Cross Platform Support in Tk

Ray Johnson
ray.johnson@eng.sun.com
Scott Stanton
scott.stanton@eng.sun.com
Sun Microsystems Laboratories

Abstract

Tk currently supports a wide variety of X based platforms. In this paper we describe work in progress at Sun Microsystems Laboratories to make Tcl and Tk support non-X platforms. The changes described here will allow script writers to migrate Tk applications to Microsoft Windows and Apple Macintosh systems with little effort. In addition, scripts running on these systems will have the appropriate look and feel for each system.

Introduction

Tk has enjoyed tremendous growth over the past few years. However, the user community could potentially grow by another order of magnitude if Tk could run on the more widely used Windows and Macintosh platforms. Currently an effort is under way at Sun Microsystems Laboratories to port Tk to the Macintosh and Windows. We see Tk as an easy way for Windows and Macintosh developers to develop new vertical applications. Combined with communications extensions, Tk may also become the best way to write collaborative applications that run in a cross platform environment.

We have two fundamental goals for the development of this port. First, we would like to present the native look and feel for each platform. Most Windows and Macintosh users are not familiar with the Motif look and feel, and there is strong evidence that users would reject a non-standard interface [Apple92]. Second, we would like to have a script written on one platform run unmodified on the other platforms. In fact, we hope we can achieve the proper look and feel on each platform without scripts needing to explicitly test and code for a given platform.

In working towards these goals, we plan to raise the level of the interface in Tk to accommodate the various features of each platform. Many features in Tk will have a more abstract interface which we hope will avoid the least common denominator problem that plagues other cross platform toolkits.

The six sections that follow highlight the solutions we currently plan to employ in our porting effort. This paper describes work in progress so the solutions discussed may not be reflected in the final implementation. However, the issues raised do reflect real problems that need to be addressed. We present here what we think are potential solutions to the problems of cross platform portability of Tk.

Using Native Widgets

In order to fit seamlessly into different environments, Tk must adopt the correct look and feel on each platform. The basic widgets (e.g., button, scrollbar, entry) are available on each platform, but they look and behave quite differently from the corresponding widgets on other platforms. There are two approaches Tk could take to presenting the right look and feel on each platform: emulating widgets, or using the native widgets.

Emulating the system widgets with platform specific rendering code and bindings is attractive because it gives Tk the most control over how the widgets interact with the rest of Tk. This is the approach Tk currently employs to present the Motif look and feel. It is easy to include all of the appropriate hooks in the emulation code in order to provide a very rich interface to the widgets. Unfortunately emulating widgets makes it difficult to keep up with changes in the window system. With each new release, the look of the system widgets tends to change in subtle ways. It requires a great deal of effort to get the emulation correct for every new version of each supported platform.

The alternative approach is to use the built-in widgets for each platform and provide wrappers that emulate the standard Tk widget interfaces. By using the native implementations of the basic widgets, Tk will automatically keep up with changes in the window system. Since system interfaces change far less frequently than the details of how a widget is rendered, the effort required to keep up with a moving target is reduced. In addition, the behavior of each widget will be appropriate for the given platform. This is the approach we currently intend to use for the Macintosh and Windows platforms.

The biggest drawback to this approach is the difficulty in wrapping the system widgets while maintaining the existing Tk interface. Each system widget will have to be wrapped in a unique way, and the interfaces may not have a one-to-one correspondence. In practice, however, we should be able to find effective workarounds to enable us to provide the proper Tk functionality.

Common Dialog Boxes

In some cases, making individual widgets look and act like the corresponding native system widgets is not enough. The system may have common composite interfaces that are just as much a part of the native look and feel as the buttons and scrollbars. For example, both the Macintosh and Windows have predefined file selection dialog boxes which applications are expected to use in order to keep a consistent look and feel. Other examples include file save, color selection and font selection. Because the components of these dialogs are arranged differently on different platforms, it is not sufficient to plug platform-specific widgets into a standard layout.

In order to achieve the correct look and feel for these common dialog boxes, Tk must provide an abstract interface to the functionality provided by each system. Rather than relying on applications to correctly construct a file selection dialog out of native widget components, Tk must provide a command that creates the appropriate system dialog and returns the user's choices. For example, to prompt the user for a file to open, a script could issue the command:

```
dialog open {{*.c "C Files"} {*.h "Include Files"}}
```

The user will see the appropriate dialog box for the system they are using. Once they have chosen a file, the `dialog` command will return the selected file name. Of course, each platform will have a different implementation for the dialog command.

Tk will supply a set of the most commonly used dialog boxes on all of the systems it supports. In addition, Tk will provide a few "generic" dialog boxes for common operations like displaying a dialog with a message and a set of buttons. By providing a generally useful set of dialog boxes that are available across platforms, Tk will allow scripts to be portable, yet still interact with users in familiar ways.

Indirect Bindings

The ability to bind to any user event makes the `bind` command one of the more powerful features of Tk. It is trivial to bind to key sequences, mouse clicks, and modifier keys, in almost any combination. Unfortunately, binding to specific keys and mouse clicks makes it difficult to create portable Tk applications. The problem is that different platforms use different keyboard or mouse bindings to invoke the same action (e.g., copy or select). For example, consider the action of pasting from the clipboard. Figure 1 shows the various ways in which the Macintosh, Windows, and UNIX platforms accomplish pasting.

Macintosh:	Command-V or the function key F4
Windows:	Control-V
UNIX:	Middle button, Control-Y, or the function key L8

Figure 1

Two important things should be noted from this example. First, different platforms may use different input devices for a given semantic event (e.g., keyboard or mouse). In fact, the various platforms may not even have certain input events such as Button-2 and Button-3. Second, a given platform may have several bindings that invoke the same action. Any solution must take into account both of these factors. Furthermore, we would like the mechanism to be extensible by Tk programmers.

The solution we have chosen to implement is *indirect bindings*. Indirect bindings will allow the programmer to bind to semantic events rather than low level keyboard and mouse events. For example, instead of binding to `<Control-v>` a Tk programmer would bind to `<sysPaste>`. The sysPaste event indirectly represents any sequence of input events that should invoke the commands associated with the paste operation. The actual keyboard or mouse events that represent sysPaste are platform dependent, but the script does not need to know about them.

Tk will come with a portable set of indirect bindings. These bindings will include common actions such as cut, paste, open, and save. In addition, Tk programmers will be able to create their own semantic events that may be specific to their application domain. Although the definitions of these bindings will be platform specific, all of the platform dependencies will be localized to a binding map rather than scattered throughout the script. By using indirect bindings, a Tk script can be much more portable than a script that specifies explicit bindings for all of its actions.

Menus

Menus present an interesting problem for those creating portable applications [Nicholson91]. Various platforms have different models of where pull down menus belong and what they might contain. UNIX applications often do not have menu bars. When they do have menus, however, the menu bars usually appear at the top of each specific toplevel window. Windows applications typically have menu bars on each toplevel that is not a popup dialog box. In addition, most windows have a system menu with a standard set of entries. Macintosh computers, on the other hand, rigidly stick to the notion of one menu bar for the entire application and no menu bar on individual windows. In addition, the menu bar typically exists even when no windows are visible.

If Tk applications are to have the correct look and feel, they should adhere to the native way of handling menus on each platform. Either Tk scripts must explicitly take into account the differences between platforms, or an abstraction must exist that takes some generic menu specification and maps it as appropriate to each platform. For maximum portability of Tk scripts we are considering the latter approach.

To address this issue, Tk will have a widget called the menu bar. Each toplevel window may have a menu bar associated with it. Menus and menu items will be attached to the menu bar along with information that will guide Tk with the placement of the menus on each platform. Let’s take as an example an application that can open and edit text or graphics files. This application would probably have a File and Edit menu that would be used for both types of documents. In addition, the text window may have a Format menu and the graphics window may have a Draw menu. Tk will handle this situation differently for each platform. The

UNIX platform would put a menu bar on each toplevel that would include the File, Edit and Draw menus for graphics windows and the File, Edit and Format menus for text windows. The Macintosh, however, would have one global menu that would change depending on which window is the currently active (or front most) window.

Furthermore, Tk may make decisions about where certain menu items should go. For example, the about menu item resides in the Apple menu on the Macintosh. Help menus also have specific locations on varying platforms. Unfortunately, the number of special cases is quite significant and our implementation will most likely not handle every situation correctly. For Tk scripts that have the best look and feel on each platform the Tk programmer will have to do some extra work. However, it is our hope that the menu bar widget will provide for instant portability in a large number of cases.

Fonts

Currently Tk uses X font names, such as `-adobe-times-bold-i-normal--10-100-75-75-p-57-iso8859-1`. These names are verbose and difficult to understand. In addition, since not all combinations of font attributes are available on a given system, it is difficult to pick font names that will match an existing font. If a requested font does not exist as specified, Tk gives up and returns an error. In order to hide the details of font naming conventions on different platforms, Tk needs a simpler and more robust way of specifying fonts.

The solution we have chosen is to create a font object that can be used anywhere a font name is used in Tk now. The "`font`" command can be used to create a new font object with a given name and attributes (see figure 2). The configuration options on a font object will specify the requested family, size, and style of the font. The resulting font object represents the closest approximation to the requested font that is available on the system; requests for fonts will never fail. The new font object can then be passed to widget commands like "`button`".

Specifying font properties as attributes of font objects is a more flexible and extensible method than the current X naming scheme. Rather than constructing baroque font names consisting of some combination of attributes and wildcard characters, the attributes can be specified directly and as needed. In addition, this allows Tk to map the requested

old style:

```
button .b -text "stop" -font -adobe-courier-bold-o-normal--
11-80-100-100-m-60-iso8859-1
```

new style:

```
font ButtonFont -family courier -style {bold italic} -size 12
button .b -text "stop" -font ButtonFont
```

Figure 2

attributes to the closest system font available. In the future, advanced font attributes could be supported without any loss of backward compatibility [White95].

This mechanism has an additional benefit. Right now, a script that needs to change the size of the font used by its widgets is required to set the `−font` configuration option on every single widget using the font. With the new font object mechanism, the script can simply reconfigure the font object, and every widget with a reference to the object will be updated immediately. By adding a level of indirection, font objects could be used as "styles" throughout an application.

Most importantly, by providing opaque, system-independent font objects, Tk can hide the details of font mapping from applications which don't need much control over fonts, while providing a flexible and extensible solution for those applications that do.

Options and Preferences

User preferences are essential to creating powerful and flexible applications for end users. Currently, configuration options and user preferences are extracted from either the option database or a "`.rc`" file in the user home directory. The option database extracts defaults for widgets from either the `RESOURCE_MANAGER` property of the root window or the `.Xdefaults` file located in the user's home directory. The other form of configuration is what are often called `.rc` files. The `.rc` file contains a script that Tk sources when the application starts. The script can, of course, encapsulate user preferences in any way it sees fit. In the end, most Tk applications create and maintain their own files to act as user preference files.

The larger problem, however, is that different platforms specify user preferences in wildly different ways. The Macintosh, for example, mandates that the preference files be stored in the Preferences folder inside the System folder. Windows applications look for their user preferences in `.ini` files. What is needed is a cross platform way of loading and saving user preferences that will abstract away the details of how user preferences should work on each platform.

We propose to extend the `option` command to meet these goals. Currently, the `option` command has a `load` subcommand that allows preferences to be read from a specified file. A `save` subcommand does not currently exist. Rather than having the load and save subcommands take a path as an argument, they will instead take a symbolic name that will map to an appropriate file on each platform. For example, the command "`option save builder`" would create a `.builderrc` file in the user's home directory on a UNIX machine and a `builder.ini` file in the Windows system directory on Windows. This keeps the application from having to specify a path to the preferences file thus insuring greater cross platform computability.

Schedule

As of June 1995, we are nearing completion on the porting of Tk to the Macintosh and Windows with the current Motif-like look and feel. We hope to have a preliminary release available by mid-summer 1995 that provides the functionality available in Tk 4.0 on all three platforms. In addition, we hope to have support for font objects as described in this paper available with this first release.

We expect to release a second version with support for native widgets in alpha form by the end of 1995. We hope to have most of the capabilities described in this paper available in time for the second release.

Conclusion

In order to write portable scripts in Tk, there are a number of changes that must be made to the basic Tk infrastructure. We have attempted to outline some of the more interesting issues and our intended solutions in this paper. Many of the proposed changes to Tk result in higher level interfaces that attempt to hide the differences between platforms. These new interfaces often result in cleaner and more powerful abstractions. The result of our efforts should be a Tk that is suitable for a wide variety of both vertical and horizontal applications without sacrificing the flexibility that is Tk's strongest feature.

Availability

For information on the most recent release of Tcl and Tk, or to obtain electronic copies of this paper, refer to the Tcl home page at http://www.smli.com/research/tcl.

References

[Apple92] Apple Computer, Inc. *Macintosh Human Interface Guidelines*. Addison-Wesley, Reading, Massachusetts, 1992.

[Nicholson91] Robert T. Nicholson. Designing a Portable GUI Toolkit. *Dr. Dobb's Journal*, pages 68-75, 117, January 1991.

[White95] Ronald G. White and John Biard. A Portable Font Specification. *Dr. Dobbs's Journal*, pages 28-34, 96, March 1995.