Current GTK+ development

Matthias Clasen

mclasen@redhat.com

Table of Contents

What is GTK+ ?

GTK+ is a fully featured toolkit for creating user interfaces. It relies on the Pango library for text rendering, which provides excellent internationalization support. GTK+ was designed with language bindings in mind, and can be used with many programming languages, for example C, C++, Python, Perl, Java, Ada. Over time, GTK+ has spun off a number of auxiliary libraries.

GLib is the low-level core library that forms the basis of GTK+ and GNOME. It provides data structure handling for C, portability wrappers, and interfaces for such runtime functionality as an event loop, threads and dynamic loading. It also offers support for character set conversion and Unicode handling.

GObject is an object system written in C with traditional features as inheritance, polymorphism and reference counting. It also contains a signal system for notification and an object attribute system.

The ATK library provides a set of interfaces for accessibility. By supporting the ATK interfaces, an application or toolkit can be used with such tools as screen readers, magnifiers, and alternative input devices.

Pango is a library for layout and rendering of text, with an emphasis on internationalization. It forms the core of text and font handling for GTK+. Pango shields from the details of the platforms native font system.

The GDK library provides a layer of abstraction that sits between GTK+ and the underlying windowing system. Instead of making calls directly to the window system, widgets call GDK when they need to draw to the screen or handle events. The GDK rendering API pretty closely matches the Xlib API.

The GTK+ library contains the widgets which make up the toolkit, together with supporting code, such as a theming system and drag-and-drop support.

GTK+ Widgets

GTK+ offers a pretty complete set of standard controls like menubars, toolbars, statusbars, entries, buttons, spinbuttons, labels. It also has a reasonable supply of layout containers like tables and boxes, which are responsible for arranging their child widgets on screen. Since 2.0, GTK+ also contains a number of more complex widgets following the model-view pattern: GtkTextView and GtkTreeView. Recently GtkComboBox also joined the family of model-view widgets.

GTK+ History

GTK+ was started as toolkit for the GIMP around 96 and reached its first stable release in April 98. GTK+ 1.0 contained the basic widgets that were needed to support the GIMP. The next stable release, 1.2, February 99 contained many new widgets which made GTK+ a reasonable toolkit to choose for general application development, it was no longer Gimp-centric. 1.2 was also the first release which featured a separate GLib library.

After 1.2, GTK+ went into a long development cycle, during which a lot of things were done. Text rendering was moved to use Pango, yielding first-class internationalization support. The object system was generalized and moved to GLib under the name GObject. A backend separation was introduced in GDK, and the win32 backend was added. Two big new widgets, the text view and the tree view, were created from scratch. Both feature a model-view architecture. During this 3 year period, Gnome was eagerly waiting for GTK+ 2.0 to get ready, since Gnome 2.0 depended on it. One of the lessons which the GTK+ team learned from the 2.0 release cycle is to try to stick to shorter 9-12 month development cycles between stable releases.

We haven't reached that goal for all 2.x releases, but we have successfully avoided multi-year development cycles since 2.0.

The releases after 2.0 had more of an incremental nature. The main new feature in 2.2 was multihead support, for traditional X11 multiscreen/multidisplay and Xinerama.One interesting aspect of the multihead support in GTK+ is that it allows you to move windows between screens and displays, a feature which only few toolkits support today. The current stable release 2.4 features a new, much anticipated new filechooser widget, as well as a new combobox, and some widgets which were "brought home" from other places in the Gnome library stack.

To get an impression of the size of GTK+, here is a rough count of the lines of code (created using David A. Wheelers sloccount utility). These numbers include GLib, ATK, Pango and GTK+:

- 1.0 (Apr 98, ca 93.000 lines of code)
- 1.2 (Feb 99, ca 160.000 lines of code)
- 2.0 (Mar 02, ca 460.000 lines of code)
- 2.2 (Dec 02, ca 488.000 lines of code)
- 2.4 (Mar 04, ca 558.000 lines of code)

GTK 2.4 Additions

The new file chooser has different modes for opening files, opening directories and saving files. What you can see here is the "file open" mode. You can also see that the entry for manually entering the pathname is no longer shown in the dialog itself, but has been relegated to a secondary dialog, which can be opened using the key combination Control-L. This design decision is still hotly debated in some circles, but it has lead to a much less cluttered file chooser UI. Note that the entry offers filename completion with a completion popup, as commonly seen in web browser location entries.

The new combo box widget features different styles, it supports the traditional optionmenu style with a popup menu - which can now also be organized in columns, as the screenshot shows. The alternative style is derived from the appearance of combo boxes on Windows.

Current Goals of GTK+ development

The longterm goal for the development of GTK+ is to provide a complete platform for the development of GUI applications. In order to achieve this, we have to close some gaps. Some of the commonly used widgets which are currently missing in GTK+ are

- a model-view table widget
- an icon list, like the icon view seen in Nautilus and other file managers
- a dock widget as seen in IDEs
- a print dialog
- an about dialog
- wizards

Implementations or at least prototypes for some of these widgets are already available somewhere in the Gnome library stack, and just need to be cleaned up and moved to GTK+. For instance, there is an icon list widget called EggIconList in libegg, and an about dialog named GnomeAbout in libgnomeui. In some other cases, the available implementation will have to be reworked more throughly before they can go into GTK+. An example of this is the print dialog. While there is an implemenation in libgnomeprintui, the plan for a print dialog in GTK+ is to first move to Cairo as the main rendering API, and then use the print support in Cairo. There are also several dock widgets available in various parts of the Gnome library stack, which need to be carefully compared to come up with a reasonable scope and feature set for a GTK+ dock widget. One complication of this particular widget is that there may actually be several non-compatible sets of requirements for a dock widget, coming from the use cases of IDEs and office style applications. Finally, some widgets will have to be written from scratch. An example of this is the model-view table widget, although it would be desirable to reuse parts of the tree view widget for it, for instance the cell renderers.

Beyond simply adding more widgets, there are some important features which you would expect in a modern toolkit, which are currently absent from GTK+. Among them are:

- support for session management
- loading widget hierarchies from textual descriptions, like libglade does
- more flexible layout using width-for-height geometry management

User interface design keeps changing. Current trends like breaking up the window metaphor, introducing transparency and animation throughout the UI, will force GTK+ to evolve.

We are confident that we can achieve most of the goals mentioned so far while maintaining binary compatibility with GTK+ 2.x.

GTK+ 2.4 maintenance

Fixing bugs

The focus of 2.4.x development is naturally bug-fixing, since it is the stable series. New features will appear in GTK+ 2.6. Fixing bugs in a mature codebase like GTK+ is not always easy, as the following examples illustrate. Some bugs are even unfixable. But still, an impressive number of bugs is fixed. During the 2.4 development phase (December 2002 - March 2004), we fixed over 1000 bugs in the stable GTK+ 2.2.x branch. This number was derived by counting references to bugzilla bug numbers in the main ChangeLog files of GLib, Pango and GTK+, so the actual number may even be a bit higher (due to bugs without bugzilla entries, and bugs in the documentation).

Borderline case

Focus drawing in buttons.

Some time ago, we noticed a bug in the size allocation function of GtkButton. The visual effect of the bug is only noticable in styles which use wide lines to draw the focus indicator, which is actually required for some accessability-oriented themes. As you can see, the child of the button (which in the example is a color swatch) is drawing over the focus indicator. The bug is simple to fix, but as a consequence, buttons actually need more space, since they now reserve extra space for the focus indicator. Immediately after 2.4.2 was released, the GIMP developers pointed out that this change is problematic for the many small buttons used in the GIMP user interface. As a compromise, the button size allocation was changed again in 2.4.3 to only request the extra space if the button can actually take the focus. This fixes the issues noticed in the GIMP, since the problematic buttons there are not focusable, and we haven't heard complaints about the new behaviour from other GTK+ users.

Performance improvements

While the performance of GTK+ is reasonable on modern hardware, we are always looking at ways to improve the percieved performance. Things we are currently looking at include predictive exposes, unsetting the background when mapping windows, and reducing the overhead of signal emissions. These patches will first appear in the HEAD branch of GTK+ (which will eventually lead to 2.6), but it is not unlikely that we backport them to GTK+ 2.4 after they have proven stable.

Predictive exposes

To understand what predictive exposes are and how they can help the performance of menus, one has to know the normal sequence of events when a window is mapped: The application sends a MapWindow request, which the X server sends on to the window manager. The window manager decides where to place the window, and sends another MapRequest for the window to the X server. This time, the X server maps the window and creates a MapNotify, followed by Expose events which cause the application to draw the contents of the newly mapped window. You will notice that this includes two roundtrips to the X server. But X also has the concept of override-redirect windows, for which the window manager is not involved in the mapping process. For these, there is no real need to wait for the Expose events before starting to draw, since we know that they will be send. Thus, predicting the expose events allows us to get rid of the context switch to the X server and back between popping up the window and drawing it, which will make it less likely that some other process gets scheduled in between.

Unsetting the background

When the X server maps a new window, it fills it with the background color or pixmap, then sends Expose events and waits for the application to draw the contents of the window. This can cause noticable flicker, e.g. when switching between tabs in a GtkNotebook. Fortunately, X allows to set the background of a window to None, and doesn't do any initial filling when windows with background None are mapped. This is not a problem for GTK+, which draws its own background color anyway, and it completely eliminates the flickering when switching notebook pages.

Reduce signal emission overhead

GTK+ makes heavy use of signals. These Signals are a generic callback mechanism implemented in GObject and have nothing to do with traditional Unix signals. Emitting a signal involves analysing varargs function arguments, and marshalling them in a form suitable for use by the connected callback function. Thus a signal emission has a considerable overhead when compared to a regular function call, and it be worth to avoid the signal emission completely if there are no callbacks connected.

GTK+ 2.6 development

The next stable release of GTK+ will be 2.6, which is planned for December 2004. This release will continue the theme of the 2.4 release, with some small additions, and a focus on improving the new widgets from 2.4.

We are still improving the new file chooser, to make sure that it works really well. The file chooser has produced and is continuing to produce an amazing number of bug reports and enhancement requests. We can't possibly implement every feature that people apparently want to see in a file chooser dialog. But a number of enhancements are very likely to appear in 2.6. The file chooser will share the setttings controlling the display of hidden and backup files, single vs. double click and the shown columns with the Nautilus file manager. The bookmarks system will be made more convenient by making it automatically maintain bookmarks for the most recently visited places. In the save mode, applications will have a way to let the user select the format to save in, as shown in the screenshot.

The new GtkComboBox widget needs a to gain number of features before it can fully replace all uses of its predecessor, the GtkCombo widget. The most notable missing features are separators, insensitive items and scrolling. Insensitive items are already working in cvs HEAD, as the screenshot demonstrates.

A number of features have been prototyped already and are likely to find their way into 2.6. These include commandline argument parsing support for GLib with the goal of replacing the popt library currently used for this purpose in Gnome. There is a prototype of an icon list widget under the name EggIconList in libegg. A progress cell renderer based on the one found in the Epiphany has already been incorporated in the HEAD branch of GTK+. We will probably also try to continue moving generally useful widgets from libgnomeui to GTK+. The candidates are a file chooser entry, which is a combination of an entry with completion for filenames and a button to bring up a file chooser dialog, an image or icon chooser, and a datetime widget, which combines an entry for entering a date (and a time) with a button to bring up a calendar widget.

Pango will make a detour from the release cycle of the other libraries. Pango 1.6 will be released in time for Gnome 2.8. Its noteworthy features will include rotated text, custom font decoders (i.e. use fonts without Unicode mapping).

The future

After the 2.6 release, we will concentrate on Cairo support in GDK. Cairo is a relatively new graphics library that closely matches the future rendering needs of GTK+. It is designed to be an easy to use 2D graphics library offering a rich set of capabilities and multiple output backends.

The Cairo API is similar to PostScript. The example shows how to construct a path from points. Cairo also has functions to create common types of paths like rectangles, arcs or circles. Cairo goes beyond PostScript by including complete alpha-compositing (not shown in this simple example, but there are many more interesting ones on http://cairographics.org).

In rough terms, the rendering capabilities of Cairo are comparable to those of Java 2D, SVG or PDF 1.4. In detail, Cairo doesn't offer all that SVG or PDF offer, but that is mostly due to their declarative nature. They have to provide everything, while with Cairo, the application has ample room to build richer systems. It is for instance possible to let Cairo render into a local buffer, tweak the pixels directly, then use the tweaked buffer as source for further rendering operations.

Cairo already has backends for X, OpenGL, PostScript, and local image buffers. The OpenGL backend has seen a lot of work recently and shows that Cairo can be efficiently accelerated on modern graphics hardware. The PostScript backend is still very basic, it just writes a huge bitmap to the PostScript file, which is of course not how you want to render a document. A real PostScript backend needs to be a very sophisticated in figuring out what parts need to be send as bitmaps, and what can be rendered using PostScript operators. This is necessary, since the Cairo (~PDF) rendering model is much richer than the PostScript model. While this is a programming challenge, it has been tackled in such software as OpenOffice and ghostscript, so it is definitively doable. A simple way out would be to write a pdf file and let ghostscript deal with the conversion to PostScript.

The traditional GDK rendering API, which wraps the Xlib drawing functions will not go away; Cairo will be an additional rendering API. Most likely the Cairo support in GDK will not wrap the complete Cairo API, it may consist of a single function which connects GdkDrawable and a Cairo surface and redirects drawing from the surface to the drawable. The reason for wrapping the entire Xlib drawing api was that the Xlib functions require explicit Display and Window parameters all the time, which makes them cumbersome to use. Another reason was that GDK allows other backend implementations beside the Xlib one, e.g. the Windows backend. Cairo on the other hand, already offers an API very similar to what the GDK wrappers would look like anyway, with a single cairo_t context parameter. And Cairo already supports multiple backends, so wrapping it again in GDK would create multiple levels of wrapping.

A big advantage of not wrapping the Cairo api further is that we don't have to maintain the wrapper layer as Cairo develops further. We don't have to maintain separate documentation. A small disadvantage of not wrapping Cairo is that the naming conventions don't match exactly (cairo_font_t instead of CairoFont, cairo_font_reference() instead of cairo_font_ref()) - application programmers will learn to live with it. The fact that Cairo doesn't use GObject causes a bit more work for language bindings. But we are convinced that the benefits outweigh the disadvantages.

While Cairo allows to do rendering with an alpha channel, the COMPOSITE X extension allows windows to have an alpha channel. While the usefulness of this is often overrated, it allows for neat 'glitz' effects like fading in and out of menus, or proper drop-shadows. GDK will support this by offering API to find visuals and colormaps with an alpha channel.

Theming is a difficult issue because there is an inherent tnesion. On the one hand, we want to have themes that can control precisely how GTK+ renders, and we want to be able to extend GTK+ with new widget types. The considerations argue for a theming system that is tightly integrated with the way GTK+ works. On the other hand, we want to write themes that chain to a platform native look: the WIMP theme has done this very effectively for Windows. And we want to be able to use the theme system to render third-party widgets, as is done by e.g. OpenOffice or Mozilla. These considerations argue for a theming system that is much more closely tied to the idea of a "standard set" of widgets. Finally, it has to be possible for libraries and applications to create new widgets and have them work with themes without the theme having explicit knowledge about the new widgets.

The current theme system in GTK+ was developed with knowledge of the above issues, and the attempt was to create a maximally flexible system. The way it works is that a theme provides replacements for the standard paint functions corresponding to different basic widget components: flat and beveled boxes, checkbutton indicators, arrows,notebook tabs, etc. These functions are called to draw the actual widgets, and they receive the drawable and area to draw in, but also some extra information: an unspecified detail string, and a reference to the widget itself. The idea is that by providing basic implementations of the functions, a theme can minimally render any widget, but it can also use the detail string or even the widget pointer to special case and improve the rendering for particular widgets. The theme engine and generic and theme specific options are bound to particular widgets using the gtkrc file which is written in a custom language whose syntax vaguely resembles C.

The theme system currently used in GTK+ has been successful in the sense that people have generally been able to get widgets to appear as they want, but the system has a number of big shortcomings.

- The style functions take a `detail` parameter, which is a freeform string. The set of used detail strings is unspecified, and style functions have to do a considerable amount of special-casing based on the detail strings in order to render all widgets reasonably.
- There is no concept of layout in the theme system. All layout decisions are done in the widgets. This means that it is not possible to write a theme which rearranges

the controls in the color selection dialog, unless the color selection dialog defines a style property for this purpose.

- The style functions are very hard to use for rendering non-GTK+ widgets, yet this is desirable to match the appearance of GTK+ in applications written in other toolkits, e.g. OpenOffice or Mozilla.

A new theme system should address these issues.

- It should be fully specified to avoid the detail string mess.
- It should provide access to layout of composite widgets.
- It should emphasize declarative ways to write themes, and the declarative parts of a theme should use a standard syntax like XML or even CSS. The unavoidable imperative parts should not depend on GTK+ internals, in order to make them useable for non-GTK+ widgets. Using Cairo as the rendering API for themes would go a long way towards achieving this.

The missing print dialog is one of the more glaring holes in the widget set. libgnomeprint and libgnomeprintui offer a print dialog, but the natural place for it to live is GTK+. Having Cairo support in GDK will make this easier, since Cairo already has PS and PDF backends. The print dialog will be implemented similarly to how the new file chooser works, with backends for the various printing systems: CUPS and maybe lpr for legacy Unix systems, GDI on Windows.

Introspection (also known as "Reflection", or "typelibs") means that GTK+ offers information about its interfaces in a form which can be programmatically used. GObject currently offers information about the type hierarchy and implemented interfaces, and about signals and properties of objects. This information is successfully used for creating automatic language bindings, it is used by gtk-doc to automatically generate a good amount of the GTK+ reference documentation. It could also be used to help with code generation and code completion in IDEs, although I don't know if there are any such projects currently using the GObject introspection.

To make the introspection mechanism of GObject even more useful, the virtual function slots in class structures and the ordinary library functions should be made available. This should allow fully automated language bindings, and would allow IDEs to assist in deriving new classes from existing ones, by knowing which virtual functions can be overridden, and what their signatures are.

Many other interesting, "blue-sky" ideas for future GTK+ development can be found on `http://www.gtk.org/plan/2.6`

Conclusion

GTK+ will continue to evolve in interesting ways, there is a lot to explore and try in the area of user interface design. GTK+ would not be where it is today if it wouldn't have been backed by such a superb team of developers.