Visual Tcl Paper - Building a Distributed MultiPersonality GUI toolkit for tcl

Mike Hopkirk
Santa Cruz Operation Inc

Visual Tcl

Building a Distributed MultiPersonality GUI toolkit for tcl

Mike Hopkirk

Santa Cruz Operation Inc

hops@sco.com

http://www.sco.com/Products/vtcl

Abstract

This paper details some of the Requirements, Design, Architecture, History and Lessons learnt in implementing Visual Tcl (Vtcl) - A distributed multiple GUI toolkit extension to Tcl oriented initially and mainly at generating User Interfaces for System Administration use.

Why Another GUI for Tcl?

Possibly the first question asked is why would we would even consider another tcl GUI extension given that Tk exists. These were some of our requirements:

- Need for a 'pure' Motif based interface - For better or worse all our existing GUI applications are Motif Toolkit based - our GUI toolkit has to be based on the Motif Toolkit, not a look alike or work alike or 'compatible' toolkit.
- Need a Character interface, We still must continue to support character consoles and terminals.
- Possibility of other Interfaces later. The GUI interface du jour changes. We want to support other interfaces in the future without having to reimplement the entire package. The most recent example of this is supporting MSWindows
- We want the above without supporting multiple Code bases.
- Higher level of abstraction - embed more policy, Simplify UI decisions. We don't want to burden developers with implementing their own (possibly variant) styles of dialogs components and we want to provide an environment where the developer can concentrate on the details of the UI rather than Toolkit interaction trivia or details of the sub components from which the UI is composed. (This and the character requirement precludes use of solutions such as TclMotif and Wafe[4]).
- Simple hooks into the native help system. We have the model for a help system that we think is pretty good and needed to provide hooks into it (its both topic and widget tree based). We want to provide something that may be able to plug into the native help system on other platforms.
- Easily learnt, easy to develop, Easily extendable and customisable. We get some of this from using Tcl.
- When this was started there was no complete GUI system for Tcl or scripting system for X and Motif.

History

1990 Decided we needed a Tool for building Administration GUI's. This came from a requirement to improve the system administration capabilities of our systems and the decisions that the right people to do that were the sysAdmin engineers and that it wasn't reasonable for them to learn Motif/X and Curses programming to the level required.

The original system was built around the Bourne shell, Forms based with very high level of policy enforcement. The programming model was procedural rather than event driven using a blocking 'allow Form filling here' style.

For a number of reasons this turned out to be rather a hard sell. It was hard to organize programs in any reasonable away around the user input state and the design didn't provide the control and flexibility the screen designers felt they needed.

1991 Switched to Tcl base. We also moved to a GUI model that was more component oriented with much more scripting flexibility and changed the main Form controls to use an event driven paradigm.

This was wildly successful, Use spread like a weed both within the SystemAdmin GUI project and for such things as Installation, application configuration, small user productivity apps and one off GUI needs.

Here was also a period of feature growth vigorously driven by user requests. The base started off as a prototype that was well received and accreted features and functionality.

1993 (July - March 94) The previous accreted prototype was becoming difficult to maintain so we rewrote it. The major user visible modification here was to accommodate a more Tk-like widget naming scheme. Architecturally we created something of a component or trait inheritance architecture that allowed us to remove significant code duplication and rationalize a lot of the option handling.

We also added a few more things that had been requested that were difficult to accommodate on the previous code base - support configuring widgets after widget creation, More flexibility and control over displaying and hiding forms and dialogs, Allow more explicit control over widget positioning, Make all dialogs non Blocking.

Since there was already an existing working script code base some of our users were not keen on this change. We managed to convince them to take the first steps through a combination of bribery, hand holding, automation aids and threats. Fortunately once they'd started using the rewritten system they became wedded to it...

1994 Mainly a period of consolidation. We put a lot of effort into the Character support library and character interface to increase capability and smooth out scripting differences from the Graphical interface.

May 1995 - Deployment of shipping product using Vtcl. ScoAdmin, Custom and (earlier in late '94) DCE Administration. Result:

~68000 lines ScoAdmin
~41000 lines Custom
(Installation)
~40000 lines DCE admin

End user response seems to have been everything we could hope for - The GUIS themselves have been well received and there has been no negative feedback about it being implemented using Vtcl (most people haven't noticed or don't care which was the point of the exercise).

Feature Set

Vtcl exposes (in some form) probably about 70% of the Motif Widget set to the scripting interface (exceptions are the Drawing Area Widgets, MainWindow and some of the separate Managers e.g Paned Window). This exposure is generally at a level higher than that of the Motif API and we don't give direct access to any underlying Support APIs like X or Xt (as opposed to interfaces like tclMotif, Wafe [4] and DtKsh) favoring instead more a component only model.

For example we have a formDialog that encapsulates the notions of a TopLevel Shell, a Form (with associated geometry on its children) and capability for providing a set of default buttons. These conform to the style and policies defined by their style Guides (Motif [2]) or where these didn't exist from Human Factors Feedback.

Geometry management is presented in a manner similar to X/Motif with parent widgets giving behavioral constraints on their children. We give a very much simplified interface to the geometry control of the Motif Manager widgets (currently only Form and RowColumn). Most of the apps use Form exclusively.

The form layout simplification is setup such that with no explicit layout directions a 'simple and reasonable' layout is generated, explicit geometry options provide both generic and direct script control, overriding or augmenting the default layout handling.

In addition to the Motif Primitive and Manager Widgets we currently add three new ones, A ComboBox, A SpinButton (implemented in Vtcl) and a DrawnList. The DrawnList is heavily used, it presents a multicolumn list interface which can contain embedded pictures - this is convenient for displaying pictorial hierarchical trees and visually tagging related items.

We abstract the notion of colors and fonts by a single level of indirection. The script specifies a purpose oriented color or font (e.g. foregroundColor, urgentColor, titleFont) which the server is responsible for mapping onto something that makes sense in that GUI environment.

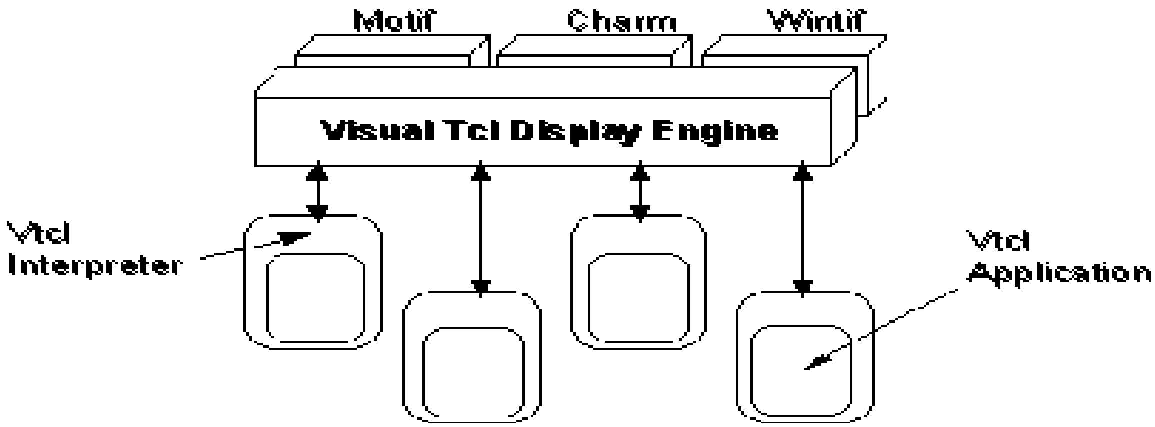
We also extended the Motif notion of displaying bitmaps to support XPM format multicolor pixmaps for those widgets that support pixmaps.

This obviously doesn't work in the character GUI where these are ignored though the script developer may provide an equivalent handling or mapping for that case.

Over and above the widgets we also provide a point help capability and support access to the platform Help system (optimized toward ScoHelp) both explicitly as a script command or implicitly from help Keys, a Help Menu or from Help buttons. This is decoupled from the scripting system.

On the interpreter side the extension supports an event driven state, callbacks firing as a result of a timeout or when nothing else is happening or when an I/O source or sink comes ready.

Architecture



Architecture

Vtcl is a client-server application. We have the notion of a GUI engine or Widget Server engine running autonomously serving out requested user interface applet interfaces as requested by a user. Multiple clients (usually Vtcl interpreters) can connect to and synchronously communicate with the server in order to get a GUI created and rendered.

User Interaction is fed back to the (interpreter) clients at ’interesting’ points as messages that eventually fire script callbacks. Additional data is passed to these callbacks providing useful information associated with the callback.

In this model variant GUI's are easily supported by changing the GUI engine that the clients connect to.

The Server keeps its own notion of the separate applet clients on a per connection basis. It does some initial applet hierarchy setup on connection, cleanup on client closedown or on losing the client connection and otherwise responds to client commands by creating widgets or modifying widget or application hierarchy state.

The server can be thought of as just a rather dumb high level display device, It does not currently contain an embedded interpreter. This is largely a legacy decision and while we may possibly simplify the server implementation we believe we would lose (or face providing server interpreter hooks which would cause loss of) a lot of the current scripting simplicity by doing so.

Client server communication is done using a simple non textual protocol that is basically just a breakdown of the command and options (to the server) and a text string or array of text strings in reply (Callback, errors and command acknowledgment or result).

As an implementation detail this was originally over named pipes, We now have the capability of supporting both that and remote socket connections.

Scripting Interface

Vtcl is obviously based on a core tcl interpreter [1] - this has been extended with two extra packages:

TclX [3] which gives us access to POSIX system interfaces, Debugging and development commands, Message Catalogs, Keyed lists and extra File String and List commands

and the

Vtcl extension itself which are just the commands that provide the graphical interface.

On top of this we provide a package library of convenience routines that encapsulate some common functionality, provide a yet higher level interface to some of the widgets and do commonly required things.

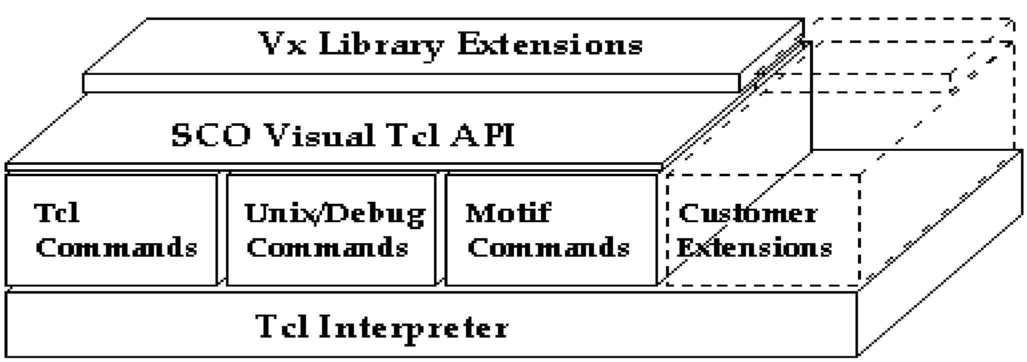
In many cases the applications using Vtcl also provide their own package libraries that encapsulate Vtcl components to give their own application specific functionality.

For the System administration functions the interpreter itself is augmented with additional commands to provide access to our System administration Framework.

With regard to Vtcl alone, GUI manipulation is connection oriented: GUI creation, manipulation and interaction happens within the bounds of a VtOpen and VtClose command (or the exit of the interpreter). There are no concurrent connections from a single interpreter though multiple sequential connections are possible

There is a single command for creation of each type of Component and two additional commands for generic access and setting of component state. Some components have additional commands for accessing or changing that specific components state.

An Extensible Interpreter



Scripting Example

Heres a script example to give some idea of the 'flavor'

```
#!/u1/hops/bin/vtcl
set app [ VtOpen Example ]                  ;# Open Server connection

# Make a dialog with single default controlling button
set f [ VtFormDialog $app.f  -title "Tcl/Tk WkShop eg" \
            -okCallback myQuitCB  -okLabel Exit \
    ]
# declare the callback proc for the 'interesting' action done on the combo box
proc myQuitCB cbs {
        global combo
       echo [VtGetValues $combo -value]                    ;# same info avail in cbs structure
       VtClose                    ;# Close connection - shutdown application
       exit                       ;# exit script
}

# Create a ComboBox in the dialog - list contains 3 items
set combo [ VtComboBox $f.combo -itemList { "banana" "plum" "apple" } ]
VtShow $f         ;# Display the Dialog for the first time
 VtMainLoop      ;# Accept events
```

Looks like this when displayed using the X/Motif Display engine

Heres the same script displayed using the character engine

Interaction and callbacks

Interaction is 'event' driven from the point a script enters the Mainloop, prior to that the script usually describes the widget hierarchy and state, sets up callback actions and makes the hierarchy visible.

Tcl Commands (usually procs) are registered on the widgets for feedback to the application of interesting actions - These are not presented directly as user events though in most cases thats where they come from.

All widgets that allow callbacks support a "-callback" option which is the callback for the 'normal' or 'usual' interesting action of that widget. e.g button pressed, list element selected. There are other callback options depending on the widgetClass - these either allow access to a variant action or to somewhat lower level of interaction.

There are also a number of additional callbacks that are not directly widget or user interaction related which allow added control over the hooks into the help system and trapping of errors in callback commands.

All callbacks are passed a TclX keyed list containing information pertaining to the callback: Common items are usually

- **dialog** - dialogName the callback occurred in
- **widget** - the widget the callback occurred on
- **value** - The 'value' of the widget when the callback fired

Additional or variant items may also exist depending on the callback

- **mode** - a hint of the presumed user intention that fired the callback e.g. done, changed, select, selectSame.
- **itemPosition** - positions of selected items (list)

Widget Naming

Widget naming is hierarchical with each level dot separated. A widget name includes its entire parent hierarchy

```
e.g. .app.form.btn
```

and if its required to be referenced later should be unique. The server enforces this though it can be told to allow a name as a duplicate of an existing widget. Each Widget Command returns the name of the widget created given the new widgets parent and desired name. The server may add its own (internal) widgets into the hierarchy as part of a widget command (this is an irritating Motif-ism we haven't attempted to hide and is most noticeable for Motif Dialogs) which means that the name you give a widget may not always be the name you end up with.

Vtcl Commands

GUI Commands are action-object oriented and generally of the form :

```
Command objectName \
        [-option [arg] ... ]
```

We've tried to provide a high level interface to components with a lot of functionality that the options trigger. There are no Key or event binding commands for example - we rely on the bindings of the native GUI component and try and present the widgets as a 'black box' with actions at a functional rather than user level.

The commands breakout into 20 Widget Commands (2 Managers), 21 Widget Manipulation commands, 3 Connection Oriented, 7 miscellaneous and 7 Interpreter only.

Widget Options

Widget options are used to trigger or describe widget actions or state (You could think of the Widget as a Class and the options as setting attributes and notification methods on that class though we make no effort to present it in those terms).

The options are categorized into several groups depending on their argument types. Options are recognized and filtered, and option arguments are validated for syntax by the interpreter, Semantics are recognized (or ignored) by the server. This implies unrecognized options are discovered early without server interaction, misplaced options rely on the server to handle.

The options (and widget capability) are presented as a Class hierarchy from a base OBJECT class supporting common options with metaclasses for DIALOG, FORM, ROWCOLUMN, LIST, LABEL and BUTTON.

Form geometry management capabilities are presented as an inheritance of the GEOMETRY metaclass on the forms children. These specify where the children are positioned in the form relative to the form and their siblings.

Convenience routines ('Vx' routines)

These are tcl procs in a package library and provide the following:

1. Another layer of abstraction over some Vtcl functionality. E.g. Menu/OptionMenu - provides specification of a menu hierarchy from a simple data structure
2. Added capability or special functionality:

   e.g. Aligned labelled widgets, Centering and aligning widgets in Forms, SpinButton Widget
3. Common/Useful functions:

   Widgets 'basename', ExitForm/ QuitApplication callbacks, Association of variables with widgets. Keyed list display, Static variables.

Some Other Issues

CHARM - the Character Support Library

As a result of an earlier project we had the beginnings of an X/Xt/Motif compatible library called CHARM (Character Motif) that mapped most of Motif and a subset of X/Xt onto a Curses interface. This library also mapped keyboard input and redraw processing into an Xlib like model.

We managed to avoid a character specific version of the server code by using this library and writing the server to a single source code base as a normal X/Motif application then addressing any differences as they arose.

These mainly entailed augmenting the CHARM library and headers where necessary to provide a more compatible Motif/X API and special casing any places where we couldn’t map or hide any differences in the two interfaces.

There is a single code base for both Character and Motif Servers of which about 5% is special case code for either of the two environments. The majority of this is additional code for utilizing graphical capability not applicable to a character environment.

Determining a Unique Server

The graphical server invocation is unique per uid/LANG/ DISPLAY triplet. This server persists until explicitly shutdown or the user logs out.

The character server invocation is unique per uid/LANG/ tty - its difficult to get coarser granularity than this without building some form of tty redirection capability onto curses. This gives Vtcl programs spawned from each other (on the same tty) use of the same server.

Startup

On VtOpen the interpreter attempts to detect its corresponding Server for the environment its running in and GUI desired (by testing a "well known point" - existence of a named pipe or success in binding to a socket). If a Server is detected the interpreter sends an Open message to the detection point.

The server, once it is up and running, listens on a 'well known point' for connections. On a connection the server saves some internal state (communication file descriptors) and creates a toplevel shell for the application which is returned to the client (interpreter).

If the server is not detected the interpreter forks and execs it (for the local case). There is an extra level of handshaking here while the server opens a Display connection, After it has done so it sends an ack (or fail) message to the client and continues on with the normal applet open sequence.

So far we've punted on the remote execution server startup and assumed its already running. A better solution would be some sort of host local registry and execution server running from inetd or equivalent that records what servers it may have started on that host and passes back their connection information or starts ones that don't currently exist...

Applet shutdown

A client close message or losing the client connection (assumed due to the client exiting) causes destruction of the applet topLevel shell. All of the applet connection state is associated with this shell so cleanup is relatively simple and done with the Shell Destroy Callback handler.

Server MultiThreading

There isn't any. Apparent multiple applet threading is done using event handlers on the file descriptor for the (server) input connection. The event handler parses and services the request and the result is sent back to the client interpreter. There is no applet state shared between applets beyond that embedded in the use of the Motif and X libraries.

Interpreter interface

Apart from seven interpreter local commands (which we can now lose since they're now available in tcl7.5) most of the Vtcl commands messages are implemented as a stub that does nothing more than send a message to the server, wait for a reply (status and optional string) and passes that back to the interpreter.

The Close and DestroyWidget commands require some additional handling to clean up any pending callbacks for those widgets that may not have been serviced yet. FileSelectionBox also has additional handling to set the server side applets notion of the current working directory to the same as that of the interpreter.

The major effort of the command stub is to parse the command (this is driven though a table of commands and a table of options) and marshall command and options data to (or from) a form conformant with the wire protocol.

Callbacks are implemented as an asynchronous message from the server containing a string (the argument to the widgets 'callback' option) and some data detailing the callback information. The callback data is assembled into a TclX keyed list and appended to the callback string. This string is then evaluated in the interpreter at global scope. Any errors in the evaluation result in a search for a registered error callback and its evaluation with some information about the error or, if none is found, return of an error status to the interpreter which causes an error return from the MainLoop command.

Callbacks received while a command is waiting for a synchronous reply are queued for execution the next time the Mainloop code is executed.

All interpreter side user interaction is serviced from the VtMainLoop command which is essentially a select call inside a loop. It fires any pending queued callbacks, waits for incoming fd activity and invokes the callback, handles signals and connection errors or fires workproc callbacks.

Communications protocol

This is (for historical reasons) a little gnarly - since its integral to the entire system it also has a lot of inertia which impedes changing it.

We use a mixed binary and text protocol organized for the convenience of the Server in breaking out commands, options and arguments. Each protocol message is typed (COMMAND, COMMAND_RETURN, CALLBACK, ERROR) and contains a field count and number of fields count. Each field is also typed and possibly counted (arrays). Field types correspond almost directly to the allowed option types - INTEGER, STRING, STRING_ARRAY, STRING_2D_ARRAY, INTEGER_ARRAY, OPTION (Special sort of string).

This format doesn't give us much beyond some parsing convenience and entails worrying about byte ordering and word sizes. We're debating changing to a fully textual protocol (which would alleviate the above and give some debugging benefits at the possible cost of some decreased security).

Security

The protocol stream is currently relatively easily snooped (for remote connections at least). We're examining the requirement to allow this to be encrypted in some form to protect information being passed across a network.

One latter request has been for encrypting scripts in such a way that they are protected from casual browsing and modification. We have a modified interpreter (and encryption utility) that handles (and creates) such scripts and refuses to run unencrypted files. The encryption algorithm is fairly laughable in terms of uncrackability for anyone who has any idea of what they're doing but the level provided is acceptable for the requirements given.
Eventually I anticipate that the provision of a Tcl compiler will enable us to sidestep this to some extent.

Lessons

What Worked

- Using Tcl
- Client Server - allows selection of GUI engine without affecting interpreter/scripting, distribution of display to a separate node regardless of the capabilities of the Display system and provides a way to minimise the X/Motif application startup time.
- Abstracting Components, Enforcing a Higher level of policy - simplifies scripting and allows concentration on the UI rather than the components to provide it.

Problems

Trading Ease of Access and Flexibility.

There is still constant

tension in this process - some users want more flexibility, more access to capabilities we have not exposed, None want to lose the current level of simplicity or have to involve themselves in more detail. We cheated on this and give a fallout into the underlying toolkit. We've been working on the premise that if a lot of users are using the fall through capability for a particular capability we should promote it into a core supported option - so far it seems to be working.

Timing Issues

Separation of callback execution from User interaction opened up a race condition between User interaction and script modification (disabling) of UI components. This we kludged around with some callback and app locking commands/options. The solution isn't ideal from the script developers point of view since it exposes some recognition of the separation of interaction and execution but the users liked the problem even less and the solution is adequate if not elegant.

Character Engine

We initially treated the character engine as the limiting factor for features/functionality we had to relax this somewhat in order to be able to generate adequate Graphical interfaces (e.g. support of pixmaps). Its turned out that ignoring functionality due to graphical constraints (e.g. pictures in a character environment) and allowing script override works out well enough.

The supporting library we use instead of Xt and Motif in this environment was originally implemented as 'Motif on Character'. This model turns out to be inadequate without a mouse and being character the user expectations of capability (especially with respect to navigation) are different from Motif. The library is migrating more towards a character interface with a Motif API.

Mistakes

In hindsight we believe we made some mistakes in our presentation of the user model and functionality:

The GUI command and option model is very X/Motif-centric in orientation (due mainly the use of interface libraries based around Motif).
e.g. menu construction is an exact copy of Motifs.

In many cases the naming of options is also similar to Motifs somewhat obscure choice of resource names. We would have been better off providing specific widget commands for some of these concepts and being a bit more generic about some of our naming. We also ended heavily 'convenience' command oriented principally with List manipulation. A more consistent use of value setting options or better subcommand orientation would have reduced the initial number of commands to become familiar with.

Due to some late decisions we have somewhat inconsistent levels of abstractions between font and color and pixmap handling, the latter is not really abstracted at all (being file oriented) yet.

There is a large area of totally missing functionality with Window Manager style manipulation and access to selections and the clipboard. Most users also have their opinion of additional widgets we should support (fortunately there are a few points of common agreement).

Finally one thing we totally missed was providing more compatibility in command model with Tk when we switched to the Tk-like naming scheme. We should have also switched (or at least supported) something more object+option oriented in order to make knowledge transfer between the tcl GUIs much easier.

Deployment and Future Work

As mentioned earlier Vtcl is currently deployed as part of the base technology in SCO Open Server 5. Ports (of the graphical system only unfortunately) are available for SunOS/Solaris, HP/UX, IBM AIX, SGI IRIX, and Digital UNIX. It will also continue to be base Technology in the Merged UnixWare/OpenServer OS.

Apart from addressing some of the deficiencies mentioned above we're also working on some of the following areas:

A GUI builder Tool (in Vtcl using Vtcl).

MS Windows display Engine - Communicates remotely from an interpreter running on a UNIX Server system. The interpreter will eventually follow (sometime after Mark gets TclX ported).

We’re also pursuing providing a similar scenario with the Display engine as a downloadable Web applet allowing us to use any suitable enabled WebBrowser as an engine for Vtcl applications.

Since we're always getting requests for added widgets and we don't want to end up supporting every widget ever created we intend to add an extensibility framework to the GUI engine to allow widgets and supporting code to be added dynamically.

Along with the GUI builder we want to provide some additional support for a scripting development environment - mainly better script debugging, profiling and introspection features.

One of the latter requirements has been to get up to date with the later revs of Tcl and examine what sorts of benefit we can gain from added capability there. The tcl load command has especially interesting possibilities.

Availability

Vtcl is provided on SCO Open Server 5.0 The ports are available as TLS’s from sosco.sco.com or as part of Premier Motif.

Theres a Vtcl Web Page at
http://www.sco.com/Products/vtcl/

giving somewhat more introductory material than I've covered above, links to the TLS's and ports and links to other presentations and tutorials and general information. It’ll probably also have a copy of this paper.

(The above web page is also referenced from the tcl Web page at **http://www.sco.com/Technology/tcl/Tcl.html**)

I can be contacted for comments, opinions or technical info as **hops@sco.com**.

Source code is freely available on signing a license form by contacting chrisr@sco.com.

References

1. John K Ousterhout, "Tcl and the Tk Toolkit" Addison-Wesley 1994

2. Open Software Foundation, "Motif Style Guide Rev 1.2" 1993

3.Karl Lehenbauer, Mark Diekhans, "Extended Tcl (TclX)"

4. "Tcl Extensions"
http://www.sco.com/Technology/Tcl.html#Tcl-GUI