would be resilient to changes in canvas.

4. INTERACTION WITH QTQUICK

At the time of this writing QtQuick is being polished for Qt 4.7 release. This is an important feature that has been naturally making noise. While researching styles we took into account the potential interactions between those and Quick.

4.1 Why not just QtQuick?

There is one kind of user interfaces that has become very popular, that are rich GUIs based on the use of pixmaps or SVGs to design widgets. Historically this was used by games and nowadays has been used with many eye-candy mobile applications as those we see on smartphones.

When designers choose this path they sacrifice platform integration for a unique look and feel of their application. In these situations, technologies that reduce development work like Quick are much more important than styles, after all the interface is meant to be well done, but done only once. Therefore it makes sense to ask why not just use QtQuick?

However, when different applications share the same visual identity of the platform, users tend to feel at home and the learning curves are more gentle. Even in games where the UI have to be written from scratch, it is interesting to use standard widgets in settings and other auxiliary screens, and today that is not provided in QtQuick.

So ignoring styling completely may not be the best solution. In the next subsections we go through a few ways of getting the best of two worlds here.

4.2 Styleable widgets inside QtQuick

The first use case we will go through is that where GUI developers want the ease of use QtQuick provides but still, want a few native looking widgets.

Our solution was to create C++ widgets that do their painting through the styling system and then export, or register, them in the declarative engine. Those could be special components like `NativeButton`, `NativeLineEdit` and so forth. That way the designer could freely mix custom, pixmap-based widgets with native looking ones by using the right components when writing QML.

Furthermore, in case upstream Qt provides stylable QGraphicsWidgets, these could potentially be the ones exported to QML, that way no additional C++ classes would be needed at all.

4.3 QtQuick as a widget styling tool

Those familiar with QtQuick will notice the similarities between that language and the five concepts used as the base of the proposed styling system. This is not a coincidence, Property Binding and event handling primitives were ideas we extended to C++.

In that sense, what if we could go a little further and use the simpler, interpreted QML language rather than C++ to implement styles that could be used to define the native look and feel of whole platforms, being used to style all widgets, even those of C++ applications?

To make that possible we created a special C++ style. That style, instead of populating widgets with several ordinary

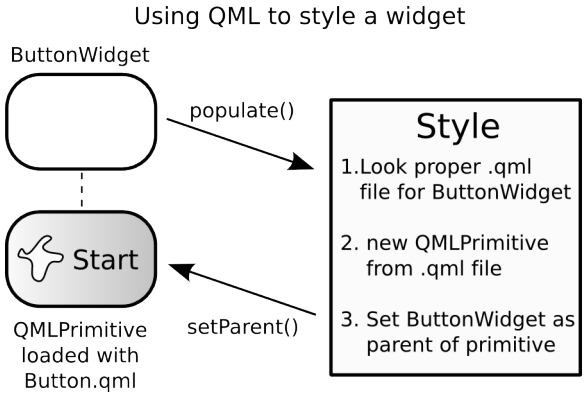

Figure 6: Diagram showing widget and QML style interaction

primitives, creates a single special one. This primitive instead of painting something simple as a rectangle or an image, is able to load a QML file and show the tree defined there by using standard QDeclarative tools. We call that the *QMLPrimitive*.

Therefore the job of this special style is to discover the right QML file for a given widget, load it into a QML primitive and then hook it on the widget. Additionally, to allow for the QML code to communicate with the widget, the latter is registered in the declarative context. With this in place, any widget that relies on the styling machinery will acquire the look and feel described in QML.

Figure 6 shows the responsibility of the custom QML Style populating a Button Widget with a single QMLPrimitive loaded with the appropriate QML file.

In a real world scenario the major benefit of this approach is the work flow. The special style coded in C++ would be done only once and from then on the styles themselves would be done in QML. That means a designer able to work with QML could create his/her own styles rather than requiring a developer to code them in C++, what can be much more time consuming.

5. CONCLUSION

In the last year we have been able to deeply study and use QtQuick as well as different C++ styling systems based in Qt, both procedural and primitive-graph based. We also discussed several use cases among us and with our friends at QtDF as well as in open lists like `plasma-devel`. With that knowledge we came up with a solution that we are confident to be in the right direction.

That said we are eager to have the feedback and collaboration of the KDE community on a tool that plays important role in KDE development.

The proof of concept code is available on Gitorious[4], clone it, change it. Also, you can always find us in `#qt-labs` at Freenode.

6. ACKNOWLEDGEMENTS

The content exposed in this paper result from the cooperative work that has been done by the author together with their co-workers at the openBossa labs and the developers from Qt Software.

I would like to thank everyone that contributed to this work. In special the KDE e.V. for their support, our co-workers Anselmo Lacerda Silveira de Melo, Artur Duque de Souza, Caio Marcelo de Oliveira Filho, Jesus Sanchez-Palencia, Kenneth Rohde Christiansen, and Renato Chencarek. Thanks also to those at Qt Development Frameworks: Marius Bugge Monsen, Jan-Arve Sæther, Alexis Ménard and Leonardo Sobral Cunha.

7. REFERENCES

[1] OpenBossa Labs - http://www.openbossa.org
[2] Qt - http://qt.nokia.com
[3] Qt Quick - http://doc.qt.nokia.com/4.7-snapshot/declarativeui.html
[4] Qt Components (Style branch) - http://gitorious.org/qt-components/qt-components/commits/style
[5] Alien Windows - http://labs.trolltech.com/blogs/2007/08/30/say-goodbye-to-flicker-aliens-are-here-to-stay
[6] Property Binding - http://doc.qt.nokia.com/4.7-snapshot/propertybinding.html
[7] OpenBossa YouTube - http://www.youtube.com/openbossa
[8] Author's Blog - http://eduardofleury.com