The Unusual Suspect: Layouts for sleeker KDE applications

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

Caio Marcelo de Oliveira Filho
openBossa/INdT
Recife, PE, Brazil
caio.oliveira@openbossa.org

ABSTRACT
This paper complements the homonimous talk presented at Akademy 2009. It covers flexible layouts, self-animated layouts and animation between different layouts, recent features we have been developing for Qt framework, together with Qt Software. Subjects include the use cases that drove us, implementation constraints we had, APIs, usage and the benefits that arise from the use of such technologies.

GENERAL TERMS AND KEYWORDS
Flexible Layouts, Custom Layouts, Layout Animation, Declarative UI

INTRODUCTION
The Qt framework has in its graphics subsystem the notion of *Layouts*, which are entities responsible for arranging the visual elements, usually Widgets or other Layouts, on the screen.

A given Widget can delegate the arrangement of its children widgets to a certain Layout, this delegation is referred as *installing* the layout on the widget. By moving the arrangement logic into a separate class, it becomes possible t *Layouts* o reuse that logic in different contexts, for different Widgets and children.

It is not enough to install the layout in the widget, it is also needed to add the items to the layout, since the layout may need more parameters to know how to arrange the items (for example, the position on a grid).

The objective of this article is, first, to describe our explorations of the Layout idea:

1. Implementation of a different and powerful layout, based on the idea of *anchoring* sides of children to each other.
2. The use of Custom Layouts to create specific interfaces from Interaction Designers specifications.
3. The animation of screen states using layouts.

And second, to illustrate how these enhanced uses of layouting can be used to improve KDE and Qt applications, making them not only more beautiful but also keeping the benefits of layout usage.

Layouts in Qt
In the framework there are two families of visual elements, `Qwidget` which represents the traditional Qt GUI elements, and `QgraphicsWidget`, built on top of the Graphics View system. In this article, we'll assume the Graphics View family of widgets and its base layout class, `QGraphicsLayout`, since our concrete work is built on that. However most of the discussion is also valid for traditional family of widgets.

In Qt 4.5 (the latest release as of today), there are two different basic layouts implemented in the Graphics View system: `QGraphicsLinearLayout`, which essentially arranges the items side by side (or vertically one on top of the other), and `QGraphicsGridLayout`, which allow a more complex arrangement based on a table-like structure. Curiously the linear layout is implemented using the same engine as for the grid, but providing a simpler API.

Layouts in KDE
Applications in KDE make use of the existing layouts in Qt, but for specific user interfaces, some custom-made layouts were developed. For `QWidget`, most of the layouts are variations of *Flow Layout* example in Qt, which arranges the items in a similar way to the one in which words are arranged in a paragraph or items are arranged in a HTML page.

The Graphics View subsystem has a few different layouts implemented, most of them as part of *Lancelot* [9], the most interesting of them is the `NodeLayout`, which will be explained better in the next section.

As of today, there is no work related to animation based on layouts, in neither of the senses that we are going to explain in this text: animation inside a custom layout or animation between two layouts.

FLEXIBLE LAYOUTS
Flexible layouts are layouts in which most of the behavior is configured after the creation of the layout. In other words, part of the implementation of how the items are going to be arranged is done by calling methods on the layout object.

With an API to configure the items in the layout it is

possible to make high level tools usable by designers. This makes it possible for them to setup the application layouts rather than requiring a C++ developer for that task.

A Flexible layout is more a tool to build layouts than a ready to use layout (which you just add and remove items). Note that this classification is not ``all or nothing'', for example, the `QGraphicsLinearLayout` can be used by just adding the items, but also have some parameters configured, like margins and item spacings.

Concrete Implementations

There are different implementations of this Flexible layout idea. One example, outside Qt and KDE, is the Edje layouting mechanism. Edje is part of the Enlightenment Foundation Libraries [10], a set of lightweight libraries used for E17 window manager and also many embedded applications.

Edje layouting is done by defining a set of areas defined by coordinates relative to the parent widget[1] or to other areas in the layout. Edje also has support to define different states to those areas.

Then items can be added to those areas, in most cases images, widgets from other existing libraries and other Edje layouts. For many interfaces that we dealt with, this proved to be more effective than mapping the designer's mockups into vertical and horizontal boxes.

A similar layout present in KDE codebase is the `NodeLayout` used in Lancelot. The idea is that you add items with a pair of \textit{nodes} which are points representing the top-left and the bottom-right corners. It is simpler than Edje, in the sense that the nodes can be specified relative to the parent widget, but not relative to other nodes.

Anchor Layout

Another example of Flexible layout is the `QGraphicsAnchorLayout`, a layout that is being developed by openBossa together with Qt Software and should be integrated at the Qt main repository soon. Its development was motivated by two main goals: to provide a way of designing layouts that is accessible to UI designers and to act as a powerful layout for applications that currently set their geometries manually or with a series of simpler layouts.

In most cases, one Anchor Layout can substitute a series of nested Vertical and Horizontal linear layouts, the layouts that are normally used to build complex arrangements of items on the screen.

An anchor layout is based on two concepts: *anchorage points* and *anchors*. The former are special points each item have (right, left, top and bottom edges, as well as its center). The latter represent links between two anchorage points and can be created from one item to another, or from one item to the layout frame itself. The logic is the same for vertical and horizontal anchoring, so we'll illustrate only one dimension.

For example, one can anchor the right side of item A with the left side of item B. So when the layout set the geometry, one restriction that the layout will respect is that the two items are glued together. In other words, the layout will enforce that the distance between these two edges is equal to the length of the anchor created between them.

It follows that anchoring the item to the layout in both sides, will make the item grow together with the layout. This property allows us to configure the items in a way they will scale, which is a desired property for a Qt layout.

Linear programming

There are other implementations of anchor-based layouts in other libraries but our understanding is that most of these have limitations that prevent them from being used by end users. Some like Java's SpringLayout seems to work as a tool for layout designers to use as their engine. Others while targeted at the end user, require too much concern from him with regard to the way the anchors are created. In many cases a slightly misplaced anchor leads to a situation where the layout is unable to calculate its geometries.

Given that experience, when developing an AnchorLayout for the Qt framework, we decided we would like a layout that were both robust and easy to use.

While such a flexible layout provides countless possibilities to the users, it also allows them to fall into linear dependencies, inconsistent setups and other corner-cases that could risk the whole user experience. Avoiding such situations was an obstacle the implementation of such layout had to deal with.

An anchor layout is fun to use as long as the user does not have to worry about

anchor dependencies or the order in which he creates them. That meant the layout itself could not make hard assumptions of how would the anchors be created and instead, it had to be solid enough to withstand some sort of abuse by those creating the anchors.

Given such situation, we decided to use a Linear Programming approach to optimize the distribution of the items. From the items and anchors the layout identifies its constraints and its objective function which are then passed to a Linear Programming solver. While that might seem as an implementation detail, it explains why `QGraphicsAnchorLayout` handles anchor setups that cause similar layouts to fail.

[1] This terminology is not exactly the one used by Edje developers, but the analogy holds. In Edje the layout itself also acts as the parent widget.

CUSTOM LAYOUTS

Another important family of advanced layouts are those we call *Custom*. In opposition to the very versatile layouts we talked about earlier, these are rather specific classes, tailored to the very special needs of each situation.

When comparing the two families of layouts, there is no answer to the question regarding which one is the best. Instead, the idea is to understand the differences between them so the right tool for the right task can be used.

Creation

Important differences exist between Flexible and Custom layouts, among those is the different creative processes that are associated to each one of them. The creative process for Custom layouts is different because each Custom layout is implemented as a layout class of its own. This means that is not possible to instantiate a ready-made layout and configure it as we did with AnchorLayout, for instance.

As before, the high level ideas come from the designers. Usually though, those are not able to use standard programming languages to implement the actual layout classes. This means that developers must collaborate in this creative process.

Developers must come up with a procedural method for calculating the geometries of the items on the screen, based on the designers feeling of what is beautiful. In other words, they must come up with an algorithm to model their way of thinking.

The algorithm is then implemented in the form of a layout engine. This is usually harder and more costly than allowing the designers to configure existing layouts. As an example, we could have a high level tool to help designers create the anchors for an AnchorLayout, while for custom layouts, C++ knowledge is required.

Usage

While the creation of a Custom layout can be complex, it pays itself in some situations.

The first advantage is that once a Custom layout class is created, it can be used like a ready-made Qt layout. All that is needed is to instantiate the class and add items, then its up with the engine to do its job.

Furthermore, depending on the way the engine was created, the addition and removal of items is straightforward. Simply add or remove the items and it works out of the box. Compare that to the use of AnchorLayout, in that case new Anchors would have to be created each time an item was added.

Finally, it is interesting to use specialized layout engines when the desired behavior has a complexity that goes beyond the boundaries of what Flexible layouts can do.

For instance, in the Canola Layout example [5] the addition or removal of an item triggers a complete change in the layout organization. In this case it was easier to create an appropriate engine than to deal with creation and destruction of several anchors each time an item was added or removed.

Animations

The Canola Layout example shows another feature that is often a requirement in applications with rich UI (user interfaces) – animations. Designers can ask developers to animate the addition of items, their removal and finally, changes in their position.

When creating Custom layouts, it is easy to add such animations, and further than that, it is easy to keep the dirty work inside the layout class itself. By keeping all the magic hidden, we ensure that users only have to worry about *using* the layout, nothing more.

LAYOUTS OF WIDGETS?

The idea of Custom Layouts described before has some similarities with the existing widgets dedicated to work as *containers*, like the classes of Qt ItemViews or ItemViews-NG (a research project in Qt Labs).

A *container widget* is a widget focused in displaying a set of data (in Qt usually this data is available through a Model class). The most common container widgets are lists and grids that create visual representations once given the data representing their elements.

These representations can be customized, by using factories of widgets for handling elements, in Qt4 the factories are called *delegates*, and in ItemViews-NG they are *item creators*. The factory creates a visual element for a given piece of data.

If you think about the previous Canola Layout example, assuming that we have a list of icons and that items would be represented by those icons,

a different implementation could be: to use the same engine as before, but implementing a container widget. The parent (container) widget would accept a list of icons and use a factory to create the necessary children widgets.

Other examples of container widgets are:

- a taskbar: the data is the window information, and the factory creates buttons to represent that window;
- a grid of photos: the data is a list of pictures, and the factory creates a widget that can show one of these pictures.
- a list of songs: the data is a list of song objects, and the factory creates a widget that show selected information from song objects.

Note that when using those specialized container widgets, you can abstract away the presentation elements -- and even encapsulate them in the container widget. The focus is to provide the correct data so the view will show it correctly. This separation is present in both Qt4 and ItemViews-NG and allows to easily swap the visual representation.

The similarities with layouts lie mostly on the fact that both Layouts and Container Widgets are responsible for positioning the children on the available screen space. However, the layout is just that, while the container widget is a more complex entity.

Essentially, it is possible to derive a Container Widget from a Layout and a Factory that is capable of converting model information into proper widgets. Using layouts is more adequated in the case that the model information is complex enough that it is better encapsulated in widgets. For example, a complex taskbar integrated with a notification area API, in which the different windows would be represented by custom widgets. In this case a layout is more fit to the job than making a widget that accepted other widgets as models.

Sometimes you can both create a custom layout or create a custom container widget, for example, the Canola Layout previously shown, could be designed as a parent widget that set the children position using the same engine as the layout. If the application does not need arbitrary widgets in the Canola grid, the parent widget could simply have a method to add a new item for a given image, or even having this information in a separated model.

The solutions provide different benefits, the layout can be reused in other contexts, with arbitrary widgets; on the other hand the container widget can provide specialized ways to add children and can result in a simpler code if the children widgets are all similar.

MULTIPLE LAYOUTS

In rich UI applications, designers try to make application events be represented in a nice way on the screen. This usually requires widgets to grow, shrink and move around, into or out of the screen.

As a result we have that, while a given layout can suit the distribution we need in some situations, the same may not be true when a change in the items distribution is required. In that case, we may need not one, but several layouts, one for each distribution. In other words, we would like a group of layouts to swap among.

Unfortunately though it was not easy to swap between layouts in Qt. In the following subsections we explain the most important changes made to make that possible and fulfill the requirements set.

From Qt Software we had two important requirements. The first was not to break binary compatibility, this is only acceptable between major release versions. The second was to support existing third-party layouts, rather than simply modifying the built-in Qt layouts to support the new functionality.

The Desired Usage

Our goal was to be able to create all the layouts we wanted only once and then use them appropriately, talking high level, see the following example:

Setup:

- Create the items of my screen (two buttons, a text box, etc).
- Create the first layout (a linear layout, for instance).
- Fill the first layout with the items in the correct order.
- Create the second layout (a grid).
- Fill the second layout with the same items, in a different distribution.

Usage:

- Apply the first layout.
- When needed, take the current layout out of the widget and apply the new one, repeat.

Taking a Layout from `QgraphicsWidget`

In the existing versions of Qt, there was no way of removing a layout from a `QgraphicsWidget` without causing it to be deleted. The same is true for `QWidget` class, which is out of the scope of this document. Therefore, if one wanted to keep switching between two layouts, he would have to keep iterating on the following process:

- Create a new layout.
- Add widgets to it.
- Apply the new layout in the widget (causing the deletion of the previous one).

This was not practical nor elegant. That's why we added the `QGraphicsWidget::takeLayout()` method to remove a layout from a widget without deleting it.

Adding an Item to Several Layouts

At that point we had the ability to remove a layout from a widget to reuse it later. However, we still could not create several layouts and keep them ready to use.

In order to do that, we would have to add the same set of widgets to more than one `QGraphicsLayout` at the same time, what was not possible in Qt. Talking to the developers at Qt Software, we came up with the solution of creating a new kind of layout element that would act as a proxy between the actual widget and the layouts.

The idea was to create several layout proxies and have all of them pointing to a single `QGraphicsWidget`, then we would add the proxies to the different layouts (one proxy in each layout) instead of trying to add the same widget to more than one layout.

For example, we would like to add a button to both a linear layout and a grid layout. In order to that we now create two proxies, add one of them to each layout and finally, have the proxies pointing at the shared button widget.

While that is not as straightforward as simply adding the same buttons to several layouts, this approach required less intrusive changes in Qt and supports existing layouts, as explained earlier.

Animation Between Layouts

We were already able to switch between layouts, and that

was good. However there are the designers, and with them, the question: “OK, you have two layouts, but it is so ugly to swap them at once! Can't you just ask them to animate too?” “Sure..., we said.

The idea once again was to add animation support to all existent layouts,thus modifying `QGraphicsLayout` instances was not an option. The solution was to add animation capability to the same proxies that allowed us to solve the previous issue.

Proxies were then blessed with `QAnimations`, and so their task became slightly different. They were then responsible for running an animation between the current item geometry and the new one, rather than simply forwarding the `setGeometry()` call as they used to do.

LAYOUTS IN DECLARATIVE UI

At the time of this writing, the layouts available for use in Qt Declarative UI are not compatible with `QGraphicsWidgets`. Instead, they are based on an alternative item class called `QFxItems`, which is out of the scope of this document.

Declarative UI is under heavy development and work is being done with regard to the addition of standard `QGraphicsLayouts` support. In the following sections we assume this will be ready in the short-term future. We understand that support for both Flexible and Custom layouts should be available in Declarative UI.

To enable the use of Flexible layouts we must ensure that there is proper API to configure such layouts. To declare the anchors, for instance. Custom layouts on the other hand, are implemented as third-party classes. This means that Declarative UI files must support custom keywords and custom user classes for the latter to work.

Property Propagation

The Qt Declarative UI project introduced a powerful feature called Property propagation. It allows users to associate *script expressions* to object properties. This means that properties like transparency, color, position and size, that otherwise would be attributed fixed values, can now be bond to a mathematical expression involving values from other items' properties and constants, to be constantly evaluated.

Such feature can be used to bind one item's behavior to those of others. For instance, one item can be set to follow the color of another one. Or else, its size can be defined to be the sum of the sizes of other two. In the layouting context, property propagation can be used in a similar fashion to how Anchor Layout is used, it is easy to simulate part of the features of the anchors by setting the geometry properties of the items involved.

In Declarative UI there are properties that define the X and Y position of an item. Once we define that an item has its X position equal to the right edge of another item, we have created an anchor. Similarly, we can define the width of an item to be equal to 2/3 the size of another item. That is how Edje's parametric layouting system work.

So, how do we compare that to the layouts we explained before? These methods are complementary, with the following differences:

Real layouts:

- A separate entity that takes care of the items.
- Only handles geometries.
- Uses a specialized C++ implementation.

Property propagation:

- No external entities are required.
- Handles anything that can be exported as a property.
- Implemented using Qt Script and recursion.

From those differences it is important to note that for some uses like binding colors, transparencies, states and everything apart from geometries, property binding is the solution. When it comes to geometries though, some care must be taken.

By using a specialized implementation, layouts can be faster and more powerful than properties. When using property propagation to bind together the geometries of several items, it is easy to follow in a circular dependency problem or to loose track of the intended setup. Furthermore, the code can become hard to maintain with geometry constraints spread among the declaration of distinct items. In such cases, having an entity whose only task is to manage geometries is a faster, more maintainable and more scalable solution.

CONCLUSION

From our experience with Qt Layouts, we understand that these layouting tools can be used in existing and new KDE applications in order to improve their appearance as well as the user experience they provide.

Part of our current work is being integrated right into the Qt main repository. We are also working on examples based on real KDE applications that can benefit from such tools and will release them as soon as they are ready.

ACKNOWLEDGEMENTS

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

We 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, Jesus Sanchez-Palencia and Renato Chencarek. Thanks also to those at Qt Software: Andreas Aardal Hanssen, Jan-Arve Sæther and Alexis Ménard.

REFERENCES

1. OpenBossa Labs. http://www.openbossa.org
2. OpenBossa Channel on YouTube. http://www.youtube.com/openbossa
3. Qt Software. http://www.qtsoftware.com
4. AnchorLayout in Gitorious. http://qt.gitorious.org/+openbossa-developers/qt/openbossa-clone/commits/anchorlayout
5. Canola Layout. http://www.youtube.com/watch?v=eJcTBJaPRZg
6. Eduardo Madeira Fleury's Blog. http://eduardofleury.com
7. Linear Programming. http://en.wikipedia.org/wiki/Linear_Programming
8. Ivan Cukić post on NodeLayout. http://ivan.fomentgroup.org/blog/2007/10/13/nodelayout-for-plasma/
9. Lancelot Application Launcher. http://lancelot.fomentgroup.org
10. Enlightenment Foundation Libraries. http://www.enlightenment.org/p.php?p=about/efl