Pango, an open-source Unicode text layout engine

Owen Taylor

Red Hat, Inc.

otaylor@redhat.com, http://people.redhat.com/otaylor/

Abstract

Open source development methologies have been used successfully for many different types of software, and are just as applicable to the task of rendering internationalized text. In particular, in the area of text rendering, an open-source approach provides the opportunity for local organizations and governments to actively contribute to development of support for scripts with smaller user communities, rather than simply waiting for for the software vendor to provide such support. The Pango project provides a highly modular framework for internationalized text layout and rendering, with the ability to incrementally add support for both new scripts for new new font technologies and rendering systems.

Introduction

Open source is no longer an unfamiliar concept these days; around a third of new server deployments run some form the open-source Linux operating system and almost 70% of deployed web servers are running the open-source Apache web server. Open-source libraries in common use include libpng, libjpeg, the libxml XML processing libraries, and IBM's ICU Unicode library [http://oss.software.ibm.com/icu].

Open source software is generally of high quality. The development model of open source software includes a large group of contributors submitting improvements that are incorporated into the code base by a small group of core developers in an open process that, by its very nature encourages code review. Open source software is responsive to the needs of its users: not every user of a piece of open source software is going to be interested in contributing code or have the necessary skill, but for a widely used piece of software, there will be a good number of users who are interested in adapting the software to their particular needs. In addition, users have confidence that the software vendor is

not trying to lock them into proprietary protocols, perhaps later to raise the price. The vendor instead attracts and keeps customers by supporting, maintaining, and enhancing the software better than its competitors.

In the area of a layout engine, perhaps the most interesting aspect of the open source process is this ability for users with particular needs to contribute back improvements in those areas. The vast majority of people in the world use only a few scripts with relatively simple rendering needs. However, even scripts with a few fractions of a percent of the worlds users still have hundreds of thousands or millions of users; some of these will be programmers with an interest in enabling the software they are using for their script.

The idea of the Pango project was to harness the advantages of open source in the area of text rendering by creating a highly modular framework with the ability to incrementally add support for new scripts and new rendering backends. It is meant as a complete solution for all scripts, rather than being restricted to use for unusual scripts, and offers strong capabilities for the layout of Western languages. However, the compelling feature is certainly the support for complex scripts. To facilitate such contributions, Pango is designed to be modular. By writing a small dynamically loaded plugin for Pango, a developer enables support for a new script in Pango, and thus throughout the system. Configuration tools, dialog boxes, word processors, spreadsheets, and so forth all gain support for the new script.

The Pango project was started in 1999. A 1.0 release was made in 2001, and used in version 2 of GTK+ user interface toolkit [http://www.gtk.org]. The follow-on release, version 1.2, added support for, among other things, OpenType Indic fonts, Uniscribe, and for the fontconfig font catalog library. Changes in version 1.4, which is being finalized as of January 2004 include support for characters beyond the BMP and for Unicode-4.0, better handling of bidirectional editing, and support for the GPOS table for the Arabic script, which enables the Nashtaliq writing styles used for Persian and especially Urdu. Pango is the basic text rendering library of the GNOME desktop which is used widely used throughout Linux and Unix, including Red Hat Linux 8, 9 and Red Hat Enterprise Linux 3, and the Sun Java desktop.

The application view

The basic job of Pango is to take Unicode text, possibly annotated with extra attributes and convert that into appropriately positioned glyphs selected from the fonts on the system. In some cases, the user uses routines from Pango directly to render the positioned glyphs; in other cases Pango is used as part of a larger system which uses the positioned glyphs; for example, by writing them into a Postscript file.

From the point of view of the application programmer, Pango looks quite simple. There is a layout object, PangoLayout that holds one or more paragraphs of Unicode text. Once a PangoLayout is created, the application can determine the size of the text, or render it to an output device:

```
PangoContext *context;
PangoLayout *layout;
int width, height;

context = pango_xft_get_context (display, screen);
layout = pango_layout_new (context);
pango_layout_set_text (layout, "Hello, world");

pango_layout_get_pixel_size (layout, &width, &height);
pango_xft_layout_render (xft_draw, xft_color, layout, 10, 10);
```

It's also possible to set attributes on the text programmatically or using a simple markup language:

```
pango_layout_set_markup (layout,
                         "<span size="x-large">Big</span> text.");
```

For applications that need to manipulate the text, rather than simply displaying entire paragraphs, APIs are provided for hit testing, determining the screen extents of ranges of text, determining cursor layout, for iterating through the text in visual or logical order, and so forth. These APIs allow building applications to edit arbitrary Unicode text while being shielded from many of the complexities of the individual scripts.

The rendering pipeline

Beneath the high-level API shown above that deals with paragraphs of text, there is a low level API that exposes the details of Pango's layout pipeline. The steps in this pipeline are:

- Itemization: Input text is broken into a series of segments with unique font, shape engine, language engine, and set of extra attributes.
- Text boundary determination: The text is analyzed to determine logical attributes such as possible line break positions, and word and sentence boundaries.
- Shaping: each item of text is passed to the shape engine for conversion into glyphs.
- Line breaking: the shaped runs are grouped into lines; if the individual runs are longer than particular lines, then they are broken at line break positions,

and the components of the run reshaped.

- Justification: white space (or Kashida glyphs) are inserted into the runs to justify the text to the margins, when desired. (A full implementation of this does not yet exist.)

Programmers familiar with Microsoft's Uniscribe system [http://www.microsoft.com/typography/developers/uniscribe] may find the above somewhat familiar. The low level parts of Pango were fairly strongly inspired by descriptions of that system, though the the details are different. This similarity has proved quite useful when running Pango *on top of* Uniscribe, as is done on Windows. On the other hand, the higher level parts of Pango borrow more from such systems as the Java text APIs.

The low-level pipeline can be used directly by toolkits and applications, but in actual practice this occurs very seldom. All the use of Pango within the GTK+ user interface toolkit is done using the high-level PangoLayout API.

The itemization step is particularly interesting in detail, because it controls the remaining steps in the process; the choice of font and shape engine determines how the transformation from text into glyphs happens. If the user has explicitely picked the font, then presumably it will have all the characters in the text, but quite commonly the font is specified as a generic alias such as Sans-serif, and Pango needs to resolve that alias into particular fonts on the system that cover the characters in the text. In the simplest form, one might just look for the first font on the system that covers each character, but this can easily lead to “ransom-note” effects where the text is a jumble of characters from different fonts.

To prevent ransom-note effects, Pango takes a multi-step approach to assigning fonts to the characters of the text. First, the text is broken up into runs by script using an algorithm from ICU. Neutral characters inherit their script from adjoining characters and paired punctuation such as parentheses is taken into account. Then language tags are assigned to the text. These language tags may be specified by the application; a web brower, for instance, would set language tags based on the content of the HTML document. If no explicit language tags are specified then default values are used based on the environment of the user. However, attention is paid to the previously assigned scripts. If we have a segment of text for which the language tag from the application or environment would be 'zh-cn' but the text is clearly in the Arabic script, then a language tag of 'ar' is used instead. This isn't perfect, since the text might, say, be in Persian, but it often results in reasonable tagging by language without any user intervention.

Once we have the text tagged by language, we then use this information in

order to do a better job in selecting fonts. The way this is done is sorting the available fonts so that fonts that cover all the characters typically used for the particular language are looked at first. For a Greek, a font that actually is meant for the display of Greek text will be preferred to a font that happens to have a Pi character included as a mathematical symbol, even if the font with the Pi is listed first in the alias specification.

Architecture

As we saw in the above example, the user uses Pango directly in some cases, when doing operations that are independent of the target rendering system, but in other cases needs to use functions that are specific to a particular target rendering system. After all, the only way that Pango could completely cover up the differences between rendering systems was if it was an entire rendering toolkit rather than a system for text layout.

So, Pango does not completely shield the application or toolkit from the rendering system, but rather sits between the two and provides both generic API and API specific to a rendering system. We can the combine Pango with a rendering toolkit to provide a complete system that provides the entire necessary abstraction for the user. If we rewrite the above example as a user of the GTK+ toolkit user would use it, we have:

```
PangoContext *context;
PangoLayout *layout;
int width, height;

context = gtk_widget_get_pango_context (widget);
layout = pango_layout_new (context);
pango_layout_set_text (layout, "Hello, world");

pango_layout_get_pixel_size (layout, &width, &height);
gdk_draw_layout (widget->window, widget->style->black_gc,
                 10, 10, layout);
```

This will then work both on Unix with FreeType, Xft, and the X RENDER extension, and on Windows using Uniscribe and the Win32 API.

As well as providing portability between operating systems, the abstraction found in Pango is useful for preserving compatibility on a single operating

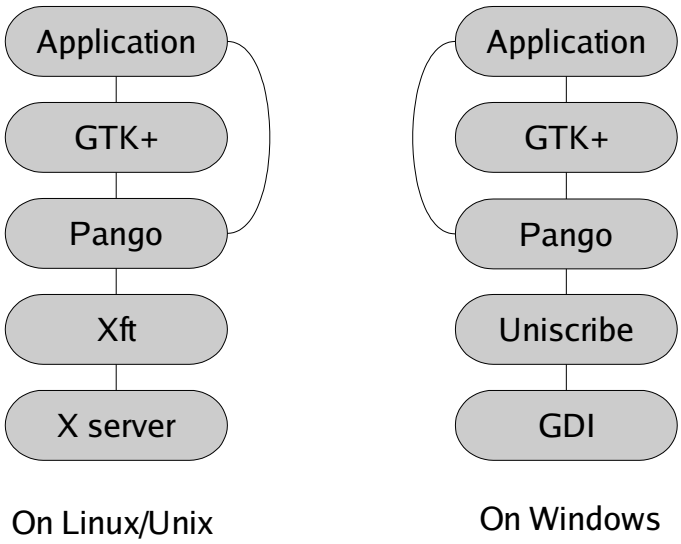

Figure 1. Comparison of Pango architecture on Windows and Linux/Unix.

system. Between Pango-1.0 and Pango-1.2 releases, the default rendering backend on X was changed from one based on an old system of bitmap fonts on the server, to a new client-side framework for scalable fonts. This all happened without visible change to applications: existing applications picked up the enhanced font rendering without any modifications.

If we peek inside the oval labeled Pango in Figure 1, we see four basic components, as shown in Figure 2. First there is the core part of Pango; this provides the main public API, high level layout objects such as PangoLayout, and the driver logic for the rendering pipeline. There are language modules; modules that provide linguistic processing for a particular script or language, such as determining word breaks and hyphenation points. There is the backend library, which provides public API specific to a particular rendering API, such as the `pango_xft_render_layout()` call we saw above. Finally, there are shape modules. These are modules that are responsible for converting Unicode text to glyphs for one script and for a particular font technology. Shape modules may be specific to one particular backend library, or they may be shared by a group of related backend libraries.

In the itemization step, the shape module is chosen along with the font. To shape the text, the shape module is passed a segment of Unicode text, the font, and any custom attributes specified by the application. The shaper then uses knowledge about the particular script, and information about the font to

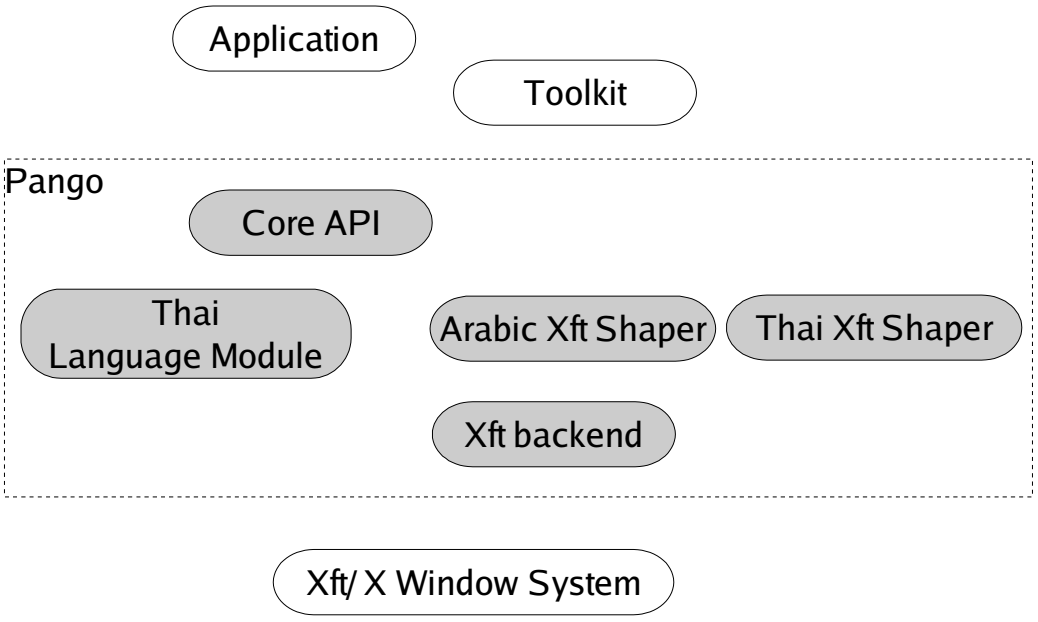

Figure 2. The Pango Architecture

produce a series of glyph indices chosen from the font along with positioning information. These glyphs don't have to be in a one-to-one correspondence with the input characters; instead the concept of *clusters* is employed. A cluster is basically the smallest unit at which we can do a mapping between input characters and output glyphs. Three input characters might map into two output glyphs. Examples of clusters include a f-i ligature in Western text or a syllable in an Indic language.

There are quite a few different possibilities for how a shaper might be implemented; many of the shapers shipped with Pango work more-or-less directly with the OpenType tables in the font to do shaping for a single script or set of related scripts. On the other hand, a single shaper is used for all scripts when running on Windows. This shaper simply calls out to Uniscribe so that rendering is consistent with other Windows applications.

Underlying technology

Pango builds upon many other open source technologies, both by using existing libraries as dependencies, that is, packages that are required to be installed on the system in order to run Pango, and by sharing code with other similar projects.

- The GLib library [http://www.gtk.org] provides a rich set of data structures, portability routines, convenience functions, and, importantly for Pango, set

of Unicode manipulation algorithms and Unicode property tables.

- The GObject library, which is distributed with Glib provides a standard framework for object oriented programming in C, with such features as inheritance, interfaces, introspection, and properties. By building on top of GObject, creating bindings for different programming languages for Pango becomes much easier.

On Unix with the X window system, GTK+ depends on a number of additional libraries to provide for font location and rendering:

- The fribidi library [http://fribidi.sourceforge.net] provides an implementation of the Unicode bidirectional algorithm. For convenience, because the amount of code is not large, Pango incorporates a the code internally; however, it is developed and maintained as a separate project.
- The FreeType library [http://www.freetype.org] provides a set of font loading and rendering routines for a large number of different font formats. Pango also includes code derived from the FreeType project to parse the OpenType tables used for complex script layout. This version of the code in Pango is shared with the support for Indic layout in version 3.2 of the Qt toolkit.
- The fontconfig library [http://freedesktop.org/software/fontconfig] provides a standardized method of locating fonts and matching names to fonts.
- The Xft library provides the glue code for taking fonts as rendered by FreeType and rendering them to the X display.
- The code used to render Indic OpenType fonts is a conversion of code first written for the layout portion of the ICU library.

On other platforms, different underlying rendering technologies are used. As mentioned above, on Windows, Pango builds on top of the Uniscribe system.

Current Status

Scripts supported in the current version of Pango include, among others, all the scripts of Europe, Han characters, Arabic, Hebrew, Thai and many Indic scripts. Some of the script support was contributed by native users of the scripts in questions, others was initially written by core Pango developers, but even where the original authors were not native script users, we've subsequently seen many bug fixes and enhancements from native users.

The main user of Pango is still the GTK+ toolkit, though it has also been used in other contexts, such as rendering XSL stylesheets [http://pangopdf.sourceforge.net], and doing text rendering within the Mozilla web browser [http://bugzilla.mozilla.org/show_bug.cgi?id=215219].

The use of Pango within GTK+ provides support for all the above scripts to a wide range of open-source and proprietary applications. In particular, the GNOME desktop has comprehensive support for these scripts, since it builds upon the GTK+ toolkit. Pango is also actively in use on Windows via the Windows port of the GTK+ toolkit, which is used by such cross-platform applications as the GNU Image Manipulation Toolkit. (GIMP.)

Future Directions

One area of future work for Pango is further extending the set of languages that Pango handles on Linux and Unix. There are some fonts with specifications for OpenType, such as Tibetan and Khmer, which Pango does not support yet, and adding shapers for these languages is definitely planned. (In fact, a Khmer shaper has just been announced on the Pango mailing list at the time of writing, though it is not yet integrated into the full Pango code base.)

Beyond the scripts handled by OpenType, work is in progress to integrate Pango with the SIL Graphite [http://graphite.sil.org] project, so that applications that use Pango can automatically have access to any Graphite fonts found on the system. Graphite has excellent coverage for scripts with less users, so this considerably increases the range of scripts and languages that Pango can handle.

The other area where work is being done on Pango is in the area of improving its typographic quality. For western languages, hyphenation, justification, and contextual variants are all possible within the Pango framework. For East Asian scripts vertical writing is an important feature in many applications.

Conclusion

By providing a modular framework for script support, Pango provides a process for native script users to easily add support for their own scripts to contexts such as the GNOME desktop. In addition, the independence of Pango from the underlying rendering technology allows applications to code to a straightforward API and gain portability across different platforms and compatibility with future enhancements to particular platforms. Future enhancements to Pango will provide even wider script support and higher quality output for currently supported scripts.