An X11 Toolkit Based on the Tcl Language

John K. Ousterhout

Computer Science Division
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

Abstract

This paper describes a new toolkit for X11 called Tk. The overall functions provided by Tk are similar to those of the standard toolkit Xt. However, Tk is implemented using Tcl, a lightweight interpretive command language. This means that Tk's functions are available not just from C code compiled into the application but also via Tcl commands issued dynamically while the application runs. Tcl commands are used for binding keystrokes and other events to application-specific actions, for creating and configuring widgets, and for dealing with geometry managers and the selection. The use of an interpretive language means that any aspect of the user interface may be changed dynamically while an application executes. It also means that many interesting applications can be created without writing any new C code, simply by writing Tcl scripts for existing applications. Furthermore, Tk provides a special `send` command that allows any Tk-based application to invoke Tcl commands in any other Tk-based application. `Send` allows applications to communicate in more powerful ways than a selection mechanism and makes it possible to replace monolithic applications with collections of reusable tools.

This paper will appear in the Proceedings of the 1991 Winter USENIX Conference.

1. Introduction

Tk is a new toolkit for the X11 window system [10]. Like other X11 toolkits such as Xt [1] or the Andrew toolkit [9], Tk consists of a set of C library procedures intended to simplify the task of constructing windowing applications. The Tk library procedures, like those of other toolkits, serve two general purposes: framework and convenience. First, they provide a framework that allows applications to be built out of many small interface elements called *widgets* (e.g. buttons, scrollbars, menus, etc.). The toolkit's framework makes it possible to design widgets independently, compose them into interesting applications, and re-use them in many different situations without re-design. The second purpose of the toolkit is to provide ready-made solutions for the most common needs of windowing applications. For example, Tk includes a set of commonly used widgets plus procedures to make it easy to build new widgets. Using Tk, it is possible to build many interesting windowing applications by plugging together existing widgets. Many other applications can be built by constructing one or two new widget types and combining them with Tk's existing widgets.

Although Tk's overall purpose is similar to that of other toolkits, its implementation has the unusual property that it is based around the Tcl command language. Tcl is a simple interpretive programming language designed to be embedded in applications and to work cooperatively with C code in the applications [8]. Tcl programs can be created and executed dynamically, and all of the functionality of Tk (and of Tk-based applications) is available through Tcl. This gives Tk a greater degree of flexibility, dynamic control, and power than other toolkits. For example, Tcl can be used to modify the entire widget configuration of an application at any time. New applications can be created by writing Tcl scripts for a windowing shell or for existing Tk-based applications; C code is needed only for creating new widget types or data structures.

The most important feature of Tk is that it allows different applications to work together in powerful ways. Tk provides a remote-procedure-call-like facility that allows any Tk-based application to invoke Tcl commands in any other Tk-based application. This results in more powerful communication than the traditional selection or cut buffer. Current windowing applications are forced by the lack of good communication to lump large amounts of functionality into a single application. Tk makes it possible to replace such monolithic applications with collections of smaller specialized applications that communicate with each other using Tcl commands. These smaller tools are often re-usable for other purposes, thereby resulting in more powerful windowing environments.

Tk and Tcl also simplify windowing environments by making a single run-time command language available everywhere. There is less need for application designers to invent special-purpose languages or protocols to handle particular situations: the application can just use Tcl. For example, Tcl serves as a user interface description language. It is also easier to build a new application because the application designer need only implement a few key primitive operations for the application; Tcl allows those primitives to be composed with other primitives within the application or in other applications. Tcl also simplifies things for users. Instead of learning a different command language for each application, a user need only learn Tcl. The user will then be able to program any Tk-based application merely by learning the application-specific primitives provided by that application.

The rest of the paper is organized as follows. Section 2 gives a brief overview of the Tcl language and how the Tcl interpreter is embedded in applications. Section 3 summarizes the framework Tk provides for building widgets. Section 4 describes how widgets are constructed and manipulated in Tk. Section 5 demonstrates the advantages of Tk with a few examples of user interface programming. Section 6 describes how Tk allows applications to work together by sending Tcl commands to each other. Section 7 presents the current status of Tk along with some size and performance measurements. Section 8 compares Tk to other toolkits and Section 9 concludes.

2. Summary of Tcl

Tcl stands for ‘‘tool command language.’’ My goal in developing Tcl was to make it easy to generate powerful command languages for interactive applications. Tcl is a library package written in C. It implements an interpreter for a simple programming language that provides variables, procedures, control constructs like `if` and `for`, arithmetic expressions, lists, strings, and other features. Tcl also allows applications to extend the generic command set with application-specific commands. An application need only implement a few basic Tcl commands related to the application; when these are combined with the Tcl library a fully-programmable command language results. The paragraphs below summarize a few of the key features of Tcl; see [4] and [8] for more information on Tcl and how it has been used.

```
set a 1000
print foo; print bar
```

Figure 1. Simple Tcl commands consist of fields separated by white space. The first field is a command name and the additional fields are arguments for the command. Commands are separated by semi-colons or newlines.

```
set msg "Hello, world"
set x {a b {x1 x2}}
```

Figure 2. Double-quotes or nested curly braces may be used to delimit complex arguments in Tcl commands. Each of the above commands has three fields in all. If an argument is enclosed in braces then the contents of the braces are passed to the command without any further interpretation (newlines and semi-colons are not command separators and the substitutions described in Figures 3-5 are not performed). If an argument is enclosed in quotes, then the substitutions in Figures 3-5 are performed on its contents.

```
print $msg
if $i<2 {set j 43}
```

Figure 3. Dollar signs invoke variable substitution in Tcl commands: the dollar sign and variable name will be replaced with the value of the variable in the argument passed to the command.

```
print [list q r $x]
set msg [format "x is %s" $x]
```

Figure 4. Tcl commands may contain other commands enclosed in brackets. When this occurs, the nested command is executed and its result is substituted into the argument of the enclosing command, replacing the bracketed command.

```
set msg "\{ and \[ are special"
print Hello!\n
```

Figure 5. Backslashes prevent special interpretation of characters like braces and brackets in Tcl commands. Backslashes can also be used to insert control characters into commands, as in the second command above.

The Tcl language has a simple syntax with features reminiscent of the UNIX shells, Lisp, and C. Figures 1-5 summarize the complete Tcl syntax. In their simplest form (Figure 1), Tcl commands are like shell commands: they contain one or more fields separated by white space; the first field is the name of a command and the other fields are arguments passed to the command. Unlike UNIX shell commands, Tcl commands return string values. The Tcl syntax includes additional features for specifying complex arguments, substituting variable values, and executing nested commands (see Figures 2-5).

Tcl is an *embedded language*: it is a library that is designed to be linked together with C applications as shown in Figure 6. The main loop of the application generates Tcl commands. This could happen in any of several ways, depending on the application. One way is to read commands from standard input; this results in a shell-like program. Another way, used by Tk, is to associate Tcl commands with X events such as button presses or keystrokes; when an X event occurs, the corresponding commands are executed. When the application has generated a Tcl command it passes it to a Tcl library procedure for evaluation. The Tcl interpreter parses the command, performs the substitutions described in Figures 2-5, uses the first field of the command to locate a *command procedure* for the command, and then calls the command procedure to actually execute the command. The command procedure carries out its function and returns a string result, which the Tcl interpreter returns back to the calling code in the application.

The Tcl library includes several *built-in commands* that implement the generic facilities such as variables and looping. Additional command procedures may be provided by each application. The application registers its own specific commands by passing their names and command procedures to Tcl. This information is used

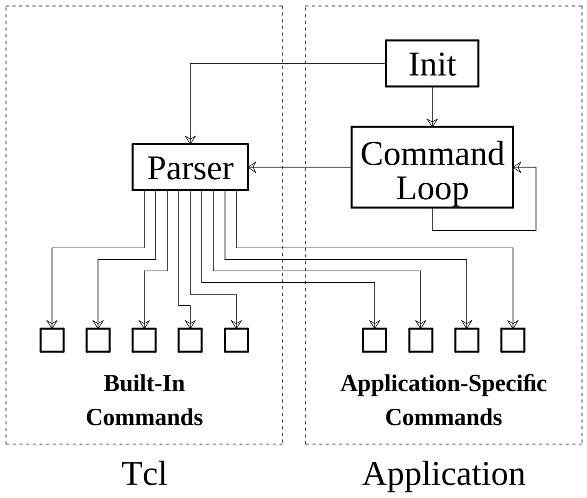

Figure 6. The Tcl interpreter is a C library package that is embedded in applications. The application generates Tcl commands and provides command procedures for application-specific commands. Tcl parses the commands and calls a command procedure to execute each command. Application-specific commands must be registered with the Tcl interpreter, usually during initialization.

later by the Tcl interpreter when it evaluates command strings. Application-specific and built-in commands have exactly the same structure; they are indistinguishable except that built-in commands are registered automatically and users may expect them to be present in all applications. New commands may be created and deleted at any time while an application executes.

Control constructs like `if` are implemented as ordinary commands that make recursive calls to the Tcl interpreter. For example, the command

```
if $i<2 {set j 43}
```

causes the command procedure for `if` to be invoked. This command procedure evaluates its first argument as an expression. If the value of the expression is non-zero, then the command procedure calls the Tcl interpreter recursively to execute the command ‘‘`set j 43`’’. It is common in Tcl-based applications for one command to take another Tcl command as argument and then execute that command, either immediately or later on.

There is only one official data type in Tcl: strings. All commands, arguments to commands, command results, and variable values are strings. Some commands expect their strings to have particular formats (e.g. arithmetic expressions or Lisp-like lists), but whenever information is passed from one place to another it is as a string. This approach makes it easy to communicate information between C procedures and Tcl programs (there are no complex data type conversions). It also means that Tcl programs have the same basic form as Tcl data, which allows new Tcl programs to be synthesized and executed on-the-fly (in this sense Tcl is similar to Lisp).

The most important aspects of Tcl are the simplicity of the language and the simplicity of its interface to C programs. The language simplicity makes Tcl easy to learn; the interface simplicity makes it easy to use Tcl in applications, easy to write new Tcl commands, and easy to use Tcl to compose primitives written in C.

3. Overview of the Tk Intrinsics

An application based on Tk is constructed by assembling a collection of user-interface components called *widgets*. A widget consists of one or more windows that display information on the screen and react to keystrokes and mouse actions. A widget may be as simple as a ‘‘button’’, which displays a text string and executes a command when a mouse button is pressed over it, or it may be as complicated as a dialog box containing sliders, buttons, text entries, and list boxes. Complex widgets may be composed out of simpler widgets.

As described in the introduction, Tk supports the creation and use of widgets by providing a standard framework in which widgets are constructed; this makes it possible for widgets to be designed and implemented independently yet still work together in interesting ways. Tk also provides a number of convenience procedures to carry out the most common operations required by widget implementors. This set of facilities (the part of the toolkit that isn’t associated with a particular widget set) is called the toolkit *intrinsics*.

In Xt and most other toolkits the intrinsics exist as a set of C library procedures. In contrast, Tk provides not only C procedures but also a collection of Tcl commands that make virtually all of the intrinsics accessible from Tcl. The Tcl interfaces allow the look and feel of an application to be queried and modified at any point in the application’s execution. They also allow new interface elements, or even new applications, to be created dynamically just by writing Tcl scripts. In these respects Tk is different from other toolkits.

The paragraphs below summarize the main facilities provided by the Tk intrinsics. Most of the facilities are similar to those provided by Xt or other toolkits; where there are differences, they exist mostly to make the Tk facilities accessible from Tcl.

3.1. Window Names

In order to refer to windows in Tcl commands, each window in Tk has a name that identifies it uniquely among all the children of the same parent window. Each window also has a class, such as `Button`, that identifies the type of widget displayed in the window. Lastly, each window has a *path name* that identifies the window uniquely within its application. A path name consists of zero or more names separated by dots. For example, the path name ‘‘`.a.b.c`’’ denotes a window `c` inside a window named `b` inside a window named `a` inside the main window of the application. The path name ‘‘`.`’’ refers to the main window of the application.

3.2. Event Dispatching

Like most toolkits, Tk provides a centralized mechanism for dispatching X events. Widgets and other interested parties inform Tk of events they care about and provide C procedures to handle the events. When an event occurs Tk invokes all the relevant handlers. The Tk dispatcher supports X events, file events (which trigger when a file becomes readable or writable), timer events, and ‘‘when-idle’’ events (which trigger when all other pending events have been processed).

Tk also provides Tcl commands for creating event bindings; in this case the events trigger the execution of Tcl commands instead of C procedures. See Figure 7 for examples.

3.3. Resource and Structure Caches

Allocating X resources such as pixel values or fonts is expensive because it requires inter-process communication with the X server. To reduce the amount of server traffic, Tk caches information about the X resources currently in use by an application. If the same resource is requested multiple times for different purposes, only the first request results in server traffic; the later requests are satisfied by sharing the existing resource. This provides a substantial boost in performance in the common case where a few resources are used in many different widgets within an application.

Tk's resource caches are indexed by textual descriptions of the resource rather than binary values (e.g. `MediumSeaGreen` might be used for a color, `coffee_mug` for a cursor, or `@star` for a bitmap stored in a file named `star`). This makes it easier to name X resources in Tcl commands or in the option database described below. In addition, given an X resource identifier, Tk will return the textual name for that resource; this feature makes it easy for widgets to provide human-readable information about their current configuration.

Tk also caches structural information about windows, such as parent-child relationships, sizes, and locations, and makes this information available to widgets so

```
bind .x <Enter> {print "hi\n"}
bind .x a {print "you typed ’a’\n"}
bind .x <Escape>q {print "you typed escape-q\n"}
bind .x <Double-Button-1> {print "mouse at %x %y\n"}
```

Figure 7. Tk provides a Tcl command called `bind`, which can be used to arrange for other Tcl commands to be executed when certain X events (or sequences of X events) occur. The four commands above cause messages to be printed on standard output when the mouse enters window `.x`, when the letter `a` is typed in `.x`, when the escape key is typed followed by the `q` key in window `.x`, or when mouse button 1 is pressed twice in rapid succession in `.x`. Before executing the command for an event Tk replaces % sequences in the command with fields from the event. For example, in the last command above the `%x` and `%y` will be replaced with the x- and y-coordinates from the X event before executing the command.

that they don’t have to fetch it from the X server.

3.4. Geometry Management

Geometry management refers to algorithms for controlling the locations and sizes of child windows within a parent, such as ‘‘all-in-a-row’’ or ‘‘all-in-a-column’’. In Tk, as in Xt, individual widgets do not control their own geometry. Instead, specialized *geometry managers* manage window arrangements. Each widget specifies a preferred size for its window (e.g. a button widget might request a size just large enough to contain the text being displayed in the button). A geometry manager then computes the actual size for each window, taking into account the requested sizes of the windows it manages, the size of the parent window, and its own particular layout algorithm (see Figure 8 for an example). Each widget must make do with whatever size it is assigned. This approach separates the internal design of a widget from its arrangement in a larger application, so that widgets can be used with a variety of geometry managers.

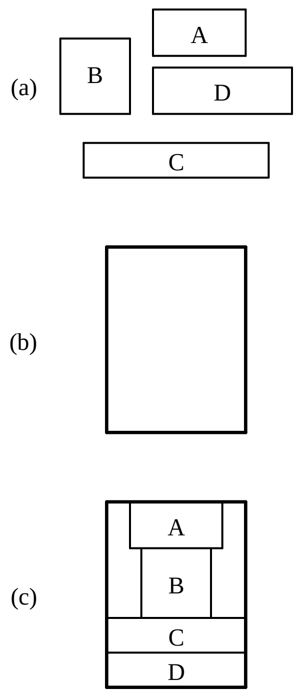

Figure 8. An example of geometry management. (a) shows the requested sizes of four windows and (b) shows the size of the parent window in which the windows are to be arranged. An ‘‘all-in-a-column’’ geometry manager might produce the layout in (c) by arranging the windows in order from top down. Window C ended up with less width than requested and window D received less height than requested because there was insufficient space in the parent. The widgets using windows A-D are expected to make do with whatever size they are assigned by the geometry manager.

Tk acts as intermediary for geometry management. It allows geometry managers to claim control over windows, and when a widget requests a particular size for its window Tk passes that information to the relevant geometry manager. Only one geometry manager manages a given window at a time.

Tcl commands are used to control the geometry managers. For example, Tk contains a built-in geometry manager called the *packer*. The command

```
pack append .x .x.a top .x.b top .x.c top
```

will cause the packer to claim control over the windows `.x.a`, `.x.b`, and `.x.c`. The packer will add those windows to the list of windows it manages inside window `.x` and arrange the windows in a column with each window placed at the top of the space not occupied by previous windows in the list. The resulting arrangement will be similar to the one shown in Figure 8. (The packer also includes a number of other features that are not evident from this one example, such as placing windows against the other sides of the parent's cavity and selectively stretching windows to fill extra space.)

3.5. Options

Tk provides a standard mechanism for users to specify their preferences about widget options such as colors and fonts. It maintains these options in a database and provides efficient mechanisms for widgets to query the database when they configure themselves. Tk's option database is the same as the *resource manager* mechanism in Xt: users specify their preferences in a `.Xdefaults` file or in a special root-window property using a simple pattern-matching language (e.g. ``*Button.background: red`' means that all button widgets should have a red background color). In addition to providing C library procedures for querying the option database, Tk also provides Tcl commands that can be used to query the database or add entries to it.

3.6. The Selection

The X11 Inter-Client Communications Conventions Manual (ICCCM) specifies a complex set of protocols that applications must use to manipulate the selection [10]. Tk provides mechanisms to implement the ICCCM protocols and hide as many of their details as possible. If a widget supports the notion of a selection, it registers a C procedure that Tk may call to retrieve the selection when it is in that widget. This procedure is called a *selection handler* and is similar in many respects to other event-handling procedures. When a widget wishes to claim the selection it calls another Tk procedure, which uses the ICCCM protocols to notify the existing selection owner that it has lost the selection. From this point on Tk will arrange for selection requests to be forwarded to the selection owner by calling its selection handler. When some other widget (potentially in another application) claims the selection, Tk will notify the current owner that it has lost the selection. Lastly, Tk provides a procedure to retrieve the selection from its current owner. Tk also provides Tcl support for the selection: selection handlers may be written in Tcl and a Tcl command is available to retrieve the selection.

3.7. Focus Management

Given that there are many windows on the screen, each of which might potentially receive keyboard input, but only one keyboard, there must be a mechanism for sharing the keyboard among the windows. At any one time keystrokes are directed to a single window, which is called the *focus window* or *input focus*. A separate *window manager* process controls the transfer of the focus among applications, but the window manager knows nothing of the internal structure of an application. Tk provides a Tcl command that can be used to assign the focus to a particular window within an application, so that all keystrokes in any window of the application are directed to the focus window. For example, when an application pops up a dialog box with a text entry, the focus may be assigned to the text entry so that the user can enter text without having to move the mouse to the dialog box; when the dialog box is complete, it can assign the focus back to the originating window again.

4. Tk Widgets

In Tk, widgets like scrollbars and buttons are implemented with C code that uses the Tk intrinsics. At runtime, Tcl commands are used to instantiate and manipulate widgets. Two different kinds of Tcl commands are used for widgets: creation commands and widget commands. For each type of widget, such as button, radiobutton, or scrollbar, there exists one Tcl command to create widgets of that type. The command's name is the same as the widget's type. For example, the command

```
button .hello -bg Red -text "Hello, world" -command "print Hello!\n"
```

will create a new button widget. The first argument gives the path name of a new window to be created for this widget. Additional arguments are used to specify options for the widget. In the example the options specify a background color to use for the widget, a string to display in the widget, and a Tcl command to execute when the button is invoked by clicking a mouse button over it. For unspecified options, the widget checks in the option database for a value; if none is found then it uses a default associated with the widget type. Once the widget has been created, a geometry manager may be invoked to position the widget in its parent and map the widget so that it is displayed.

As part of creating a widget, a new Tcl command is created whose name is the same as the path name of the widget's window (‘‘`.hello`’’ in the example above). This command is called a *widget command* and may be used to manipulate the widget. For example, the following Tcl commands could be used to manipulate the button widget created above:

```
        .hello flash
        .hello configure -bg PalePink1 -relief sunken
```

The first command causes the button to change colors back and forth a few times. The second command resets some of the widget's configuration options: it changes the background color to light pink and changes the 3-D appearance of the button so that it appears to be depressed instead of raised. The `configure` form is supported by all widget commands and allows any configuration option of any widget to be changed at any time in the same way that it may be specified when creating the widget.

Most widgets are *active*: they carry out some function when manipulated with the mouse and/or keyboard in a particular way. For example, if a mouse button is clicked over a button widget or menu widget, some action will be invoked in the application; a mouse click over a scrollbar causes the view to change in some associated widget, and so on. In Tk widgets, all of these actions are specified as Tcl commands. In the button example above, the command was specified as ``print Hello!\n``. When the button is invoked the widget's C code invokes the Tcl interpreter to execute the command, which just prints a message on standard output.

In some cases the widget will augment the user-supplied command with additional information. For example, consider the case of a listbox with an associated scrollbar. When the user clicks on the scrollbar, the scrollbar must notify the listbox that it should adjust its view. It does this by issuing a Tcl command. As part of creating the scrollbar widget the application designer specifies the first part of the command. For example, if the listbox is in a window named `.list` then the command will be specified as ``.list view`` to invoke the widget command for the listbox. Before executing the command, the scrollbar adds an additional number to it, producing a command like ``.list view 40``; this command requests that the listbox adjust its view so that item 40 appears at the top of its window.

The use of Tcl commands for all widget actions provides both flexibility and power. In the scrollbar example of the previous paragraph it allowed two independent widgets, a listbox and a scrollbar, to be connected so that they work together. In the most general case a user or application designer could write an arbitrary Tcl procedure and specify that procedure as the command for a widget. In this way, for example, a single scrollbar could be made to control several windows.

5. Programming Within An Application

For many users I expect the Tcl language to be invisible: users will manipulate applications using the keyboard and mouse and be unaware of the fact that an interpretive language underlies the user interface. However, advanced users and application designers can use Tcl to gain power and flexibility. For example, a Tcl command could be invoked to add a new keystroke binding to an existing widget (e.g. backspace over a whole word when Control-w is typed in an entry widget). Such a command could be typed to a running application (if the application provides a command type-in window) or placed in a startup file to be read automatically whenever the application is executed. The application itself would not have to be modified in any way to support the new binding — as long as the entry widget allows its contents to be fetched and modified from Tcl, it will be possible to implement the backspace-over-word operation using a Tcl command or command procedure.

In addition to all the other purposes it serves, Tcl also serves as a user interface description language; there is no need to design a special user interface language, and Tcl's general programming constructs provide quite a bit of power in creating and modifying interfaces. For example, Tcl can be used to modify the arrangement of windows within an application, e.g. put the diagnostic message window at the top of the application rather than the bottom, or change the order of menus in the pull-down menu bar. Tcl can even be used to create entirely new interface elements such as dialog boxes while an application is running. In fact, Tk contains no special support for dialog boxes. The basic commands for creating and arranging widgets are already sufficient to create dialog boxes: even in the normal case, dialogs are created by writing short Tcl scripts.

Tcl can also be used to create new applications without writing any C code. For example, I have built a simple windowing shell called `wish`, which consists of Tcl, Tk, and a main program that reads Tcl commands from standard input or from a file.

```
1   #!wish -f

2   scrollbar .scroll -command ".list view"
3   listbox .list -scroll ".scroll set" -relief raised -geometry 20x20
4   pack append . .scroll {right filly} .list {left expand fill}

5   proc browse {dir file} {
6       if {[string compare $dir "."] != 0} {set file $dir/$file}
7       if [file $file isdirectory] {
8           set cmd [list exec sh -c "browse $file &"]
9           eval $cmd
10      } else {
11          if [file $file isfile] {exec mx $file} else {
12              print "$file isn’t a directory or regular file\n"
13          }
14      }
15  }

16  if $argc>0 {set dir [index $argv 0]} else {set dir "."}
17  foreach i [exec ls -a $dir] {
18      .list insert end $i
19  }

20  bind .list <space> {foreach i [selection get] {browse $dir $i}}
21  bind .list <Control-q> {destroy .}
```

Figure 9. A simple directory browser, implemented as a script for `wish`, the windowing shell. This script is stored in a file named `browse` (without the line numbers). Line 1 is a comment line; when the file is executed, it causes `wish` to be invoked as command interpreter for the file. Lines 2-4 create a scrollbar and a listbox, arrange for the scrollbar to control the view in the list box as described in Section 4, and place them side-by-side in the application’s main window. Lines 5-15 create a procedure `browse`, which is invoked to browse subdirectories (by running another version of the browser) or files (by running an editor called `mx`). Lines 16-19 initialize the listbox to hold the contents of a particular directory. Lines 20-21 create bindings to invoke the `browse` procedure when space is typed, or to exit when Control-q is typed.

Entire windowing applications can be written as scripts for `wish`, just as UNIX commands can be written as scripts for `sh` or `csh`. For example, a simple directory browser can be written as a 21-line `wish` script (see Figures 9 and 10). I plan to enhance `wish` with drawing commands for shapes and text and a few other features; once this is done it will be possible to code a large class of interesting applications entirely in Tcl.

6. Programming Between Applications

In spite of the claims of the previous sections, Tk's greatest benefit of all is not within an application but rather the way it allows different applications to work cooperatively. Currently, the only widely-available communication mechanism between applications is a selection or cut buffer: the user selects information in one application, then invokes a command in another application to retrieve the selection and use it in some way. Besides being limited as a form of communication, this approach is also tedious since the user must take some action for each transfer of the

Paste browse.ps here.

Figure 10. A screen dump showing the appearance of the browser produced by the script in Figure 9. The three darkened items are selected. The window's title bar was generated by the `twm` window manager.

selection.

Given such weak communication, application implementors tend to lump functions together into large monolithic applications. This occurs even when the functions are mostly independent, just so that the functions can communicate in ways other than the selection. For example, many debuggers contain built-in editors so that they can display source code and highlight the current line of execution. Commercial spreadsheet programs tend to be lumped together with chart packages, databases, word processors, and communication packages in order to allow the different functions to work together. The lumping results in unnecessary re-implementation of functions: each spreadsheet contains its own chart package, each debugger its own editor, and so on.

Tk solves the problem of poor communication with a Tcl command called `send`. `Send` takes two arguments: the name of an application and a Tcl command. Each Tk-based application has a unique name, and information about all existing applications is registered in a special property on the root window of the display. When `send` is invoked, Tk locates the target application by reading the registry property. Then Tk forwards the command to the target application (using other window properties). The Tk of the target application executes the command and returns the result of the command back to the originating application. This allows any Tk-based application to control any other Tk-based application on the same display. Any command that could be invoked within an application may be invoked by other applications using `send`, including commands to manipulate the application's interface and also commands to manipulate the application itself.

`Send` is a form of remote procedure call [2]; as such it provides a more general and powerful form of communication than the selection. For example, Tk-based debuggers and editors can be built as separate programs. The debugger can send commands to the editor to highlight the current line of execution, and the editor can send commands to the debugger to print the contents of a selected variable or set a breakpoint at a selected line. A Tk-based spreadsheet might permit cells to contain embedded Tcl commands. When such a cell is evaluated the Tcl command would be executed automatically; it could fetch information from an independent database package or from any other program in the environment. A Tk-based word processor might permit embedded Tcl commands in the body of a document. When the document is formatted, the Tcl commands would be executed; they could retrieve information from spreadsheets, databases, or drawings.

Interface editing provides another example of the power of `send`. Existing interface editors generally operate on application mock-ups. The editor displays something that looks like an application and allows its interface to be edited, but the thing being edited isn't the actual application, so it isn't possible to try out the interface under ''real-life'' conditions. The interface editor produces an interface description file, which must then be compiled and linked with the application before it can actually be tested. With Tk and `send` it becomes possible for an interface editor to work on live applications, using `send` to query and modify the application's interface. The effects of interface changes can be tested immediately with the application.

When a satisfactory interface has been created, the interface editor can produce a Tcl command file for the application to read at startup time to configure its interface in the future.

The overall effect of `send` is that it makes it possible to program applications to work together in powerful ways, so it will no longer be necessary to lump functions into monolithic applications. This encourages the development of lots of small specialized tools that can be programmed with `send` to work together in interesting ways. The tools could be developed and maintained independently, yet be used in many different ways. I believe that this could result in much richer and more powerful interactive environments than we have today.

The combination of Tcl and Tk and `send` also allows hypertext and other kinds of active objects to be implemented easily. All that an individual application needs to do is to allow Tcl command strings to be embedded in its internal structures and provide a mechanism for invoking those commands at ‘‘interesting’’ times. Tcl commands can then be written to extend and enhance the behavior of objects. For example, in the spreadsheet envisioned above, commands may be stored in spreadsheet cells; they will be executed whenever the spreadsheet is evaluated. The embedded Tcl commands allow the spreadsheet to ‘‘reach out’’ and retrieve fresh data values from databases or other applications. Or, a hypertext system can be implemented by associating Tcl commands with pieces of text or graphics in an editor; when a mouse button is clicked over an item then the associated commands are executed. A hypertext ‘‘link’’ can be produced by writing a Tcl command that opens a new view and associating that command with some piece of text or graphics. A hypermedia link can be produced using a Tcl command that sends a ‘‘play’’ command to an audio or video application.

7. Status and Measurements

Development of Tcl began in early 1988, and it has been distributed publicly since 1989. The Tcl distribution does not include Tk or any other windowing support. Based on mail I have received about Tcl, I estimate that about 50 Tcl-based applications exist or are under construction.

I began implementing Tk in late 1989. At present the intrinsics are complete, although they are evolving rapidly as I gain experience using them to implement widgets. I have built a number of Motif-compatible widgets, including panes, labels, buttons, check buttons, radio buttons, messages, listboxes, scrollbars, and scales. Two major widget types, entries and menus, are still left to be implemented (I hope to complete both of these before this paper is published). I expect to begin distributing Tk in early 1991. As with Tcl, the code will be freely distributed without any licensing restrictions.

Table I shows the sizes of Tk and Tcl in lines of code and in compiled bytes, and compares them to the sizes of corresponding portions of the Xt toolkit and the Motif widget set. Tk and Tcl together have only three-quarters the compiled size of Xt, even though they provide more flexibility and power. Tk’s widgets and geometry manager are 2-5x smaller than the corresponding Motif modules. As Tk's widgets mature I expect them to grow slightly, but I believe that their final sizes will still be substantially smaller than the Motif widgets.

Tcl simplified Tk and its widgets by making a single unifying language available everywhere in the system. Tk implements only a few key primitives, which can then be composed with Tcl. In systems without a composition language, such as Xt/Motif, all run-time needs must be predicted and addressed explicitly in the C code; this increases the amount of code that must be written. In addition, the lack of a single unifying language resulted in many different protocols and ''little languages'' to handle different situations in Xt and Motif (examples are the ICCCM selection protocols, the Xt translation manager, and Motif's UIL interface description language). These additional protocols add to the complexity of the system.

Table II gives a few sample performance numbers for the Tk toolkit. On a machine with 10 MIPS or more, the Tcl interpreter is fast enough to execute many hundreds of Tcl commands within a human response time; this permits relatively lengthy Tcl scripts to be executed without noticeable delays. The `send` command currently takes a few tens of milliseconds. At this speed, it is possible to paint with the mouse in one application, have all the mouse motion events bound into Tcl commands, which in turn use `send` to forward commands to another application in a different process, which finally draws the painted object in its own window, and have all of this take place with no noticeable time lag. Tk is fast enough to instantiate relatively complex applications (many tens of widgets) in a fraction of a second. Tk has not undergone any performance tuning yet; when it does there should be some improvement in these numbers.

	Source Lines		DS3100 bytes	
	Xt/Motif	Tk	Xt/Motif	Tk
Intrinsics	24900	15100	216400	92800
Tcl		9300		61100
Geometry Manager	2100	1000	17100	7400
Buttons	6300	1000	43700	8600
Scrollbar	3000	1200	24900	8000
Listbox	6400	1600	53100	10700
Total	42700	29200	355200	188600

Table I. A comparison between Tk and Xt/Motif based on lines of source code and bytes of compiled object code (for the DECstation 3100) for selected modules. ''Geometry Manager'' refers to the PanedW module in Motif and the ''packer'' geometry manager in Tk; the packer is somewhat more general and flexible than PanedW. ''Buttons'' consists of three files in Motif (Label, PushB, and Toggle); in Tk a single file implements labels, buttons, check buttons, and radio buttons. The totals reflect only the modules in the tables; Both Tk and Motif contain additional widgets not reflected in the table.

Operation	Time
Simple Tcl command (`set a 1`)	68 µs
Send empty command	15 ms
Create, display, delete 50 buttons	440 ms

Table II. Execution times for selected operations in Tk. All times were measured on a DECstation 3100 running Ultrix 4.2 and X11R4. In the bottom measurement of the table, about half of the elapsed time was spent executing in the client and about half in the X server.

8. Comparisons

Of the existing X11 toolkits, Tk is most similar to Xt [1]. The major facilities provided by Tk were inspired by Xt and are similar to the corresponding facilities of Xt. There are also similarities between Tk and the InterViews and Andrew toolkits [5,9] in that all support some sort of widget-like notion to decompose applications. However, InterViews and Andrew have more support for the underlying application object structures whereas both Tk and Xt focus almost exclusively on the interface aspects, with little support for the application structures.

The most significant difference between Tk and the other toolkits is the presence of Tcl in Tk. Run-time languages are starting to appear in other systems, such as Ness, which is used to embed executable programs into documents in the Andrew toolkit [3], and UIL, which is used to specify interfaces in Motif [7]. However, these languages have three disadvantages relative to Tcl. First, they are less dynamic. For example, UIL programs must be compiled before being processed by a running application, and Ness appears to require many decisions to be made statically. In contrast, Tcl is interpretive, so any available operation can be invoked at any time. Second, the other languages are less complete. For example, UIL does not include control constructs such as `if` and `while`, and Ness functions are not first-class objects. In contrast, Tcl is a complete programming language that even provides access to its own internals (e.g. it is possible to retrieve the body of a Tcl procedure or a list of all defined variable names). Third, the other languages are special-purpose: they only control a portion of an application’s functions. In contrast, Tcl is used for virtually all aspects of an application, which makes it possible to compose all of those aspects to work together.

Another difference between Tcl and other toolkits is the `send` command for inter-application communication. I know of no equivalent construct in other X toolkits. The closest existing facility is Microsoft Windows’ Dynamic Data Exchange protocol (DDE), which allows applications to communicate in several ways including passing commands for remote execution [6]. However, for remote execution to be most useful it must allow access to all the internals of the remote application. For this to happen, the language used by the remote execution facility should be the same as the language used to control the user interface and internals of the target application, as it is with Tcl and Tk. Unfortunately, the Windows environment does not include a universal command language. Although a standard syntax is suggested for remote commands, there is no built-in connection between these remote commands and the internals of the remote application. Each application must provide special code to parse and execute all the remote commands it wishes to support. This will probably limit the use of remote execution in DDE to a small set of functions. In contrast, Tk's `send` command provides access to all aspects of other Tk-basd applications without any extra effort on the part of the applications' developers.

One final difference between Tk and other toolkits is object orientation. InterViews, Xt, and Andrew are all strongly object-oriented with support for classes and inheritance. In contrast, Tk is not strongly object-oriented. The widget commands described in Section 4 give Tk an object-like feel, and Tk makes extensive use of procedure variables and callbacks, but there is no official class mechanism and no inheritance among widget types. Instead of providing inheritance, Tk focuses on *composition*: mechanisms for assembling independent widgets into interesting arrangements. In my opinion, composition is more important for a toolkit than inheritance. There isn't enough commonality between widgets for inheritance to provide much benefit, and inheritance adds complexity (to understand one widget you must understand all the widgets it inherits from). Inheritance mechanisms only benefit a small group of people (widget implementors), whereas composition mechanisms allow any user to create new interface elements out of existing widgets. Further support for this view comes from the InterViews system: although it is written in C++ and claims to be object-oriented, the primary benefit claimed for the system is its support for composition [5].

9. Conclusions

I believe that Tk provides a large increase in power and flexibility over existing windowing toolkits. Tk's power comes from two sources: the power of programming and the power of building interchangeable tools. The use of Tcl within Tk (and within Tk-based applications) means that a single programming language is available at run-time to control all aspects of an interactive application, from its look to its feel to its function. This in turn makes it possible to modify and extend all of these aspects of an application at any time. The second source of power is from composition: the ability to build independent units that can work together and be re-used in many different ways unforeseen by their designers. Tcl acts as a composition language both for composing widgets within an application and for making different applications work together.

I hope that Tcl and Tk can do for interactive applications of the 1990's what the UNIX shells did for stream-based applications of the 1970's. The UNIX shells encouraged the construction of small tools that read from standard input, perform some operation on the data, and write the results to standard output. The shells provided mechanisms for these ‘‘filters’’ to be hooked together in many different ways to perform interesting functions. I hope that Tcl and Tk will encourage the development of many small specialized windowing tools that present simple Tcl interfaces. Tk permits the tools to work together by sending commands to each other. With this approach I hope it will become possible to build more powerful interactive applications with much less effort than is needed today.

10. Acknowledgments

The work described here was supported in part by NASA and the Defense Advanced Research Projects Agency under Contract No. NAG2-591, and in part by Digital Equipment Corporation. Joel Bartlett, Mike Kupfer, Joel McCormack, and Ken Shirriff provided numerous suggestions that improved the presentation of the paper. The design of Tk benefited from discussions with Joel Bartlett and Joel McCormack; I am grateful for their stimulating comments and criticisms.

11. References

[1] Asente, P. and Swick, R., with McCormack, J. *X Window System Toolkit: The Complete Programmer's Guide and Specification*. Digital Press, 1990.

[2] Birrell, A. and Nelson, B. ''Implementing Remote Procedure Calls.'' *ACM Transactions on Computer Systems,* Vol. 2, No. 1, February 1986, pp. 39-59.

[3] Hansen, W. ''Enhancing Documents With Embedded Programs: How Ness Extends Insets in the Andrew Toolkit.'' *Proc. 1990 International Conference on Computer Languages*, March 1990.

[4] Libes, D. ''expect: Curing Those Uncontrollable Fits of Interaction.'' *Proc. USENIX Summer Conference*, June 1990, pp. 11-15.

[5] Linton, M., Vlissides, J., and Calder, P. ''Composing User Interfaces with InterViews.'' *IEEE Computer*, Vol. 22, No. 2, February 1989, pp. 8-22.

[6] *Microsoft Windows Software Development Kit, Guide To Programming, Version 3.0*. Microsoft Corporation, 1990.

[7] *OSF/Motif Programmer's Guide, Revision 1.0.* Prentice Hall, Englewood Cliffs, NJ, 1990.

[8] Ousterhout, J. ''Tcl: An Embeddable Command Language.'' *Proc. USENIX Winter Conference*, January 1990, pp. 133-146.

[9] Palay, A., et al. ''The Andrew Toolkit – An Overview.'' *Proc. USENIX Winter Conference*, February 1988, pp. 9-21.

[10] Scheifler, R., and Gettys, J., with Flowers, J., Newman, R., and Rosenthal, D. *X Window System: The Complete Guide to Xlib, X Protocol, ICCCM, XLFD* (Second Edition). Digital Press, 1990.