VICTORIA UNIVERSITY OF WELLINGTON

Department of Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 471 5328
Fax: +64 4 495 5232
Internet: Tech.Reports@comp.vuw.ac.nz

Using the Tk Canvas Facility

Robert Biddle

Technical Report CS-TR-94/5
May 1994

Abstract

Most GUI toolkits allow the programmer to create interfaces with a variety of standard components, but sometimes programs need interfaces with custom designed pictorial components. The TCL/Tk language provides for this need with the "canvas" facility. This allows the programmer to create interface components which are constructed like drawings, but which allow dynamic reconfiguration and provide for full user interaction. This document is a practical introduction to using the canvas facility.

Publishing Information

This was originally published as: Biddle, R., "GUI Display of Data with TCL/Tk: Using the Tk Canvas Facility", In *Proceedings of the 11th Uniforum NZ Conference*, Uniforum NZ, 1994.

GUI Display of Data with TCL/Tk

Using the Tk Canvas Facility

Robert Biddle
Department of Computer Science
Victoria University of Wellington
`robert.biddle@comp.vuw.ac.nz`

Most GUI toolkits allow the programmer to create interfaces with a variety of standard components, but sometimes programs need interfaces with custom designed pictorial components. The TCL/Tk language provides for this need with the "canvas" facility. This allows the programmer to create interface components which are constructed like drawings, but which allow dynamic reconfiguration and provide for full user interaction. This document is a practical introduction to using the canvas facility.

1 Introduction

Graphical user interfaces allow programs to communicate with people by displaying pictures, and allow people to communicate with programs by interacting with these pictures. Since their early development, several particular kinds of pictures and interactions have become accepted as suitable for general use. It is now entirely commonplace for program interfaces to feature "buttons", "check boxes", "scroll bars", etc. However, such general interface techniques are not always enough, and are not always appropriate. Many programs can be made dramatically clearer and easier to use if they have an interface with custom designed components for graphical display and interaction. This document is an introduction to creating such custom interface components in TCL/Tk, by using the Tk "canvas" facility.

TCL (Tool Command language) is a small language designed to be embedded into other systems to enable simple yet flexible extensibility and customisation. Tk is an X window system toolkit providing graphic user interface and event driven facilities in connection with TCL. TCL and Tk are principally the work of John Ousterhout at the University of California, Berkeley. Further references, and information about obtaining TCL/Tk, are given at the end of this document. Probably the most significant aspect of TCL/Tk is that despite offering great flexibility, it is astonishingly easy to use. Programs may be developed very quickly, and are small for what they do. The language seems to emphasise immediate practicality over sophisticated theory, though there is subtle and unobtrusive design involved.

This introduction is intended to be very practical. Source code is given frequently, and diagrams show the graphical results. However, it is more difficult to show interactivity in a static document. Moreover, it is difficult for a document to impart the sensation of small easy-to-write programs producing impressive results. For the full effect, it is necessary to run the programs, modify the programs, and write new programs. I hope this document will provide some guidance for such exploration. TCL/Tk is available free, as is all program code in this document — details are at the end of the document.

2 TCL/Tk Basics

This section is a brief overview of TCL/Tk programming, in order to make later section more understandable to those new to the language. For more details, consult the references: my 1992 tutorial provides a quick but more comprehensive introduction; Ousterhout's 1994 book is more lengthy and definitive.

TCL is a small language with only one basic construct (procedure call), one basic data type (string), and one basic data structure (associative array). However, many procedures are built into the language, and the syntax for procedure calls is simply the name of procedure and a list of blank separated argument strings. In this way, the built-in procedures provide the basic statements of the language: `if`, `while`, and so on. The `set` procedure is used to associate names with values, and names are evaluated if preceded by a dollar-sign. Other built-in procedures provide facilities for processing strings, file input/output, and so on. Procedures are called just by giving the name and a list of arguments; enclosing the call in square brackets makes the procedure's return value available. The `expr` procedure is used for arithmetic and boolean expressions. For example, to initialise a variable `num`, the statement would be `set num 0`; the typical statement to increment `num` would be `set num [expr "$num + 1"]`. Alternatively, the increment function could be used, and the statement would then just be `incr num`.

Statments may be grouped together with braces, leading to a syntax that looks familiar to C programmers. Procedures can be specified in TCL itself, using the `proc` procedure, with parameters and local variables also provided for. In practice, the standard TCL procedures (and Tk procedures described below) are usually referred to as "commands".

The TCL language is interpreted, and TCL procedures are acted upon and arguments evaluated as they are recognised at run-time. While TCL is a complete language, it is designed to work together with software written in C. This is accomplished by the C program using a TCL run-time library to interact with the TCL environment, and to call the TCL interpreter. The principal ways a C program can interact with the TCL environment are by calling TCL procedures via a TCL library function, and by installing C functions that can be called from TCL as procedures. In such ways, TCL may be extended and customised for any particular application. However, it remains possible to create useful programs that only use TCL and Tk: if you have a TCL/Tk program called `myprog.tcl`, for example, it can be interpreted directly using the "wish" (window shell) command `wish -f myprog.tcl`. (All the examples programs in this document can be run this way.)

The Tk toolkit of graphical user interface facilities consists simply of additional TCL procedures, as follows. Firstly, there is a procedure to create each kind of standard graphical component (called a "widget"). The commands to create widgets are `button`, `checkbutton`, `scrollbar`, etc., and there are options to specify various details of their appearance or functionality. It is possible to affect the appearance and behaviour of widgets after their creation by using the names of individual widgets as if they were commands. Secondly, there are procedures to manage exactly how these components are mapped to the screen and displayed together: `pack` and `place`. Thirdly, there is a single command, `bind`, to link together a widget, an X event (such as a mouse click or keyboard character) and TCL statements to execute when the event occurs. These are the three central kinds of Tk procedures — widget procedures, pack or place, and bind — there are also some miscellaneous support procedures.

In this document, we are concerned not with standard graphical components, but with designing custom components. In Tk, this is made possible by a single kind of general purpose component: the "canvas" widget.

3 The Canvas Widget

In many ways, the canvas widget is like any other widget: it is created with the `canvas` command, may be configured and reconfigured, must be mapped to the screen with pack or place, and the bind command may be used to manage X events within its bounds. The canvas widget can be used together with the other Tk components for creating a complete program interface. But a canvas is initially a very boring widget, just displaying a blank rectangle.

Like all Tk widgets, a canvas may be manipulated and reconfigured at run-time. However, the

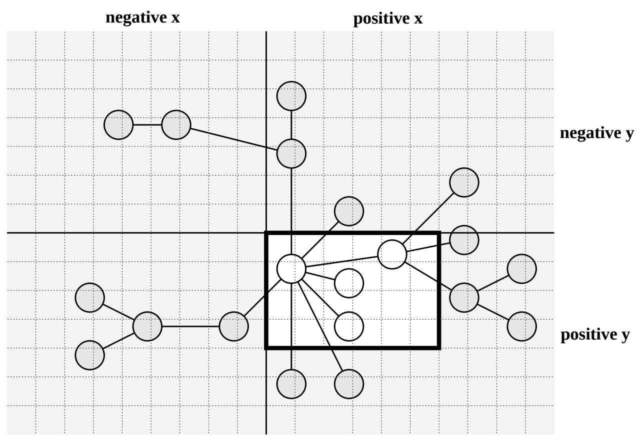


Figure 1: *Although a canvas widget displays a limited rectangular window, it is only part of an unlimited underlying 2 dimensional coordinate space. Items may be placed anywhere on the canvas, and the limited window may be moved or resized.*

variety of possibilities for the canvas widget is much richer than with other widgets. A canvas widget is essentially a two dimensional coordinate space. On this plane various kinds of pictorial "items" may be positioned. These items are of a much more general nature that the standard widgets. Yet the items on a canvas may also be reconfigured arbitrarily, and may also be associated with X events. In this way, the canvas widget allows custom interface components for display and interaction.

While a canvas widget is displayed on the screen as a rectangle, the two dimensional coordinate space of a canvas widget is unbounded. The screen display is just a rectangular window of some size, positioned some place on the plane. The size and location of the window are configurable and may be changed at run-time. The coordinate space is addressed in terms of the usual horizontal axis ("x" axis) and vertical axis ("y" axis). The x coordinates increase from left to right, as usual, but the the y coordinates increase from top to bottom. This is unlike cartesian geometry, but is like many screen graphics models (including the X window system). Canvas dimensions may be specified in pixels, centimeters, or inches, and may be floating point quantities — making complicated screen calculations easier. A canvas widget is created initially with the rectangular window positioned so the top left of the window is coordinate $(0, 0)$. (See figure 1.)

As with all Tk widgets, a canvas widget is manipulated with two commands: the general `canvas` command, and the command specific to a particular canvas. The `canvas` command takes a name and any initial configuration options, and creates the new canvas widget. The name of the new widget is then itself the command for manipulating that specific widget. A specific canvas command is multi-functional, and depends on its first argument, which in effect specifies a sub-command. One sub-command is `configure`, which is used to dynamically reconfigure general specifications of the widget when necessary. The basic configuration options for a canvas are `-width` and `-height` (followed by the size), and `-background` (followed by a colour).

4 Canvas Items

Canvas "items" are flexible graphical components that can be created and positioned on a canvas, bound to events, and reconfigured as necessary. The basic set roughly corresponds to the drawing elements in many structured drawing programs: rectangle, oval, arc, line, polygon, text, and bitmap. Items are created using the specific canvas command, with the `create` sub-command. Initial coordinates of the item follow, and any initial configuration specifications for the new item itself. The following table lists how coordinates and options are specified, and figure 2 gives examples of item configuration and the resulting graphical effects.

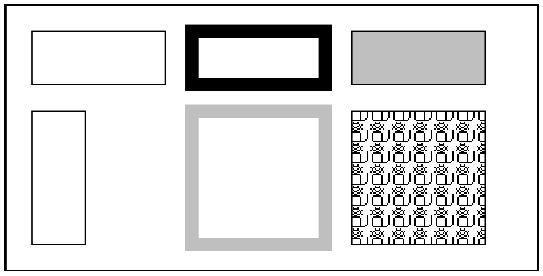
If item locations overlap, items created later are placed on top — but this may be controlled specifically with the `raise` and `lower` sub-commands. It is possible to create items at any time, and it is also possible to destroy items at any time with the `delete` sub-command.

Item Type	**Coordinates Specifications**
`rectangle`	x_1 y_1 x_2 y_2 *(top left and bottom right corners)* `-width` *outlinesize* `-stipple` `@`*bitmapfile* `-outline` *Xcolourname* `-fill` *Xcolourname*
`oval`	x_1 y_1 x_2 y_2 *(corners of containing rectangle)* `-width` *outlinesize* `-stipple` `@`*bitmapfile* `-outline` *Xcolourname* `-fill` *Xcolourname*
`arc`	x_1 y_1 x_2 y_2 *(corners of containing rectangle)* `-style` *arcstyle* *(either* `pieslice`, `chord`, *or* `arc`*)* `-start` *degrees* `-extent` *degrees* `-width` *outlinesize* `-stipple` `@`*bitmapfile* `-outline` *Xcolourname* `-fill` *Xcolourname*
`line`	x_1 y_1 x_2 y_2 $\ldots x_n$ y_n *(points on line)* `-arrow` *headend* *(either* `first`, `last`, *or* `both`*)* `-smooth` *boolean* *(either* `0`, *or* `1`*)* `-width` *outlinesize* `-stipple` `@`*bitmapfile* `-fill` *Xcolourname*
`polygon`	x_1 y_1 x_2 y_2 $\ldots x_n$ y_n *(vertices of polygon)* `-smooth` *boolean* *(either* `0`, *or* `1`*)* `-fill` *Xcolourname* `-stipple` `@`*bitmapfile*
`text`	x y *(anchor point for the text)* `-text` *actualtext* `-font` *Xfontname* `-justify` *orientation* *(either* `left`, `right`, *or* `center`*)* `-width` *blocksize* *width of the block of text* `-anchor` *compasspoint* *(one of:* `center n s e w ne nw se sw`*)* `-fill` *Xcolourname* `-stipple` `@`*bitmapfile*
`bitmap`	x_1 y_1 x_2 y_2 *(anchor point for the bitmap)* `-bitmap` `@`*bitmapfile* `-anchor` *compasspoint* *(one of:* `center n s e w ne nw se sw`*)* `-background` *Xcolourname* `-foreground` *Xcolourname*

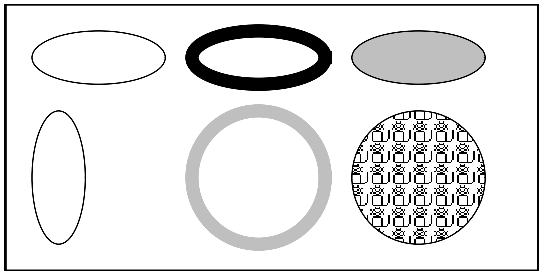
5 Canvas Item Addressing

A canvas can include a large number of items of the various kinds, and managing the all the items is a serious concern. Every time the item creation sub-command is called, it returns a numeric item "id" which is the unique and permanent identifier for the newly created item. Such an identifier is sometimes necessary, but it is often less than helpful. To start with, the number just has to be recorded in a variable, resulting in a significant data management exercise. Moreover, there is frequently a need for identifiers that can specify several items at the same time: items in a structured group, for example, or items in related roles throughout the canvas. To meet these needs, canvas items may be given "tags" for use in later identification. Tags are usually set when items are created by using the `-tag` option of the item creation sub-command. A tag can be any string, but cannot begin with a digit, because ids and tags are used in the same syntax, and must be distinguishable. The `-tag` option may be used to associate *several* tags with an item, by specifying a list of tags. It is useful for items to have a unique tag, a tag in common with a structured group, or a tag in common with other similar items around the canvas. The tag lookup

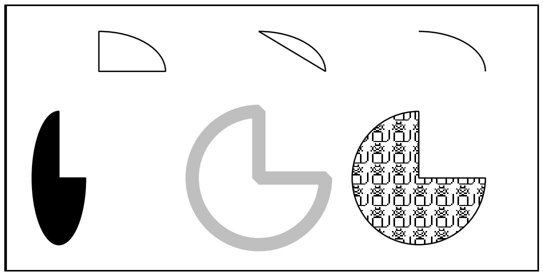
```
canvas .demo -width 400 -height 200
pack .demo

.demo create rectangle 20 20 120 60
.demo create rectangle 140 20 240 60 -width 10
.demo create rectangle 260 20 360 60 -fill grey
.demo create rectangle 20 80 60 180
.demo create rectangle 140 80 240 180 -width 10 -outline grey
.demo create rectangle 260 80 360 180 -fill black -stipple @cat
```



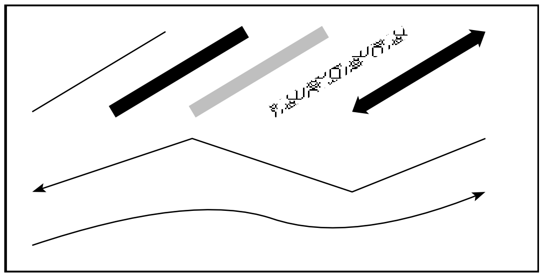
```
canvas .demo -width 400 -height 200
pack .demo

.demo create oval  20 20 120  60
.demo create oval 140 20 240  60 -width 10
.demo create oval 260 20 360  60 -fill grey
.demo create oval  20 80  60 180
.demo create oval 140 80 240 180 -width 10 -outline grey
.demo create oval 260 80 360 180 -fill black -stipple @cat
```



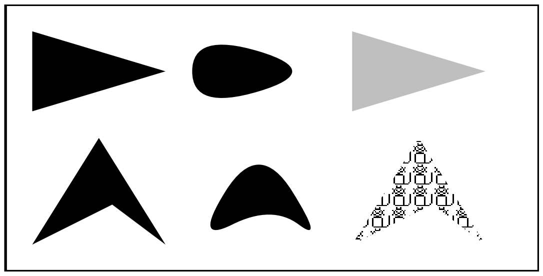
```
canvas .demo -width 400 -height 200
pack .demo

.demo create arc 20 20 120 80 -start 0 -extent 90
.demo create arc 140 20 240 80 -start 0 -extent 90 -style chord
.demo create arc 260 20 360 80 -start 0 -extent 90 -style arc -fill black
.demo create arc 20 80 60 180 -start 90 -extent 270 -fill black
.demo create arc 140 80 240 180 -start 90 -extent 270 -width 10 -outline grey
.demo create arc 260 80 360 180 -start 90 -extent 270 -fill black -stipple @cat
```



```
canvas .demo -width 400 -height 200
pack .demo

.demo create line 20 80 120 20
.demo create line 80 80 180 20 -width 10
.demo create line 140 80 240 20 -width 10 -fill grey
.demo create line 200 80 300 20 -width 10 -stipple @cat
.demo create line 260 80 360 20 -width 10 -arrow both
.demo create line 20 140 140 100 260 140 360 100 -arrow first
.demo create line 20 180 140 140 260 180 360 140 -smooth 1 -arrow last
```



```
canvas .demo -width 400 -height 200
pack .demo

.demo create polygon 20 20 20 80 120 50
.demo create polygon 140 20 140 80 240 50 -smooth 1
.demo create polygon 260 20 260 80 360 50 -fill grey
.demo create polygon 70 100 20 180 80 150 120 180
.demo create polygon 190 100 140 180 200 150 240 180 -smooth 1
.demo create polygon 310 100 260 180 320 150 360 180 -stipple @cat
```



```
canvas .demo -width 400 -height 200
pack .demo

.demo create text 100 20 -text Victoria
.demo create text 100 40 -text Victoria -anchor e
.demo create text 100 60 -text Victoria -anchor w
.demo create text 100 80 -text Victoria -fill grey
.demo create text 100 110 -text Victoria -font -*-times-*-i-*-*-*-200-*-*-*-*-*-*
.demo create text 100 160 -text "Victoria\nUniversity\nof Wellington"
.demo create text 200 160 -text "Victoria\nUniversity\nof Wellington" -justify center
.demo create text 300 160 -text "Victoria\nUniversity\nof Wellington" -justify right
.demo create bitmap 250 70 -bitmap @vuw.bitmap
```

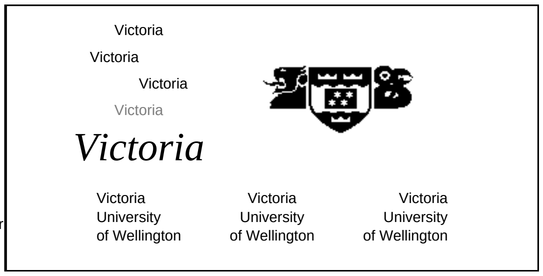

Victoria
Victoria
Victoria
Victoria
Victoria
Victoria University of Wellington
Victoria University of Wellington
Victoria University of Wellington

Figure 2: *Creation commands on canvases illustrating the main configuration options for rectangles, ovals, arcs, lines, polygons, text and bitmap each with the graphical results.*

```
canvas .hw -width 200 -height 200
pack .hw

.hw create oval 50 50 150 150 -fill khaki -tag face
.hw create oval 70 80 85 95 -fill lightblue -tag "face eye"
.hw create oval 115 80 130 95 -fill lightblue -tag "face eye"
.hw create arc  70 75 85 90 -style arc -fill black -start 45 -extent 90 -tag "face brow"
.hw create arc 115 75 130 90 -style arc -fill black -start 45 -extent 90 -tag "face brow"
.hw create line 75 125 100 125 125 125 -smooth 1 -tag "face mouth"

.hw bind face <Enter> { .hw move brow 0 -10 }
.hw bind face <Leave> { .hw move brow 0 10 }

.hw bind face <ButtonPress-1> {
  .hw coords mouth 75 125 100 145 125 125
  .hw create text 100 25 -anchor c -text "Hello World!"
}

bind .hw <ButtonPress-3> { exit }
```

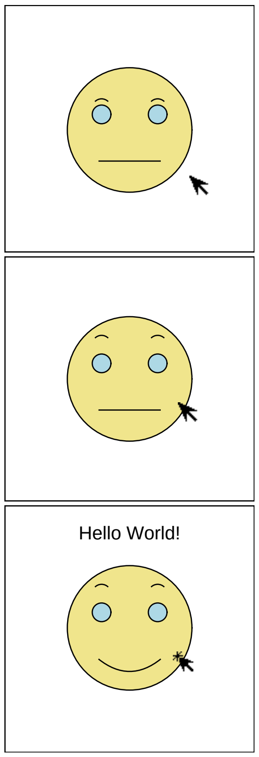


Figure 3: *A complete example program using a canvas widget. This program creates a picture of a face (top), and makes some event bindings. The eyebrows raise when the cursor enters the face region (middle), and lower again when the cursor leaves. If mouse button 1 is pressed while the cursor is in the face region, the mouth changes to a smile, and the message is displayed (bottom).*

mechanism is carefully designed to be efficient, so it is reasonable to use a tag to address a single item, or many items all at once. See figure 3 for a simple example of the use of tags, and figure 4 for a more detailed example.

The tagging facility is quite flexible, and there are several sub-commands for tag management, for example: `addtag` to attach tags to an item, `dtag` to delete tags from an item, and `gettags` to return the tags associated with an item.

6 Dynamic Canvas Reconfiguration

As with all Tk widgets, a canvas may be reconfigured at any time. For the canvas itself, this means that changing the width and height of the canvas window, or the background colour, is simply a matter of using the specific canvas command and the `configure` sub-command.

Individual items on a canvas may also be reconfigured at any time. The general purpose way to change item specifications is to use the `itemconfigure` sub-command. In this way, any of the item specifications may be changed.

It is also possible to change the coordinates of an item, using the `coords` sub-command. By using this it is possible to change the shape, size, and location of an item on the canvas, just by specifying completely new coordinates. If no new coordinates are specified, this same sub-command returns the current coordinates. Another sub-command, `bbox`, is useful for returning the coordinates of a bounding box around any item or set of items.

Two well-defined ways in which coordinates change are when the item size is scaled, and when it is moved without changing shape. For scaling, an alternative is to use the the `scale` sub-command. This requires two (x, y) pairs to be specified, one to fix the origin of the scaling, and the second to specify the scale factor, with x and y factors independent. Motion of canvas items is discussed at length later in this document.

7 Canvas Item Events

As with all ordinary Tk widgets, any X event within a canvas widget may be associated with some desired behaviour by using the `bind` command. This is sometimes useful when the event concerns the entire canvas. Sometimes, though, an event only concerns part of a canvas. The specific canvas command with the `bind` sub-command may be used to specify this. In fact, once the item is specified, the rest of the sub-command is exactly like the `bind` command itself: the event must be specified in angle brackets, followed by the TCL code to be executed when the event occurs. As with the bind command, the TCL code specified may use various pseudo-variables (beginning with a percent sign) to retrieve information about the event.

All of the kinds of canvas items may be bound to various kinds of events. However, the `text` item type is usually the context for keyboard events, and the more pictorial item types are usually the context for mouse and mouse-button events.

With the more ordinary widgets, there are several default bindings for mouse events that are very helpful to the user: for instance, the `<Enter>` event usually causes the background colour of widgets to change, alerting the user that the cursor has entered that particular widget — the `<Leave>` event causes the background colour to revert to normal. (See figures 3 and 4 for some alternative bindings.) Because canvas graphics are custom designed, no such sensible defaults are possible. Accordingly, it is important that the programmer provide such behaviour specifically. This is a situation where it is often useful to have associated a single tag for several items in a group. For example, when the cursor enters any one of the items, the tag can be used to change the colour of the entire group of widgets together. This can be especially important when several canvas items are used to graphically represent something that is conceptually unified in the interface.

Modifying the cursor shape is also useful for assisting the user to understand operations on the canvas. Tk associates cursor shape with entire widgets: any widget may be configured with a cursor shape, and Tk will use that shape when the cursor is positioned on that widget. It is not possible to automatically associate cursor shapes with individual items, but it is possible to use the `<Enter>` and `<Leave>` events to reconfigure the shape of the cursor for the containing canvas widget. Any of the shapes in the X cursor font may be used, as may any bitmap file.

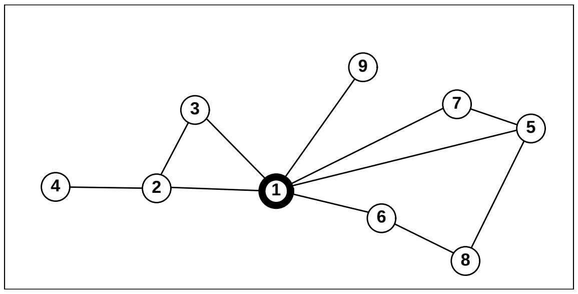
For text items, there are several sub-commands that are especially provided. Following the way that the separate text widget is used, there are facilities to insert characters (`insert`), delete characters (`dchars`), return the numerical index of characters (`index`), insert the cursor at a specific position (`icursor`), and manipulate the X "selection" for copy-and-paste operations (`select`). For any keypress events to be recognised, the canvas and any particular item must have the X "focus" directed to them. For the canvas as a whole, the Tk `focus` command must be used; for particular items the specific widget sub-command `focus` must be used.

8 Canvas Item Motion

To move a canvas item when you know the new coordinates the item should have, the `coords` sub-command is easiest and most appropriate. When you don't know the coordinates but do know the offset from the current position, it is easier to use the `move` sub-command, which simply takes the x and y offsets (positive or negative) to apply. Really, either sub-command can be used in both situations: it's just a matter of choosing which one requires the least amount of other coordinate processing. Moreover, there are several situations which involve item motion that need particular consideration.

```
canvas .net -width 400 -height 200 -background white
pack .net

bind .net <ButtonPress-1> {makenode %x %y}
bind .net <ButtonPress-2> {startline %x %y}
bind .net <ButtonRelease-2> {stopline %x %y}

set nodes 0; set lines 0

proc makenode {x y} {
  global nodes
  set nodes [expr "$nodes+1"]

  set x1 [expr "$x-10"]; set y1 [expr "$y-10"]
  set x2 [expr "$x+10"]; set y2 [expr "$y+10"]

  .net create oval $x1 $y1 $x2 $y2 -tag node$nodes -fill white
  .net create text $x $y -anchor c -text $nodes -tag node$nodes
  .net bind node$nodes <Enter> ".net itemconfigure node$nodes -width 5"
  .net bind node$nodes <Leave> ".net itemconfigure node$nodes -width 1"
}

proc startline {x y} {
  global linex1 liney1
  set linex1 $x; set liney1 $y
}

proc stopline {x y} {
  global lines linex1 liney1
  set lines [expr "$lines+1"]

  .net create line $linex1 $liney1 $x $y -tag line$lines
  .net bind line$lines <Enter> ".net itemconfigure line$lines -width 5"
  .net bind line$lines <Leave> ".net itemconfigure line$lines -width 1"
}
```

Figure 4: *This is the complete source code for a program to allow the user to draw network diagrams, such as the one shown below. Pressing mouse button 1 creates a node, and dragging with mouse button 2 create a line. Nodes are labeled with ascending numbers in this example, but could take input from the keyboard or elsewhere. The width of lines and node outlines is changed to show enter and leave events. Note especially the use of tags to distinguish each node and line, and way that the* `bind` *sub-command is called for each new item created. (This program has some limitations which are addressed in later examples — note for example that lines may be drawn anywhere, not just between nodes.)*



```
proc makenode {x y} {
  # same main part of procedure as before
  # ...
  .net bind node$nodes <ButtonPress-3> "beginmove %x %y"
  .net bind node$nodes <B3-Motion> "domove node$nodes %x %y"
}
proc stopline {x y} {
  # same main part of procedure as before
  # ...
  .net bind line$lines <ButtonPress-3> "beginmove %x %y"
  .net bind line$lines <B3-Motion> "domove line$lines %x %y"
}
proc beginmove {x y} {
  global movex movey
  set movex $x; set movey $y
}
proc domove {item x y} {
  global movex movey
  .net move $item [expr "$x - $movex"] [expr "$y - $movey"]
  set movex $x; set movey $y
}
```

Figure 5: *These additions to the earlier network drawing program allow both nodes and lines to be moved about the canvas, by dragging them with mouse button 3. Note that because the node circle and label share the same tag, one move command affects both together.*

8.1 Dragging

One common situation occurs when the item is being moved with the mouse cursor to effect "dragging". In this situation you cannot predict where the item will end up, so you have to constantly follow the cursor to provide feedback to the user. This is normally done by specifying the items with binding to the `<ButtonPress>` and `<Motion>` events. In both cases, the critical information is given by the event pseudo-variables `%x` and `%y` which indicate the coordinates of the cursor in the X window system coordinate space. Usually the strategy is to use the `<ButtonPress>` binding to initialise variables to record the current coordinates, and then use the `<Motion>` event to move the widget and update the variables recording the current coordinates. (Because the line position is always set with the cursor motion, there is no particular action that needs to be associated with the ¡ButtonRelease¿ event.) An example of program code is shown in figure 5. Notice that the user will expect to be able to drag the item from any part of its shape, and for larger items it is important not to confuse the coordinates of the cursor with the coordinates of the item. Often it's best to work with offsets calculated by changes in the cursor coordinates, and then use `move`, rather than `coords`.

A special case of dragging is in drawing lines. Typically one endpoint of the line is determined by first pressing the mouse button, and then the mouse is dragged to the other endpoint. The drawing program in figure 4 allows this, but does not display the line until the second endpoint is determined. A better alternative is to draw the line immediately as the mouse button is first pressed, treating both endpoints as being in the same place. Then while the mouse remains in motion, the second endpoint can be updated, and the line re-displayed. This appears as if the line was a rubber band with one end fixed and the other end being moved. The method is useful because it allows the user to see the visual effect of the line before its final position is determined, allowing the user to make fine adjustments easily. Figure 6 shows code adding "rubber band" line drawing to the drawing program shown earlier.

8.2 Animation

Dragging is the common way for the user to specify movement of items, but sometimes it is the program that needs to specify movement. This can be done directly in one step, via the `move`

```
bind .net <ButtonPress-2> {beginline %x %y}
bind .net <B2-Motion> {doline %x %y}
proc beginline {x y} {
  global lines linex1 liney1
  set lines [expr "$lines+1"]
  set linex1 $x; set liney1 $y
  .net create line $x $y $x $y -tag line$lines
  .net bind line$lines <Enter> ".net itemconfigure line$lines -width 5"
  .net bind line$lines <Leave> ".net itemconfigure line$lines -width 1"
  .net bind line$lines <ButtonPress-3> "beginmove %x %y"
  .net bind line$lines <B3-Motion> "domove line$lines %x %y"
}
proc doline {x y} {
  global lines linex1 liney1
  .net coords line$lines $linex1 $liney1 $x $y
}
```

Figure 6: *These changes to the network drawing program allow lines to be shown as they are being positioned. This "rubber band" effect is accomplished by creating the line at its starting point, and then updating its coordinates while mouse button 2 remains in motion.*

or **coords** sub-commands mentioned earlier. At times, however, it is useful to show several steps along the way: using this primitive animation to keep the user aware of what is happening. There are several considerations necessary to do this effectively.

Firstly, the movement must be divided into several smaller movements. The size of these smaller movements should be uniform in time regardless of the direction or magnitude of the movement, if smooth animation is to be achieved. Where the ultimate destination has already been determined, the smaller movements must be calculated by finding the magnitude of the total movement (recalling the pythagorean relationship), and dividing it some number of uniform size movements in the same direction.

Secondly, these small movements must be carried out at uniform time intervals. Tk includes the **after** command that schedules a procedure to be executed after a given time interval (specified in milliseconds). By using the internal Tk scheduling facilities, the command allows pauses in behaviour with efficiency and without disabling other Tk concurrency and event handling. For motion with animation the **after** command can be used to schedule the small movements to be made at a constant rate until the entire movement is complete. Figure 7 shows a small program to demonstrate this method.

8.3 Moving Items Together

In designing pictorial representations with the canvas widget, we frequently need to use several items to graphically represent a single concept. Accordingly, we sometimes have to move such groups of items together without destroying the illusion of unity. One advantage of using the **move** sub-command is that the movement offsets that need to be specified are usually the same for each item in a group. If the items in the group have been given a group tag, then the movement of the entire group can be achieved with just one efficient sub-command.

Sometimes individual items can have requirements that need special handling, and this means it is necessary to carefully move the items individually. Normally, even when widgets or canvas items are being changed or moved, Tk does not actually change their screen appearance until all activity is complete and the interpreter idle. This allows several changes to be made "all at once", which is often desirable. Where this is *not* desirable, because seeing the interim states is intended, the programmer must use the Tk command **update** to force the screen to reflect the current situation.

Sometimes, items or groups to be moved are joined by lines to other items and groups that are

```
canvas .court -width 400 -height 300
pack .court

.court create oval 0 0 50 50  -fill red -tag ball

set dx 10; set dy 10

proc movenext {} {
  global dx dy

  set loc [.court coords ball]
  set bx1 [lindex $loc 0]; set by1 [lindex $loc 1]
  set bx2 [lindex $loc 2]; set by2 [lindex $loc 3]
  if {$bx1 < 0 || $bx2 > 400} {set dx [expr "- $dx"]}
  if {$by1 < 0 || $by2 > 300} {set dy [expr "- $dy"]}
  .court move ball $dx $dy
}

proc move {} { movenext; after 100 move }
move
```

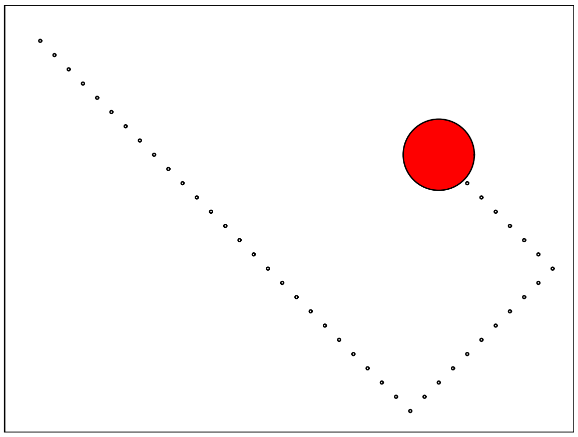


Figure 7: *An example to illustrate program controlled motion: this is a complete program that displays a ball endlessly moving within the rectangular window of a canvas, bouncing off the boundaries of the window as it hits them. Program controlled animation can be useful for illustrating to the user the nature of changes made to graphical structures. (The dotted line was not part of the program, but was added to illustrate the movement of the ball.)*

not being moved (consider network diagrams, for example). To accomplish such a move correctly, the items must be first moved ignoring the joining lines. Secondly, the joining lines should be reconfigured to change only the endpoint joined to the moved items. This is possible, and requires use of the `coords` sub-command — specifying one endpoint unchanged and the other in a new location. It is necessary to calculate the new endpoint location explicitly, though this can usually be done by separately applying the move offset to the former coordinates. This can be done repeatedly so that each of the joining lines appears as a “rubber band” during the move. Program code for this technique is shown in figure 8.

9 Canvas Window Adjustment

As explained earlier, the “real” canvas is unlimited, and the displayed rectangular window only shows part of it. It is often useful to maintain large structures on the canvas, even though only part of them can be visibile at any one time. In order to use this approach, there needs to be a way to adjust the window so that it shows different parts of the underlying canvas. There are in fact two methods, scrolling and scanning, and the ability to resize the window is a related concern. These techniques are dicussed below, and illustrated in figure 10.

9.1 Scrolling

“Scrolling” involves making horizontal and vertical adjustments independently. Frequently only one of them is adjusted, and sometimes no provision is made for adjusting the other. Scrolling the canvas widget uses the same method as is used for the simpler Tk “listbox” widget. This method involves coupling separate “scrollbar” widgets with the canvas, as follows. The scrollbar is typically packed horizontally or vertically alongside the canvas, and is configured specifying a command to execute in accordance with motion in the scrollbar. There are specific canvas sub-commands provided for this purpose: **`xview`** for adjusting the horizontal view, and **`yview`** for adjusting the vertical view. The scrollbar widget shows the placement of the adjustment by means of a contrasting section placed along the graphical bar. In order for this to correctly reflect the situation of the canvas, the canvas must be given a command to execute and report back to

```
proc makenode {x y} {
  global nodes nodectr nodelines
  set nodes [expr "$nodes+1"]
  set x1 [expr "$x-10"]; set y1 [expr "$y-10"]
  set x2 [expr "$x+10"]; set y2 [expr "$y+10"]
  .net create oval $x1 $y1 $x2 $y2 -tag node$nodes -fill white
  .net create text $x $y -anchor c -text $nodes -tag node$nodes
  .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$nodes <ButtonPress-2> "linefrom node$nodes"
  .net bind node$nodes <Shift-ButtonPress-2> "lineto node$nodes"
  .net bind node$nodes <ButtonPress-3> "beginmovenode %x %y"
  .net bind node$nodes <B3-Motion> "domovenode node$nodes %x %y"
  set nodectr(node$nodes,x) $x
  set nodectr(node$nodes,y) $y
  set nodelines(node$nodes) ""
}
set fromnode node1
proc linefrom {node} {
  global fromnode
  set fromnode $node
}
proc lineto {tonode} {
  global fromnode lines nodectr linenodes nodelines
  set lines [expr "$lines+1"]
  set x1 $nodectr($fromnode,x); set y1 $nodectr($fromnode,y)
  set x2 $nodectr($tonode,x); set y2 $nodectr($tonode,y)
  .net create line $x1 $y1 $x2 $y2 -tag line$lines
  .net lower line$lines
  .net bind line$lines <Enter> ".net itemconfigure line$lines -width 5"
  .net bind line$lines <Leave> ".net itemconfigure line$lines -width 1"
  set nodelines($fromnode) "$nodelines($fromnode) line$lines"
  set nodelines($tonode) "$nodelines($tonode) line$lines"
  set linenodes(line$lines,from) $fromnode
  set linenodes(line$lines,to) $tonode
}
proc beginmovenode {x y} {
  global movex movey
  set movex $x; set movey $y
}
proc domovenode {node x y} {
  global movex movey nodectr nodelines linenodes
  .net move $node [expr "$x - $movex"] [expr "$y - $movey"]
  set nodectr($node,x) [expr "$nodectr($node,x) + $x - $movex"]
  set nodectr($node,y) [expr "$nodectr($node,y) + $y - $movey"]
  foreach line $nodelines($node) {
    set x1 $nodectr($linenodes($line,from),x)
    set y1 $nodectr($linenodes($line,from),y)
    set x2 $nodectr($linenodes($line,to),x)
    set y2 $nodectr($linenodes($line,to),y)
    .net coords $line $x1 $y1 $x2 $y2
  }
  set movex $x; set movey $y
}
```

Figure 8: *In the earlier versions of the network drawing program, the nodes and lines were positioned independently. The changes shown here link lines and nodes together, so that lines must start at a node and end at a node. Moreover, when a node is dragged to a new position, any connecting lines are continually adjusted so as to remain properly connected. The data structures to support this are the associative arrays* `nodectr`*,* `nodelines`*, and* `linenodes`*.*

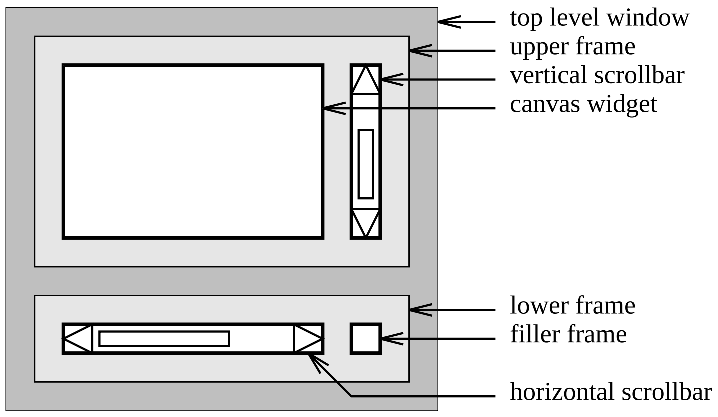


Figure 9: *To arrange a display with both horizontal and vertical scrollbars, "frame" widgets must be used to organise packing. The canvas itself and the vertical scrollbar are packed into the upper frame, the horizontal scrollbar and a filler frame are packed into the lower frame, and both frames are packed into the top level window. (The size of the frames is exaggerated in this diagram — normally only the canvas and the scrollbars themselves would be visible.)*

the scrollbar. The scrollbar provides the command option for this (**`set`**) and the canvas widget provides configuration settings (**`xscrollcommand`** and **`yscrollcommand`**) to specify the command.

There is a canvas widget configuration specification (**`scrollincrement`**) that can be used to control the granularity and speed of scrolling, and another (**`scrollregion`**) to control how much of the real canvas is accessible. This concept of a scrolling region indicates an interesting limitation of the scrolling approach. Scrollbars show "where" the viewing window is positioned, but they can only show this with respect to some specified region — not with respect to the real unlimited canvas. A boolean configuration specification (**`confine`**) controls whether scrollbars limit the user to the scrolling region, or not; when unconfined and beyond the scrolling region, the scrollbars are unable to sensibly indicate the relative position of the viewing window.

A significant implication of adjusting the window position on the underlying canvas involves mouse events. When **`bind`** is used to associate some TCL code with a mouse event, the TCL code is provided with the pseudo-variables **`%x`** and **`%y`** to indicate the position of the cursor. These are *window* coordinates, in pixels relative to the top left of the window. While the canvas window is positioned with the canvas origin at the top left of the window, they are also canvas coordinates. However, if the window has been adjusted to view some other part of the canvas, the window coordinates cannot be used directly to address the canvas. Accordingly, two specific canvas sub-commands are provided to allow easy translation from window coordinates to correct canvas coordinates: **`canvasx`** and **`canvasy`**.

Arranging one scrollbar (either horizontal or vertical) for a canvas is easy, the **`pack`** command can just be used for the scrollbar after it has been used for the canvas. However, arranging both scrollbars requires use of explicit Tk **`frame`** widgets to organise the space before "packing". The structure that must be built is illustrated in figure 9.

9.2 Scanning

"Scanning" involves making both horizontal and vertical adjustments together, and the operation is done directly with the canvas without involving scrollbars or other widgets. It is usually accomplished by the user pressing a mouse button on the canvas itself, and dragging in some direction. The effect is that the real large canvas is moved directly by the user, so that a different part is exposed to view through the rectangular window. The specific canvas command has a special **`scan`** sub-command to provide this facility: when the mouse is first pressed **`scan mark`** is used,

```
# Ask window manager to allow window resizing...
wm minsize . 100 100
wm maxsize . 1500 1500
#
#-------------------------------------------------------------------------
# Set-up for scrolling...
frame .upper
frame .lower
frame .filler
canvas .net -width 400 -height 200 -background white
scrollbar .vscroll -orient vertical
scrollbar .hscroll -orient horizontal
pack .upper -fill both -expand 1
pack .lower -fill x
pack .net -in .upper -fill both -expand 1 -side left
pack .vscroll -in .upper -fill y -side right
pack .hscroll -in .lower -fill x -expand 1 -side left
pack .filler -in .lower -padx 7
.net config -scrollregion "0 0 1500 1500"  -confine 0
.net config -yscrollcommand ".vscroll set" -xscrollcommand ".hscroll set"
.vscroll config -command ".net yview"
.hscroll config -command ".net xview"
#
#-------------------------------------------------------------------------
# Set-up to for scanning...
bind .net <Meta-ButtonPress-1> {beginscan %x %y}
bind .net <Meta-B1-Motion> {doscan %x %y}
proc beginscan {x y} {
  global scanx1 scany1
  set scanx1 $x; set scany1 $y
  .net configure -cursor fleur
  .net scan mark $x $y
}
proc doscan {x y} {
  global scanx1 scany1
  set scanx2 [expr "$scanx1 + (($x - $scanx1) / 10)"]
  set scany2 [expr "$scany1 + (($y - $scany1) / 10)"]
  .net scan dragto $scanx2 $scany2
}
bind .net <Meta-ButtonRelease-1> {.net configure -cursor arrow}
#
#-------------------------------------------------------------------------
# Remember to map from window to canvas coordinates...
bind .net <ButtonPress-1> {makenode [.net canvasx %x] [.net canvasy %y]}
```

Figure 10: *The earlier network drawing program works only with a fixed window on the canvas. These changes add considerable flexibility by allowing resizing, scrolling, and scanning.*

and when the mouse is then dragged **scan dragto** is used. These sub-commands work so that the canvas is moved in the direction of the mouse movement, but at a factor of ten times, to effect "high speed dragging"; of course the programmer can use explicit calculations to change this factor.

It is perfectly acceptably to allow *both* scrolling and scanning: even if the adjustment is made by scanning, the automatic communication between canvas and scrollbar will ensure that the scrollbar will display the correct position.

9.3 Resizing

In addition to being able to adjust the relative placement of the viewing window on the canvas, it is often useful to be able to change the size of the window. Of course, this can be simply done by using the **configure** sub-command to change the **-width** or **-height** specifications. However, it is also useful to allow the user to resize the canvas window using a conventional X window manager to resize the X window that contains the canvas. For this to be possible, it is necessary for the canvas to have been mapped to the screen with the **pack** command with **-fill both** and **expand**

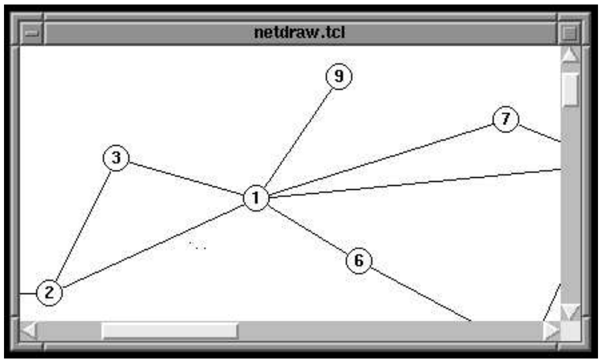


Figure 11: *A screen dump of the network drawing program, showing horizontal and vertical scroll-bars. The next step in development would be to go beyond just drawing.*

`1` specified, indicating it should expand to fill any additional space (both horizontal and vertical) in the containing window. Moreover, it is also necessary to inform the window manager that such flexibility is allowable. The Tk command **`wm`** is provided for communicating with the X window manager, and to indicate flexibility the **`maxsize`** and **`minsize`** sub-commands are used to set the bounds of the flexibilty for the window. It's sensible to set especially the lower bound carefully, so that the window cannot be shrunk so small the display becomes difficult to interpret.

The program may at any time determine the current width and height of the window (in pixels) by using the return values of the Tk commands **`winfo width`** and **`winfo height`**.

10 From Pictures to Applications

This document has introduced all the essential facilities and techniques for using the Tk canvas facility to create custom pictorial user interface components. After reading through this, the next step is to try out the examples yourself, then experiment further, and then tackle building some interfaces for real applications.

However, the document has under-emphasised some points that will become very important in developing more realistic applications. The examples shown have concerned interactive pictures. Sometimes pictures *are* the intended output of a program, and the canvas widget is well-prepared for this: there is a **`postscript`** sub-command that will take any rectangular region of a canvas, and create an image in a file in postscript format for later processing, display, or printing (most of the canvas illustrations in this document were done this way).

But canvases are *not* restricted to the items introduced earlier. Any of the Tk standard components can also be located on a canvas, by using the **`create window`** sub-command. So a canvas can combine interactive pictorial elements with the standard components (buttons, check boxes, etc.) in a completely integrated way. Moreover, for more specialised needs still, the C interface to Tk allows a programmer to design entirely new item types for the canvas. And, of course, the canvas widget will typically be just one part of the complete graphic user interface for an application program.

Most importantly, canvases are *not* merely interactive pictures, because the interaction can control actions of arbitrary complexity. Even from within TCL itself, files may be read and written, and any program may be executed. And by using the simple C language interface, any C function may be called. So interacting with pictures is a completely general way of interacting with computer facilities. The network drawing program used in the examples does offer a reasonable way to draw network diagrams. The diagrams can be displayed, printed, and it is a simple matter to write the coordinates to a file for later manipulation. But the diagram can be used as a general interface, and so could be used to connect to real network data, and a real network. Instead of a network drawing program, it could become a network information program, or a network control program.

As with TCL and Tk generally, the primary weakness of the canvas facility is noticeable when managing large complicated code. For managing complicated composite structures on the canvas, there are really only two methods: the "tag" facility, and data structures using associative arrays. These methods are very flexible, but for more complex tasks require careful program planning. Some better way of building the structures into the canvas would probably be a good idea. More generally, while the interpreted nature of TCL speeds early development because of the flexible nature of character strings, it means there is no compile time type checking, and so sometimes errors manifest themselves late in development.

As with TCL and Tk generally, the primary strength of the canvas facility is that it makes complicated things easy to do. Impressively powerful interfaces can be built surprisingly quickly. Essentially, the facility allows the interface programmer to construct interactive pictures, and so allows the interface designer great flexibilty. An immediate application of this flexibility is that many common pictorial representations of systems and data can be directly used in the user interface: network diagrams, directory trees, maps, circuits, and so on. In the longer term, this flexibility will allow and encourage designers to come up with new pictorial representations for many situations and applications. If a picture is worth a thousand words, a dynamic interactive picture should be even more attractive.

Figure 12: *A screen dump of the interface for a complete application, showing a Tk canvas widget used for network drawing. This is from the program "tkinded", an extensible network editor with TCP/IP network query facilities built-in. (From Schönwälder, J., and Langendörfer, H., "How to Keep Track of Your Network Configuration", In Proceedings of the 7th Large Installation System Administration Conference, The Usenix Association, 1993.)*

11 Further Information

11.1 References

John Ousterhout created TCL and Tk, and wrote the main documents describing them:

> Ousterhout, J. K., "Tcl: An Embeddable Command Language", in *Proceedings of the Winter Usenix Conference*, The Usenix Association, 1990.
>
> Ousterhout, J. K., "An X11 Toolkit Based on the Tcl Language", in *Proceedings of the Winter Usenix Conference*, The Usenix Association, 1991.

These two papers provide an introduction to the aims and design philosophy behind Tcl and Tk, make comparisons with related work, and indicate future directions for development. They are not, however, suitable as either detailed reference material for programming with TCL and Tk, or as a practical introduction.

For a quick practical introduction to using TCL and Tk, my 1992 tutorial is still worth consideration:

> Biddle, R. L., "Graphic User Interfaces Made Easy: A Tcl/Tk Tutorial", In *Proceedings of the 9th Annual Conference of Uniforum New Zealand*, UniForum NZ, New Zealand Unix System User Group, 1992.

A revised version of this paper is also available as a Technical Report from the Department of Computer Science, Victoria University of Wellington.

The canvas facility was introduced about the same time as the extended text management facility. There are no introductory tutorial to these, but their essential design was discussed by Ousterhout in the following paper:

> Ousterhout, J. K., "Hypertext and Hypergraphics in Tk", In *Proceedings of the 7th Annual X Technical Conference*, MIT X Consortium, 1993.

Since 1992 Ousterhout has been working on a comprehensive book about TCL and Tk. This is being published this year:

> Ousterhout, J. K., *Tcl and the Tk Toolkit*, Addison-Wesley, 1994.

A Usenet news group `comp.lang.tcl` has existed since the beginning of 1992, and is a good source of news and advice about TCL and Tk. The group has a comprehensive FAQ (list of answers to frequently asked questions) posted periodically.

11.2 Software Availability

The complete software and reference documentation for TCL and Tk are available free, as are many useful utility programs and demonstrations. The programs are written in C; the TCL distribution works on practically any platform, and the Tk distribution requires the X window system libraries. (There are efforts underway to port Tk to graphics systems other than X, but nothing is available yet.)

In New Zealand, the TCL and Tk distributions are available from the Computer Science department of Victoria University of Wellington: if on the Internet, use *ftp* to connect to `ftp.comp.vuw.ac.nz` and see the directory `/pub/languages/tcl`; otherwise contact the department for advice. The main ftp site for the distributions is `ftp.cs.berkeley.edu`.