Pango: internationalized text handling

Owen Taylor
Red Hat, Inc.
otaylor@redhat.com, http://people.redhat.com/otaylor

Abstract

Pango is a library for laying out and displaying internationalized text. It handles almost every writing system in the world, and can work on top of multiple different display systems - including traditional X fonts, or client-side OpenType fonts. Pango is used for all text handling in the soon-to-be-released version 2.0 of the commonly used GTK+ widget toolkit.

1 Introduction

Although technically inclined users have historically been willing to learn English in order to free software, in order for free software to gain an acceptance in the wider community of users, it must able to both display a user interface in the user's native language and allow the user to manipulate text in their native language.

A number of separate areas must be addressed when making a program suitable for handling text in multiple languages (a process known as *internationalization*. First, the user must be able to input text in their native language — this may simply be a matter of changing the keyboard mapping, but it may also be a more complicated process involving dictionary lookup by separate programs known as *input methods*. Then, the program must preserve the structure of the text during any manipulation it does of the text. Some common assumptions, such as assuming each character is one byte may mangle a stream of internationalized characters. The program must pick appropriate translations of any messages it displays to the user, and finally, it must be able to render strings in the user's native language correctly. This paper will concentrate on the last aspect, rendering.

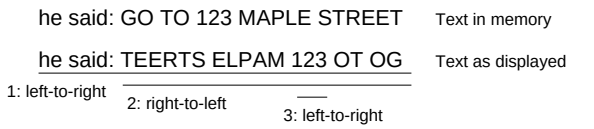

Figure 1: Transformations while displaying complex-text languages

Rendering internationalized text is often assumed to be simply a matter of fonts. All languages are thought to be like English where it is simply a matter of picking the right symbols from the font and displaying them in the order they occur in the memory representation. In this view, internationalized rendering is a simple matter. All you need is a character set that includes all the characters in the world's languages (Unicode [Unicode] fits this role) and a font for that character set. Of course, things are not that simple.

First, a number of languages, notable Arabic and Hebrew are written from right-to-left, instead of from left-to-right, so the rendering process needs to be able to deal with that ordering. In fact, text in these languages usually consists of a mix of right-to-left text and left-to-right text (numbers, foreign words.) So, a complicated reordering process is needed between the in-memory representation and the actual drawing process. Figure 3 shows a schematic representation of the reordering process.

Arabic also introduces some other complications. The shape of each character is different depending on whether it occurs at the beginning of a word, in the middle of a word, at the end of a word, or by itself. So the right *glyph* needs to be selected in each context.

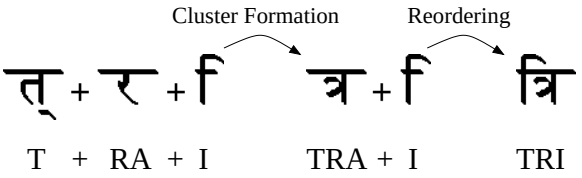

Figure 2: Transformations while displaying complex-text languages

Another group of languages that needs special attention are the languages of South Asia, often known as *complex text* languages. In these languages, the characters making up a syllable interact in complex ways to produce the final rendered form. This can involve reordering, combining characters to make ligatures that appear very different from the original character, and stacking multiple glyphs on top of each other vertically. A group of interacting characters in one of these languages is known as a *cluster*. (See Figure 2.)

Algorithms such as line-breaking also need detailed knowledge of a language. Although for western languages, a simple job of line-breaking can be done by breaking on white space, other languages, such as the languages of East Asia or Thai, are written without any white space at all, so linguistic information is needed. For Thai, correct line breaking actually needs to be done by first splitting the text into words using a dictionary.

So, we see that to properly render many of the world's languages, we have to be able to deal with a rendering process which is considerably more complex than just slapping down glyphs in the order that characters appear in memory. It is interesting also to note that many of the same issues that appear above, such as ligatures, alternate glyph selection, and repositioning of characters (kerning) are also important when doing a high-quality job of displaying English or Western-European languages. So, if we can properly handle these issues for internationalization, we gain, as a side benefit, a much higher level of typographic sophistication.

Clearly, you don't want to be handling all of these details in your application. So, to be able to have a

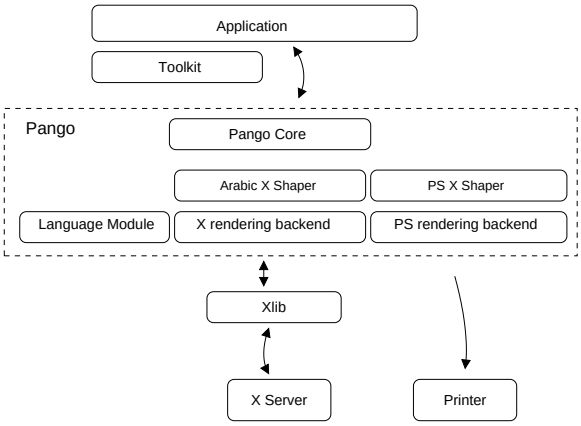

Figure 3: Architecture of Pango

system where all these details, and many more are properly handled for each language, what we need to do is move to a higher level of abstraction. We need a system where the application programmer simply presents the system with a chunk of text, and all the details of wrapping lines, laying out the text, choosing glyphs and rendering is handled for the programmer. The Pango library has designed for this purpose; it encapsulates all the necessary knowledge about various languages and scripts and presents the application programmer with generic model of lines and paragraphs.

2 Architecture

There are several principles that underly the design of Pango. The first one is that Unicode is used as a common character set throughout the system. Although, as mentioned above, supporting Unicode is not enough to handle internationalized rendering, by standardizing on Unicode, and on UTF-8 as the encoding of Unicode, there is no need for the application, the toolkit, Pango, and Pango's language-specific modules to negotiate the encoding to be used.

The second principle of Pango is modularity. The code specific to each language is contained in a separate, dynamically loaded module. This has several benefits. First, it reduces the amount of code that is contained in the main library. Second, it allows modules for specific languages to be

developed and distributed by teams familiar with those languages, instead of tying the development of support for a particular language to the release cycle of the core system.

The third principle of Pango is rendering system independence. The same tasks need to be performed whether using X fonts to draw to the screen, drawing into a off-screen buffer in some other fashion, or printing to paper. Pango provides an API that can be used for all purposes to format the text. Only in the final rendering step do rendering-system-dependent calls need to be used. Because some of the intermediate steps (for instance, positioning glyphs with respect to each other) do depend knowledge from the rendering system, these portions need to be rendering system dependent. So each language module is split into pieces: a rendering system-independent module (that knows how to do such tasks as find the permissible breaks within a line) — the *language module* and a rendering-system-dependent module for each supported rendering system — the *shaper module*.

At the lowest level, the process of rendering text consists of a number of steps:

- **I**temization. In this step, the input text is divided a input Unicode string is analyzed and broken into *items* that each handled by a single language module and have a single direction (left-to-right or right-to-left). If the application has applied additional markup on the string to control things like style or font size, it may further subdivide each item into pieces that have the same font.

- **B**oundary Resolution. Textual boundaries such as word boundaries and line breaks are determined for each item. In most cases, a generic algorithm suffices for this process, but in some cases, a language module will override the generic algorithm with a more specific one. The `pango_break()` call handles boundary resolution.

- **S**haping. The next step is to take the characters within each item and convert them into glyphs. There may potentially be either more glyphs or less glyphs in the final string then there were characters. The `pango_shape()` is used to convert characters to glyphs. Until we have done this conversion, we cannot compute the size on the screen of the text we need to draw for use in a line-breaking algorithm.

- **L**ine Breaking. The results of shaping and boundary resolution are used to choose where to break lines that need to be wrapped. If breaking lines involves dividing items, then we'll need to call `pango_shape()` again to do final glyph selection and positioning, since breaking words may require different glyphs to be selected. (The line breaking algorithm described here is a simple one that works for most uses. High quality text display, as for publishing may require a more sophisticated algorithm that globally optimizes for the best line break positions taking into account changes in the glyphs that occur during shaping.)

- **R**endering. The result of the shaping and line breaking process is a set of *glyph strings*, which is a list of glyphs from the font, along with positioning information for each glyph. Since rendering is specific to the type of fonts being used and how they are being rendered, the core Pango library doesn't handle rendering; the rendering system specific libraries included with Pango, such as `libpangox` for X fonts and `libpangoft2` for using TrueType and Postscript fonts via the FreeType library, include basic rendering routines, or applications can do their own rendering.

Since we claim rendering-system independence, we should quickly mention how we handle font information. Pango has an abstract class *PangoFont* which represents a font in some rendering system. The *PangoFont* class includes operations such as determining the overall metrics of a font, determining the metrics for an individual glyph, and determining which Unicode characters a font covers. It doesn't include, however, any operations for actually rendering the font, or finding out how glyph indicies correspond to Unicode. To use Pango with a particular rendering system, a subclass of *PangoFont* is created that handles a particular type of font. For instance, *PangoXFont* is the subclass of Pango font for handling X fonts.

While dealing with these steps may be easier than having to deal with the particular details of how to render Arabic or Hindi, it still is pretty complicated.

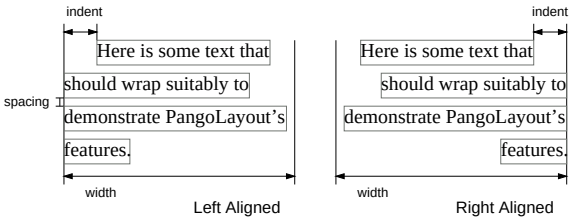

Figure 4: Parameters that can be set on a PangoLayout object

Most applications writers would certainly not want to deal with this level of complexity, and even people writing new widgets for a GUI toolkit might be scared off. So Pango provides a higher level abstraction, the *PangoLayout* object.

A *PangoLayout* object is initialized with two things - a block of Unicode text and a list of attributes to apply to runs of characters within the text. These attributes include, among other things, font family, font style, font size, color, and language tags (is the text in English or in German — important for such things as hyphenation.) Various properties, such as the line width, line spacing, and indentation, can also be set on the entire layout. (See Figure 4.) At this point, PangoLayout internally handles all of the steps of boundary resolution, itemization, shaping and line breaking, and the final glyphs can be immediately extracted.

PangoLayout also contains extensive support for interactive editing. For example, it contains functions for mapping between locations on the screen and locations within the text and for handling properly internationalized cursor movement with arrow keys. Because there isn't a one-to-one correspondance between characters in the text and the displayed glyphs, and because the text can be reordered in the output process, these operations are considerably more complex than they would be without internationalization.

The architecture of Pango is inspired by a number of existing systems. The basic layout pipeline follows Microsoft's Uniscribe system [Uniscribe] fairly closely, although it has been somewhat simplified to combine excess steps, and provide a nicer interface to the user. However, the high level *PangoLayout* object has no equivalent in Microsoft's API. Similar high level text objects can be found in other systems, such as the Java2D [Java2D] text API. Of course, a fundamental difference between Pango and all existing proprietary systems is that the code is completely open for inspection and modification. Anybody can extend Pango to work with new languages and new rendering systems.

3 Implementation and Status

Pango is written in C using the GObject system. This object is based on the GtkObject system found in older versions of the GTK+ widget toolkit. As part of the major revision of GTK+ between GTK+-1.2 and GTK+-2.0 (which includes moving to Pango for text rendering), this facility was split out into a separate library, allowing its use from libraries not tied specifically to GTK+, such as Pango. The use of the GObject system provides several benefits: it provides a framework for writing object oriented code in C, making it convenient to use such techniques as inheritance and polymorphism, and it provides a standard type system and method of memory management making it easy to writing bindings between Pango and different interpreted and compiled languages.

The GObject library is distributed as part of the GLib package. Pango also takes extensive advantage of other facilities in GLib: data structures such as lists and hash tables, unicode manipulation and code conversion, the xml-like GMarkup parser, and the GModule library for loading dynamic modules in a portable fashion.

At the time of writing, Pango is nearing version 1.0 and is extensively used in the current version of the GTK+ toolkit. While future extensions of the API are planned to support high-end features for applications such as desktop publishing, the current API is completely sufficient for use in a widget toolkit, for editing and screen display.

Support has been written for a number of different rendering systems: for traditional X fonts, for client-side fonts using the Xft library and Xrender extension [Packard], for fonts rendered locally using the FreeType library, and for fonts in the Win32 API. The rendering systems that are currently present correspond more-or-less to the ports of GTK+ that are underway: the X and Xft backends for the X11 port, the FreeType backend for the Linux-framebuffer port, and the Win32 backend for the Win32 port of GTK+.

The available set of shaper modules for different languages is equally broad: since shaper modules are specific to both language and rendering system, the exact set varies for the different rendering systems. The X font backend has the largest set, including modules for Arabic, Hebrew, Thai, Korean and 7 different Indic languages, in addition to the "basic" module, which handles rendering for all languages that don't require special shaping algorithms. The languages handled by the basic module include, among other things: languages written in the Roman, Greek, and Cyrillic alphabets, and the ideographic scripts of East Asia, used for Chinese and Japanese.

4 Pango and GTK+

While it should be emphasized that Pango is not GTK+ specific: it can be used for both applications other than widget toolkits, such as printing, and with widget toolkits other than GTK+, the primary use of Pango to this point has been within the development version of GTK+, and this influenced what parts of Pango development are most advanced.

In the development version of GTK+, Pango is used for all text handling, with the exception of a few deprecated widgets left for compatiblity purposes. This includes, in particular, the *GtkLabel* widget for static text display, *GtkEntry* single line text editing widget, and the *GtkTextView* multi-line text widget. This means that everywhere text is found within GTK+, the full capabilities of Pango are supported.

In all cases, GTK+ uses the high-level *PangoLayout* interface to Pango, rather than going directly to the lowlevel layout functions. This gives an indication that *PangoLayout* is powerful enough for a wide range of uses, and only the most specialized applications would need to go to the low-level interfaces. The *GtkLabel* and *GtkEntry* widgets each contain a single *PangoLayout* structure. The *GtkTextView* widget, which is designed to handle much larger amounts of text, takes a different route. It uses a PangoLayout for each paragraph, but instead of keeping these layouts around, it creates them as necessary, then destroys them again when no longer needed.

The obvious benefit of the move to Pango for users of GTK+-2.0 is the enhanced internationalization capabilities. The improvements from Pango enable and are accompanied by other internationalization improvements in GTK+-2.0 — a new framework for text input (input methods), better keyboard layout handling, and support reversing widget layouts for right-to-left languages.

But while internationalization is the obvious benefit, Pango also brings substantial benefits to users who don't need specialized internationalized text handling. Foremost amongs these improvements is a sane font system. Font handling in GTK+-1.2 was closely tied to the handling of fonts in the core X protocol. To load a font required dealing with the obscurity of names like `"-*-times-bold-r-normal--*-140-*-*-*-*-iso8859-1"` in GTK+-2.0, the name for the same font is simply `"Times Bold 14"`. Moreover, since Pango abstracts font handling away from the toolkit, portability to other windowing systems is simplified, and within X11, moving to using client side scaleable fonts with Xft and Xrender is very natural, allowing, among other things, support for anti-aliased fonts.

A simple, but handy feature enabled by Pango is support for a simple XML-like format for embedding formatting in labels. Creating a label with colors and italics can be as simple as:

```
GtkWidget *label = gtk_label_new (NULL);

gtk_label_set_markup (GTK_LABEL (label),
           "<span color=\"red\">Red</span>"
           "<big><i>Text</i></big>");
```

5 Future Plans

Future plans for Pango center on expanding the range of applications for which it can be used by adding support for the sort of features that are needed for high-quality printing. Some of these include:

- **Better hyphenation and line breaking**. Pango currently uses a simple line breaking algorithm, and does not break words by inserting hyphens at all. Better algorithms are needed for applications other than screen display.

- **Justification**. Pango currently only allows setting paragraphs ragged-left or ragged-right. Adjusting spacing to create even paragraphs is needed for This process also requires internationalization. For examlple,Arabic is notable for justifying by extending the baselines within words rather than adding space around words.
- **Vertical text**. While Chinese and Japanese are most frequently written left-to-right in rows for computer work, in printed material such as books, writing vertically in columns is very common.

Extending the range of languages that Pango supports is also a goal, though even today the range of supported languages covers the vast majority of potential users, especially for X fonts. In the future, the primary focus for language support for X will move away from supporting the legacy X font system, to rendering client-side fonts via the Xrender extension.

In particular, the OpenType font format[OT] is promising as a way of getting the font information needed to do really good internationalized text rendering. OpenType, developed jointly by Microsoft and Adobe is essentially an extension to the TrueType format, with the addition of a number of extra tables. These tables include information about selecting different glyphs depending on context, as is needed for Arabic, information about glyph for character combinations, as is needed for complex text languages, and information about positioning accents and characters. While OpenType is an extension of TrueType, it allows for either TrueType outlines or Postscript Type1 outlines. The ability to have Postscript outlines is attractive for free software, since patent problems cause difficulties for free implementations of the TrueType format.

Current versions of Pango already include basic support for reading OpenType tables, and an Arabic shaper that can take advantage of this information. Substantial future work still needs to be done to extend this to a wider set of languages and handle fine details such as accent placement.

Work on Pango is certainly not complete with Pango-1.0. However, Pango as it currently exists already provides a broad set of capabilities within a flexible framework that can be extended to new languages and to new rendering systems. This allows toolkits such as GTK+ to provide internationalization capabilities for users that go significantly beyond existing capabilities in the open source world.

More information about Pango and GTK+ can be found at:

`http://www.pango.org/`
`http://www.gtk.org/`

6 Acknowledgments

The set of contributors to Pango is already too large to list here, but I'd like to especially thank the people who dove in and wrote shaper modules with little or no documentation, including, among others, Changwoo Ryu, Sivaraj Doddannan, Karl Koehler, and Robert Brady. Also, Havoc Pennington, for cleaning up various messes I left in *PangoLayout*.

References

[Java2D] *Programmer's Guide to the Java 2D API* Sun Microsystems. `http://java.sun.com/j2se/1.3/docs/guide/2d/spec/` (1999)

[OT] *OpenType Specification* `http://www.microsoft.com/typography/otspec/`, (2001).

[Packard] Keith Packard, *A New Rendering Model for X* Usenix Technical Conference 2000.

[Unicode] The Unicode Consortium, *The Unicode Standard, version 3.0*, Addison Wesley Longman (2000).

[Uniscribe] Microsoft Corporation *Uniscribe* `http://microsoft.com/typography/developers/uniscribe` (2001).