The following paper was originally published in the
Proceedings of the Fourth USENIX Tcl/Tk Workshop
Monterey, CA, July 10-13, 1996

In Search of the Perfect Mega-widget

Stephen A. Uhler
Sun Microsystems Laboratories

In Search of the Perfect Mega-widget

Stephen A. Uhler

Sun Microsystems Laboratories

ABSTRACT

Mega-widgets are Tk widgets that can be created entirely in Tcl, yet behave indistinguishably from their native counterparts. Although several different frameworks have been constructed to create and manage mega-widgets, none does a perfect job, as there are aspects of a native widget's behavior that can't be simulated strictly in Tcl. We propose a small set of enhancements to the Tk event binding and focus models that will enable mega-widget frameworks, including those that are currently available, to do a better job supporting the semantics of native widgets. This allows a script that uses a native widget on one platform, to use a mega-widget implementation of the same functionality on a different platform without the need to know whether the widget is native or not.

Introduction

One of the primary reasons for Tk's success is the ease in which user interfaces can be constructed, by gluing together one or more of the 13 native user interface elements, called widgets, in Tcl. However just as it is easy to use the native widgets, it has proven difficult to create new ones. They need to be written in C and conform to a fairly complex set of behaviors. As a result, each new user interface ends up being constructed out of the same basic building blocks every time. Even though common combinations of widgets can be manipulated as a unit by Tcl procedures (like tk_dialog), such procedures are hard to share, as their procedural interfaces tend to be different than the object based paradigms used to manipulate the native widgets.

Mega-widgets

The sharing and reuse of common combinations of widgets can be be greatly enhanced if they are packaged to look and behave exactly like the native widgets. These widget combinations that act like native widgets were dubbed mega-widgets by Mike McKlennan, who implemented them for his [incr Tk][1] mega-widget framework that is based upon [incr tcl].[2]

There have been other frameworks constructed for building mega-widgets, such as Tix by Ioi Lam,[3] and [Incr Widgets] by Mark Ulferts.[4]

Tk now runs on several different computing platforms that use a different look and feel for common user interface actions. As Tk moves to support the native widgets on each of its platforms, programmers will need to program to higher level interfaces than provided by the basic widgets, to insure that scripts not only run unchanged on various platforms, but conform to the native look and feel as well.[5]

Where useful native widgets are only available on some platforms they will need to be created on those platforms that don't provide them. Most likely these widgets will be created as mega-widgets, rather than the more difficult and arduous task of building them in C. As a consequence, a script that uses a native widget on one platform, might find itself using a mega-widget on another, completely unbeknownst to the program's author or user.

In the past it might have been acceptable for mega-widgets to behave almost like native widgets, whereas in the cross platform environment it is essential that they behave exactly like native widgets, as script writers might not have control over which widgets are native, and which ones are constructed as mega-widgets.

The Missing Pieces

Overall, the existing mega-widget frameworks do a good job providing users with the illusion that they are dealing with native widgets. The frameworks support the standard widget commands, such as `cgets` and `configure`, and handle configuration options in the expected way. Shannon Jaegar[6] suggests the features that a mega-widget framework should provide, as well as compare the various features of several existing frameworks.

In order to pass the litmus test of an ideal mega-widget, the user of that widget (the application programmer) must not be required to know whether it is native or not. Although it is acceptable for the programmer to ask about the inner workings of a mega-widget if they choose, assuming the mega-widget framework permits it, it is not acceptable to force the user to know about information that would be hidden if the widget was native. There are two areas where the frameworks fall short: event bindings and focus behavior.

The primary failing of the existing mega-widget frameworks is their inability to permit users to associate event bindings with a mega-widget without compromising the illusion of the widget as an atomic entity. Consider the following example:

```
some_widget .widget
bind .widget <ButtonPress-1> {
    puts "%W %x,%y"
}
```

If `some_widget` was a native widget, then no matter where inside the boundaries of `.widget` the user pressed the mouse button, the value of *%W* would be `.widget`, and *%x* and *%y* would be pixel offsets from the top left corner of `.widget`. On the other hand, if `some_widget` was a mega-widget, comprised of many sub-parts within its border, the same binding would likely report `.widget.sub-part` for *%W* where *sub-part* is the name of one of the mega-widget components, and *%x* and *%y* would no longer be relative to the `.widget` container, but instead be offsets from `.widget.subpart`. Thus the user is forced to deal with the details of the mega-widget's components, which they shouldn't need to know anything about.

Similarly, if the user issues the Tk `focus` command when the focus is inside a component of a mega-widget, `focus` would return `.widget.sub_part`, again exposing the user to information about the inter-workings of the widget that they shouldn't be required to know about.

Both of these problems result from Tk's notion that there is a one-to-one relationship between a widget and the underlying *C* window object for that platform. The events and focus information come from the underlying window object, not the widget command. In a mega-widget, there can be multiple underlying window objects for each widget instance, which causes Tk to report information about hidden parts of the mega-widget.

A Solution

Tk needs to be extended to correct for the discrepancies in the event bindings and focus command when using mega-widgets. These extensions should allow the existing mega-widget frameworks to work better without major changes, and do so in an architecturally neutral way that doesn't favor one style of mega-widget framework over another.

We have modified the `frame` command to accept another configuration parameter, `–command` to provide a convenient hook for attaching Tcl code to a widget command. Like the current `–class` option, `–command` may only be set at widget creation time, and not modified using `configure`. When the command option is specified, it changes the behavior of the frame, making it a mega-widget container. Calls to the created widget run the tcl code specified as the `–command` value, with any arguments concatenated. A new frame subcommand, `really` causes any additional arguments to be passed to the actual frame widget, and not be diverted to the `–command` string. For example, to create an instance of the *mega* widget called `.mega-widget,` the mega-widget framework would issue the following commands:

```
frame .mega-widget \
  –class Mega \
  –command {mega .mega-widget}

proc mega {name args} {
  $name really configure...
  # other processing...
}
```

The procedure `mega` would be called to actually process the arguments of the `.mega-widget` command, which would implement the behavior of the widget in Tcl code. The `–class` option could be used to initialize the default bindings for the `mega` widget. The `mega` procedure can access its containing frame using the `.mega-widget really` option.

In addition to diverting the widget processing to tcl code, the `–command` option turns on special behavior for event bindings and focus, so that keyboard and mouse events that arrive for any children of `.mega-widget` will be reissued as if they occurred for `.mega-widget` directly. This process is repeated for each enclosing mega-widget container, supporting arbitrary levels of mega-widget nesting. Thus if the user sets a mouse binding to a mega-widget that happens to fire while the pointer is over one of the widget's sub-windows, the event will fire once for the sub-window, to process any bindings that were set on the sub-window by the mega-widget builder. Then the event will be reformulated as if the sub-window wasn't there, and trigger again, executing the user's binding set on the container. The user sees the mega-widget as an atomic entity - just like a native widget, yet the mega-widget creator can set and process events on the sub-windows as needed.

As an aid to mega-widget builders and debuggers, an new subcommand of `winfo` has been added:

`winfo container` *window_pathname*.

This command returns the empty string if the specified window is not in a container. If the window is a container, *window_pathname* is returned. Finally, if *window_pathname* is a sub-component of a mega-widget, then the name of its nearest enclosing container is returned.

The final modification to Tk is in the `focus` command. The focus options that return window names (e.g. `focus` and `focus –lastfor`) now report the outermost container for the widget that has the focus. This brings the behavior of mega-widgets in line with native widgets. Focus will not report a sub-window name that should be hidden from the user. The new command:

`focus –container` *container_name*

can be used to determine the actual sub-window within a mega-widget that has the focus, by returning the outermost container (or native widget) that is inside *container_name* .

Implementation considerations

This mega-widget extension is implemented by adding two new flags to the Tk window structure, an *is_container* flag and an *is_contained_in* flag. The event rewriting and propagation only occurs if the *is_contained_in* flag is set, so there is no performance penalty if mega-widgets are not used. The entire extension is about 200 lines of C code which implements five functions: the recursive binding dispatch, the event rewriting, the new frame command option, changes to the `focus` command, and the implementation of `winfo container`.

Two *C* interfaces are provided to set and query the new state bits.

Summary and conclusions

We have made small changes to the Tk core that enables mega-widgets to behave indistinguishably from native widgets. This change permits some platforms to implement native look and feel using system widgets, and other platforms to implement the same functionality with mega-widgets. Scripts will be able to run unchanged with either type of widget, with no code changes required.

References

1. M. J. McLennan, ‘‘[incr TK]: Building Extensible Widgets with [incr Tcl],’’ in *1994 Tcl/Tk Workshop Conference Proceedings*, AT&T Bell Laboratories, Allentown Pa., July, 1994.
2. M. J. McLennan, ‘‘[incr Tcl]: Object Oriented Programming in Tcl,’’ in *1993 Tcl/Tk Workshop Conference Proceedings*, AT&T Bell Laboratories, Allentown Pa., June, 1993.
3. Ioi K. Lam, ‘‘Designing Mega Widgets in the Tix Library,’’ in *1995 Tcl/Tk Workshop Conference Proceedings*, Computer Graphics Laborarory, University of Pennsylvania, July, 1995.
4. Mark L. Ulferts, ‘‘[incr Widgets] An Object Oriented Mega-Widget Set,’’ in *1995 Tcl/Tk Workshop Conference Proceedings*, DSC Communications Corp., Switching Products Division, July, 1995.
5. Ray Johnson and Scott Stanton, ‘‘Cross Platform Support in Tk,’’ in *1995 Tcl/Tk Workshop Conference Proceedings*, Sun Microsystems Laboratories, July, 1995.
6. Shannon Jaeger, ‘‘Mega-widgets in Tcl/Tk: Evaluation and Analysis,’’ in *1995 Tcl/Tk Workshop Conference Proceedings*, Department of Computer Science, Univerity of Calgary, July, 1995.

Acknowledgements

Ken Corey created the prototype implementation, and Scott Stanton helped winnow the changes to TK to the smallest number that still do the job.