CLIF FLYNT

the tclsh spot

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.

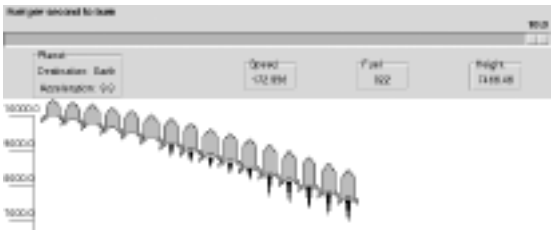
clif@cflynt.com

THE PAST FEW TCLSH SPOT ARTICLES described a software architecture in which the main-line code is compiled, and then Tcl/Tk scripts are invoked to read input and display results. A major strength of this architecture is that the look and feel of an application can be drastically modified without touching a line of compiled code.

An old FORTRAN TTY-based program, the Lunar Lander, was modified to run in real time, with a display that looks like this:

This article will show how to replace that GUI with one that shows the state of the lander graphically at each second of the descent.

The first few seconds of a landing on Earth might look like this:



The numeric values are displayed using the Tk label widget, as in the previous GUI. The area below the labels is a graph displaying the height of the lander on the Y axis, and time into the landing on the X axis. The points on the graph are drawn with rockets.

The rockets show:

- The current height of the rocket (the Y axis of the graph)
- The number of seconds into the landing (the X axis of the graph)
- The current speed (the height of the rocket)
- The remaining fuel (the width of the rocket)
- The amount of fuel burned in this timestep (the height of the flame below the rocket)

You can see in the display that the rocket falls a little faster each second when there is no thrust applied. With half-thrust it still accelerates, while reducing the amount of fuel. At full thrust, the speed is close to constant (in fact, there is a very slight negative acceleration), and the fuel is consumed more rapidly. This is

not as extensive as Charles Joseph Minard's graph of Napoleon's Russian campaign, but it does convey five dimensions of data in two dimensions.

The Tk canvas widget makes this sort of data representation easy. The Tk canvas widget is an object-based drawing surface inspired by Joel Bartlett's ezd program. It enables the programmer to define a window into an arbitrarily large drawing surface and place graphic objects on that surface. The graphic objects are each defined by a location and a set of configuration options specific to the type of object.

For example, a text object can be configured to display a certain set of text in a certain font, while a rectangle object can be assigned height, width, and colors.

Once an object is created, it can't change type, but its location and configuration options can be modified as necessary. Thus, the words displayed by a text object can be changed, as can the font or color.

A new canvas widget can be created and displayed with the same syntax as other Tk widgets. The command is `canvas`, followed by the name of this canvas, followed by a list of key/value option pairs. The newly created canvas is displayed using either the `pack`, `place`, or `grid` geometry manager.

Syntax: canvas canvasName ?options?

canvasName — The name for this canvas

?options? — Some of the options supported by the canvas widget are:

`-background` *`color`*
: The color to use for the background of this canvas. The default color is light gray.

`-height` *`size`*
: The height of the displayed portion of the canvas. If `-scrollregion` is declared larger than this, and scrollbars are attached to this canvas, this defines the height of the window into a larger canvas.
The `size` parameter may be in pixels, inches, millimeters, etc.

`-width` *`size`*
: The width of this canvas widget. This may define the size of a window into a larger canvas.

`-scrollregion` *`{left top right bottom}`*
: The region of a larger canvas for the window to scroll over. These coordinates define the area of a canvas that can scroll into view when the canvas is attached to a scrollbar widget.

Like other Tk widget creation commands, the `canvas` command returns the name of the canvas it created, and also creates a new procedure by that name to use to interact with the canvas.

For example, this command:

```
set cvs [canvas .c -height 500 -width 700 -background white]
```

creates a new canvas named `.c` and a new procedure named `.c` and assigns the value `.c` to the variable `cvs`.

The new `.c` procedure supports several subcommands, including:

`create`
: Create a new object on the canvas. Returns a unique ID for the new object.

`configure`
: Query or set canvas configuration options.

`itemconfigure`

Query or set configuration options for an item on the canvas.

`xview`

Define the window into a canvas to be displayed.

`bbox`

Returns the bounding rectangle that encloses a set of canvas items. The command *`canvasName`* `bbox all` returns the bounding rectangle that includes all items displayed in a canvas.

`bind`

Assign a binding to an item on the canvas.

The most used canvas command is the `create` command. The syntax for this command is:

Syntax: *`canvasName`* `create` *`itemType coords ?options?`*

`itemType`

The type of item to create may be `arc`, `bitmap`, `image line`, `oval`, `polygon`, `rectangle`, `text`, or `window.`

`coords`

The coordinates for this item. The coordinates are X/Y pairs. All `itemTypes` require at least one X/Y pair. Some `itemTypes` (ovals, rectangles) require two pairs to define the opposing corners of a bounding rectangle for the object. Lines and polygons can have multiple X/Y pairs to define the corners of the figure.

The `canvas` coordinate system places 0,0 on the upper left corner. X values increase toward the right, and Y values increase toward the bottom.

`options`

Keyword/value pairs to define configuration options for a graphic item. The supported keywords are different for different types of graphic items.

The axes for a graph can be created with line and text objects. The commands to create a canvas, short horizontal line, and text would look like this:

```
set cvs [canvas .c -height 500 -width 700 -background white]
$cvs create line 10 10 20 10
$cvs create text 25 10 -text "10000"
```

By adding a couple of loops, the code to draw the X and Y axes and ticks looks like this:

```
  # Create the axis lines
  .c create line 40 20 40 480
  .c create line 40 460 4000 460
  # Create and label the ticks on the Y axis
  for {set i 0} {$i <= $height}\
       {set i [expr {$i + $height/10}]} {
    set y [expr {460 - ($i * .044)}]
    .c create text 3 $y -text $i -anchor sw
    .c create line 10 $y 40 $y
  }
  # Create and label the ticks on the X axis
  for {set i 40; set j 0} {$i < 4000 }\
       {incr j; incr i 25} {
```

```
    .c create text $i 482 -text $j -anchor nw
    .c create line $i 460 $i 475
  }
```

The horizontal X axis line goes from pixel 40 to pixel 4000. That's a bit longer than most monitors. The canvas widget has commands that make it easy to attach the canvas to horizontal and vertical scrollbars.

Create a scrollbar with the `scrollbar` command:

`Syntax: scrollbar` *`scrollbarName ?options?`*

`scrollbar` Create a `scrollbar` widget.

`scrollbarName` The name for this `scrollbar`.

`options` This widget supports several options. The `-command` option is required.

`-command` *`"procName ?args?"`*
This defines the command to invoke when the state of the `scrollbar` changes. Arguments that define the changed state will be appended to the arguments defined in this option.

`-orient` *`direction`*
Defines the orientation for the scrollbar. The *`direction`* may be `horizontal` or `vertical`. Defaults to vertical.

`-troughcolor` *`color`*
Defines the color for the trough below the slider. Defaults to the default background color of the frames.

The `wish` interpreter handles the interaction between the `canvas` and `scrollbar` by registering a callback procedure with the `scrollbar` and `canvas` widgets. Whenever one of these widgets changes state, it will evaluate the registered script to update the other widget.

The code to create and display a canvas and scrollbar resembles this:

```
canvas .c -height 500 -width 700 -background white \
  -xscrollcommand {.sb set}
scrollbar .sb -orient horizontal -command {.c xview}
grid .c -row 4 -column 1
grid .sb -row 5 -column 1 -sticky ew
```

The canvas `xview` and `yview` subcommands are invoked by a scrollbar when it changes state (e.g., a user drags the slider). The scrollbar `set` command is invoked by the canvas when it changes state.

After creating the canvas and drawing the X and Y axes with the code above, the canvas widget would show the X axis from pixel 0 to 700, and the scrollbar would have a slider that extended from edge to edge.

By default, the canvas widget is set to scroll for the width of the canvas, i.e., not to scroll at all.

The `-scrollregion` option defines the portion of the canvas that can be scrolled into. The argument to the `-scrollregion` option is a list of the left, top, right, and bottom coordinates of the rectangle that can be scrolled into. Adding the command

```
.c configure -scrollregion {0 0 4000 500}
```

after drawing the axes causes the canvas to change its state, which invokes the scrollbar's `set` command to make the slider show that the leftmost portion of a larger window is being displayed.

In this case, we know the area of the canvas we need to scroll around in. If your application is creating objects without knowing the boundaries, the `bbox` command can be used to find the bounding rectangle.

Syntax: *`canvasName`* `bbox` *`tagOrId`*
`bbox`

Return the coordinates of a box that would enclose the item, or items with the same tag.

`tagOrId`

A tag or unique ID that identifies the item. If a tag is used, and multiple items share that tag, then the return is the bounding box that would cover all the items with that tag. The tag `all` can be used to return the bounding box for all items on a canvas.

The `bbox` command returns the bounding rectangle in the same format that the `-scrollregion` configuration option requires, so a command like:

```
$cvs configure -scrollregion [$cvs bbox all]
```

can be used to set the scrollregion for a canvas without tracking where graphic items have been placed.

The `xview` and `yview` commands can be used to bring newly placed objects into view.

Syntax: *`canvasName`* `xview moveto` *`fraction`*
`fraction`

The fraction of the scrollwindow to place to the left of the leftmost edge of the viewed window. A value of 0 will display the leftmost area of the scrollregion, while a value of 0.5 would not display the left half of a scrollregion.

As the Lander program runs, it will start drawing rockets to the right of the 700 pixels that are displayed by default. The user can scroll to the latest rocket, but it's friendlier if the application automatically scrolls to display the latest output. The application can display the latest rocket, and the previous 20 rockets (500 pixels) with this command:

```
.c xview moveto [expr { ($x-500.0) / 4000.0}]
```

The next step is to draw the rockets. The rockets are created with the `create polygon` subcommand. The `create polygon` command can accept an arbitrary number of X/Y pairs to define the nodes on the polygon. The syntax looks like:

Syntax: *`canvasName`* `create polygon` *`coord ?option value? coord`*

A list of X/Y pairs to define corners of the polygon.

`?option value?`

Keyword/Value pairs that define configuration options for this polygon. Options include:

`-fill` *`color`*

A color to fill the polygon.

`-outline` *`color`*

A color for the line outlining the polygon.

`-width` *`distance`*

A value for how wide to make the outline. By default this is a number of pixels, but it can also be defined in points, inches, millimeters, or centimeters.

The mainline FORTRAN code calls a Tcl procedure named `showState` to display the current lander status. The `showState` procedure is passed the time, height, speed, and remaining fuel of the rocket. The time and height define the X and Y coordinates, and the speed and remaining fuel can be scaled to define the height and width of the rocket.

With these values, the position of each node of the polygon describing the rocket can be calculated.

The command for performing arithmetic operations in Tcl is the `expr` command. The `expr` command takes an arithmetic expression as an argument and returns a numeric result. For most Tcl applications, commands this verbose are a bit unwieldy, but not a serious problem.

```
set y2 [expr {$y - ($speed / 10)}]
set wid [expr {2 + $fuel/150.0}]
set tall [expr abs($speed)/5.0]
```

However, for calculating each node on a polygon or line, the successive `expr` commands can make the application overly verbose and difficult to read.

Lars Hellstrom described an elegant solution to this on the Tclers' Wiki (http://wiki.tcl.tk/8389). Since any string can be the name of a Tcl procedure, we can define procedures named "+" and "-", to return simple arithmetic operations. The code to do this looks like this:

```
proc + {a b} {
    return [expr $a + $b]
}
proc - {a b} {
    return [expr $a - $b]
}
```

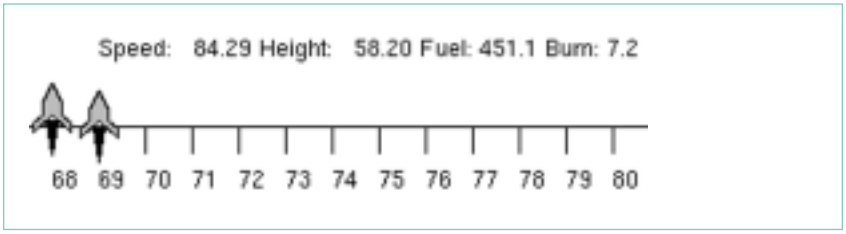
With this trick, describing the outline of the rocket looks like this:

```
set rocketID [.c create polygon $x $y \
    [+ $x $wid] [+ $y 10 ] \
    [+ $x $wid] [+ $y $tall] \
    [+ $x (5+$wid)] [+ $y ($tall+4)] \
    [+ $x (5+$wid)] [+ $y ($tall+9)] \
    [+ $x ($wid-5)] [+ $y $tall ] \
    [- $x ($wid-5)] [+ $y $tall ] \
    [- $x (5+$wid)] [+ $y ($tall+9)] \
    [- $x (5+$wid)] [+ $y ($tall+4)] \
    [- $x $wid] [+ $y $tall ] \
    [- $x $wid] [+ $y 10] \
    -fill $fill -outline black]
```

A failing of this display is that it doesn't show the exact height, speed, fuel burned at any given time. This failing can be solved with the canvas `bind` command. The `bind` command lets us put a binding for some action on a graphic item.

We can add a binding on each rocket so that when the rocket is clicked, it will display the speed, altitude, remaining fuel, and burn.

Clicking the rocket at X location 69 would generate this display:



The `bind` command links an event to a widget and a script. If the event occurs while the focus is on that widget, the script associated with that event will be evaluated. Each object on the canvas can also be bound to certain actions such as having a cursor pass over the object or a button clicked while the cursor is over the object.

Syntax: *canvasName* `bind` *tagOrID eventType script*

tagOrID

The tag or ID number of the canvas item to have this action bound to it.

eventType

The event to trigger this action. Events can be defined in one of three formats:

alphanumeric: A single printable (alphanumeric or punctuation) character defines a KeyPress event for that character.

<<virtualEvent>>: A virtual event defined by your script with the event command.

<modifier-type-detail>: This format precisely defines any event that can occur. The fields of an event descriptor are the X windows codes (e.g., Button1 or B1).

script

The script to evaluate when this event occurs and the cursor is over a canvas item with this tag or ID.

The line below shows a `bind` command that links the rocket that was just drawn to a button click event. When the left mouse button is clicked over the rocket, the Tcl interpreter will evaluate the procedure `showDetails` with an X and Y location and the speed, altitude, fuel, etc. The script can be any valid Tcl script. In this case, the `showDetails` procedure is a user-provided script that clears an old information line and then creates new text on the canvas.

```
.c bind $rocketID \
"showDetails $x [- $y 30] $speed $fuel $ht $burn"
```

Notice that the script is enclosed within quotes, not curly braces. Tcl will perform substitutions on lines enclosed within quotes, while the curly braces will force the substitution to be delayed until the script is evaluated.

In this case, we want the speed, fuel, etc., to be substituted when the rocket is drawn and the binding is created. If we needed to have the substitution done when the event occurred (for instance, if the diagram were a real-time monitor for our network), we'd enclose the script in curly braces, and the variables would be substituted when the object is clicked.

The one problem with a FORTRAN mainline linked to a Tcl/Tk library is that it requires that the user have Tcl/Tk installed in order to run the FORTRAN program. The next Tclsh Spot article will explain how to use TOBE to make a single, stand-alone executable that you could ship to a customer who had never heard of Tcl/Tk.