An Overview of Limbo/Tk

Lucent Technologies Inc.
Revised June 2000 by Vita Nuova

Overview

Limbo/Tk is a concise and powerful way to construct graphical user interfaces without directly using the Draw module primitives. Standard interfaces can quickly be created from collections of menus, buttons, and other widgets that are part of Limbo/Tk's visual toolkit. It is modelled on Ousterhout's Tk 4.0 [1, 2], commonly deployed with the scripting language Tcl as Tcl/Tk. Although inspired by Tk 4.0, Inferno's Tk implementation is new, and unrelated to Ousterhout's. It is intended to be used with the new programming language Limbo, not Tcl. Limbo/Tk applications make extensive use of Limbo's concurrent programming constructions and data structures, and that is reflected in the interface. Section 9 of Volume 1 of the *Inferno Programmer's Manual* provides a detailed reference for Limbo/Tk. This paper provides an overview of its use in some simple staged examples, and concludes with a summary of the differences between the Limbo/Tk implementation and Tk 4.0. In the rest of this paper, Tk refers to Limbo/Tk, and Tk 4.0 refers to Ousterhout's original implementation.

1. The Limbo/Tk environment

Limbo applications access Tk by means of a built-in module, `$Tk`. The standard distribution also includes the window manager `wm` and the Limbo module `Wmlib`. Unlike `Tk`, `Wmlib` is not built-in but implemented in Limbo by `/appl/lib/wmlib.b`. It simplifies and standardises the construction of windowed applications; it also contains some graphical devices such as tabbed notebooks not provided directly by Limbo/Tk. The essentials of both Tk and Wmlib are discussed here.

Programmers usually need only three functions from the Tk module:

- `toplevel`
  Creates a graphical window — a Tk top level widget — that can be used to build a Limbo/Tk application. The function returns a reference to an adt `Tk->Toplevel` adt that represents the window in subsequent Tk operations.
- `cmd`
  Creates and arranges graphic objects within the toplevel window by processing Limbo/Tk command strings. The interface to Tk is primarily the passing of strings between the application and Tk of the toolkit using `cmd`. Each call to `cmd` returns a string representing the result of the Tk command; a string starting with `!` diagnoses an error.
- `namechan`
  Gives a name within Tk (in the scope of a given window) to a Limbo `chan of string` that Tk commands can use to send data to a Limbo program.

Other functions in the module have more specialised uses that will not be discussed here. For instance, `mouse` and `keyboard` are used by a window manager to send mouse and keyboard events to the Tk implementation for distribution to applications.

Even `toplevel` is not commonly used in the window manager environment: a function `Wmlib->titlebar` provides the usual interface to `toplevel`. The low-level interface will be described first, for completeness, then the normal case using `titlebar`.

2. Basic Limbo/Tk

This section shows a simple Tk application that uses only the fundamental Tk functions.

2.1. Preliminaries

The example assumes that the Tk module is loaded as tk :

```
include "tk.m";
    tk: Tk;
    ...
init(ctxt: ref Draw->Context, nil: list of string)
{
    tk = load Tk Tk->PATH;
    ...
}
```

2.2. Creating a toplevel

The following fragment makes the Limbo identifier top refer to a new ref Tk->Toplevel for use in later Tk commands:

```
    top := tk->toplevel(ctxt.screen, "-x 150 -y 150");
```

The upper left corner of this window will be at point (150,150), where (0,0) is the upper left corner of the screen; *x* coordinates increase from left to right, and *y* coordinates increase from top to bottom.

In general, Tk->toplevel takes a screen argument and a string containing further options, and it returns a reference to a top-level Limbo/Tk widget on the given screen. The options argument contains -option *value* pairs, such as -relief raised . As well as the generic options, toplevel accepts the options -x *int* and -y *int* to specify the upper left corner of the toplevel widget, where (0,0) is the top left corner of the screen, and -debug *bool* to cause a trace of all Tk commands to be printed, if the boolean value is true.

2.3. Creating a named channel to Tk

The following fragment creates a chan of string called c, then associates the name cmdchan within Tk with the Limbo channel c:

```
    c := chan of string;
    tk->namechan(top, c, "cmdchan");
```

The named channel cmdchan can now be used in a special Tk send command to send strings to be processed by a Limbo program, typically to notify it of an event. Note that the Limbo identifier name need not match the name given to Tk, although it is invariably easier to follow the code if the two are the same.

2.4. Defining and positioning widgets

The following fragment uses tk->cmd to define four widgets: two buttons, a label, and an entry widget. The widgets are positioned in their parent window (in this case the toplevel window top) using the Tk command pack :

Figure 1. Two buttons, a label and an entry widget.

```
# define widgets
tk->cmd(top, "button .b1 -text Exit -command {send cmdchan exit}");
tk->cmd(top, "button .b2 -text Send -command {send cmdchan send}");
tk->cmd(top, "label .l -text {Name: }");
tk->cmd(top, "entry .e");

# bind newline character in entry widget to command
tk->cmd(top, "bind .e <Key-\n> {send cmdchan send}");

# pack widgets
tk->cmd(top, "pack .b1 .b2 .l .e -side left; update");
```

This particular pack command packs the widgets named .b1, .b2, .l, and .e into the top window beginning at the left side. The update command forces Tk to update the screen right away. The result is shown in Figure 1.

Entering a newline (return or enter key) the character \n in Limbo in the entry box results in the execution of the Tk command {send cmdchan send}, because of the binding set by bind .e <Key-\n> previously executed by tk->cmd. The bind command is often used to bind specific widget events (including key presses, mouse button presses, and mouse motion) to Tk send commands.

2.5. Processing widget events

This next fragment defines what will happen when a user selects either the Exit or the Send buttons. The Exit behaviour is simple: the program ends. If a user touches Send, the program executes tk->cmd to get whatever text is in the entry widget .e then prints it to standard output.

```
for(;;) {
    s := <- c;
    case s {
    "exit" =>
        return;

    "send" =>
        sys->print("name was: %s\n", tk->cmd(top, ".e get"));
    }
}
```

3. Example – using Tk and Wmlib

This section uses both Tk and Wmlib to create a simple window manager application with a titlebar, including resize and exit buttons. This is the usual way to create new windows.

4. Preamble

The example assumes that the Tk module is loaded as before, as module variable tk, but furthermore that the Wmlib module is also loaded, as wmlib:

```
include "tk.m";
    tk: Tk;
include "wmlib.m";
    wmlib: Wmlib;
    ...
    tk = load Tk Tk->PATH;
    wmlib = load Wmlib Wmlib->PATH;
    wmlib->init();
```

Note that `wmlib->init` is called once to initialise the `wmlib` module just loaded, before any other functions are called.

In window manager applications the `Tk->toplevel` function is not normally used directly. Instead, a window manager interface is used to create both the top level widget and a channel to receive events from the window manager. The `titlebar` function has the signature:

```
titlebar(scr: Draw->Screen, tkargs: string, title: string, buts: int):
    (ref Tk->Toplevel, chan of string);
```

The `Screen` is the one on which the window is to be created, normally the one passed in the `Context` parameter to a program's `init` function. The *tkargs* parameter can control the position and appearance of the window, but is best left `nil` (or the empty string) to use the window manager's defaults (see *wmlib* (2) for details otherwise), including automatic placement. The *title* string gives the title that appears in the title bar. Finally *buts* is a bit set that selects the buttons to appear. The value `Wmlib->Appl` gives the usual resize and hide buttons; the exit (delete) button always appears. The following is used in the example:

```
(top, titlechan) := wmlib->titlebar(ctxt.screen, nil,
                    "Text Browser", Wmlib->Appl);
```

Note that `titlebar` returns a tuple. The first element is a reference to the Tk top level widget for use in later Tk commands. The second element of the tuple is a Limbo channel of type `chan of string` that passes window manager events to the application.

The channel `titlechan` is used by `wmlib` to send messages, but it is normally necessary to create a channel to Tk to receive events from widgets the application creates:

```
cmdchan := chan of string;
tk->namechan(top, cmdchan, "cmdchan");
```

4.1. Defining and positioning widgets

The function `Wmlib->tkcmds` takes two arguments, a `ref Tk->Toplevel` that identifies a top level window, and an `array of string`. Each element of the array is a Tk command acceptable to `Tk->cmd`; `Wmlib->tkcmds` simply applies it to each element of the array.

Most of the following fragment consists of Tk command strings that are members of the array of strings `tk_config`. The comments describe the widgets being created. Not all widgets and menu items in this example are functional. The last line executes the array of commands using `wmlib->tkcmds`:

```
tk_config := array[] of {

# define menubar  frame, widget  frame, text  frame
"frame .mbar -relief groove -bd 2",
"frame .w",
"frame .text",

# define and  pack  menus
"menubutton .file -text File -menu .file.m",
"menubutton .edit -text Edit -menu .edit.m",
"menubutton .help -text Help -menu .help.m",

"menu .file.m",
".file.m add command -label Send -command {send cmdchan send}",
".file.m add command -label Exit -command {send cmdchan exit}",
"menu .edit.m",
".edit.m add command -label Cut",

"menu .help.m",
".help.m add command -label Index -underline 0",

"pack .file .edit -side left -in .mbar; update",
"pack .help -side right -in .mbar",

# define and  pack  buttons and  text  entry  box (for  file name)
"button .b1 -text Send -command {send cmdchan send}",
"button .b2 -text Open -command {send cmdchan open}",
"label .l -text {Name: }",

"entry .e",
"bind .e <Key-\n> {send cmdchan open}",

"pack .b1 .b2 .l .e -side left -in .w",

# define and  pack  text  panel and  its scrollbar
"text .t -yscrollcommand {.scroll set} -bg white",
"scrollbar .scroll -command {.t yview}",
"pack .scroll -side left -fill y -in .text",
"pack .t -side right -in .text -expand 1 -fill both",

# pack  frames
"pack .text -side bottom -fill both -expand 1",
"pack .mbar .w -fill x; update",
"pack propagate . 0",
};

# run the Tk  commands
wmlib->tkcmds(top, tk_config);
```

The result of executing these Tk commands is shown in Figure 2.

The arrays defining Tk widgets are sometimes made global to a module when they can sensibly be used by several functions. It is also common to use small Limbo functions to replicate similar widgets by building Tk commands from the value of parameters, using Limbo string concatenation or sys->sprint .

4.2. Processing widget events

This fragment uses an alt block to wait for strings to arrive from either titlechan or cmdchan .

When a string is received on titlechan , the case statement either handles it directly (as with exit) or passes it to wmlib->titlectl for normal handling by the window manager.

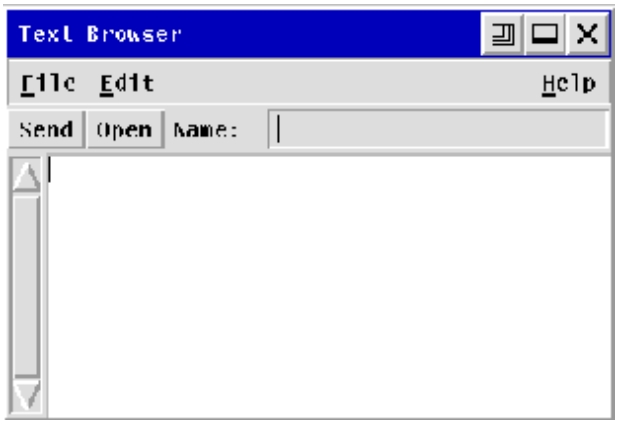

Figure 2. A Wm application with a menu bar, a tool bar, and a text window.

When a string is received on cmdchan, the program acts accordingly: it writes the text in the entry widget to standard output (send); calls do_open to open the file name currently in the entry box (open); or returns from the processing loop (exit).

```
for(;;) {
    alt {
    s := <-titlechan => # message from title bar buttons
        case s {
        "exit" =>
            return;
        * =>
            wmlib->titlectl(top,s);
        }

    com := <-cmdchan => # message from widgets created above
        case com {
        "send" =>
            sys->print("name was: %s\n", tk->cmd(top, ".e get"));
        "open" =>
            do_open(top, tk->cmd(top, ".e get"));
        "exit" =>
            return;
        }
    }
}
```

Although this example uses a Tk text entry widget, Wmlib provides a function filename that pops up a graphical device that allows a user to select a file by typing a name, browsing the file system, or a mixture of both. See *wmlib* (2) for details.

4.3. Putting text into the text widget

The do_open function below uses the buffered I/O module Bufio to read lines from the file named in the entry widget and add them to the text currently in the text widget .t.

```
do_open(top:    ref  Tk->Toplevel,    file:  string)
{
     iofd  :=  bufio->open(file,     Bufio->OREAD);
     if(iofd   ==  nil){
          wmlib->dialog(top,     "error   -fg  red",  "Open  file",
               sys->sprint("%s:     %r",  file),   0,  "Ok"::nil);
          return;
     }

     tk->cmd(top,    ".t  delete   1.0  end");
     tk->cmd(top,    "cursor   -bitmap   cursor.wait");

     for(;;){
          line  :=  iofd.gets('\n');
          if(line   ==  nil)
               break;
          tk->cmd(top,    ".t  insert   end  '"  +  line);
     }
     tk->cmd(top,    "cursor   -default");
}
```

If the file cannot be opened, do_open calls wmlib->dialog to pop up a diagnostic message panel, rather than (say) printing a message to standard error, and returns. If the file was opened, do_open deletes the current contents of the frame, and reads the file into it, inserts one line at a time. Tk allows the data inserted to contain embedded newlines, and a more efficient implementation could read blocks of data from the file and insert them, but some care is required. A text file in Inferno contains Unicode characters in UTF-encoding, and the bytes of a single character might be split across separate reads. Iobuf.gets by contrast is guaranteed to reassemble complete Unicode characters from the buffered input stream. A program using Iobuf.read (or Sys->read) to fetch blocks of data would typically use Sys->utfbytes to find maximal sequences of UTF-encoded characters and insert large chunks of text at once. See the function loadtfile in /appl/wm/edit.b for example.

5. Limbo/Tk command syntax

Once a toplevel widget has been built, an application calls tk->cmd to issue commands to Tk and receive results. This section describes in more detail the contents of the string argument that conveys the commands.

5.1. Command strings

The command string may contain one or more commands, separated by semicolons. A semicolon is not a command separator when it is nested in braces ({}) or brackets ([]), or it is escaped by a backslash (\).

Each command is divided into *words*: sequences of characters separated by one or more blanks or tabs, subject to the following quoting rules:

A word beginning with an opening brace ({) continues until the balancing closing brace (}) is reached. The outer brace characters are stripped. A backslash (\) can be used to escape a brace, preventing special interpretation.

A word beginning with an opening bracket ([) continues until the balancing closing bracket (]) is reached. The enclosed string is then evaluated as if it were a command string, and the resulting value is used as the contents of the word.

At any point in the command string a single quote (') causes the rest of the string to be treated as one word.

Single commands are executed in order until they are all done or an error is encountered. By convention, an error is signalled by a return value starting with an exclamation mark. The return value from cmd is the return value of the first error-producing command or else the return value of the final single command.

To execute a single command, the first word is examined. It can be one of the following:

One of the following widget creating commands:

```
button          menu
canvas          menubutton
checkbutton     radiobutton
entry           scale
frame           scrollbar
label           text
listbox
```

The second word of each of these commands is the name of the widget to be created. The remaining words are option/value pairs.

A widget name (beginning with a dot .) that corresponds to an existing widget. The second word gives the name of a particular widget subcommand and the remaining words are arguments for the subcommand.

A pack , bind , focus , grab , put , destroy , image , or update command. These commands manipulate existing widgets or control Tk. Most are the same as documented for Tk 4.0. The bind command is significantly different, and the image command is more limited.

The send command, which sends a string to a Limbo process. The second word is the Tk name of a Limbo channel (previously registered with wmlib->chan), and the rest of the command is sent as a single string along the channel.

The variable command. Limbo/Tk generally does not provide the variables of Tcl/Tk; radiobuttons are an exception. The variable command takes the name of a variable defined in a radiobutton as the second word, and the value of the variable is the result of the command. Furthermore, there is one predefined variable whose value can be retrieved this way: the lasterror variable is set every time a Tk command returns an error. The value is the offending command (possibly truncated) followed by the error return value. The lasterror variable is cleared whenever it is retrieved using the variable command. This allows several Tk commands to be executed without checking error returns each time. A call to the variable command with lasterror at strategic points can make sure that an unexpected error has not occurred.

The cursor command. This command takes a number of option/value pairs to control the appearance and placement of the cursor. Available options are: -x *int* and -y *int* , to change the cursor position to align its hotpoint at the given point (in screen coordinates); -bitmap *filename* or -image *imagename* to change the appearance of the cursor, and -default to change back to the default appearance of the cursor.

Because the language accepted by the cmd function has no user-defined functions, no control flow and very few variables, almost all applications need to have some of their logic in Limbo programs. The modern concurrency constructions provided by Limbo processes, channels, send/receive operators and alt replace unstructured interrupts (callbacks), often used by other graphics systems, by structured control flow. (The Inferno shell does provide support, however, for rapid prototyping using Tk and a scripting language: see the manual pages for *sh-tk* (1) and *wish* (1) in Volume 1.)

5.2. Widget options

In Tk, all widget creation commands, and all cget widget commands accept a common set of generic options in addition to widget-specific options. Except as noted otherwise, the meanings are the same as they are in Tk 4.0. The allowable forms of things like *color*, *dist*, and *font* are slightly different in Limbo/Tk. See *types* (9) in Volume 1 for precise definitions. The generic options are as follows:

```
-activebackground    color
-activeforeground    color
-actwidth   dist
-actheight   dist
```

Note: the -actwidth and -actheight variables are overridden by the packer, but are useful as arguments to cget to retrieve the actual width and height (inside the border) of a widget after packing.

```
-background   color (or -bg color)
-borderwidth   dist (or -bd dist)
-font  font
-foreground   color (or -fg color)
-height  dist
-padx  dist
-pady  dist
-relief  relief
-state  normal, -state  active, or -state  disabled
```

Note: -state is only relevant for some widgets (for example, entry widgets).

```
-selectbackground    color
-selectborderwidth    dist
-selectcolor   color
```

Note: -selectcolor is the colour of the box in selection menu items.

```
-selectforeground    colour
-width  dist
```

In general, the manual page for each widget in section 9 of Volume 1 tells which of the generic Tk options the widget accepts.

The *dist* parameters are lengths, expressed in the following form: an optional minus sign, then one or more decimal digits (with possible embedded decimal point), then an optional units specifier. The units specifiers are the following:

```
c   centimetres
m   millimetres
i   inches
p   points
h   height of widgets font(*)
w   width of 0 character in widgets font(*)
```

The ones marked (*) are specific to Limbo/Tk.

Tcl/Tk 4.0 widgets do not uniformly take -width and -height options; instead, each widget may take either or both, and the interpretation of a number lacking a units specifier varies from widget to widget. For example, in Tk 4.0 -width 25 means 25 characters to an entry widget, but 25 pixels to a canvas widget. In Limbo/Tk, all widgets may specify width and height, and bare numbers always mean screen pixels.

A *colour* parameter can be a colour name or an RGB value. Only a few names are known:

aqua	fuchsia	maroon	purple	yellow
black	gray	navy	red	
blue	green	olive	teal	
darkblue	lime	orange	white	

For RGB values, either `#`*rgb* or `#`*rrggbb* can be used, where *r*, *rr*, etc. are hexadecimal values for the corresponding colour components.

A *font* parameter gives the full path name of an Inferno font file, for example, `/fonts/pelm/unicode.9.font`.

A *bitmap* parameter is not used by any of the generic options, but is worth mentioning here. Unlike Tk 4.0, a *bitmap* in Limbo/Tk is not restricted to a 1-bit deep bitmap to be coloured with foreground and background. Instead, it can be a full-colour image (pixmap in X11 terminology), which is displayed as is. If *bitmap* begins with a `@`, the remaining characters should be the path name of an Inferno image file. If *bitmap* begins with the character `<`, the remaining characters must be a decimal integer giving a file descriptor number of an open file from which the bitmap can be loaded. Otherwise, *bitmap* should be the name of a bitmap file in the directory `/icons/tk`.

Options not supported in Limbo/Tk

The following options provided by Tk 4.0 are not supported by any Limbo/Tk widget:

`-cursor`	`-insertofftime`	`-wraplength`
`-disabledforeground`	`-insertontime`	
`-exportselection`	`-insertwidth`	
`-geometry`	`-repeatdelay`	
`-highlightbackground`	`-repeatinterval`	
`-highlightcolor`	`-setgrid`	
`-highlightthickness`	`-takefocus`	
`-insertbackground`	`-textvariable`	
`-insertborderwidth`	`-troughcolor`	

6. Limbo/Tk commands

This section lists all the commands documented in the Tk 4.0 man pages, giving the differences between the behaviour specified in those man pages and the behaviour implemented in Limbo/Tk. Some common Tcl commands are listed as well. Bear in mind that some Tk 4.0 options are unsupported, as noted above.

`bell [-displayof` *window* `]`
: Not implemented.

`bind` *widget* `<`*event*`-`*event*`-...-`*event*`>` *command*
`bind` *widget* `<`*event*`-`*event*`-...-`*event*`> +` *command*
: The `bind` command is perhaps the command that differs most from Tk 4.0. In general, only a subset of its functionality is implemented. One difference is that *widget* must be the name of an existing widget. The notion of a widget class is completely absent in Limbo/Tk. Event sequence specifications are also more restricted. A sequence is either a single character (rune), meaning a KeyPress of that character, or a sequence of *events* in angle brackets. *Events* are separated by blanks or minus signs. See *bind* (9) for a complete discussion.

`bindtags` *window* `[`*taglist* `]`
: Not implemented.

`button` *pathname* `[`*options* `...]`
: As in Tk 4.0 (but note differences in units for `-height` and `-width`).

`canvas` *pathname* [*options ...*]
The Postscript subcommand is not implemented.

`checkbutton` *pathname* [*options ...*]
Unimplemented options: `-indicatoron`, `-offvalue`, `-onvalue`, and `-selectimage`. The `flash` subcommand is not implemented.

`clipboard` *operation*
Not implemented.

pathname `configure` [*option ...*]
Configure options for widget *pathname*. Widget-specific, see the manual entry for the widget in section 9 of Volume 1.

`destroy` [*window ...*]
As in Tk 4.0, but note that `destroy .` is rarely needed because top level windows are automatically destroyed by the Inferno garbage collector immediately when the last reference vanishes.

`entry` *pathname* [*options ...*]
The `scan` subcommand is not implemented. Some key bindings are not implemented when there is currently no way to type those keys to Inferno (for example, Home). Note difference in units for `-height` and `-width`.

`event` *operation*
Not implemented: normally replaced by Tk `send` or Limbo channel send operation within the application.

`focus` *window*
The focus model in Inferno is different. Only one widget has the keyboard focus at a given time. Limbo/Tk does not maintain a private keyboard focus for each toplevel tree and automatically move the focus there whenever the tree is entered. (Canvas and text widgets, however, do maintain a private keyboard focus.) The Limbo/Tk `focus` command moves the keyboard focus to the given *window*. By default, the first press of the primary button in an `entry`, `listbox` or `text` widget causes the focus to be moved to that widget. Just entering a `menu` widget gives it the focus. The `-displayof`, `-force` and `-lastfor` options are not implemented.

`frame` *pathname* [*options ...*]
Unimplemented options: `class`, `colormap`, and `visual`.

`grab` *window*
`grab` *option* [*arg ...*]
Limbo/Tk implements only global grabs, so the `-global` option is not recognised. The `grab current` command is not implemented. The `grab` command is not recognised as a synonym for `grab set`.

`grid` *operation* [*arg ...*]
Not implemented.

`image create bitmap` [*name*] [*options*]
`image` *option* [*arg arg ...*]
Only bitmap image types are implemented, but, as documented under `bitmap`, Inferno bitmaps are not just 1-bit deep; they encompass both bitmaps and photo (colour) images as provided by Tk/4.0. Limbo/Tk does not, however, recognise the wide variety of graphics formats that Tk 4.0 does. Instead, only Inferno's own format is supported internally, and external programs are provided to convert between that and other formats such as JPEG. The file descriptor syntax for specifying bitmaps is useful when an external program writes the bitmap to a file descriptor. If a mask file is given, it may also have a depth

greater than 1 bit; the meaning is that if a pixel of the mask is non-zero then the corresponding pixel of the image should be drawn. (But see the handling of bitmaps used as stipples in *canvas* (9).) The -data and -maskdata options are not implemented.

label *pathname* [*options* ...]
Unimplemented options: -justify and -wraplength. Note difference in units for -height and -width.

listbox *pathname* [*options* ...]
The bbox and scan subcommands are not implemented. Note difference in units for -height and -width.

lower *window*
The belowThis optional parameter is not recognised.

menu *pathname* [*options* ...]
Unimplemented options: -postcommand, -tearoff, -tearoff command, and -transient. In the add subcommand, the -accelerator, -indicatoron, and -selectimage options are not implemented. In the index subcommand, the last and pattern index forms are not implemented. The configure and entrycget subcommands are not implemented.

menubutton *pathname* [*options* ...]
Unimplemented options: -indicatoron, -justify, and -wraplength.

message *pathname* [*options* ...]
Not implemented (subsumed by label).

option *operation* [*arg* ...]
Not implemented. There is no option database.

pack *option arg* ...
pack *slave* ... [*options* ...]
pack configure *slave* ... [*options* ...]
pack forget *slave* ...
pack propagate *master* [0|1]
pack slaves *master*
The info subcommand is not implemented.

place *operation* [*arg* ...]
Not implemented.

radiobutton *pathname* [*options* ...]
Unimplemented options: -indicatoron, -justify, -selectimage, and -wraplength. The flash subcommand is not implemented.

raise *window*
The aboveThis optional parameter is not recognised.

scale *pathname* [*options* ...]
Unimplemented options: -digits and -variable.

scrollbar *pathname* [*options* ...]
The old syntax of set and get is not supported.

selection
Not implemented.

send *channame string*
Rather than sending data to a different application, the send command sends a

given *string* down the Limbo channel associated with *channame* , as set by namechan .

text *pathname* [*options ...*]
The dump subcommand is not implemented. The -regexp mode of the search subcommand is not implemented.

tk *operation* [*arg ...*]
Not implemented.

tkerror
Not implemented.

tkwait *operation name*
Not implemented.

toplevel *pathname* [*option value ...*]
There is no toplevel Tk command implemented by the cmd function; instead, the Tk module entrypoint toplevel is used to make toplevel widgets (windows) as described above.

update
In Tcl/Tk, update is a Tcl command that invokes the event handler loop. In Limbo/Tk, it flushes any pending updates to the screen. The optional idletasks argument is not recognised.

winfo *operation* [*arg ...*]
Not implemented. Much of the information that winfo would return can be got by applying cget to each widget.

wm *operation window* [*arg ...*]
Not implemented.

6.1. References

1. John K Ousterhout, *Tcl and the Tk Toolkit* , Addison-Wesley Publishing Company, Reading, Massachusetts, 1994.
2. Paul Raines and Jeff Tranter, *Tcl/Tk in a Nutshell* , O Reilly, Sebastopol, California, 1999.
3. B W Kernighan, Descent into Limbo , elsewhere in this volume.
4. See *draw-intro* (2), *tk* (2) and *wmlib* (2) in *The Inferno Programmer's Manual* , Volume 1.