The future of rendering in GNOME

Owen Taylor `<otaylor@redhat.com>`

GUADEC 5

Kristiansand, Norway

June 28-30, 2004

Introduction

GNOME currently has a diverse set of rendering technologies, and of programming interfaces to use those rendering technologies. The oldest rendering technology is the core X drawing commands. These provide a very traditional 2D rendering API, including basic graphics primitives: lines, rectangles, arcs, and so forth, all without antialiasing or alpha compositing. The RENDER extension to X provides a more modern 2D set of graphics operations; in GNOME usage of RENDER is confined mostly to alpha-compositing text and graphics. On the client side, the libart library provides routines for drawing antialiased shapes, and in a more specialized domain, font rasterization is done by the FreeType library.

GNOME Applications seldom use the above technologies directly, but instead use a number of programming interfaces built on top of them. The GDK drawing interfaces used by the GTK+ toolkit generally map directly onto the Xlib commands and have similar limitations. However, for GTK+-2.0 this was slightly extended by adding support for anti-aliased text and for images with an alpha-channel. Where the RENDER extension is available, it is used to implement these capabilities. Where it isn't available, they are implemented by grabbing the destination drawable into local memory, compositing against it, and writing the result back to the X server.

The GNOME Canvas provides a retained-mode interface for drawing in GNOME. (*retained-mode* means that the application builds up a tree of graphics objects that the system redraws as necessary. Most of the other interfaces discussed here are *immediate-mode*; the application reexecutes all the drawing commands each time drawing needs to be done.) The GNOME canvas offers two drawing modes; one is non-anti-aliased and implemented using GDK. The second mode uses libart to implement anti-aliased drawing.

The gnome-print library provides yet another set of drawing interfaces for GNOME. The interfaces in gnome-print are postscript-like, but with the addition of an alpha-channel.

So, what issues need to be addressed in GNOME, moving forward? One should be apparent from the above description. The diversity of different rendering interfaces is confusing for the application programmer, and also causes problems for the implementation. Each set of rendering technologies has its own separate set of bugs and performance issues. If we can move to a unified rendering interface for screen display and printing, we solve these problems.

We might also want to provide rendering capabilities that go beyond what is offered by gnome-print. gnome-print, while it has alpha-transparency and antialiasing, is still a quite simple graphics API. It doesn't have gradients. It can't combine objects in any way other than the simplest alpha-compositing.

Cairo

Recently, a graphics library has been under development that is quite closely suited to the future rendering needs of GTK+ and GNOME. The Cairo library [Cairo] is designed to be an easy to use 2D graphics library offering a rich set of capabilities and multiple output backends.

As is the case for gnome-print, the Cairo API is closely modeled on the way that postscript works. Given a Cairo drawing context, `cairo_t *cr`, drawing a red triangle looks like:

```
cairo_set_rgb_color (cr, 1.0, 0.0, 0.0);
cairo_move_to (cr, 50.,  0.);
cairo_line_to (cr, 100., 87.);
cairo_line_to (cr, 0.,   87.);
cairo_close_path (cr);
cairo_fill (cr);
```

There are also functions to create common types of paths like rectangles and arcs and circles. But Cairo also goes beyond the simple postscript-like model in various. It makes it easy to create temporary surfaces, draw to them, then combine them back with the main surface with any of a number of different compositing modes. This gives capabilities similar to the concept of groups in PDF-1.4. It supports linear and radial gradient patterns. (Postscript Level 3 has very complicated gradient support, but this isn't found in gnome-print.)

In rough terms, the rendering capabilities of Cairo are similar to those of Java 2D, SVG, or PDF-1.4. In detail Cairo has a smaller set of capabilities than either SVG or PDF-1.4, but this makes sense; SVG and PDF-1.4 are purely declarative, so they need to cover pretty much anything an application might want to do. Cairo on the other hand, provides more opportunity for the application to build richer systems on top if needed. For screen rendering, we can even do things like render to a local buffer, perform direct operations on the pixels, then use the tweaked buffer as a source for further operations, something that would be useful, for instance, when implementing some of the SVG filter modes. An fairly implementation of SVG on top of Cairo already exists [SvgCairo].

One the backend side, Cairo currently supports a number of output targets. It has a backend that targets RENDER for X display, another that renders to a local image buffer. There's a backend that targets OpenGL that has gotten a lot of work recently, and shows that Cairo can be efficiently accelerated on modern graphics hardware. Finally there is very basic postscript backend. The backend as it exists now just renders a huge bitmap and writes that into a postscript file, which is, of course, not what you want for rendering documents. A real postscript backend needs to be quite sophisticated in figuring out what parts of the output can be represented as postscript, and what parts of the document need to be sent as bitmaps, since the Cairo rendering model goes considerably beyond what Postscript can do. While this is definitely a programming challenge, similar problems have been tackled before in such software as OpenOffice and Ghostscript, so it should definitely be doable. Another possibility is to simply write out PDF-1.4 files and let Ghostscript do the conversion to postscript; however it would be nice if a usable PS backend didn't depend on the presence of Ghostscript on the system.

GTK+ integration

GDK wraps Xlib entirely. `XDrawLine()` has the corresponding `gdk_draw_line()`, and so forth. The natural question is then whether we should do the same for Cairo. If we look at the reasoning behind the wrapping in GDK, we see that doing the same thing for Cairo doesn't make sense. By wrapping Xlib, we gain in convenience: the Xlib APIs are cumbersome to use for a number of reasons, the most obvious being that all the functions take a separate Display parameter. drawable surfaces objects rather than just numeric IDs. Unlike Xlib, programmer convenience has been one of the most important considerations when creating the Cairo APIs. That parameter was eliminated in GDK by making. Also importantly, GDK is actually a portable wrapper that can run on top of multiple rendering systems. In addition to X, GDK also runs on top of the Win32 GDI, the directfb windowing system, and several other drawing APIs. But Cairo already supports multiple backends adding another layer on top of that would add no additional benefits.

With the above considerations not being a factor, we can gain a lot by presenting the Cairo APIs directly to the application programmer. We don't need to maintain the wrapper layer; if additions are made to the Cairo APIs, they are directly available to the GNOME programmer. We don't need to maintain a separate set of documentation for our drawing library. If we have drawing libraries that we share with other non-GNOME-specific applications (for themes, for SVG rendering, or whatever), we an can simply pass the Cairo objects to them directly. There are some disadvantages to not wrapping Cairo; Cairo uses different naming conventions for types and functions then the GNOME libraries. (cairo_font_t rather than CairoFont, `cairo_font_reference()` rather than `cairo_font_ref()`). Also, because Cairo doesn't use GObject, wrapping Cairo in a language binding requires much more custom glue code than for a GNOME library. But these disadvantages don't outweigh the strong advantages of presenting Cairo directly to the application programmer.

Since we aren't wrapping individual cairo functions, the amount of API that we need to add to GDK is actually quite limited. The only function that is actually required is:

```
void gdk_drawable_update_cairo(GdkDrawable *drawable, cairo_t *cr);
```

This function redirects drawing for the Cairo surface to the given drawable. The implementation needs to take care of handling some of the internal complexities of GDK like double buffering and 32-bit coordinate emulation, but this is hidden from the user. So, an expose handler that uses Cairo is quite simple.

```
void
my_widget_expose (GtkWidget      *widget,
                  GdkEventExpose *event)
{
  cairo_t *cr = cairo_create ();
  gdk_drawable_update_cairo (event->window, cr);

  cairo_set_rgb_color (cr, 1.0, 1.0, 0);
  cairo_rectangle (widget->allocation.x, widget->allocation.y,
                   widget->allocation.width, widget->allocation.height);
  cairo_fill (cr);

  cairo_destroy (cr);
}
```

But since virtually every expose handler performs these same operations, it makes sense to provide a default expose handler that handles the creation of the Cairo context and calls a `paint` handler with that additional argument.

```
void
my_widget_paint (GtkWidget      *widget,
                 GdkEventExpose *event,
                 cairo_t         cr)
{
  cairo_set_rgb_color (cr, 1.0, 1.0, 0);
  cairo_rectangle (widget->allocation.x, widget->allocation.y,
                   widget->allocation.width, widget->allocation.height);
  cairo_fill (cr);
}
```

That's really all there is with rendering integration with GDK and GTK+. But to the ability to do rendering with an alpha-channel, we'd like to add another capability: having windows that actually have an alpha channel for a background. This is implementable with the DAMAGE and COMPOSITE extensions to X now under development, and the GDK interface is quite trivial:

```
GdkVisual *gdk_screen_get_rgba_visual(GdkScreen *screen);
GdkColormap *gdk_screen_get_rgba_colormap(GdkScreen *screen);
```

These functions gets the best available visual and colormap with an alpha channel, similar to the existing `gdk_screen_get_rgb_visual()` and `gdk_screen_get_rgb_colormap()`.

There's one further rendering trick that the COMPOSITE extension would allow us to play. Currently some GTK+ widgets have their own X windows, other GTK+ widgets draw directly into their parent's window. This is a useful compromise because not having a separate window is slightly more efficient and allows better rendering (in particular proper blending with the parent widget), while having a separate window allows taking advantage of X's facilities for scrolling and clipping. However, the mixture causes problems for controlling Z order: widgets with a window will inevitably draw above widgets without a widget. The COMPOSITE extension allows for fixing this; we could add a function:

```
void gdk_window_set_parent_draw(gboolean parent_draw);
```

When this flag is turned on, changes to a child window do not have any immediate affect on the screen. Instead, they just cause the corresponding areas to be added to the parent's invalid region. The application or widget is responsible for drawing the child windows onto the parent widget, and can properly interleave child widgets with and without child windows into the proper Z order. The question here is whether this facility is useful without being available everywhere; it's not possible to implement a fallback implementation that works on older X servers. [1]

Text

One thing that was glossed over in the preceding section is how text drawing works. This is an area where what GTK+ applications use will be significantly different from the raw API, because GTK+ is based on Pango, which provides a much richer than the simple text API that Cairo provides itself. The simplest use of Pango in a Cairo program will look like:

```
PangoLayout *layout = pango_cairo_create_layout (cr);
pango_layout_set_text (layout, "Hello world");
pango_cairo_show_layout (cr);
g_object_unref (layout);
```

The `pango_cairo_create_layout()` here is a convenience function that looks like:

```
PangoLayout *
pango_cairo_create_layout (cairo_t *cr)
{
  PangoFontMap *font_map = pango_cairo_get_default_font_map ();
  PangoContext *context = pango_cairo_font_map_create_context (font_map);
  PangoLayout *layout = pango_layout_new (context);

  pango_cairo_context_update (context, cr);
  g_object_unref (context);

  return layout;
}
```

This may look a little different than the initial expectations. You might expect a PangoContext to be created for a particular cairo_t. However, this doesn't work because Cairo context is a transient object that we create just when we are rendering, but it's useful to keep PangoLayout objects around in many cases, since layout is an expensive operation. And each PangoLayout contains a persistant pointer to a PangoContext. So, the PangoContext doesn't contain a pointer to the cairo_t; rather it just copies the information about the current transformation matrix and destination surface out of the cairo_t that is needed for doing layout.

It should be emphasized that all the dimensions and positions associated with a PangoLayout object are in user coordinates, not device coordinates. The advantage of this is considerably simplicity. We can say that lines of text always run in the X direction, and that paragraphs always lay out as an axis-aligned rectangle. In fact, even if we have Chinese writing where the lines of text run top-to-bottom on the page, the text still runs in the X direction, we just require the application to use a 90 degree rotation. But then you might wonder why the layout depends on the current transformation matrix at all. Shouldn't the positions just scale and transform exactly with the size of the font? In some cases, using layout that is independent of the current transformation is useful; this is what we want if the final output is a high resolution printer. But if we are optimizing text for display on the screen, we can produce significantly better looking output by positioning using the metrics for a particular pixel size [Taylor1].

Themes

[1]It might seem you could implement the child windows as pixmaps, but this doesn't work because we expose the fact that each GdkWindow corresponds to an X window

Now that we know how widgets interact with Cairo to render themselves, we then need to look at what gets drawn by these widgets. Theming is difficult issue because there is an inherent tension. On one hand, we want to be able have themes that can control precisely how GTK+ renders, and we want to be able to extend GTK+ with new and novel types of widgets. These considerations argue for a theming system that is very tightly tied to the way that GTK+ works. On the other hand, we want to be able to write GTK+ themes that chain to a platforms native look; the GTK-WIMP [GTK-WIMP] project has done this very effectively for Windows. And we want to be able to use the GTK+ theme system to render other widget sets; this is currently being done by OpenOffice and Mozilla. Those considerations militate for a theming system that is much more closely tied to an idea of a “standard set” of widgets. In addition there is the issue of third party widgets. It has to be possible to for libraries and applications to create new types of widgets and have them integrate into the theming system and in fact work with themes that were created without any knowledge of these new widget types.

Many of these considerations were in fact known when the current GTK+ theming system was created in 1998, and the attempt was to create a maximally flexible system. The way that the GTK+ theming system works is that there is a set of paint functions corresponding to different basic widget system components: flat and beveled boxes, checkbutton indicators, arrows, notebook tabs, and so forth. A theme engine provides implementations of each of these functions, and when the function is called receives not only the destination drawable and information about where to draw the component, but also extra information: a detail string, which is an unspecified string giving extra information about the particular usage of the component and a pointer to the widget itself. The idea is that by providing basic implementations of the component functions, the theme engine can minimally render any widget, but it can also use the detail string and even the widget pointer to special case and provide improved rendering for particular widgets. The theme engine along with generic and theme-engine specific options are bound to particular widgets using the gtkrc file language [Taylor2].

While the current theming system has been successful in the sense that people have generally managed to work with it and get the appearance that they desire, many deficiencies have been found. The set of detail strings used is unspecified and in practice themes engines *do* need to special-case a consider number of details to render GTK+ widgets correctly, so creating a theme engine is an exercise in cut-and-paste and trial-and-error. Typically theme engines also end up referencing the widget pointers that are passed in; while these pointers are in theory allowed to be null and theme engines are supposed to check this, in practice an attempt to pass in null pointers when rendering controls that aren't GTK+ widgets will crash most theme engines. The tight binding of theme engine rendering to particular types of widgets also causes problems when creating custom widget types. It's very difficult to create a custom widget that appears like a `GtkEntry` but isn't actually one. Even the simplest case of deriving a new widget from an existing class can break themes. And finally, there is no conception of layout in the theme system; the way that the components of a widget are fit together to create the widget has to be figured out by studying the GTK+ sources. This causes particular problems for people using the GTK+ system to render non-GTK+ widgets, because they have to duplicate considerable amounts of layout code from GTK+.

While designing a new theme system for GTK+ is beyond the scope of this paper, we can lay out some general principles for how it should work:

- It should be implemented without reference to widget specifics so that it can be used by anybody who needs to render a control that matches the GTK+ look. In fact, it may be desirable to implement it with minimal dependencies on the GTK+ stack to further widen the set of possible consumers. Making Cairo the rendering interface would bypass a lot of potential problems with finding a common ground for rendering.

- It should be multi-layered. While the idea in the current GTK+ theme system of standard components that can be recycled to create custom widgets is likely useful, it's not sufficient. If we add an extra layer on top of that that has an idea of complete controls and of layout, then we make the theme system both more flexible for theme creators and more easily usable for theme consumers.

- As much as possible should be declarative; config files should be used instead of code. However the ability to chain to native code needs to be preserved to do things like a native theme on Windows.

- Careful specification is essential. If there is only one provider: one theme engine, or only one consumer: one set of controls, then implicit specification by implementation may work, but we are in a case where we have multiple providers and multiple providers and multiple consumers, so a formal specification is crucial.

- For declarative parts, it should use standard file formats such as XML and possibly CSS instead of inventing custom syntax.

It should be possible to compatibly introduce the theme system sketched above into GTK+ by implementing it as a theme engine; parts of GTK+ and 3rd party widgets could then be gradually transitioned over to using the new system directly.

Printing

The existence of PS and PDF backends for Cairo clearly goes a long ways toward providing consistent interfaces for rendering to screen and printing, but it's not a complete solution for application printing. We need a way to put a dialog for the user to select a printer and choose options for the printer. The application needs to be able to get basic information about the chosen printer; information like paper size and whether it's a color device or monochrome device. And finally the application needs to be able to create a Cairo context that spools output to this printer taking the selected options into account. This type of functionality is currently provided by libgnomeprint and libgnomeprintui.

The natural place for this functionality to live is in GTK+. This is functionality that virtually all applications need. It's functionality that needs to be done significantly different depending on the platform; on Windows, for example, we want to see the printers configured on the system, to print to them with a GDI backend for Cairo, and possibly to use the native print dialogs. For Linux, we'd want to have tight integration with CUPS. We might also have a lpr-based backend for legacy Unix systems. And finally its not a huge amount of code once we have the Cairo backend. libgnomeprintui is only about 15,000 lines of code.

The API here should be straightforward. We'd have a print dialog that would would work similarly to `GtkFileChooser`; the application could then retrieve a `GtkPrintContext` object reflecting the printer and options that the user selected in the dialog. The application could retrieve information about the printer from the `GtkPrintContext` and create a cairo_t that renders to the printer.

Conclusion

We've seen that currently rendering in GNOME is done with a ad-hoc collection of different interfaces and technologies. Cairo offers an appealing way to both unify on a single rendering interface, and to improve the rendering capabilities provided to GNOME applications. Moving to Cairo provides us the opportunity to revisit such areas as printing and themes, solve some of the long-outstanding issues, and make sure that these capabilities are provided at the right level in the platform stack.

References

[Cairo] *Cairo vector graphics library [http://cairographics.org/].*

[GTK-WIMP] *GTK-WIMP [http://gtk-wimp.sourceforge.net/].*

[SvgCairo] *libsvg-cairo [http://cairographics.org/libsvg-cairo].*

[Taylor1] Taylor, Owen. *Rendering good looking text with resolution independent layout [http://people.redhat.com/otaylor/grid-fitting/].*

[Taylor2] Taylor, Owen. *The GTK+ Theme Architecture, version 2 [http://www.gtk.org/~otaylor/gtk/2.0/theme-engines.html].*