224

Geometry Managers

A Tk Graphic User Interface comprises a number of component elements known as widgets. Geometry managers allow the programmer to control the layout of the widgets within the windows, and to specify how the layout is to be changed should the widgets be of uneven size or if the window is resized.

Widgets are arranged in frames by geometry managers:

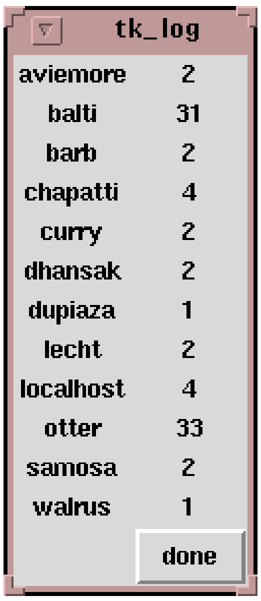
- The **`pack`** geometry manager packs the widgets in from one (or more) of the sides, with the widgets expanding to fill the available width or height if that's called for.
- The **`grid`** geometry manager creates a row and column layout rather like a table,
- For exact placement (by position, with specified size), you can use the **`place`** geometry manager. You'll probably only want to use this in a few special situations as it can become very long-winded.

All three window managers arrange widgets in frames, and in all three cases those frames can be other frames.

224.1 The grid geometry manager

Let's count the number of accesses from a particular client to a server computer, and lay out the results in a nice table, as shown in Figure 1006.

```
while {[set line [gets $fhandle]] >0} {
        set hitlist [split $line]
        set hostname [lindex $hitlist 0]
        if {[info exists counter($hostname)]} {
                incr counter($hostname)
        } else {
                set counter($hostname) 1
        }
}

foreach host [lsort [array names counter]] {

        label .host_$host -text $host
        label .count_$host -text $counter($host)
        grid .host_$host .count_$host

        }

label .final -text " "
button .last -text "done" -command exit
grid .final .last
```



```
aviemore    2
balti      31
barb        2
chapatti    4
curry       2
dhansak     2
dupiaza     1
lecht       2
localhost   4
otter      33
samosa      2
walrus      1
```

Figure 1006 Counting the number of accesses from a particular client to a server computer

Yes, that's it! Let's look at the Tk code in that:

`label .host_$host -text $host`

Create a label widget for each host. By using the name of a variable as part of the widget name, we're able to come up with as many widgets as we need all on simple variables - no need to switch to arrays which would give problems when we came to look at the corresponding command.

`label .count_$host -text $counter($host)`

That's another label which is also host-name-related, this time holding the count of the number of hits from the particular computer.

After we've created each row of widgets, we'll pass them across to the grid layout manager so that it can (in due course) display them:

`grid .host_$host .count_$host`

The code above is part of a loop in our program, once around for each host. At the end, we want to add a final line which has an exit button on the right.

Here's the code for that final row of buttons:

```
label .final -text " "
button .last -text "done" -command exit
grid .final .last
```

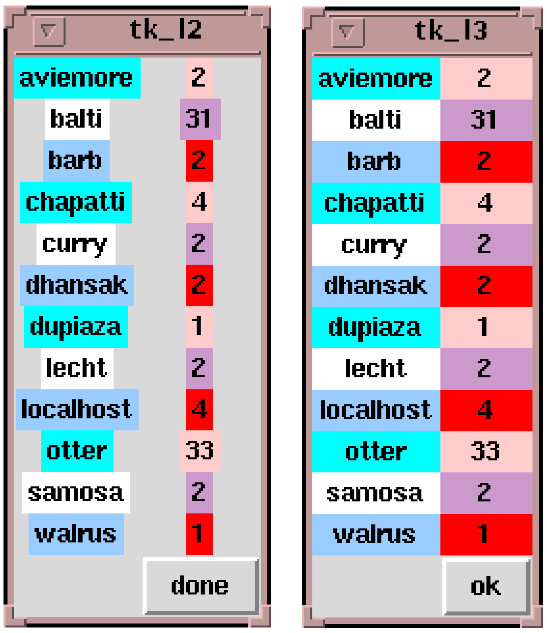


Figure 1007 Taking the last example and colouring each of the cells (left) and the same example with the colours filled in by -sticky (right)

Sizing and filling cells

Let's take that last example and colour each of our cells so that we can see how our layout was put together:

Why so much grey space? Because each column must have the same width on all the rows. The host computer "localhost" is the widest name, and the "done" button at the bottom has controlled the width of the second column.

The host names and number-of-hits text all varied in size when drawn as text, so we have a messy-looking display - most of them are in the middle of a sea of grey in their grid cells.

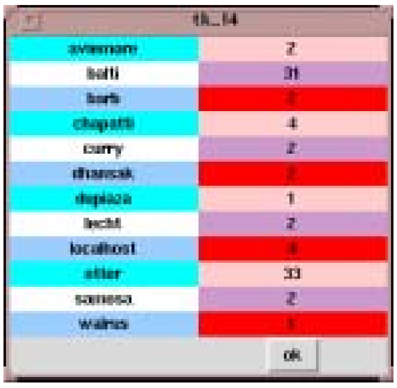
With a grid widget, we can specify "sticky" parameters to indicate which side of the cell we want the widget to go to:

`N`	Top (North)
`E`	Right (East)
`W`	Left (West)
`S`	Bottom (South)

and we can specify multiple letters too, right up to "news" to stick the widget to all four sides.

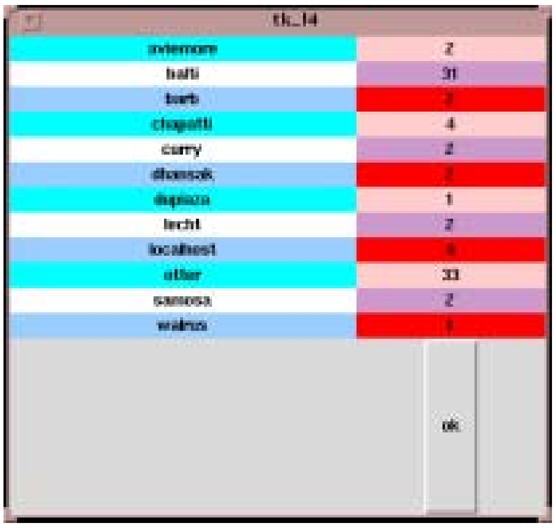
Here's the complete code:

```
#!/usr/bin/wish

# Count accesses from a series of computers to a server

set fhandle [open seal_log r]

while {[set line [gets $fhandle]] >0} {
        set hitlist [split $line]
        set hostname [lindex $hitlist 0]
        if {[info exists counter($hostname)]} {
                incr counter($hostname)
        } else {
                set counter($hostname) 1
        }
}

set col 0

foreach host [lsort [array names counter]] {

        incr col
        set col [expr $col %3]
        set colname [lindex "lightblue Cyan White" $col]
        set colname2 [lindex "Red Pink Plum" $col]

        label .host_$host -text $host -bg $colname
        label .count_$host -text $counter($host) -bg $colname2
        grid .host_$host .count_$host -sticky news

        }

label .final -text " "
button .last -text "ok" -command exit
grid .final .last
```



tk_14

aviemore	2
balti	31
barb	[illegible]
chapatti	4
curry	2
dhansak	[illegible]
dopiaza	1
lecht	2
localhost	[illegible]
other	33
samosa	2
walrus	[illegible]



tk_14

aviemore	2
balti	31
barb	[illegible]
chapatti	4
curry	2
dhansak	[illegible]
dopiaza	1
lecht	2
localhost	[illegible]
other	33
samosa	2
walrus	[illegible]

Figure 1008 Starting the application (left) and resizing (below)

Managing rows and columns

The **grid** command has **rowconfigure** and **columnconfigure** amongst its options, which lets you make settings for whole rows or columns.

-pad	lets you add padding to each cell in the row or column
-minsize	lets you set a minimum size
-weight	allows you to set a growth "weight" to rows and columns with a positive value grow in proportion to those values when the window is expanded.

grid size returns a list containing the number of rows and columns in your grid.

Let's adjust the display and resize behaviour of our host access count.

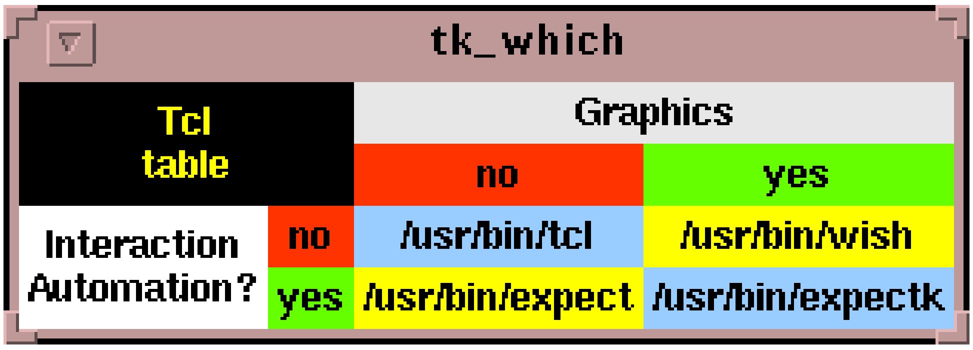
Here's the changed code, all at the end of the application

```
grid columnconfigure . {0 1} -minsize 150
grid columnconfigure . 0  -weight 1
set rowsandcols [grid size .]

label .final -text " "
button .last -text "ok" -command exit
grid .final .last -sticky ns

grid rowconfigure . [lindex $rowsandcols 1] -weight 1
```

The only change to effect the initial display is the **-minsize 150** on each column. Note how the **columnconfigure** command can specify a list of column numbers all at the same time. In essence, we've used this capability to call up column widths which are (initially) identical. As they're a minimum rather than a fixed size, we would be in trouble if the contents of a cell turned out to be over 150 pixels.

The second display is affected by the **-weight** setting on the first column. This setting tells the grid manager to put otherwise-unallocated space into that column as opposed to any other.

Similarly, the **-weight 1** on the **rowconfigure** command asks for any spare vertical space to be put into the last row (the "ok" button). We didn't bother to keep count of the rows; we used **grid size** to find out, and we did so before we added the last row to save us the bother of subtracting 1 from the number of rows to get the highest index number!



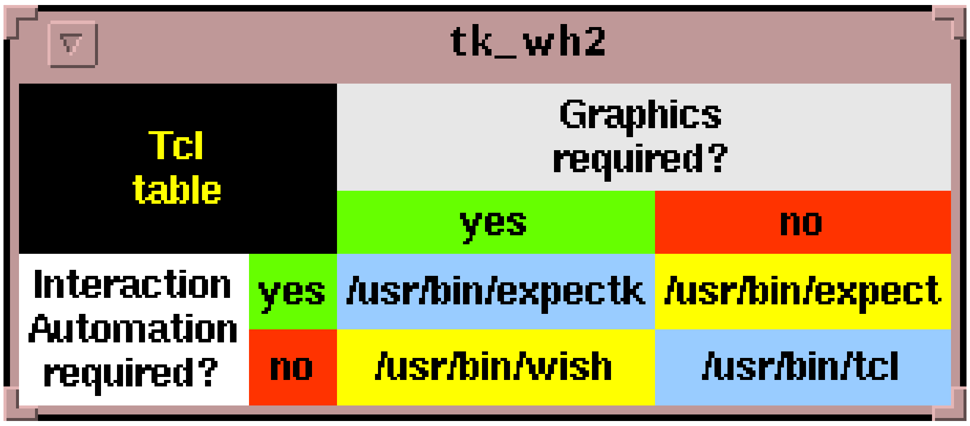
Tcl table		Graphics	
		no	yes
Interaction Automation?	no	/usr/bin/tcl	/usr/bin/wish
	yes	/usr/bin/expect	/usr/bin/expectk

Figure 1009 Specifying rows and columns

Cells that span several rows or columns

On a grid, you can specify explicit **-row** and **-column** options to place information into specified rows and columns, and you can specify **-rowspan** and **-columnspan** to give the width of the widget in row and columns.

```
#!/usr/bin/wish

# A table of tcl applications / programs

label .tl -text "Tcl\ntable" -bg black -fg yellow

label .graphics -text "Graphics" -bg bisque
label .interaction -text "Interaction\nAutomation?" -bg honeydew

label .y1 -text "yes" -bg lawngreen
label .y2 -text "yes" -bg lawngreen
label .n1 -text "no" -bg orangered
label .n2 -text "no" -bg orangered

label .nn -text "/usr/bin/tcl" -bg skyblue
label .yy -text "/usr/bin/expectk" -bg skyblue
label .ny -text "/usr/bin/wish" -bg yellow
label .yn -text "/usr/bin/expect" -bg yellow

grid .tl -row 0 -column 0 -rowspan 2 -columnspan 2 -sticky news

grid .graphics -row 0 -column 2 -columnspan 2 -sticky news
grid .interaction -row 2 -column 0 -rowspan 2 -sticky news

grid .n1 -row 1 -column 2 -sticky news
grid .y1 -row 1 -column 3 -sticky news
grid .n2 -row 2 -column 1 -sticky news
grid .y2 -row 3 -column 1 -sticky news

grid .nn -row 2 -column 2 -sticky news
grid .yy -row 3 -column 3 -sticky news
grid .yn -row 3 -column 2 -sticky news
grid .ny -row 2 -column 3 -sticky news
```



		Graphics required?	
Tcl table		yes	no
Interaction Automation required?	yes	/usr/bin/expectk	/usr/bin/expect
	no	/usr/bin/wish	/usr/bin/tcl

Figure 1010 A similar table using alternative grid commands.

Yes, we did plan that out on a scrap of paper first!

There is an alternative using the **grid** command to specify multiple widgets per row, adding in the special characters

^	to skip a column
-	to skip a row

You can also leave a cell blank using "x"

```
#!/usr/bin/wish

# A table of tcl applications / programs

label .tl -text "Tcl\ntable" -bg black -fg yellow

label .graphics -text "Graphics\nrequired?" -bg bisque
label .interaction -text "Interaction\nAutomation\nrequired?" -bg
honeydew

label .y1 -text "yes" -bg lawngreen
label .y2 -text "yes" -bg lawngreen
label .n1 -text "no" -bg orangered
label .n2 -text "no" -bg orangered

label .nn -text "/usr/bin/tcl" -bg skyblue
label .yy -text "/usr/bin/expectk" -bg skyblue
label .ny -text "/usr/bin/wish" -bg yellow
label .yn -text "/usr/bin/expect" -bg yellow

grid .tl - .graphics - -sticky news
grid ^ ^ .y1 .n1 -sticky news
grid .interaction .y2 .yy .yn -sticky news
grid ^ .n2 .ny .nn -sticky news
```

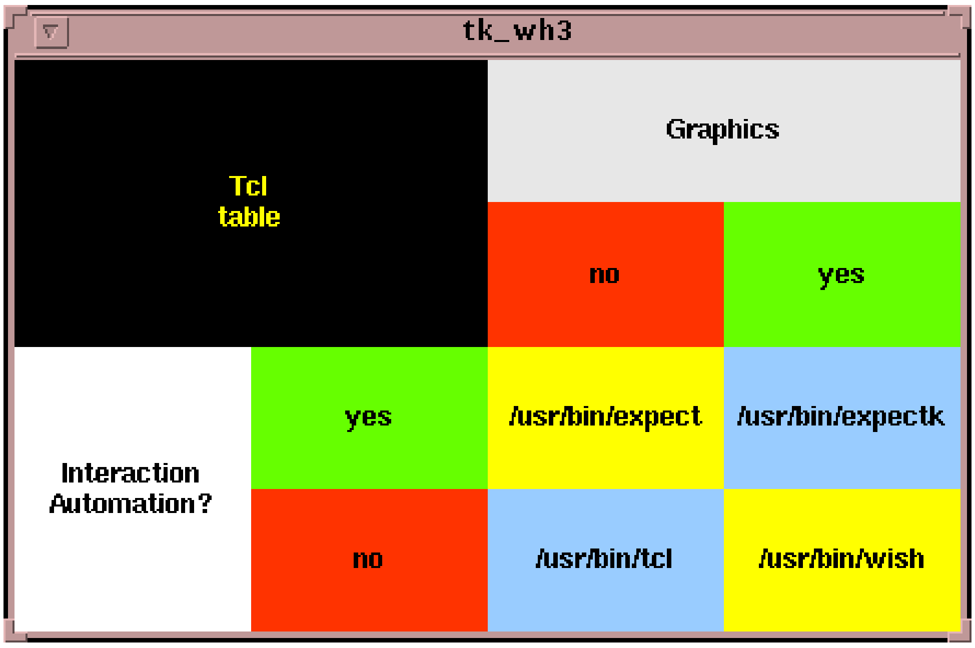


Figure 1011 Using the place geometry manager.

224.2 Placing geometry

If you want to explicitly place widgets in your frame, you can do so using the **place** geometry manager. The code is precise, but long.

```
#!/usr/bin/wish

# A table of tcl applications / programs

label .tl -text "Tcl\ntable" -bg black -fg yellow

label .graphics -text "Graphics" -bg bisque
label .interaction -text "Interaction\nAutomation?" -bg honeydew

label .y1 -text "yes" -bg lawngreen
label .y2 -text "yes" -bg lawngreen
label .n1 -text "no" -bg orangered
label .n2 -text "no" -bg orangered

label .nn -text "/usr/bin/tcl" -bg skyblue
label .yy -text "/usr/bin/expectk" -bg skyblue
label .ny -text "/usr/bin/wish" -bg yellow
label .yn -text "/usr/bin/expect" -bg yellow

place .tl -in . -x 0 -y 0 -relwidth 0.5 -relheight 0.5
place .graphics -in . -relx 0.5 -y 0 -relwidth 0.5 -relheight 0.25
place .interaction -in . -x 0 -rely 0.5 -relwidth 0.25 -relheight 0.5

place .n1 -in . -relx 0.5 -rely 0.25 -relwidth 0.25 -relheight 0.25
place .y1 -in . -relx 0.75 -rely 0.25 -relwidth 0.25 -relheight 0.25
place .y2 -in . -relx 0.25 -rely 0.5 -relwidth 0.25 -relheight 0.25
place .n2 -in . -relx 0.25 -rely 0.75 -relwidth 0.25 -relheight 0.25

place .nn -in . -relx 0.5 -rely 0.75 -relwidth 0.25 -relheight 0.25
place .yy -in . -relx 0.75 -rely 0.5 -relwidth 0.25 -relheight 0.25
place .yn -in . -relx 0.5 -rely 0.5 -relwidth 0.25 -relheight 0.25
place .ny -in . -relx 0.75 -rely 0.75 -relwidth 0.25 -relheight 0.25
```

Positions may be specified using absolute coordinates within the window (`-x` and `-y`). We've only used these for the top left (`0,0`) as relative positioning. A proportion of the way across the window, using `-relx` and `-rely` is much more flexible.

Similarly, widths may be specified using `-width` and `-height` if you're specifying pixels, or `-relwidth` and `-relheight` if you're specifying proportions.

Let's interpret one example:

```
place .interaction -in . -x 0 -rely 0.5 -relwidth 0.25 -relheight 0.5
```

The widget called **`.interaction`** is to be placed in the root frame, inset 0 pixels from the left hand side, and half way down the window. The **`place`** manager assumes that you're telling it where to place the top left of the widget (unless you've also specified an anchor option). The widget will be a quarter of the width of the window, and half its height.

Exercise