

## Applying the text widget

Arjen Markus<sup>1</sup>  
WL | Delft Hydraulics  
PO Box 177  
2600 MH Delft  
The Netherlands

### Abstract

The text widget is one of the most ingenious widgets available in Tk. It can be used for many other things than merely displaying multiline text by exploiting the properties of tags, embedded windows and event bindings. This can lead to very different applications.

To illustrate the wide variety, a number of them will be described: ranging from the display of coloured text to input forms that use the text widget as a specialised geometry manager.

## Introduction

The text widget is one of the two most sophisticated and powerful tools found in Tk. The aim of this paper is to show how it can be used in a wide variety of applications – from a simple display of formatted and decorated text to a specialised geometry manager.

The capabilities of the text widget that make this possible are:

- Its ability to lay out (or hide) text with different fonts and sizes
- The support for event bindings
- The possibility to incorporate other widgets in between the text
- Its mechanism of tags to prescribe the visual properties of the text

Each application will be discussed separately, with emphasis on how the above capabilities are exploited. Most have already appeared on the Tcl'ers Wiki – see the references. Finally, some things are mentioned that can not (easily) be done with the text widget, as even this marvellous tool has limitations.

## A simple slide show

Presentation graphics in its most basic form involves:

- Presenting a limited number of lines of text on the screen (slides)
- Formatting the text using bullet lists, a centred title, perhaps a fixed-width font to display program code or other features
- Navigating through the slides via the keyboard or mouse

The first item means that the text widget will contain a few lines only, no need for scroll bars. But making it fill the screen is desirable:

```
proc maximizedTopLevel { widget } {  
    #  
    # Calculate the screen size and therefore the window's size  
    #  
    set width [winfo screenwidth .]
```

---

<sup>1</sup> E-mail address: arjen.markus@wldelft.nl

```

set height [winfo screenheight .]

toplevel $widget
wm overriddenirect $widget 1

#
# Hand the geometry off to the window manager
#
wm geometry $widget ${width}x${height}+0+0
wm focusmodel $widget active
focus -force $widget
}

```

Text formatting is relatively easy: you introduce a tag with the right configuration:

```

font create Text -family Helvetica -size 24 -weight normal

$textwindow tag configure bullet -justify left -font Text \
-foreground $fgcol -wrap word -lmargin1 1c \
-tabs "1.5c center 2c left" -lmargin1 1c -lmargin2 2c

```

which gives us indented text, useable in a bullet list:

```

$textwindow insert end "\t*\t$text" bullet

```

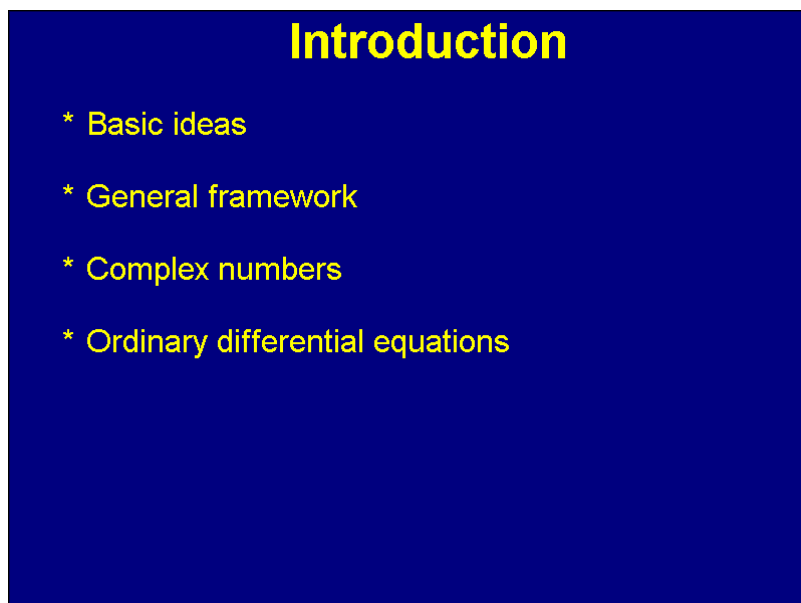
(the tabs are necessary to take advantage of the indentation)

For this particular application I used a simple scheme inspired by the Wiki's formatting rules to parse the input. That way:

Introduction

- \* Basic ideas
- \* General framework
- \* Complex numbers
- \* Ordinary differential equations

is turned into a slide as shown in the figure below.



What is left to discuss are the event bindings, such as the up and down arrow keys display the previous and next slides

```
bind $textwindow <KeyPress-Down> {displayNewSlide 1}
bind $textwindow <KeyPress-Up> {displayNewSlide -1}
```

Several special formatting commands are also implemented, so that it is possible to add an image and a push button to invoke an external command (for details: see the source code at the Wiki-page).

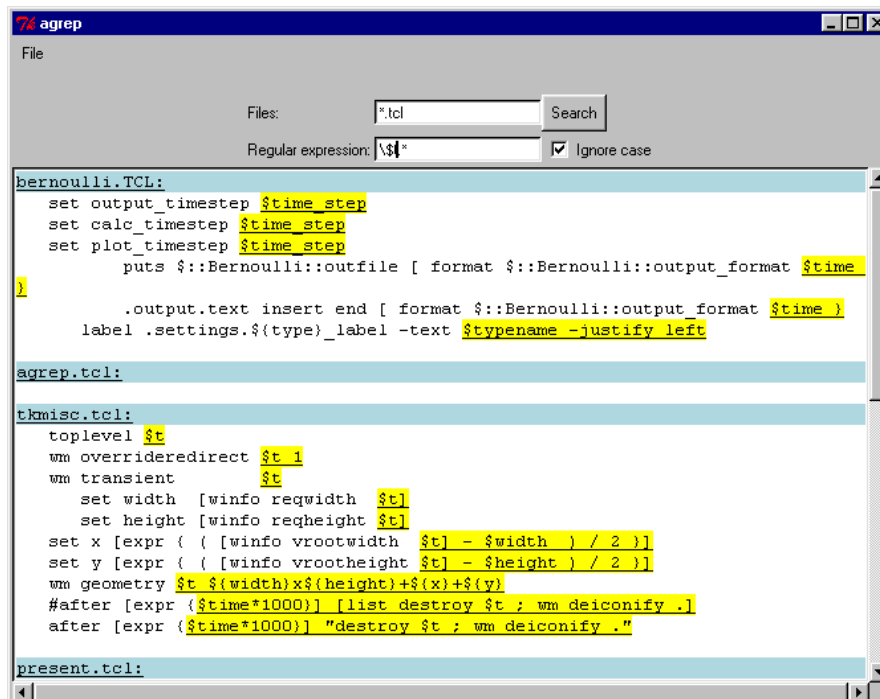
The aim was to create a basic but useable presentation tool and I succeeded in that with a mere 300 lines of code – including the comments.

## Searching text with regular expressions

The principle of the UNIX *grep* utility is simple: look for lines that satisfy a certain regular expression in any files whose names match a certain pattern. All of this is well within the reach of Tcl:

```
while { [gets $infile line] >= 0 } {
    if { $ignore_case } {
        set match [regexp -nocase -indices $pattern $line indices]
    } else {
        set match [regexp -indices $pattern $line indices]
    }
    if { $match } {
        puts $line
    }
}
```

But to make it a bit more useful, on Windows scrolling in a DOS box is a nuisance, I added a little user-interface (see the figure below). The output is shown in a text widget and the lines are shown such a way that the (first) substring that matches the regular expression is clear. The colours are obtained via specifically configured tags.



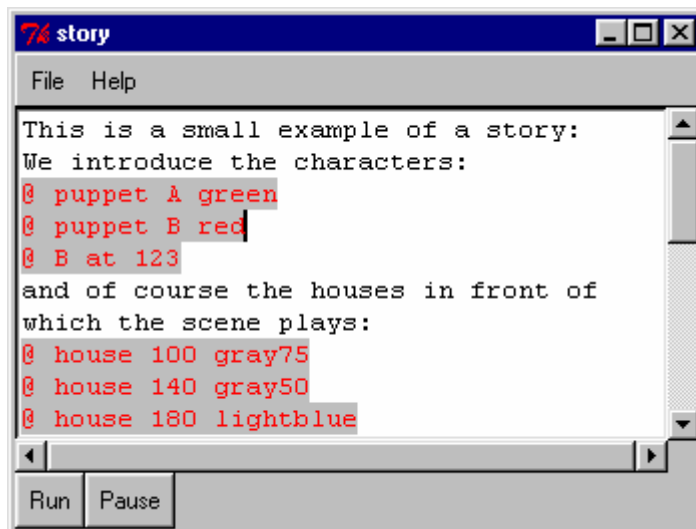
A surprising side effect was that you can use it view the contents of a file by specifying an empty regular expression – two applications, portable and all, for the price of one.

## Programmed story telling

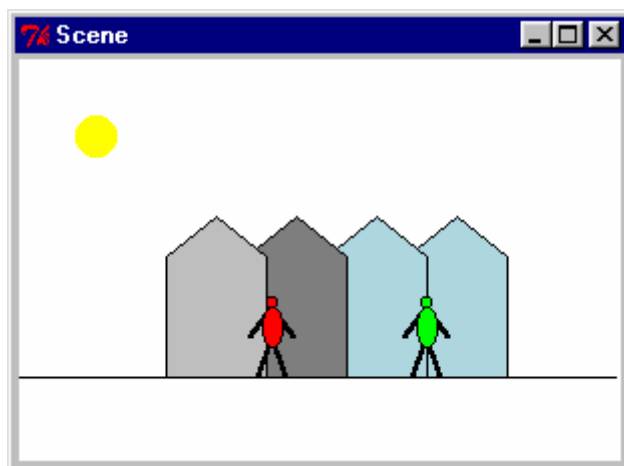
Literature programming, that is mixing program code and detailed explanations or documentation in such a way that you can extract the program's source and the documentation from a single source, was discussed recently on the Tcler's Wiki. It is a very appealing idea and with the way the Wiki works, we have almost achieved this. But what about a complement to this? Why not have program code support the documentation? You might call it "programmed story telling".

The idea: you write down the text and mix in some commands that can visualise whatever the text is about. Think of describing the classical geometrical construction of a regular polygon ... Words alone would be very abstract. If you add commands that actually draw the circle, the auxiliary arcs and the line segments, it will all become much clearer – especially when you see it happen.

Of course, this requires a mathematically inclined audience or audience that at least could be expected to have some such sympathies. So instead of arcs and line segments, let us manipulate simple puppets.



```
7% story
File Help
This is a small example of a story:
We introduce the characters:
@ puppet A green
@ puppet B red
@ B at 123
and of course the houses in front of
which the scene plays:
@ house 100 gray75
@ house 140 gray50
@ house 180 lightblue
Run Pause
```



Again the main widget for this application is a text widget. It gets one special binding and a single tag:

```
bind $widget <Key-Return> "+actUponKey %W %K"

$widget tag configure command -background gray -foreground red
```

The binding to the *return* or *enter* key is meant to invoke the following script:

```
proc actUponKey { widget key } {

    #
    # If the key was "Return", then examine the current line
    #
    if { $key == "Return" } {
        set line [$widget get "insert linestart" "insert lineend"]

        #
        # Does it contain a @ (special story command)?
        if { [regexp {^ *@} $line] } {
            set tags [$widget tag names insert]
            if { [lsearch $tags command] == -1 } {
                $widget tag add command "insert linestart" "insert lineend"
            }

            #
            # Run the command
            #
            runStoryCommand [lrange $line 1 end]
        }
    }
}
```

It checks if the line is a command (the first non-blank character is the at-sign @), and adds the proper tag if necessary. In any case it immediately executes the command.

The effect is that when you type in the story together with the commands, you actually see it happen. You can also load in existing files and run these automatically. For this the following script is executed:

```
proc runStory { {lineno 1} } {
    global cnv
    global widget

    if { [$widget compare $lineno.0 < end] } {
        set line [$widget get "$lineno.0" "$lineno.0 lineend"]

        #
        # Does it contain a @ (special story command)?
        if { [regexp {^ *@} $line] } {
            runStoryCommand [lrange $line 1 end]
        }

        after 250 [list runStory [incr lineno]]
    }
}
```

Note the use of the *compare* subcommand to check that we are still within range. That way we let the text widget take care of most of the computations.

Another technique that proves useful with the implementation of the puppets too is the “indirect” recursion via *after* – schedule the same command over and over again so that in between these invocations the events that are generated can be handled properly.

## Implementation of the objects

Puppets get a name upon creation and that name becomes a command too. So you can type:

```
@ puppet A green
@ A walks 20 steps
```

This is achieved via the *interp alias* command:

```
proc puppet { name colour } {
    ...
    $cnv create line -2 160 5 140 12 160 -fill black -tag $name -width 2
    $cnv create line -6 140 5 125 16 140 -fill black -tag $name -width 2
    $cnv create oval 0 125 10 145 -fill $colour -outline black -tag $name
    $cnv create oval 2 125 7 120 -fill $colour -outline black -tag $name

    interp alias {} $name {} PuppetAct $name
}
```

All parts of the puppets (arms, legs, body, head) are tagged with the puppet's name, so that moving the puppet is a simple move command of canvas widget:

```
$cnv move $name $dx 0 ;# move horizontally
```

## A poor man's GUI

The last example is of a somewhat different nature: it uses the text widget merely as a specialised geometry manager. Consider the following type of user-interface:

- The user-interface should gather a large number of data items via straightforward entry forms.
- Among these items there are only a few relationships or none at all that the user-interface needs to handle.
- Storing and retrieving the data is of course an essential part of it.

Due to the large number of items, you will need a large number of windows and variables holding the entered data. Now, would it not be nice if the generation of the windows could be automated, as well as the storage and retrieval? That way a lot of effort on trivial matters could be saved.

The PMG package ("poor man's GUI") achieves this goal in the following way:

- You define variables by name and type – the name makes it possible to refer to them later and the type makes it possible to determine what widgets are appropriate when the variable becomes part of a window:

```
#
# General parameters
#
realParameter      time_end
integerParameter   time_step
integerParameter   time_end_secs
```

- You define the layout of the window using plain text:

```
#
# Define the layout of the window(s)
#
```

```

defineForm main "Input general data" {
  Hydrodynamics:
  [(o)hyd_file           ]
  [(o)hyd_file           ]

  Period:  [time_end   ] days
  Timestep: [time_step ] seconds

  Waste load data:
  [(T)wasteload_data    ]

  <ok_b>  <cancel_b>  <help_b>
}

```

Within this text square brackets delimit the information about the widget that needs to be filled in. This consists of a code for the type of widget - (x) for a checkbox, (o) for a radiobutton and so on – and the variable that is associated with it. Several shortcuts as triangular brackets are defined for common widgets as pushbuttons. From there on all is handled automatically.

- When the window is created, the text is parsed and the appropriate widgets are created and embedded (essentially via this type of code):

```

checkboxbutton $w -variable ::PMG::params($param) \
  -text $params($param,text) \
  -font $font -width $width \
  -anchor nw

$t window create end -window $w

```

- You can (and must) associate action code with the pushbuttons:

```

formButton main help_b $help {
  # For the moment, just static text
  showForm help
}

```

The result is a window that looks like this (full code in the appendix):

X	Y	Flow rate	Tracer	BOD
0	55	0.0	0.0	0.0

Because all variables that are used in the application are registered, it is a simple matter to write out their values in the form of a Tcl script:

```
namespace eval ::PMG {
  SetValue hyd_file com-spr.hyd
  SetValue time_end 2
  SetValue time_end_secs 172800
  SetValue time_step 600
  setTableValue wasteload_data 0 5 BOD
  setTableValue wasteload_data 1 4 0.0
  setTableValue wasteload_data 1 5 0.0
  setTableValue wasteload_data 0 1 X
  setTableValue wasteload_data 0 2 Y
  setTableValue wasteload_data 1 1 0
  setTableValue wasteload_data 0 3 {Flow rate}
  setTableValue wasteload_data 1 2 55
  setTableValue wasteload_data 0 4 Tracer
  setTableValue wasteload_data 1 3 0.0
}
```

Retrieving the previous values simply means to *source* that file.

The same technique – using plain text forms – can be used to create reports or input files for external programs:

```
#
# Define the Water Quality input file (is output for the GUI)
#
defineOutput all_data {\
;
; Just an example of how you can set up automatically generated
; input files, not a real DELWAQ input file
;
;
;           0 ; Start time
[time_end_secs] ; Stop time
[time_step   ] ; Timestep
#1
;
; Hydrodynamics
;
[hyd_file           ] ; Hyd-file to be used
#2
;
; Waste load data
;
[wasteload_data    ]
#3
...

```

To keep the package flexible it is possible to define new types of variables. One has to implement a small set of procedures (to define a variable, to output it to file and to create the relevant widget for instance): it is a “data-oriented” approach. Future enhancements include R. Suchenwirth’s hyper-simple menubars (see the Wiki page “Visual menus”).

## Concluding remarks

The text widget is a remarkably versatile widget and can be put to to very different uses, some that have nothing to do with its primary task. Despite its capabilities at least two things are currently possible:

- It is not (easily) possible to create a PostScript file from the contents, something desirable for word-processing for instance.



- It is not possible to render mathematical formulae, as one can not control the the vertical position. Luckily this can be done via the canvas widget.

## References

The following Wiki pages contain the full code or more information:

A simple slide show: <<http://wiki.tcl.tk/3236>>  
 Programmed story telling: <<http://wiki.tcl.tk/8430>>  
 A grep-like utility: <<http://wiki.tcl.tk/8405>>  
 A poor man's GUI: <<http://wiki.tcl.tk/4326>>  
 Rendering mathematical formulae: <<http://wiki.tcl.tk/6004>>  
 Visual menus: <<http://wiki.tcl.tk/8765>>

## Appendix: full code of the example for PMG

```
package require Tk
source "pmgui.tcl"

namespace import ::PMG::*

#
# Constants
#
set ok          "OK"
set cancel     "Cancel"
set help       "Help"

set positive    0.0001
set positive_int 1

#
# General parameters
#
realParameter   time_end          $positive
integerParameter time_step        $positive_int
integerParameter time_end_secs

#
# Hydrodynamics
#

choiceParameter hyd_file \
  {com-spr.hyd "Spring tide" com-np.hyd "Neap tide"}

tableParameter wasteload_data {"X" "Y" "Flow rate" "Tracer" "BOD"} 5
setTableValue  wasteload_data 1 1 0
setTableValue  wasteload_data 1 2 0
setTableValue  wasteload_data 1 3 0.0
setTableValue  wasteload_data 1 4 0.0
setTableValue  wasteload_data 1 5 0.0

#
# Define the layout of the window(s)
#
defineForm main "Input general data" {
  Hydrodynamics:
  [(o)hyd_file ]
  [(o)hyd_file ]

  Period: [time_end ] days
  Timestep: [time_step ] seconds
}
```

```

Waste load data:
[(T)wasteload_data ]

<ok_b> <cancel_b> <help_b>
}

formButton main ok_b $ok {
    closeSave
}

formButton main cancel_b $cancel {
    exit
}

formButton main help_b $help {
    # For the moment, just static text
    showForm help
}

defineForm help "Online help" {

    This is an overly simple help screen.
    Actually it does not provide any information whatsoever

    (I want to replace this by a hypertext help system,
    TODO: write some help text, the system is already available!)

    <ok_b>
}

formButton help ok_b $ok {
    closeForm help
}

proc closeSave {} {
    global prvfile
    set period [GetValue time_end]
    SetValue time_end_secs [expr {int(86400.0*$period)}]
    saveData $prvfile
    appendOutput all_data "wqdata.inp" 1
    exit
}

#
# Define the Water Quality input file (is output for the GUI)
#
defineOutput all_data { \
;
; Just an example of how you can set up automatically generated
; input files, not a real DELWAQ input file
;
;
;           0 ; Start time
[time_end_secs] ; Stop time
[time_step    ] ; Timestep
#1
;
; Hydrodynamics
;
[hyd_file           ] ; Hyd-file to be used
#2
;
; Waste load data
;
[wasteload_data    ]
#3
;
; Initial conditions (not editable)

```

```
;
    0.0      ; Tracer
    0.0      ; BOD
#4
}

#
# Code to run the GUI
#
set prvfile "wqdata.prv"
if { [file exists $prvfile] } {
    loadData $prvfile
}
showForm main "."
setExitCommand closeSave
```