Building Applications with BLT

George A. Howlett
Cadence Design Systems, Inc.
Allentown, Pennsylvania
What is BLT?

Set of widgets and new commands.
- Extends the Tcl interpreter, no patching required.

Features:
- Graph, stripchart, barchart widgets.
- Table geometry manager
- Hierarchical listbox/table widgets.
- 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 soon
What is BLT?
Graphing widgets

Three plotting widgets available:

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

Many features span across all three widgets.
How do I use BLT?

Run special shell with statically linked BLT commands.

```bash
$ bltwish
```

Dynamically load the BLT package into a vanilla wish.

```bash
$ wish
% package require BLT
```
Where are the BLT commands?

Commands live in `blt` namespace.
- Not automatically exported into global namespace.

Two ways to access the BLT commands.
- Prefix BLT commands with `blt::`
  ```
  package require BLT
  blt::graph .g
  ```
- Import all the BLT commands into the global namespace.
  ```
  package require BLT
  namespace import blt::*
  graph .g
  ```

Can't find "graph" command in the global namespace.
Building applications with BLT

How to plot data with the graph widget.
Zooming and scrolling.
Annotations.
Building your own zooming graph.
Customizing the graph:
  ● Axes, legend, grid, crosshairs.
Interactive graphs.
Data handling.
Printing.
Advanced features.
Managing graphs with tabsets.
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

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

X-Y coordinates are lists of numbers.

Assorted configuration options control element’s appearance.

- `-symbol`
- `-linewidth`
- `-fill`
- `-outline`
- `-smooth`

```
package require BLT
namespace import blt::*

graph .g -title "My Graph"
pack .g

.g element create linel -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 ... }
```
Data elements

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.

```plaintext
.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 and scrolling

Graph’s **axis** operation controls range of points displayed.

```plaintext
.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.

To scroll, add or subtract *same* amount from both min and max.
Scrolling (cont’d)

Can attach a scrollbar to any axis.

- Just like attaching scrollbar to any Tk widget.
- Viewport defined by the current \(-\text{min}\) and \(-\text{max}\) values.

```bash
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 }
```
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.

Labels customized by specifying callback proc.

```tcl
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.
Annotations

Markers are used to highlight or annotate areas.

Six types of markers:
- text
- line
- polygon
- bitmap
- image
- window

```
.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.

```tcl
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.
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)

First corner of region saved in global variables $x0$ and $y0$.

Line marker can be erased with redrawing graph with \texttt{-xor} option.

Marker \texttt{coords} operation changes line coordinates.

Delete marker when done.
Converting to/from graph coordinates

Example doesn’t work. Need to translate screen to graph coordinates.

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

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

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

```tcl```
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.

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

```tcl
proc PushZoom { graph } {
    global zoomStack
    set x1 [string range [lindex $graph options axes 0] 10 12]
    set x1 [string range [lindex $graph options axes 0] 10 12]
    set x1 [string range [lindex $graph options axes 0] 10 12]
    set x1 [string range [lindex $graph options axes 0] 10 12]
    set y1 [string range [lindex $graph options axes 0] 10 12]
    set y1 [string range [lindex $graph options axes 0] 10 12]
    set y1 [string range [lindex $graph options axes 0] 10 12]
    set y1 [string range [lindex $graph options axes 0] 10 12]
    set cmd "$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**

The legend component controls the position/appearance of the legend.

```plaintext
graph legend configure ?option value...
```

- **-position right**
- **-position left**
- **-position top**
- **-position bottom**
- **-position plotarea**
- **-position @450,100**
- **-raised yes**
- **-hide yes**
Grids

**grid** component controls appearance of built-in grid.

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

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.

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

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

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 linel

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.
Binding to graph components

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

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

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

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

```bash
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)

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

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

- 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).

```bash
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.

**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.

```
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)
```

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.

```
vector create x
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):
  
<table>
<thead>
<tr>
<th>Index</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>end</code></td>
<td>Returns the last value.</td>
</tr>
<tr>
<td><code>++end</code></td>
<td>Automatically appends new slot to vector.</td>
</tr>
<tr>
<td><code>min</code></td>
<td>Returns the minimum value.</td>
</tr>
<tr>
<td><code>max</code></td>
<td>Returns the maximum value.</td>
</tr>
</tbody>
</table>

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

```tcl
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 or 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.

```tcl
proc myProc { vector } {
    $vector variable $vector variable xx
    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`
- Addition functions: `adev kurtosis length max mean median min norm prod q1 q3 random round random sdev skew sort var`

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

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

Can build data analysis routines from vector expressions.

**Graphs and vectors**

Graph widgets accept vectors instead of Tcl lists for data.

```tcl
vector create x
vector create y
graph .g1
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.**

```tcl
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.

```
vector create x
vector create y
graph .g
.gs 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

```c
#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;
}
```
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`

```c
#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`.
Printing graphs

Graph’s **postscript** operation generates encapsulated PostScript.

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

- File name is optional. PostScript is returned as the result of **output** operation.
- Other options control graph, border, and paper size.
Printing under Windows 95/NT

**printer** command lets you send raw EPS to a PostScript printer.

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

Query printer settings with **getattr** operation. Written to array variable.

```
set pid [printer open {QMS ColorScript 100 v49.4}]
printer getattr $pid myArray
puts "Paper size is $myArray(PaperSize)"
puts "Page orientation is $myArray(Orientation)"
```

Adjust printer settings with **setattr** operation.

```
set myArray(PaperSize) Letter
set myArray(Orientation) Landscape
printer setattr $pid myArray
printer write $pid $output
printer close $pid
```
Printing to non-PS printers

Graph has two Windows-specific print operations (still experimental).

**print1**
- Write bitmap image to printer.
- Usually works regardless of printer capabilities.
- Poorer quality. Jagged lines and fonts.

**print2**
- Draws directly to print device.
- Might not work on all printers.
- Quality is much better.

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

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

How do I tile graphs on a single a page?
- Graph outputs only a single plot.

New **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.

```bash
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 gl.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.**
- Ok for tiling or page templates.
- Not very good for interactive layout.

**But can also display EPSI preview image.**
- Poorer quality: 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 to get snapshot of graph.

```
Better than snapping graph window.

set image [image create photo]
.g snap $image
.g postscript output myfile.ps
.c create eps 500 10 -file myfile.ps -image $image
```
Graph: advanced features

Mirror axes.
Virtual axes.
Pens and weights.
Controlling graph margins.

http://www.tcltk.com/blt/
Graph: advanced features

Can display more than 2 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.
- \(-\text{mapx}\) and \(-\text{mapy}\) 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
```

Mirror axes: Use \texttt{axis limits} operation to get current axis range.

```
set lx [g axis limits x]
set ly [g axis limits y]
g axis configure x2 -min [lindex $lx 0] -max [lindex $lx 1]
g axis configure y2 -min [lindex $ly 0] -max [lindex $ly 1]
```

Returns list of min and max limits.
Advanced features (cont’d)

Graph also supports **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 one or more axes.

```
.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
```
Advanced features (cont’d)

Each element has its own default pen.

- Represents element’s symbol, color, line style, etc.

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 -lineweight 0 -symbol square
.g element configure -activepen pen1
```
Advanced features (cont’d)

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}
}
```
Advanced features (cont’d)

Margins automatically calculated (based on axis values, etc.)

Sometimes want to override computed values:

- You can reset margins with `-leftmargin`, `-rightmargin`, `-topmargin`, and `-bottommargin` graph configuration options.

- Useful when displaying graphs side-by-side.

To determine current margin:

- Graph’s `extents` operation reports margin sizes.
- Options `-leftvariable`, `-rightvariable`, `-topvariable`, and `-bottomvariable` 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 tabsets

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.
Managing graphs with tabsets (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.
Managing graphs with tabsets (cont’d)

**winop resample** operation does arbitrary resizing.

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

```tcl
proc Thumbnail { graph w h } {
    set image [image create photo] $graph snap $image
    set thumb [image create photo -width $w -height $h] winop resample $image $thumb 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 $tab ]
    table .f .f.b$tab $row,$col ...
}
```

Makes tab selected.
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:
```bash
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.
```bash
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 }
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 many 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.

```
busy hold .frame
update
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.

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

- Displays tree of data.
- Incremental (lazy) insertions.
- Much faster than Tcl-based versions.
- Designed to work with companion widget.
  - Canvas, listbox, etc.
  - Example: `-selectcommand` callback proc.
- Multi-mode selection: single, multiple, non-contiguous.

```bash
hierbox .h -opencommand {AddEntries %P} \ 
  -closecommand {.h delete %n 0 end}
proc AddEntries { dir } {
  if { [file isdirectory $dir] } {
    eval .h insert end [lsort [glob $dir/*]]
    eval .h entry configure [lsort [glob $dir/*/]] \ 
      -button yes
  }
}
.h configure root -label "C:"
```
hierbox (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.

Built-in search and selection functions.
- Search on name, data, entry attributes, etc.
- hieritable widget uses new tree data object.

```
set nodes [.h find -glob -name *.c]
eval .h entry configure $nodes -labelcolor red

.h find -glob -name *.gif -exec { 
   .h entry configure %n \ 
   -image [image create photo -file %P] 
}
```
Grid-based geometry manager.

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

```
\begin{table}
\begin{tabular}{ll}
0,0 & .label -cspan 2 \\
1,0 & .text -fill both \\
1,1 & .vs -fill y \\
2,0 & .hs -fill x
\end{tabular}
\end{table}
```

- Insert/delete rows and columns.
- Debugging mode.
Tabbed notebook widget.

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

```bash
$tabset .t -bg red -scrollcommand {.s set}
$scrollbar .s -command {.t view} -orient horizontal
$.
$t insert end First -window .t.graph \ 
  -selectcommand {SelectTab "First"} \ 
  -image [image create photo -file book.gif]
$t bind First <Enter> {ToolTips "Graph of Interpolating Splines"}
$t bind First <Leave> {ToolTips ""}
```
tabset (cont’d)

Variety of styles supported.
- Controlled via tabset’s configuration options.

Variety of styles supported.
- Controlled via tabset’s configuration options.

Attach scrollbar to single or multi-tiered tabs.
General Information

What version of Tcl/Tk is required?
- Tcl 7.5, Tk 4.1 through Tcl/Tk 8.3.0 (latest release) all work.

Can I use BLT in a commercial product?
- Free to copy and use. No royalties.

Where do I get the latest version?
- [http://www.tcltk.com/blt](http://www.tcltk.com/blt)
  - 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@fast.net
  - gah@cadence.com
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…