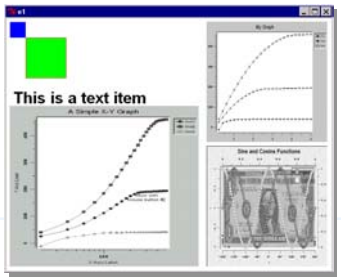
Building Better Applications with

George A. Howlett

Silicon Metrics Corporation

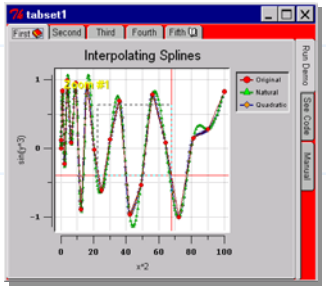
Austin, Texas

What is BLT?



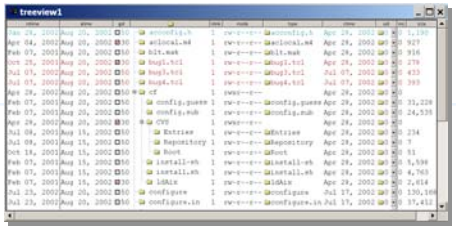
Set of widgets and new commands.

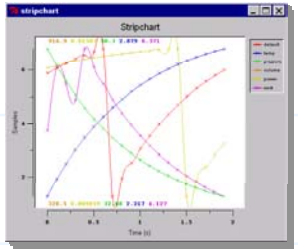
- Extends the Tcl interpreter, no patching required.

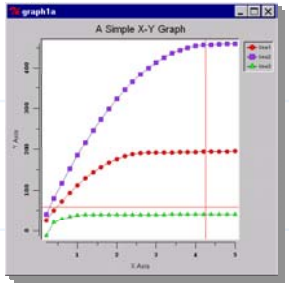


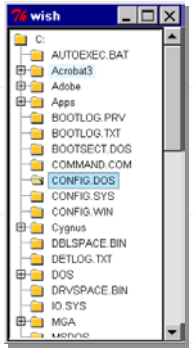
Features:

- Graph, stripchart, barchart widgets.
- Table geometry manager
- Treeview widget.
- Tabbed notebook widget.
- Drag-and-drop facility.
- Container widget.
- Busy command.
- Bgexec command.
- **…things I need for my Tcl/Tk applications.**



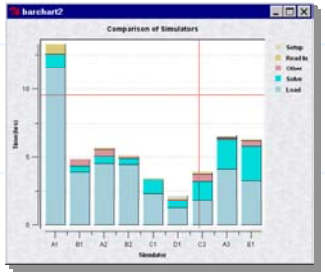






Platforms:

- Unix
- Windows 95/98/NT
- Macintosh OSX soon.



Building better applications with BLT

How to plot data with the graph widget.

Zooming and scrolling.

Printing.

Annotating graphs.

Build your own zooming graph.

Customizing the graph:

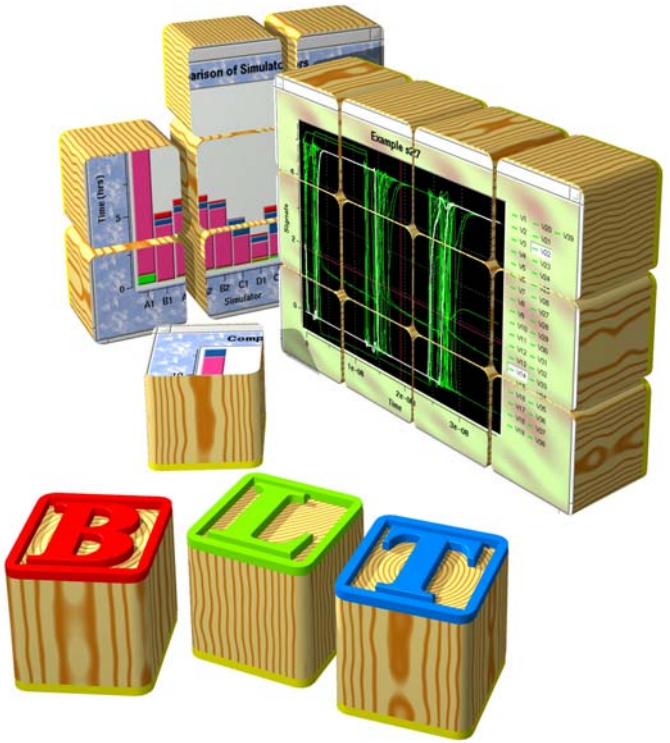
- Axes, legend, grid, crosshairs.

Interactive graphs.

Data handling.

Advanced features.

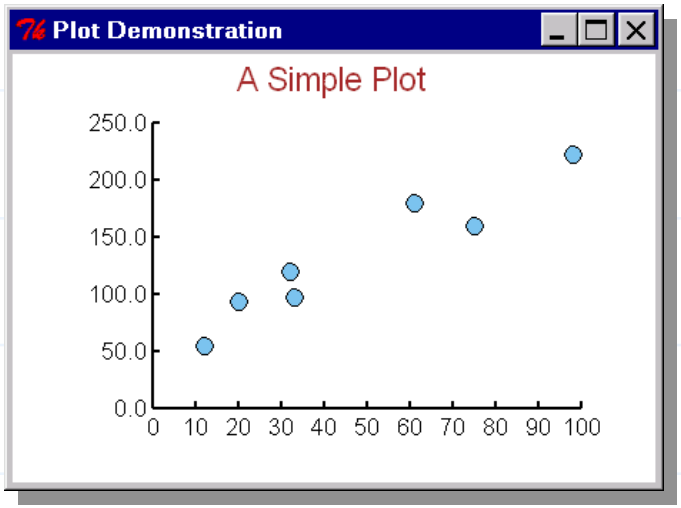
Managing graphs with tabnotebooks.

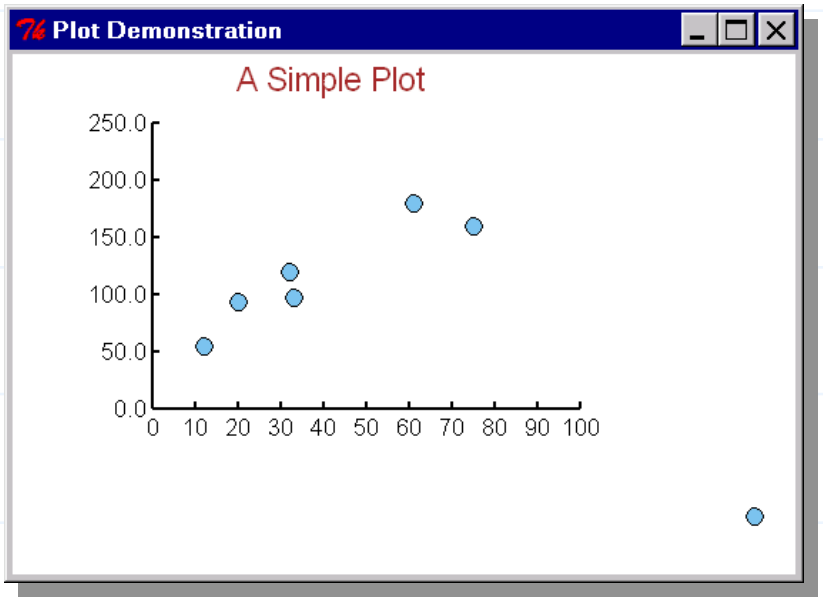


Using the canvas widget

Graph drawn on the canvas using Tcl code.

Example in Tk widget demo.



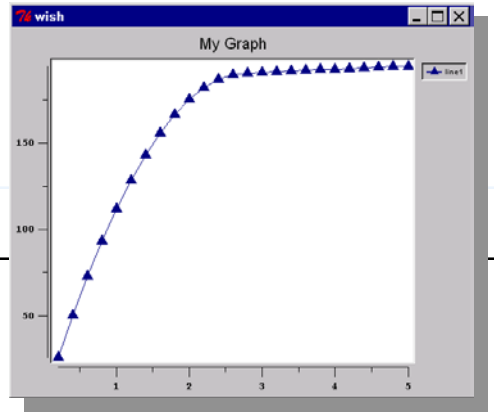


Problems:

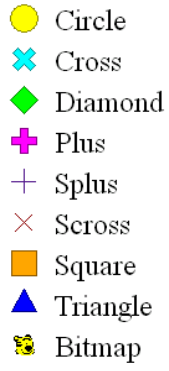
- Lots of Tcl code, lots of details to handle.
- Slow, scales badly with large data sets.
- Zooming broken.

No code for resizing.

Graph widget




```
package require BLT

blt::graph .g -title "My Graph"
pack .g
.g element create line1 -symbol triangle \
   -xdata {0.2 0.4 0.6 0.8 1.0 1.2 1.4 1.6 1.8 … } \
   -ydata {2.61825e+01 5.04696e+01 7.28517e+01 … }
```



Symbol types

Create graph widget and add **data elements** with **`element`** operation.

- X-Y coordinates are lists of numbers.
- Configuration options control element's appearance.

`-symbol`	Symbol displayed at each data point.
`-xdata`	Real numbers representing X-coordinates.
`-ydata`	Real numbers representing Y-coordinates.

Elements

Elements are a graph *component*.

- Represents a set of data.
- Symbols are the data points.
- Usually drawn as a single trace.

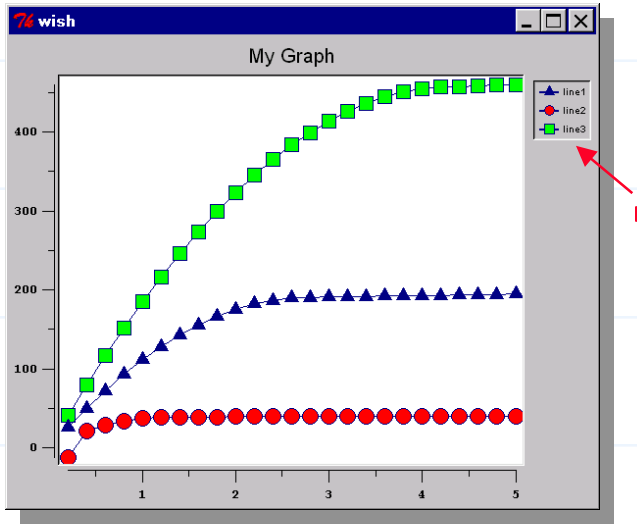
Each element has entry in legend.

Z-ordering

- First elements created sit on top of later.

Axes auto-scale

- Data determines range of axes.



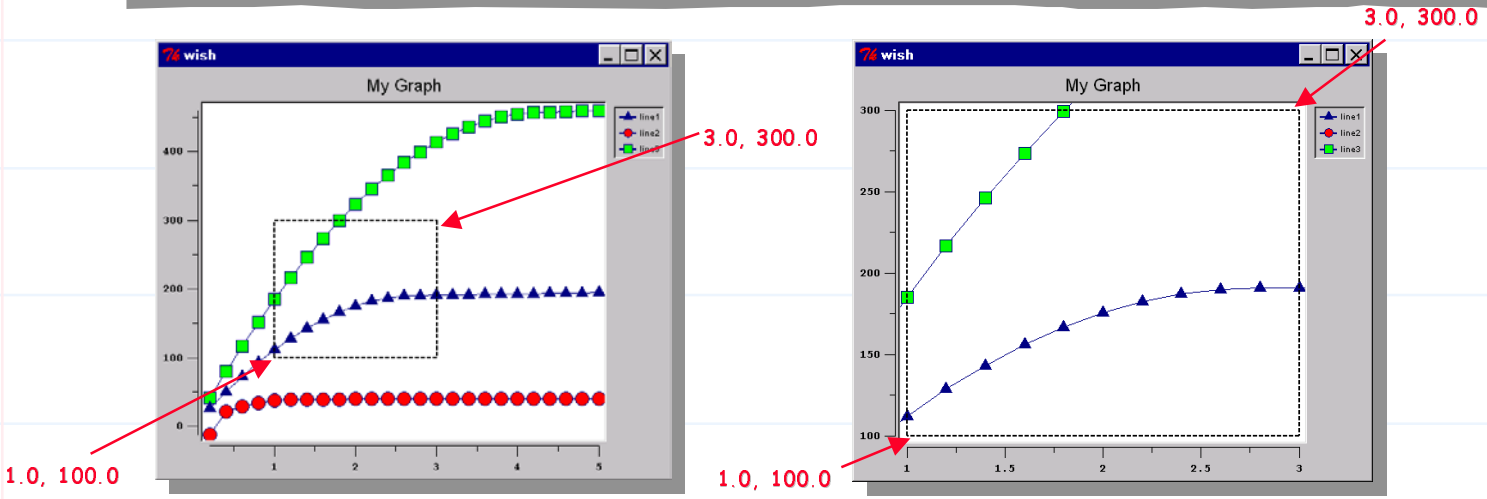
Legend order reflects depth.

```
.g element create line2 -symbol circle -fill red \
   -xdata {0.2 0.4 0.6 0.8 1.0 1.2 1.4 1.6 1.8 … } \
   -ydata {-1.14471e+01 2.09373e+01 2.84608e+01 … }

.g element create line3 -symbol square -fill green \
   -xdata {0.2 0.4 0.6 0.8 1.0 1.2 1.4 1.6 1.8 … } \
   -ydata {4.07008e+01 7.95658e+01 1.16585e+02 … }
```

Zooming

Graph's **axis** component controls range of points displayed.

```
.g axis configure x -min 1.0 -max 3.0
.g axis configure y -max 100.0 -max 300.0
```



Graph is automatically redrawn displaying the selected range.

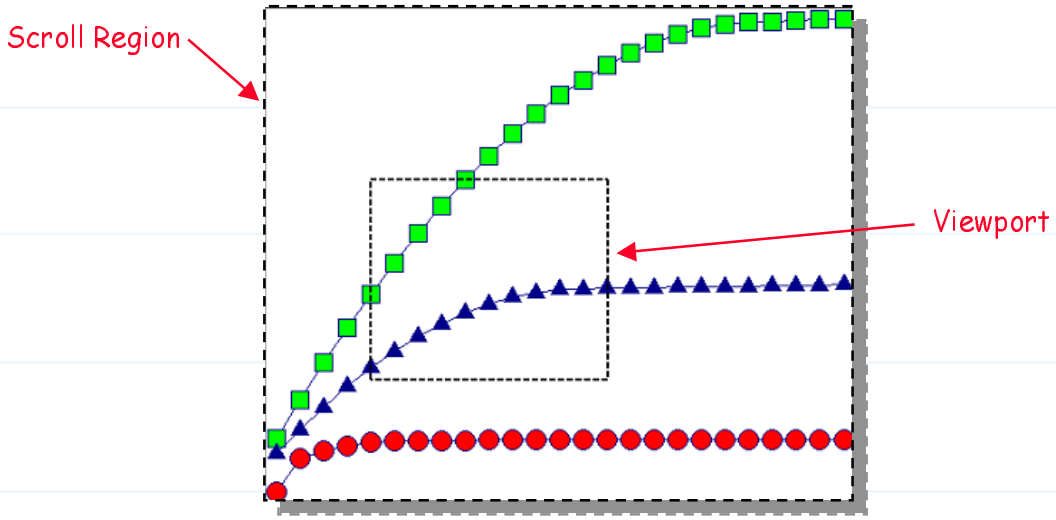
- Set **-min** and **-max** to the empty string to restore auto-scaling.

```
.g axis configure x -min {} -max {}
.g axis configure y -max {} -max {}
```

Scrolling

To scroll, add or subtract *same* amount from both min and max.

```
.g axis configure x -min [expr 1.0 + $delta] \
    -max [expr 3.0 + $delta]
```



Viewport defined by the current axis **`-min`** and **`-max`** values.

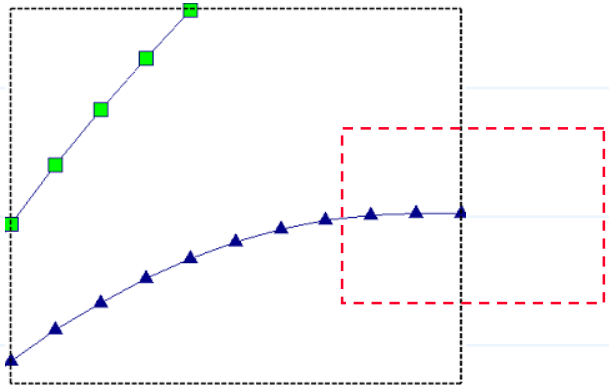
Scroll region defined by the range of data.

Scrolling (cont'd)

Can attach a scrollbar to any axis.

```
scrollbar .hs -command { .g axis view x } -orient horizontal
scrollbar .vs -command { .g axis view y } -orient vertical
.g axis configure x -scrollcommand { .hs set }
.g axis configure y -scrollcommand { .vs set }
```

Like attaching scrollbar to any Tk widget.



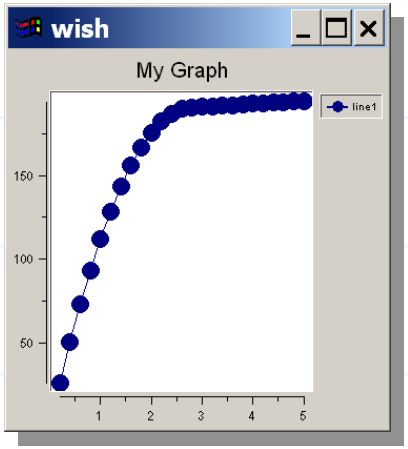
What if viewport and default scroll region overlap?

Scroll region overridden by **`-scrollmin`** and **`-scrollmax`** options.

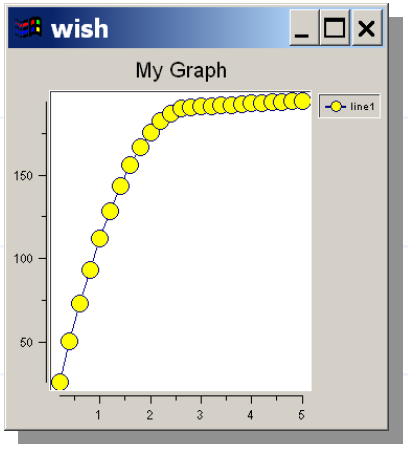
Customizing elements

Lots and lots of element options.

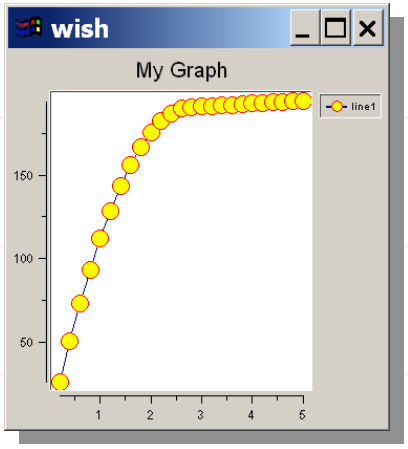
Most don't need to be set.



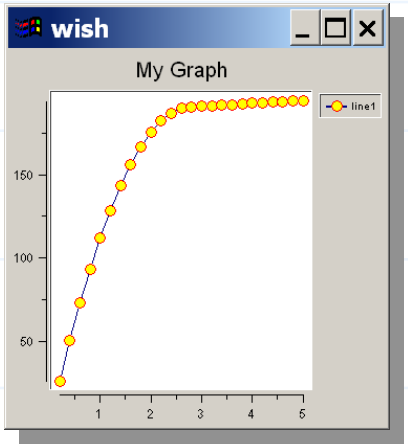
`-symbol circle`



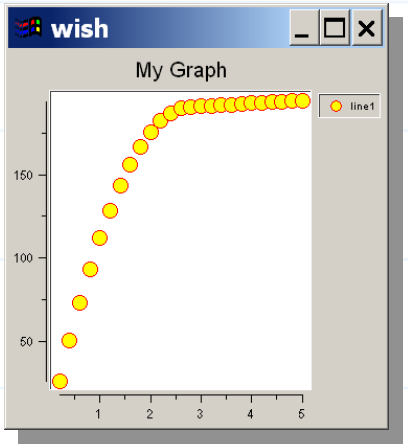
`-fill yellow`



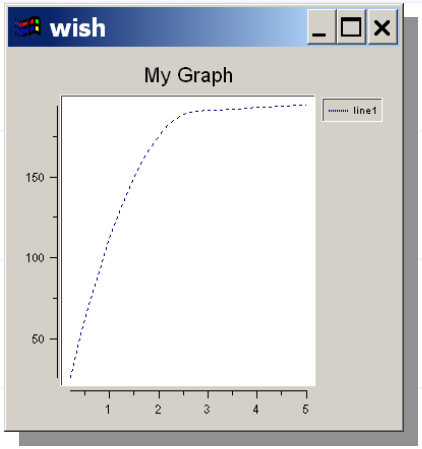
`-outline red`



`-pixels 8`

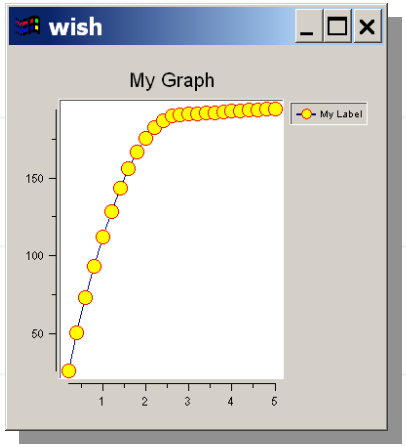


`-linewidth 0`

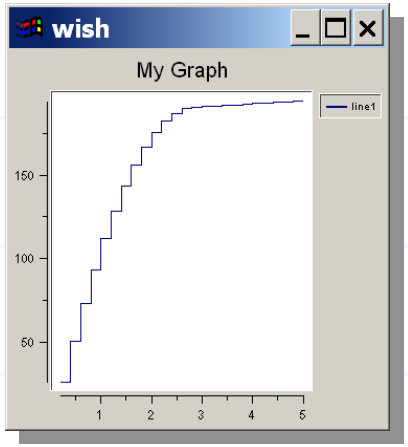


`-dashes dot`

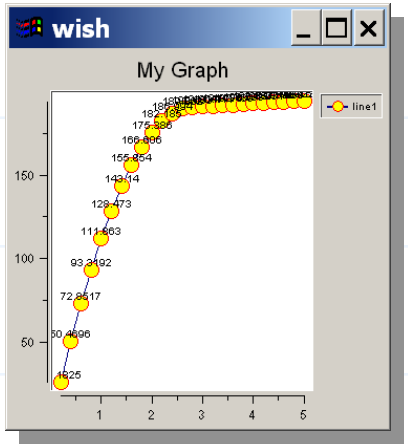
Customizing elements (cont'd)



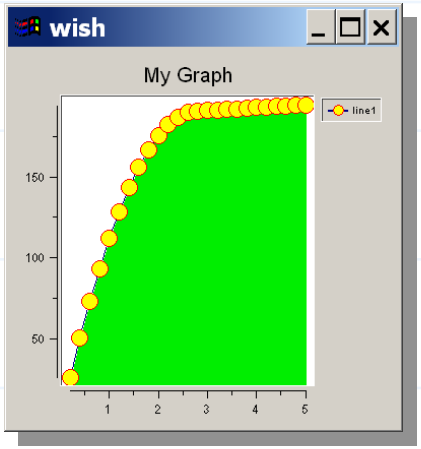
`-label "My Label"`



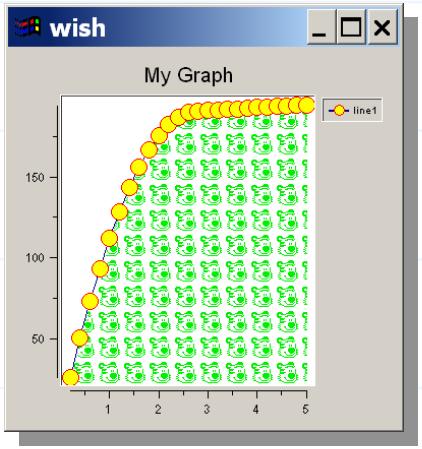
`-smooth step`



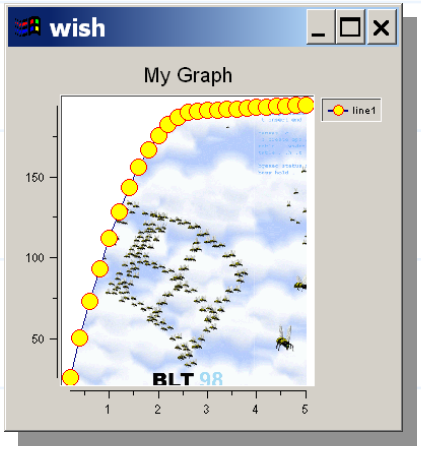
`-showvalues y`



`-areapattern solid`

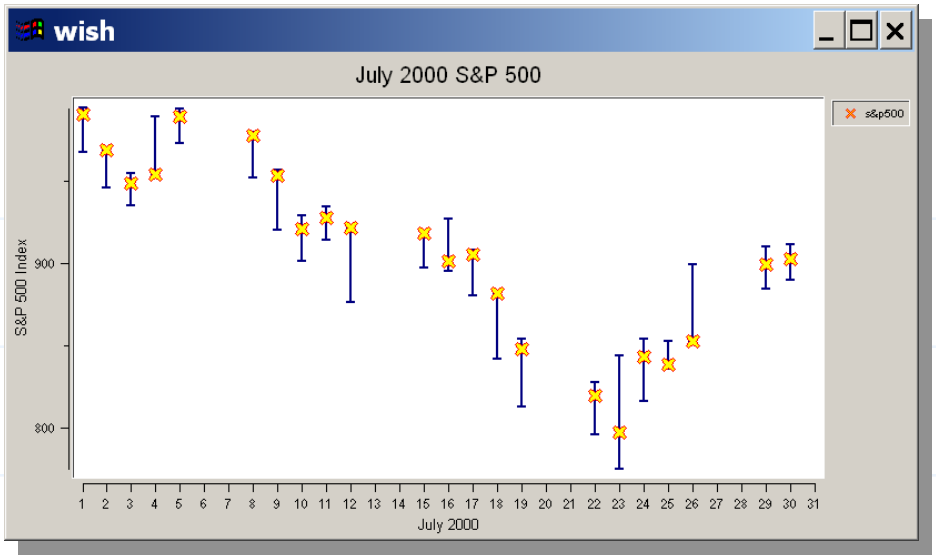


`-areapattern @hobbes.xbm`



`-areatile $image`

Error bars



Both X and Y error bars can be displayed for elements.

- Element configurations options take list of values:

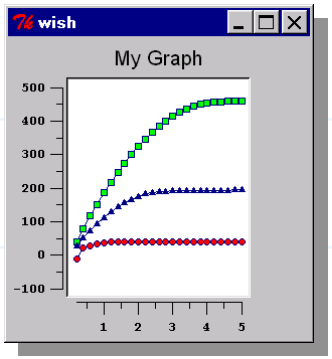
-xhigh, **-xlow**, **-yhigh**, **-ylow**	High/low sample for each data point.
-xerror, **-yerror**	Defines delta for symmetric error range.

```
.g element create s&p500 -symbol cross –fill yellow \
   -outline red –linewidth 0 –errorbarwidth 2
   -x { 1 2 3 4 5 8 9 10 11 12 15 16 17 18 19 22 23 … } \
   -y { 989.82 968.65 948.09 953.99 989.03 976.98 … } \
   -yhigh { 994.46 968.65 954.30 989.07 993.56 979.63 … } \
   -ylow { 967.43 945.54 934.87 953.99 972.91 951.71 … }
```

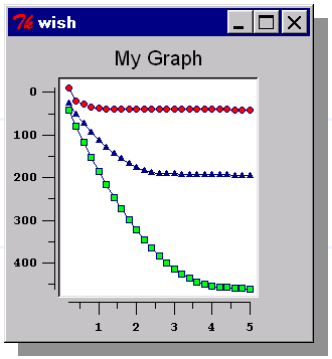
Customizing axes

Assorted options set appearance using **`axis configure`** operation.

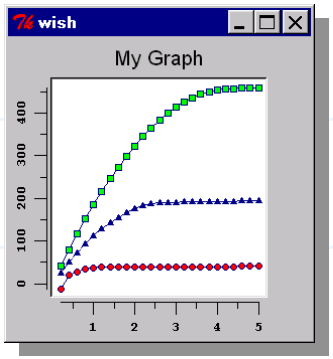
Changes made on Y-axis only.



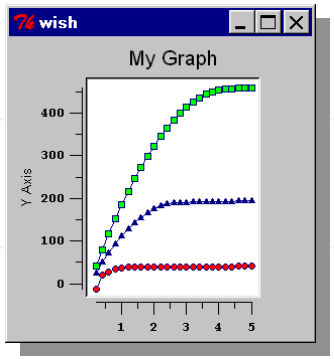
`-loose yes`



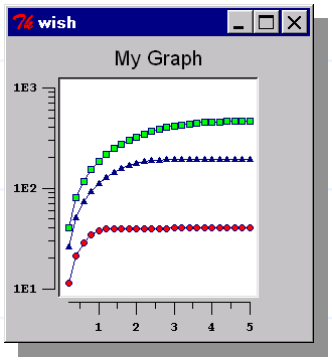
`-descending yes`



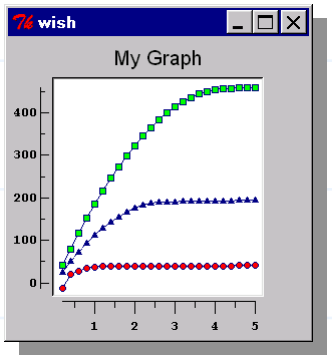
`-rotate 90`



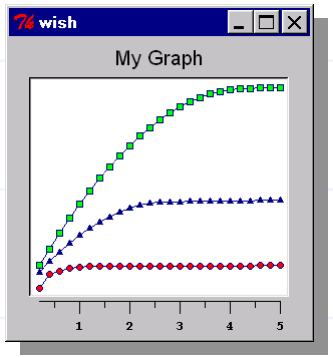
`-title "Y Axis"`



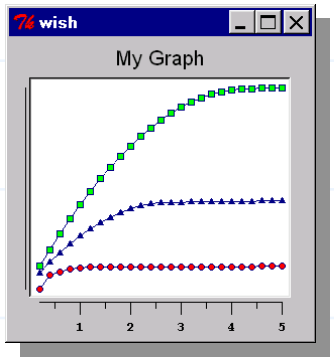
`-logscale yes`



`-ticklength -5`



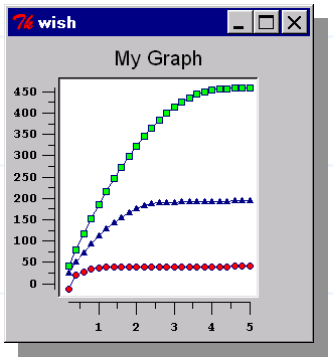
`-hide yes`



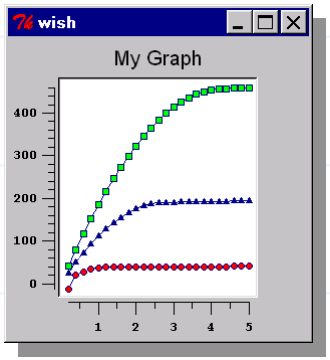
`-showticks no`

Customizing axes (cont'd)

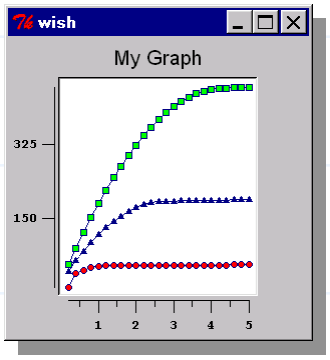
Tick positions and labels also controlled by axis configuration options.

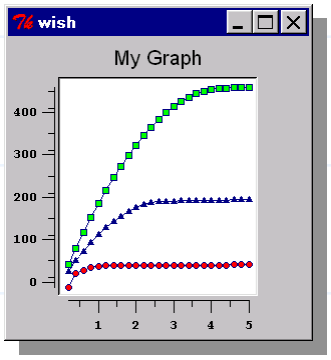


`-stepsize 50.0`



`-subdivisions 5`

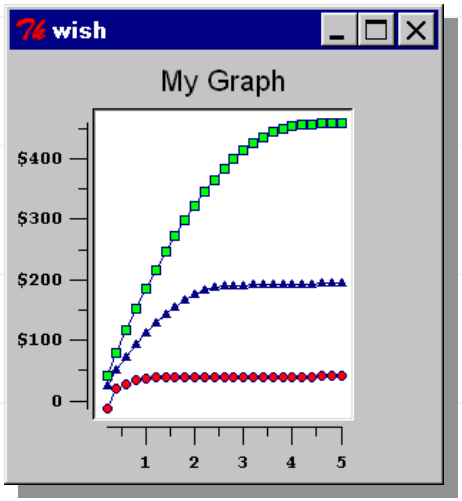




`-majorticks {150 325} -minorticks {.1 .5}`

Labels customized by specifying callback proc.

```
proc FormatTick { widget x } {
    if { $x != 0.0 } { return \$$x }
    return $x
}
.g axis configure y \
    -command FormatTick
```



Don't modify graph within callback proc.

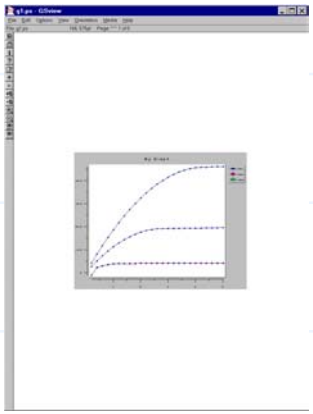
Printing graphs

Graph's **`postscript`** component generates encapsulated PostScript.

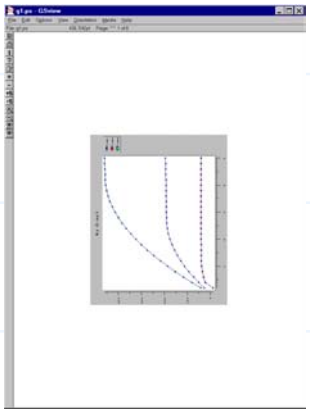
```
.g postscript configure -landscape yes -maxpect yes
.g postscript output myFile.ps
```

- File name optional. PostScript returned as the result of output operation.

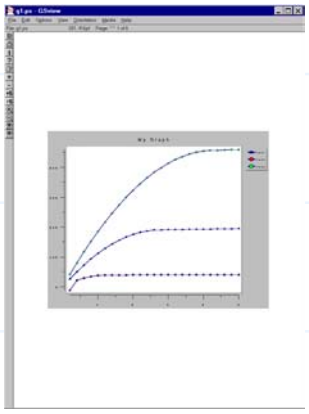
Other options control graph, border, and paper size.



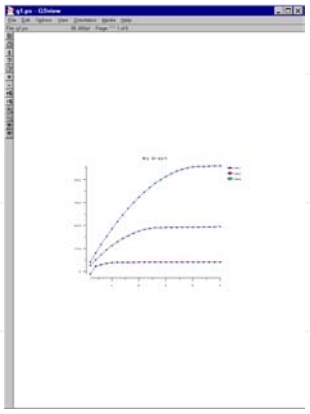
`-center yes`



`-landscape yes`



`-maxpect yes`



`-decorations no`

Printing under Windows

Scenario #1: You have a PostScript printer

BLT **printer** command sends raw EPS to a PostScript printer.

```
set output [.g postscript output]

set pid [blt::printer open {QMS ColorScript 100 v49.4}]
blt::printer write $pid $output
blt::printer close $pid
```

Query printer settings with **getattr** operation.

- Written to array variable.

Adjust printer settings with **setattr** operation.

```
blt::printer getattr $pid myArray
set myArray(PaperSize) Letter
set myArray(Orientation) Landscape
blt::printer setattr $pid myArray
blt::printer write $pid $output
blt::printer close $pid
```

Printing to non-PS printers

Scenario #2: You have a non-PostScript printer.

Graph has two Windows-specific print operations.

print1

- Writes bitmap image to printer.
- Usually works regardless of printer capabilities.
- Poorer quality. Jagged lines and fonts.

```
set pid [blt::printer open {QMS ColorScript 100 v49.4}]
.g print1 $pid
blt::printer close $pid
```

print2

- Draws directly to print device.
- Doesn't print correctly on PostScript printers (print raw PS instead).

```
set pid [blt::printer open {QMS ColorScript 100 v49.4}]
.g print2 $pid
blt::printer close $pid
```

Embedding graphs

Scenario #3: You want to include the graph in a Word document.

Graph's **snap** operation takes snapshot of the graph.

- By default, writes to Tk image of graph.
- Better than screen dump: graph doesn't have to be mapped.

```
set image [image create photo]
.g snap $image
$image write file.gif -format GIF
```

Better yet: Write Windows-specific metafile with **-format** switch.

- Two types of meta files: Aldus placeable metafile (WMF) and Enhanced Windows metafile (EMF)
- Can write directly to clipboard.
- Better than GIFs: metafiles are independent of screen resolution.

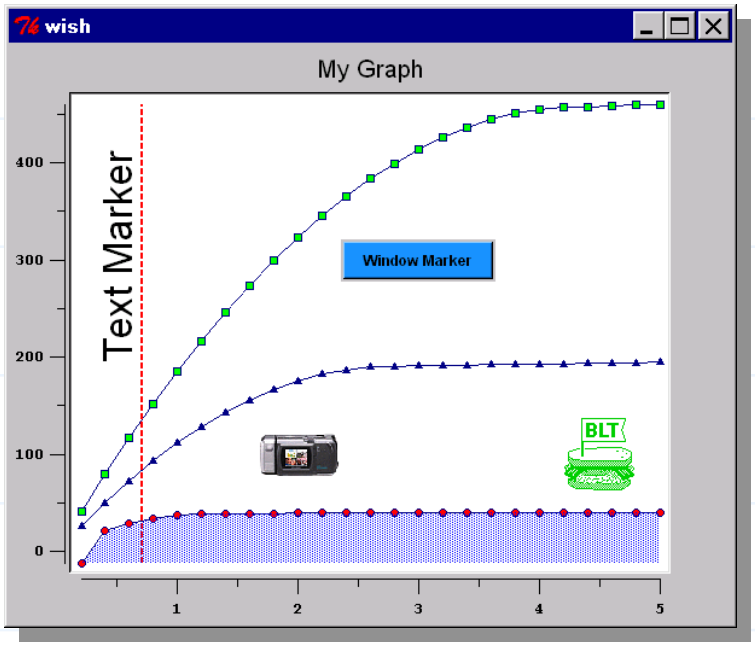
```
.g snap -format EMF CLIPBOARD
```

Annotating graphs

Markers are used to highlight or annotate areas.

6 types of markers:

- **text**
- **line**
- **polygon**
- **bitmap**
- **image**
- **window**



Marker positions in graph coordinates.

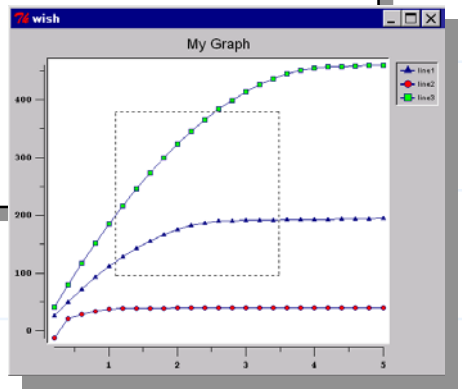
```
.g marker create text -text "Text Marker" -rotate 90 \
    -coords { 0.5 300 } -font { Helvetica 20 }
.g marker create line -coords { 0.7 -Inf 0.7 Inf } \
    -dashes dash -linewidth 2 -outline red
image create photo myImage -file images/qv100.t.gif
.g marker create image -image myImage -coords {2.0 100.0}
button .g.button -text "Window Marker" -bg dodgerblue
.g marker create window -window .g.button -coords {3 300}
```

Example: Zooming graph

Already know how to zoom in/out of a graph.

```
proc Zoom { graph x1 y1 x2 y2 } {
    if { $x1 > $x2 } {
        $graph axis configure x -min $x2 -max $x1
    } elseif { $x1 < $x2 } {
        $graph axis configure x -min $x1 -max $x2
    }
    if { $y1 > $y2 } {
        $graph axis configure y -min $y2 -max $y1
    } elseif { $y1 < $y2 } {
        $graph axis configure y -min $y1 -max $y2
    }
}
proc Unzoom { graph } {
    $graph axis configure x y -min {} -max {}
}
```

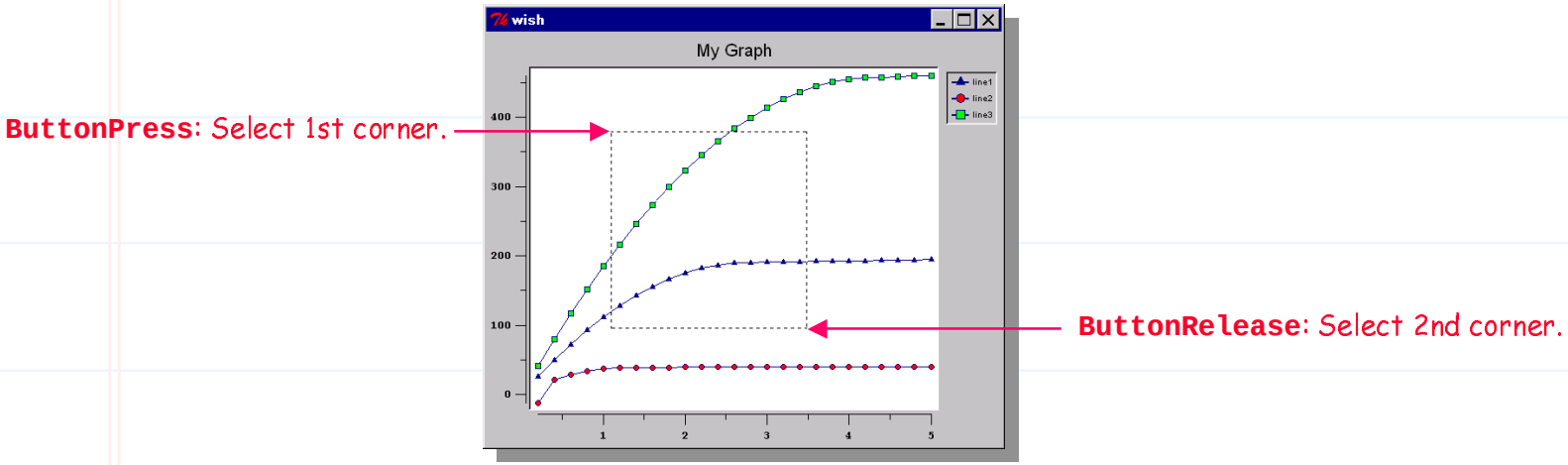
Can configure more than one axis at a time.



Zooming graph (cont'd)

Create user-selectable zoom region. Drawn with a line marker.

- ButtonPress-1 Selects first corner of zoom region.
- B1-Motion Draws outline. Position is opposite corner of region.
- ButtonRelease-1 Deletes outline, zooms to selected region.



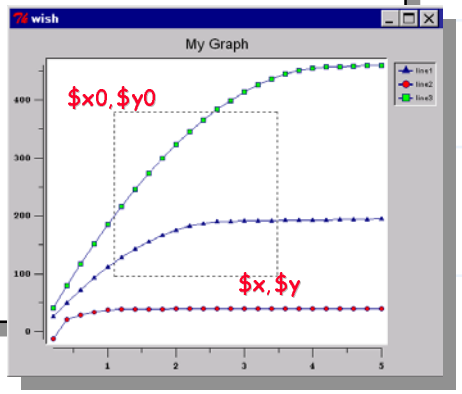

```
bind .g <ButtonPress-1>   { RegionStart %W %x %y }
bind .g <B1-Motion>       { RegionMotion %W %x %y }
bind .g <ButtonRelease-1> { RegionEnd %W %x %y }
bind .g <ButtonRelease-3> { Unzoom %W }
```

Zooming graph (cont'd)

```
proc RegionStart { graph x y } {
    global x0 y0
    $graph marker create line -coords { } -name myLine \
        -dashes dash -xor yes
    set x0 $x;  set y0 $y
}
proc RegionMotion { graph x y } {
    global x0 y0
    $graph marker coords myLine \
      "$x0 $y0 $x0 $y $x $y $x $y0 $x0 $y0"
}
proc RegionEnd { graph x y } {
    global x0 y0
    $graph marker delete myLine
    Zoom $graph $x0 $y0 $x $y
}
```

Markers without coordinates aren't drawn.

Name the marker, so we can refer to it.



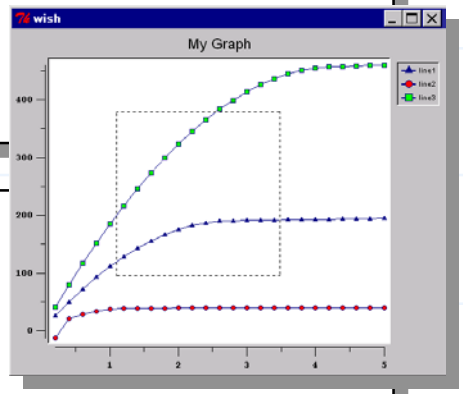
- First corner of region saved in global variables **`x0`** and **`y0`**.
- Line marker can be erased with redrawing graph with **`-xor`** option.
- Marker **`coords`** operation changes line coordinates.
- Delete marker when done.

Converting to/from graph coordinates

Why doesn't example work? Need to translate **screen** to **graph** coordinates.

- Mouse location is in **screen** coordinates (relative to the widget).
- Markers are positioned in **graph** coordinates.

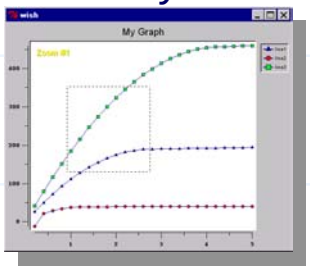
```
# Screen to graph coordinates
set graphX [.g axis invtranform x $screenX]

# Graph to screen coordinates
set screenX [.g axis transform x $graphX]
```

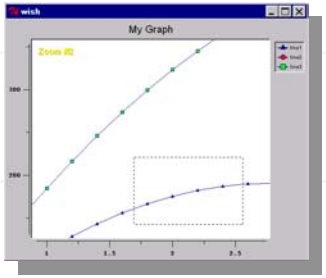


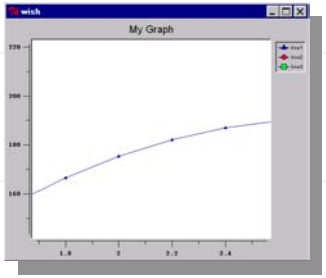

```
proc RegionStart { graph x y } {
    set x [$graph axis invtransform x $x]
    set y [$graph axis invtransform y $y]
    …
}
proc RegionMotion { graph x y } {
    set x [$graph axis invtransform x $x]
    set y [$graph axis invtransform y $y]
    …
}
```

Zooming graph (cont'd)

Can recursively zoom further and further in.







Add feature: Stack zoom levels so user can pop back to previous zoom.

```
set zoomStack {}
proc Zoom { graph x1 y1 x2 y2 } {
    PushZoom $graph
    blt::busy hold $graph
    update
    blt::busy release $graph
}
proc Unzoom { graph } {
    if { ![EmptyZoom] } { eval [PopZoom] }
    blt::busy hold $graph
    update
    blt::busy release $graph
}
```

Use Tcl list as zoom stack.

Busy command prevents accidental zoom/unzoom.

Zooming graph (cont'd)

Create zoom stack. Push/pop graph commands to restore axis ranges.

```
proc PushZoom { graph } {
    global zoomStack
    set x1 [$graph axis cget x -min]
    set x1 [$graph axis cget x -max]
    set y1 [$graph axis cget y -min]
    set y2 [$graph axis cget y -max]
    set cmd "$graph axis configure x -min $x1 -max $x2 ;
      $graph axis configure y -min $y1 -max $y2"
    lappend zoomStack $cmd
}
proc PopZoom {} {
    global zoomStack
    set cmd [lindex $zoomStack end]
    set zoomStack [lreplace $zoomStack end end]
    return $cmd
}
proc EmptyZoom {} {
    global zoomStack
    expr {[llength $zoomStack] == 0}
}
```

Get current axis ranges.

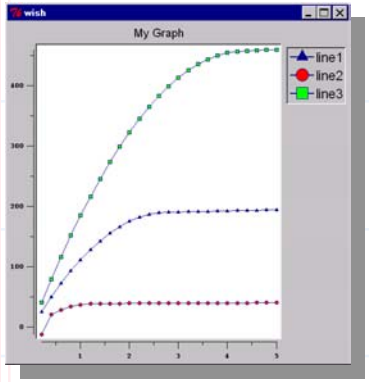
Stack commands to restore to current zoom level.

Pop last command off and remove it.

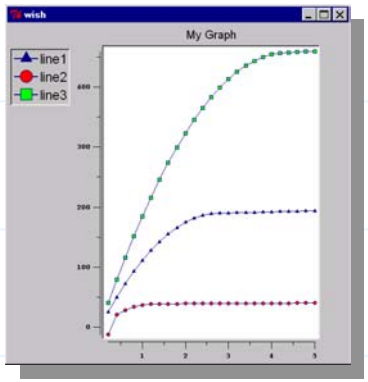
Legend component

Controls position/appearance of legend.

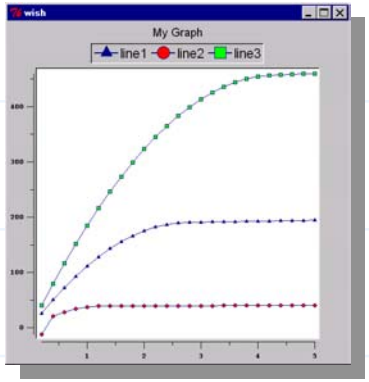
graph **legend configure** ?*option value...*?



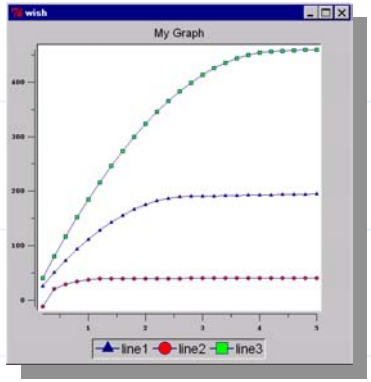
-position right



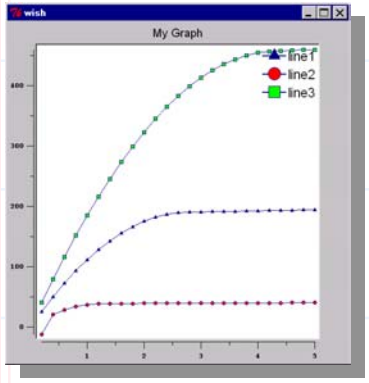
-position left



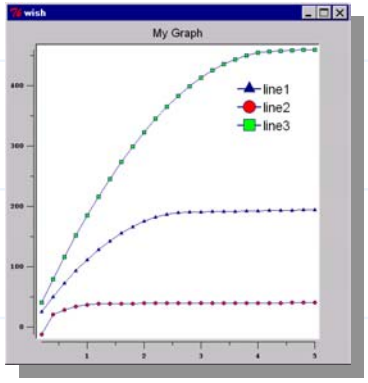
-position top



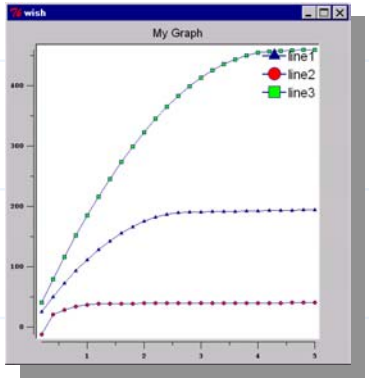
-position bottom



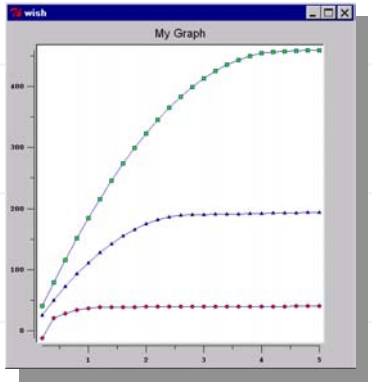
-position plotarea



-position @450,100



-raised yes

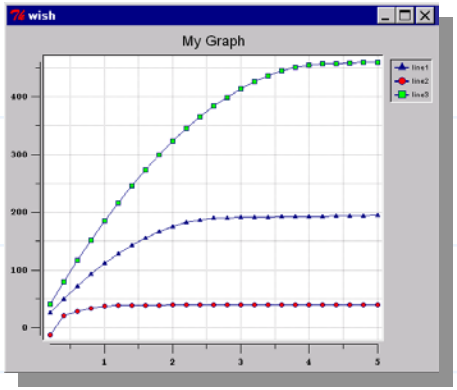


-hide yes

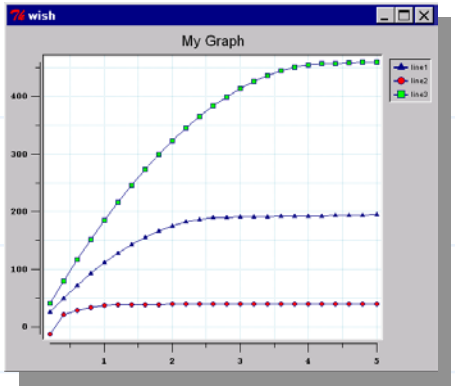
Grid component

Controls appearance of built-in grid.

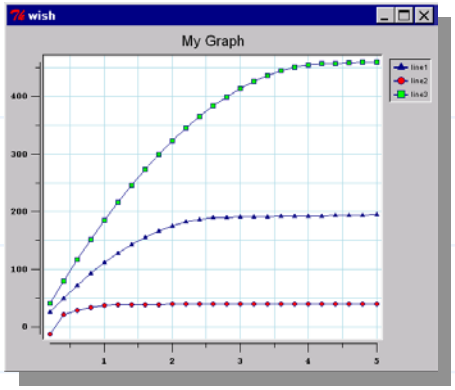
- Extensions of major/minor ticks of each axis running across the plotting area.



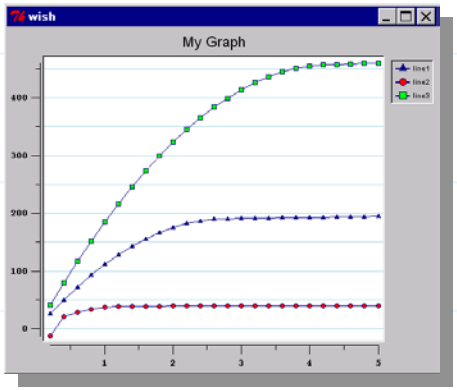
`-hide no`



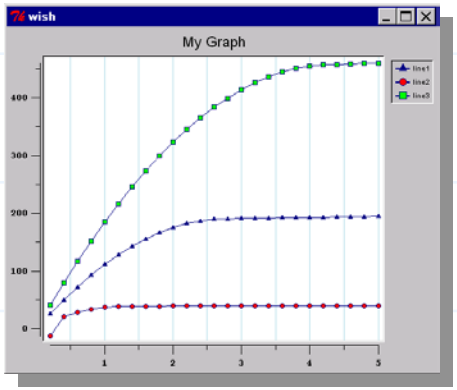
`-color lightblue`



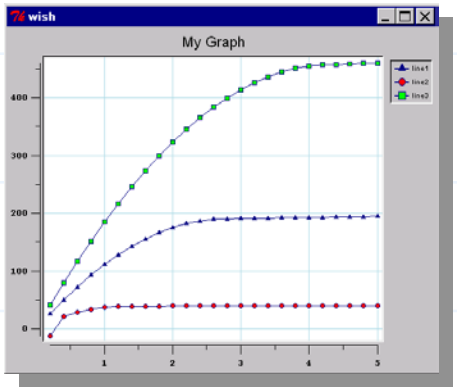
`-dashes 0`



`-mapx {}`



`-mapy {}`

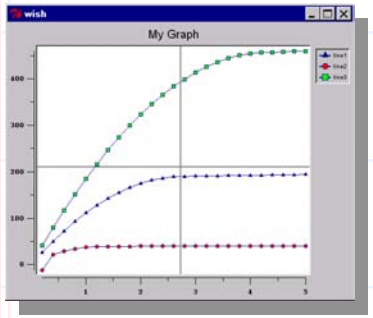


`-minor no`

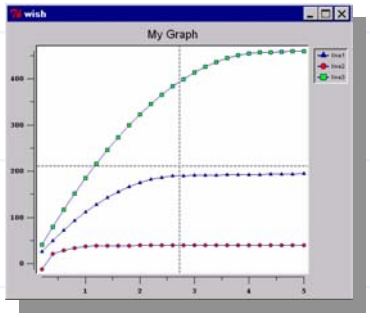
Crosshairs component

Controls position/appearance of crosshairs.

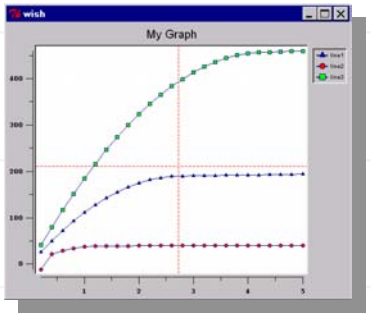
- Two intersecting lines (one vertical and one horizontal) running across plotting area.
- Used to finely position mouse in relation to coordinate axes.



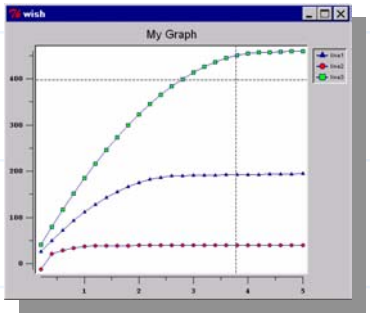
`-hide no`



`-dashes dash`



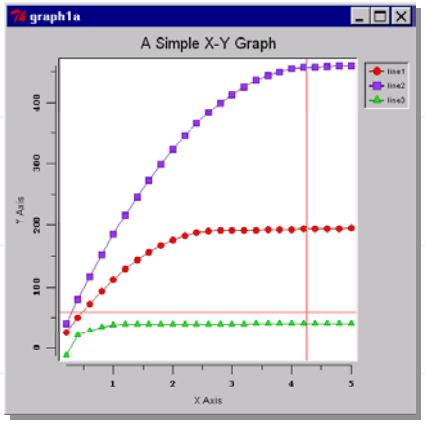
`-color red`

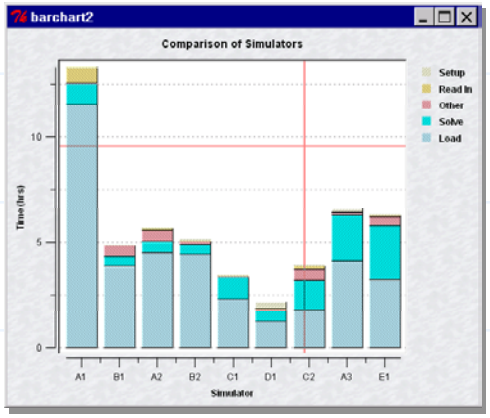


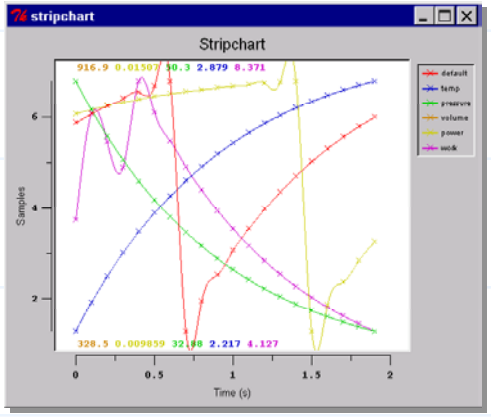
`-position @450,100`

```
.g crosshairs on
.g crosshairs configure -color red -dashes 2

bind .g <Motion> {
    .g crosshairs configure -position @%x,%y
}
```

Types of graphing widgets







BLT has 3 graphing widgets available:

- **`graph`** X-Y coordinate graph.
- **`barchart`** Displays bars at X-Y coordinates.
- **`stripchart`** Similar to X-Y graph, extra features.

Most features span across all three widgets.

Every feature we've seen so far works in all three widgets..

Interactive graphs

Zooming graph example of interactive graph.

All graph widgets/plotting packages draw graphs.

- Convert data points to screen pixels.
- Graphs *better* on paper. Higher resolution.

Two-way communication (back annotation) lets graph become powerful tool.

- Convert screen coordinates back to data points.
- Examples: identify data points, compute slopes, area under curve, etc.

Identifying data points

Elements have **`closest`** operation to identify points/traces.

graph **`element closest`** *x y varName* ?*options*? ?*elemName*…?

Returns "1" if a closest element is found, otherwise "0".

Writes information into a Tcl array variable.

- **`name`** Name of closest element.
- **`dist`** Distance from element.
- **`index`** Index of closest data point.
- **`x`** and **`y`** The X-Y graph coordinates of the closest point.

```
.g element closest 300 400 myInfo
.g element closest 200 400 myInfo -halo 1.0i
.g element closest 1 40 myInfo -interpolate yes
.g element closest 20 10 myInfo line2 line1

puts "$myInfo(name) is closest at $myInfo(index)"
```

Options:

`-halo`	Selects cut-off radius from screen coordinate.
`-interpolate`	Search for closest point on trace, not just data points.
`-along`	Search perpendicular to X or Y axis.

Binding to graph components

You can **bind** to elements, markers, axes, and legend entries.

```
.g element bind line1 <Enter> {
    puts "Touched element"
}
.g marker bind myLine <Enter> {
    puts "Touched marker"
}
.g legend bind line1 <ButtonPress-1> {
    puts "selected line1"
}
```

Each component has its own **bind** operation.

- Similar to binding to canvas items.
- Can bind to mouse and key events, create binding tags, etc.

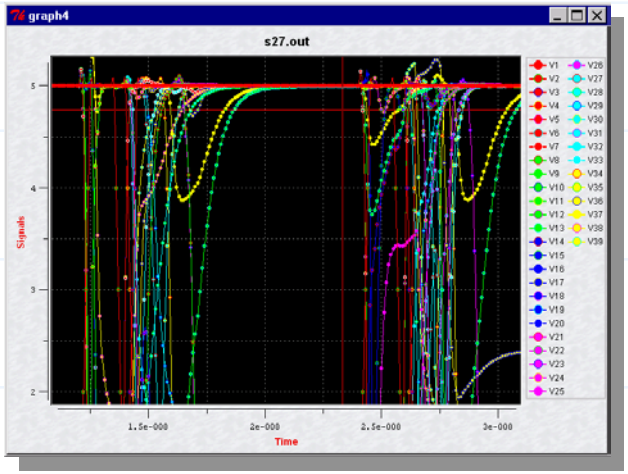
Find currently selected item using **get** operation.

```
set elem [.g element get current]
set marker [.g marker get current]
set elem [.g legend get current]
```

Example: active legend

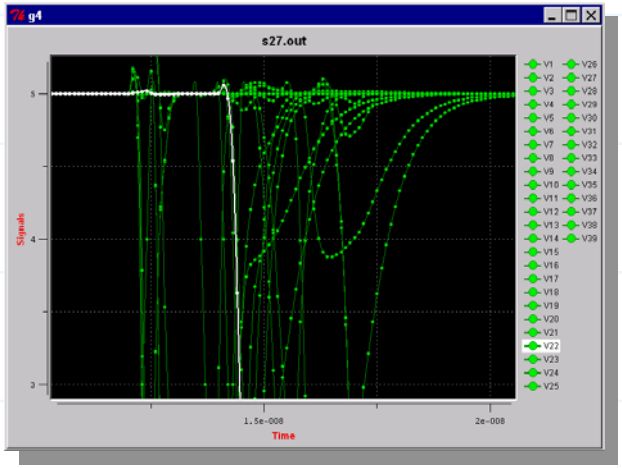
How do you display many elements?

- Typical to have lots of elements.
- Rotating colors/line styles doesn't help.
- Clutter hides behavior of data.

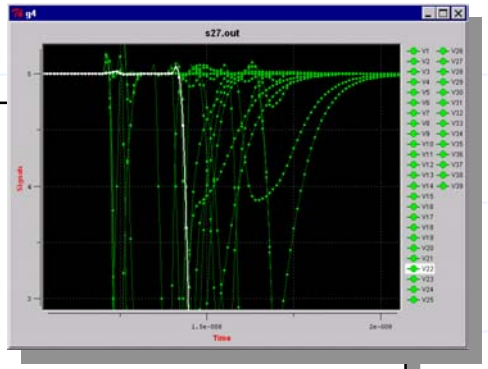


Let user interactively highlight elements.

- Draw all elements the same style.
- Moving mouse over element highlights legend entry.
- Clicking on entry highlights its element.



Active legend (cont'd)




```
.g element bind all <Shift-Enter> {
    Highlight %W [%W element get current]
}
.g element bind all <Shift-Leave> {
    Unhighlight %W [%W element get current]
}
.g legend bind all <ButtonPress-1> {
    Highlight %W [%W legend get current]
}
.g legend bind all <ButtonRelease-1> {
    Unhighlight %W [%W legend get current]
}
```

Binding tag **all** is automatically set for elements, markers, legend entries.

Can include/exclude tags with **-bindtags** configuration option.

```
.g element configure line1 -bindtags { myTag all }
.g marker configure myLine -bindtags { myTag all }
```

- Element and marker tags reside in different tables.
- Legend uses element's tags.

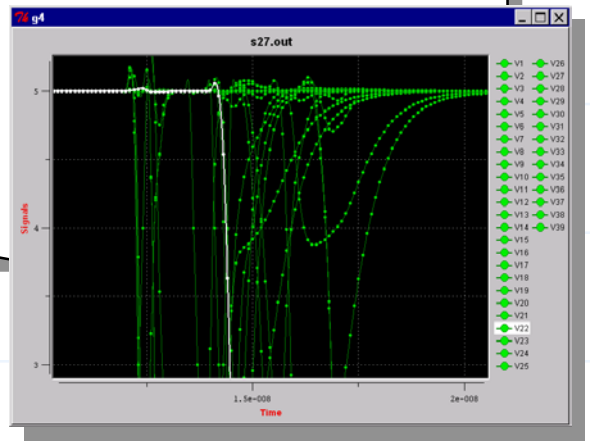
Active legend (cont'd)

Both legend and elements have **`activate`** and **`deactivate`** operations.

When active:

- Legend entry drawn with **`-activebackground`** color.
- Element is drawn with active colors, on top of plot (regardless of Z-order).

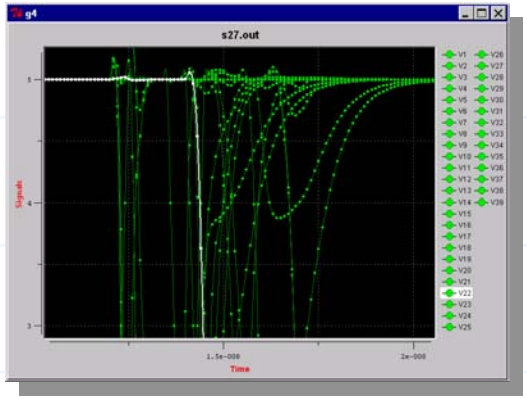
```
proc Highlight { graph elem } {
    $graph element activate $elem
    $graph legend activate $elem
}
proc Unhighlight { graph elem } {
    $graph element deactivate $elem
    $graph legend deactivate $elem
}
```



Data handling

Managing large sets of X-Y coordinate data as Tcl lists is slow, cumbersome.

- Doesn't scale. Ok for demos, not for real life problems.



39 elements
361 x-values
361 y-values

28,158 String-to-double conversions

Don't they all have the same x-values?

Problems:

- Two representations of data.
  - ✓ Tcl lists representing X and Y coordinate vectors.
  - ✓ Internal binary format (doubles) stored in graph widget.
- String-to-binary conversions are expensive.
  - ✓ Often, data starts in binary format. Converted to strings, just to be converted back to doubles.
- Widget doesn't have data analysis operations.
  - ✓ Data *trapped* inside of widget.

Vectors

Vector is a *data object*.

- Represents array of doubles.

Access data via either Tcl command or array variable.

- Creating vector automatically creates both new Tcl command and array.

New Tcl command by the same name as vector is created.

```
blt::vector create x
x set { 0 1e-10 2e-10 3e-10 4e-10 5e-10 6e-10 7e-10 8e-10
   9e-10 1e-09 1.1e-09 1.2e-09 1.3e-09 1.4e-09 1.5e-09 … }
puts [x length]
puts $x(0)
```

Variable by the same name as vector is also created

Recognized by graph widgets.

- Can be used instead of lists of numbers.
- Graph automatically redraws when vector is changed.
- Data is shared. More than one graph can use same vector.

```
blt::vector create x
blt::vector create y
x set {…}
y set {…}
.g element configure -xdata x -ydata y
```

Vectors: array interface

Can access vector data via Tcl array variable.

- Arrays indexed by integers, starting from 0.
- Special indices (user-defined ones can be added):

end	Returns the last value.
++end	Automatically appends new slot to vector. Index of new slot.
min	Returns the minimum value.
max	Returns the maximum value.

- Range of elements can be specified (with colons).

```
blt::vector create x(50)
set x(0) 20.0
set x(end) 30.0
set x(++end) 31.0
puts "Range of values: $x(min) to $x(max)"
puts "First twenty values are $x(0:19)"
set x(40:50) -1
```

Can specify initial vector size.
All values default to 0.0.

Vectors: command interface

Tcl command associated with vector has several operations:

- **append** Appends the lists of values or other vectors.
- **binread** Reads binary data into vector.
- **delete** Deletes elements by index.
- **dup** Creates a copy of vector.
- **expr** Computes vector expressions.
- **length** Queries or resets number of elements.
- **merge** Returns list of merged elements of two of more vectors.
- **range** Returns values of vector elements between two indices.
- **search** Returns indices of a specified value or range of values.
- **seq** Generates a sequence of values.
- **sort** Sorts the vector. If other vectors are listed, rearranged in same manner.
- **variable** Maps a Tcl variable to vector.

```
proc myProc { vector } {
    $vector variable x
    set x(0) 20.0
    set x(end) 30.0
}
```

Kind of like "upvar". Remaps the vector's variable to the local variable "x".

Vectors: expressions

Vector's **expr** operation does both scalar and vector math.

- Arithmetic operators `+- * / ^ %`
- Logic operators `== != ! && || < > <= >=`
- Math functions `abs acos asin atan ceil cos cosh exp floor hypot log log10 sin sinh sqrt tan tanh`
- Additional functions `adev kurtosis length max mean median min norm prod q1 q3 random round srandom sdev skew sort var`

```
x expr { x + 1 }
x expr { x + y }
x expr { x * (y + 1)}
x expr { sin(x) + cos(y) + sin($number) }

set sum [blt::vector expr sum(x)]
```

Can build data analysis routines from vector expressions.

- Fast. Computes vector expressions in C, not Tcl.

Graphs and vectors

Graph widgets accept vectors instead of Tcl lists for data.

```
blt::vector create x
blt::vector create y
blt::graph .g1
blt::graph .g2
.g1 element create line1 -xdata x -ydata y
.g2 element create line1 -xdata x -ydata y
```

- Two different graphs can share the same vectors.

Graphs automatically notified/redrawn when vector changes.

```
set x(0) 2.0
set y(0) 3.2
```

- No code needed to reconfigure the graph elements.

Vectors: C API

C API also exists for vectors.

- Read data from special file format.
- Custom data analysis routines.
- Fast graph updates.

Example: Load data from C.

- Add new Tcl command **`LoadData`** to call vector C API.

```
blt::vector create x
blt::vector create y
blt::graph .g
.g element create line1 -xdata x -ydata y
…
LoadData x y
```

Use two vector C API functions:

`Blt_GetVector`	Retrieves an existing vector.
`Blt_ResetVector`	Resets the vector data and notifies graphs.

Example: writing to vectors

```
#include "tcl.h"
#include "blt.h"
static int
LoadDataCmd(ClientData clientData, Tcl_Interp *interp, int argc, char **argv)
{
   Blt_Vector *xVec, *yVec;
   double *x, *y;
   if (Blt_GetVector(interp, argv[1], &xVec) != TCL_OK) {
         return TCL_ERROR;
   }
   if (Blt_GetVector(interp, argv[2], &yVec) != TCL_OK) {
         return TCL_ERROR;
   }
   x = (double *)malloc(sizeof(double) * 1000);
   y = (double *)malloc(sizeof(double) * 1000);
   /* Fill the arrays */
   if (Blt_ResetVector(interp, xVec, x, 100, 1000, TCL_DYNAMIC) != TCL_OK) {
         return TCL_ERROR;
   }
   if (Blt_ResetVector(interp, yVec, y, 100, 1000, TCL_DYNAMIC) != TCL_OK) {
         return TCL_ERROR;
   }
   return TCL_OK;
}
```

argv[1] is "x"

argv[2] is "y"

Arrays of doubles.

elements used.

elements in array.

Example: reading from vector

Vector token really pointer to actual vector, not a copy (so be careful).

Use macros to access vector fields:

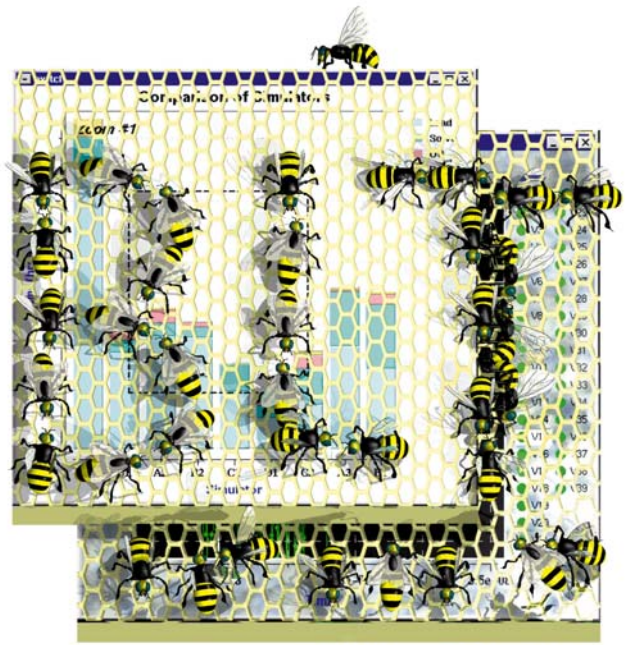
- **`Blt_VecData, Blt_VecLength, Blt_VecSize`**

```
#include “tcl.h”
#include “blt.h”
static int
GetDataCmd(ClientData clientData, Tcl_Interp *interp, int argc, char **argv)
{
   Blt_Vector *xVec;
   double *x;
   int size, length, n;
   if (Blt_GetVector(interp, argv[1], &xVec) != TCL_OK) {
          return TCL_ERROR;
   }
   x = Blt_VecData(xVec);
   length = Blt_VecLength(xVec);
   size = Blt_VecSize(xVec);
   for (n = 0; n < length; n++) {
     /* Do something with data */
     printf(“#%d is %f\n”, n, x[n]);
   }
   printf(“There are %d free slots left\n”, size - length);
   return TCL_OK;
}
```

If you change the array, you must call Blt_ResetVector.

Graph: advanced features

- Tiling graphs.
- Multiple axes.
- Pens and weights.
- Controlling graph margins.



http://www.tcltk.com/blt/

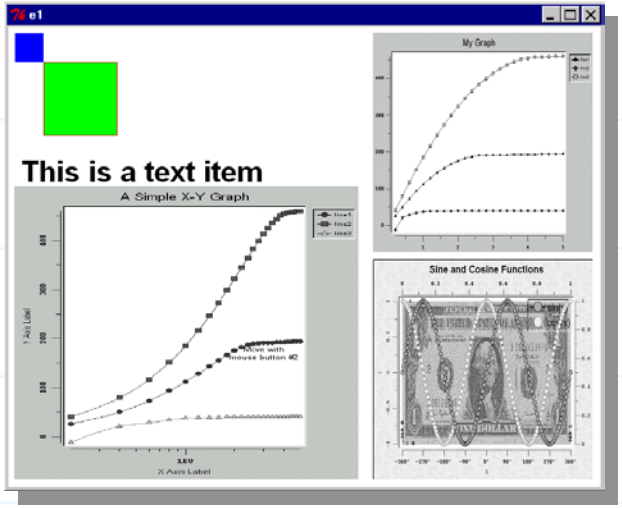
Customized printing

How do I tile graphs on a single a page?

- Graph outputs only a single plot.

BLT **eps** canvas item places EPS files.

- Reads preview image format output by graph.
- Prints item using encapsulated PostScript, not screen image.
- EPS is scaled/translated accordingly to match canvas item.
- Use canvas code as template for tiling, etc.
- Regular canvas items for annotations.




```
canvas .c -width 6.75i -height 5.25i -bg white
.c create eps 10 620 -file xy.ps -anchor sw
.c create eps 500 10 -file g1.ps -width 300 -height 300
.c create eps 500 320 -file out.ps -width 300 -height 300
.c create text 20 200 -text "This is a text item" \
        -anchor w -font { Helvetica 24 bold }
.c create rectangle 10 10 50 50 -fill blue
.c create rectangle 50 50 150 150 -fill green -outline red
```

EPS item (cont'd)

eps item usually drawn as filled rectangle on canvas.

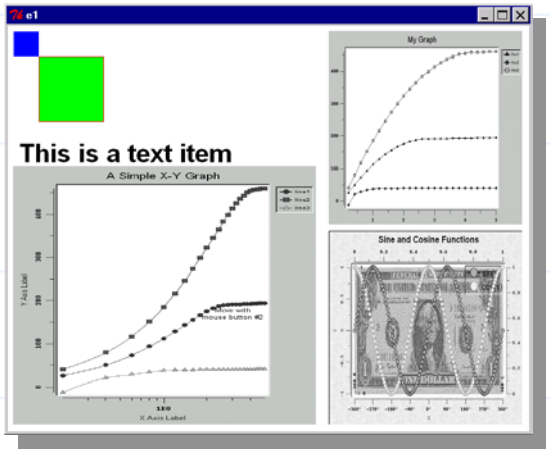
- Okay for tiling or page templates.
- Not okay for interactive layout.

Can also display EPSI preview image.

- Bitmap or grayscale (graph only).
- Must generate preview for EPS file.

Better: use Tk photo image.

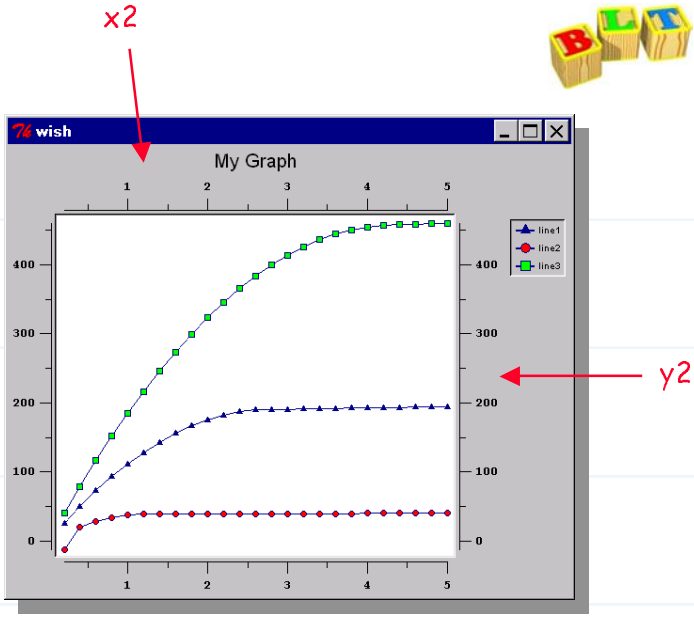
- Full color image.
- Must supply photo image for **eps** item to display.
- Use graph's **snap** operation.



Better than snapping graph window.

```
set image [image create photo]

.g snap $image                                  Image of graph drawn into photo.
.g postscript output myFile.ps
.c create eps 500 10 -file myFile.ps -image $image
```

Multiple axes



Graph has 4 default axes:

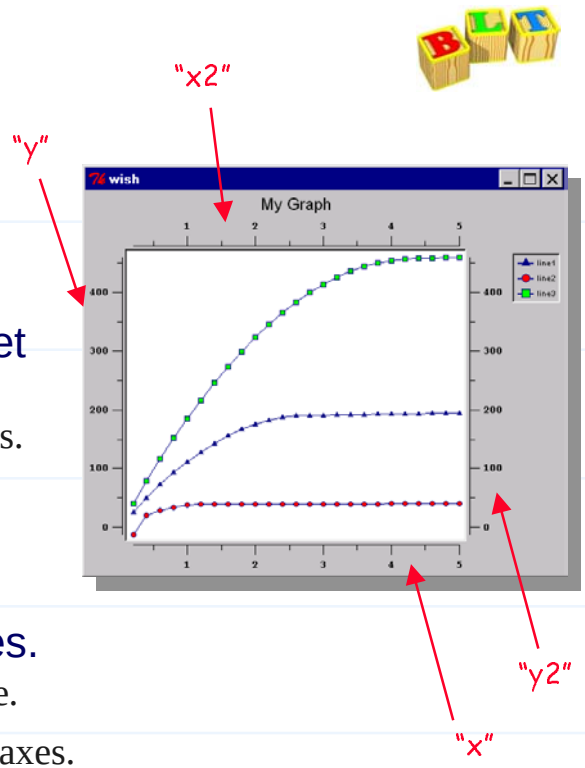
- Axis names are **x**, **y**, **x2**, and **y2**.
- **x2** and **y2** are hidden by default.

Elements, markers, and grids are mapped to specific axes.

- Mapped to **x** and **y** by default.
- **-mapx** and **-mapy** switch axes.

```
.g axis configure x2 y2 -hide no
.g element configure line1 -mapx x2 -mapy y2
.g marker configure myLine -mapx x2 -mapy y2
.g grid configure -mapx x2 -mapy y2
```

Mirroring axes



Want axes “x2” and “y2” to mirror “x” and “y”

Problem: Data range is defined only on one set of axes.

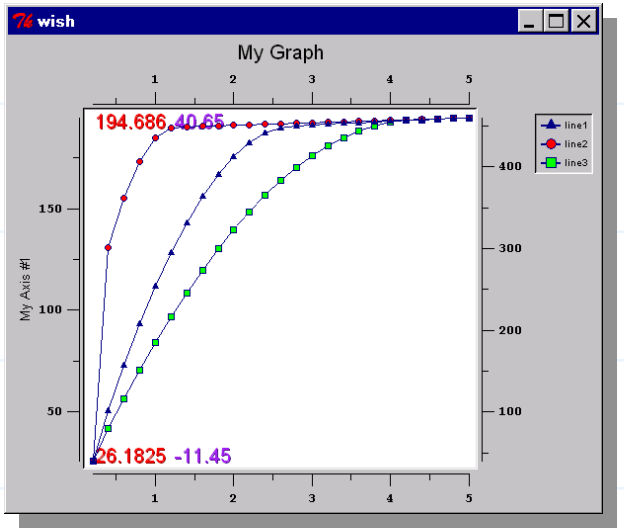
- Can only map elements to one X-axis and one Y-axis.

Replicate the data ranges on the mirrored axes.

- Use axis **`limits`** operation to get current axis range.
- Set **`-scrollmin`**, **`-scrollmax`** if scrolling both axes.

```
set lx [.g axis limits x]
set ly [.g axis limits y]
.g axis configure x2 -scrollmin [lindex $lx 0] \
        -scrollmax [lindex $lx 1]
.g axis configure y2 -scrollmin [lindex $ly 0] \
        -scrollmax [lindex $ly 1]
```

Virtual axes



Can also create virtual axes.

- Create any number of new axes.
- Axis minimum and maximum displayed in plotting area.
- Limits string is floating-point format descriptor.

Can replace normal axes with **xaxis**, **yaxis**, **x2axis**, and **y2axis** operations.

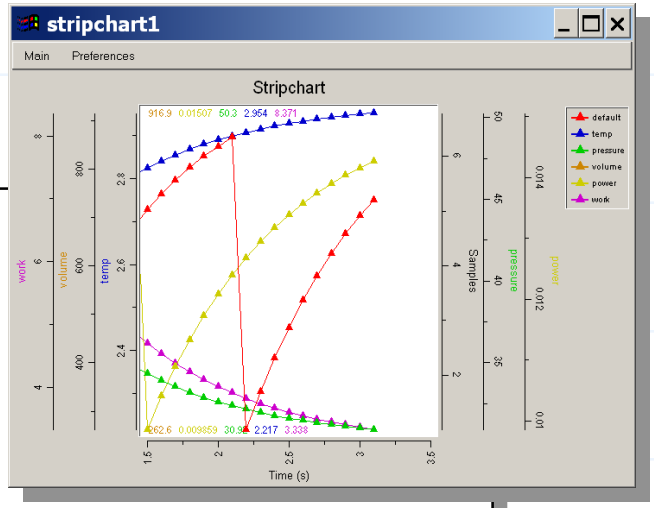
- **use** operation maps axis to margin.

```
.g axis create axis1 -title “My Axis #1” -limits “%g” \
    -limitscolor red -limitsshadow red4
.g axis create axis2 -limits “%4.2f” -limitscolor purple \
    -limitsshadow purple4
.g axis configure axis1 axis2 -limitsfont {Helvetica 12}

.g element configure line1 -mapy axis1
.g element configure line2 -mapy axis2

.g xaxis use axis1
```

Multiple axes (cont'd)

Any number of axes can be displayed.

```
.g axis create axis1 -title "temp"
.g axis create axis2 -title "volume"
.g axis create axis3 -title "work"
.g axis create axis4 -title "Samples"
.g axis create axis5 -title "pressure"
.g axis create axis6 -title "power"

.g yaxis use "axis1 axis2 axis3"
.g y2axis use "axis4 axis5 axis6"
```



use operation can takes a list of axis names.

Pens

Pen component represents drawing style for an element.

- Symbol, color, line style, etc.
- Each element has its own default pen.

Can create new pens and swap them in/out of elements.

```
.g pen create pen1 -symbol circle -color blue -linewidth 2
.g pen create pen2 -symbol cross -color red

.g element configure line1 -pen pen1
.g element configure line2 -pen pen2

.g pen configure pen1 -color yellow
```

Element is redrawn with new color.

All elements use a standard “active” pen (**`activate`** operation).

- **`activeLine`** graph, stripchart widgets
- **`activeBar`** barchart

```
.g pen configure activeLine -linewidth 0 -symbol square
.g element configure -activepen pen1
```

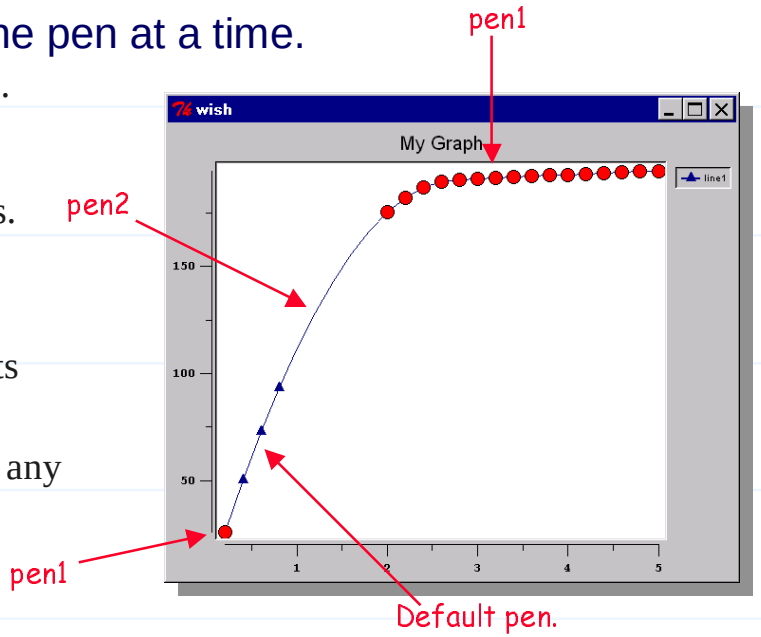
Weights

Data elements can use **more** than one pen at a time.

- Highlight outliers, unexpected values, etc.

Each element has a **weight** vector.

- Weight values correspond to a data points.
- Set **`-weight`** option. Like **`-xdata`** or **`-ydata`** options, takes vector or Tcl list.
- **`-styles`** option maps pens to data points according to weight value.
- Default pen used if weight doesn't match any range.




```
.g pen create pen1 -color red -symbol circle -outline black
.g pen create pen2 -symbol none

.g element configure line1 -weight y -styles {
    {pen1 -5 50} {pen2 100 175} {pen1 175 500}
}
```

Setting margins

Margin sizes automatically calculated (based on axis values, etc.)

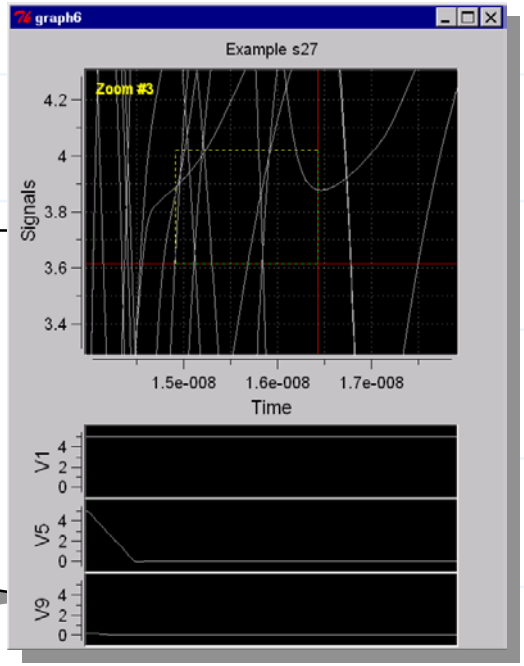
Sometimes want to override computed values:

- Useful when displaying two graphs side-by-side.
- You can reset margins with **`-leftmargin`**, **`-rightmargin`**, **`-topmargin`**, and **`-bottommargin`** graph configuration options.

To determine current margin:

- Graph's **`extents`** operation reports margin sizes.
- Options **`-leftvariable`**, etc. specify variables, set when margins are updated.

```
.g1 configure -leftvariable left
trace variable left w UpdateMargins
proc UpdateMargins { p1 p2 how } {
    global left
    .g2 configure -leftmargin $left
    .g3 configure -leftmargin $left
    .g4 configure -leftmargin $left
}
```



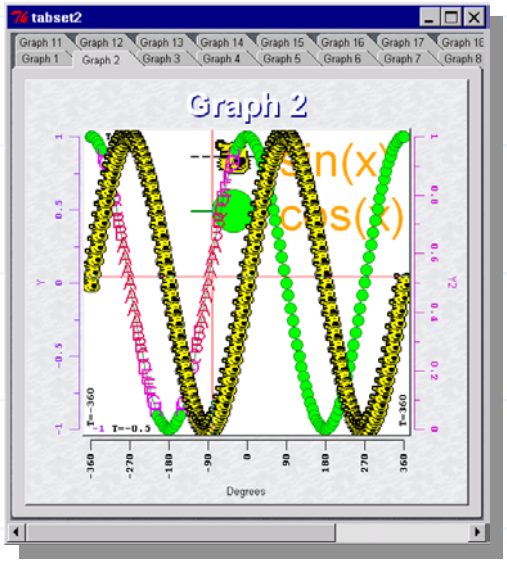
Managing graphs with tabnotebooks

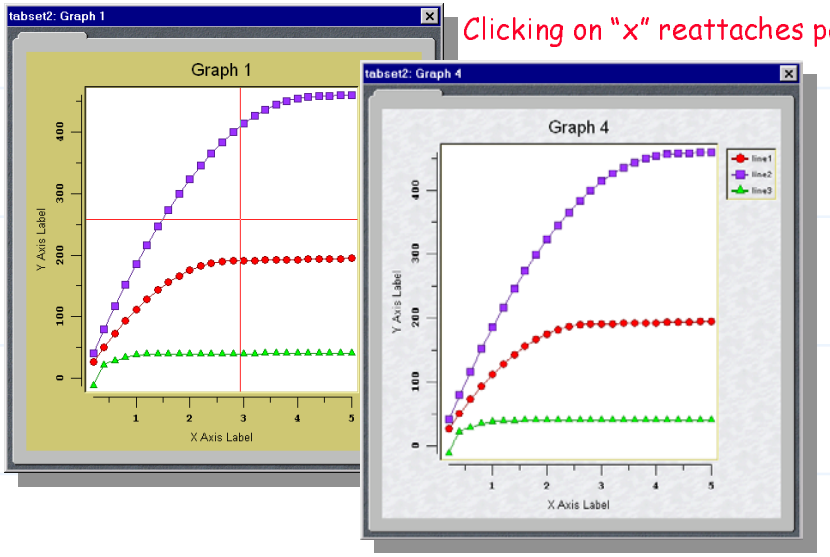
Typical for applications to generate dozens of graphs.

- Clutters screen. Hard to manage.
- Tend to reduce size of graphs.

Put graphs in tabbed notebook.

- Tear-off feature lets you compare plots side-by-side.
- Same graph can be shared by different pages.



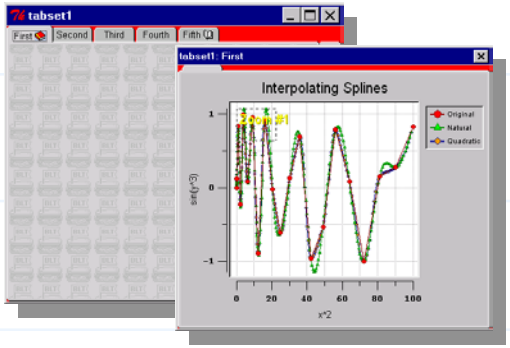


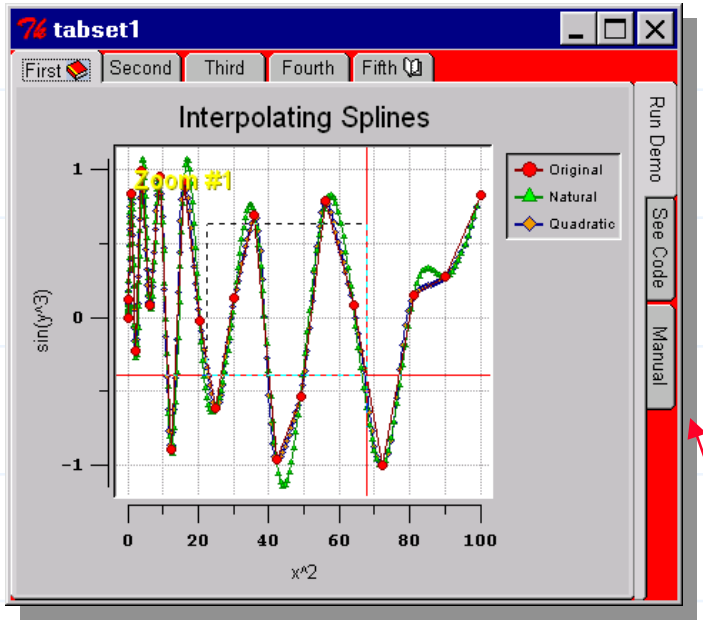
Clicking on "x" reattaches page.

tabnotebook

Tabbed notebook widget.

- Single or multi-tiered tabs.
- Scrollable (with/without scrollbar).
- Bind to individual tabs (tool tips)
- Tear off and re-attach pages.





Can use tabnotebook without pages.

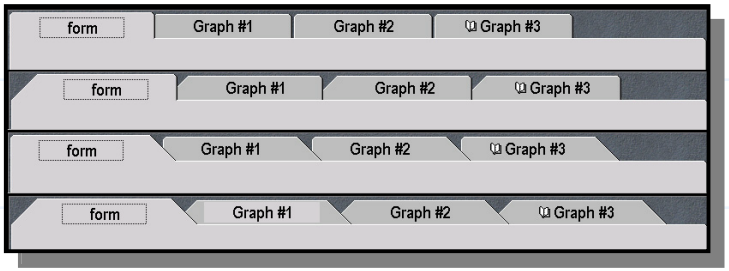
```
blt::tabnotebook .t -bg red -scrollcommand {.s set}
scrollbar .s -command {.t view} -orient horizontal
.t insert end First -window .t.graph \
        -image [image create photo -file book.gif]
.t bind First <Enter> {ToolTips "Graph of Interpolating Splines"}
.t bind First <Leave> {ToolTips ""}
```

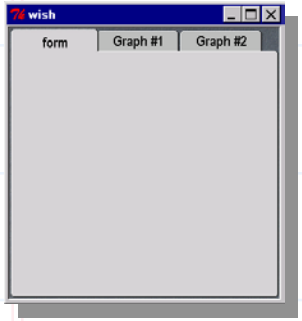
tabnotebook (cont'd)

Variety of styles supported.

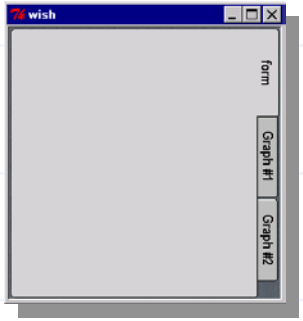
- Controlled via tabnotebook's configuration options.

`-slant none`
`-slant left`
`-slant right`
`-slant both`

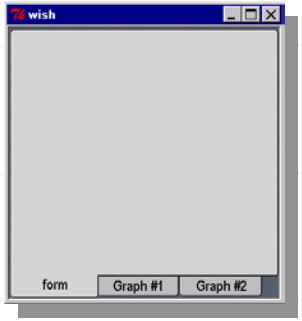




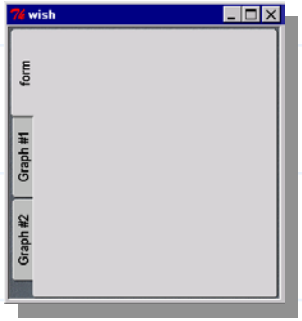
`-side top`



`-side right`

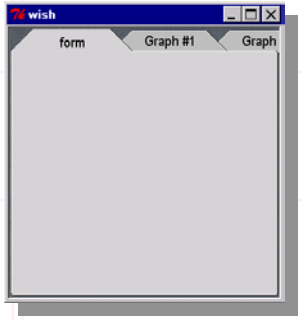


`-side bottom`

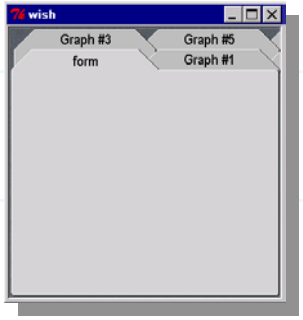


`-side left`

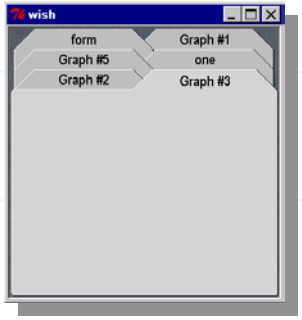
`-rotate 0`



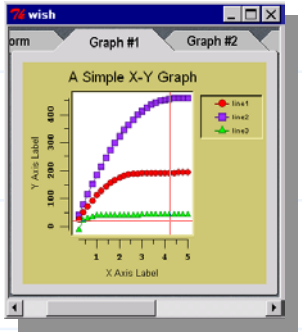
`-tiers 1`

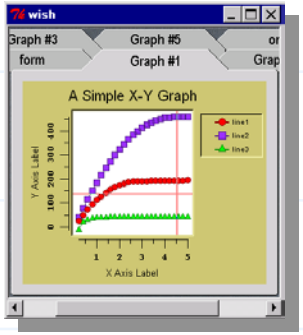


`-tiers 2`



`-tiers 3`



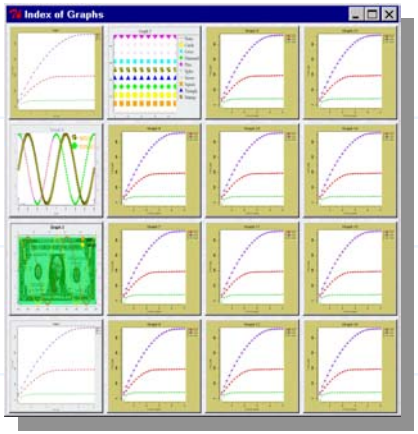


Attach scrollbar to single or multi-tiered tabs.

Managing graphs with tabnotebooks (cont'd)

Idea: Create index for graphs.

- Table of buttons, each contains thumbnail image of a graph.



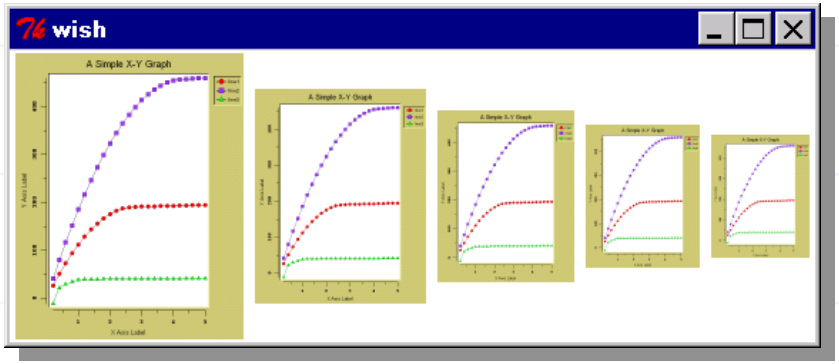
Thumbnails

- Already know how to get snapshot of a graph.

```
set image [image create photo]
.g snap $image
set thumb [image create photo]
$thumb copy $image -subsample 4 4
```

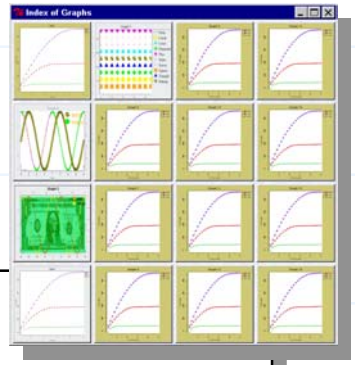
Problem: How do you resize snapshot to arbitrary size?

- Want thumbnails scaled the same.
- Tk image subsample reduces only by integer values.
- Image quality poor. Detail lost.



-subsample 3 through 7.

Managing graphs with tabnotebooks (cont'd)



winop resample operation does arbitrary resizing.

- 1-D image filters: **box**, **triangle**, **sinc**, etc.
- Eventually function will move to new image type.

```
proc Thumbnail { graph w h } {
    set image [image create photo]
    $graph snap $image
    set thumb [image create photo -width $w -height $h]
    blt::winop resample $image $thumb box box
    image delete $image
    return $thumb
}
set nTabs [.t size]
for { set tab 0 } { $tab < $nTabs } { incr tab } {
    set graph [.t tab cget $tab -window]
    button .f.b$tab -image [Thumbnail $graph 200 200] \
        -command [list .t invoke $tab ]
    blt::table .f .f.b$tab $row,$col
    …
}
```

Makes tab selected.

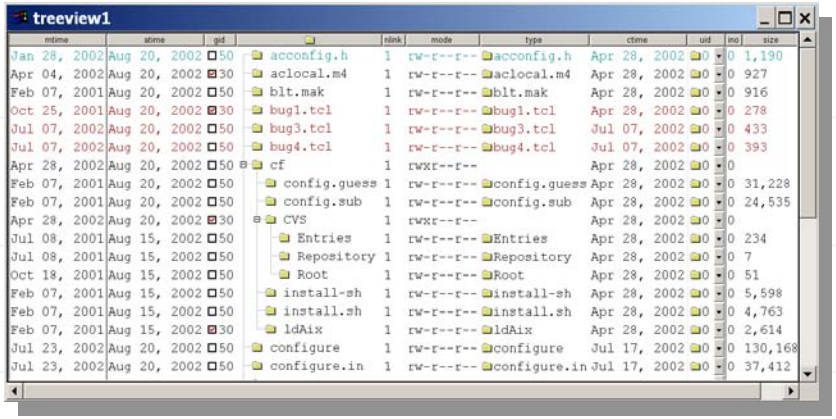
Miscellaneous features

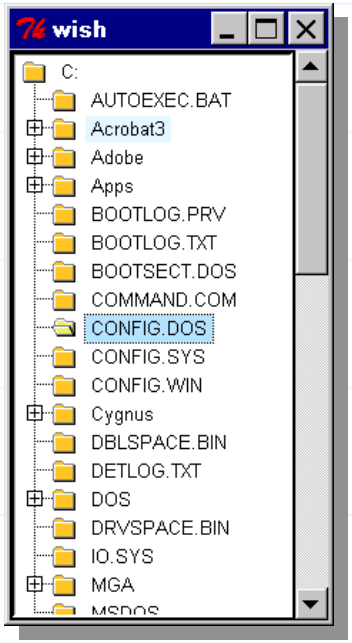
- treeview widget.
- tree data object.
- bgexec command.
- busy.
- drag&drop.
- table geometry manager.

treeview widget

Displays tree of data.

- Can show single tree or with multiple columns.





Supports incremental (lazy) insertions.

Much faster than Tcl-based versions.

- Not usual to have trees >150,000 nodes.

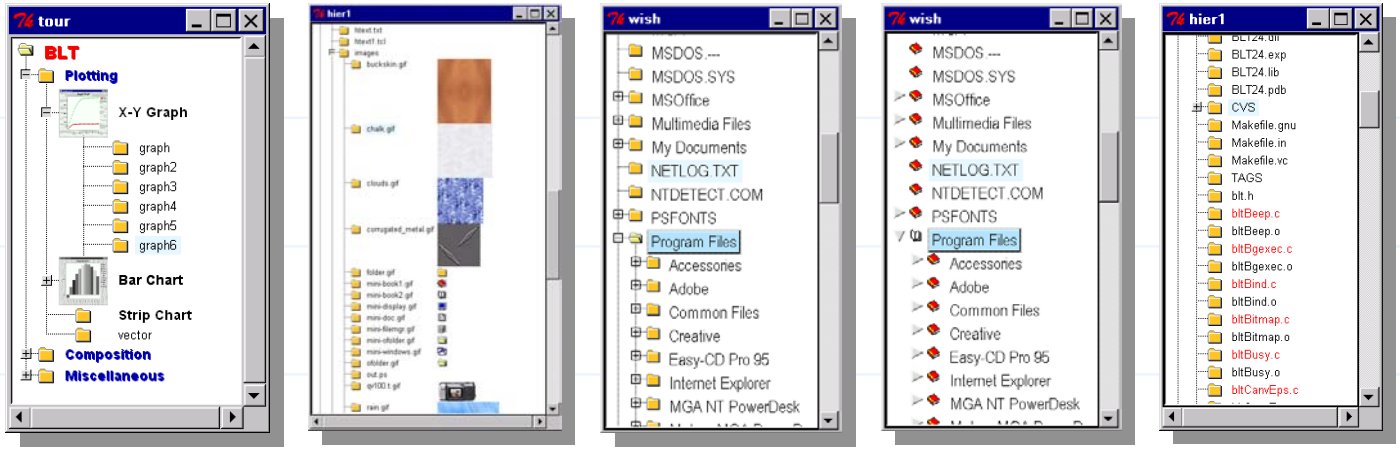
Multi-mode selection

- single, multiple, non-contiguous.

Built-in search and selection functions.

- Search on name, data, entry attributes, etc.

treeview widget (cont'd)



Variety of styles supported.

- Read-only/editable entries.
- Bind to individual entries (e.g. tool tips).
- Font, color, icons, images configurable for single entries.
- Auxiliary text and/or images displayed for entries.

Uses BLT tree data object.

tree data object

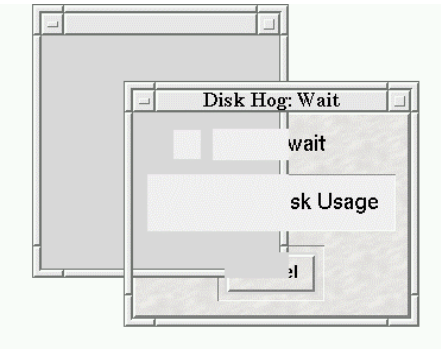
Tree data object represents a general-ordered tree of data.

Data is accessible through Tcl command or C API.

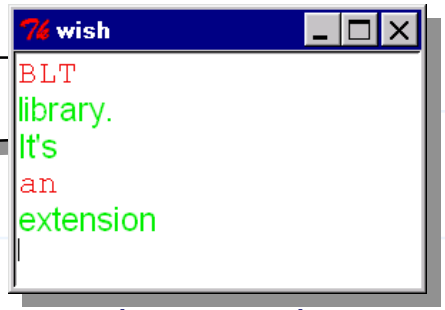
Data can be shared among clients (e.g. treeview widget).

```
set tree [blt::tree create]
proc LoadTree { tree parentNode dir } {
    set saved [pwd]
    cd $dir
    foreach f [glob *] {
        set node [$tree insert $parentNode -label $f]
        if { [file isdir $f] } {
            LoadTree $tree $node $f
        }
    }
    cd $saved
}
LoadTree $tree 0 /etc
blt::treeview .t -tree $tree
```

bgexec



set usage [exec du $dir]



Executes programs while still handling events.

- Same syntax as exec: I/O redirection and pipes.
- Collects *both* stdout and stderr.
- Faster/simpler than **`fileevent`**.

Example:

```
set info [bgexec myVar du $dir]
```

- Variable **`myVar`** is set when program finishes.
- Setting **`myVar`** yourself terminates the program.

Callback proc invoked whenever data is available on stdout or stderr.

```
text .text
.text tag configure outTag -foreground green2
.text tag configure errTag -foreground red2
proc DrawStdout {data} { .text insert end $data outTag }
proc DrawStderr {data} { .text insert end $data errTag }
blt::bgexec myVar -onoutput DrawStdout \
    -onerror DrawStderr myProgram &
```

busy

Makes widgets busy. Widgets ignore user-interactions.

- Mouse, keyboard events etc.
- Creates invisible window. Shields widgets from receiving events.

Better than **grab** for most situations.

- Allow interactions in more than one widget.
- Stopping interactions in specific widgets.
- De-bouncing mouse clicks/key presses.

Configurable cursor.

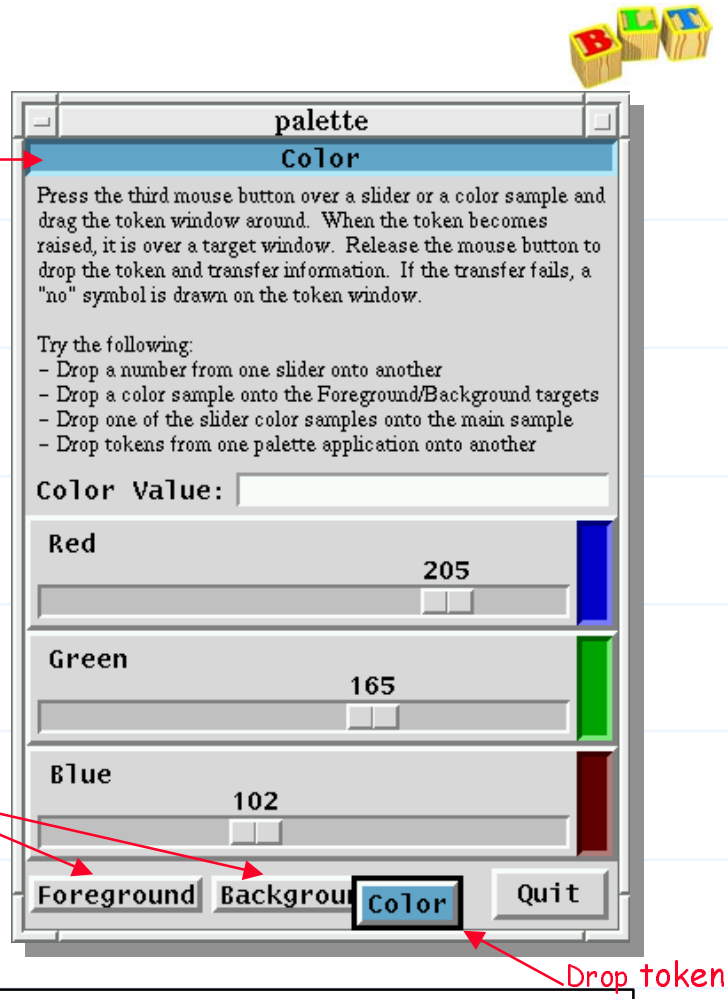
- Defaults to hourglass/watch.

```
blt::busy hold .frame
update
blt::busy release .frame
```

drag&drop

Transfers data between widgets and applications

- Data transferred with **send** command.
- Any widget can be registered as drag source, drop target, or both.
- Configurable drop token.
- Soon: interoperation with Windows CDE drag-and-drop.



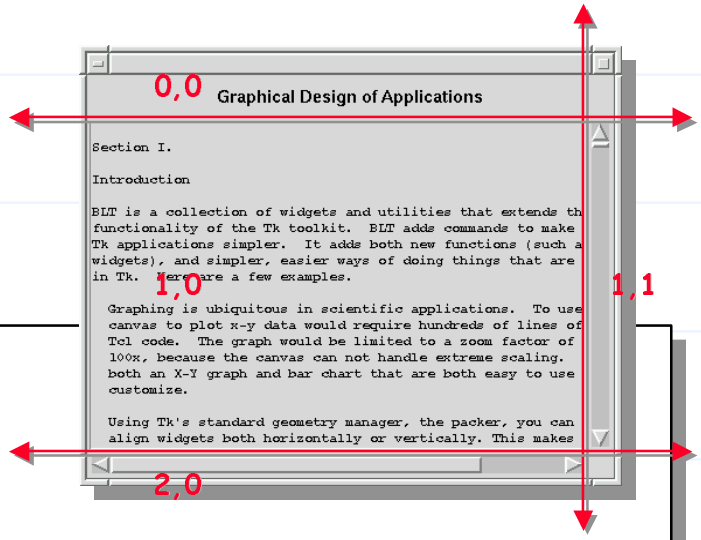

```
blt::drag&drop source .sample \
        -packagecmd {PackageColor %t}
blt::drag&drop source .sample handler Color
blt::drag&drop target .sample handler Color \
        {ReceiveColor %v}
```

table

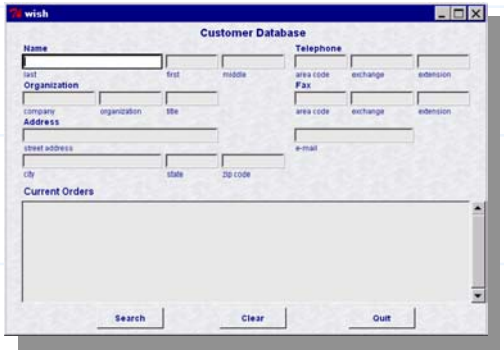
Grid-based geometry manager.

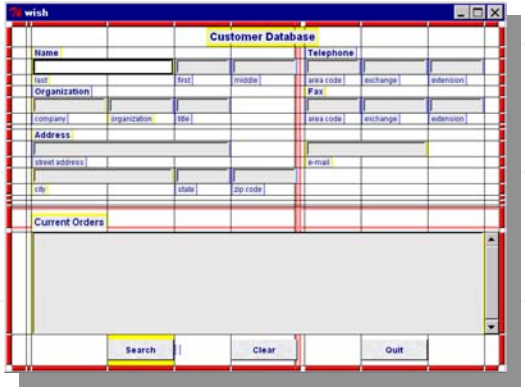
- Position widgets by row/column.
- Father of Tk grid.
- Can bound row/column sizes.

```
blt::table . \
    0,0 .label -cspan 2 \
    1,0 .text -fill both \
    1,1 .vs -fill y \
    2,0 .hs -fill x
```



- Insert/delete rows and columns.
- Debugging mode.





General information

What version of Tcl/Tk is required?

- Any stable release. Tcl 7.5, Tk 4.1 through Tcl/Tk 8.3.4 all work.

Can I use BLT in a commercial product?

- Yes, it's free to copy and use. No royalties.

Where do I get the latest version?

- **`http://www.sourceforge.net/projects/blt/files`**
- Sources for latest version.
- Windows binaries available.

Where do I send bug reports and requests?

- Send to both addresses. Put "BLT" in the subject line:

  `ghowlett@grandecom.net`

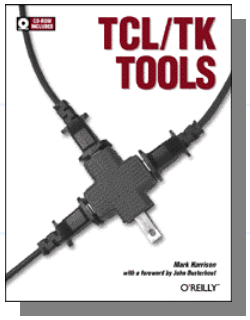
  `gah@siliconmetrics.com`

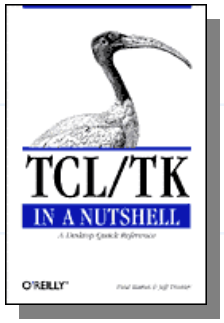
- If you can, send a script that demonstrates the bug.

General information (cont'd)

Books

- **Tcl/Tk Tools**
  edited by Mark Harrison.
- **Tcl/Tk in a Nutshell**
  by Paul Raines and Jeff Trantor.





What does BLT stand for?

- *Bell Labs Toolkit*
  Bacon, Lettuce, and Tomato
  Better Luck Tomorrow
- Whatever you want it to…