fpGUI - A technical reference

Graeme Geldenhuys

June 18, 2007

Contents

Chapter 1

Introduction

After developing many cross platform applications with Kylix and Delphi I started getting very frustrated with the differences between the look and behavior of the applications under Linux and Windows. The code was also riddled with IFDEF statements.

Then I stumbled across the Free Pascal and Lazarus projects. I thought this is it, the answer to all my cross platform development problems. Unfortunately after working with Lazarus for a few months I started finding more and more issues with the widget sets, though the IDE was great.

The Lazarus LCL is a wrapper for each platforms native widget set. This brought with it the same issues I experienced with Kylix and Delphi. This got me thinking about how I could resolve this issue.

Then it hit me - implement the widget set myself using Free Pascal! Painting the widgets myself to get a consistent look and implementing a consistent behaviour. Instead of reinventing the wheel, I thought I would do some searching to see if there is another project I can contribute to or help by giving me a head start.

The first version of my widget set was based around a heavily modified version of the Light Pascal Toolkit[1]. I then discovered the discontinued fpGUI and fpGFX projects. I tried to contact the original author to no avail. The fpGUI code hasn't been touched in four years (since early 2002). After studying the code for a few weeks, I came to the conclusion that fpGUI is much closer to what I strived to accomplish with my modified LPTK. A lot was still missing from fpGUI though.

After thinking long and hard, I decided to start my widget set again, but this time based on the work done in fpGUI and fpGFX. I also added to the mix some good ideas I saw in Qt 4.1. So far I have completed quite a

[1] http://sourceforge.net/projects/lptk

few things missing in fpGUI, but I still need to do a lot to get to the point where I can test drive it in a commercial application. I set myself a list of things outstanding which should get it to a usable level. I also added a lot of documentation as I went as there was no documentation included with the original fpGUI and fpGFX projects. Documentation is important to attract other developers in using the widget set.

Chapter 2

GUI and Events

This chapter is currently a copy and paste of an email I wrote explaining how the events work. The difference between the GFX events and the GUI events. Soon I will rewrite this chapter and add a lot more detail. Here follows the email text for now.

The GFX part gets events from the windowing system, be that X11 or GDI. That means that TFCustomWindow gets all the events from the windowing system.

Now the GUI part. Every widget is *not* a TFCustomWindow, so every widget doesn't have a Handle. The GUI is implemented with only one handle per Form (TFForm). All widgets are just painted onto the canvas of the TFForm.

If you look at the TFCustomForm you will see it has a instance of TFCustomWindow stored in FWnd. That instance gets all the events from the windowing system. So to let the widgets also get events we have to implement our own event system. The TFCustomForm will then send those custom events (TEventObj descendants) by translating the windowing events into our custom GUI events.

For example:
Lets say we have a Form with a Button on it. Now we want to handle a OnMouseMove event on the Button. The flow of events will go as follows:

- TFCustomForm.Wnd will receive the OnMouseMove from the windowing system and process it in the WndMouseMoved() method.
- WndMouseMoved() will translate that windowing system event into whatever GUI events are needed and start sending them.
- Any TWidget descendant handles events in the ProcessEvent() method. TWidget being the big one.

- Because TFCustomForm is a TFContainerWidget descendant, it means it can contain other widgets. So it starts distributing the GUI events to the children. Distribution is done by the DistributeEvent() method.
- If TFCustomButton needed to do any special processing with the GUI event, it would handle it in its ProcessEvent() method. TWidget.ProcessEvent normally does most of the generic work.
- As an example of a widget that does custom processing, have a look at the TFMenuItem.ProcessEvent(). In this case it handles the TMouseEnterEventObj and TMouseLeaveEventObj events so it can changes it's look when the mouse enters a menu item or leaves a menu item.

So as a summary:
TFCustomForm contains a instance of a GFX window. Translates all the GFX events (underlying windowing events) to GUI events (TEventObj descendants) and distributes them to the children of the Form.

Chapter 3

Layout Algorithm

3.1 Initialisation of a window (form)

If a window presents itself for the first time, and no standard size was given, then it must compute these. This is a recursive process, with which the event TCalcSizesEventObj is set for all children of the window from top to bottom in the Widget tree (beginning with the window). TWidget reacts to the receipt of this event as follows (in TWidget.EvCalcSizes):

- The event is passed on with TWidget.DistributeEvent to all children.
- The virtual protected method TWidget.DoCalcSizes is called. Again derived Widgets overwrites this method, in order to compute its sizes of (minimum, maximum, optimum).
- The results of DoCalcSizes are if necessary corrected, e.g. the maximum size may not be smaller than the minimum size.
- If the code for the window finished the dispatch of this event, all Widgets in the window has valid statements of size. Now it can do its own, initials size sets (this is the before computed optimum size of the window). This is accomplished by TWidget.SetBounds.

... to be continued ...

Chapter 4

Single vs Multi handle decision

... to be written ...

What follows below are just some snippets from the newsgroup and mailing lists that I want to reorganise into a chapter. These are things that were discussed that I don't want to forget.

4.1 Notes from the newsgroup

One handle per Form. Widgets are just painted onto the Form canvas. This will support the most platforms and give us a consistent behavior.

See the /prototype/multihandle directory.

I spoke to Martin Schreiber (MSEgui author) about this as well and the better way seems to be to have one handle per Form (window) and do you own custom events. Implementing as much as you can yourself, will give you a consistent behavior on all platforms and make it more portable.

Martin also tried to implement one handle per widget and came to the following conclusion with makes sense. If you want you GUI Framework to work the same across multiple platforms, you cannot rely on specific behavior on a platform. With one handle per widget you are relying on the underling windowing systems event handling and may very well differ from another platform (which it normally does). This is what Martin experienced, and the only way to get consistent event handling was to do if yourself. Clipping regions was another issue Martin mentioned.

All his points made perfect sense to me, so I continued with the fpGUI based on one handle per Form and will guarantee that all events, and clipping regions will work exactly the same on all platforms. Just creating the simple

prototype I already experienced different behaviors between Windows and Linux.

See the thread titled "Comments from Martin Schreiber" dated 2006-12-07. There I posted some of the discussions with Martin.

But then I remembered a very strong argument for one handle per window: Some platforms I would like to see fpGUI running on the future simply don't support one window inside another. Like Linux Framebuffer for example. I'm not sure about Symbian OS UIQ 3, but I think too.

4.2 Email from Martin Schreiber

Pro's for to have a window handle for every widget: It is possible to use more code from the operating system (clipping, focus handling, mouse enter/leave...

Con's: The functionality of your widgets depends on the OS. It is not easy to achieve the same behavior with different systems. You have less control over the behavior of the widgets. If there are very much widgets in a application there can be performance and resource problems (in win95/98 the maximal count of gdi objects and window handles is limited). You probably need to implement simplified widgets without handles too (TGraphicControl) which brakes orthogonality.

I did three attempts to develop a GUI environment, the first approach was based on VCL, the second on CLX and now MSEgui which is done from scratch. With VCL and CLX I needed very much time to find workarounds to change their behavior, some things where simply not possible to realize. With about the same expenditure of time I have reached in MSEgui a really enjoyable level. As an example, the implementation of transparent widgets took me weeks in VCL/CLX, semi-transparency was almost unreachable. In MSEgui I could implement transparent widgets in 10 minutes because my self developed clipping handling was flexible enough.

So my tip is to do as much as possible by your self, at the end you will need less time and the quality will be much better. But don't underestimate the expenditure of the project. For comparison: I invested about 10'000 hours into the development of MSEide+MSEgui up to now.

Martin

Chapter 5

Database Components

... to be written ...