the tclsh spot

by Clif Flynt

Clif Flynt is president of Noumena Corp., which offers training and consulting services for Tcl/Tk and Internet applications. He is the author of *Tcl/Tk: A Developer's Guide* and the *TclTutor* instruction package. He has been programming computers since 1970 and a Tcl advocate since 1994.

clif@cflynt.com

The previous Tclsh Spot article showed how to extend a FORTRAN program by embedding the Tcl interpreter into the application.

This architecture—compiled mainline code with an embedded interpreter—offers a number of advantages. It provides a robust runtime configuration facility and enables the developer to concentrate on one portion of a task at a time, rather than mixing calculation, communication, and GUI elements. Writing the GUI in a higher-level language reduces development time, since this subsystem usually requires several redesigns and iterations before users are happy with it. And, from a marketing end, it means you can easily distribute the core product with different front ends.

The previous article demonstrated how to rewrite the old FORTRAN Lunar Lander using FORTRAN for the mainline code and calculation subroutine and Tcl for the user interface.

This article will expand the user interface without touching the core FORTRAN code (much).

The one change to the original FORTRAN code is to move the hard-coded FORTRAN constants into Tcl variables so that they can be defined at runtime.

The original code was:

```
! Set constants
impulse = 2000
fheight = 10000.0
speed = 100.0
fuel = 1000.0
gross = 900.0
CALL ftcl_start('config.tcl')
```

And the new code is:

```
CALL ftcl_start('config.tcl')
! Fetch constants defined in Tcl script
CALL ftcl_get_int('impulse', impulse)
CALL ftcl_get_real('ht', fheight)
CALL ftcl_get_real('speed', speed)
CALL ftcl_get_real('fuel', fuel)
CALL ftcl_get_real('gross', gross)
```

This is the obvious use for a Tcl configuration file. Config.tcl can contain code like:

```
set impulse 2000
set fheight 10000.00
set speed 100.0
...
```

Or the configuration file could contain a script to let the user set configuration options. This could be a glass tty set of questions and answers, a form to fill out, or even a button bar like this to set gravitational acceleration for different destinations:

The 8.4 release of Tcl/Tk (2003) introduced a new widget (the labelframe) and a new option to the button command (-compound) to make it easier to create button bars in Tk.

The labelframe widget (described in the previous Tclsh Spot) behaves like a normal frame; it holds other widgets, and it also supports options to control the outline and label.

Syntax: labelframe widgetName ?-option value?

The code to create the labelframe that holds the button bar is simply:

```
set w [labelframe .planet -text "Choose your Destination"]
```

The new button option -compound makes it easy to create buttons with both an image and text. Prior to 8.4, a button could contain either an image or text, but not both.

The Tk image command is quite powerful, and the newer versions of Tk are adding more facilities. Pure Tk supports images in GIF, PBM or X-Bitmap format. The img extension adds support for JPG, TIF, BMP and other formats.

You can create a Tk image either from a file or by embedding the data as base-64 data. The syntax for the image create command is:

Syntax: image create type ?name? ?options?

image create	Create an image object of the desired type, and return a handle for referencing this object.
type	The type of image that will be created. May be
	bitmap a two-color graphic.
	photo a multicolor graphic.
?name?	The name for this image.
?options?	Options that are specific to the type of image being created.

Embedding the image as a base-64 data is an easy way to create task button bars.

For example, this code will create a simple two-button taskbar that will invoke procedures named do_open and do_save when the appropriate button is clicked.

```
image create photo open -data {
        R0lGODlhEgASAPIAAAAAAICAAMDAwPj8APj8
        +AAAAAAAAAAAACH5BAEAAAIALAAAAAASABIA
        AAM7KLrc/jAKQCUDC2N7t6JeA2YDMYDoA5hs
        WYZk28LfjN4b4AJB7/ue1elHDOl4RKAImQzQ
        cDeOdEp1JAAAO/// }

image create photo save -data {
        R0lGODlhEgASAPEAAAAAAICAAMDAwAAAACH5
        BAEAAAIALAAAAAASABIAAAI3lI+pywYPY0Qg
        AHbvqVpBamHhNnqlwIkdeoJrZUlPcML0jde0
        DOnxxHrtJA6frLhC8YiNpnNRAAA7//// }

set column 0
foreach img [lsort [image names]] {
```

```
    set w [button .b_$img -image $img -command do_$img]
    grid $w -row 1 -column [incr column] }
```

The button bar that opens the Lander program is a bit more complex. It uses the -compound option to show both text and images, and it waits for the user to select a destination before continuing.

The -compound option specifies that a button should show both text and image, and defines where to place the image. The -compound key can be bottom, center, left, right, top, or none, to define where to place the image relative to the text. The value none defines the button as having either image or text, but not both. This is the default.

The buttons, created using inline base-64 data as done in the previous example, are named for and contain small images of the destination they represent: Venus, Earth, Moon, Mars.

The buttons are created with this code:

```
set w [labelframe .planet -text "Choose your Destination"]

foreach planet {Venus Earth Moon Mars} {
    set b [button $w.b_$planet -compound bottom -text $planet \
        -image $planet -command "setConditions $w $planet"]
    pack $b -side left
}
```

The button bar can be turned into a modal interaction using the vwait command (discussed in the previous "Tclsh Spot"). The vwait command will cause the Tcl interpreter to wait until a variable has changed value.

After the window is displayed, a vwait can hold the Tcl interpreter in the event loop (waiting for the user to select a destination) until the variable changes state.

```
    # Create buttons
    #...   pack $w
    vwait gravity
```

When the user clicks a button, it will invoke the setConditions procedure with the name of the parent window (the labelframe) and the destination. This procedure assigns a value to the gravitational acceleration and destination variables and destroys the parent window.

```
array set Gravities {Venus 8.8 Earth 9.8 Moon 1.7 Mars 3.9}

proc setConditions {parent dest} {
    global Gravities gravity ready destination
    set destination $dest
    set gravity $Gravities($destination)
    destroy $parent
    set ready 1
}
```

When setConditions assigns a value to gravity and returns control to the event loop, the vwait command is satisfied, and the evaluation of the script continues.

The previous version of the Lander program would wait for the user to type in a new fuel burn and click the **Go** button before calculating the new lander conditions. A more modern version of the lander will use the Tcl scale command to accept the input and run the simulation in realtime.

The scale widget allows a user to drag a slider to select a numeric value.

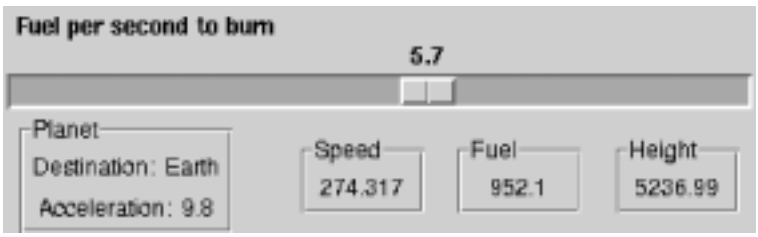
Syntax: scale scaleName ?options?

scaleName	The name for this scale widget.	
?options?	There are many options for this widget. The minimal set is:	
	-orient orientation	Whether the scale should be drawn horizontally or vertically: orientation may be horizontal or vertical. The default orientation is vertical.
	-length numPixels	The size of this scale. The height for vertical widgets, and the width for horizontal widgets. The height may be any valid Tk distance format, including inches, mm, points, or pixels.
	-from number	One end of the range to display. This value will be displayed on the left side (for horizontal scale widgets) or top (for vertical scale widgets).
	-to number	The other end for the range.
	-label text	The label to display with this scale.
	-command script	The command to evaluate when the state changes. The new value of the slider will be appended to this string, and the resulting string will be evaluated.
	-variable varName	A variable which will contain the current value of the slider.
	-resolution number	The resolution to use for the scale and slider. Defaults to 1.
	-tickinterval number	The resolution to use for the scale. This does not affect the values returned when the slider is moved.

Several Tk widgets support linking a variable to the widget, as is done using the -variable option in this scale widget. When the user moves the scale, the value of that variable automatically changes, and if the script modifies the value of the variable, the scale widget will move to reflect the change.

A horizontal scale bar to control the fuel burn can be built with this code.

```
scale .s -digits 3 -from 0 -to 10 \
   -resolution .1 -showvalue true -length 400 \
   -label "Fuel per second to burn"  \
   -variable burn -orient horizontal
grid .s -row 2 -column 1  -sticky ew -columnspan 5
```



Using this scale widget and some labelframes and labels, we can make a control GUI that looks like this to set the fuel to burn and display the current status of the lander:

You may notice that this GUI lacks the **Go** button the previous lander had for accepting the amount of fuel to burn.

We can even change the behavior of the application from stoptime to realtime in the Tk GUI, without modifying the compiled code.

The FORTRAN code calls a Tcl procedure wait4click to wait for a user to hit the **Go** button. In the stoptime version of this GUI, that procedure used vwait to pause execution until the user clicked the **Go** button.

```
proc wait4click {} {
      global ready
      vwait ready
}
...
          button .b -text "Go" -command "set ready 1"
```

To make the application run in realtime, we can use the Tcl after command to set the ready variable. The after call acts like an automated user clicking the **Go** button once a second.

```
proc wait4click {} {
      global ready
      vwait ready
      after 1000 {set ready 1}
}
```

The ready variable is modified the first time the user selects the **Gravity** (in the setConditions procedure) and is then modified by the after script once a second.

A realtime Lander game with a slider, though an improvement over the old FORTRAN type interface requiring numbers to be typed in, would be nicer yet with a graphical display. The next "Tclsh Spot" will describe better ways to represent the data.