A Primer on the R-Tcl/Tk Package

by Peter Dalgaard

Introduction

Tcl/Tk is a combination of a scripting language and a toolkit for graphical user interfaces. Since version 1.1.0, R has had a **tcltk** package to access the Tk toolkit, replacing Tcl code with R function calls (Dalgaard, 2001). There are still some design problems in it, but it is quite useful already in its current state.

This paper intends to get you started with the R-Tcl/Tk interface. Tcl/Tk is a large package, so it is only possible to explain the basic concepts here.

The presentation here is based on the X11/Unix version of R. The **tcltk** package also works on Windows. It (currently) does not work with Mac OS Classic but it does work with OS X. It is only the Linux variants that come with Tcl/Tk; on other systems some footwork will be necessary to get it installed.

Widgets

A *widget* is a GUI element. Tk comes with a selection of basic widgets: text editing windows, sliders, text entry fields, buttons, labels, menus, listboxes, and a canvas for drawing graphics. These can be combined to form more complex GUI applications.

Let us look at a trivial example:

```
library(tcltk)
tt <- tktoplevel()
lbl <- tklabel(tt, text="Hello, World!")
tkpack(lbl)
```

This will cause a window to be displayed containing the "Hello, World!" message. Notice the overall structure of creating a container widget and a child widget which is positioned in the container using a *geometry manager* (`tkpack`). Several widgets can be packed into the same container, which is the key to constructing complex applications out of elementary building blocks. For instance, we can add an "OK" button with

```
but <- tkbutton(tt, text="OK")
tkpack(but)
```

The window now looks as in Figure 1. You can press the button, but no action has been specified for it.

Figure 1: Window with label widget and button widget.

The title of the window is "1" by default. To set a different title, use

```
tktitle(tt) <- "My window"
```

Geometry managers

A geometry manager controls the placement of *slave* widgets within a *master* widget. Three different geometry managers are available in Tcl/Tk. The simplest one is called the *placer* and is almost never used. The others are the *packer* and the *grid* manager.

The packer "packs widgets in order around edges of cavity". Notice that there is a *packing order* and a *packing direction*.

In the example, you saw that the window autosized to hold the button widget when it was added. If you enlarge the window manually, you will see that the slave widgets are placed centered against the top edge. If you shrink it, you will see that the last packed item (the button) will disappear first. (Manual resizing disables autosizing. You can reenable it with `tkwm.geometry(tt,"")`.)

Widgets can be packed against other sides as well. A widget along the top or bottom is allocated a *parcel* just high enough to contain the widget, but occupying as much of the width of the container as possible, whereas widgets along the sides get a parcel of maximal height, but just wide enough to contain it. The following code may be illustrative (Figure 2):

```
tkdestroy(tt) # get rid of old example
tt <- tktoplevel()
edge <- c("top","right","bottom","left")
buttons <- lapply(1:4,
  function(i) tkbutton(tt, text=edge[i]))
for ( i in 1:4 )
  tkpack(buttons[[i]], side=edge[i],
         fill="both")
```

Figure 2: Geometry management by the packer

The `fill` argument causes each button to occupy its entire parcel. Similarly `expand=TRUE` causes parcels to increase in width or height (depending on the packing direction) to take up remaining space in the container. This occurs *after* allotment of parcels; in the above example only "left" can expand.

If an object does not fill its parcel, it needs to be *anchored*. The `anchor` argument to `tkpack` can be set to compass-style values like `"n"` or `"sw"` for placement in the middle top, respectively bottom left. The default is `"center"`.

It is useful at this point to consider what the packer algorithm implies for some typical layouts:

Simple vertical or horizontal stacking is of course trivial, you just keep packing against the same side.

For a text widget with a scrollbar on the side, you want `fill="y"` for the scrollbar and `fill="both"` and `expand=TRUE` for the text area. The scrollbar should be packed before the text widget so that the latter shrinks first.

A text widget with scrollbar and a row of buttons beneath it? You cannot do that with the packer algorithm! This is where *frames* come in. These are containers for further widgets with separate geometry management. So you pack the buttons inside a frame, pack the frame against the bottom, then pack the scrollbar and text widget.

The combination of the packer and frames gives a lot of flexibility in creating GUI layouts. However, some things are tricky, notably lining widgets up both vertically and horizontally.

Suppose you want multiple lines, each containing an entry widget preceded by a label. With the packer there is no simple way to keep the beginning of the entry fields lined up.

Enter the *grid manager*. As the name suggests it lays out widgets in rows and columns. Using this manager the labeled-entry problem could be solved as follows (Figure 3

```
t2 <- tktoplevel()
heading <- tklabel(t2, text="Registration form")
l.name <- tklabel(t2, text="Name")
l.age  <- tklabel(t2, text="Age")
e.name <- tkentry(t2, width=30)
e.age  <- tkentry(t2, width=3)

tkgrid(heading, columnspan=2)
tkgrid(l.name, e.name)
tkgrid(l.age, e.age)
tkgrid.configure(e.name, e.age, sticky="w")
tkgrid.configure(l.name, l.age, sticky="e")
```

Figure 3: A registration form using the grid manager

With the grid manager it is most convenient to specify a full row at the time, although options let you do it otherwise. The `columnspan` argument joins grid cells horizontally. The `sticky` argument works somewhat like anchoring in the packer. The value can be any subset of n, s, e, and w and specifies that the widget should stick to the specified sides of the cell. If it contains opposite sides, e.g. both n and s, the widget stretches to fill the space.

You can mix the geometry managers, although not in the same master frame. Some parts of an application may be best handled by the packer and others by the grid manager.

Communication with widgets

We need a way to get data from widgets to and from R, and a way to make things happen in response to widget events. There are two general patterns for this, namely *control variables* and *callbacks*.

Control variables associate the state of some aspect of a widget with a variable in Tcl. These Tcl variables can be accessed from R as (pseudo-)components of the `tclvar` object. So we could control the name entry field of the example above with

```
tkconfigure(e.name, textvariable="foo")
tclvar$foo <- "Hello, World"
```

and conversely any change to the content of the entry field is reflected in `tclvar$foo` *This mechanism is not optimal and will likely change in future versions of R!*

Control variables are also used by checkbuttons, radiobuttons, and scales. Radiobutton widgets allow a `value` argument so that the button lights up when the control variable has that value, and the variable is given that value when the radiobutton is clicked. A checkbutton will indicate whether its control variable is 0 (`FALSE`) or not.

Callbacks are functions that are linked to GUI events. Callbacks are often set up using arguments named `command`.

For a simple example consider

```
t3 <- tktoplevel()
b <- tkbutton(t3, text = "Don't press me!")
```

```
tkpack(b)
change.text <- function() {
  cat("OW!\n")
  tkconfigure(b, text = "Don't press me again!")
}
tkconfigure(b, command = change.text)
```

This callback function doesn't take any arguments, but others do. There are two ways to take account of this, depending on whether the callback is actively soliciting information or not. An example of the latter is the scrollbar protocol as exemplified below

```
t4 <- tktoplevel()
txt <- tktext(t4)
scr <- tkscrollbar(t4,
  command=function(...) tkyview(txt,...))
tkconfigure(txt,
  yscrollcommand=function(...) tkset(scr,...))
tkpack(scr, side="right", fill="y")
tkpack(txt, fill="both", expand=TRUE)
```

This sets up a bidirectional link: Manipulating the scrollbar changes the view of the text widget and vice versa. Some care is taken not to add a callback that refers to a widget before the widget exists.

We don't need to care what the arguments to the callbacks are, only to pass them through to `tkyview` and `tkset` respectively. In fact the arguments to `tkyview` will be different depending on which part of the scrollbar is engaged.

In Tcl, you can define a callback command as `myproc %x %y` and `myproc` will be invoked with the pointer coordinates as arguments. There are several other "percent codes". The parallel effect is obtained in R by defining the callback with specific formal arguments. From the `tkcanvas` demo:

```
plotMove <- function(x, y) {
  x <- as.numeric(x)
  y <- as.numeric(y)
  tkmove(canvas, "selected",
         x - lastX, y - lastY)
  lastX <<- x
  lastY <<- y
}
tkbind(canvas, "<B1-Motion>", plotMove)
```

The coordinates are passed as text strings, requiring the use of `as.numeric`.

Events and bindings

The previous example showed a binding of a callback to a windows event, containing the *event pattern* `<B1-Motion>` — mouse movement with Button 1 pressed.

An event pattern is a sequence of fields separated by hyphens and enclosed in <>. There are three kinds of fields, *modifier*, *type*, and *detail*, in that order. There can be several modifier fields. A generic example is `<Control-Alt-Key-c>`, where `Control` and `Alt` are modifiers, `Key` is the event type, and `c` is the detail. If `c` is left out any key matches. The `Key` part can be omitted when there's a character detail field. Similarly, a numeric detail field is assumed to refer to a button press event (notice that `<Key-1>` is different from `<1>`).

Callbacks are associated with events using `tkbind`, or sometimes `tktag.bind` or `tkitembind`.

Text widgets

The text widget in Tk embodies the functionality of a basic text editor, allowing you to enter and edit text, move around in the text with cursor control keys, and mark out sections of text for cut-and-paste operations. Here, we shall see how to add or delete text and how to extract the text of pieces thereof. These methods revolve around *indices*, *tags*, and *marks*.

A simple index is of the form `line.char` where `line` is the line number and `char` is the character position within the line. In addition there are special indices like `end` for the end of the text.

Tags provide a way of referring to parts of the text. The part of the text that has been marked as selected is tagged `sel`. Any tag can be used for indexing using the notation `tag.first` and `tag.last`.

Marks are somewhat like tags, but provide names for locations in the text rather than specific characters. The special mark `insert` controls and records the position of the insertion cursor.

To extract the entire content of a text widget, you say

```
X <- tkget(txt, "0.0", "end")
```

Notice that you have to give `0.0` as a character string, not as a number. Notice also that the result of `tkget` is a single long character string; you may wish to convert it to a vector of strings (one element per line) using `strsplit(X, "\n")`.

In a similar fashion, you can extract the selected part of the text with

```
X <- tkget(txt, "sel.first", "sel.last")
```

However, there is a pitfall: If there is no selection, it causes an error. You can safeguard against this by checking that

```
tktag.ranges(txt, "sel") != ""
```

Inserting text at (say) the end of a file is done with

```
tkinsert(txt, "end", string)
```

The string needs to be a single string just like the one obtained from `tkget`. If you want to insert an entire character array, you will need to do something along the lines of

```
tkinsert(txt, "end",
         paste(deparse(ls), collapse="\n"))
```

You can set the insertion cursor to the top of the text with

```
tkmark.set(txt, "insert", "0.0")
tksee(txt, "insert")
```

The `tksee` function ensures that a given index is visible.

An insertion leaves the insertion mark in place, but when it takes place exactly at the mark it is ambiguous whether to insert before or after the mark. This is controllable via *mark gravity*. The default is "right" (insert before mark) but it can be changed with

```
tkmark.gravity(txt, "insert", "left")
```

Creating menus

Tk menus are independent widgets. They can be used as popup menus, but more often they attach to the menu bar of a toplevel window, a menubutton, or a *cascade entry* in a higher-level menu.

Menus are created in several steps. First you setup the menu with `tkmenu`, then you add items with `tkadd`. There are so many possible options for a menu item that this is a more practicable approach.

Menu items come in various flavours. A *command entry* is like a button widget and invokes a callback function. *Cascade entries* invoke secondary menus. *Checkbutton* and *radiobutton* entries act like the corresponding widgets and are used for optional selections and switches. Special entries include *separators* which are simply non-active dividing lines and *tear-offs* which are special entries that you can click to detach the menu from its parent. The latter are on by default but can be turned off by passing `tearoff=FALSE` to `tkmenu`.

Here is a simple example of a menubutton with a menu which contains three radiobutton entries:

```
tclvar$color<-"blue"
tt <- tktoplevel()
tkpack(mb <- tkmenubutton(tt, text="Color"))
m <- tkmenu(mb)
tkconfigure(mb,menu=m)
for ( i in c("red", "blue", "green"))
  tkadd(m, "radio", label=i, variable="color",
        value=i)
```

A simple application: Scripting widgets

The following code is a sketch of a scripting widget (Figure 4. The widget can be used to edit multiple lines of code and submit them for execution. It can load and save files using `tk_getOpenFile` and `tk_getSaveFile`. For simplicity, the code is executed with `parse` and `eval`.

Notice that `tkcmd` is used to call Tcl commands that have no direct R counterpart. Future versions of the **tcltk** package may define functions `tkclose`, etc.

Tcl has file functions that by and large do the same as R connections do although they tend to work a little better with other Tcl functions.

You may want to experiment with the code to add features. Consider e.g. adding an Exit menu item, or binding a pop-up menu to Button 3.

```
tkscript <- function() {
  wfile <- ""
  tt <- tktoplevel()
  txt <- tktext(tt, height=10)
  tkpack(txt)
  save <- function() {
    file <- tkcmd("tk_getSaveFile",
      initialfile=tkcmd("file", "tail", wfile),
      initialdir=tkcmd("file", "dirname", wfile))
    if (!length(file)) return()
    chn <- tkcmd("open", file, "w")
    tkcmd("puts", chn, tkget(txt,"0.0","end"))
    tkcmd("close", chn)
    wfile <<- file
  }
  load <- function() {
    file <- tkcmd("tk_getOpenFile")
    if (!length(file)) return()
    chn <- tkcmd("open", file, "r")
    tkinsert(txt, "0.0", tkcmd("read", chn))
    tkcmd("close", chn)
    wfile <<- file
  }
  run <- function() {
    code <- tkget(txt,"0.0","end")
    e <- try(parse(text=code))
    if (inherits(e, "try-error")) {
      tkcmd("tk_messageBox",
            message="Syntax error",
            icon="error")
      return()
    }
    cat("Executing from script window:",
        "-----", code, "result:",  sep="\n")
    print(eval(e))
  }
  topMenu <- tkmenu(tt)
  tkconfigure(tt, menu=topMenu)
  fileMenu <- tkmenu(topMenu, tearoff=FALSE)
  tkadd(fileMenu, "command", label="Load",
        command=load)
  tkadd(fileMenu, "command", label="Save",
        command=save)
  tkadd(topMenu, "cascade", label="File",
        menu=fileMenu)
  tkadd(topMenu, "command", label="Run",
        command=run)
}
```

Further information

Some further coding examples are available in the demos of the **tcltk** package.

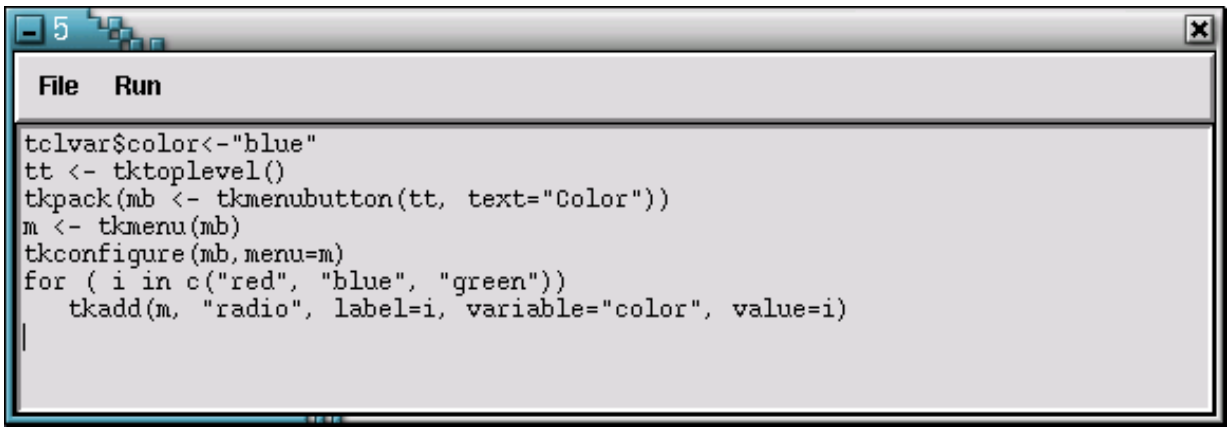

Figure 4: A simple scripting widget.

Most of the functions in the **tcltk** package are really just a thin layer covering an underlying Tcl command. Converting all the Tcl/Tk documentation for R is a daunting task, so you have to make do with the help for Tcl/Tk itself. This is fairly easy once you get the hang of some simple translation rules.

For the `tkbutton` function, you would look at the help for `button`. The R functions add a `tk` prefix to avoid name clashes. The `button` command in Tcl/Tk has a `-text` argument followed by the text string to display. Such options are replaced in the R counterpart by named arguments like `text="B1"`. The argument value is a string, but logical or numerical values, as well as (callback) functions are automatically converted.

When translating commands, there are a couple of special rules which are briefly outlined below.

One general difference is that Tcl encodes the widget hierarchy in the name of the widgets so that widget `.a` has subwidgets `.a.b` and `.a.c`, etc. This is impractical in R so instead of Tcl's

```
button .a.b -text foo
```

we specify the parent directly in the widget creation call

```
but <- tkbutton(parent, text="foo")
```

This pattern is used for all commands that create widgets. Another difference is that Tcl has *widget commands* like

```
.a.b configure -text fum
```

which in R is replaced by a command acting on a widget

```
tkconfigure(but, text="fum")
```

Some widget commands have subcommands as in

```
.a.b selection clear 0 end
```

which are turned into separate functions

```
tkselection.clear(lb, 0, "end")
```

In a few cases, the translation rules create ambiguities — for instance there is both a general `bind` command and a `bind` widget command for canvases. This has been resolved by making the widget commands `tkitembind`.

There is quite a large literature on Tcl and Tk. A well-reputed book is Welch (2000). A smaller reference item is Raines and Tranter (1999), although it is mostly a paper copy of online information. The main web site is at `http://tcl.activestate.com/`.

The useful `tclhelp` program comes with the TclX package. There is also a nice widget demo in the Tk distribution.

Bibliography

Peter Dalgaard. The R-Tcl/Tk interface. In Kurt Hornik and Fritz Leisch, editors, *Proceedings of the 2nd International Workshop on Distributed Statistical Computing, March 15-17, 2001, Technische Universität Wien, Vienna, Austria*, 2001. URL `http://www.ci.tuwien.ac.at/Conferences/DSC-2001/Proceedings/`. ISSN 1609-395X. 27

Paul Raines and Jeff Tranter. *Tcl/Tk in a Nutshell*. O'Reilly, 1999. 31

Brent B. Welch. *Practical Programming in Tcl and Tk*. Prentice-Hall PTR, New Jersey, 3rd edition, 2000. 31

Peter Dalgaard
University of Copenhagen, Denmark
`P.Dalgaard@biostat.ku.dk`