The canvas widget in the Tk toolkit is a powerful free software tool for creating and manipulating structured graphics.

The Tk Canvas Widget

The canvas widget in the Tk Graphical User Interface toolkit is a free software tool used to present graphical data. Like the Tk text widget, which I discussed in my previous article, the canvas widget is accessible from most modern scripting languages including Tcl, Perl and Python. It provides those languages with a best of breed facility for structured graphics work.

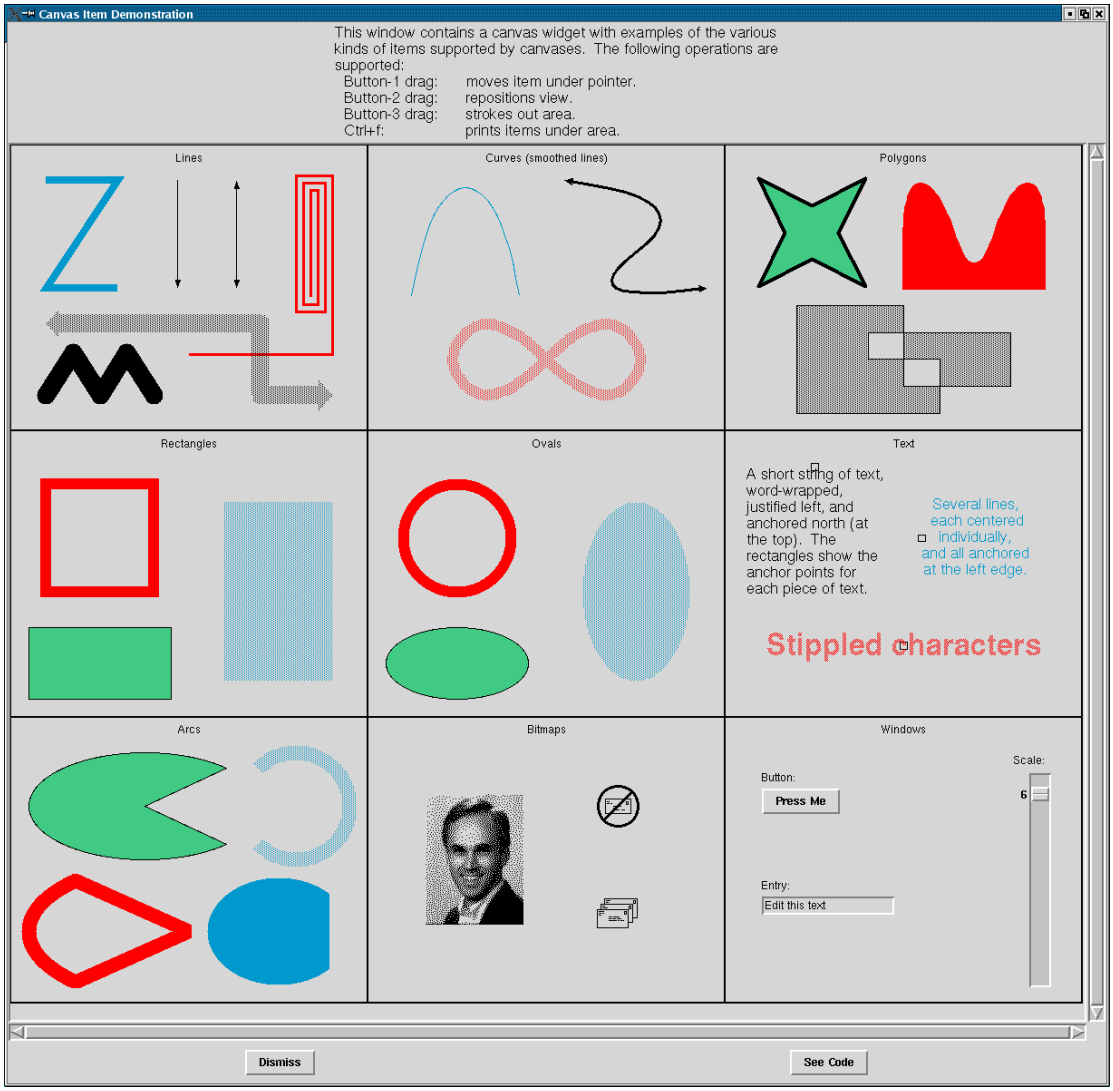
A canvas widget can be thought of as a blank piece of paper, upon which can be drawn lines, shapes, text, images and other Tk widgets. These items, once drawn, can be reconfigured in order to change their positions, colour, size, format or contents. They can also be given abilities to respond to inputs from the user or to react to changes to data manipulated elsewhere in the script.

The canvas widget isn't a solution to any particular problem; rather it is an extremely flexible tool which allows developers to quickly and easily build solutions to all sorts of problems. This article will describe some of the facilities it provides, and suggest some of the uses those facilities can be put to.

Items - The Building Blocks

The basic idea when using the canvas widget is to draw on it what are known as items. An item can be a line, an image, some text, one of a number of geometric shapes, and so on. See the sidebar for a description of the item types which can be drawn and manipulated. Figure 1 shows a screenshot of a canvas widget displaying some of these items.

Item type	Description
Arc	An arc shaped region, empty or filled
Bitmap	A simple, two colour image as is often used for an icon
Image	A full colour image such as a JPEG image
Line	A line or sequence of lines, straight or bezier smoothed
Oval	An oval or circle, empty or filled
Polygon	A multi-sided, irregular shaped region, empty or filled
Rectangle	A rectangle or square, empty or filled
Text	Some text, either static or editable
Window	A Tk widget or set of widgets
Other	A user defined item type which must be coded in C



Exactly how an item is drawn depends on the options it is configured with. There are many dozens of options available and this article will discuss and demonstrate some of them. See the canvas widget man page for a comprehensive list of all the options available.

IDs, Tags and Binding

When an item is drawn on a canvas it becomes an independent entity. Each individual item is given a unique ID which the developer can use to reconfigure the item's options. When any of an item's options are changed, the canvas widget will ensure the item is redrawn with its new configuration. Not only does this redrawing happen immediately, it is also quick enough to support animations and direct mouse control of hundreds or possibly thousands of items.

As well as their unique Tk-assigned ID, items can also be tagged with one or more names chosen by the developer. Tags work in the same way as those offered by the Tk text widget. As well as providing an easier way to reference a single item (a sensible name instead of a number), this tagging mechanism also allows items to be logically grouped together. All items given the same tag can be treated as one single item.

Once an item has been given one or more tags, the canvas widget allows pieces of code to be bound to those tags. This is the feature of the canvas widget which enables dynamic behaviour. I covered the tag and bind approach in some detail in my article

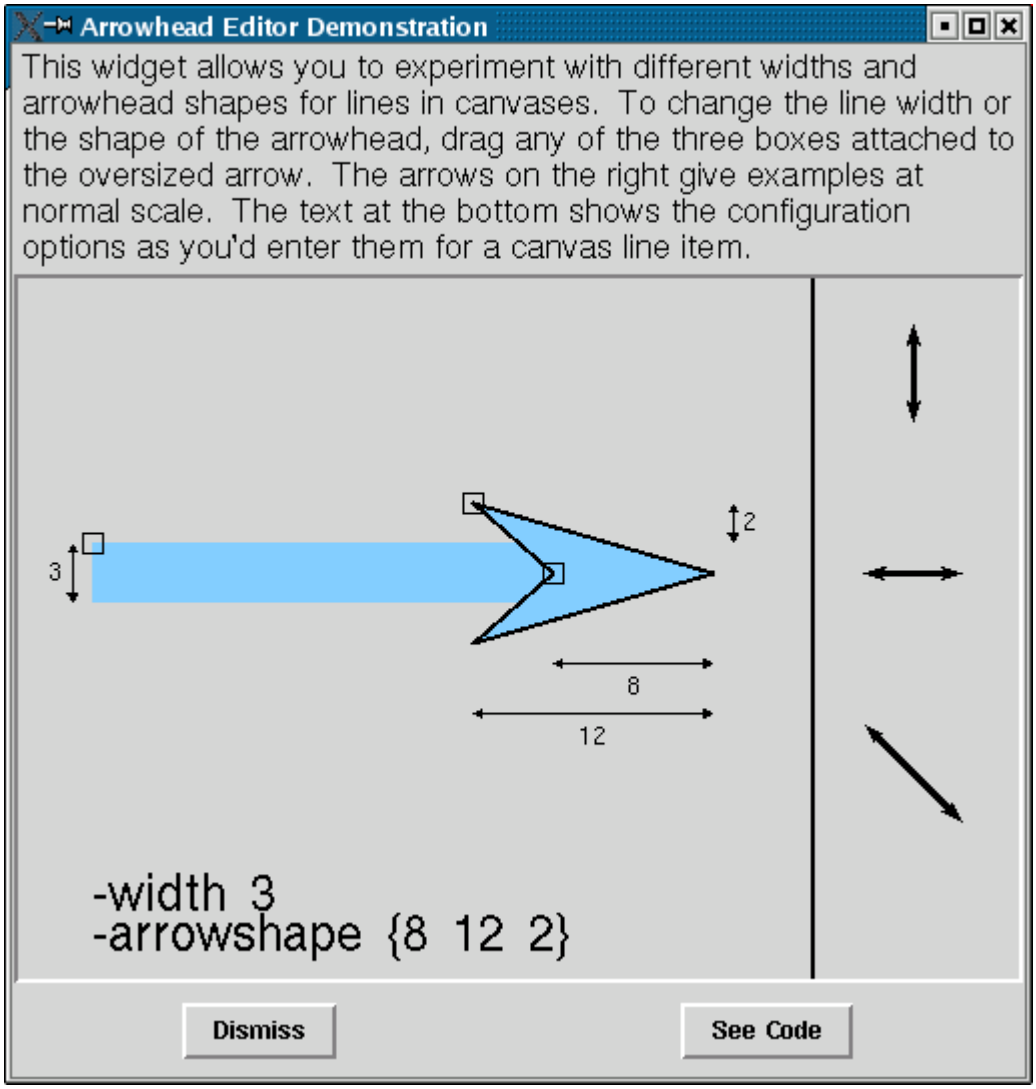
on the Tk text widget, and since the principles and implementation are virtually identical for the canvas widget I won't repeat the information here. A simple example of tag and bind applied to a canvas item should suffice:

```
.canvas create line 0 0 100 100 -tag diagonal_line
.canvas bind diagonal_line <Double-Button-1> {
      puts "Leave that alone!"
}
```

This creates a line item on the canvas from point 0,0 to point 100,100 and tags it with the name diagonal_line. If the user double clicks mouse button one on the line (or any other item with the tag diagonal_line) a message gets printed. Most uses of the canvas widget use this tag and bind approach extensively

Developing Applications with the Tk Canvas Widget

Given a problem which requires a graphical solution, the Tk canvas widget is often the first tool experienced script writers reach for. Complex problems can often be solved using a canvas widget and a few dozen lines of script. Figure 2 shows a screenshot of a script which uses a canvas to present an interactive editor for the shapes of the arrowheads which can terminate line items. This program is part of the demonstration suite which comes with Tcl/Tk, and takes about 200 lines of code.

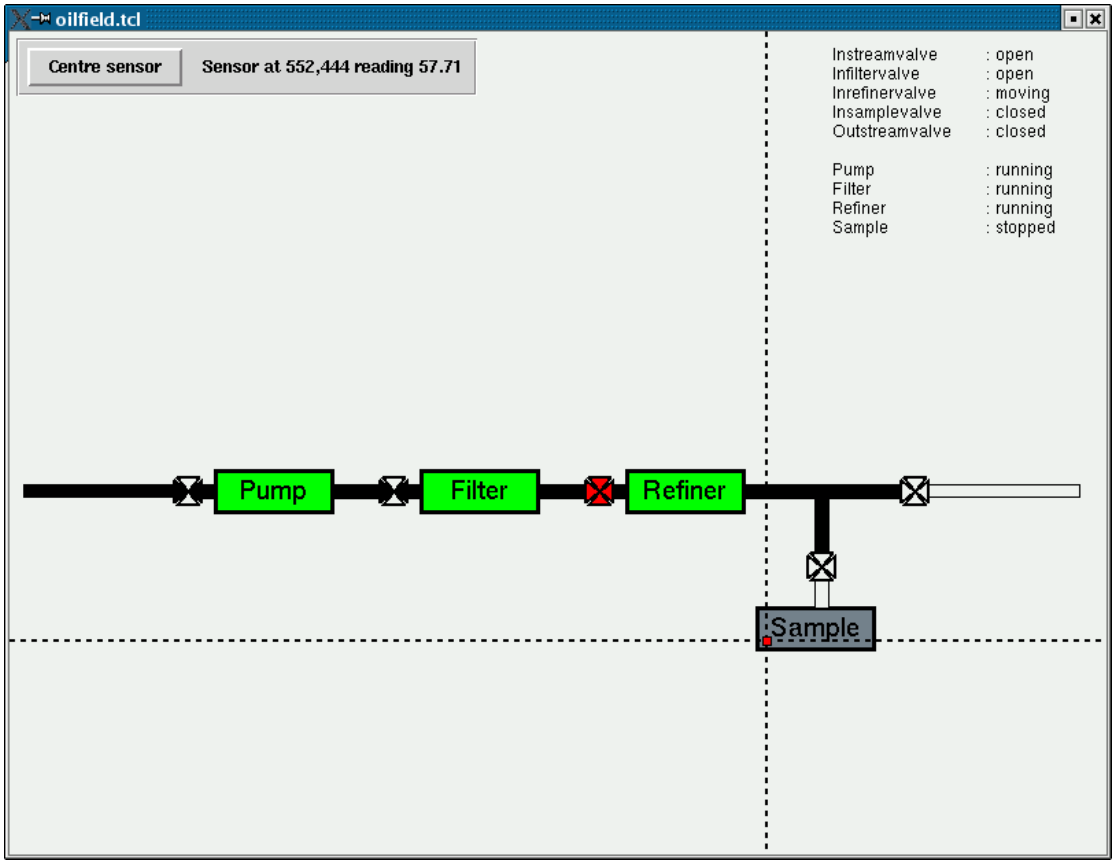


Useful as the canvas widget is for small applications and tools, it also has the flexibility and scalability to work well as a solution to enterprise level problems. Let's consider, using an example, how a canvas widget can be used to solve a requirement I once came across while working in the oil industry.

An Example script

Consider an oil refining installation, where oil is pumped through a series of machines under the control of a number of valves. As a software developer you might have written the code which drives the machines and their controlling valves, and now you'd like to write a graphical front end where the operator can click on a valve or machine, and have it open or close, start or stop.

I have written a script to demonstrate how the Tk canvas widget can be used to implement a solution to just such a scenario. See the resources section for the URL where the script can be downloaded. Figure 3 shows a screenshot.



In my simple, fictitious oil installation there are 4 machines: a pump, a filter, a refiner and a machine for taking oil samples. These are all controlled by a set of valves, any one of which can be open, closed or in the process of moving. There's also a temperature sensor which can be positioned over any point of the installation.

The demonstration script enables the graphical representations of the machines and valves to respond to mouse clicks, and the temperature sensor to be moved around. In a real world scenario, integrating the machine and valve controlling code with a front end like this would be a fairly straightforward task.

Let's have a closer look at how the canvas widget is used to represent and control this hardware.

The machines

The machines are drawn on the canvas as a simple rectangle with the name as a text string inside it. The rectangle is filled with a colour to indicate the machine's status.

```
.canvas create rect [list $x $y $w $h] \
      -tag [list machine $name] -width 3 -fill green
.canvas create text [expr $x + $w/2] [expr $y + $h/2] -text $text \
      -tag [list machine $name] -anchor c -font $font]
```

The text is placed in the calculated centre of the rectangle. Note how both items are given two tags -- "machine" and the actual machine name itself. The machine tag puts the items in a group which includes all machines. This is ensures a mouse click on any item which is part of any machine is caught and handled by the correct piece of code. This is the binding:

```
.canvas bind machine <Button-1> { operateMachine %W [%W canvasx %x] \
                                                    [%W canvasy %y] }
```

In a real situation, the operateMachine procedure would call the code which drives the machine hardware. In my example I just change the status of the machine in the script, which in turn changes the colour of the machine in the display.

The valves

The valves are represented on screen by an icon made of 4 triangles. The colour of these triangles shows whether the valve is open, closed or moving. I draw a valve as 4 filled polygon items, each triangle shaped. These 4 polygons are given a shared tag so I can control a valve icon with one command when necessary. Each polygon is also given a unique tag so I can control the colour of each individual triangle.

Each valve icon is wrapped in an invisible box which is sized and placed so it just encloses the 4 triangle items. The purpose of this item is to catch mouse clicks. Without it, the user might try to click on the valve icon, but might actually hit one of the gaps between the triangles, in which case the mouse click would be ignored. Using an invisible wrapper item over the top of the displayed items is a common technique for ensuring all mouse clicks are caught.

The canvas widget has a useful subcommand which returns the coordinates of a box which just encloses the specified item or tagged items. For example, to get the bounding box which encloses the 4 triangles which make up the instream valve:

```
set boxCoords [.canvas bbox instreamValve]
```

Given the information in boxCoords, an unfilled polygon item with line width of zero pixels can be placed over the valve icon. It's this polygon item which the valve handling code is bound to.

Temperature Sensor

The temperature sensor is drawn as a small red rectangle item at the intersection of two dashed lines:

```
.canvas create line $x $top $x $bottom -width 2 \
      -tag [list sensor_v sensor_line sensor] -dash { 2 2 }
.canvas create line $left $y $right $y -width 2 \
      -tag [list sensor_h sensor_line sensor] -dash { 2 2 }
.canvas create rect [expr $x-3] [expr $y-3] [expr $x+3] [expr $y+3] \
      -tag [list sensor_box sensor] -width 1 -fill red
```

The dash pattern a line item can be drawn with is configurable. In this case I've used a pattern of 2 pixels on and 2 pixels off. Regularly updating a dash pattern is a simple way to animate the marching ants effect found on many selection mechanisms.

The canvas contains a binding for dragging (i.e. mouse motion with button 1 pressed) these items, so the sensor can be moved by dragging it directly, or by dragging the lines which control it:

```
.canvas bind sensor <B1-Motion> \
      { operateSensor %W move [%W canvasx %x] [%W canvasy %y] }
```

This binding applies to all items which make up the sensor graphic – both lines and the centre rectangle. In order to find out which item is being dragged, the operateSensor code asks the canvas which item is currently under the mouse pointer using the special "current" tag:

```
set draggedItem [.canvas find withtag current]
```

In a real application, the operateSensor code would call the hardware which repositions the sensor. The sensor would also be polled on occasion, whereas in my demonstration script a random number is used for temperature display purposes.

Tcl Traces

My demonstration script uses Tcl's very useful ability to attach a procedure to variable accesses. This "tracing" is often used when a canvas widget is employed to maintain a graphical representation of a data set in a program. Tracing can be used to ensure that whenever data is changed, the routine to update the canvas is called automatically. A simple example from my demonstration script is the way I keep the sensor status text up to date. A trace is set in place:

```
trace add variable sensorState write setSensorText
```

which ensures that whenever the variable sensorState is written to, the procedure setSensorText is called automatically. This procedure updates the text displayed on the canvas widget. Other languages have features which can be used in similar ways. Users of Perl/Tk might want to look at Perl's tie mechanism, for example.

Other Interesting Canvas Features

The canvas widget has a few more features in its toolbox which can occasionally prove huge time savers.

Scaling

All items on a canvas which have been drawn using vectors (i.e. items with specified coordinates such as polygons and rectangles) can be dynamically scaled – i.e. resized. A single command can be issued to the canvas widget to request that it rescale any or all such items. For example, this command will rescale all the items on the canvas by 1.1:

```
.canvas scale all 0 0 1.1 1.1
```

In other words, this single line of code is a +10% zoom facility.

Postscript Output

The canvas widget also has a built in ability to generate a postscript representation of its current display. There are many options available to control the postscript output. The simplest way of getting a postscript dump of a canvas display is to call the postscript generation procedure with its default parameters from a specified keypress:

```
bind .canvas <KeyPress-p> ".canvas postscript -file /tmp/canvas.ps"
```

Conclusion

The Tk canvas widget has largely succeeded in hitting the perfect balance point between functionality and usability. It provides pretty much everything similar widgets from other toolkits do, plus a whole lot more in most cases. Yet, partnered with any one of a number of scripting languages, it is simple to use and incredibly efficient in terms of developer effort and lines of code needed to drive it.

Given its ability to provide solutions to a wide range of problems, the Tk canvas widget is a tool that all application software developers should be familiar with.

References

Tcl/Tk headquarters:

http://www.tcl.tk

The Tcl/Tk canvas widget man page:

http://www.tcl.tk/man/tcl8.4/TkCmd/canvas.htm

The Tkinter canvas widget reference page:

http://www.pythonware.com/library/tkinter/introduction/canvas.htm

Demonstration scripts used in this article:

ftp://ftp.scc.com/somewhere1.tar.gz

The Wiki canvas widget page, with lots of uses and examples:

http://mini.net/tcl/canvas