Internationalizing the GTK+ User Interface Toolkit

Owen Taylor
Red Hat Software
March 23, 1999

1 Introduction

A graphical user interface toolkit is one of the core components of a modern system. The toolkit is responsible for providing the interface between the raw windowing system's button presses and line drawing code and the application. It provides facilities for displaying such common user interface elements as menus, push buttons,and text entry fields.

Because of the critical position that a toolkit fills, sitting between the user and the desktop, it also provides an essential role in internationalized and multilingual applications. The toolkit typically handles the conversion of the raw keystrokes from the user into the final text. For some languages, in particular those of East Asia, this process can be quite involved, and may involve talking to a external server, called an input method.

Once the text has been entered into the application, any internal manipulations that the toolkit does on the text must also respect its international nature. For instance, when the toolkit is handling cursor motion within a text entry field, the motion must be by complete characters, even if the internal representation of the data uses more than one byte per character.

The toolkit is also responsible for properly displaying text in the appropriate font and otherwise respecting the conventions of the user's language. This text may come from user input, or from the application, but the toolkit may also need to provide strings itself. That situation commonly arises when the toolkit provides such high level widgets as a file selection or font selection dialog. Strings such as "filename" or "font" that occur in these dialogs must be displayed in the user's native language. This process, called *localization* is the last component of the toolkit's responsibilities for internationalization.

GTK+, the GIMP Toolkit, has recently become one of the most widely used open source toolkits for the windowing system. It is used in a number of major projects such as the GIMP (GNU Image Manipulation Project) and the GNOME desktop.

GTK+ began as part of the GNU Image Manipulation Program (GIMP) project. In preparation for the 1.0 release of the GIMP, the authors of the GIMP, Spencer Kimball and Peter Mattis decided to switch from using the propietary Motif toolkit to a toolkit that would encourage contributions from the free software community. Not finding anything suitable, Peter began work on GTK+ in the Fall of 1996. A great deal of interest in GTK+ appeared, including from people interested in using GTK+ in international settings. In late 1997, a set of patches from Takashi Matsuda were added to enable internationalized input into GTK+. After the 1.0 release of GTK+ this work continued and the 1.2 release that was made in February 1999 includes comprehensive support for the X locale model. The current work in GTK+ is largely aimed at extending the internationalization of GTK+ beyond the scripts of Asia and Europe, and at providing the ability to display text in multiple scripts simultaneously. This

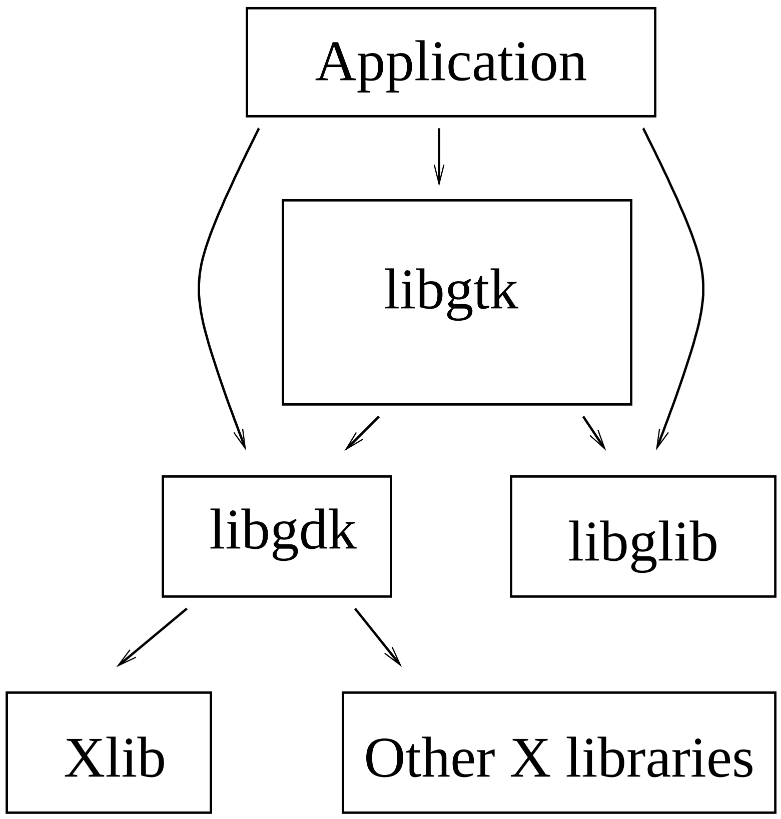

Figure 1: The components of GTK+

work will be described in the second part of this paper.

GTK+ consists of three component libraries (see Figure **??**), libgtk, which contains the widgets of GTK+, libgdk, which provides low level event handling and drawing functions, and glib, a library of portability routines and abstract data structures that makes programming in C convenient and efficient.

2 The current state of GTK+

2.1 Input

For handling internationalized input, GTK+ currently uses the X Input Method Extension, or for short, XIM. XIM provides a framework for communicating with input methods. These input methods either can be external or they can live in a separate process. External input methods use a protocol defined by the

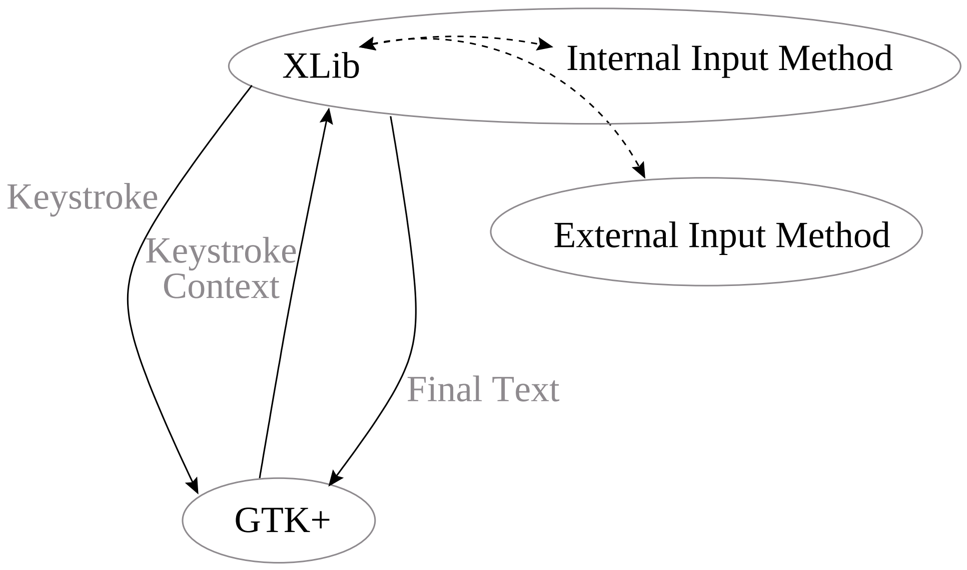

Figure 2: The architecture of the X Input Method Extension

XIM. An internal input method is used for compose key handling for European languages.

The architecture of XIM is shown in Figure **??**. Each keystroke is first retrieved by the toolkit from the X libraries. The toolkit then calls `XFilterEvent()` which allows the input method to intercept keystrokes and process them internally. The final string is then returned to the library.

During input, feedback may need to be displayed to the user. For instance, when a user is entering Japanese text, it first is entered as a string of phonetic symbols (kana), the phonetic symbols are then converted into the final ideographic form in a separate step. XIM supports various ways of of displaying feedback. (See Figure **??**). In the Root Window style, all feedback is shown in a separate window. In the Off-the-spot the feedback is shown in a separate portion of the applications window. The Over the spot style, *pretends* to display the feedback in place, by putting an undecorated window on top of the location where the user is entering the text. Finally, in the On-the-spot style, display is handled by the application instead of the input method, and is actually drawn in place.

Input methods are available for quite a few different languages. In addition to the built-in Input Method for European languages mentioned above, XIM-compliant input methods are available for Japanese, Korean and Chinese.

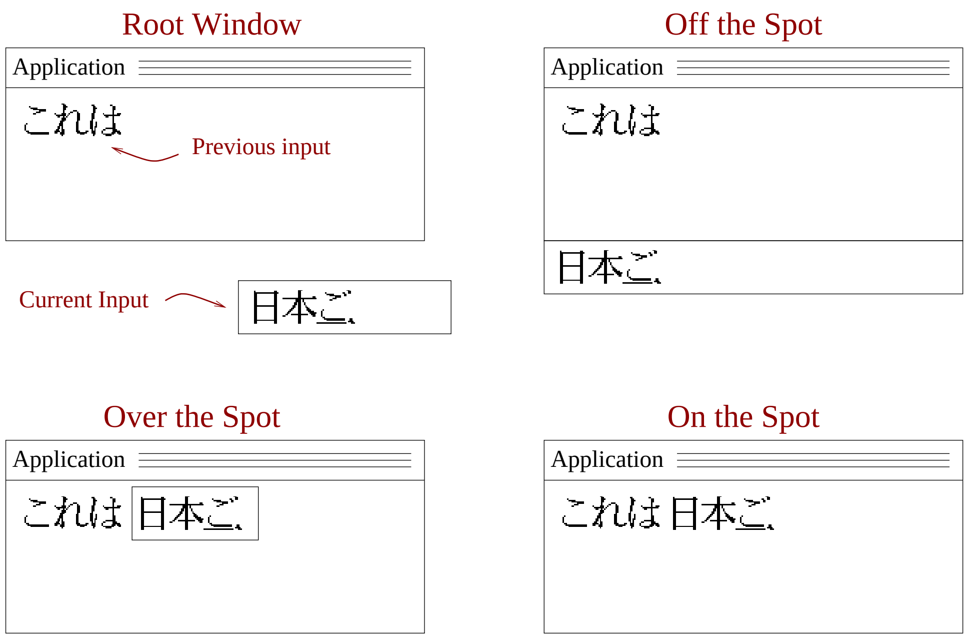

Figure 3: Possible input styles

2.2 Manipulation

3 Manipulating Strings - GTK+ 1.2

- Uses a locale-specific character set (e.g. JIS 0208-1983)
- External representation is multi-byte
- Wide characters used internally for complex manipulation

4 Difficulties with multi-byte

- Hard to work with multi-byte string in unknown encoding
- Only can rely on old ISO C `mbstowcs`, etc.
  - Not reentrant
  - Dependent on current locale
- Sometimes C library does not agree with X about multi-byte encoding.
  - C library may have strange ideas

– X may be compiled with `-DX_LOCALE`

– GTK+ does all multibyte `<->` widechar conversion by converting to an X Text Property and back.

4.1 Output

Currently, output of internationalized text in GTK+ is done using the facilities of X. Instead of using a font, the application uses a *fontset* which is a list of fonts for different character sets.

A typical fontset might be something like:

```
 -adobe-helvetica-bold-r-normal-*-12-*-*-*-*-*iso8859-1
 -*-*-bold-*-*--12-*-*-*-*-*-ksc5601.1987-*
```

Which says to use Adobe Helvetica for the character set iso8859-1 (Latin 1) and to use whatever fonts are available for the character sets in ksc5601.1987 (the Korean national standard for character sets).

Note that the interpretation of a string is done according to the locale when Xlib is initialized; if the locale is ja˙JP, then X will expect the fontset to contain Japanese fonts, while if the locale us kr˙KR, then it should contain Korean fonts. So displaying text in multiple different languages is difficult.

4.2 Localization

As mentioned above, the text in GTK+'s standard dialogs (for example the file, font, and color selection dialogs) needs to be translated for the user's native language. To accomplish this, GTK+ uses the `gettext()` interface; this interface was originally devoloped by Sun, and is available as part of a number of operating systems. The GNU implementation of gettext is also available separately and can be installed on systems without a native implementation of gettext.

When GTK+ needs to display a string for a user, it calls the `dgettext()` function. For instance, to display "Cancel", GTK+ calls:

```
dgettext("gtk+", "Cancel")
```

This will return "Cancel" when the locale is set to "C" or "en˙US", but "Annuler" when the locale is set to "fr˙FR". Because GTK+ uses dgettext() which takes an additional *domain* parameter, it uses a separate message catalog from the application. This means that applications can localize themselves however they want: they can use `gettext()` without interference from GTK+, or they can use a different system of localization.

5 Future Directions

In the future, GTK+ will be switching over from the system of internationalization described above, where a different encoding is used for each locale

There are a couple of reasons for this change. Second, Unicode makes it possible to extend the range of supported languages to languages (such as those of Africa or Central Asia) that do not have long traditions of local standards. Second, Unicode eases the load on the application developer, because they only need to deal with a single encoding. Finally, Unicode is the emerging standard for representation of multi-lingual text. It is being used in such systems as Microsoft Windows NT, Java and Perl.

- Unicode support
- Bidirectional rendering (Arabic, Hebrew)
- Complex-text languages (Hindi, other Indian languages) (After GTK+ 1.4)
- Multi-language support at same time.

6 Unicode

- Makes things simpler for application writers
- Good support for languages beyond Europe and CJK
- Enables multi-lingual applications.

7 UTF-8

- *One* multi-byte encoding isn't bad
  - Can determine if a byte starts a charcter, so can iterate backwards
  - Efficient for size. (Covers all two-byte characters in no more than three bytes)
  - Size efficiency can mean fast if memory bandwidth is limiting factor.
- Not limited to 16 bits.
- Compatible with existing interfaces (`gettext()`, `sprintf()`, etc.)

8 Bidirectional Rendering

One complication that will be dealt with in future versions of GTK+ is *bi-directional rendering.* In a number of languages, including Arabic and Hebrew, the direction of righting is not right-to-left, as is it is for languages written with the Roman script, but right-to-left. This, in itself, would not be a big problem - you would just have to reverse the order of drawing all characters onto the screen. However, normally text in these languages is a mix of those right-to-left characters (the language itself) and left-to-right characters (numbers and foreign words). So a somewhat complex algorithm is needed to take a string stored in memory and figure out the correct ordering for displaying it on the screen.

- Arabic, Hebrew mostly right-to-left
- But typically have embedded sequences of left-to-right text
- Use Unicode standard algorithm for determining directionality
- Selecting, cursor movement complicated

8.1 Widget Mirroring

It is not simply enough to reverse the order of the each string in the interface — since a speaker of Hebrew or Arabic expects things to read from right to left, the interface itself needs to be reversed. Labels, for instance, should appear to the right of the entry field that they label, instead of to the left of the field.

Since GTK+ does layout with heirarchical containers, this is generally straightforward. When adding a series of widgets into, for instance, a GtkHBox. (A widget that contains a horizontal sequence of other widgets), GTK+ merely needs to add the widgets from right to left instead from left to right. There are some cases that do, however, require intervention from the application.

Figure **??** shows some of these cases. In (b), the "left", "right", and "center" labels should not be reversed, because there positioning on the screen is physical, not sequential. So there needs to be the ability for the application writer to flag containers that should not be reversed.

In (c), reversing the back and forward labels is appropriate, but not only the positions, but the pixmaps themselves need to be modified. For a speaker of a right-to-left language, forward is not to the right, it is to the left.

9 Current questions

- Is it possible to only support the On-the-spot style of input?

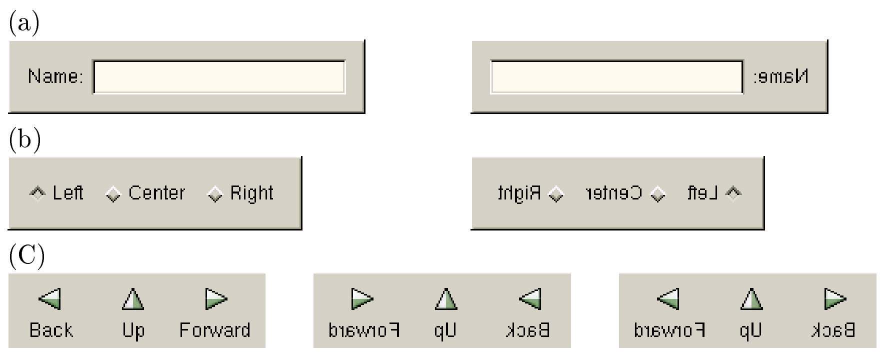

Figure 4: (a) User interface reversal (b) User interface elements that should *not* be reversed (c) Pixmaps that need to be reversed

- How do we do code conversion on legacy OS's?
- How do we render "Better than X"?
  - Integration with GnomeText (Raph Levien)

10 Acknowledgements

- Internationalization of GTK+
  Takashi Matsuda Akira Higuchi
- Original Authors of GTK+
  Peter Mattis Spencer Kimball Josh Macdonald
- The GTK+ Team
  Shawn Amundson Jerome Bolliet Damon Chaplin
  Tony Gale Jeff Garzik Lars Hamann
  Raja Harinath Carsten Haitzler Tim Janik
  Stefan Jeske Elliot Lee Raph Levien
  Ian Main Frederico Meña Paolo Molaro
  Jay Painter Manish Singh