Chapter 5. Using Tcl/Tk

Authors: *Edward A. Lee*

Other Contributors: *Brian L. Evans*
Wei-Jen Huang
Alan Kamas
Kennard White

5.1 Introduction

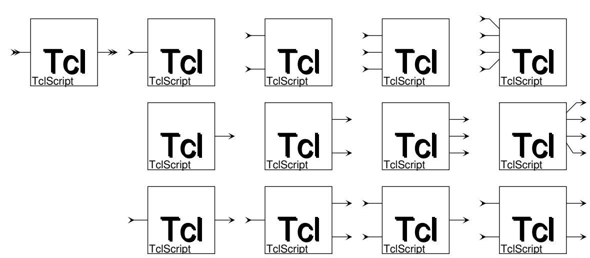
`Tcl` is an interpreted "tool command language" designed by John Ousterhout while at UC Berkeley. `Tk` is an associated X window toolkit. Both have been integrated into Ptolemy. Parts of the graphical user interface and all of the textual interpreter `ptcl` are designed using them. Several of the stars in the standard star library also use Tcl/Tk. This chapter explains how to use the most basic of these stars, `TclScript`, as well how to design such stars from scratch. It is possible to define very sophisticated, totally customized user interfaces using this mechanism.

In this chapter, we assume the reader is familiar with the Tcl language. Documentation is provided along with the Ptolemy distribution in the `$PTOLEMY/tcltk/itcl/man` directory in Unix man page format. HTML format documentation is available from the other.src tar file in `$PTOLEMY/src/tcltk`. Up-to-date documentation and software releases are available by on the SunScript web page at `http://www.sunscript.com`. There is also a newsgroup called `comp.lang.tcl`. This news group accumulates a list of frequently asked questions about Tcl which is available `http://www.teraform.com/%7Elvirden/tcl-faq/`.

The principal use of Tcl/Tk in Ptolemy is to customize the user interface. Stars can be created that interact with the user in specialized ways, by creating customized displays or by soliciting graphical inputs.

5.2 Writing Tcl/Tk scripts for the TclScript star

Several of the domains in Ptolemy have a star called `TclScript`. This star provides the quickest and easiest path to a customized user interface. The icon can take any number of forms, including the following:



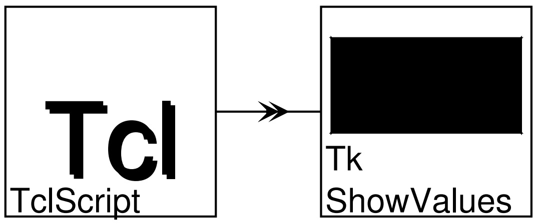
All of these icons refer to the same star, but each has been customized for a particular number of input and output ports. You should select the one you need on the basis of the number of input and output ports required. The left-most icon has an unspecified number of inputs and outputs (as indicated by the double arrows at its input and output ports).

The `TclScript` star has one parameter (settable state):

tcl_file A string giving the full path name of a file containing a Tcl script

The Tcl script file specifies initialization commands, for example to open new windows on the screen, and may optionally define a procedure to be invoked by the star every time it runs. We begin with two examples that illustrate most of the key techniques needed to use this star:

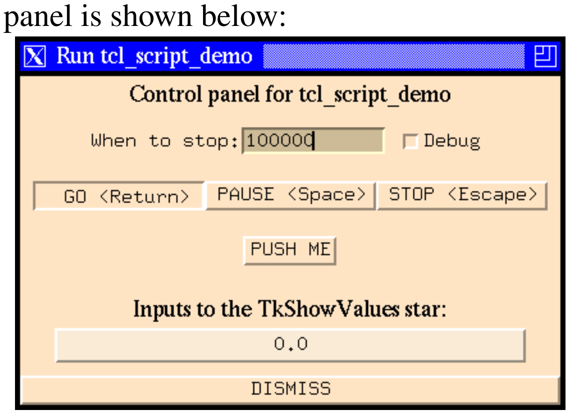
Example 1: Consider the following simple schematic in the SDF domain:



The `TkShowValues` star is in the standard SDF star library. It displays whatever input values are supplied in a subpanel of the control panel for the system. Suppose we specify the following Tcl script for the `TclScript` star:

```
set s $ptkControlPanel.middle.button_$starID
if {! [winfo exists $s]} {
      button $s -text "PUSH ME"
      pack append $ptkControlPanel.middle $s {top}
      bind $s <ButtonPress-1> "setOutputs_$starID 1.0"
      bind $s <ButtonRelease-1> "setOutputs_$starID 0.0"
      setOutputs_$starID 0.0
}
unset s
```

This script creates a pushbutton in the control panel. When the button is depressed, the star outputs the value 1.0. When the button is released, the star outputs value 0.0. The resulting control panel is shown below:



While the system is running, depressing the button labeled “PUSH ME” will cause the value displayed at the bottom to change from 0.0 to 1.0. Releasing the button will change the value back to 0.0. The lines in the Tcl script are explained below:

```
set s $ptkControlPanel.middle.button_$starID
```

This defines a Tcl variable "s" whose value is the name of the window to be used for the button. The first part of the name, `$ptkControlPanel`, is a global variable giving the name of the control panel window itself. This global variable has been set by pigi and can be used by any Tcl script. The second part, `.middle`, specifies that the button should appear in the subwindow named `.middle` of the control panel. The control panel, by default, has empty subwindows named `.high`, `.middle`, and `.low`. The last part, `.button_$starID`, gives a unique name to the button itself. The Tcl variable `starID` has been set by the `TclScript` star to a name that is guaranteed to be unique for each instance of the star. Using a unique name for the button permits multiple instances of the star in a schematic to create separate buttons in the control window without conflict.

```
if {! [winfo exists $s]} {
    ...
}
```

This conditionally checks to see whether or not the button already exists. If, for example, the system is being run a second time, then there is no need to create the button a second time. In fact, an attempt to do so will generate an error message. If the button does not already exist, then it is created by the following lines:

```
button $s -text "PUSH ME"
pack append $ptkControlPanel.middle $s {top}
```

The first of these defines the button, and the second packs it into the control panel (see the Tk documentation). The following Tcl statement binds a particular command to a mouse action, thus defining the response when the button is pushed:

```
bind $s <ButtonPress-1> "setOutputs_$starID 1.0"
```

When button number 1 of the mouse is pressed, the Tcl interpreter invokes a procedure named `setOutputs_$starID` with a single argument, `1.0` (passed as a string). This procedure has been defined by the `TclScript` star. It sets the value(s) of the outputs of the star. In this case, there is only one output, so there is only one argument. The next statement defines the action when the button is released:

```
bind $s <ButtonRelease-1> "setOutputs_$starID 0.0"
```

The next statement initializes the output of the star to value 0.0:

```
setOutputs_$starID 0.0
```

The last command unsets the variable s, since it is no longer needed:

```
unset s
```

As illustrated in the previous example, a number of procedures and global variables will have been defined for use by the Tcl script by the time it is sourced. These enable the script to modify the control panel, define unique window names, and set initial output values for the star. Much of the complexity in the above example is due to the need to use unique names for each star instance that sources this script. In the above example, the Tcl procedure for setting the output values has a name unique to this star. Moreover, the name of the button in the control panel has to be unique to handle the case when more than one `TclScript` star sources the same Tcl script. These unique names are constructed using a unique string defined by the star prior to sourcing the script. That string is made available to the Tcl script in the form of a global Tcl variable `starID`. The procedure used by the Tcl script to set output values is called `setOutputs_$starID`. This procedure takes as many arguments as there are output ports. The argument list should contain a floating-point value for each output of the star.

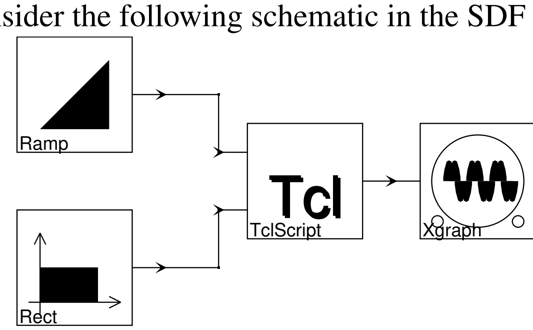
In the above example, Tcl code is executed when the Tcl script is sourced. This occurs during the setup phase of the execution of the star. After the setup phase, no Tcl code will be executed unless the user pushes the "PUSH ME" button. The command

```
bind $s <ButtonPress-1> "setOutputs_$starID 1.0"
```

defines a Tcl command to be executed asynchronously. Notice that the command is enclosed in quotation marks, not braces. Tcl aficionados will recognize that this is necessary to ensure that the `starID` variable is evaluated when the command binding occurs (when the script is sourced), rather than when the command is executed. There is no guarantee that the variable will be set when the command is executed.

In the above example, no Tcl code is executed when the star fires. The following example shows how to define Tcl code to be executed each time the star fires, and also how to read the inputs of the star from Tcl.

Example 2: Consider the following schematic in the SDF domain:



Suppose we specify the following Tcl script for the `TclScript` star:

```
proc goTcl_$starID {starID} {
      set inputVals [grabInputs_$starID]
      set xin [lindex $inputVals 0]
      set yin [lindex $inputVals 1]
      setOutputs_$starID [expr $xin+$yin]
}
```

Unlike the previous example, this script does not define any code that runs when the script is sourced, during the setup phase of execution of the star. Instead, it simply defines a procedure with a name unique to the instance of the star. This procedure reads two input values, adds them, and writes the result to the output. Although this would be a very costly way to accomplish addition in Ptolemy, this example nonetheless illustrates an important point. If a Tcl script sourced by a `TclScript` star defines a procedure called `goTcl_$starID`, then that procedure will be invoked every time the star fires. The single argument passed to the procedure when it is called is the `starID`. In this example, the procedure uses `grabInputs_$starID`, defined by the `TclScript` star, to read the inputs. The current input values are returned by this procedure as a list, so the Tcl command `lindex` is used to index into the list. The final line adds the two inputs and sends the result to the output.

As shown in the previous example, if the Tcl script defines the optional Tcl procedure `goTcl_$starID`, then that procedure will be invoked every time the star fires. It takes one argument (the `starID`) and returns nothing. This procedure, therefore, allows for *synchronous* communication between the Ptolemy simulation and the Tcl code (it is synchronized to the firing of the star). If no `goTcl_$starID` procedure is defined, then communication is *asynchronous* (Tcl commands are invoked at arbitrary times, as specified when the script is read). For asynchronous operation, typically X events are bound to Tcl/Tk commands that read or write data to the star.

The inputs to the star can be of any type. The `print()` method of the particle is used to construct a string passed to Tcl. Although it is not illustrated in the above examples, asynchronous reads of the star inputs are also allowed.

Below is a summary of the Tcl procedures used when executing a `TclScript` star:

`grabInputs_$starID`
: A procedure that returns the current values of the inputs of the star corresponding to the given `starID`. This procedure is defined by the `TclScript` star if and only if the instance of the star has at least one input port.

`setOutputs_$starID`
: A procedure that takes one argument for each output of the `TclScript` star. The value becomes the new output value for the star. This procedure is defined by the `TclScript` star if and only if the instance of the star has at least one output port.

`goTcl_$starID`
: If this procedure is defined in the Tcl script associated with an instance of the `TclScript` star, then it will be invoked every time the star fires.

`wrapupTcl_$starID`
: If this procedure is defined in the Tcl script associated with an instance of the `TclScript` star, then it will be invoked every

time the `wrapup` method of the star is invoked. In other words, it will be invoked when a simulation stops.

`destructorTcl_$starID`
: If this procedure is defined in the Tcl script associated with an instance of the `TclScript` star, then it will be invoked when the destructor for the star is invoked. This can be used to destroy windows or to unset variables that will no longer be needed.

In addition to the `starID` global variable, the `TclScript` star makes other information available to the Tcl script. The mechanism used is to define an array with a name equal to the value of the `starID` variable. Tcl arrays are indexed by strings. Thus, not only is `starID` a global variable, but so is `$starID`. The value of the former is a unique string, while the value of the latter is an array. One of the entries in this array gives the number of inputs that are connected to the star. The value of the expression `[set ${starID}(numInputs)]` is an integer giving the number of inputs. The Tcl command "`set`", when given only one argument, returns the value of the variable whose name is given by that argument. The array entries are summarized below:

`$starID`
: This evaluates to a string that is different for every instance of the `TclScript` star. The `starID` global variable is set by the `TclScript` star.

`[set ${starID}(numInputs)]`
: This evaluates to the number of inputs that are connected to the star.

`[set ${starID}(numOutputs)]`
: This evaluates to the number of outputs that are connected to the star.

`[set ${starID}(tcl_file)]`
: This evaluates to the name of the file containing the Tcl script associated with the star.

`[set ${starID}(fullName)]`
: This evaluates to the full name of the star (which is of the form universe.galaxy.galaxy.star).

5.3 Tcl utilities that are available to the programmer

A number of Tcl global variables and procedures that will be useful to the Tcl programmer have been incorporated into Ptolemy. Any of these can be used in any Tcl script associated with an instance of the `TclScript` star. For example, in example 1 on page 5-2, the global variable `ptkControlPanel` specifies the control panel that is used to run the system. Below is a list of the useful global variables that have been set by the graphical interface (`pigi`) when the Tcl script is sourced or when the `goTcl_$starID` procedure is invoked:

`$ptkControlPanel`
: A string giving the name of the control panel window associated with a given run. This variable is set by pigi.

`$ptkControlPanel.high`
The uppermost panel in the control panel that is intended for user-defined entries.

`$ptkControlPanel.middle`
The middle panel in the control panel that is intended for user-defined entries.

`$ptkControlPanel.low`
The lowest panel in the control panel that is intended for user-defined entries.

In addition to these global variables, a number of procedures have been supplied. Using these procedures can ensure a consistent look-and-feel across a variety of Ptolemy applications. The complete set of procedures can be found in `$PTOLEMY/lib/tcl`. We list a few of the more useful ones here. Note also that the entire set of commands defined in the Tcl-based textual interpreter for Ptolemy, `ptcl`, are also available. So for example, the command `curuniverse` will return the name of the current universe. See the `ptcl` chapter in the *User's Manual*.

`ptkExpandEnvVar`
Procedure to expand a string that begins with an environment variable reference. For example,

```
ptkExpandEnvVar $PTOLEMY/src
```

will return something like

```
/usr/users/ptolemy/src
```

Arguments:
path the string to expand

`ptkImportantMessage`
Procedure to pop up a message window and grab the focus. The process is suspended until the message is dismissed.
Arguments:
win window name to use for the message
text text to display in the pop-up window

`ptkMakeButton` Procedure to make a pushbutton in a window. A callback procedure must be defined by the programmer. It will be called whenever the user pushes the button, and takes no arguments.
Arguments:
win name of window to contain the button
name name to use for the button itself
desc description to be put into the display
callback name of callback procedure to register changes

`ptkMakeEntry` Procedure to make a text entry box in a window. A callback procedure must be defined by the programmer. It will be called whenever the user changes the value in the entry box and types

<Return>. Its single argument will be the new value of the entry.

Arguments:

win	name of window to contain the entry box
name	name to use for the entry box itself
desc	description to be put into the display
default	the initial value of the entry
callback	name of callback procedure to register changes

`ptkMakeMeter` Procedure to make a bar-type meter in a window.

Arguments:

win	name of window to contain the entry box
name	name to use for the entry box itself
desc	description to be put into the display
low	the value of the low end of the scale
high	the value of the high end of the scale

`ptkSetMeter` Procedure to set the value of a bar-type meter created with `ptkMakeMeter`.

Arguments:

win	name of window to contain the entry box
name	name to use for the entry box itself
value	the new value to display in the meter

`ptkMakeScale` Procedure to make a sliding scale. All scales in the control panel range from 0 to 100. A callback procedure must be defined by the programmer. It will be called whenever the user moves the control on the scale. Its single argument will be the new position of the control, between 0 and 100.

Arguments:

win	name of window to contain the scale
name	name to use for the scale itself
desc	description to be put into the display
position	initial integer position between 0 and 100
callback	name of callback procedure to register changes

Note:

A widget is created with name `$win.$name.value` that should be used by the programmer to display the current value of the slider. Thus, the callback procedure should contain a command like:

```
$win.$name.value configure -text $new_value
```

to display the new value after the slider has been moved. This is not performed automatically because the fixed range from 0 to 100 may be correct from the user's perspective. So, for example, if you divide the scale value by 100 before displaying it, then to the user, it will appear as if the scale ranges from 0.0 to 1.0. It is also possible to control the position of the slider from Tcl (overriding the user actions) using a command like

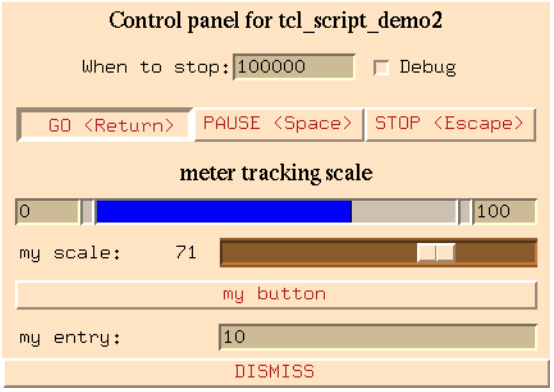
```
$win.$name.scale set $position
```

where `position` is an integer-valued variable in the range of 0 to 100.

Example 3: The following Tcl script can be used with the `TclScript` star in the system configuration given in example 1 on page 5-2:

```
ptkMakeMeter $ptkControlPanel.high meter_$starID \
      "meter tracking scale" 0 100
proc scale_update_$starID {new_value} \
      "ptkSetMeter $ptkControlPanel.high \
            meter_$starID \$new_value
       $ptkControlPanel.high.scale_$starID.value \
            configure -text \$new_value"
ptkMakeScale $ptkControlPanel.high scale_$starID \
      "my scale" 50 scale_update_$starID
ptkMakeButton $ptkControlPanel.middle button_$starID \
      "my button" button_update
proc button_update {} {ptkImportantMessage .msg "Hello"}
ptkMakeEntry $ptkControlPanel.low entry_$starID \
      "my entry" 10 entry_update_$starID
proc entry_update_$starID {new_value} \
      "setOutputs_$starID \$new_value"
```

It will create the rather ugly control panel shown below:



The commands are explained individually below.

```
ptkMakeMeter $ptkControlPanel.high meter_$starID \
```

```
          "meter tracking scale" 0 100
```

This creates a meter display with the label “meter tracking scale” in the upper part of the control panel with range from 0 to 100.

```
proc scale_update_$starID {new_value} \
     "ptkSetMeter $ptkControlPanel.high \
          meter_$starID \$new_value
      $ptkControlPanel.high.scale_$starID.value \
          configure -text \$new_value"
```

This defines the callback function to be used for the slider (scale) shown below the meter. The callback function sets the meter and updates the numeric display to the left of the slider. Notice that the body of the procedure is enclosed in quotation marks rather than the usual braces. This ensures that the variables `ptkControlPanel` and `starID` will be evaluated at the time the procedure is defined, rather than at the time it is invoked. To make sure that `new_value` is not evaluated until the procedure is invoked, we use a preceding backslash, as in `\$new_value`. We could have alternatively passed the `ptkControlPanel` and `starID` values as arguments.

```
ptkMakeScale $ptkControlPanel.high scale_$starID \
     my_scale 50 scale_update_$starID
```

This makes the slider itself, and sets its initial value to 50, half of full scale.

```
ptkMakeButton $ptkControlPanel.middle button_$starID \
     "my button" button_update
```

This makes a button labeled “my button”.

```
proc button_update {} {ptkImportantMessage .msg "Hello"}
```

This defines the callback function connected with the button. This callback function opens a new window with the message “Hello”, and grabs the focus. The user must dismiss the new window before continuing.

```
ptkMakeEntry $ptkControlPanel.low entry_$starID \
     "my entry" 10 entry_update_$starID
```

This makes the entry box with initial value “10”.

```
proc entry_update_$starID {new_value} \
     "setOutputs_$starID \$new_value"
```

This defines the callback function associated with the entry box. Again notice that the procedure body is enclosed quotation marks.

5.4 Creating new stars derived from the TclScript star

A large number of useful stars can be derived from the `TclScript` star. The `TkShowValues` star used in example 1 on page 5-2 is such a star. That star takes inputs of any type and displays their value in a window that is optionally located in the control panel. It has three parameters (settable states):

label A string-valued parameter giving a label to identify the display.

put_in_control_panel A Boolean-valued parameter that specifies whether the display should be put in the control panel or in its own window.

wait_between_outputs A Boolean-valued parameter that specifies whether the execution of the system should pause each time a new value is displayed. If it does, then a mouse click in the display restarts the system.

Conspicuously absent is the *tcl_file* parameter of the `TclScript` star from which this is derived. The file is hard-wired into the definition of the star by the following C++ statement included in the setup method:

```
tcl_file =
"$PTOLEMY/src/domains/sdf/tcltk/stars/tkShowValues.tcl";
```

The parameter is then hidden from the user of the star by the following statement included in the constructor:

```
tcl_file.clearAttributes(A_SETTABLE);
```

Thus, the user sees only the parameters that are defined in the derived star. This is a key part of customizing the star.

A second issue is that of communicating the new parameter values to the Tcl script. For example, the Tcl script will need to know the value of the *label* parameter in order to create the label for the display. The `TclScript` star automatically makes all the parameters of any derived star available as array entries in the global array whose name is given by the global variable `starID`. To read the value of the *label* parameter in the Tcl script, use the expression `[set ${starID}(label)]`. The confusing syntax is required to ensure that Tcl uses the *value* of `starID` as the *name* of the array. The string "label" is just the index into the array. The `set` command in Tcl, when given only one argument, returns the value of the variable whose name is given by the argument.

Some programmers may prefer an alternative way to refer to parameters that is slightly more readable. The Tcl statement

```
upvar #0 $starID params
```

allows subsequent statement to refer to parameters simply as `$param(param_name)`. The `upvar` command with argument `#0` declares the local variable `params` equivalent to the global variable whose name is given by the value of `starID`.

Many more examples can be found in `$PTOLEMY/src/domains/sdf/tcltk/stars`.

5.5 Selecting colors

Since X window installations do not necessarily use consistent color names, a particular color database has been installed in Ptolemy. The available colors can be found in the file `$PTOLEMY/lib/tcl/ptkColor.tcl`. To access this color database, use the Tcl function

```
ptkColor name
```

which returns a color defined in terms of RGB components. This color can be used anyplace that Tk expects a color. If the given name is not in the color database, the color returned is black.

5.6 Writing Tcl stars for the DE domain

In the discrete-event (DE) domain, stars are fired in chronological order according to the time stamps of the new data that has arrived at their input ports. The Tcl interface class `TclStarIfc`, which was originally written with the SDF domain in mind, works well for some types of DE stars. Specifically, any star with an input in the DE domain can use this class effectively. Consequently, a basic Tcl/Tk star, `TclScript`, has been written for the DE domain.

The `TclScript` star can have any number of input or output portholes. As of this writing, it will not work if it is instantiated with no inputs. The problem is that with no inputs, there will be no events to trigger a firing of the star. This will be corrected in the future.