Module 4

Scripting language - Tcl/Tk

Scripting is difficult to define. It has existed for a long time - the first scripting languages were job control languages such as the shell program found in Unix systems. Modern scripting languages such as Perl, Tcl, Python, awk, Ruby and so on are general purpose, but often they have more powerful basic operations than those found in conventional general purpose computer languages. For example it is common to have operators that perform regular-expression pattern matching in a scripting language.

Scripting languages are normally interpreted, and the interpreter contains the routines to do the pattern matching. One line of script code may be equivalent to 100 lines of C. However, the overhead in having a (say) 3MB script interpreter is sometimes a problem, although less so these days.

Perl is widely used, as it is found in active web page developments. Tcl/Tk is useful for GUI development, allowing us to prototype new GUI applications quickly.

4.1 How not to use scripting languages

Don't use to the exclusion of other languages!

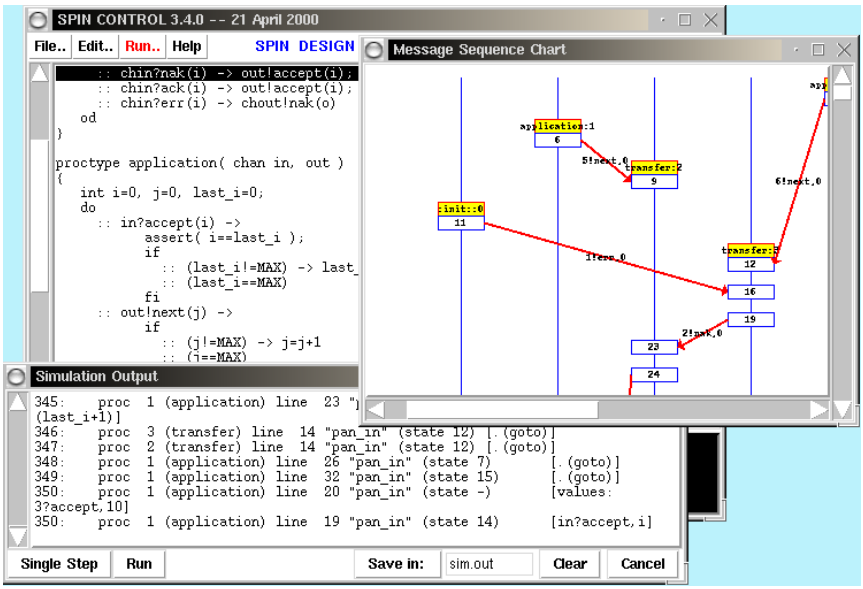
Scripting languages are very good at some things, but sometimes frustratingly bad at other things. For example, many scripting languages use associative, text-based array indexes, and so a simple array lookup may take 1000 times longer than an equivalent lookup in a compiled language.

For this reason, it is common to mix scripting and other languages.

4.2 Tcl/Tk

Wish - the windowing shell, is a simple scripting interface to the Tcl/Tk language. The language Tcl (Tool Command Language) is an interpreted scripting language, with useful inter-application communication methods, and is pronounced 'tickle'. Tk originally was an X-window toolkit implemented as extensions to 'tcl'. However, now it is available *native* on all platforms.

The program ***xspin*** is an example of a portable program in which the entire user interface is written in wish. The program also runs on PCs using NT or Win95, and as well on Macintoshes.



A first use of wish could be the following:

```
manu> wish
wish> button .quit -text "Hello World!" -command {exit}
.quit
wish> pack .quit
wish>
```

You can encapsulate this in a script:

CODE LISTING	**HelloWorld.tcl**

```
#!/usr/local/bin/wish8.1 -f

button .quit -text "Hello World!" -command {exit}
pack .quit
```

If you create this as a file, and make it executable, you should be able to run this simple graphical program.

4.2.1 The structure of Tcl/Tk

The Tcl language has a tiny syntax - there is only a single *command* structure, and a set of rules to determine how to interpret the commands. Other languages have special syntaxes for control structures (if, while, repeat...) - not so in Tcl. All such structures are implemented as *commands*.

There is a runtime library of compiled 'C' routines, and the 'level' of the GUI interface is quite high.

Comments: If the first character of a command is #, it is a comment.

Tcl commands: Tcl commands are just words separated by spaces. Commands return strings, and arguments are just further words.

```
command argument argument
command argument
```

Spaces are important:

```
expr 5*3        has a single argument
expr 5 * 3      has three arguments
```

Tcl commands are separated by a new line, or a semicolon, and arrays are indexed by text:

```
set a(a\ text\ index) 4
```

Tcl/Tk quoting rules :

The "quoting" rules come in to play when the " or { character are first in the word. ".." disables a few of the special characters - for example space, tab, newline and semicolon, and {..} disables everything except \{, \} and \nl. This facility is particularly useful for the control structures - they end up looking very like 'C':

```
while {a==10} {
  set b [tst a]
}
```

Tcl/Tk substitution rules:

Variable substitution: The dollar sign performs the variable value substitution. Tcl variables are strings.

```
set a 12b      a will be "12b"
set b 12$a     b will be "1212b"
```

Command substitution: The []'s are replaced by the value returned by executing the Tcl command 'doit'.

```
set a [doit param1 param2]
```

Backslash substitution:

```
set a a\ string\ with\ spaces\ \
and\ a\ new\ line
```

Tcl/Tk command examples:

Procedures	File Access	Miscellaneous
proc name {parameters} {body}	open <name>	source <NameOfFile>
	read <fileID>	global <varname>
	close <fileID>	catch <command>
	cd <directoryname>	format <formatstring> <value>
		exec <process>
		return <value>

List operators:

```
split   <string> ?splitcharacters?
concat  <list> <list>
lindex  <list> <index>
... + lots more
```

Control structures:

```
if {test} {thenpart} {elsepart}[1]while {test} {body}
for {init} {test} {incr} {body}
continue
case $x in a {a-part} b {b-part}
```

[1]The Tcl/Tk words *then* and *else* are noise words, which may be used to increase readability.

Widget creation commands:

The first parameter to each is a 'dotted' name. The dot heirarchy indicates the relationships between the widgets.

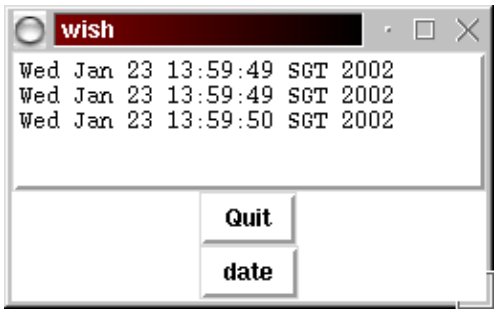
```
% label  <name>   -  optional parameter pairs ...
% canvas <name>   -  optional parameter pairs ...
% button <name>   -  optional parameter pairs ...
% frame  <name>   -  optional parameter pairs ...
% ... and so on
```

When you create a widget ".b", a new command ".b" is created, which you can use to further communicate with it. The geometry managers in Tk assemble the widgets:

```
% pack <name> .... where ....
```

4.2.2 Tcl/Tk example software

Here is a very small Tcl/Tk application, which displays the date in a scrollable window:



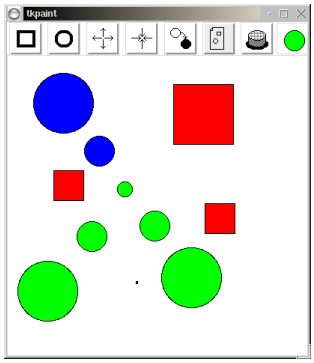
The code for this is:

CODE LISTING **SimpleProg.tcl**

```
#!/usr/local/bin/wish8.1 -f

text .log -width 60 -height 5 -bd 2 -relief raised
pack .log
button .buttonquit -text "Quit" -command exit
pack .buttonquit
button .buttondate -text "date" -command getdate
pack .buttondate
proc getdate {} {
   set result [exec date]
   .log insert end $result
   .log insert end \n
}
```

Here is tkpaint - a drawing/painting program written in Tcl/Tk:



The mainline of the source just creates the buttons, and packs the frame:

CODE LISTING **tkpaint1.tcl**

```
#! /usr/local/bin/wish -f

set thistool rectangle
set thisop grow
set thiscolour black
button .exitbtn   -bitmap @exit.xbm      -command exit
button .squarebtn -bitmap @square.xbm    -command setsquaretool
button .circlebtn -bitmap @circle.xbm    -command setcircletool
button .shrnkbtn  -bitmap @shrink.xbm    -command "set thisop shrnk"
button .growbtn   -bitmap @grow.xbm      -command "set thisop grow"
button .printbtn  -bitmap @print.xbm     -command printit
button .colorbtn -bitmap @newcolour.xbm -command setanewcolour
canvas .net -width 400 -height 400 -background white -relief sunken
canvas .status -width 40 -height 40 -background white -relief sunken

pack .net -side bottom
pack .status -side right
pack .squarebtn .circlebtn -side left -ipadx 1m -ipady 1m -expand 1
pack .exitbtn .printbtn -side right -ipadx 1m -ipady 1m -expand 1
pack .colorbtn .shrnkbtn .growbtn -side right -ipadx 1m -ipady 1m -expand 1

bind .net <ButtonPress-1> {makenode %x %y}
.status create rectangle 10 10 37 37 -tag statusthingy -fill $thiscolour
set nodes 0; set oldx 0; set oldy 0;
```

Routines for dragging, scaling and printing:

CODE LISTING **tkpaint4.tcl**

```
proc beginmove {x y} {
    global oldx oldy
    set oldx $x; set oldy $y
}

proc domove {item x y} {
    global oldx oldy
    .net move $item [expr "$x – $oldx"] [expr "$y – $oldy"]
    set oldx $x; set oldy $y
}

proc altersize {item x y z} {
    .net scale $item $x $y $z $z
}

proc printit {} {
    .net postscript -file "pic.ps"
}
```

Node operations for tkpaint:

CODE LISTING **tkpaint2.tcl**

```
proc makenode {x y} {
   global nodes oldx oldy thistool thiscolor
   set nodes [expr "$nodes+1"]

   set x1 [expr "$x-20"]; set y1 [expr "$y-20"]
   set x2 [expr "$x+20"]; set y2 [expr "$y+20"]
   if {[string compare $thistool "oval"] == 0} {
      .net create oval $x1 $y1 $x2 $y2 -tag node$nodes -fill $thiscolor
   }
   if {[string compare $thistool "rectangle"] == 0} {
      .net create rectangle $x1 $y1 $x2 $y2 -tag node$nodes -fill $thiscolor
   }
   .net bind node$nodes <Enter> ".net itemconfigure node$nodes -width 5"
   .net bind node$nodes <Leave> ".net itemconfigure node$nodes -width 1"
   .net bind node$nodes <ButtonPress-3> "beginmove %x %y"
   .net bind node$nodes <B3-Motion> "domove node$nodes %x %y"
   .net bind node$nodes <ButtonPress-2> "dothisop node$nodes %x %y"
}

proc dothisop {item x y} {
   global thisop
   if {[string compare $thisop "shrink"] == 0} {
       altersize $item $x $y 0.5
   }
   if {[string compare $thisop "grow"] == 0} {
       altersize $item $x $y 2.0
   }
}
```

More routines:

CODE LISTING **tkpaint3.tcl**

```
proc setcircletool {} {
    global thistool thiscolor
    set thistool oval
    .status delete statusthingy
    .status create oval 10 10 37 37 -tag statusthingy -fill $thiscolor
}

proc setsquaretool {} {
    global thistool thiscolor
    set thistool rectangle
    .status delete statusthingy
    .status create rectangle 10 10 37 37 -tag statusthingy -fill $thiscolor
}

proc setanewcolor {} {
    global thiscolor

    if {[string compare $thiscolor "black"] == 0} {
            set thiscolor green
    } { if {[string compare $thiscolor "green"] == 0} {
            set thiscolor blue
        } { if {[string compare $thiscolor "blue"] == 0} {
                set thiscolor red
            } { if {[string compare $thiscolor "red"] == 0} {
                    set thiscolor orange
                } { set thiscolor black }
              }
          }
       }
    .status itemconfigure statusthingy -fill $thiscolor
}
```

4.2.3 C/Tk

In the following example, a Tcl/Tk program is integrated with a C program, giving a very small codesize GUI application, that can be compiled on any platform - Windows, UNIX or even the Macintosh platform without changes.

CODE LISTING **CplusTclTk.c**

```
#include <stdio.h>
#include <tcl.h>
#include <tk.h>

char tclprog[] = "\
      proc fileDialog {w} {\
        set types {\
          { \"Image files\" {.gif} }\
          { \"All files\" *}\
        };\
        set file [tk_getOpenFile −filetypes $types −parent $w];\
        image create photo picture −file $file;\
        set glb_tx [image width picture];\
        set glb_ty [image height picture];\
        .c configure −width $glb_tx −height $glb_ty;\
        .c create image 1 1 −anchor nw −image picture −tags \"myimage\";\
      };\
      frame .mbar −relief raised −bd 2;\
      frame .dummy −width 10c −height 0;\
      pack .mbar .dummy −side top −fill x;\
      menubutton .mbar.file −text File −underline 0 −menu .mbar.file.menu;\
      menu .mbar.file.menu −tearoff 1;\
      .mbar.file.menu add command −label \"Open...\" −command \"fileDialog .\";\
      .mbar.file.menu add separator;\
      .mbar.file.menu add command −label \"Quit\" −command \"destroy .\";\
      pack .mbar.file  −side left;\
      canvas .c −bd 2 −relief raised;\
      pack .c −side top −expand yes −fill x;\
      bind . <Control−c> {destroy .};\
      bind . <Control−q> {destroy .};\
      focus .mbar";

int
main (argc, argv)
     int argc;
     char **argv;
{
    Tk_Window mainWindow;
    Tcl_Interp *tcl_interp;

    setenv ("TCL_LIBRARY", "/cygnus/cygwin−b20/share/tcl8.0");
    tcl_interp = Tcl_CreateInterp ();
    if (Tcl_Init (tcl_interp) != TCL_OK || Tk_Init (tcl_interp) != TCL_OK) {
        if (*tcl_interp->result)
            (void) fprintf (stderr, "%s: %s\n", argv[0], tcl_interp->result);
        exit (1);
    }
    mainWindow = Tk_MainWindow (tcl_interp);
    if (mainWindow == NULL) {
        fprintf (stderr, "%s\n", tcl_interp->result);
        exit (1);
    }

    Tcl_Eval (tcl_interp, tclprog);
    Tk_MainLoop ();

    exit (1);
}
```

The first half of the listing is a C string containing a Tcl/Tk program. The second part of the listing is C code which uses this Tcl/Tk.

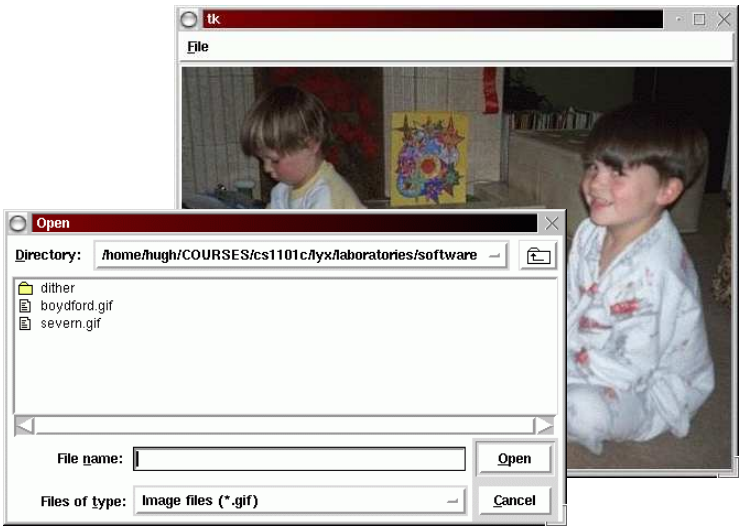
On a Win32 system, we compile this as:

```
gcc -o CplusTclTk CplusTclTk.c -mwindows -ltcl80 -ltk80
```

On a UNIX system we use:

```
gcc -o CplusTclTk CplusTclTk.c -ltk -ltcl -lX11 -lm -ldl
```

And the result is a simple viewer for GIF images. The total code size is 57 lines. The application looks like this when running:



4.3 Summary of topics

In this module, we introduced the following topics:

- Practical programming in Tcl/Tk
- Other Tk language bindings
- Some sample programs

Tutorial 4 - questions for week 4 (Jan 28-Feb 1, 2002)

1. Given the frame **.frm** containing a canvas and a quit button, give sensible names for the canvas and the button.
2. Modify **SimpleProg.tcl** to have an extra button **clear** above the **quit** button which clears the date display.
3. Modify **SimpleProg.tcl** to have an extra button **clear** to the left of the **quit** button which clears the date display.
4. What is the effect of the following tcl command? **set a [exec ls]**
5. What is the effect of the following tcl command? **set a expr 3 + 4**
6. Write a minimal Tk application which puts up a single **File** menu with a **Quit** item in it.

Further study

- http://www.pconline.com/~erc/tclwin.htm
- http://tcl.activestate.com/scripting/
- http://www.msen.com/~clif/TclTutor.html