;login:
THE MAGAZINE OF USENIX & SAGE
August 2002 volume 27 • number 4

inside:



USENIX & SAGE
The Advanced Computing Systems Association &
The System Administrators Guild

# 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 for Real Programmers* and the *TclTutor* instruction package. He has been programming computers since 1970 and a Tcl advocate since 1994.

*clif@cflynt.com*

One of the most useful features of modern GUIs is the little pop-up help window. Whenever I end up with a new application, and no time to actually read a manual, I'll let the cursor rest on a bizarrely named button and hope I get a hint for what it will do.

Tk does not include a pop-up help widget as one of the basic widgets, but one can be created with just a few lines of code. The help balloon described in this article was submitted to the Tcler's Wiki by Daniel Steffen (*http://www.maths.mq.edu.au/~steffen/tcltk*).

The code for a help balloon is fairly short, but not trivial. Creating a help balloon requires interacting with a several aspects of the window manager and Tk interpreter, and it's not always obvious how to gain access to the feature you need. Knowing what types of information are controlled by the window manager, and which are controlled by the Tk interpreter makes it a bit easier.

The first trick with a help balloon is that we need to know when the cursor has entered a window that has a help balloon associated with it.

Tcl handles linking an action to an event with the bind command. The bind command links a Tcl script to a window and event. When that window has focus, and that event occurs, the registered script will be evaluated.

The command looks like this:

**Syntax:** bind *window event script*
This causes script to be evaluated if event occurs while window has focus.

window The name of the window to which this script will be bound

event The event to use as a trigger for this script

script The script to evaluate when the event occurs

The events that will trigger evaluating the script are defined as zero or more modifiers, followed by an event-type descriptor, followed by a detail field. You must have at least a type or detail field in the event descriptor. Depending on the event, more fields may be required. The fields can be separated by white-space or dashes.

The event types include all the events supported by the X Window System:

| | | |
|---|---|---|
| Activate | Enter | Map |
| ButtonPress, Button | Expose | Motion |
| ButtonRelease | FocusIn | MouseWheel |
| Circulate | FocusOut | Property |
| Colormap | Gravity | Reparent |
| Configure | KeyPress, Key | Unmap |
| Deactivate | KeyRelease | Visibility |
| Destroy | Leave | |

A simple event would be something like <H>, which would trigger on someone typing an uppercase H. In this case the event descriptor is just a detail field, with an implicit type of KeyPress.

The detail field describes the event in more detail. For example <KeyPress-H> would also describe the event when someone types an uppercase H, and <ButtonPress-1> describes the event when someone clicks the leftmost button.

The modifier field adds information about events that must happen simultaneously (like Control, Alt and Delete being held down together), or sequentially, like mouse double clicks.

Modifiers include Control, Shift, Lock and Alt, to describe a key that must be depressed when the event occurs, or Double, Triple, and Quadruple to describe how many times the event must occur: <Double-ButtonPress-1> describes the event when someone double-clicks the left mouse button. We could watch for someone triple-clicking while holding the Control key with <Triple-Control-ButtonPress-1>.

To make a help balloon, we want to know when the cursor enters or leaves a widget. The Enter and Leave events are generated when a cursor enters or leaves a widget, so a pair of lines like the following would display and destroy a balloon when the cursor enters and leaves a widget named .needsHelp:

```
bind .needsHelp <Enter> "create Balloon"
bind .needsHelp <Leave> "destroy Balloon"
```

Since a help balloon should appear below the widget that it relates to, the code that will create a balloon needs to know where that window is. Though the window name sounds like a line from a bad fantasy novel, knowing it allows you to learn its location.

The bind command will let us pass certain runtime values to the script that is evaluated when the event occurs. These values are defined in the script as a percent-item, which will be substituted for the actual value just before the script is evaluated.

The bind command supports many percent-items, including:

| | |
|---|---|
| %b | The number of the button that was pressed to generate this event. Valid only for ButtonPress or ButtonRelease. |
| %d | The detail field from the event. |
| %h | The height field from the event. Valid for Configure and Expose. |
| %k | The key that was pressed or released. Valid only for KeyPress or KeyRelease. |
| %x %y | The X or Y coordinates for the event. Valid for events such as mouse events that have an X or Y field. |
| %R %S | The root or subwindow identifier for the event. |
| %W | The window for which this event is being reported. |

The %W option lets us tie a help balloon to the window that created it. A command to bind help-balloon creation to a widget might look more like this:

```
bind .needsHelp <Enter>
                "createBalloonProc %W $helpMessage"
```

Help windows should appear, not immediately, but a second or two after the cursor enters a widget. This means we need to have a way to schedule an event to occur in the future.

The Tcl after command enables a script to react to a timer event or idle condition. This command has several subcommands that will let an application interact with the queue of scripts waiting for a chance to happen, but for a help balloon we only need the simple form of:

**Syntax:** after *milliseconds script*

| | |
|---|---|
| after | Schedule a script to be processed in the future. |
| *milliseconds* | The number of milliseconds to pause the current processing, or the number of seconds in the future to evaluate another script. |
| *script* | The script to be evaluated after the number of *milliseconds* have elapsed. |

Given a procedure to create the balloon named balloon:show, the beginning of a procedure to add a help balloon to a widget looks like this:

```
proc balloon {w help} {
    bind $w <Enter>
        "after 1000 [list balloon:show %W [list $help]]"
```

We can't leave that balloon up forever, so we need to be able to destroy the balloon when the cursor leaves the target widget.

The Tcl command to destroy a window is destroy.

**Syntax:** destroy *windowName ?window2...?*

Destroy one or more Tcl widgets.

| | |
|---|---|
| *windowName* | The name of the Tcl widgets to be destroyed. |

We can decide to name the help balloon the .balloon child of the window it relates to. This makes the entire balloon registration procedure look like this:

```
proc balloon {w help} {
    bind $w <Any-Enter>
        "after 1000 [list balloon:show %W [list $help]]"
    bind $w <Any-Leave> "destroy %W.balloon"
}
```

The balloon:show procedure will create and display the help balloon. There are a few steps in this process.

1. Confirm that the cursor is still inside the window that is associated with this help balloon.
2. Destroy any previous balloon associated with this window.
3. Create a new window with the appropriate text.
4. Map this window to the screen in the appropriate place.

Several of these steps require information from the window system: finding the location of the cursor, the location of a widget, etc.

Tk provides for interaction with a windowing system via two commands: the winfo command that returns information about the windows Tk controls, and the wm command that interacts with the window manager.

These commands have many subcommands, most of which aren't needed for this application. I'll just discuss a few of them as they become necessary.

The first step, confirming that the cursor is still within the window, can be done with two winfo commands. The pointerxy subcommand will return the coordinates of the cursor, and the containing subcommand will return the name of a window that encloses a pair of coordinates.

**Syntax:** winfo pointerxy *window*

Return the X and Y location of the mouse cursor. These values are returned in screen coordinates, not application window coordinates.

*window* The mouse cursor must be on the same screen as this window. If the cursor is not on this screen, then the coordinates will each be -1.

**Syntax:** winfo containing *rootX rootY*

Returns the name of the window that encloses the X and Y coordinates.

*rootX* An X screen coordinate (0 is the left edge of the screen).

*rootY* A Y screen coordinate (0 is the top edge of the screen).

The containing subcommand requires two separate arguments, while the pointerxy returns a pair of arguments. If we tried to write this code:

```
winfo containing [winfo pointerxy .]
```

the Tcl interpreter would throw an error.

The return from winfo pointerxy . would be substituted into the command as a single unit. The command evaluated by the Tcl interpreter would resemble:

```
winfo containing {120 300}
```

instead of

```
winfo containing 120 300
```

The solution to this is to use the eval command to evaluate the string.

**Syntax:** eval *string1 ?string2...?*

Concatenate the arguments into a single string and evaluate that string as a command.

*string** Strings that will compose a command.

Because lists are concatenated onto the end of the previous data, the eval command loses one level of grouping information. If you need to maintain the grouping of some sets of data, use the list command to make a list of it.

Using a string match command to compare the window that currently has the cursor with the window that requested the help balloon, we get code like this:

```
proc balloon:show {w arg} {
    if {![string match [eval winfo containing
                    [winfo pointerxy .]] $w]} {
        return
    }
```

The next step is to destroy any previous existing balloon. This might seem unnecessary – after all, a cursor has to leave one window before it can enter another.

However, there are circumstances when a cursor *can* enter a second window without leaving the first. For example, if one widget is contained within another, the cursor can enter the inner widget without leaving the outer widget.

The example below shows an unlikely example of this situation:

```
# Create and display a canvas
canvas .c
pack .c

# Create and display a label within the canvas
label .c.l -text label
.c create window 50 50 -anchor nw -window .c.l

# Add bindings to report when the mouse enters
# and leaves the windows.
bind .c <Enter> {puts {in .c}}
bind .c <Leave> {puts {out .c}}
bind .c.l <Enter> {puts {in .c.l}}
bind .c.l <Leave> {puts {out .c.l}}
```

As a mouse cursor enters the canvas, then the label, and then leaves the label and canvas, the following output is generated:

```
in .c
in .c.l
out .c.l
out .c
```

To make the balloon help code a bit more readable, the name of the new balloon help window is saved in the variable top.

If the window does not exist, it can't be destroyed, and Tcl will throw an error. A script can catch an error with the catch command, which will evaluate a script in a safe way, and return the results and status of the script separately.

The syntax is:

**Syntax:** catch *script ?varName?*

The catch command returns the status from evaluating the script, and optionally places the results of evaluating the script in the variable *varName*.

In this case, we don't need the results from the destroy, so we can destroy any previous balloons associated with this window with:

```
set top $w.balloon
catch {destroy $top}
```

The next step is to create the new window. Tcl supports two types of windows:

- Windows that are managed within a Tk window
- Windows that are managed by the window manager

A window managed within a Tk window (like most buttons, labels, scrollbars, etc. that your script creates) must fit within the parent window. Windows that are managed by the window manager (called top level windows) can appear anywhere on the screen and may have decorative borders set by the window manager.

For a help balloon, we want a top level window (in case the widget this balloon is associated with is at the bottom corner of the application), and we want the window to not have any decorations. Our script will place a message widget inside this top level to hold the help text.

The command for creating a new top level window is toplevel.

**Syntax:** toplevel *windowName ?-option value ...?*

The options include setting the border width, relief, background, class, etc.

This application wants a very simple top level with a one-pixel-wide border.

```
toplevel $top -borderwidth 1
```

A help balloon window should not have the decorations added by the window manager – we don't want the user to be able to move this window, iconify it, etc. The decorations are added by the window manager, not managed by Tk, so removing the decorations is done with the wm command. The subcommand that handles this is override-redirect.

**Syntax:** wm override-redirect *windowName boolean*

| | |
|---|---|
| | Sets the override-redirect flag in the requested window. If true, the window is not given a decorative frame and can not be moved by the user. By default, the override-redirect flag is false. |
| *windowName* | The name of the window for which the override-redirect flag is to be set. |

*boolean* A boolean value to assign to the override-redirect flag.

The wm override-redirect command should be given before the window manager transfers focus of a window. Unlike most Tcl/Tk commands, you may not be able to test this subcommand by typing commands in an interactive session.

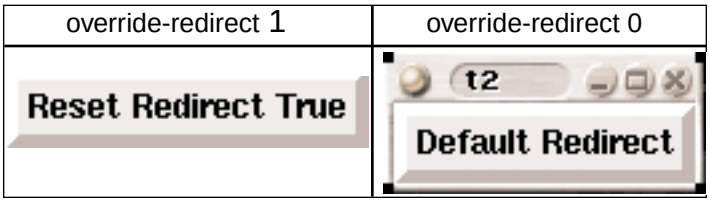
The difference between override-redirect true and false looks like this:

```
catch {destroy .t1 .t2}
toplevel .t1 -border 5 -relief raised
label .t1.l -text "Reset Redirect True"
pack .t1.l
wm override-redirect .t1 1

toplevel .t2 -border 5 -relief raised
label .t2.l -text "Default Redirect"
pack .t2.l

wm geometry .t1 +300+300
wm geometry .t2 +300+400

raise .t1
raise .t2
```




Creating the new top level and getting rid of the borders looks like this:

```
toplevel $top -borderwidth 1
wm override-redirect $top 1
```

The next step is to add the help message. Tk supports three widgets for displaying textual information:

| | |
|---|---|
| label | Displays a single line of text. |
| message | Displays one or more lines of text. |
| text | Displays one or more lines of text with support for editing, multiple fonts, tagged areas, etc. |

Any of these widgets could be used for the help message, but the help message may be longer than can fit on a single line, and the text widget is a bit heavyweight for this application. The message widget combines some of the features of the text widget and some features of the label widget, making it the best widget for this application.

**Syntax:** message *name ?options?*

| | |
|---|---|
| message | Create a message widget. |
| *name* | A name for the message widget. Must be a proper window name. |
| *?options?* | Options for the message include: |

| | |
|---|---|
| -text | The text to display in this widget. |
| -textvar | The variable which will contain the text to display in this widget. |
| -aspect | An integer to define the aspect ratio: (Xsize/Ysize) * 100 |
| -background | The background color for this widget. |

When a widget creation command is evaluated, it returns the name of the widget that was just created. This can be used with

the geometry managers to make a single-command create and display command like this:

```
pack [message $top.txt -aspect 200
      -background lightyellow \
      -font fixed -text $arg]
```

The final step is to place the new window just under the widget that requested the help balloon.

The window that requests the help will be a window managed by Tk, so we can use the winfo command to determine its height and X/Y locations.

The subcommands for these data are:

| | |
|---|---|
| winfo height*winName* | Return the height of a window in pixels. |
| winfo rootx *winName* | Return the X location of this window in screen coordinates. |
| winfo rooty *winName* | Return the Y location of this window in screen coordinates. |

These two lines of code set variables for the X coordinate to be the same as the left edge of the window requesting the help balloon, and the Y coordinate to be just below that window.

```
set wmx [winfo rootx $w]
set wmy [expr [winfo rooty $w]+[winfo height $w]]<
```

Placing a top level window on the screen is a task for the window manager, so the wm geometry command gets used.

**Syntax:** wm geometry *windowName ?geometry?*

| | |
|---|---|
| | Query or set the geometry for a window. |
| *windowName* | The name of the window to be queried or set. |
| *?geometry?* | If this is present, it's a geometry string following the X windows convention of *width*x*height*+/-*Xposition*+/-*Yposition*. The x and + or - separators are required.<br><br>If this field is not present, the wm geometry command returns the current geometry of the window. |

For most X Window window managers, we could just provide the X and Y locations for the new window:

```
wm geometry $top +$wmx+$wmy
```

But, to be completely safe on multiple platforms, with different window managers, we should provide a complete geometry specification with the width and height of the window included.

The requested width and height is known by the Tk interpreter, and is returned by the winfo reqwidth and winfo reqheight commands.

A better geometry command resembles this:

```
wm geometry $top \
    [winfo reqwidth $top.txt]x[winfo reqheight
            $top.txt]+$wmx+$wmy
```

The final step is to make sure the new window isn't hidden behind other windows. The raise command places one window above another, or above all other windows, if no other window is defined.

```
    raise $top
}
```

Wrapping all these code fragments together, the balloon:show procedure looks like this:

```
proc balloon:show {w arg} {
    if {![string match [eval winfo containing
                [winfo pointerxy .]] $w]} {
        return
    }
    set top $w.balloon
    catch {destroy $top}
    toplevel $top -borderwidth 1 -background black
    wm overrideredirect $top 1

    pack [message $top.txt -aspect 200
                -background lightyellow \
                -font fixed -text $arg]
    set wmx [winfo rootx $w]
    set wmy [expr [winfo rooty $w]+[winfo height $w]]
    wm geometry $top \
        [winfo reqwidth $top.txt]x[winfo reqheight
                $top.txt]+$wmx+$wmy
    raise $top
}
```

This code, with a tweak for Macintosh platforms, is available at *http://mini.net/tcl/534.html*.