Pad++ Reference Manual (Version 0.9)

Introduction

This reference manual describes the complete Tcl API to Pad++. It describes how to create and modify a Pad++ widget, and all the commands associated with a Pad++ widget that allow you to create and modify items, attach event bindings to them, navigate within the Pad++ widget, etc.

This document in organized into the following sections:

- **Padwish Synopsis**
- **TCL Synopsis**
- **Widget-Specific Options**
- **Widget Commands**
- **Overview of Item Types**
- **Default Bindings**
- **Global TCL Variables**
- **KPL-Pad++ Interface**

Each section contains all the relevant entries in alphabetical order. Related commands and options are also grouped together here to show which commands are related. Every command and itemconfigure option are listed. Note that change bars appear wherever this document differs from the previous version.

Related Commands and Options

Items

`create`**[21]**	Create new items
`delete`**[23]**	Delete existing items
`find`**[31]**	Search for items by various keys
`isvisible` **[55]**	Return true if the specified item is visible.
`itemconfigure`	Configure existing items
`lower`**[60]**	Push an item lower in the drawing order
`pick`**[65]**	Find the item under a point
`popcoordframe`**[66]**	Pop a relative coordinate frame off of the stack
`pushcoordframe`**[68]**	Add a new relative coordinate frame to the stack
`raise`**[69]**	Bring an item higher in the drawing order
`resetcoordframe`**[75]**	Reset coordinate frame stack to empty
`setid`**[82]**	Change the id of an item
`text`**[91]**	Modify text item
`type`**[93]**	Get the type of an item

`-aliases [1]`	(Read-only) Returns all aliases of the item
`-arrow` [7]	Whether to draw arrow heads with this item
`-arrowshape [8]`	The shape of drawn arrow heads
`-height [26]`	Height of an item. Normally computed, but can be set to squash/stretch item
`-html` [27]	The HTML item associated with an htmlanchor
`-htmlanchors [28]`	The anchors associated with an HTML page
`-image [29]`	Image data associated with item (allocated by image alloc)
`-info [30]`	A place to store application-specific information with an item

`-ismap [31]`	True if an htmlanchor is an image map
`-lock [35]`	Locks an item so it can not be modified or deleted
`-state [53]`	State of an item (such as visited, unvisited, etc.)
`-sticky` [54]	Specifies if an item should stay put when the view changes
`-text [56]`	The text of any item containing text
`-timerrate [57]`	Frequency timerscript should fire
`-timerscript [58]`	Script associated with an item that fires at regular intervals
`-title [59]`	Some items only: Title of an item
`-url [63]`	The URL associated with an item
`-width [68]`	Width of an item. Normally computed, but can be set to squash/stretch item
`-zoomaction [70]`	A script that gets evaluated when an item is scaled larger or smaller than a set size

Item Transformations

`animate` **[1]**	Animate item options asynchronously
`bbox` **[10]**	Get the bounding box of an item
`coords` **[20]**	Change the coordinates of an item
`getsize`**[44]**	Get the size of an item (possibly within portals)
`rotate` **[76]**	Rotate an item
`scale` **[77]**	Change the size of an item relatively
`slide` **[88]**	Move an item relatively in (x, y)
`-anchor [3]`	The part of the item that `-position` refers to
`-anchorpt [4]`	The (x, y) portion of `-position`
`-angle [5]`	Specifies absolute rotation of item
`-anglectr [6]`	Specifies absolute rotation of item, rotating about specified point
`-position [47]`	The absolute position of the object (x, y, scale)
`-rposition [51]`	The relative position of the object (to groups)
`-scale [52]`	The (scale) portion of `-position`

View Transformations

`center` **[15]**	Change the view so as to center an item
`centerbbox` **[16]**	Change the view so as to center a bounding box
`getview` **[47]**	Get the current view (possibly within portals)
`getzoom` **[48]**	Get the current view magnification (possibly within portals)
`moveto` **[62]**	Change the view (possibly within portals)
`zoom`**[99]**	Zoom the view around a specified point
`-lookon [36]`	Specifies the pad widget this item sees
`-view [65]`	Specifies the view this item sees
`-viewscript` [66]	A script that gets evaluated whenever the view is changed

Tags

`addtag`**[5]**	Add a tag to an item
`deletetag` **[23]**	Delete a tag from an item
`dtag`**[23]**	Synonym for deletetag
`gettags` **[45]**	Get the tags an item has
`hastag` **[51]**	Determine if an item has a particular tag
`-tags [55]`	List of tags associated with an item

Events

`bind` **[11]**	Create, modify, access, or delete event bindings
`bindtags` **[12]**	Specify whether events should go to the most-specific or most-general description

`focus` **[32]**	Set the focus for keyboard events
`modifier` **[61]**	Manipulate user-defined modifiers
`-events` [20]	True if item receives events, false otherwise

Groups

`addgroupmember` **[2]**	Add an item to a group
`getgroup` **[38]**	Get the group an item belongs to
`removegroupmember` **[72]**	Remove an item from a group
`-divisible` [16]	True if events go through a group to its members
`-members [38]`	The list of members of a group

Layout

`grid` **[50]**	Manage item layout in a grid as with the Tk grid command
`layout` **[58]**	Layout items once
`tree` **[92]**	Manage item layout with a dynamic graphical-fisheye view tree

Rendering

`damage`**[22]**	Specify that a group of items needs to be redrawn
`update` **[94]**	Force any requested render requests to occur immediately
`-alwaysrender` [2]	True if the item must be rendered, even if the system is slow and the item is small
`-border [10]`	Specifies border color of item
`-borderwidth` [11]	Specifies width of border
`-capstyle [12]`	Specifies how to draw line ends
`-clipping [13]`	Controls if items are clipped to their bounding box when rendered
`-dither` [15]	Render with dithering
`-faderange [21]`	Range over which an item fades in or out
`-fill` [23]	Specifies fill color of item
`-font` [24]	Specifies font to use for text
`-joinstyle [32]`	Specifies how to draw the joints within multi-point lines
`-layer [33]`	The layer an item is on
`-noisedata` [42]	Specifies parameters to render item with noise
`-maxsize` [37]	The maximum size an item is rendered it (absolute or relative to window size)
`-minsize` [41]	The minimum size an item is rendered it (absolute or relative to window size)
`-pen` [45]	Specifies pen color of item
`-penwidth` [46]	Specifies width of pen
`-relief [49]`	Specifies how border should be rendered (raised, flat, sunken, ridge, groove)
`-transparency [61]`	Transparency of an item. 0 is completely transparent, 1 is completely opaque
`-visiblelayers [67]`	The layers that are visible within this view (just for portals and surface, item #1)

Renderscripts

`border` **[13]**	Manipulate a fake 3D border for use in a render callback
`color` **[18]**	Manipulate a color for use in a render callback
`render` **[73]**	Configure and use renderer
`renderitem` **[74]**	Render an item in a render callback
`-renderscript [50]`	A script that gets evaluated every time an item is rendered
`-bb [9]`	A script that gets evaluated to specify the bounding box of an item

File I/O

`cache` **[14]**	Control item cache

`read` **[71]**	Read a .pad file
`write` **[98]**	Write a .pad file (all the items on a widget)

`-file` [22]	File an item should be defined by
`-writeformat` [69]	Controls whether disk-based item is written out by copy or reference

Miscellaneous

`configure` **[19]**	Modify the pad widget
`font` **[33]**	Manipulate fonts and the font path
`html` **[52]**	Manipulate and query an html page and its anchors.
`image` **[53]**	Manipulate images
`info` **[54]**	Get type-specific information about an item
`layer` **[57]**	Manipulates layers
`random` **[70]**	Generates a random integer
`setlanguage` **[84]**	Set the language to be used for future callback scripts
`settoplevel` **[87]**	Set the language to be used by the top-level interpreter
`sound` **[89]**	Manipulate and play sounds
`windowshape` **[97]**	Modify the shape of the top-level window that a pad widget is in

`-donescript` [17]	A script to evaluate when a background action has completed
`-errorscript` [19]	A script to evaluate when a background action has an error
`-reference` [48]	What item an alias references
`-updatescript` [62]	A script to evaluate when a background action has made progress

Utilities

`clock` **[17]**	Create a clock to measure elapsed milliseconds
`getdate` **[37]**	Get the current date in unix format
`getpads` **[42]**	Get a list of all pad widgets currently defined
`line2spline` **[59]**	Generate points for a spline that approximate a line
`noise` **[63]**	Generate 'perlin' noise
`padxy` **[64]**	Convert a window point (x, y) to pad coordinates
`spline2line` **[90]**	Generate points for a line that approximate a spline
`urlfetch` **[95]**	Retrieve a URL over the internet in the background
`warp` **[96]**	Warp (move) the core pointer

Widgets

`-command` [14]	Callback for widgets
`-editable` [18]	True if text item is editable
`-from` [25]	Starting value of valuator widget
`-linesize` [34]	Amount widget should change to represent a line change
`-memberlabels` [39]	List of labels for a pull-down or pop-up menu
`-menubar` [40]	Menubar associated with a frame
`-orientation` [43]	Orientation of widget (horizontal or vertical.)
`-pagesize` [44]	Amount widget should change to represent a page change
`-to` [60]	Ending value of valuator widget
`-value` [64]	Current value of valuator widget

Debugging

`printtree` **[67]**	Print all the items on the pad surface in their internal tree structure

Extensions

`addoption` **[4]**	Create a new option for an existing type
`addtype` **[6]**	Create a new item type

Item Types

- **Item Options**
- **Alias Items**
- **Button Items**
- **Canvas Items**
- **Checkbox Items**
- **Checkboxmenuitem Items**
- **Choicemenu Items**
- **Frame Items**
- **Grid Items**
- **Group Items**
- **HTML Items**
- **Image Items**
- **KPL Items**
- **Label Items**
- **Line Items**
- **Menu Items**
- **Menubar Items**
- **Menuitem Items**
- **Menu Items**
- **Pad Items**
- **Panel Items**
- **Polygon Items**
- **Portal Items**
- **Rectangle Items**
- **Scrollbar Items**
- **Spline Items**
- **TCL Items**
- **Text Items**
- **Text items have default event bindings which can be used for emacs-style editing of them. See the section on Default Bindings for more info.**
- **Textfield Items**
- **Window Items**

Executables

When Pad++ is built and installed correctly, there are two executable files that may be run. *padwish* runs a version of the Tcl interpreter extended with the pad widget. This is a complete superset of the standard Tk wish program. The **`pad`** command is the sole addition which is described below. In addition, the Pad++ distribution comes with an application written entirely in Tcl called PadDraw. This application is a general-purpose drawing and demo program that shows many capabilities of the pad widget. There are two scripts which can be used to run Pad++. '*pad*' is a script which sets the appropriate environment variables and runs padwish, giving a Tcl prompt. '*paddraw*' is started by running the *paddraw* script which automatically runs *padwish* and starts the Tcl PadDraw program. When running PadDraw by executing *paddraw*, the Tcl interpreter is not available.

Padwish Synopsis

```
padwish [options] [arg arg ...]
```

Valid options are:

`-colormap colormap`
: Specifies the colormap that padwish should use. If colormap is "new", then a private colormap is allocated for padwish, so images will look nicer (although on some systems you get a distracting flash when you move the pointer in and out of a PadDraw window and the global colormap is updated).

`-display display`
: Display (and screen) on which to display window.

`-geometry geometry`
: Initial geometry to use for window.

`-help`
: Print a summary of the command-line options and exit.

`-language`
: Specifies what scripting language the top-level interpreter should use. Pad++ always supports Tcl, but can be compiled to use the Elk version of Scheme also. In addition, Pad++ provides a mechanism to support other interpreted scripting languages as well. Defaults to 'tcl'.

`-name name`
: Use name as the title to be displayed in the window, and as the name of the interpreter for send commands.

`-norgb`
: Force all images to be loaded without RGB data. This means that images will be stored with one byte per pixel instead of the normal 5 bytes per pixel. As a result, images will not be able to be dithered.

`-sharedmemory`
: Specifies if Pad++ should try and use X shared memory. Some machines (notably a particular Solaris 5.4 machine) crashes and the X server dies when Pad++ is used with shared memory, so it can be disabled if there is trouble. Defaults to 1 (true).

`-sync`
: Execute all X server commands synchronously, so that errors are reported immediately. This will result in much slower execution, but it is useful for debugging.

`-visual visual`
: Specifies the visual type that padwish should use. The valid visuals depend on the X server you are running on. Some common useful ones are "truecolor 24" and "truecolor 12", which specify 24 bit and 12 bit mode, respectively.

`--`
: Pass all remaining arguments through to the script's argv variable without interpreting them. This provides a mechanism for passing arguments such as -name to a script instead of having padwish interpret them.

TCL Synopsis

```
pad [pathName [options]]
```

The **pad** command creates a new window (given by the pathName argument) and makes it into a Pad++ widget. If no pathName is specified, a unique top-level window name will be generated. Additional options may be specified on the command line or in the option database to configure aspects of the Pad++. The **pad** command returns the name of the created window. At the time this command is invoked, there must not exist a window named pathName, but pathName's parent must exist.

Once a Pad++ widget is created, there are five ways of writing Tcl code for it. They are:

- **Configuring the widget:** Each widget has several configuration options that control the widget as a whole. For example, `-width` and `-height` control the geometry of the widget.
- **Executing widget commands:** There are many commands associated with the widget. They are actually sub-commands of the primary widget command. When a new pad widget is created, a command is also created whose name is the name of the widget. For instance, evaluating `pad .pad` creates a widget named `.pad`, and a command named `.pad`. For example, to find out what the current view on the pad widget is, use the **getview** command with: `.pad getview`.
- **Creating items on the widget:** Each pad widget can contain many graphical items, such as lines, text, etc. These are all created with the **create** sub-command. For example, `.pad create line 0 0 10 10` creates a line from the origin to the point (10, 10).
- **Configuring those items:** Once items have been created, they can be modified with the **itemconfigure** sub-command. For example, supposing that the previous line had an id of 2, we could change its pen color and width with: `.pad itemconfigure 2 -pen red -penwidth 5`
- **Accessing global Pad variables:** The pad widget declares certain global Tcl variables that can be used by applications. For example, to see the current version of Pad++, examine the *Pad_Version* variable.

This version of Pad++ works with either Tcl7.5/Tk4.1 or Tcl7.6/Tk4.2.

Note that in this reference manual, optional parameters are listed in square brackets, [...]. While this is traditional for reference documentation, the Tcl/Tk documentation uses ?...? to denote optional parameters in order to avoid confusion with the meaning of [...] in the Tcl language. We decided to risk the confusion with Tcl for the increased clarity of square brackets.

Widget-Specific Options

Name: **background**
Class: **Background**
Command-Line Switch: **-background**

Specifies the normal background color to use when displaying the widget.

Example:
```
.pad config -background gray50
```

Name: **closeEnough**
Class: **CloseEnough**
Command-Line Switch: **-closeEnough**

Specifies a floating-point value indicating how close the mouse cursor must be to an item before it is considered to be "on" the item. Defaults to 3.0.

Name: **cursor**
Class: **Cursor**
Command-Line Switch: **-cursor**

Specifies the mouse cursor to be used for the widget. The value may have any of the forms acceptable to Tk_GetCursor.

Name: **debugBB**
Class: **DebugBB**
Command-Line Switch: **-debugBB**

Turns on and off display of bounding boxes. Default is 0.

Name: **debugEvent**
Class: **DebugEvent**
Command-Line Switch: **-debugEvent**

Turns on and off debugging of events. Default is 0. When event debugging is turned on, pad outputs a description of event handlers as they fire. In addition, if a break or event in a handler stops some events from firing, those events not fired are shown. By default, the event debugging output goes to stdout, however, it can be sent to a Tcl variable with the *-debugOut* configure option. Also note that PadDraw comes with a graphical interface that creates a GUI for seeing and examining events as they fire. This graphical event debugger can be used in other pad applications. See *draw/debugevent.tcl*.

Name: **debugGen**
Class: **DebugGen**
Command-Line Switch: **-debugGen**

Turns on and off general debugging. Default is 0.

Name: **debugOut**
Class: **DebugOut**
Command-Line Switch: **-debugOut**

Controls where debug output goes. By default, debug output is sent to stdout. However, the *-debugOut* configure option can specify a Tcl variable that all debug output will be appended to. It is then possible to set a Tcl trace on that variable to be notified whenever debug output is available. Currently, only *-debugEvent* uses the *-debugOut* variable.

Example: Evaluating "`.pad config -debugOut foo`" will cause all future debug output to be appended to the Tcl variable '`foo`'.

Name: **debugRegion**
Class: **DebugRegion**
Command-Line Switch: **-debugRegion**

Turns on and off visual display of portion of the screen that actually gets re-rendered. Used to debug region management. Default is 0.

Name: **defaultEventHandlers**
Class: **DefaultEventHandlers**
Command-Line Switch: **-defaultEventHandlers**

Turns on and off the default navigation event handlers. The default handlers are very simple. They allow basic panning with mouse button #1, zooming in with button #2, and zooming out with button #3. Default is 0.

Name: **`defaultRenderLevel`**
Class: **`DefaultRenderLevel`**
Command-Line Switch: **`-defaultRenderLevel`**

Specifies the default render level to use to display the Pad if no specific level is specified. The render level is generally used for efficiency where render level 0 is the fastest and least pretty way to render the pad (text is uglier, smaller items are not rendered, some items are rendered at a lower resolution). As the render level goes higher, the pad is rendered slower and prettier

Name: **`desiredFrameRate`**
Class: **`DesiredFrameRate`**
Command-Line Switch: **`-desiredFrameRate`**

Specifies the desired frame rate (in frames per second). This number is used by the Pad++ rendering engine to decide how to render the scene while animating. If a high frame rate is requested, small objects may not be rendered (see `-alwaysrender`) flag, and some objects may be rendered at low resolution. The default is 20 frames/second.

Name: **`dissolveSpeed`**
Class: **`DissolveSpeed`**
Command-Line Switch: **`-dissolveSpeed`**

Specifies how quickly dissolves should occur upon refinement. When the pad widget refines, it uses a dissolve effect instead of a simple buffer swap. The dissolve is controlled by *-dissolveSpeed*. This option may vary between 0 and 3 where 0 is a simple buffer swap, 1 is a fast dissolve, and 3 is the slowest dissolve. The default is 2.

Name: **`doubleBuffer`**
Class: **`DoubleBuffer`**
Command-Line Switch: **`-doubleBuffer`**

Specifies if the system should use double buffering for rendering. If doubleBuffer is set to 0 (off), rendering will be a little faster, but the screen will flash quite a bit. Mostly useful for debugging. Default is 1.

Name: **`fastPan`**
Class: **`FastPan`**
Command-Line Switch: **`-fastPan`**

Pad++ normally does fast pans, i.e., copying the portion of the screen that doesn't change, and re-rendering the new portion. This results in an approximation which can make the view be off by up to a half of a pixel. Fast panning can be disabled by setting this flag to 0 which results in slower but more accurate pans. Default is 1.

Name: **`fontCacheSize`**
Class: **`fontCacheSize`**
Command-Line Switch: **`-fontCacheSize`**

Pad++ employs a simple caching mechanism when drawing text in Type1 fonts. The caching mechanism remembers what size, font and bitmap it used when it last drew a particular character, and if that character is drawn again at the same size and font, Pad++ reuses the last bitmap image for that character rather than generating the bitmap for the character from its outline description. This greatly increases the speed of rendering large quantities of text.

You can configure the caching mechanism using the *-fontCacheSize* option. The font cache size is measured in Kilobytes (rounded to the nearest 100K). Setting *-fontCacheSize* to 0 turns off font caching, and characters are always drawn from their outline descriptions. The default value is 100 which produces significantly faster font rendering than using no font cache. Values above 100 have a lesser impact on performance, but may be effective for applications which use a lot of text with different fonts and sizes.

Name: **gamma**
Class: **Gamma**
Command-Line Switch: **-gamma**

Specifies 'gamma' used for allocating colors for images. This number controls how light or dark an image appears to be. Larger numbers will make images appear lighter. Default is 1.0.

Name: **height**
Class: **Height**
Command-Line Switch: **-height**

Specifies the height of the Pad in pixels. Defaults to 400.

Name: **heightmmofscreen**
Class: **HeightMMOfScreen**
Command-Line Switch: **-heightmmofscreen**

Specifies the height of the physical screen in millimeters. Normally, this information is given by the X server, but sometimes it is incorrect (for example, on some laptops). If it is incorrect, coordinates on the Pad++ surface will be incorrect. If this value is set to 0, the X server information will be used. Defaults to 0.

Name: **interruptible**
Class: **interruptible**
Command-Line Switch: **-interruptible**

If this flag is true (1), then animations and slow renders will be interrupted by events (mouse and keyboard). Defaults to true (1).

Name: **maxZoom**
Class: **MaxZoom**
Command-Line Switch: **-maxzoom**

This controls the maximum zoom (in and out) that any view is allowed. This way, it not possible to crash pad by zooming in or out too far. It defaults to 100,000,000 which gives 16 orders of magnitude of zooming (8 in and 8 out). Note that the amount one can zoom in is determined by the product of the (x, y) position and the zoom. So, while you can zoom into the position (0, 0, 100000000), you can only zoom into (1000, 1000, 100000). Setting *-maxzoom* to 0 disables the checking.

Name: **mediumObjSize**
Class: **MediumObjSize**
Command-Line Switch: **-mediumObjSize**

Pad++ tries to keep up the display rate, even when the scene gets complicated. If the system becomes slower than the requested frame rate, it both stops drawing small objects, and it draws medium-sized objects in a very ugly fashion. This option configures the size below which objects are considered to be medium-sized. Default is 100 pixels. (Also see **-smallObjSize** configuration option.)

Name: **refinementDelay**
Class: **RefinementDelay**
Command-Line Switch: **-refinementDelay**

Specifies the delay in milliseconds after the last X event to start refinement. Default is 1000.

Name: **smallObjSize**
Class: **SmallObjSize**
Command-Line Switch: **-smallObjSize**

Pad++ tries to keep up the display rate, even when the scene gets complicated. If the system becomes slower than the requested frame rate, it both stops drawing small objects, and it draws medium-sized objects in a very ugly fashion. This option configures the size below which objects are considered to be small-sized. Default is 10 pixels. (Also see **-mediumObjSize** configuration option.)

Name: **sync**
Class: **Sync**
Command-Line Switch: **-sync**

Specifies if X event synchronization should be turned on. When it is on, the X server executes every command as it is executed rather than caching them and executing commands in groups. Generally useful just for debugging. Default is 0.

Name: **units**
Class: **Units**
Command-Line Switch: **-units**

Specifies unit dimensions for all coordinates used by Pad++. It can be any of "points", "mm", "inches", or "pixels". Default is points.

Name: **width**
Class: **Width**
Command-Line Switch: **-width**

Specifies the width of the Pad in pixels. Defaults to 400.

Name: **widthmmofscreen**
Class: **WidthMMOfScreen**
Command-Line Switch: -**widthmmofscreen**

Specifies the width of the physical screen in millimeters. Normally, this information is given by the X server, but sometimes it is incorrect (for example, some laptops). If it is incorrect, coordinates on the Pad++ surface will be incorrect. If this value is set to 0, the X server information will be used. Defaults to 0.

Widget Commands

The **pad** command creates a new Tcl command whose name is `pathName`. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option [arg arg ...]

Option and the *args* determine the exact behavior of the command. The following widget commands are possible for Pad++ widgets:

[1] `pathName` **`animate`** `subcommand [args ...]`

The **`animate`** command is the key to a sophisticated animation engine that allows asynchronous animations of most options of items on the Pad++ surface. An item can be moved across the screen while its color is being changed while another is being rotated. This all happens in the background so that Pad++ continues to process events while animations happen.

The basic units of an animation are:

- *path*: Defines the values visited by the changing option
- *channel*: Associates an object with a path and the property to animate
- *animation*:Groups channels and animations as a single unit

A very simple example that creates and then zooms some text follows:

```
set txt [.pad create text -text "Hello World" -pen yellow \
   -pos {0 -50 0.2} -anchor center]
set txtpath [.pad anim create path -path {{0 -50 0.2} {0 -50 5}} \
   -endtime 2]
set txtchannel [.pad anim create channel -object $txt \
   -path $txtpath -option -pos]
.pad anim start $txtchannel
```

The animate command contains several subcommands. Briefly, they are:

- *create*: Create an animation unit
- *configure*: Configure an animation unit
- *delete*: Delete an animation unit
- *start*: Begin play of channel or animation
- *interrupt*: Stop channel or animation before it is complete
- *getvalue*: Get interpolated value from path

The animate subcommands in more detail are:

pathName anim **`create`** AnimationUnit [option value ...]

This creates one of the basic animation units (paths, channels, and animations.) When one creates an animation unit, a unique token for that unit is returned. Use the returned token to refer to that unit for future configuration/manipulations.

Example:

```
set path [.pad anim create path -path {0 1}]
set channel [.pad anim create channel -path $path]
.pad anim configure $channel -endtime 3
```

Following are all of the options that can be specified for each AnimationUnit:

`Path` configuration options:

`-path` [path]:

path is the set of "points" that defines the values to be visited by the curve. The "points" define a "polyline" in one, two, or three dimensions.

Example:

```
.pad anim config -path {1.0 2.0 3.0}
.pad anim config -path {{1.0 2.0} {3.0 4.0} {5.0 6.0}}
.pad anim config -path {{1.0 2.0 3.0} {4.0 5.0 6.0} \
   {7.0 8.0 9.0}}
```

`-timepath` [timepath]:

Timepath is a set of time value pairs that define both the values and the time that each value should be reached. This allows one to more exactly specify the the timing of the animation and to produce animations that do not operate at a constant speed. One use for such animations is when one wants to simulate a physics-like animation by calculating position of an object at specific times to define an animation. If a path is specified with `-endtime` instead of `-endtime`, the first value of each data set is treated as a time. Times must increase in value from one data set to the next. Each data set must contain at least two values, a time value and one, two, or three values to specify the configuration values defining the path.

Examples:

```
.pad anim config -timepath {{1.0 2.0} {3.0 4.0} {5.0 6.0}}
.pad anim config -timepath {{1.0 2.0 3.0} {4.0 5.0 6.0} \
   {7.0 8.0 9.0}}
```

(See bouncing ball example below.)

`-begintime` [timeInSec]

Time, in seconds, that defines when the first value of the path is obtained

`-endtime` [timeInSec]:

Time, in seconds, that defines when the last value of the path is obtained

`-intime` [inTime]:

One may want to have the animation start somewhere other than at the first value of the path. This can be accomplished be specifying the `-endtime` which must lie between `-endtime` and `-endtime` (inclusive). In combination with `-endtime`, a slice of an animation can be specified.

`-order` order

The number of parameters per entry in the path. **Order** may be 1, 2, or 3, and must match the order of the option the path is used with.

`-outtime` [inTime]:

One may want to have the animation end somewhere other than at the last value of the path. This can be accomplished be specifying the `-endtime` which must lie between `-endtime` and `-endtime` (inclusive). In combination with `-endtime`, a slice of an animation can be specified.

`-post` [postCondition]

This specifies what happens when the current time passes *-endtime*. The possible values for this option are: constant, cycle, or oscillate

- *constant*: If the current time is after `-endtime`, the last value of the path is applied.
- *cycle*: If the current time passes `-endtime`, the interpolated value is projected back to the first value in the path, and the path is cycled through from the beginning.
- *oscillate*: If the current time passes `-endtime`, the interpolated value is reflected from the final value in the path, toward the first value.

-pre [preCondition]:

This specifies what happens when the current time is before `-endtime`. The possible values for this option are: constant, cycle, or oscillate

- constant: If the current time is before `-endtime`, the first value of the path is applied.
- cycle: If the current time is just before `-endtime`, the interpolated value is projected to the last value in the path, and the path is cycled through from the end toward the beginning.
- oscillate: If the current time is before `-endtime`, the interpolated value is reflected back from the first value in the path, toward the final value.

-siso [boolean]

Indicates whether to apply a slow-in-slow-out effect to the animation. Default value is "0" (off).

Example:

```
set rec [.pad create rectangle 0 0 100 100 -fill blue]
set recp [.pad anim create path -path {{-100 0 1} {100 0 1}} \
   -endtime 2 -siso 1]
set recc [.pad anim create chan -path $recp -object $rec \
   -option -pos]
.pad anim start $recc
```

Channel configuration options:

-path [pathToken]

pathToken specifies the animation path to be applied to the channel's option

-object [tagOrId]

tagOrId specifies the object/objects that are to be affected by the channel

-option [option]

Option specifies the item configuration option that will be animated by interpolating along the animation path. The channel's path must be of the same order as the option. This means that if the option to be animated is `-endtime`, the path is a single list of values (i.e. -path {1.0 2.0 4.0 8.0}), if the option is `-endtime`, the path should be a list of lists containing three values each (i.e. -path {{1.0 3.0 4.0} {2.0 1.0 7.0} {5.0 8.0 10.0}}).

Presently supported options that one may want to animate:

Order 1:

- -angle
- -penwidth
- -transparency

Order 3:

- -position (min. abbreviation -pos)
- -rposition (min. abbreviation -rpos)
- -fillcolor (rgb values from 0 to 255)
- -pencolor (rgb values from 0 to 255)
- -view (applies only to pad surfaces)
- -anglectr (angle Xcenter Ycenter)

`-begintime` [timeInSec]

Sets the `-endtime` of the *path* associated with the channel. This is just a convenience. Beware that if you set the `-endtime` of a channel, all channels using this same path are affected. (See `-endtime` for paths, above)

`-endtime` [timeInSec]

Sets the `-endtime` of the *path* associate with the channel. This is just a convenience. Beware that if you set the `-endtime` of a channel, all channels using this same path are affected. (See `-endtime` for paths, above)

`Animation` configuration options:

`-members` [listOfChannelsAndAnimations]:

Used to add animatables to an animation. Both channels and animations are animatable and can be a member of an animation. An animation cannot be a member of itself.

Example:

```
.pad anim config anim0 -members "chan0 chan1 anim1"
```

`-begintime` [timeInSec]:

If an animation (anim0) is a member of another animation (anim1), `-endtime` specifies the delay time after anim1 is started, that anim0 should be started

`-endtime` [timeInSec]

By default, this is the amount of time for all animatables to finish their animation. If set to a value less than the default, all animatables will stop at the parent animation `-endtime`. If set to a value greater than the default, there is no noticeable effect.

`-speedfactor` [speedFactor]

By default this is 1.0. If one sets it to 2, the animation will be played twice as fast etc.

pathName anim **`configure`** AnimationUnit [option value ...]

One can configure an animation unit by using its configuration options. If a configuration option is entered without specifying a value to set the option, the current value of the option is returned. The options that can be configured on any animation unit are the same that apply with the **`create`** command.

Example:

```
.pad anim config path0 -endtime 10
```

pathName anim **`delete`** AnimationUnit

Deletes an AnimationUnit.

pathName anim **start** AnimationUnit

Begins the playing of an animation or of a channel.

pathName anim **interrupt** AnimationUnit

Stops the playing of an animation or of a channel, before play has completed.

pathName anim **getinterpval** timeInSec

Returns the the interpolated value along the path given the the time in seconds. This may be useful when one wants to use a path for something other than animations, or just to check values along an animation path. This command applies only to animation paths.

Notes concerning animations:

Commands and options can be abbreviated with any string that uniquely identifies the command or option of interest.

When one changes the `-endtime` or `-begintime` of a channel, it is the path that is associated with the channel that is actually affected. Be careful that the affected path is not also being used in another channel that needs a different `-endtime` and `-begintime`. If it is, make another path with the same data and the desired `-endtime` and `-begintime`.

When directly playing channels via a command such as '`.pad anim start channelToken`', each channel has its own timer. So if you have several channels playing in overlapping time, you have several timers going. If you place several channels into a parent animation, when you play that animation all the channels are played using one timer. If you place animations within a parent animation, when the child animations are playing, each uses its own timer.

There is a conflict between angles, groups, and animations. For example, if we have a group (10) with items 5 and 6 in it, and we make an animation that changes that relative position of item 5 and tries to rotate the group at the same time, there is a problem. `-rposition` does not account for the -angle of the group. So the orientation of the motion of item 5 is not rotated with the group.

Example animations:

```
########################################################################
# IMPROVED HELLO WORLD ANIMATION
# Here is an animation to illustrate the combining of
# channels and animations into a single animation.
########################################################################

# Set up hello world channel
set txt [.pad create text -text "Hello World" -pen yellow \
   -pos {0 -50 0.2} -anchor center]
set txtpath [.pad anim create path -path {{0 -50 0.2} {0 -50 5}} \
   -endtime 2]
set txtchannel [.pad anim create channel -object $txt -path $txtpath \
   -option -pos]

# Make four rectangles
for {set i 0} {$i<4} {incr i} {
```

```
    .pad create rectangle 0 0 100 100 -fill black -pos {0 50 1} -tags rect$i
}

# Make a two paths, one first order and one third order
set p0 [.pad anim create path -path {0 180 90}]
set p1 [.pad anim create path -path {{0 0 0} {255 0 0} {0 255 0} \
    {0 0 255} {0 0 0}} -post cycle]

# Make two channels for each object.
# One channel for changing -angle
# the other for changing -fill

set j 0
for {set i 0} {$i<8} {incr i 2} {
    set obj [.pad find withtag rect$i]
    set c$i [.pad anim create channel -object $obj -option -angle \
        -path $p0 -begintime 0 -endtime 6]
    set c[expr $i+1] [.pad anim create channel -object $obj -option -fill  \
        -path $p1 -begintime 3 -endtime 12]
    incr j
}

# Make three animations containing only channels
# and one animation containing channels and the
# other animations
set a0 [.pad anim create anim -members "$c0 $c1" -endtime 10 \
    -begintime 2.5]
set a1 [.pad anim create anim -members "$c2 $c3" -endtime 10 \
    -begintime 5.0]
set a2 [.pad anim create anim -members "$c4 $c5" -endtime 10 \
    -begintime 7.5]
set a3 [.pad anim create anim -members "$txtchannel $c6 $c7 $a0 $a1 $a2" \
    -endtime 10 -begintime 0]

.pad anim start $a3



#################################################
# BOUNCE:
# This example shows a bouncing ball animation.
# Kinematic equations are used to calculate the
# path for a bouncing ball that looses energy.
# A -timepath is created and applied to an oval
# in a rectangular box.
#################################################

.pad moveto 0 500 0.2

set box     [.pad create rectangle -165 -50 165 1400 -penwidth 20]
set ball    [.pad create oval 0 0 100 100 -fill blue -pos "0 0 1"]
set boxBall [.pad create group -members "$ball $box"]
```

```
set t  0.0
set a  -98.0
set v0 500.0
set x0 0.0
set tpath ""
set delT 0.01
set coefRes 0.90

set endt [expr 2.0*$v0/$a]
set endit [expr abs(int($endt/$delT))]
set refTime 0.0

for {set j 1} {$j < 20} {incr j 1} {
   for {set i 0} {$i <= $endit} {incr i 1} {
      set t [expr $i*$delT ]
      lappend tpath "[expr $refTime + $t] 0 [expr ((0.5)*$a*$t*$t \
         + $v0*$t + $x0)] 1"
   }

   set v0 [expr ($v0*pow($coefRes,$j))]
   set refTime [expr $refTime + $t]
   set endt [expr 2.0*$v0/$a]
   set endit [expr abs(int($endt/$delT))]
}

set ballpath [.pad anim create path -timepath $tpath \
   -endtime [expr $refTime + $t] -intime 5]
set ballchan [.pad anim create channel -object $ball -path $ballpath \
   -option -rpos]
set ballanim [.pad anim create animation -members $ballchan]

# run the animation with:

.pad anim start $ballanim



####################################################
# USING A POLYLINE TO DEFINE A PATH:
# Here is an example of using a polyline to define
# a -path (a -timepath could be created by adding
# times in the "for" loop creating the pathlist).
####################################################

# To use the script, create a polyline (try one in the
# form of a big spiral), make sure it is selected,
# then enter the following code:

set coordlist [.pad coords [.pad find withtag selected]]
if {$coordlist == ""} {
   set coordlist {0 0 100 0 100 100 0 100 0 0}
}
set len [llength $coordlist]
set curscale [lindex [.pad getview] 2]
```

```
set pathlist ""
for {set i 0} {$i < $len} {incr i 2} {
  lappend pathlist "[lindex $coordlist $i] [lindex $coordlist \
     [expr $i + 1]] $curscale"
}

# Here is an example of using "pathlist" to
# create an animation to move a rectangle

set obj       [.pad create rectangle 0 0 50 50 -fill red]
set coordPath [.pad anim create path -path $pathlist]
set rectChan  [.pad anim create channel -path $coordPath -object $obj \
  -option -pos -endtime 10]
set myanim    [.pad anim create anim -members $rectChan]

.pad anim start $myanim



##############################################
# USING A TIMEPATH
# Here is an example of using the coordinates
# to make a -timepath from coordPath, enter:
# This assumes that the previous example has
# already been run.
##############################################

set timepathlist ""
set time 0.0
for {set i 0} {$i < $len} {incr i 2} {
  lappend timepathlist "[expr $time + log(int($i+1))] \
     [lindex $coordlist $i] [lindex $coordlist [expr $i + 1]] $curscale"
  set time [expr $time + (1.0/($i+1.0))]
}

set coordPath [.pad anim create path -timepath $timepathlist]
set rectChan  [.pad anim create channel -path $coordPath -object $obj \
  -option -pos]
set mytanim   [.pad anim create anim -members $rectChan]

.pad anim start $mytanim
```

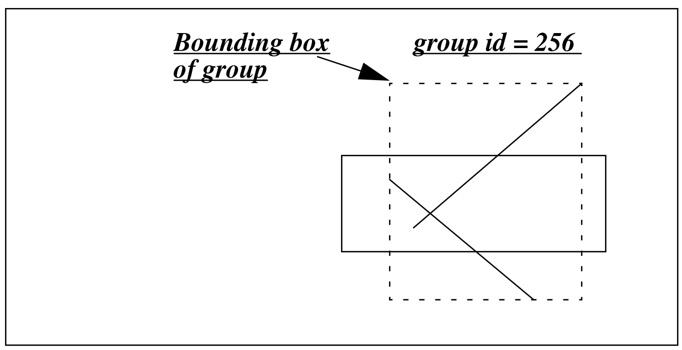
[2] `pathName` **`addgroupmember`** `[-notransform] tagOrId groupTagOrId`

Add all items specified by *tagOrId* to the group specified by *groupTagOrId*. If *groupTagOrId* specifies more than one item, the first one is used. The items are added to the end of the group in the order specified by *tagOrId*. Groups automatically update their bounding boxes to enclose all of their members. Thus, they will grow and shrink as their members change.

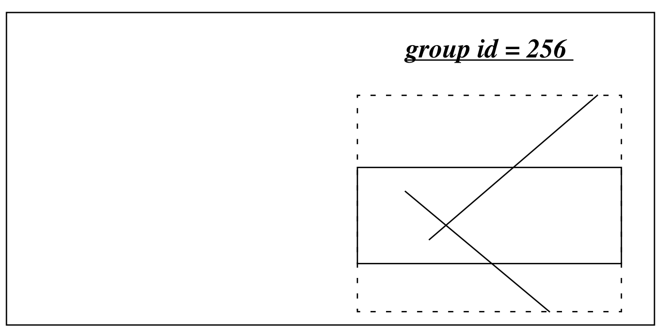
By default, items are transformed so they don't change their location when added to a group, even if the group has a transformation. This is implemented by transforming the item's transformation to be the inverse of the group's transformation. If the *-notransform* flag is specified, this inverse transformation is not applied, and the item will move by the group's transformation when added. (Also see the **removegroupmember**, and **getgroup** commands). Returns an empty string.

Example :

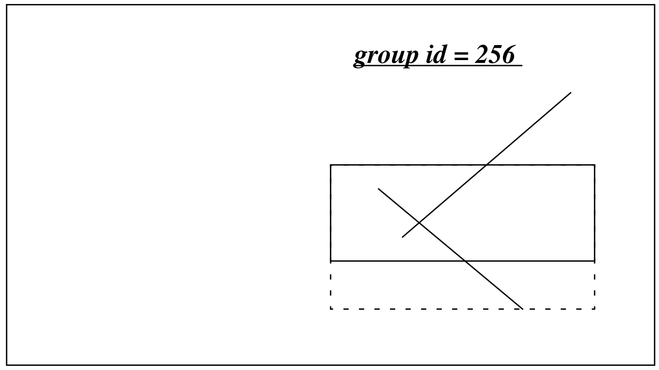
```
set id0 [.pad create line 0 0 100 100]
254
set id1 [.pad create line -10 20 80 -60]
255
set gid [.pad create group -members "$id0 $id1"]
256

.pad ic $gid -members
254 255

set id3 [.pad create rectangle -20 -20 130 40]
266
```




```
.pad addgroupmember $id3 $gid
.pad ic $gid -members
254 255 266
```




```
.pad removegroupmember $id0 $gid
.pad ic $gid -members
255 266
```



.pad getgroup $id2
256

[3] `pathName` **`addmodifier`** `modifier`

WARNING: **`addmodifier`** is an obsolete command and will be removed in the next release. Replace all uses of **`addmodifier`** with the '**`modifier add`**' command.

[4] `pathName` **`addoption`** `[-nowrite] typename optionname optionscript default`

Add a new option (named `optionname`) to all objects of type `typename`. `typename` must either be a built-in type, a user-defined type previously defined by **`addtype`**, or the special word "all" which means that this option applies to all types. When `optionscript` is called, the following arguments will be added on to the end of the script:

pathName:	The name of the pad widget the item is on
item:	The id of the item being configured
[value]:	Optional value. If value is specified, then the option must be set to this value.

`optionscript` must return the current (or new) value of the option. `default` specifies the default value of this option. This is used to determine if the option should be written out when the **`write`** command is executed. Note that the option will only be written out if the value is different than the default. If `-nowrite` is specified, then this option won't be written out. See the section APPLICATION-DEFINED ITEM TYPES AND OPTIONS in the Programmer's Guide for more information. (Also see the **`addtype`** command.)

[5] `pathName` **`addtag`** `tagToAdd tagOrId ...`

For each item specified by the list of tagOrIds, add *tagToAdd* to the list of tags associated with the item if it isn't already present on that list. It is possible that no items will be specified by tagOrId, in which case the command has no effect. This command returns an empty string.

This command is designed to be used in conjunction with the **find** command. Notice the necessity of using **eval** in this example: `eval .pad addtag foo [.pad find withtag bar]`

[6] `pathName` **`addtype`** `typename createscript`

Add `typename` to the list of allowed user defined types. When a new object of type `typename` is created, the `createscript` will be evaluated, and it must return an object id. When `createscript` is evaluated, the pad widget the object is being created on will be added on as an extra argument, followed by any parameters before the options. See the section APPLICATION-DEFINED ITEM TYPES AND OPTIONS in the Programmer's Guide for more information. (Also see the **`addoption`** command.)

[7] `pathName` **`allocborder`** `color`

WARNING: **`allocborder`** is an obsolete command and will be removed in the next release. Replace all uses of **`allocborder`** with the '**`border alloc`**' command.

[8] `pathName` **`alloccolor`** `color`

WARNING: **`alloccolor`** is an obsolete command and will be removed in the next release. Replace all uses of **`alloccolor`** with the '**`color alloc`**' command.

[9] `pathName` **`allocimage`** `file [-norgb]`

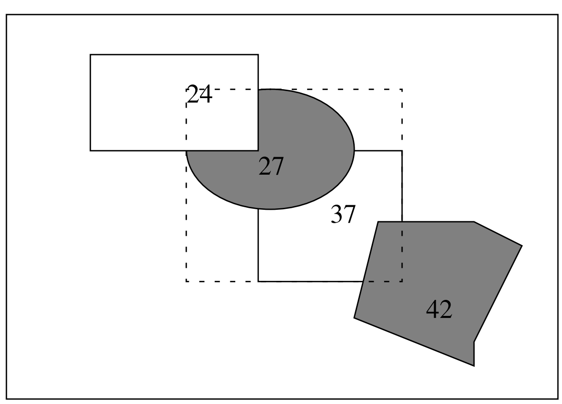
WARNING: **`allocimage`** is an obsolete command and will be removed in the next release. Replace all uses of **`allocimage`** with the '**`image alloc`**' command.

[10] `pathName` **`bbox`** `[-sticky] tagOrId [tagOrId tagOrId ...]`

Returns a list with four elements giving the bounding box for all the items named by the *tagOrId* argument(s). The list has the form "x_1 y_1 x_2 y_2" such that the drawn areas of all the named elements are within the region bounded by x_1 on the left, x_2 on the right, y_1 on the bottom, and y_2 on the top. If *-sticky* is specified, then the bounding box of the item in sticky coordinates, that is, the coordinates of a sticky item that would appear at the same location on the screen is returned. If no items match any of the *tagOrId* arguments then an empty string is returned.

If the item is sticky then **`bbox`** returns the bounding box of the item as it appears for the current view. That is, the bounding box will be different when the view is different. If *-sticky* is specified, then the bounding box returned is independent of the current view (i.e., it returns the bounding box as if the view was "0 0 1").

If the item is the Pad++ surface (item #1), then **`bbox`** will refer to the bounding box of the portion of the surface that is currently visible (based on the view and window size).



.pad bbox 27 37
-75 -55 68 79

[11] `pathName` **`bind`** `tagOrId [sequence [command]]`

This command associates *command* with all the items given by *tagOrId* such that whenever the event sequence given by *sequence* occurs for one of the items the command will be invoked.

This widget command is similar to the Tk bind command except that it operates on items on a Pad++ widget rather than entire widgets. See the Tk bind manual entry for complete details on the syntax of *sequence* and the substitutions performed on *command* before invoking it. The Pad++ widget defines extensions described below, but it is implemented as a complete superset of the standard **`bind`** command. I.e., you can do everything you can with the canvas with exactly the same syntax, but you can also do more.

If all arguments are specified then a new binding is created, replacing any existing binding for the same *sequence* and *tagOrId* (if the first character of command is "+" then *command* augments an existing binding rather than replacing it). In this case the return value is an empty string. If both *command* and *sequence* are omitted then the command returns a list of all the sequences for which bindings have been defined for *tagOrId*.

The only events for which bindings may be specified are those related to the mouse and keyboard, such as *Enter*, *Leave*, *ButtonPress*, *Motion, ButtonRelease*, *KeyPress* and *KeyRelease*. In addition, Pad++ supports some extra bindings including: *Create, Modify, Delete, PortalIntercept*, and *Write*. The handling of events in Pad++ uses the current item defined in *Item IDs and Tags* in the Programmer's Guide. *Enter* and *Leave* events trigger for an item when it becomes the current item or ceases to be the current item; note that these events are different than *Enter* and *Leave* events for windows. Mouse-related events are directed to the current item, if any. Keyboard-related events are directed to the focus item, if any (see the **`focus`** command below for more on this).

It is possible for multiple bindings to match a particular event. This could occur, for example, if one binding is associated with the item's id and another is associated with one of the item's tags. When this occurs, all of the matching bindings are invoked. The order of firing is controlled by the pad **`bindtags`** command. The default is that a binding associated with the *all* tag is invoked first, followed by one binding for each of the item's tags (in order), followed by a binding associated with the item's id. If there are multiple matching bindings for a single tag, then only the most specific binding is invoked. A **`continue`** command in a binding script terminates that script, and a **`break`** command terminates that script and skips any remaining scripts for the event, just as for the **`bind`** command.

If bindings have been created for a **`pad`** window using the Tk bind command, then they are invoked in addition to bindings created for the pad's items using the **`bind`** widget command. The bindings for items will be invoked before any of the bindings for the window as a whole.

The Pad++ bind command is extended in three ways:

- Extra macro expansions are added
- New events are added: <Create>, <Modify>, <Delete>, <Write>, and <PortalIntercept>.
- User-specified modifiers are added

Extra macro expansions
When a command is invoked, several substitutions are made in the text of the command that describe the specific event that invoked the command. In addition to the substitutions that the Tk **`bind`** command makes, Pad++ makes a few more. As with the Tk bind command, all substitutions are made on two character sequences that start with '%'. The special Pad++ substitutions are:

%P: The pad widget that received the event. This is normally the same as %W, but could be different if the event goes through a portal onto a different pad widget.

- %O: The id of the specific item that received the event.
- %I: Information about this event. This has different meanings for different event types. For **`<Modify>`** events, it specifies the command that caused the modification. For **`<PortalIntercept>`** events, it specifies the name of the event type generating the PortalIntercept. Standard Tcl event names, such as ButtonPress or ButtonRelease are used. This can be used by PortalIntercept events to only let certain event types go through the portal. Note that only a single PortalIntercept event is generated for a Button, Motion, ButtonRelease sequence, so these three events can not be distinguished in this manner.
- %i: The X-coordinate of the event on the Pad++ surface. This is specified in the current units (i.e., pixels or inches) of the pad widget.
- %j: The Y-coordinate of the event on the Pad++ surface. This is specified in the current units (i.e., pixels or inches) of the pad widget.
- %z: Size of event in pad coordinates. This is dependent on the view. It effectively says how much the event is magnified. I.e., if the view is zoomed in by a factor of two, then this will have a value of two. It is also affected by portals that the event travels through.
- %U: The X-coordinate of the event in object coordinates. This means that the point will be transformed so that it is in the same coordinate system of the object (independent of the object's transformation as well as the current view). This is specified in the current units (i.e., pixels or inches) of the pad widget.
- %V: The Y-coordinate of the event in object coordinates. This means that the point will be transformed so that it is in the same coordinate system of the object (independent of the object's transformation as well as the current view). This is specified in the current units (i.e., pixels or inches) of the pad widget.
- %Z: Size of event in object coordinates. This is dependent on the view and the magnifications of the object.
- %l: The list of portal ids that the event passed through.
- %L: The list of pad surfaces of the portals the event passed through. This list corresponds to the list of portal ids from '%l'.

New Events
Several new events fire at special times, depending on the semantics of the event.

`<create>:` This event gets fired whenever new pad items are created. Because items that this is attached to don't have id's yet, it only makes sense to attach this event to a tag. Then this event gets fired immediately after any item of the relevant tag is created. Example:

```
.pad bind foo <Create> {puts "A foo was created, id=%O"}
.pad create rectangle 0 0 50 50 -tags "foo"
   => A foo was created, id=5
```

`<Modify>:` This event gets fired whenever an item is modified. Modification occurs whenever an item's configuration options are changed, and whenever the following commands are executed on an item: **`coords`**, **`itemconfigure`**, **`scale`**, **`slide`**, **`text,`** and **`moveto`** (on a portal) The %I macro specifies the command that caused the modification. Example:

```
.pad bind foo <Modify> {puts "A foo was modified, cmd=%I"}
.pad create rectangle 0 0 50 50 -tags "foo"
```

```
.pad itemconfigure foo -pen red
   => A foo was modified, cmd=itemconfigure
```

<Delete>: This event gets whenever an item is deleted. It is typically used to clean up application resources associated with the item that was deleted.

<Write>: This event fires whenever an item is written out with the pad **write** command. While Pad++ knows how to generate the Tcl code necessary to recreate itself, items are often part of an application with associated data structures, etc. When an item is written out, it is frequently necessary to write out these associated structures. Sometimes, the application may prefer to substitute its code for pad's. This event provides a mechanism to augment or replace (possibly with an empty string) the Tcl code written out to recreate a pad item.

Whatever string a **<Write>** event returns is appended on to the string pad uses to write out that object. In addition, the application may modify the special global Tcl variable, `Pad_Write` which controls whether the item will get written out. This defaults to 1 (true), but may be set to 0 (false) by the event binding. In addition, the **<Write>** event gets fired on the special tags "`preWrite`" and "`postWrite`" at the beginning and end of the file, respectively, to allow an application to write out code at the ends of the file. Example:

```
.pad bind preWrite <Write> {
        return "Stuff at the beginning of the file"
}
.pad bind postWrite <Write> {
        return "Stuff at the end of the file"
}
.pad bind foo <Write> {
        return "Stuff after foo objects"
}
.pad bind bar <Write> {
        set Pad_Write 0
        return "Stuff instead of bar objects"
}
                # This forces all objects with the "cat" tag
                # to have nothing written out. Notice that an
                # empty string must be returned, or "0", the
                # result of the set command, will be written out.
.pad bind cat <Write> {
        set Pad_Write 0
        return ""
}
                # This example also has nothing written out,
                # but in addition, no other event handlers
                # will fire (the object could have multiple
                # tags, each with <Write> event handlers).
.pad bind dog <Write> {
        Set Pad_Write 0
        break
}
```

<PortalIntercept>: This event gets fired just before an event passes through a portal. If the event handler executes the break command, then the event stops at the portal and does not pass through. Example:

```
                    # Events will not go through portals of type "foo"
.pad bind foo <PortalIntercept> {
        break
}
```

User-specified modifiers
Event handlers are defined by *sequences* as defined in the Tk **bind** reference pages. A sequence contains a list of *modifiers* which are direct mappings to hardware such as the shift key, control key, etc. Event handlers fire only for sequences with modifiers that are active, as defined by the hardware.

Pad++ allows user-defined modifiers where the user can control which one of the user-defined modifiers is active (if any). The advantage of modifiers is that many different sets of event bindings may be declared all at once - each with a different user-defined modifier. Then, the application may choose which set of event bindings is active by setting the active user-defined modifier. This situation comes up frequently with many graphical programs where there are modes, and the effect of interacting with the system depends on the current mode.

New modifiers must be declared before they can be used with the pad **addmodifier** command (and may be deleted if they are no longer needed with the pad **deletemodifier** command.) Then, the modifier can be used in the pad **bind** command just like a system defined modifier. There may be at most one active user-defined modifier per pad widget. The active user-defined modifier is set with the **setmodifier** command (and may be retrieved with the **getmodifier** command). The current modifier may be set to "" (the default) in which case no user-defined modifier is set. Example:

```
.pad addmodifier Create
.pad addmodifier Run
.pad bind all <Create-ButtonPress-1> {
        # Do stuff to create new objects
}
.pad bind all <Run-ButtonPress-1> {
        # Do stuff to interact with existing objects
}
                    # Now the system will be in "Create" mode
.pad setmodifier Create
...
                    # Now the system will be in "Run" mode
.pad setmodifier Run
```

[12] `pathName` **`bindtags`** `tagOrId [type]`

If *type* is specified, this command changes the ordering of event firings on all objects referred to by *tagOrId*. Since more than one event handler may fire for a given event, this controls what order they fire in. If type is "general", events fire most generally first. That is, a binding associated with the *all* tag is invoked first, followed by one binding for each of the item's tags (in order), followed by a binding associated with the item's id. (i.e., *all*, *tags*, *id*). If type is "specific", then events fire most specific first. That is, a binding associated with the item’s id is invoked first, followed by one binding for each of the item's tags (in order), followed by a binding associated with the *all* tag (i.e., *id*, *tags*, *all*).

If *tagOrId* is *pathName*, then it does not change the ordering of any objects, but controls the default ordering of objects created in the future.

The default event firing order for all objects is "general". This command returns the current event firing order for the first item specified by *tagOrId*.

[13] `pathName` **`border`** `subcommand arg ...`

This is the command for manipulating borders. There are several subcommands:

`border alloc <bordercolor>`

Allocates a border for future use by render callbacks. A border is a fake 3D border created by a slightly lighter and a slightly darker color than specified. *Color* may have any of the forms accepted by Tk_GetColor. This returns a bordertoken. (Also see the **`render`** command for an example of how to use a border).

`border free <bordertoken>`

Frees the *border* previously allocated by **`allocborder`**.

[14] `pathName` **`cache`** `subcommand arg ...`

`cache in tagOrId`

Forces the items specified by tagOrId to be cached in

`cache out tagOrId`

Forces the items specified by tagOrId to be cached out

`cache configure [option [value] ...]`

Configures the state of the cache manager. Option-value pairs may be specified as with the itemconfigure command, or if no options are specified, a list of all options and values are returned.

`-dir` `dir`

Specifies the directory to use for the cache. The actual directory will be <*dir*>/<pid> where pid is the process id of Pad++. It will be removed when the process exits. The cache should be on a local disk for reasonable I/O performance. It is not set by default and caching is disabled until the cache dir is explicitly set by the application.

`-size` `size`

Size is the total memory available to the cache manager before it starts to cache out objects. It defaults to two megabytes. Caching can be disabled by setting *size* to zero.

`-viewmultiple` `viewmultiple`

Viewmultiple specifies a multiple of the view area the cache manager should use when deciding object visibility for purposes of caching. Its default value is 2 (so objects visible within twice the view are not cache out candidates). Setting it to 1 will cause images to be potentially get cached out when not in the view.

`-delay` `delay`

Delay specifies the interval (in seconds) the cache manager should check and perform any actual cache outs. Its default value is 5 seconds. Setting it to 0 will cause immediate cache outs.

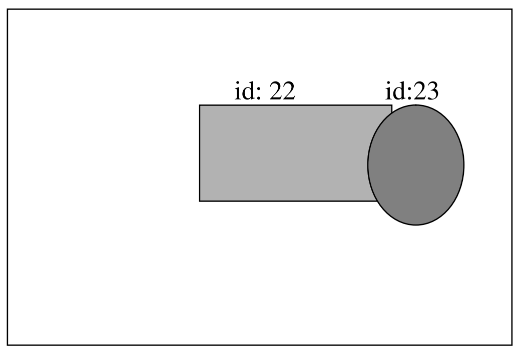
The following criteria are used for caching:

- When an object has to be rendered, the cache manager is requested to cache it in if necessary (ensures its data are in memory). Other objects may be cached out to make room for this object.
- When an object doesn't need to be rendered the cache manager marks it as a cache out candidate.

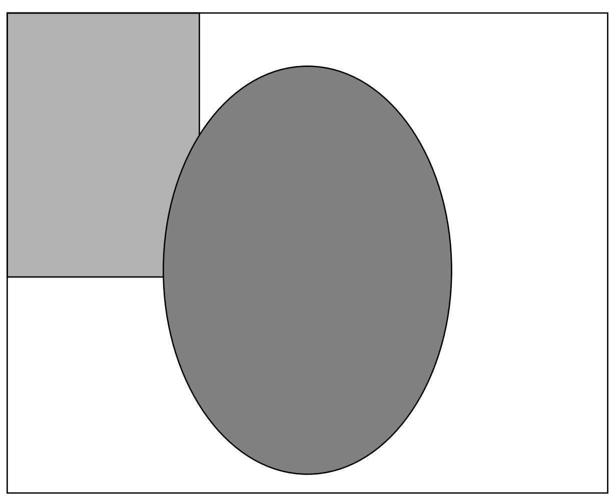
Cache out candidates are selected by a least-recently-rendered policy. The cache manager only selects objects that have been marked for cache out and does not attempt to select objects currently rendered (or visible within its multiple of the view area).

[15] `pathName` **`center`** `[-twostep] tagOrId [time x y [z [portalID ...]]]]`

Change the view so as to center the first of the specified items so the largest dimension of its bounding box fills the specified amount of screen (z). If *-twostep* is specified, then make the animation in two steps if appropriate (i.e., points not too close). The two steps are such that it zooms out to the midpoint between the two points far enough so that both start and endpoints are visible, and then zooms to the final destination. If *time* is specified, then make a smooth animation to the item in *time* milliseconds. The view is changed so that the item appears at the position determined by (x, y), both of which are in the range (0.0 ... 1.0). Here, 0.0 represents the left or bottom side of the window, and 1.0 represents the right or top side of the window. (x, y) specifies the portion of the item that should appear at the portion of the screen, relatively. So, specifying (0, 0) puts the lower left corner of the item on the lower left corner of the screen. (1, 1) puts the upper right corner of the item on the upper right corner of the screen. x and y default to (0.5, 0.5), i.e. the center of the screen. If a list of *portalID's* is specified, change the view within the last one specified.



.pad center 23



[16] `pathName` **`centerbbox`** `[-twostep]` x_1 y_1 x_2 y_2 `[time [x y [z [portalID`

`...]]]]`

Change the view so as to center the specified bounding box so that its largest dimension fills the specified amount of screen (`z`). If *-twostep* is specified, then make animation in two steps if appropriate (i.e., points not too close). The two steps are such that it zooms out to the midpoint between the two points far enough so that both start and endpoints are visible, and then zooms to the final destination. If *time* is specified, then make a smooth animation to the item in *time* milliseconds. The view is changed so that the bounding box appears at the position determined by (x, y), both of which are in the range (0.0 ... 1.0). Here, 0.0 represents the left or bottom side of the window, and 1.0 represents the right or top side of the window. (x, y) specifies the portion of the item that should appear at the portion of the screen, relatively. So, specifying (0, 0) puts the lower left corner of the bounding box on the lower left corner of the screen. (1, 1) puts the upper right corner of the bounding box on the upper right corner of the screen. x and y default to (0.5, 0.5), i.e. the center of the screen. If a list of *portalID's* is specified, change the view within the last one specified.

[17] `pathName` **`clock`** `[clockName [reset | delete]]`

Creates a clock that is set to 0 at the time of creation. Returns the name of the clock. Future calls with *clockName* return the number of milliseconds since the clock was created (or reset). Calls with *reset* specified reset the clock counter to 0, and return an empty string. Calls with *delete* specified delete the clock, and return an empty string.

.pad clock
clock1
.pad clock clock1
8125
.pad clock clock1 reset
.pad clock clock1
1825
.pad clock clock1 delete

[18] `pathName` **`color`** `subcommand arg ...`

This is the command for manipulating color. There are several subcommands:

`color alloc <file>`

Allocates a color for future use by render callbacks. *Color* may have any of the forms accepted by Tk_GetColor. This returns a colortoken. (Also see the **`render`** command).

`color free <colortoken>`

Frees the *color* previously allocated by **`alloccolor`**.

[19] `pathName` **`configure`** `[option] [value] [option value ...]`

Query or modify the configuration options of the widget. If no option is specified, returns a list describing all of the available options for pathName (see `Tk_ConfigureInfo` for information on the format of this list). If option is specified with no value, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no option is specified). If one or more option-value pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. Option may have any of the values accepted by the pad command. See the section on WIDGET-SPECIFIC OPTIONS for a description of all the options and their descriptions.

[20] `pathName` **`coords`** `[-objectcoords] [-append] [-nooutput] tagOrId` [x_0 y_0

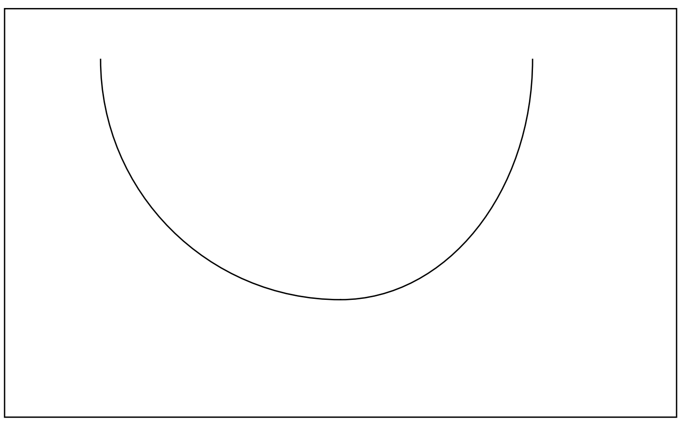
...]

Query or modify the coordinates that define an item. This command returns a list whose elements are the coordinates of the item named by *tagOrId*. If coordinates are specified, then they replace the current coordinates for the named item. If *tagOrId* refers to multiple items, then the first one in the display list is used. The flags may be specified in any order. Note that the **coords** command generates a **<Modify>** event on the items modified by it (see the **bind** command for a description of the **<Modify>** event). Locked items may not be modified by the coords command (see the *-lock* itemconfigure option). The **coords** command can only be used on line, rectangle, polygon and portal items.

If the flag *-objectcoords* is specified, then all coordinates are returned in the item's local coordinate system (i.e., as they were originally specified). If this flag is not specified, then all coordinates are returned in the global coordinate system (i.e., they are transformed by that item's translation and scale parameters).

If the flag *-append* is specified, then all the specified coordinates are appended on to the existing coordinates rather than replacing them.

If the flag *-nooutput* is specified, then this command returns an empty string. Typically, the *-append* and *-nooutput* flags are specified together when adding points to an item and time is of the essence.



```
set id [.pad create line -200 200]

for {set i -20} {$i <= 20} {incr i} {
   set x [expr $i * 10 ]
   set y [expr 0.5 * ($i * $i)]
   .pad coords -append -nooutput $id $x $y
}
```

[21] `pathName` **`create`** `type [option value ...]`

Create a new item in *pathName* of type *type*. The exact format of the arguments after type depends on *type*, but usually they consist of the coordinates for one or more points, followed by specifications for zero or more item options. See the **Overview of Item Types** section below for details on the syntax of this command. This command returns the id for the new item.

The available item types are: **Alias Items**, **Button Items**, **Frame Items**, **Grid Items**, **Group Items**, **HTML Items**, **Image Items**, **KPL Items**, **Label Items**, **Line Items**, **Menu Items**, **Pad Items**, **Panel**

Items, **Polygon Items**, **Portal Items**, **Rectangle Items**, **Scrollbar Items**, **Spline Items**, **TCL Items**, **Text Items**, **Text items have default event bindings which can be used for emacs-style editing of them. See the section on Default Bindings for more info.**, **Note that when the -width or -height of a textfile item is set, the textfile item is clipped to those dimensions rather than being squashed or stretched as most items are.**, and **Textfield Items**.

[22] `pathName` **`damage`** `[tagOrId]`

Indicates that some of the screen is damaged (needs to be redrawn). Damages the entire screen if *tagOrId* is not specified, or just the bounding box of each of the objects specified by *tagOrId*. The damage will be repaired as soon as the system is idle, or when the **`update`** procedure is called. Returns an empty string.

[23] `pathName` **`delete`** `tagOrId [tagOrId ...]`

Delete each of the items given by each *tagOrId*, and return an empty string. Note that the **`delete`** command generates a **`<Delete>`** event on the items modified by it (see the **`delete`** command for a description of the **`<Delete>`** event). Locked items may not be modified by the **`delete`** command (see the *-lock* itemconfigure option).

[24] `pathName` **`deletemodifier`** `modifier`

WARNING: **`deletemodifier`** is an obsolete command and will be removed in the next release. Replace all uses of **`deletemodifier`** with the '**`modifier delete`**' command.

[25] `pathName` **`deletetag`** `tagToDelete tagOrId [tagOrId ...]`
`dtag` is an alias for **`deletetag`**

For each item specified by the list of tagOrIds, delete *tagToDelete* from the list of tags associated with the item if it isn't already present on that list. It is possible that no items will be specified by tagOrId, in which case the command has no effect.

This command is designed to be used in conjunction with the **find** command. Notice the necessity of using **eval** in this example: `eval .pad deletetag foo [.pad find withtag bar]`

[26] `pathName` **`drawborder`** `border type width` x_1 y_1 x_2 y_2

WARNING: **`drawborder`** is an obsolete command and will be removed in the next release. Replace all uses of **`drawborder`**with the '**`render draw border`**' command.

[27] `pathName` **`drawimage`** `imagetoken x y`

WARNING: **`drawimage`** is an obsolete command and will be removed in the next release. Replace all uses of **`drawimage`** with the '**`render draw image'`** command.

[28] `pathName` **`drawline`** x_1 y_1 x_2 y_2 [x_n y_n ...]

WARNING: **`drawline`** is an obsolete command and will be removed in the next release. Replace all uses of **`drawline`** with the '**`render draw line`** ' command.

[29] `pathName` **`drawpolygon`** x_1 y_1 x_2 y_2 [x_n y_n ...]

WARNING: **`drawpolygon`** is an obsolete command and will be removed in the next release. Replace all uses of **`drawpolygon`** with the '**`render draw polygon`**' command.

[30] `pathName` **`drawtext`** `string xloc yloc`

WARNING: **`drawtext`** is an obsolete command and will be removed in the next release. Replace all uses of **`drawtext`** with the '**`render draw text`**' command.

[31] `pathName` **`find`** `[-groupmembers] [-regexp | -glob] searchCommand \`
`[arg arg ...] ["&&" | "||"] [searchCommand [arg arg ...]]`

This command returns a list consisting of all of the items that meet the constraints specified by the searchCommands and arg's. All found items are returned in display list order. Multiple searchCommands may be used as long as they are delimited by "&&" or "||". Parenthesis are allowed to group expressions. The following characters are reserved: '&', '|', '(', ')', and '!'. To search for these symbols, they must be escaped. The escaping of reserved characters requires two backslashes, i.e. "\\".

If *-groupmembers* is specified, then group members to also be returned, otherwise, they are not.

If *-regexp* is specified, this causes all of the strings in ensuing *searchCommands* to be treated as regular expressions.

If *-glob* is specified, this causes all of the strings in ensuing *searchCommands* to be treated as glob-style expressions. This means that the special character '*' will be expanded to mean any number of any kind of character. I.e., 'foo*' means all the strings starting with 'foo'.

The find command does not return the pad surface (id #1). All digits are treated as item ids, i.e. `".pad find -regexp withtag 5*"` will look for the object with an id of 5.

The fastest find possible is a **`withtag`** *searchCommand* without a regular or glob-style expression. The slowest finds occur when regular or glob-styles expression are used on string arguments. In this case, for every item on the surface, the regular or glob-styles expression is compared to the particular attribute of each object.

SearchCommand may take any of these forms:

`all`
: Returns all the items on the pad.

`above tagOrId:`
: Returns the items above (after) the one given by *tagOrId* in the display list. If *tagOrId* denotes more than one item, then the lowest (first) of these items in the display list is used to search above. If the search type is a regular expression or glob-style search which denotes more than one item, then the first tag will be used, based on alphabetical order, and then the highest (last) of these items is used to search above.

`below tagOrId`
: Returns the item just before (below) the one given by *tagOrId* in the display list. If *tagOrId* denotes more than one item, then the first (lowest) of these items in the display list is used.

`closest x y [halo]`
: Returns the items closest to the point given by x and y. If halo is specified, then any items closer than halo to the point will be returned. Halo must be a non-negative number. If halo is not specified, then only items overlapping the point (x, y) will be returned.

`withinfo info`

If a regular expression or glob-style search is used, this returns all the items for which their info itemconfigure option matches the pattern *info*. If an exact search is used, this returns all the items for which their info itemconfigure option is the same as the string *info*.

`withlayer layer`

If a regular expression or glob-style search is used, this returns all the items for which the name of their layer matches the pattern *layer*. If an exact search is used, this returns all the items in which the name of their layer is the same as the string *layer*.

`withname name`

If a regular expression or glob-style search is used, this returns all the items for which their name matches the pattern *name*. If an exact search is used, this returns all the items for which their name is equal to the string *name*. A name is a URL for an HTML item, and a filename for textfile and image items.

`withsticky type`

Returns all the items that are sticky *type*.

`withtag tagOrId`

If *tagOrId* is a number, this returns that item. If a regular expression or glob-style search is used, this returns all the items for which their tag matches the pattern *tagOrId*. If an exact search is used, this returns all the items for which their tag is equal to the string *tagOrId*.

`withtext text`

If a regular expression or glob-style search is used, this returns all the items for which their text matches the pattern *text*. If an exact search is used, this returns all the items for which their text is equal to the string *text*.

`withtype type`

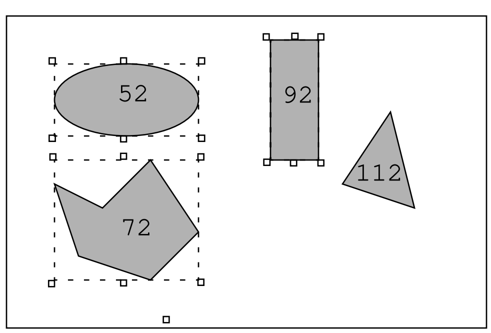
If a regular expression or glob-style search is used, this returns all the items for which their type matches the pattern *type*. If an exact search is used, this returns all the items for which their type is equal to the string *type*.

enclosed x_1 y_1 x_2 y_2

Returns all the items completely enclosed within the rectangular region given by x_1, y_1, x_2, and y_2. x_1 must be no greater then x_2 and y_1 must be no greater than y_2.

overlapping x_1 y_1 x_2 y_2

Returns all the items that overlap or are enclosed within the rectangular region given by x_1, y_1, x_2, and y_2. x_1 must be no greater then x_2 and y_1 must be no greater than y_2.



```
.pad find withtag selected
       52 72 92

.pad find withtag selected && !withtype rectangle
       52 72
```

[32] `pathName` **`focus`** `[tagOrId [portalID ...]]`

Set the keyboard focus for the Pad++ widget to the item given by *tagOrId*. If a list of *portalID*'s are specified, then the item sits on the surface looked onto by the last portal. If *tagOrId* refers to several items, then the focus is set to the first such item in the display list. If *tagOrId* doesn't refer to any items then the focus isn't changed. If *tagOrId* is an empty string, then the focus item is reset so that no item has the focus. If *tagOrId* is not specified then the command returns the id for the item that currently has the focus, or an empty string if no item has the focus. If the item sits on a different surface than *pathName*, then this command also returns the pathName of the item.

Once the focus has been set to an item, all keyboard events will be directed to that item. The focus item within a Pad++ widget and the focus window on the screen (set with the Tk focus command) are totally independent: a given item doesn't actually have the input focus unless (a) its pad is the focus window and (b) the item is the focus item within the pad. In most cases it is advisable to follow the focus widget command with the focus command to set the focus window to the pad (if it wasn't there already). Note that there is no restriction on the type of item that can receive the Pad++ focus.

[33] `pathName` **`font`** `subcommand [args ...]`

`This command is used for manipulating fonts.` Fonts are specified using a logical font naming scheme similar to Java's, rather than using a platform-specific filename as a font name. Font names follow the format "<facename>-<stylename>-<size>", where <facename> is the typeface, e.g. Times, Helvetica, etc. <stylename> is "plain", "bold", "italic", or "bolditalic". <size> is the height of the font in pixels. <style> is optional (default is "plain"). <size> is also optional (default is 12). Fonts are substituted when the original cannot be located. Fonts specified using the old scheme are automatically translated to this scheme. The special font name "Line" specifies to use the Pad++ built-in line font. This font is ugly, but is faster than the regular fonts. Some Example font names are: "Times", "Helvetica", "Times-12", "Helvetica-bold", "Times-bold-18". The font subcommands are:

`font bbox string font [fontheight]`

Returns a list with four elements giving the bounding box of *string* if it is drawn with the **`render draw text`** command. The list has the form "x_1 y_1 x_2 y_2" such that the text is within the region bounded by x_1 on the left, x_2 on the right, y_1 on the bottom, and y_2 on the top. The bounding box is affected by the **`render configure -font`** and **`-fontheight`** commands.

`font path [[+]path]`

Pad++ uses a search path to locate font files. Set or get the global font path used in Pad++. *path* is a list of directory names, separated by spaces. Font files in these directories are expected to have the extension ".pfa". The default path is /usr/lib/X11/fonts/Type1

If the '+' character is included, then the specified path is appended on to the existing search path. Otherwise, it replaces the path.

`font loadbitmaps font`

Attempts to load a set of X Bitmaps for *font*, which are used for drawing text at small sizes. e.g. "`.pad loadbitmaps Helvetica-Bold`".

`font maxbitmapsize size`

Specifies the maximum size for which X font bitmaps should be loaded when the 'font loadbitmaps' command is executed. This can be useful when making presentations if you want to force large fonts to be loaded.

`font names`

Returns the names of all the font faces/styles available on the current system as a list.

[34] `pathName` **`freeborder`** `border`

WARNING: **`freeborder`** is an obsolete command and will be removed in the next release. Replace all uses of **`freeborder`** with the '**`border free`**' command.

[35] `pathName` **`freecolor`** `color`

WARNING: **`freecolor`** is an obsolete command and will be removed in the next release. Replace all uses of **`freecolor`** with the '**`color free`**' command.

[36] `pathName` **`freeimage`** `imagetoken`

WARNING: **`freeimage`** is an obsolete command and will be removed in the next release. Replace all uses of **`freeimage`** with the '**`image free`**' command.

[37] `pathName` **`getdate`**

Returns the current date and time in the standard unix time format.

% .pad getdate
Wed May 29 20:01:49 1996

[38] `pathName` **`getgroup`** `tagOrId`

Return the group id that *tagOrId* is a member of. If *tagOrId* is not a member of a group, then this command returns an empty string. If *tagOrId* specifies more than one object, then this command refers to the first item specified by *tagOrId* in display-list order. (Also see the **`addgroupmember`**, and **`removegroupmember`** commands).

[39] `pathName` **`getlevel`**

WARNING: **`getlevel`** is an obsolete command and will be removed in the next release. Replace all uses of **`getlevel`** with the '**`render configure -level`**' command.

[40] `pathName` **`getmag`** `tagOrId`

WARNING: **`getmag`** is an obsolete command and will be removed in the next release. Replace all uses of **`getmag drawtext`** with the '**`render configure -mag`**' command.

[41] `pathName` **`getmodifier`**

WARNING: **`getmodifier`** is an obsolete command and will be removed in the next release. Replace all uses of **`getmodifier`** with the '**`modifier get`**' command.

[42] `pathName` **`getpads`**

Returns a list of all the Pad++ widgets currently defined.

[43] `pathName` **`getportals`**

WARNING: **`getportals`** is an obsolete command and will be removed in the next release. Replace all uses of **`getportals`** with the '**`render configure -portals`**' command.

[44] `pathName` **`getsize`** `tagOrId ?portalID ...?`

Returns the largest dimension of the first item specified by tagOrId. If a portal list is specified, then the size of the item within the last portal is returned.

[45] `pathName` **`gettags`** `tagOrId`

Return a list whose elements are the tags associated with the item given by *tagOrId*. If *tagOrId* refers to more than one item, then the tags are returned from the first such item in the display list. If *tagOrId* doesn't refer to any items, or if the item contains no tags, then an empty string is returned.

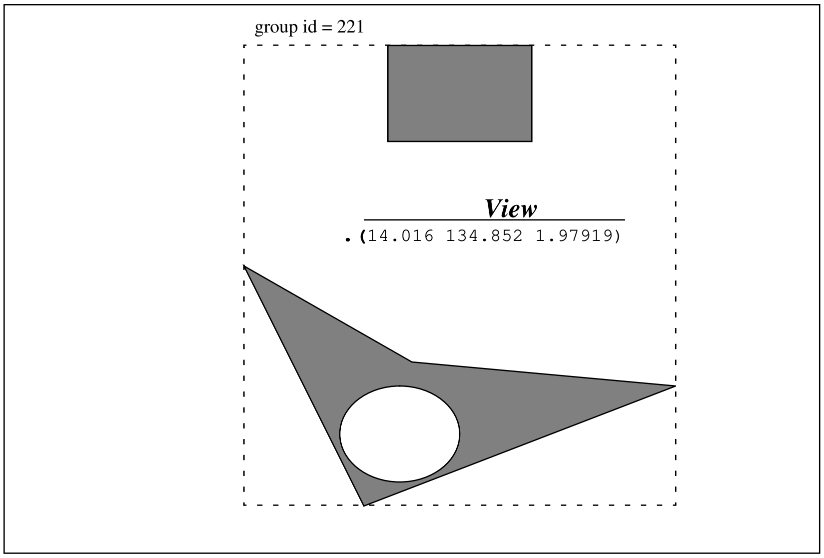
[46] `pathName` **`gettextbbox`** `string`

WARNING: **`freeborder`** is an obsolete command and will be removed in the next release. Replace all uses of **`freeborder`** with the '**`border free`**' command.

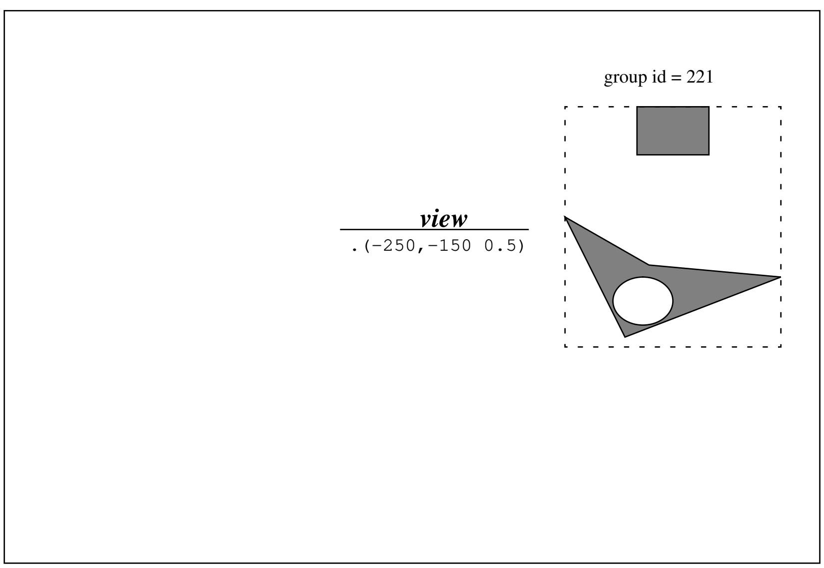
[47] `pathName` **`getview`** `[portalID ...]`

Returns the current view of the main window in "*xview yview zoom*" form. Here, (*xview*, *yview*) specifies the point at the center of the window, and *zoom* specifies the magnification. If a list of *portalID's* is specified, than the view of the last portal is returned instead of the view of the main window. (See **`moveto`** to set the current view).

```
.pad getview
14 134 2
.pad ic 221 -position
8 118 1
```



.pad moveto -250 -150 0.5
.pad getview
-250 -150 0.5
.pad ic 221 -position
8.1125 118.753 1



[48] `pathName` **`getzoom`** `[portalID ...]`

Returns the current magnification of the main window. If a list of *portalID's* is specified, than the view of the last portal is returned instead of the view of the main window. This is a shortcut for the last parameter returned by the **`getview`** command. (See **`moveto`** to set the current view).

[49] `pathName` **`grab`** `[-root | -path pathName | -win winId] \`
`[-dim {width height}] x y width height`

This captures a rectangular portion of the screen and makes an imagedata which can then be used to create image items (Also see **`image`** **[53]** command and **Image Items**.) The **`grab`** command takes a region (x, y, width, height) which specifies the area to grab. Note that *y* represents the top of the region. The region can be relative to a specific Tk window, any other X window, or the entire screen. By default, the region is relative to the pad widget window. The region actually grabbed is clipped to the specified window (or to the screen for root grabbing.)

The window the region is relative to can be specified with the *-root*, *-path*, or *-win* flags. *-path* is used to specify an existing Tk window. -root is used to specify that the region is relative to screen. *-win* is used to specify any X window by the window id. X window id's can get accessed from the *xlswins* program that comes with most X systems. If a dimension is specified, then the image is shrunk through simple decimation to produce the desired resolution before it is returned. Images can only be shrunk, not grown with the dimension flag. Note that images can be distorted by setting a dimension with a different aspect ratio than the source.

X displays support multiple display characteristics called visuals (8-bit pseudocolor, 24-bit truecolor, etc.). Windows on the same screen can use different visuals. Because of this, grabbing from the root can grab from different windows. If the windows have a different visual than the root, those colors of those windows will be undefined.

Grab returns an imagedata token that could be used to create an image:

```
set imagedata [.pad grab 0 0 200 200]
.pad create image -image $imagedata
```

[50] `pathName` **`grid`** `option arg [arg ...]`

The **`grid`** command arranges one or more objects in rows and columns and treats them as a group. It is based on the Tk grid geometry manager and its behavior and Tcl syntax are very similar to it. In pad, all grid commands are sub-commands of the pad command. See the section on GRID ITEMS for a complete description of this command, and how to create and use grids.

[51] `pathName` **`hastag`** `tag tagOrId`

Determines if the item specified by *tagOrId* contains the specified *tag*. This command returns "1" if the item does contains the specified tag, or "0" otherwise. If *tagOrId* refers to more than one item, then the comparison is performed on the first item in the display list. If *tagOrId* doesn't refer to any items, then "0" is returned.

[52] `pathName` **`html`** `subcommand arg ...`

This is the command for manipulating html pages and html anchors. There are several subcommands:

`html configure tagOrId [option [value] ...]`

Configures the specified html page. Option-value pairs may be specified as with the itemconfigure command, or if no options are specified, a list of all options and values are returned.

`-source` (read only)

Returns the HTML source of the page

`-type` (read only)

Returns the mime type of the page contents

-lastchangedate (read only)
Returns the last time the html source was modified, as specified by the server.

-length (read only)
Returns the length of the html source in characters.

`html anchor configure tagOrId [option [value] ...]`

Configures the specified html anchor. Option-value pairs may be specified as with the itemconfigure command, or if no options are specified, a list of all options and values are returned.

-html (read only)
Returns the id of the html page this anchor is associated with.

-image (read only)
Returns the image token this anchor is represented by if the anchor is an image anchor.

-ismap (read only)
Returns true if the anchor is an imagemap.

-name (read only)
Returns the name of the anchor

-state
Returns the current state of the anchor (unvisited, visited, or active).

-url (read only)
The URL this anchor is linked to.

[53] `pathName` **`image`** `subcommand arg ...`

This is the command for manipulating image data. In Pad++, the data associated with an image is manipulated separately from an image item. With this approach, the multiple Pad++ image items can use the same image data. There are several subcommands:

`image alloc <file>`

Allocates an image data for future use by image items and render callbacks. *file* specifies the name of a file containing an image. **`image alloc`** can always read gif file formats. In addition, if Pad++ is compiled with the appropriate libraries, it can also read jpeg and tiff image file formats, and will automatically determine the file type. The image may have transparent pixels. This returns an image token which can be used by related commands.

`image free <imagetoken>`

Frees the image data previously allocated by **`image alloc`**.

`image names`

Returns a list of all allocated image data tokens.

`image configure <imagetoken> [option [value] ...]`

Configures the specified image data. Option-value pairs may be specified as with the itemconfigure command, or if no options are specified, a list of all options and values are returned.

-dimensions (read only)
Returns a list of the dimensions of the image data (width, height).

-name (read only)

Returns the file the image data token was created from.

-rgb (can set only to 0)

Normally, image data are stored internally with their full rgb colors in addition to a colormap index. This allows images to be rendered with dithering, but takes 5 bytes per pixel. If the -*norgb* option is specified, then the original rgb information is not stored with the image and the image can not be rendered with dithering, but only takes 1 byte per pixel.

For example, the following code creates two image items that use the same image data:

```
set imagedata [.pad image alloc "foo.gif"]
.pad create image -image $imagedata -anchorpt "0 0"
.pad create image -image $imagedata -anchorpt "200 0"
```

[54] `pathName` **`info`** `subcommand`

A command for accessing information about the pad. *subcommand* may be any of the following: *status*. Each subcommand may have sub-subcommands and options. All the subcommands and their options follow:

`status render`

This returns a debugging line specifying some information relevant to the last render. It returns the number of objects on the surface, the number ob objects rendered in the last render, the render level, and the time in milliseconds the last render took.

`status sharedmemory`

When Pad++ is running on X, it uses X shared memory to render images quickly. This return true if Pad++ is using X shared memory.

[55] `pathName` **`isvisible`** `tagOrId [portalId ...]`

Returns true if the first item specified by *tagOrId* is visible. If any portals are specified, then this returns true if the item is visible within the last portal on the list.

[56] `pathName` **`itemconfigure`** `[-nondefaults] tagOrId [option [value] ...]`
`ic` is an alias for **`itemconfigure`**

A command for accessing information about the pad. *subcommand* may be any of the following: *status*. pathName **`itemconfigure`** [-nondefaults] tagOrId [option [value] ...]

This command is similar to the **`configure`** command except that it modifies item-specific options for the items given by *tagOrId* instead of modifying options for the overall pad widget. If no *option* is specified, then this command returns a list describing all of the available options for the first item given by *tagOrId*. If the -*nondefaults* flag is specified, then only those options modified by an application will be returned. If *option* is specified with no *value*, then the command returns the value of that option. If one or more option-value pairs are specified, then the command modifies the given widget option(s) to have the given value(s) in each of the items given by *tagOrId*; in this case the command returns an empty string. If *value* is an empty string, then that option is set back to its default value.

The options and values are the same as those permissible in the **`create`** command when the item(s) were created; see the sections below starting with OVERVIEW OF ITEM TYPES for details on the legal options. Note that the **`itemconfigure`** command generates a **`<Modify>`** event on the items modified by it (see the **`itemconfigure`** command for a description of the **`<Modify>`** event). Locked items may not be modified by the **`itemconfigure`** command (see the -*lock* itemconfigure option).

[57] `pathName` **`layer`** `subcommand [args ...]`

This command controls creation and deletion of layers, and provides a method to return the current layers. Layers are used to control rendering order, and visibility. Every item sits on a single layer. Each surface can have any number of layers, and the layers are rendered in sequence. In addition, each view can specify which layers can be seen within that view (via the **`-visiblelayers`** [67] itemconfigure option.)

While layers are implicitly defined when they are used, this command allows the creation of a layer before it is used, and thus ordering of layers can be defined before objects are created. There are several subcommands:

`layer create <layer>`

Creates a new layer, and gives it the name *layer*. There are no items on a new layer, and the layer is put on top of all existing layers.

`layer delete <layer>`

Deletes the specified layer. If any items are on a layer when it is deleted, then all of those items are deleted as well.

`layer names`

Returns a list of all the current layer names.

[58] `pathName` **`layout`** `subcommand [args ...]`

This command performs various kinds of one-time layouts. That is, it repositions and resizes objects based on subcommands, but does not manage the objects in the future. Attaching a layout call to a <Modify> event provides a way to define custom layout managers. The subcommands are:

```
layout align <type> [-anchor] [-coords {x y ...} [-overlaponly]] \
     tagOrId [tagOrId ...]
```

<type> can be -left, -right, -top, or -bottom

Align the specified items so that their bounding boxes line up on the specified side. If *-anchor* is specified, then line up by anchor point instead of by bounding box. If coordinates are specified with *-coords*, then align items to the path specified by those coordinates. Otherwise, use the item furthest in the alignment direction to align the others to. If *-coords* is specified, then *-overlaponly* may be specified which means that items should only be aligned if they overlap the specified path. In all cases, items are aligned so the furthermost object doesn't move. That is, if you are aligning to the left, then all objects are moved to be aligned with the left-most object.

A simple example moves all the items that have the tag "`foo`" so they are aligned on top:

```
.pad layout align -top foo
```

`layout distribute <type> [-space space] tagOrId [tagOrId ...]`

<type> must be "-horizontal", "-vertical", or "-coords {x_1 y_1 x_2 y_2 ...}"

Distribute the specified items so that the space between them is equalized horizontally or vertically (by bounding box). Alternatively, *-coords* can be specified in which case the items will be distributed along the path specified by the coordinates with equal spacing between items. *-space* can be specified in which case the items will be distributed so there is *space* between each item. If *-space* and *-coords* are specified, then the items will be distributed along with the path specified by the coordinates with *space* between each item. If the items take up more space than is available on the specified path, they will continue along an extension of the last portion of the path.

```
layout position [-time animationTime] x1 y1 <type> x2 y2 tagOrId \
   [tagOrId ...]
```

<type> must be "-ref tagOrId" or "-bbox {bbx1 bby1 bbx2 bby2}"

Position the specified objects relative to a target object, or a bounding box. Specify target point by a point on the unit square, and specify the source point by a point on the unit square. If *animationTime* is specified, then the objects are animated to their new position in the specified time (in milliseconds).

The following code moves all the objects with the tag "`foo`" so they have the same lower left corner as item #72. Then, all the objects with the tag "`bar`" are moved so that their upper right corner is at the same position as the lower left corner of item #72.

```
.pad layout position 0 0 72 0 0 foo
.pad layout position 0 0 72 11 bar
```

```
layout size <type> [-ref tagOrId] [-scale scale] tagOrId [tagOrId ...]
```

<type> must be "-width", "-height"

Scale the specified objects so that their bounding boxes are scaled (width or height) to the target. If a reference object is specified, then scale relative to that object. Otherwise, scale to an absolute dimension. Objects are scaled around their anchor points.

```
layout snap grid tagOrId [tagOrId ...]
```

Position the objects so that their anchor points are snapped to *grid*.

[59] `pathName` **`line2spline`** `error` x_1 y_1 `...` x_n y_n

Takes the coordinates for a line, and uses an adaptive curve fitting algorithm to generate the coordinates for a spline that approximates the line. The spline coordinates are returned. *error* is a floating point number indicating how closely the spline curve should follow the line. Using a smaller error will tend to generate a spline made with more bezier segments that follow the line more accurately. Using a larger error will produce fewer bezier segments but the fit will be less accurate. See the section on SPLINE ITEMS on how splines are specified in Pad++. (Also see **`spline2line`**.)

[60] `pathName` **`lower`** `[-one] [-layer] tagOrId [belowThis]`

Move all of the items given by *tagOrId* to a new position in the display list just before the item given by *belowThis*. If *tagOrId* refers to more than one item then all are moved but the relative order of the moved items will not be changed. *belowThis* is a tag or id; if it refers to more than one item then the first (bottommost) of these items in the display list is used as the destination location for the moved items. If *belowThis* is not specified, then *tagOrId* is lowered to the bottom of the display list. If the *-one* flag is specified, then *tagOrId* is lowered down one item in display order which may or may not have a visible effect. *-one* and *aboveThis* may not both be specified. If any items to be lowered are group members, they are lowered within their group rather than being lowered on the pad surface. Returns an empty string.

If *-layer* is specified, then rather than lowering a set of items, it lowers the layer specified by *tagOrId*. (See the **`layer`** **[57]** command for more information about layers.)

[61] `pathName` **`modifier`** `subcommand [args ...]`

The modifier command manipulates the user-specified modifier for event bindings. A user-specified modifier is a software equivalent of the Shift, Control, or other modifier keys. They can be used to isolate event bindings that all belong to one mode. See the documentation of the **bind** command for a more complete description. There are several subcommands:

`modifier create <modifier>`

Define *modifier* to be a user-defined modifier that can be used in future event bindings.

`modifier delete <modifier>`

Return the current active modifier.

`modifier get`

Delete *modifier* from the list of valid user-defined modifiers. Any event bindings that are defined with this modifier become invalid.

`modifier set <modifier>`

Make *modifier* be the current active modifier for this widget. *modifier* must have been previously defined with the '**modifier create**' command.

[62] pathName **moveto** [-twostep] xview yview zoom [time [portalID ...]]

Change the view so that the point "*xview yview*" is at the center of the screen with a magnification of *zoom*. If *xview*, *yview*, or *zoom* is specified as "", then that coordinate is not changed. If *-twostep* is specified, then make animation in two steps if appropriate (i.e., points not too close). The two steps are such that it zooms out to the midpoint between the two points far enough so that both start and endpoints are visible, and then zooms to the final destination. If *time* is specified, then the change in view will be animated in enough evenly spaced frames to fill up *time* milliseconds. If a list of *portalID*'s are specified, then the view will be changed within the last specified *portalID* rather than within the main view. The return value is the current view. (See **getview** to get the current view). Note that the **moveto** command generates a **<Modify>** event if a portal's view is changed (see the **bind** command for a description of the **<Modify>** event).

[63] pathName **noise** index

Returns a repeatable noise value based on the floating-point value of index. This noise function is equal to 0 whenever index is an integer. Typically, noise is called with slowly incrementing values of index. The closer the consecutive values of index are, the higher the frequency of the resulting noise will be. This noise function is from Ken Perlin at New York University (http://www.mrl.nyu.edu/perlin).

Example:

```
set coords ""
set noiseindex_x 0.1928
set noiseindex_y 100.93982
set noiseincr 0.052342
for {set i 0} {$i < 100} {incr i } {
   set x [expr 500.0 * [.pad noise $noiseindex_x]]
   set y [expr 500.0 * [.pad noise $noiseindex_y]]
   lappend coords $x
   lappend coords $y
   set noiseindex_x [expr $noiseindex_x + $noiseincr]
   set noiseindex_y [expr $noiseindex_y + $noiseincr]
}
eval .pad create line $coords
```

[64] `pathName` **`padxy`** `[-sticky] [-portals] winx winy [-gridspacing value]`

Given a window x-coordinate *winx* and y-coordinate *winy*, this command returns the pad x-coordinate and y-coordinate that is displayed at that location. If *-sticky* is specified, the coordinate transform is done ignoring the current view (i.e., as for sticky objects.) If *-portals* is specified, then the point (*winx*, *winy*) is passed through any portals it on. If *-gridspacing* is specified, then the pad coordinate is rounded to the nearest multiple of value units.

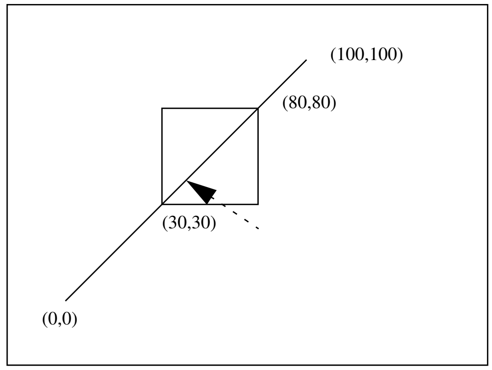
[65] `pathName` **`pick`** `[-divisible] [-indivisible] winx winy`

Given a window coordinate (*winx*, *winy*), it returns the visible object underneath that point. If the point should pass through any portals, a **`<PortalIntercept>`** event will be fired which will determine if the event will pass through that portal. By default, the **`pick`** command uses the divisibility of individual groups to determine if group members should be picked. However the *-divisible* or *-indivisible* flags (only one of which may be specified) override group's divisibility. If *-divisible* is specified, then group members will be picked from any group the point hits. If *-indivisible* is specified, then group objects and not group members will be picked.

```
% .pad create line 0 0 100 100
22
.pad create rectangle 30 30 80 80
23

.pad addmodifier  Pick
.pad bind all <Pick-ButtonPress-1> {
   event_Press  %i %j %x %y %O
}

proc event_Press {i j x y obj} {
               # Get the group object not the group members
               # underneath the point x y
   set container [.pad pick -indivisible $x $y]
   puts "container $container object: $obj coords: ($i, $j)"
}

.pad setmodifier Pick
```

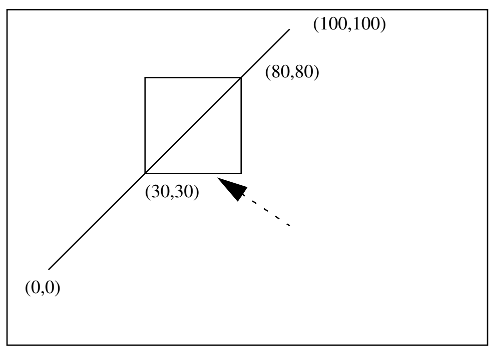
Now, group the line and rectangle:

```
% .pad create group -members "22 23"
24
```



Now, click on the line, the system response with:

```
container 24 object: 22 coords: (37.5, 36)
```



Now, click on the rectangle, system response with:

```
container 24 object: 23 coords: (66.5, 28)
```

Now, change the pick command as:

```
set container [.pad pick -divisible $x $y]:
```

Then click on the line:

```
container 22 object: 22 coords: (52.5, 52)
```

Click on the rectangle:

```
container 23 object: 23 coords: (63.5, 30)
```

[66] `pathName` **`popcoordframe`**

Pops the top frame off the stack of coordinate frames. The resulting frame on the top of the stack becomes active. Also see **`pushcoordframe`** and **`resetcoordframe`**. Returns the frame popped off the stack.

[67] `pathName` **`printtree`**

Prints the current hierarchical tree of items to stdout (used for debugging). Returns an empty string.

[68] `pathName` **`pushcoordframe`** `tagOrId`
`pathName` **`pushcoordframe`** x_1 y_1 x_2 y_2

Pushes a coordinate frame onto the stack of coordinate frames. When any coordinate frames are on the stack, all coordinates are interpreted relative to the frame instead of as absolute coordinates. A frame is a bounding box, and all coordinates are specified within the unit square where the unit square is mapped to the frame.

Note that the *-penwidth* and *-minsize* and *-maxsize* itemconfigure options are also relative to the coordinate frame. In these cases, a value of 1 refers to the average of the frame dimensions.

Text and images are scaled so that one line of text, or the height of the image is scaled to the height of the coordinate frame at a scale of 1 (using the *-position* or *-scale* itemconfigure options).

For example, the following code makes 50 nested rectangles. Note that the width of the rectangles shrinks proportionally.

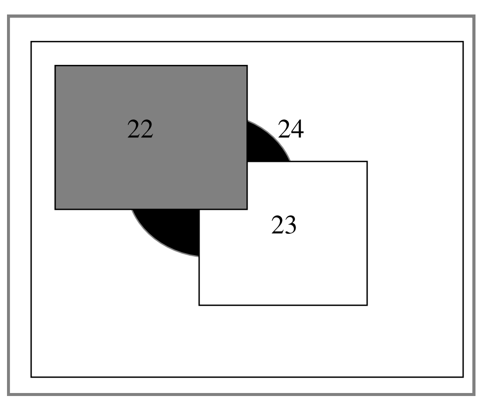
```
for {set i 0} {$i < 50} {incr i} {
   set id [.pad create rectangle 10 10 80 80 -penwidth 2]
   .pad pushcoordframe $id
}
.pad resetcoordframe
```

Also see **`popcoordframe`** and **`resetcoordframe`**. Returns the current coordinate frame.

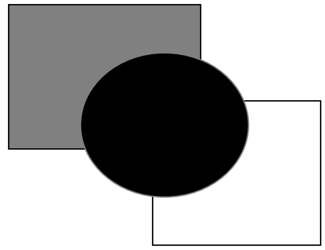
[69] `pathName` **`raise`** `[-one] [-layer] tagOrId [aboveThis]`

Move all of the items given by *tagOrId* to a new position in the display list just after the item given by *aboveThis*. If *tagOrId* refers to more than one item then all are moved but the relative order of the moved items will not be changed. *aboveThis* is a tag or id; if it refers to more than one item then the last (topmost) of these items in the display list is used as the destination location for the moved items. If *aboveThis* is not specified, then *tagOrId* is raised to the top of the display list. If the *-one* flag is specified, then *tagOrId* is raised up one item in display order which may or may not have a visible effect. *-one* and *aboveThis* may not both be specified. If any items to be raised are group members, they are raised within their group rather than being raised on the pad surface. Returns an empty string.

If *-layer* is specified, then rather than raising a set of items, it raises the layer specified by *tagOrId*. (See the **`layer`** **[57]** command for more information about layers.)

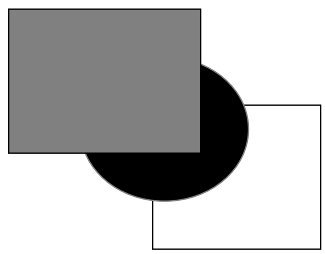


.pad raise 24



If we use the `-one` option:
`.pad raise -one 24`

The original position turns to be:



[70] `pathName` **`random`** `min max`

Returns a random integer between the specified *min* and *max* points, inclusively.

[71] `pathName` **`read`** `filename`

Executes the tcl commands in the filename. If filename is created with the write command, then this command reads the pad scene back in. Returns an empty string.

[72] `pathName` **`removegroupmember`** `[-notransform] tagOrId`

Remove all items specified by *tagOrId* from the group they are a member of, and return them to the pad surface. If any of the items were members of hierarchical groups, they are removed from all groups. If any of the items are not a member of a group, then they are not affected. Items removed are added to the pad surface just after the group in terms of display-list order.

By default, items are transformed so they don't change their location when removed from a group - even if the group has a transformation. This is implemented by transforming the item's transformation to be the inverse of the group's transformation. If the *-notransform* flag is specified, this inverse transformation is not applied, and the item will move by the group's transformation when removed. *(*Also see the **`addgroupmember`**, and **`getgroup`** commands). Returns an empty string.

[73] `pathName` **`render`** `subcommand arg ...`

The **`render`** command is used to manipulate the state of the renderer, and to render onto the screen during a renderscript. This command can only be called within a render callback.

`render scale dz`

Magnifies all rendering performed in the current renderscript by dz.

`render translate dx dy`

Translates all rendering performed in the current renderscript by (dx, dy).

`render draw border bordertoken relief width` x_1 y_1 x_2 y_2
`render draw filledborder bordertoken relief width` x_1 y_1 x_2 y_2

Draws a fake 3D border connecting the specified coordinates. (See **`border`** commands). This command can only be called within a render callback. *Border* must have been previously allocated by **`border`**. *Type* must be one of "`raised`", "`flat`", "`sunken`", "`groove`", "`ridge`", "`barup`", or "`bardown`". The '**`draw border`**' command draws just the border while the '**`draw filledborder`**' command draws the border with the inside filled with the color of the border. The following example creates an object that draws a border:

```
set border [.pad allocborder #803030]
   .pad create rectangle 0 0 100 100 -renderscript {
   .pad render draw border $border raised 5 0 0 100 100
}
```

`render draw image imagetoken x y`

Draws the image specified by imagetoken at the point (x, y). This command can only be called within a render callback.

`render draw line` x_1 y_1 x_2 y_2 `[`x_n y_n `...]`

Draws a multi-segment line connecting the specified coordinates.This command can only be called within a render callback.

`render draw polygon` x_1 y_1 x_2 y_2 `[`x_n y_n `...]`

Draws a closed polygon connecting the specified coordinates. This command can only be called within a render callback.

`render draw text string x y`

Draws the specified text at the specified location. This command can only be called within a render callback.

`render configure [option [value] ...]`

Configures the state of the renderer. Option-value pairs may be specified as with the itemconfigure command, or if no options are specified, a list of all options and values are returned.

`-capstyle` `capstyle`

Sets the capstyle of lines for drawing within render callbacks. *Capstyle* may be any of: "butt", "projecting", or "round".

`-joinstyle` `joinstyle`

Sets the joinstyle of lines for drawing within render callbacks. *Joinstyle* may be any of: "bevel", "miter", or "round".

`-linewidth` `width`

Sets the linewidth (in current units) to *width* for future drawing with render callbacks. The actual width of the line will depend on the size of the object and the magnification of the view. If width is 0, then the line is always drawn 1 pixel wide.

-color `color`
Sets the color for future drawing with render callbacks. *Color* must have previously been allocated by **`color alloc`**.

-font `fontname`
Sets the font for future drawing with render callbacks. This affects the result of the **`font bbox`** command. *Fontname* must specify a filename which contains an Adobe Type 1 font, or the string "`Line`" which causes the Pad++ line-font to be used. Defaults to "`Times-12`".

-fontheight `height`
Sets the height of the font for future drawing with render callbacks. *Height* is specified in the current pad units. This affects the result of the **`font bbox`** command.

-level (read-only)
Returns the current render level. (See the sections on *Refinement* and *Region Management and Screen Updating* in the Programmer's Guide for more information about render levels).

-mag (read-only)
Returns the current magnification of *tagOrId* for this specific render (it could be rendered multiple times if visible through different portals). Magnification is defined as the multiplication of the current view (including portals) with the object's size (from the *-position* itemconfigure option).

-portals (read-only)
Returns the list of the portals the current object is being rendered within.

[74] `pathName` **`renderitem`** `[tagOrId]`

During a render callback triggered by the *-renderscript* option, this function actually renders the object. During a *-renderscript* callback, if `renderitem` is not called, then the object will not be rendered. If *tagOrId* is specified, then all the items specified by *tagOrId* are rendered (and the current item is not rendered unless it is in *tagOrId*). This function may only be called during a render callback. Returns an empty string.

[75] `pathName` **`resetcoordframe`**

Pops all the frames off of the coordinate stack. Results in an empty stack, so all coordinates are back to absolute coordinates. Also see **`pushcoordframe`** and **`popcoordframe`**. Returns an empty string.

[76] `pathName` **`rotate`** `tagOrId angle [xctr yctr]`

Rotates all the items specified by *tagOrId* angle degrees. If (*xctr*, **yctr**) is specified, then all the items are rotated around the specified point. If the rotation point is not specified, then each item is rotated around its anchor point. All item types are rotatable except html pages, and widgets (such as buttons, scrollbars, and textfields). If a non-rotatable item is rotated, a Tcl error will be generated. (see the *-angle* itemconfigure option).

[77] `pathName` **`scale`** `tagOrId [scaleAmount [ctrx ctry [animationTime]]]`

Scale each of the items given by *tagOrId* by multiplying the size of the item with *scaleAmount*. Scale the items around the item's center, or around the point (*ctrx*, *ctry*), if specified. This command returns the scale of the first item. Note that the **`scale`** command generates a **`<Modify>`** event on the items modified by it (see the **scale** command for a description of the **<Modify>** event). Locked items may not be modified by the **scale** command (see the *-lock* itemconfigure option).

If *animationTime* is specified, then all the items moved will be animated over a period of *animationTime* milliseconds.

[78] `pathName` **`setcapstyle`** `capstyle`

WARNING: **setcapstyle** is an obsolete command and will be removed in the next release. Replace all uses of **setcapstyle** with the '**render configure -capstyle**' command.

[79] `pathName` **`setcolor`** `color`

WARNING: **setcolor** is an obsolete command and will be removed in the next release. Replace all uses of **setcolor** with the '**render configure -color**' command.

[80] `pathname` **`setfont`** `fontname`

WARNING: **setfont** is an obsolete command and will be removed in the next release. Replace all uses of **setfont** with the '**render configure -font**' command.

[81] `pathname` **`setfontheight`** `height`

WARNING: **setfontheight** is an obsolete command and will be removed in the next release. Replace all uses of **setfontheight** with the '**render configure -fontheight**' command.

[82] `pathname` **`setid`** `tagorid id`

Sets the id of an existing item to *id*. If tagorid specifies more than one item, then the first item is used. Returns an empty string. This generates an error if an invalid id is specified (i.e., if it is in use), or if *tagorid* does not specify an object.

[83] `pathName` **`setjoinstyle`** `joinstyle`

WARNING: **joinstyle** is an obsolete command and will be removed in the next release. Replace all uses of **joinstyle** with the '**render configure -joinstyle**' command.

[84] `pathName` **`setlanguage`** `language`

Sets the language to be used for callback scripts that are created in the future. All callback scripts that have already been created will be evaluated in the language that was active at the time they were created. This command refers to all callback scripts including event handlers, render scripts, timer scripts, zoom actions, etc. Pad++ always includes at least the Tcl scripting language, but others may be active, depending on how Pad++ was built. This command controls whatever languages are currently installed. The language defaults to "automatic" where it tries to guess the language based on the syntax of the script. See the SCRIPTING LANGUAGES section in the Programmer's Guide for more details. (Also see the **settoplevel** command.)

[85] `pathName` **`setlinewidth`** `width`

WARNING: **setlinewidth** is an obsolete command and will be removed in the next release. Replace all uses of **setlinewidth** with the '**render configure -linewidth**' command.

[86] `pathName` **`setmodifier`** `modifier`

WARNING: **setmodifier** is an obsolete command and will be removed in the next release. Replace all uses of **setmodifier** with the '**modifier set**' command.

[87] `pathName` **`settoplevel`** `language`

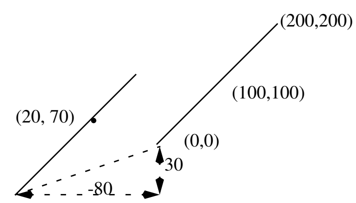
Sets the language that the top-level interpreter should use. Pad++ always includes at least the Tcl scripting language, but others may be added. Returns an empty string. See the SCRIPTING LANGUAGES section in the Programmer's Guide for more details. (Also see the **setlanguage** command.)

[88] `pathName` **`slide`** `tagOrId [dx dy [animationTime]]`

Slide each of the items given by *tagOrId* by adding *dx* and *dy* to the x and y coordinates of the item's transformation (i.e., their *-position* itemconfigure option). This command returns a string with the (x, y) position at the item's anchor point. Note that the **slide** command generates a **<Modify>** event on the items modified by it (see the **slide** command for a description of the **<Modify>** event). Locked items may not be modified by the **slide** command (see the *-lock* itemconfigure option).

If *animationTime* is specified, then all the items moved will be animated over a period of *animationTime* milliseconds.

```
.set id [.pad create line 0 0 200 200]
```



.pad slide $id -80 30
20.000000 70.000000

[89] `pathName` **`sound`** `subcommand args ...`

Rudimentary sound support is available for the Irix (SGI) and Linux platforms. Currently, only ".au" sound file formats are supported. By default, Pad++ is built without sound. See the README file for instructions on building Pad++ with sound. The following subcommands implement sound:

`.pad sound load file`
This command loads a sound file, and returns a sound token that can be used to play the sound later.

`.pad sound play sound_token [-volume volume]`
This will play a sound specified by *sound_token* which is a sound loaded with the "**sound load**" command. This returns a token that is used to stop the sound if needed. If *volume* is specified (at a range of [0-100]), then the sound is played at the given volume (temporarily overriding the system configuration). Short sounds are played asynchronously. There are no guarantees, but in practice, sounds under about a half second are played in the background, and this function immediately. In the future, there will be better control over this.

`.pad sound stop play_token`

This stops the sound referenced by *play_token*

`.pad sound configure [option [value] ...]`

`-sounds` (read-only)

This returns a list of the currently loaded sounds.

`-volume` `master`
`-volume` `{left right}`

This sets the volume of all sounds to be played. If a single parameter is given, it is treated as the *master* volume, and sets the sound for both channels. If two parameters are given (as a two-element list), they set the left and right speaker volumes separately. In all cases, this returns a list of the left and right speaker volumes. Volumes are specified in the range [0-100].

[90] `pathName` **`spline2line`** `error` x_1 y_1 ... x_n y_n

Takes the coordinates for a spline and uses an adaptive bezier algorithm to generate the coordinates for a line that approximates the spline. *error* is how much error is allowed - a small error produces a greater number of points and more accuracy. A large error yields fewer points but the line is less accurate. See the section on SPLINE ITEMS for details on how splines are created. (Also see **`line2spline`**.)

[91] `pathName` **`text`** `tagOrId option [arg ...]`

Allows interaction with all text item types. This includes text, textfile, textfield, and textarea items. See TEXT ITEMS for a description of indices and marks. *tagOrId* specifies the text item to apply the following command to. *Option* and the *args* determine the exact behavior of the command. Note that the **`text`** command generates a **`<Modify>`** event on the items modified by it (see the **`text`** command for a description of the **`<Modify>`** event). Locked items may not be modified by the **`text`** command (see the *-lock* itemconfigure option). The following command options are available:

- **`compare`** `index1 op index2`

  Compares the indices given by *index1* and *index2* according to the relational operator given by *op*, and returns 1 if the relationship is satisfied and 0 if it isn't. *Op* must be one of the operators <, <= ==, >=, >, or !=. If op is == then 1 is returned if the two indices refer to the same character, if op is < then 1 is returned if *index1* refers to an earlier character in the text than *index2*, and so on.

- **`delete`** `index1 [index2]`

  Delete a range of characters from the text. If both *index1* and *index2* are specified, then delete all the characters starting with the one given by index1 and stopping just before *index2* (i.e. the character at *index2* is not deleted). If *index2* doesn't specify a position later in the text than *index1* then no characters are deleted. If *index2* isn't specified then the single character at *index1* is deleted. The command returns an empty string.

- **`get`** `index1 [index2]`

  Return a range of characters from the text. The return value will be all the characters in the text starting with the one whose index is *index1* and ending just before the one whose index is *index2* (the character at *index2* will not be returned). If *index2* is omitted then the single character at *index1* is returned. If there are no characters in the specified range (e.g. *index1* is past the end of the file or *index2* is less than or equal to index1) then an empty string is returned.

- **`index`** `index [char]`

Returns the position corresponding to index in the form line.char where line is the line number and char is the character number. If char is specified, then the position is returned in the form char which is the character index from the beginning of the file. Index may have any of the forms described under INDICES.

- **`insert`** index chars

Inserts chars into the text just before the character at *index* and returns an empty string.

- **`mark`** `option [arg arg ...]`

This command is used to manipulate marks. The exact behavior of the command depends on the *option* argument that follows the mark argument. The following forms of the command are currently supported:

`mark` `names`

Returns a list whose elements are the names of all the marks that are currently set.

`mark` `set markName index`

Sets the mark named *markName* to a position just before the character at index. If *markName* already exists, it is moved from its old position; if it doesn't exist, a new mark is created. This command returns an empty string.

`mark` `unset markName [markName ...]`

Remove the mark corresponding to each of the *markName* arguments. The removed marks will not be usable in indices and will not be returned by future calls to `pathName mark names`. This command returns an empty string.

[92] `pathName` **`tree`** `subcommand [args ...]`

This command creates, maintains, and animates dynamic trees of Pad items. Items are created by other pad functions, and are placed into hierarchical tree structures to be managed by this code. These trees support a focus + context viewing structure, multiple foci, and a focus function which has a controlled level of influence on the tree.

Each node has a layout object associated with it which controls the position and resizing of the pad item at that node during a layout. Each layout controls a link item - a pad item created by the tree code, which graphically connects the node to its parent. This link item is maintained automatically by the tree code, but may be accessed and manipulated through the **`tree`** subcommand.

Each pad has a treeroot object, which is a list of all pad tree nodes on the surface. Each of these "root nodes" is an invisible treenode which controls certain subtrees on the pad surface. This organization is necessary to keep trees independent. Animation done at a node affects that node and its children, so we need to be careful to organize the nodes in such a way that all nodes we wish to "know" about each other are connected in some manner. Separate hierarchies can be made to "avoid" each other during animation by connecting them together under an invisible root node. When the layout function is called on the root node, both hierarchies will be laid out according to the layout object which resides at the root node.

A dynamic tree supports an arbitrary number of foci. Management of these foci is left up to the user. A node's focus is spread by a function which has several parameters. See the **`setfocus`** subcommand for more information.

Manipulation of the tree structure falls into four parts - tree management, layout, animation control, and parameter control.

Tree Management
A tree can be added to by creating new nodes and adding them to the existing tree structure. Nodes and subtrees can be moved within trees. Nodes and subtrees can be deleted, which will also delete the pad item associated with the treenode. Nodes and subtrees can be removed, which simply removes the treenode associated with the object, but leaves the object itself alone.

Layout
The default layout provided with the current version of this code creates a hierarchical tree in which a node's children are laid out to the right of the node. This layout prevents any overlapping of nodes by calculating the bounding box of the subtree rooted at a node, and laying out nodes so that these bounding boxes do not intersect.

Animation control
A tree always animates its members. It may also animate the view at the same time the members are being animated.

Parameter control.
There are a variety of parameters associated with the layout at a node, and the control of animation of a tree.

Trees are created and manipulated through the tree subcommands:

`addnode` `childtagOrId parenttagOrId`

> Adds *childtagOrId* to *parenttagOrId* as a child. If *childtagOrId* already has a parent, this command also removes *childtagOrId* from that parent. When it is added to the tree, the item's current zoom is recorded, and is used in all future calculations in the dynamic tree layouts. This means that an item's size when it is added to the tree is the size that it will have when it has a focus of 1.0. (See the tree **`setscale`** command to modify the size of an item after it has been added to a tree.)

`animatelayout` `tagOrId [-view view]`

> Used in conjunction with computelayout, this command performs the animated layout of a tree. It may be given a view, which forces the system to animate the system view while the tree animation is taking place. Use **`getlayoutbbox`** to calculate a view for the finished animation. See **`computelayout`** for specific implementation instructions.
>
> Using **`animatelayout`** with the *-view* option forces an animation of the view as the tree is animating. The view animates from the current view to the one specified as the tree animation is taking place.

`animateview` `tagOrId [value]`

> Sets the animateView flag at *tagOrId*. Controls whether or not a layout will animate the view when layout is called at *tagOrId*.

`connect` `tagOrId`

> Draws links from *tagOrId* to its parent, and from *tagOrId*'s children to *tagOrId.*

computelayout tagOrId

Computes the final layout state for a dynamic tree. This places final layout state information in the tree, some of which can be accessed in order to control the layout. For information on accessing some of this information, see the **getlayoutbbox** command.

This code computes the future layout of a tree, then animates its view so that the center of the tagOrId's future position is in the center of the screen at the end of the animation. Note that any treenode which is a descendant of *tagOrId* will return valid information on a call to get **layoutbbox**. Other nodes are not guaranteed to have valid information.

```
.pad tree computelayout $node
set futureBbox [.pad tree getlayoutbbox $node]
set view [bbcenter $futureBbox]
.pad tree animatelayout -view $view
```

create tagOrId

Creates a treenode to monitor *tagOrId*. Creates default layout for treenode. Adds *tagOrId* to the padroot, in preparation for placement somewhere else in the hierarchy.

createroot

Creates an invisible root node which is used to organize subtrees of information, and returns the pad id of the dummy object at that node. Used to connect several nodes together so that they appear to be root nodes at the same level. Because this is an invisible node, no links will be drawn to it.

delete [-subtree] tagOrId

Delete the *tagOrId* and its associated pad object, layout, and link. By default, when there is no subtree option, *tagOrId*'s children are promoted to be children of *tagOrId*'s parent. If the -*subtree* option is used, the entire subtree and all of its associated data and pad objects are deleted.

getchildren tagOrId

Returns a list of the ids of the pad objects which are children of *tagOrId*

getfocus tagOrId

Returns the focus value at a *tagOrId*, which is a number on the interval [0.0, 1.0]

getlayoutbbox tagOrId

Returns the approximate bbox *tagOrId* will have at the end of the current animation. This is only valid when used after **computelayout**, and before any manipulation of any member of the tree. Moving or resizing any object affected by **computelayout** will cause a few bugs in the animation of those objects when **animatelayout** is called. The system will not break, but any moved object will first instantly move to the position it held when **computelayout** was called, and then will animate to the position **computelayout** determined for that object. Relative sizing of objects will be ignored by the system.

getlink tagOrId

Return the id of the item which graphically links *tagOrId* to its parent.

getparent `tagOrId`

Return the id of the parent of *tagOrId*.

getroot `tagOrId`

Gets the root node of *tagOrId*'s hierarchy - the node which resides just below the padroot.

isnode `tagOrId`

Returns a boolean indicating whether or not *tagOrId* has a treenode attached to it, and is therefore a member of a hierarchy.

layout `tagOrId [-view view]`

Performs a recursive layout of the subtree rooted at *tagOrId*. If the *-view* option is used, the tree will animate to the view provided.

lower `tagOrId [belowtagOrId]`

Controls the position of *tagOrId* in the order of its siblings. If *belowtagOrId* is not provided, *tagOrId* is moved to the bottom of the list. If *belowtagOrId* is provided, *tagOrId* is moved to a position just above (after) *belowtagOrId*.

raise `tagOrId [abovetagOrId]`

Controls the position of *tagOrId* in the order of its siblings. If *abovetagOrId* is not provided, *tagOrId* is moved to the top of the list. If *abovetagOrId* is provided, *tagOrId* is moved to a position just above (after) *abovetagOrId*.

removenode `[-subtree] tagOrId`

Removes the treenode and layout objects associated with *tagOrId*. If the *-subtree* is not included, *tagOrId*'s information is removed, and *tagOrId*'s children are promoted. If the *-subtree* option is used, the entire treenode hierarchy is removed.

reparent `[-subtree] tagOrId parenttagorid`

Reparents *tagOrId* to belong to *parenttagorid*. The default case, in which the *-subtree* option is not used, reparents *tagOrId*, and promotes any children *tagOrId* may have to be children of *tagOrId*'s original parent. If the *-subtree* option is used, the subtree rooted at *tagOrId* is moved.

setanimatespeed `tagOrId milliseconds`

Sets the time for an animation to occur. If this number is 0, the animation will proceed immediately to the end state. During an animation, if any event is detected, the animation will proceed to the end state. Thus, a double click on a treenode forces the animation to happen instantaneously.

setfocus `tagOrId [value [levels [falloff]]]`

Set the focus value at a *tagOrId*. This must be a number on the range [0,1]. If no *value* is provided, the focus is set to 1.0. The *levels* parameter controls the number of levels this focus is allowed to spread. The *falloff* parameter is a multiplier which controls the portion of focus which is passed on to the next level of recursion. For example, if this number is 0.75, then *focus**0.75 of the focus is passed on at the next level of recursion.

setfocusmag `tagOrId value`

Recursive set command - works on the entire subtree of the *tagOrId* is is given. Set the magnification difference between an object of focus 0 and an object of focus 1.

setlinkmode `mode`

If *mode* is "`fixed`", this sets the penwidth of treelinks to fixed width 1 pixel. If mode is "`scaling`", then penwidth scales. The change is applied to all descendants of the specified treenode

setscale `tagOrId value`

Set the scale that an object will have when its focus is 0. This is the smallest size that an object will have in a dynamic tree. When a tree *tagOrId* is created, this value is automatically set to the z value of the object.

setspacing `tagOrId xvalue [yvalue]`

Set the x and y spacing at a *tagOrId*. This is the amount of spacing between a *tagOrId* and its spatial neighbors.

[93] `pathName` **type** `tagOrId`

Returns the type of the item given by *tagOrId*, such as rectangle or text. If *tagOrId* refers to more than one item, then the type of the first item in the display list is returned. If *tagOrId* doesn't refer to any items at all then an empty string is returned.

[94] `pathName` **update** `[-dissolve speed [withRefinement]]`

This forces any outstanding updates to occur immediately. If the -dissolve flag is specified, then speed determines how quickly the update is done. If speed is 0, the update will happen quickly with a swap buffer. If speed is between 1 and 3, the update will happen with a dissolve effect where 1 is the fastest and 3 is the slowest. If the *withRefinement* flag is specified, this forces all refinements to occur immediately as well - which could be a slow process. Returns an empty string.

[95] `pathName` **urlfetch** `URL ?option value ...?`

```
pathName urlfetch Token
where valid options are:
     -file <filename>
-var <variable>
     -updatescript <updateScript>
     -donescript <doneScript>
     -errorscript <errorScript>
```

Retrieves the specified *URL* (Universal Resource Locator) from the World Wide Web. This command returns immediately, and the retrieval is done in the background (within the same process using a file handler.) As portions of the data comes in, *updateScript* will be executed, and *doneScript* will be executed when all of the data has completely arrived. If there are any errors retrieving the data, then *errorScript* will be executed. **`urlfetch`** returns a token that can be used to interact with this retrieval. This token is appended to *updateScript*, *doneScript* and *errorScript* when the scripts are executed.

There are three methods to access the data retrieved by urlfetch. The first method is to specify a file (with *-file*) in which case the data is written to that file as it is retrieved. The second method is to specify a Tcl variable (with *-var*) in which case the data is stored in that global variable as it is retrieved. The variable will be updated with the current data before *updateScript* and *doneScript* are executed. Note that the variable is not cleared by **`urlfetch`** and it is the responsibility of the caller to free it (with **`unset`**). The third method is to use the second form of **`urlfetch`** by passing it url token during an updatescript callback in which case it will return the data retrieved by that fetch. Three code segments follow which show the use of urlfetch.

```
#
# urlfetch example using a file
#
proc done {filename token} {
   set file [open $filename "r"]
   ...  # handle file
}
set file "foo"
.pad urlfetch http://www.cs.unm.edu -file $file \
   -donescript "done $file"

#
# urlfetch example using a Tcl global variable
#
proc done {token} {
   global foo

   ...  # handle data in "foo"
   unset foo     ;# no longer need URL data
}
.pad urlfetch http://www.cs.unm.edu -var foo \
   -donescript "done"
#
# urlfetch example using a token to incrementally
# handle data as it comes in.
#
proc update {token} {
   set data [.pad urlfetch $token]
   ...  # handle incremental data
}
.pad urlfetch http://www.cs.unm.edu \
   -updatescript "update" -donescript "done"
```

[96] `pathName` **`warp`** `dx dy`

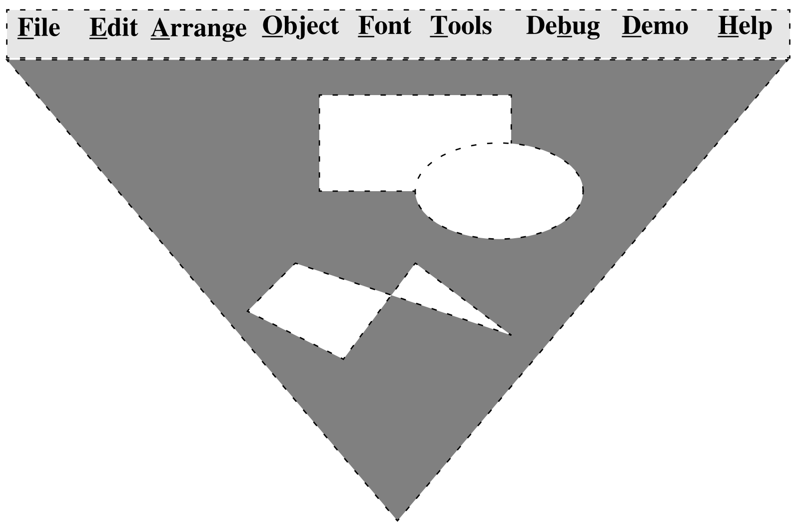
This moves the core pointer relative to its current position by (*dx*, *dy*) pixels. Moving the pointer is often called "warping", and thus the name of the command is warp. Note that generally speaking, warping the cursor is frowned upon in user interfaces, but this command is supplied as there are some cases where it is ok.

[97] `pathName` **`windowshape`** `[innercoords outercoords]`

Changes the shape of the top-level window containing the Pad++ widget specified by *pathName*. The two parameters each specify lists of coordinates that specify the shape of the window. All coordinates are scaled to fit the existing width of the window, larger numbers in X go to the right, and larger numbers in Y go up. *innercoords* represents the area that can be painted in, and *outercoords* represents the overall window shape. The difference between these two shapes becomes the windows border. If *innercoords* and *outercoords* are both empty strings, then the window returns to its default rectangular shape. This command returns the current window shape. If window has the default shape, it returns {} {}.

For example, the following command changes the top-level window shape to an inverted triangle.

```
.pad windowshape {0 50 50 50 25 0} {0 50 50 50 25 0}
```



[98] `pathName` **`write`** `[-format type] [-relative] file [tagOrId tagOrId ...]`

Writes out all the items on the Pad++ surface into *file*. If *tagOrId's* are specified, then just those items are written out. The file that is written out should be read back in with the **read** command. If *file* is an empty string, than this command returns a string containing the data instead of writing it to a file. If a valid filename is specified, then this command returns an empty string. Only non-default slots of each object are written out.

Files may be written in different formats (all of which can be read with the **`read`** command.) If *type* is "text", then the Tcl code that is used to recreate the items is written. If *type* is "binary-interchange", then a custom binary format is used. Both formats are intended to be readable by all future versions of Pad++. The text format is somewhat larger and slower to read, while the binary format is somewhat smaller and faster to read. The binary file format is described in the document *doc/fileformat.txt*.

If the *-relative* flag is specified, then all referenced files (such as textfiles and images) are referenced with filenames relative to the file that is saved. This makes it easier to move files between machines as an entire directory structure with the images can be copied, and the pad files will still work. If the *-relative* flag is not specified, then all referenced files are referenced with absolute pathnames.

As the **write** command writes out objects on the pad, it generates a **`<Write>`** event for each item it writes. The return string from the **`<Write>`** event handler will be appended to whatever string this function writes out for each item. See the **bind** command for more information on this.

[99] `pathName` **`zoom`** `zoomFactor padXloc padYloc [animateTime [portalID ...]]`

Zoom around specified pad position by multiplicitive factor. If *animateTime* is specified, then animate the zoom and take *animateTime* milliseconds for the animation. If an optional list of portals is specified, then change the view within the last portal. The entire list is necessary in case the last portal is sitting on a different surface then this function is called with. Returns an empty string.

Overview of Item Types

The sections below describe the various types of items supported by Pad++. Each item type is characterized by two things: first, the form of the command used to create instances of the type; and second, a set of itemconfiguration options for items of that type, which may be used in the **`create`** and **`itemconfigure`** widget commands. See the **`itemconfigure`** command for the syntax to use these options.

The available item types are:

Alias, Button, Canvas, Checkbox, Checkboxmenuitem, Choicemenu, Frame, Grid, Group, HTML, Image, KPL, Label, Line, Menu, Menubar, Menuitem, Oval, Pad, Panel, Polygon, Portal, Rectangle, Scrollbar, Spline, Tcl, Text, Textarea, Textfield, Textfile, and Window.

Several of the items are designed to mirror the functionality and usage of the standard widgets in Java's Abstract Windowing Toolkit (AWT). They are also accessible from Tcl along with all the other item types. These include:

Button, Canvas, Checkbox, Checkboxmenuitem, Choicemenu, Frame, Menu, Menubar, Menuitem, Panel, Scrollbar, Textarea, Textfield, and Window.

Item Options

Every item has several options that can be configured. Some options are available for all item types, and some options are available for just specific item types. All the options are summarized here followed by a complete description of each option. Afterwards, each item *type* is described with a list of which options apply to that item type.

This is a summary of every itemconfigure option in alphabetical order. Each option either applies to every possible item type, or has a list of item types to which it applies

`-aliases [1]`	(Read-only) Returns all aliases of the item
`-alwaysrender [2]`	True if the item must be rendered, even if the system is slow and the item is small
`-anchor [3]`	The part of the item that `-position` refers to
`-anchorpt [4]`	The (x, y) portion of `-position`
`-angle [5]`	Specifies absolute rotation of item
`-anglectr [6]`	Specifies absolute rotation of item, rotating about specified point
`-arrow [7]`	Whether to draw arrow heads with this item (Available only for line, spline types)
`-arrowshape [8]`	The shape of drawn arrow heads (Available only for line, spline types)
`-bb [9]`	A script that gets evaluated to specify the bounding box of an item (Available only for kpl, tcl types)
`-border [10]`	Specifies border color of item (Available only for html, portal types)
`-borderwidth [11]`	Specifies width of border

	(Available only for html, portal types)
-capstyle [12]	Specifies how to draw line ends (Available only for line, spline types)
-clipping [13]	Controls if items are clipped to their bounding box when rendered
-command [14]	Callback for widgets (Available only for button, command types)
-dither [15]	Render with dithering (Available only for image types)
-divisible [16]	True if events go through a group to its members (Available only for frame, grid, group, html, panel types)
-donescript [17]	A script to evaluate when a background action has completed (Available only for html types)
-editable [18]	True if text item is editable (Available only for text, textfile, textfield types)
-errorscript [19]	A script to evaluate when a background action has an error (Available only for html types)
-events [20]	True if item receives events, false otherwise
-faderange [21]	Range over which an item fades in or out
-file [22]	File an item should be defined by (Available only for textfile types)
-fill [23]	Specifies fill color of item (Available only for button, frame, html, label, scrollbar, panel, fill, polygon, portal, rectangle, textfield types)
-font [24]	Specifies font to use for text (Available only for button, html, label, portal, text, textfile, textfield types)
-from [25]	Starting value of valuator widget (Available only for scrollbar types)
-height [26]	Height of an item. Normally computed, but can be set to squash/stretch item
-html [27]	The HTML item associated with an htmlanchor (Available only for htmlanchor types)
-htmlanchors [28]	The anchors associated with an HTML page (Available only for html types)
-image [29]	Image data associated with item (allocated by image alloc) (Available only for htmlanchor, image types)
-info [30]	A place to store application-specific information with an item
-ismap [31]	True if an htmlanchor is an image map (Available only for htmlanchor types)
-joinstyle [32]	Specifies how to draw the joints within multi-point lines (Available only for line, spline, oval, polygon, rectangle types)
-layer [33]	The layer an item is on
-linesize [34]	Amount widget should change to represent a line change (Available only for scrollbar types)
-lock [35]	Locks an item so it can not be modified or deleted
-lookon [36]	Specifies the pad widget this item sees (Available only for portal types)
-maxsize [37]	The maximum size an item is rendered it (absolute or relative to window size)
-members [38]	The list of members of a group (Available only for frame, group, html, panel types)
-memberlabels [39]	List of labels for a pull-down or pop-up menu (Available only for menu and choicemenu types)
-menubar [40]	Menubar associated with a frame (Available only for frame types)
-minsize [41]	The minimum size an item is rendered it (absolute or relative to window size)
-noisedata [42]	Specifies parameters to render item with noise

	(Available only for line line types)
`-orientation [43]`	Orientation of widget (horizontal or vertical.) (Available only for scrollbar types)
`-pagesize [44]`	Amount widget should change to represent a page change (Available only for scrollbar types)
`-pen` [45]	Specifies pen color of item (Available only for button, label, line, spline, oval, polygon, portal, rectangle, text textfile, textfield types)
`-penwidth` [46]	Specifies width of pen (Available only for line, spline, oval, polygon, rectangle types)
`-position [47]`	The absolute position of the object (x, y, scale)
`-reference [48]`	What item an alias references (Available only for alias types)
`-relief [49]`	Specifies how border should be rendered (raised, flat, sunken, ridge, groove) (Available only for button, portal types)
`-renderscript [50]`	A script that gets evaluated every time an item is rendered
`-rposition [51]`	The relative position of the object (to groups)
`-scale [52]`	The (scale) portion of `-position`
`-state [53]`	State of an item (such as visited, unvisited, etc.) (Available only for button, htmlanchor types)
`-sticky` [54]	Specifies if an item should stay put when the view changes
`-tags [55]`	List of tags associated with an item
`-text [56]`	The text of any item containing text (Available only for button, label, text, textfile, textfield types)
`-timerrate [57]`	Frequency timerscript should fire
`-timerscript [58]`	Script associated with an item that fires at regular intervals
`-title [59]`	Some items only: Title of an item (Available only for portal types)
`-to [60]`	Ending value of valuator widget (Available only for scrollbar types)
`-transparency [61]`	Transparency of an item. 0 is completely transparent, 1 is completely opaque
`-updatescript` [62]	A script to evaluate when a background action has made progress (Available only for html types)
`-url [63]`	The URL associated with an item (Available only for html, htmlanchor types)
`-value [64]`	Current value of valuator widget (Available only for scrollbar types)
`-view [65]`	Specifies the view this item sees (Available only for pad, portal types)
`-viewscript` [66]	A script that gets evaluated whenever the view is changed
`-visiblelayers [67]`	The layers that are visible within this view (just for portals and surface, item #1) (Available only for pad, portal types)
`-width [68]`	Width of an item. Normally computed, but can be set to squash/stretch an item
`-writeformat [69]`	Controls whether disk-based item is written out by copy or reference (Available only for image types)
`-zoomaction [70]`	A script that gets evaluated when an item is scaled larger or smaller than a set size

This is a list of every itemconfigure option with their complete definition in alphabetical order.

`[1]-aliases   (read-only)`

(Available for all item types)

This returns all the alias items that reference this item.

`[2]-alwaysrender boolean`
(Available for all item types)

The rendering engine may decide to not render an item for reasons of efficiency (although it may get rendered at higher levels of refinement). When this flag is set (i.e., equals 1), the item will be rendered no matter how big it is (as long as it is bigger than its *-minsize*. Defaults to false (0).

`[3]-anchor anchorPos`
(Available for all item types)

AnchorPos tells how to position the object relative to the positioning point for the item (see `-position`); it may have any of the forms accepted by Tk_GetAnchor. For example, if anchorPos is "center" then the object is centered on the point; if anchorPos is "n" then the object will be drawn so that its top center point is at the positioning point. This option defaults to center.

`[4]-anchorpt {x y}`
(Available for all item types)

This is an alias for the first two elements of the -position itemconfigure option. (*x*, *y*) specifies the anchor point of the item. This means that the item will be positioned so that its anchor (north, center, southwest, etc.) will appear at the specified anchor point. (*x*, *y*) are absolute quantities, independent of the current view and independent of any group membership. (Also see the **`-anchor`**, **`-position`**, **`-rposition`**, and **`-scale`** itemconfigure options.)

`[5]-angle angle`
(Available for all item types except HTML and widgets)

Sets the absolute rotation of an item in degrees. The item is rotated around its anchor so that it is rotated *angle* degrees relative to its creation. (Also see the **`rotate`** command.)

`[6]-anglectr {angle xctr yctr}`
(Available for all item types except HTML and widgets)

Sets the absolute rotation of an item in degrees. The item is rotated around the point (*xctr*, *yctr*) so that it is rotated *angle* degrees relative to its creation. (Also see the **`rotate`** command.)

`[7]-arrow where`
(Available only for line, spline types)

Indicates whether or not arrowheads are to be drawn at one or both ends of the line. *where* must have one of the values "`none`" (for no arrowheads), "`first`" (for an arrowhead at the first point of the line), "`last`" (for an arrowhead at the last point of the line), or "`both`" (for arrowheads at both ends). This option defaults to "`none`".

`[8]-arrowshape shape`
(Available only for line, spline types)

This option indicates how to draw arrowheads. The *shape* argument must be a list with three elements, each specifying a distance. The first element of the list gives the distance along the line from the neck of the arrowhead to its tip. The second element gives the distance along the line from the trailing points of the arrowhead to the tip, and the third element gives the distance from the outside edge of the line to the trailing points. If this option isn't specified then Pad++ picks a "reasonable" shape.

`[9]-bb boundingboxScript`
(Available only for kpl, tcl types)

A script that will be evaluated to compute the bounding box of this item. For Tcl, It should return a 4 element list whose members are "x_1 y_1 x_2 y_2" which are the lower left and upper right corners of this items bounding box. For KPL, It should return two two-element vectors that specify ($x_{1,}$ y_1), ($x_{2,}$ y_2).

`[10]-border color`
(Available only for html, portal types)

Color specifies a color to use for drawing the border of the portal; it may have any of the forms accepted by Tk_GetColor. If *color* is "none", the outline will not be drawn. This option defaults to the fill color.

`[11]-borderwidth width`
(Available only for html, portal types)

Width specifies the width of the border in current units to be drawn around the item. Wide borders will be drawn completely inside the path specified by the points of this object. Note that this is different than pens. If *width* is 0, then the border will always be drawn one pixel wide, independent of the zoom. *Width* defaults to 1 pixel.

`[12]-capstyle cap`
(Available only for line, spline types)

Specifies how the ends of the line are drawn. *cap* may be one of:

- `butt`: The ends are drawn square, at the end point.
- `projecting`: The ends are drawn square, past the endpoint.
- `round`: The ends are rounded.

`[13]-clipping boolean`
(Available for all item types)

By default, built-in items (such as lines, text, etc.) do not get clipped to their bounding box, and procedural items (items with *-renderscripts*) do. This flag turns clipping on or off. Be warned, that turning off clipping for a procedural object is dangerous. If you draw outside the object's bounding box, you can end up with screen garbage. Defaults to true (1) for items with *-renderscripts*, and false (0) for all other items.

`[14]-command script`
(Available only for button, command types)

A script that gets executed when the widget is activated. The definition of activation for each widget is different. For example, a button is activated when it is pressed and released while the pointer is still over the button. A scrollbar is activated whenever the thumb is moved. Some widgets append information about the activation on the end of the script (for instance, scrollbars append the current value). See the description of each widget for information about this.

`[15]-dither dithermode`
(Available only for image types)

Specifies if and when the image is rendered with dithering. Dithering is a rendering technique that allows closer approximation to the actual image colors, even when the requested colors are not available. Rendering images with dithering is much slower than without, so this option allows control as to when (if at all), dithering is used. dithermode may be any of

- `nodither`: The image is never rendered with dithering.
- `dither`: The image is always rendered with dithering.
- `refinedither`: The image is initially rendered without dithering, and then refined with dithering.

Defaults to *refinedither* (dither only on refinement).

`[16]-divisible boolean`
(Available only for frame, grid, group, html, panel types)

Specifies whether events should go to the grid members. If *-divisible* is 1 (true), events never go to the grid object, but pass through it to the members. If the event is within the bounding box of the group, but does not hit any members, then it will be ignored by the group. If *-divisible* is 0 (false), then the event will go to the group if it is within the bounding box of the group whether there is a member at the place the event points to or not. Defaults to 1 (true).

`[17]-donescript script`
(Available only for html types)

If *script* is specified, it gets evaluated when the html item has completed loading - including all in-line images. *script* is postpended with the id of the html object. This is necessary because the script is typically specified on the create line where the id of the html object is not yet known.

`[18]-editable boolean`
(Available only for text, textfile, textfield types)

If editable is TRUE, then the item's default event handlers will allow the item to be edited. This applies only to text widgets. Default text editing includes mouse copy and paste, and uses basic emacs-like bindings for manipulating the cursor.

`[19]-errorscript script`
(Available only for html types)

If *script* is specified, it gets evaluated if there is an error creating the html item. An error can occur for many reasons - especially because creating an html typically starts a network communication process for fetching the URL. *script* is postpended with the id of the html object. This is necessary because the script is typically specified on the create line where the id of the html object is not yet known.

`[20]-events boolean`
(Available for all item types)

Controls whether an item receives input events. If set to false (0), it does not respond to events. Defaults to true (1).

`[21]-faderange value`
(Available for all item types)

Controls over how long a period an item fades out as it approaches its minimum or maximum size. *value* specifies this period as a percentage of the object's size (from 0.0 to 1.0). Where 0.0 means that the item doesn't fade out all, it just blinks off when its extreme is reached, and 1.0 means that it slowly fades out over its entire range of sizes. Defaults to 0.3. (Also see the *-minsize* and *-maxsize* itemconfigure options.)

`[22]-file fileName`

(Available only for textfile types)

fileName specifies the filename to read a text file from.

`[23]-fill color`

(Available for button, frame, html, label, scrollbar, panel, fill, polygon, portal, rectangle, textfield types)

Fill the background of the html item with *color*, which may be specified in any of the forms accepted by Tk_GetColor. If *color* is "none", the background will not be drawn. It defaults to the background of the Pad++ widget it is created on.

`[24]-font fontname`

(Available only for button, html, label, portal, text, textfile, textfield types)

Specifies the font to be used for rendering text for this item. fontname must specify a filename which contains an Adobe Type 1 font, or the string "`Line`" which causes the Pad++ line-font to be used. Defaults to "`Times-12`".

`[25]-from value`

(Available only for scrollbar types)

Specifies the starting (lowest) value for a valuator widget to use. (Also see the **`-to`**, **`-linesize`** and **`-pagesize`** itemconfigure options.)

`[26]-height height`

(Available for all item types)

By default, the height of every item is automatically computed based on its contents. If the `-height` option is set, however, then this overrides the automatically computed value. Most items are squashed or stretched to fit the specified *height*. Note that text and alias items, however, are clipped instead of being squashed or stretched. (Also see the `-width` itemconfigure option.)

`[27]-html`

(Available only for htmlanchor types)

Returns the html item this anchor belongs to. This is a read-only option.

`[28]-htmlanchors`

(Available only for html types)

Returns all the anchors that are part of this HTML item. This is a read-only option, and may not be set.

`[29]-image imagetoken`

(Available only for htmlanchor, image types)

Specifies the image data associated with this item. Note that changing which image data an item uses does not effect the image data. Specifically, if the *-image* is set to the empty string, the image data it previously specified is unaffected and still needs to be deallocated with the "**`image free`**" command if it is no longer being used. (Also see the **`image`** command.)

`[30]-info info`
(Available for all item types)

A generic info field where the user may place any string. (See the **find** withinfo command).

`[31]-ismap`
(Available only for htmlanchor types)

Returns true if this anchor is an imagemap. This is a read-only option.

`[32]-joinstyle join`
(Available only for line, spline, oval, polygon, rectangle types)

Specifies how the joints at vertices are drawn. *join* may be one of:

- `bevel`: The joints are drawn without protruding. They are cut-off and sharp.
- `miter`: The joints are drawn protruding to a point.
- `round`: The joints are rounded.

`[33]-layer layer`
(Available for all item types)

Specifies the layer the item is on. Every item sits on a layer (which is specified by a string), and each view (top-level window and portals) specifies which layers are visible within that view. This gives control over objects are visible where and can be used with portals to implement very simple filters. (See the *-visiblelayers* itemconfigure option of portals and the top-level window which is specified by the surface (item 1). Defaults to "main".

`[34]-linesize value`
(Available only for scrollbar types)

Specifies the amount a valuator widget should change to represent a line change. For a scrollbar, this is the amount changed when the trough is clicked. (Also see the **`-from`**, **`-to`** and **`-pagesize`** itemconfigure options.)

`[35]-lock lock`
(Available for all item types)

When an item is locked, it can not be deleted or modified (except for changing the lock status). Note that attempting to modify or delete a locked item does not generate an error. It fails silently. This is so it is easy to modify all items associated with a tag and if certain items are locked they will just not get modified. The restricted commands on locked items are: **`coords`**, **`delete`**, **`itemconfigure`**, **`scale`**, **`slide`**, and **`text`**.

`[36]-lookon surface`
(Available only for portal types)

Specifies which Pad++ surface this portals looks onto. surface should be the complete pathName of a Pad++ widget. Defaults to the surface the portal was created on.

[37]-maxsize size

(Available for all item types)

Specifies the maximum size (in current units) this item should be rendered at. That is, if the view is such that the largest dimension of this object is greater than size units, it will not be displayed. When an object is not displayed because it is too large, it does not receive events. When an object approaches its maximum size it will fade out until it completely disappears when it reaches its maximum size. If *size* is -1, then it has no maximum size and will never disappear because it is too large. See the *-faderange* itemconfigure option to control how quickly an item fades out.

size may also be specified as a percentage of the view it is visible in (top-level window or portal). To specify size as a percentage, it should be in the range from 0 to 100 and end with a "%". Example:

```
.pad ic 5 -minsize 55%
```

size defaults to 10,000 pixels.

Also note that the rendering engine may decide to not display an item for reasons of efficiency if it is reasonably small. See the *-alwaysrender* flag to avoid this.

[38]-members members

(Available only for frame, group, html, panel types)

members is a list of object ids that specify the list of members of this group. Setting the members of a group first removes all existing members, and then inserts the new members. The members are rendered in the order they are specified in *members*.

[39]-memberlabels labels

(Available only for menu and choicemenu types)

Specifies a list of labels that can be used when creating a menu or choicemnu instead of explicitly creating a menuitem for each label. I.e.:

```
.pad create menu -memberlabels {"Content" "Index" "Help"} \
   -text "Help"
```

[40]-menubar menubar

(Available only for frame types)

Specifies the menubar associated with this frame (if any). By associating a menubar with a frame, the menubar is resized so as to be positioned along the top of the frame in the traditional manner. When the frame is resized, the associated menubar is also resized.

[41]-minsize size

(Available for all item types)

Specifies the minimum size (in current units) this item should be rendered at. That is, if the view is such that the largest dimension of this object is less than size units, it will not be displayed. When an object is not displayed because it is too small, it does not receive events. When an object approaches its minimum size it will fade out until it completely disappears when it reaches its minimum size. See the *-faderange* itemconfigure option to control how quickly an item fades out.

size may also be specified as a percentage of the view it is visible in (top-level window or portal). To specify size as a percentage, it should be in the range from 0 to 100 and end with a "%". Example:

```
.pad ic 5 -minsize 55%
```

size defaults to 0.

Also note that the rendering engine may decide to not display an item for reasons of efficiency if it is reasonably small. See the *-alwaysrender* flag to avoid this.

`[42]-noisedata noisedata`
(Available only for line types)

Specifies the noise parameters used to make rough-looking lines. `noisedata` is a four element list of numbers of the form:

"Pos Freq Amp Steps"

Rough lines are generated using the Perlin noise function. The Perlin noise function is like a sin function with a very irregular amplitude - like sin, noise has a constant period (one), but no two segments of the noise curve are alike. Noisy lines are generated by adding noise to the tangent direction of a line.

In the current implementation, there are four noise parameters: Pos, Freq, Amp, and Steps. Pos determines what part of the noise curve is sampled for that object. Freq determines the rate of sampling, Amp indicates the level, and Steps indicates how many samples to introduce per line segment. The drawing algorithm is straightforward. For each line segment, coordinates are generated as follows:

```
DrawRoughLine(x1, y1, x2, y2, Pos, Freq, Amp, Steps) :
  step  = 1.0/Steps;
  mag   = length(x1,y1,x2,y2);
  theta = direction(x1,y1,x2,y2);

  xmag = Amp * sin(theta) * mag;
  ymag = Amp * cos(theta) * mag;

  vertex(x1, y1);

  for (a = step; a < steps; a += step) {
     n = noise(Pos);
     vertex(lerp(a,x1,x2) + n*xamp, lerp(a,y1,y2) + n*yamp);
     Pos += Freq;
  }
  vertex(x2, y2);
```

Note that we multiply Amp by mag, the length of the line. This is necessary in Pad++ since the zooming functionality means that lines can be of nearly any size. Making the level of noise proportional to the length of the line keeps the informality uniform at all sizes. (We should probably also modulate the number of points generated by the thickness of the line, so small thin lines are cheap).

Values of 0.3 for Freq, 0.1 for Amp, 10 for Steps produces pleasant-looking lines. Pos can be an arbitrary floating point number - giving different objects unique values for Pos ensures that each object has a different appearance.

`[43]-orientation orientation`
(Available only for scrollbar types)

Specifies the *orientation* of a rectangular widget. orientation can be "`horizontal`" or "`vertical`".

`[44]-pagesize value`
(Available only for scrollbar types)

Specifies the amount a valuator widget should change to represent a page change. For a scrollbar, this is the amount changed when an arrow is clicked. (Also see the **`-from`**, **`-to`** and **`-pagesize`** itemconfigure options.)

`[45]-pen color`
(Available for button, label, line, spline, oval, polygon, portal, rectangle, text, textfile, textfield types)

Color specifies a color to use for drawing the line; it may have any of the forms acceptable to Tk_GetColor. It may also be "none", in which case the line will not be drawn. This option defaults to black.

`[46]-penwidth width`
(Available only for line, spline, oval, polygon, rectangle types)

Width specifies the width of the pen in current units to be drawn around the item. Wide lines will be drawn centered on the path specified by the points. If *width* is 0.0, then the pen will always be drawn one pixel wide, independent of the zoom. *Width* defaults to 1 pixel.

`[47]-position {x y scale}`
`-pos` is an alias for `-position`
(Available for all item types)

This specifies an items position and size through three variables (*x*, *y*, *scale*). (*x*, *y*) specifies the anchor point of the item. This means that the item will be positioned so that its anchor (north, center, southwest, etc.) will appear at the specified anchor point. *scale* specifies the magnification of an item. (*x*, *y*, *scale*) are all absolute quantities, independent of the current view and independent of any group membership. Items that have coordinates (lines, rectangles, polygons, and portals) have a default **`-position`** which depends on the coordinates of the item. For a "center" anchor (the default), the position will be the center of the coordinates. Other items (that don't have coordinates) have a default position of "0 0 1". (Also see the **`-anchor`**, **`-anchorpt`**, **`-rposition`**, and **`-scale`** itemconfigure options.)

The **`-position`** option may alternatively be given the special token "`center`" which means that the item should positioned and scaled so that it biggest dimension fills up 75% of the window, and it is centered. (This is dependent on the current view, and the current window dimensions.)

`[48]-reference id`
(Available only for alias types)

Specifies the id of an item that an alias references.

`[49]-relief relief`
(Available only for button, portal types)

Specifies the relief to be used by the border of this item. *relief* may be any of: *raised*, *sunken*, *flag*, *ridge*, or *groove*. Defaults to "*ridge*"

`[50]-renderscript TclScript`
(Available for all item types)

Specifies a Tcl script that will be evaluated every time the object is rendered. The script gets executed when the object normally would have been rendered. By default, the object will not get rendered. The script may call the renderitem function at any point to render the object. An example is:

```
.pad itemconfigure 22 -renderscript {
   puts "Before"
   .pad renderitem
   puts "After"
}
```

It would be possible to get in an endless render loop with the `-renderscript` option. If a `-renderscript` callback triggers a render which causes that item to be redrawn, the system will be in an endless render loop. To avoid this problem, items do not implicitly trigger damage within a `-renderscript` callback. If you do want to explicitly damage an item within a `-renderscript` callback, you must use the **damage** command. Be very careful to avoid infinite render loops.

[51]`-rposition {x y scale}`
`-rpos` is an alias for `-rposition`
(Available for all item types)

This is similar to the **`-position`** itemconfigure option, but (x, y, scale) are relative to the current group's position (whereas they are absolute for **`-position`**.) If setting this option on an item that is not a member of a group, then it behaves identically to -position. If setting this option on an item that is a member of a group, then the item will actually appear at a position that is first specified by the item's position, and then transformed by the group's position. Note that this option can be difficult to use and generally is not recommend. (Also see the **`-anchor`**, **`-anchorpt`**, **`-position`**, and **`-scale`** itemconfigure options.)

[52]`-scale scale`
(Available for all item types)

This is an alias for the first last (third) element of the -position itemconfigure option. *scale* specifies the magnification of an item, and is an absolute quantity, independent of the current view and independent of any group membership. (Also see the **`-anchor`**, **`-anchorpt`**, **`-position`**, and **`-rposition`** itemconfigure options.)

[53]`-state state`
(Available only for button, htmlanchor types)

Specifies the state of the anchor (which controls its color). There is no direct control over an anchor's color. Rather, it uses the default colors unless the HTML page specifies anchor colors. *State* may be one of "unvisited", "active", "visited", or "notloaded". In-line images that haven't been loaded yet are "notloaded".

[54]`-sticky style`
(Available for all item types)

Specifies if this item should be "sticky". Sticky items are constrained by the view so that whenever the view changes, sticky items are moved in response. There are several different kinds of sticky constraints. The simplest one (*style* == '`1`') makes the sticky item "stick" to the screen, independent of the current view. That is, as the view pans and zooms, sticky items appear effectively stuck to the screen. The different kinds of sticky constraints are described in detail below. Sticky items are rendered in their normal stacking order, and thus sticky items can appear above or below non-sticky items. (See the **getview** and **moveto** commands.) Defaults to 0 (false).

There are four kinds of sticky objects. They are:

- Regular sticky items (*style* == `'1'`). These don't move at all as the view changes.
- "Sticky Z" items (*style* == `'z'`). These do not zoom, but they pan normally. That is, they when the view changes, their (x, y) position does not change, but their scale is recalculated so their size does not change. This can be appropriate for handles or labels where you don't want their size to change, but you do want them to stay with other related objects. As a result of this, the old 'handle' object type has now been deleted. The previous handles never worked quite right within portals (they left screen junk), and their functionality is almost completely replaced by sticky z objects. Note that one thing sticky z objects can not do that handles did do is that sticky z objects don't scale only at the top-level view. Since they are otherwise regular objects, they can appear scaled within portals.

Example: The following code creates a rectangle with 4 non-zooming "handles" on its corners.

```
set rect [.pad create rectangle 0 0 100 100 -fill white]
.pad create rectangle 0 0 6 6 -fill red -pos "0 0 1" -sticky z
.pad create rectangle 0 0 6 6 -fill red -pos "100 0 1" -sticky z
.pad create rectangle 0 0 6 6 -fill red -pos "0 100 1" -sticky z
.pad create rectangle 0 0 6 6 -fill red -pos "100 100 1" -sticky z
```

- "Sticky X" items (*style* == `'x'`). This is like sticky z, but the items also don't pan horizontally.
- "Sticky Y" objects (*style* == `'y'`). This is like sticky z, but the items also don't pan vertically.
- "Sticky view" objects (*style* == `'view'`). This is like sticky z, but the items always stays within the view - and the constrained position is remembered, so that the object does not "want" to stay in its original position as it does with the other sticky types. Instead, once it is moved to stay within the view, the new position is its preferred position, and it sticks there.

`[55]-tags tagList`
(Available for all item types)

Specifies a set of tags to apply to the item. *TagList* consists of a list of tag names, which replace any existing tags for the item. *TagList* may be an empty list.

`[56]-text string`
(Available only for button, label, text, textfile, textfield types)

String specifies the characters to be displayed in the text item. Newline characters cause line breaks, and tab characters are supported. This option defaults to an empty string.

`[57]-timerrate rate`
(Available for all item types)

Specifies the frequency in milliseconds that the object's timerscript should be evaluated. If it is set to 0, the timer is turned off. Defaults to off (0). (see `-timerscript`).

`[58]-timerscript TclScript`
(Available for all item types)

Specifies a Tcl script that will be evaluated regularly, every rate milliseconds as specified by *-timerrate* (if *-timerrate* is greater than zero). This evaluation is independent of rendering and events. Returns the current TclScript for the object. (see *-timerrate*).

`[59]-title title`

(Available only for portal types)

If *title* is specified, then the portal will be rendered with a titlebar consisting of *title*. Otherwise, no title bar is drawn. Defaults to the empty string.

`[60]-to value`

(Available only for scrollbar types)

Specifies the ending (highest) value for a valuator widget to use. (Also see the **`-from`**, **`-linesize`** and **`-pagesize`** itemconfigure options.)

`[61]-transparency value`

(Available for all item types)

Specifies the transparency an item is drawn with. *value* must be a value between 0.0 and 1.0 where 0.0 is completely transparent and 1.0 is completely opaque. 1.0 is the default. If a portal or group is partially transparent, all of its member or visible objects, respectively, will have their transparency multiplied by the portals or groups.

`[62]-updatescript script`

(Available only for html types)

If *script* is specified, it gets evaluated when the html source has loaded, and then once every time an in-line is loaded. *script* is postpended with the id of the html object. This is necessary because the script is typically specified on the create line where the id of the html object is not yet known.

`[63]-url urlname`

(Available only for html, htmlanchor types)

Specifies the URL (Universal Resource Locator, or World-Wide Web address) that this html page should be accessed from. It must be specified with a valid address. Some examples are: "http://www.unm.edu", "http://www.cs.unm.edu/bederson", "file://nfs/u3/bederson/public_html/begin.html", "home-page.html".

`[64]-value value`

(Available only for scrollbar types)

Specifies the value of valuator widget. For instance, this specifies the position of the thumb on a scrollbar.

`[65]-view {x y zoom}`

(Available only for pad, portal types)

Specifies the location of this view. For top-level views (i.e., Pad++ surfaces), this changes the whole view. For portals, this changes the view within the portal. (x, y) specifes the point at the center of the view, and zoom specifies the magnification. For Pad++ surfaces, this defaults to (0, 0, 1). For portals, this defaults to directly under the location the portal was created at. (Also see the **`moveto`** command.)

`[66]-viewscript TclScript`

(Available for all item types)

Specifies a Tcl script that will be evaluated every time the view onto the Pad++ surface is changed. This script gets executed after the view coordinates have changed, but before the new scene gets rendered.

Returns the current viewscript.

`[67]-visiblelayers layers`
(Available only for pad, portal types)

Specifies what layers are visible within this portal. *layers* can be either a list of layers which will specify which items will be displayed within this portal, or take the special form of "all -layer1 -layer2 -layer3 ..." in which case all layers except the ones specified will be displayed. Defaults to "all". *layers* may also take the special value "`none`" which means that no layers are visible. (See the *-layer* itemconfigure option that all items have.)

`[68]-width width`
(Available for all item types)

By default, the width of every item is automatically computed based on its contents. If the `-width` option is set, however, then this overrides the automatically computed value. Most items are squashed or stretched to fit the specified *width*. Note that text and alias items, however, are clipped instead of being squashed or stretched. (Also see the `-height` itemconfigure option.)

`[69]-writeformat [copy | reference]`
(Available only for image types)

This option controls whether objects that are created from disk-based data are saved by storing a copy of all of the original data, or it references the original file. If the file is written as text, then the copy option to writeformat makes copies of the original datafiles so that there are multiple files created. The binary option results in writing the data in the output file.

`[70]-zoomaction {size growScript shrinkScript}`
(Available for all item types)

Specifies a pair of Tcl scripts that gets evaluated when an item grows or shrinks such that its size crosses the specified zoomaction size. This is a simple way of making "semantically zoomable" objects - that is, objects that look different when they are rendered at different sizes. When the item grows larger than *size*, *growScript* is evaluated, and when it shrinks smaller than size, *shrinkScript* is evaluated.

Any number of pairs of scripts may be associated with different sizes. Each use of *-zoomaction* may specify a different size, or modify scripts for an existing size. If both scripts are empty strings, then that zoomaction is deleted. This returns a list of zoomaction *size*, *growScript*, *shrinkScript* triplets.

Note that for a zoomaction to work, the item must get rendered on both sides of the size. It is possible to create an object and then immediately change its size before it gets rendered. In this case, the zoomaction will not get fired.

The script gets executed when the object normally would have been rendered. By default, the object will not get rendered. The script may call the **`renderitem`** function at any point to render the object. See the description of *-renderscript* for an example. The deletion of items during a zoomaction is delayed until after the current render is finished.

Here is an example that turns a rectangle into an image when it is zoomed in, and back into the rectangle when zoomed out:

```
proc grow {} {
   .pad ic rect -transparency 0
   .pad pushcoordframe rect
```

```
      set image_token [.pad image alloc images/unm_logo_orig.gif]
      .pad create image -image $image_token -anchor sw -tags "image"
      .pad popcoordframe
      .pad renderitem
   }

   proc shrink {} {
      .pad ic rect -transparency 1
      set image_id [.pad find withtag image]
      if {$image_id != ""} {
         set image_token [.pad ic image -image]
         .pad freeimage $image_token
         .pad delete image
      }
      .pad renderitem
   }

   proc testzoomaction {} {
      .pad create rectangle 0 0 341 222 -pen black -fill yellow3 \
         -zoomaction {250 grow shrink} -tags "rect"
   }
```

Alias Items

Items of type alias are items that mirror another existing item. They are separate items and have their own position, tags, event bindings, etc., but use the rendering of another item. Aliases are created with widget commands of the following form:

```
pathName create alias [option value option value ...]
```

Aliases refer to the item specified by the -reference option . If the reference is not specified, or is deleted, then the alias item is not rendered. Note that aliases are still somewhat buggy, and their behavior on groups of items is not guaranteed. The following options are supported for aliases:

`-reference [48]`	What item an alias references

Note that when the -width or -height of an alias set, the alias is clipped to those dimensions rather than being squashed or stretched as most items are.

Button Items

Button items are widgets that can be pressed and let go. If the pointer is over the button when the mouse button is released, an associated script will be fired. Buttons are created with widget commands of the following form:

```
pathName create button [option value option value ...]
```

Buttons are one of several widgets that are designed to mirror the functionality and usage of the standard widgets in Java's Abstract Windowing Toolkit (AWT). When buttons are created, they automatically get the tag "Button". Buttons have default event handlers which define their behavior. These event handlers are defined on the tag "Button" for the "Run" event mode. See the section on **Default Bindings** for more details about the event bindings.

The following options are supported for buttons:

`-command [14]`	Callback that is executed when button is pressed

`-fill` [23]	Specifies fill color of button
`-font` [24]	Specifies font to use for text
`-pen` [45]	Specifies pen color of button
`-relief [49]`	Specifies how border should be rendered (raised, flat, sunken, ridge, groove)
`-state [53]`	State of the button (normal, active, or disabled)
`-text [56]`	The text of the button

Canvas Items

Canvas items are widgets that are rendered as a simple rectangle, and their only purpose is to be derived from. From Tcl, this can be done with a -renderscript. There are no default event handlers. Canvases are created with widget commands of the following form:

```
pathName create canvas [option value option value ...]
```

Canvases are one of several widgets that are designed to mirror the functionality and usage of the standard widgets in Java's Abstract Windowing Toolkit (AWT). When buttons are created, they automatically get the tag "Canvas". Canvases have no default event handlers which define their behavior.

The following options are supported for canvases:

`-fill` [23]	Specifies fill color of canvas

Checkbox Items

Checkbox items are widgets that can be pressed and let go. If the pointer is over the checkbox when the mouse button is released, an associated script will be fired. Checkboxes maintain a binary state, and represent its state visually with a little box. Checkboxes are created with widget commands of the following form:

```
pathName create checkbox [option value option value ...]
```

Checkboxes are one of several widgets that are designed to mirror the functionality and usage of the standard widgets in Java's Abstract Windowing Toolkit (AWT). When checkboxes are created, they automatically get the tag "Checkbox". Checkboxes have default event handlers which define their behavior. These event handlers are defined on the tag "Checkbox" for the "Run" event mode. See the section on **Default Bindings** for more details about the event bindings.

The following options are supported for checkboxes:

`-command [14]`	Callback that is executed when checkbox is pressed
`-fill` [23]	Specifies fill color of checkbox
`-font` [24]	Specifies font to use for text
`-pen` [45]	Specifies pen color of checkbox
`-relief [49]`	Specifies how border should be rendered (raised, flat, sunken, ridge, groove)
`-state [53]`	State of the checkbox (normal, active, or disabled)
`-text [56]`	The text of the checkbox

Checkboxmenuitem Items

Checkboxmenutem items are widgets that are elements of pull-down or pop-up menus. When they are a member of a menu, they can be activated by moving the mouse over them and letting go. They maintain a binary state that is visually represented on the itme. When a checkboxmenuitem is activated, an associated script will be fired. Checkboxmenuitems are created with widget commands of the following form:

```
pathName create checkboxmenuitem [option value option value ...]
```

Checkboxmenuitems are one of several widgets that are designed to mirror the functionality and usage of the standard widgets in Java's Abstract Windowing Toolkit (AWT). When checkboxmenuitems are created, they automatically get the tag "CheckboxMenuItem". Checkboxmenuitems have default event handlers which define their behavior. These event handlers are defined on the tag "CheckboxMenuItem" for the "Run" event mode. See the section on **Default Bindings** for more details about the event bindings.

The following options are supported for checkboxmenuitems:

`-command [14]`	Callback that is executed when menuitem is pressed
`-fill [23]`	Specifies fill color of menuitem
`-font [24]`	Specifies font to use for text
`-pen [45]`	Specifies pen color of menuitem
`-relief [49]`	Specifies how border should be rendered (raised, flat, sunken, ridge, groove)
`-state [53]`	State of the menuitem normal, active, or disabled)
`-text [56]`	The text of the menuitem

See the documention for Menu and Choicemenu items for some example code that uses checkboxmenuitems.

Choicemenu Items

Choicemenu items are widgets that implement a pop-up menu. They contain a list of menuitems or checkboxmenuitems. When they are pressed, the member menuitems and checkboxmenuitems are displayed and may be selected. They always display the value of the currently selected menuitem or checkboxmenuitem. Choicemenus are created with widget commands of the following form:

```
pathName create choicemenu [option value option value ...]
```

Choicemenus are one of several widgets that are designed to mirror the functionality and usage of the standard widgets in Java's Abstract Windowing Toolkit (AWT). When choicemenus are created, they automatically get the tag "ChoiceMenu". Choicemenus have default event handlers which define their behavior. These event handlers are defined on the tag "ChoiceMenu" for the "Run" event mode. See the section on **Default Bindings** for more details about the event bindings.

The following options are supported for choicemenus:

`-command [14]`	Callback that is executed when choicemenu is pressed
`-fill [23]`	Specifies fill color of choicemenu
`-font [24]`	Specifies font to use for text
`-members [38]`	The list of members of a choicemenu
`-pen [45]`	Specifies pen color of choicemenu
`-relief [49]`	Specifies how border should be rendered (raised, flat, sunken, ridge, groove)
`-state [53]`	State of the choicemenu (normal, active, or disabled)
`-text [56]`	The text of the choicemenu

The following example shows how a pop-up menu can be created.

```
set c1 [.pad create menuitem -text "Times"]
set c2 [.pad create menuitem -text "Helvetica"]
set c3 [.pad create menuitem -text "Courier"]
.pad create choicemenu -members "$c1 $c2 $c3" -text "Font"
```

Frame Items

Frame items are widgets that act like top-level windows within Pad++. They are used to group a collection of items. They are similar to panels, except they window dressing that is used to manipulate the frame. Frames are created with widget commands of the following form:

```
pathName create frame [[x1 y1 x2 y2] option value option value ...]
```

Frames are one of several widgets that are designed to mirror the functionality and usage of the standard widgets in Java's Abstract Windowing Toolkit (AWT). When frames are created, they automatically get the tag "Frame". Frames have default event handlers which define their behavior. These event handlers are defined on the tag "Frame" for the "Run" event mode. See the section on **Default Bindings** for more details about the event bindings.

Unlike group items, frames do not set their size based on their contents. Rather, they are fixed size as specified by the command line coordinates, or by the **`-width`** [68] and **`-height`** [26] itemconfigure options. Frames have their own coordinate system where (0, 0) specifies the panels lower left corner. Adding items to a frame adds them relative to the frame's coordinate system.

The frame window dressing gives a pseudo-3D titlebar and border which can be used to move and resize the frame.

The following options are supported for frames:

`-divisible` [16]	True if events go through the frame to its members
`-fill` [23]	Specifies fill color
`-members` [38]	The list of members of the frame
`-menubar` [40]	Menubar associated with a frame

Also, see the **`addgroupmember` [2]** and **`removegroupmember` [72]** commands that can be used to add and remove items from the frame.

Grid Items

Items of type grid arrange one or more items in rows and columns and treats them as a group. It is based on the Tk grid geometry manager and its behavior and Tcl syntax are very similar to it. In pad, all manipulations of a grid once it is created are affected through the **`grid`** sub-command. Note that rows and columns start from the top left corner of the grid (as in the Tk grid). The complete grid sub-command is described in this section.

Grids are created with widget commands of the following form:

```
pathName create grid [slaves...]
```

Grid creation is slightly different from creation of other pad objects. Instead of the normal command-line option-value pairs a list of slaves and their grid configuration can be specified (see the section below on sub-commands and slave configuration). Grids are special group objects and inherit much of the group functionality and support the "-divisible" option which can be set (using itemconfigure) once the grid is created:

`-divisible` [16]	True if events go through a group to its members

The syntax of the grid sub-command is:

```
pathName grid slave [slave...] option value [option value...]
pathName grid command arg [arg...]
```

If the first argument of the grid command is a slave object then the remainder of the command line is processed in the same way as the grid configure command. The "-in" option can be used to add a slave to a grid. The following grid sub-commands are allowed:

```
$PAD grid arrange master
```

Forces arrangement of the given grid. Any pending layout request for the grid is removed. This can be useful when an application has done several grid configuration and wants them to take effect immediately. Normally, grid arrangement is done at "idle" times.

```
$PAD grid bbox master column row
```

The bounding box (in pixels) is returned for the space occupied by the grid position indicated by column and row. The return value consists of 4 integers. The first two are the pixel offset from the master window (x then y) of the top-left corner of the grid cell, and the second two are the width and height of the cell.

```
$PAD grid columnconfigure master index [-option value...]
```

Query or set the column properties of the index column of the geometry master, master. The valid options are -minsize and -weight. The -minsize option sets the minimum column size, in screen units, and the -weight option (a floating point value) sets the relative weight for apportioning any extra spaces among columns. If no value is specified, the current value is returned.

```
$PAD grid configure slave [slave ...] [options]
```

The arguments consist of one or more slaves followed by pairs of arguments that specify how to manage the slaves. The characters -, x and ^, can be specified instead of a window name to alter the default location of a slave, as described in the ``RELATIVE PLACEMENT'' section, below. If any of the slaves are already managed by the grid then any unspecified options for them retain their previous values rather than receiving default values. The following options are supported:

-column n
: Insert the slave so that it occupies the nth column in the grid. Column numbers start with 0. If this option is not supplied, then the slave is arranged just to the right of previous slave specified on this call to grid, or column "0" if it is the first slave. For each x that immediately precedes the slave, the column position is incremented by one. Thus the x represents a blank column for this row in the grid.

-columnspan n

: Insert the slave so that it occupies n columns in the grid. The default is one column, unless the slave is followed by a -, in which case the columnspan is incremented once for each immediately following -.

-in other

: Insert the slave(s) in the grid object given by other (which must be an existing grid).

-padx amount
: The amount specifies how much horizontal external padding to leave on each side of the slave(s). The amount defaults to 0.

-pady amount
: The amount specifies how much vertical external padding to leave on the top and bottom of the slave(s). The amount defaults to 0.

-row n
: Insert the slave so that it occupies the nth row in the grid. Row numbers start with 0. If this option is not supplied, then the slave is arranged on the same row as the previous slave specified on this call to grid, or the first unoccupied row if this is the first slave.

-rowspan n
: Insert the slave so that it occupies n rows in the grid. The default is one row. If the next grid

command contains ^ characters instead of slaves that line up with the columns of this slave, then the rowspan of this slave is extended by one.

-sticky style

If a slave's parcel is larger than its requested dimensions, this option may be used to position (or stretch) the slave within its cavity. Style is a string that contains zero or more of the characters n, s, e or w. The string can optionally contains spaces or commas, but they are ignored. Each letter refers to a side (north, south, east, or west) that the slave will "stick" to. If both n and s (or e and w) are specified, the slave will be stretched to fill the entire height (or width) of its cavity. The sticky option subsumes the combination of -anchor and -fill that is used by pack. The default is { }, which causes the slave to be centered in its cavity, at its requested size.

```
$PAD grid forget slave [slave ...]
```

Removes each of the slaves from their grid.

```
$PAD grid info slave
```

Returns a list whose elements are the current configuration state of the slave given by slave in the same option-value form that might be specified to grid configure. The first two elements of the list are ``-in master'' where master is the slave's master.

```
$PAD grid location master x y
```

Given x and y values in screen units relative to the master object, the column and row number at that x and y location is returned. For locations that are above or to the left of the grid, -1 is returned.

```
$PAD grid rowconfigure master index [-option value...]
```

Query or set the row properties of the index row of the geometry master, master. The valid options are -minsize and -weight. Minsize sets the minimum row size, in screen units, and weight sets the relative weight for apportioning any extra spaces among rows. If no value is specified, the current value is returned.

```
$PAD grid size master
```

Returns the size of the grid (in columns then rows) for master. The size is determined either by the slave occupying the largest row or column, or the largest column or row with a minsize or weight.

```
$PAD grid slaves master [-option value]
```

If no options are supplied, a list of all of the slaves in master are returned. Option can be either -row or -column which causes only the slaves in the row (or column) specified by value to be returned.

Relative Placement

The grid command contains a limited set of capabilities that permit layouts to be created without specifying the row and column information for each slave. This permits slaves to be rearranged, added, or removed without the need to explicitly specify row and column information.

When no column or row information is specified for a slave, default values are chosen for column, row, columnspan and rowspan at the time the slave is managed. The values are chosen based upon the current layout of the grid, the position of the slave relative to other slaves in the same grid command, and the presence of the characters -, ^, and ^ in grid command where slave names are normally expected.

- This increases the columnspan of the slave to the left. Several -'s in a row will successively increase the columnspan. S - may not follow a ^ or a x.
x This leaves an empty column between the slave on the left and the slave on the right.
^ This extends the rowspan of the slave above the ^'s in the grid. The number of ^'s in a row must match the number of columns spanned by the slave above it.

Restrictions on Master Windows

In pad, the master for each slave is the slave's parent (which is a grid object). This means if an object belongs to an existing group then it cannot be added to a grid.

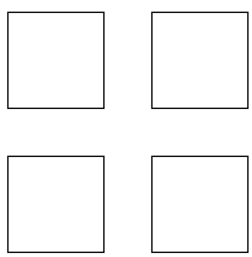
Differences Between Pad++ and TK Grid Commands

- The *-ipadx* and *-ipady* grid item configuration options are not available in pad.
- Master window geometry propagation flag is not available in pad.
- The parent-child and stacking restrictions and rules for master and slave items are not supported in pad (slaves can only be in the master group).
- If the grid is not positioned then it places itself around its first item. Once all grid items have been positioned the grid bounding box will be computed to enclose them all.
- Added the **`arrange`** command for forcing grid arrangement.
- Items that are removed from grids are not unmapped.

Examples

1) put four objects in a 2x2 grid with 10 pixels horizontal and vertical pading:

```
set obj1 [.pad create rectangle 0 0 50 50]
set obj2 [.pad create rectangle 50 50 100 100]
set obj3 [.pad create rectangle 100 100 150 150]
set obj4 [.pad create rectangle 150 150 200 200]
set thegrid [.pad create grid $obj1 $obj2 -padx 10 -pady 10]
.pad grid $obj3 $obj4 -in $thegrid -row 1 -padx 10 -pady 10
```



2) read objects from pad files in a directory and place them in a Nx2 grid (this can be useful for creating palettes):

```
proc read_files {PAD dir} {
   set objs ""
            # Go though list of files
   foreach file [glob $dir/*.pad] {
            # Read file and put all its object in a group (Pad_ObjectList will be
            # set to list of objects read from file).
       $PAD read $file
       set group [$PAD create group -members $Pad_ObjectList]
       lappend objs $group
   }
   return $objs
}
```

```
proc create_palette {PAD objs} {
            # Create the grid object
   set thegrid [$PAD create grid]
   set row 0
   set col 0

            # Go through objects and place them two per row
   foreach obj $objs {
            # Add obj to the grid
      $PAD grid $obj -in $thegrid -row $row -column $col -padx 10 -pady 5

            # Set row and column position for next object
      if {$col == 0} {
         incr col
      } else {
         set col 0
         incr row
      }
   }

            # Have the grid arrange itself now
   $PAD grid arrange $thegrid

   return $thegrid
}

create_palette .pad [read_files .pad $env(PADHOME)/draw/scrapbook]
```

Alternatively,

```
proc create_palette {PAD objs} {
   # create the grid object
   set thegrid [$PAD create grid]

   # go through list of objects and place them two per row
   set numobjs [llength $objs]
   for {set i 0} {$i < $numobjs} {incr i 2} {
      set obj1 [lindex $objs $i]
      if {$i < [expr $numobjs-1]} {
         set obj2 [lindex $objs [expr $i+1]]
      } else {
         set obj2 ""
      }
      $PAD grid $obj1 $obj2 -in $thegrid -padx 10 -pady 5
   }

   $PAD grid arrange $thegrid
   return $thegrid
}

create_palette .pad [read_files .pad $env(PADHOME)/draw/scrapbook]
```

3) Draw horizontal and vertical grid lines and a bounding rectangle for an existing grid. Make a group for the line objects and the existing grid. Assume the grid is a normal MxN table (i.e. all rows have N columns and all columns have M rows).

```
proc create_gridlines { PAD thegrid } {
               # Get bounding box, width and height and location of the grid
   set gbbox [$PAD bbox $thegrid]
```

```
    set gwidth [expr [lindex $gbbox 2] - [lindex $gbbox 0]]
    set gheight [expr [lindex $gbbox 3] - [lindex $gbbox 1]]
    set gx [lindex $gbbox 0]
    set gy [lindex $gbbox 1]

                  # Get number of rows and columns
    set numrows [lindex [$PAD grid size $thegrid] 1]
    set numcols [lindex [$PAD grid size $thegrid] 0]

                  # Create the bounding rectangle
    set grect [eval $PAD create rectangle $gbbox]

    set items "$grect"
    set scale [$PAD scale $thegrid]

                  # Create horizontal lines by looking at the <r, 0> grid elemments.
    for {set r 1} {$r < $numrows} {incr r} {
                  # Get location of the <r, 0> element (including padding)
       set rinfo [$PAD grid bbox $thegrid 0 $r]
       set x1 [expr [lindex $rinfo 0]*$scale + $gx]
                  # Transform the y coord for pad (grid's is from top left corner)
       set y1 [expr ($gheight - [lindex $rinfo 1]*$scale) + $gy]
       set x2 [expr $x1 + $gwidth]
       set y2 $y1
       lappend items [$PAD create line $x1 $y1 $x2 $y2 -tags gridrowline_$thegrid]
    }

                  # Draw vertical lines by looking at the <0, c> elements
    for {set c 1} {$c < $numcols} {incr c} {
       set cinfo [$PAD grid bbox $thegrid $c 0]
       set x1 [expr [lindex $cinfo 0]*$scale + $gx]
       set y1 [expr ($gheight - [lindex $cinfo 1]*$scale) + $gy]
       set x2 $x1
       set y2 [expr $y1 - $gheight]
       lappend items [$PAD create line $x1 $y1 $x2 $y2 -tags gridcolline_$thegrid]
    }

                  # Create a group for all the grid lines
    set glines [$PAD create group -members $items -divisible 0 \
                   -tags gridlines_$thegrid]

                  # Create a group for the lines and the grid
    set newgrp [$PAD create group -members "$glines $thegrid" -tags grid_$thegrid \
                  -divisible 1]

    return $newgrp
}

set thegrid [create_palette .pad [read_files .pad ./draw/scrapbook]]
create_gridlines .pad $thegrid
```

Group Items

Items of type group are special items that group other items. Group items do not have any visual appearance, but rather are used just for creating structure. Groups are implemented very efficiently, and may be hierarchical (i.e., contain other groups). Modifying the position of a group implicitly affects all of the members of the group, recursively. Pad++ also supports "tags" which are implicit way of grouping items - but this only works for events.

That is, giving several items the same tag allows them all to respond to the same event handlers. Groups explicitly bring items together. Group members are rendered sequentially in the display list. That is, no other objects can appear inbetween group members - they are always above or below all the group members. Raising or lowering a group object raises or lowers all the group members. Raising or lowering a group member raises or lowers the member within the group.

Groups automatically resize themselves to contain all of their members - thus adding, removing, or repositioning a member implicitly changes the size of the group. See the pad **`addgroupmember`** and **`removegroupmember`** commands and the *-member* itemconfigure option below for setting group membership, and the **`getgroup`** command for testing group membership.

When an event hits a group, it normally passes through the group object to its members. However, it is possible to configure a group object so that it grabs the events and does not pass them through. See the *-divisible* flag.

Groups are created with widget commands of the following form:

```
pathName create group [option value option value ...]
```

There may be any number of option-value pairs, each of which sets one of the configuration options for the item. These same option-value pairs may be used in itemconfigure widget commands to change the item's configuration. The following options are supported for groups:

`-divisible` [16]	True if events go through a group to its members
`-members [38]`	The list of members of a group

HTML Items

Items of type html are compound items representing the specified html file. (HTML is HyperText Markup Language. Based on SGML, HTML is most commonly known as the language describing items for the World-Wide Web.) HTML items know about the internet and will automatically fetch a file from a URL (Universal Resource Locator) as well as in-line images. URL's may also specify local files. When the html data is fetched, it is parsed and the HTML item is created which contains a method for rendering the page. HTML anchors are created as separate items which may have events bound to them. HTML items are an extension of **`group`** items, and thus have several of the same options as groups.

There is a Tcl file (*draw/html.tcl*) which describes default event bindings for html items which follow hyperlinks, and lay them out with scale. See the end of the description of HTML items for a description of html anchors.

HTML items are created with widget commands of the following form:

```
pathName create html [option value option value ...]
```

There may be any number of option-value pairs, each of which sets one of the configuration options for the item. These same option-value pairs may be used in itemconfigure widget commands to change the item's configuration. The following options are supported for html items:

`-border [10]`	Specifies border color of item
`-borderwidth` [11]	Specifies width of border
`-divisible` [16]	True if events go through a group to its members
`-donescript [17]`	A script to evaluate when a background action has completed
`-errorscript` [19]	A script to evaluate when a background action has an error
`-fill` [23]	Specifies fill color of item
`-font` [24]	Specifies font to use for text
`-htmlanchors [28]`	The anchors associated with an HTML page
`-members [38]`	The list of members of a group
`-updatescript` [62]	A script to evaluate when a background action has made progress
`-url [63]`	The URL associated with an item

`Note that` when the width of an html page is changed, the page is re-laid out, and the height of the page could change as a result.

HTML Anchors

The anchors are special Pad++ items of type "htmlanchor". They are automatically grouped with the HTML object. As such, they can not be deleted independently, and are automatically deleted when the html object they are associated with is deleted. Some anchors have multiple components (i.e., and image and some text). In this case, they all have the same URL, and changing the pen color of one component automatically changes the pen color of the other components.

Anchors may be configured with the itemconfigure command. The following options are supported for html anchors:

`-html` [27]	The HTML item associated with an htmlanchor
`-image [29]`	Image data associated with item
`-ismap [31]`	True if an htmlanchor is an image map
`-state [53]`	State of an item (such as visited, unvisited, etc.)
`-url [63]`	The URL associated with an item

Image Items

Items of type image appear on the display as color images. Images are created with widget commands of the following form:

```
pathName create image [option value option value ...]
```

There may be any number of option-value pairs, each of which sets one of the configuration options for the item. These same option-value pairs may be used in itemconfigure widget commands to change the item's configuration. The following options are supported for images:

`-dither` [15]	Render with dithering
`-image [29]`	Image data associated with item (allocated by image alloc)
`-writeformat [69]`	Controls whether disk-based item is written out by copy or reference

KPL Items

Items of type Kpl provide a method for creating an item with a user-described render method. Sometimes the Pad++ items available do not have exactly what you want, or you'd like a complex item consisting of several primitives. Rather than create several different Pad++ items and group them together, a single Kpl item can be created with a kind of display list.

Kpl is a language (designed at New York University by Ken Perlin, et. al.) that is very simple, but extremely fast. It is the best language we found for writing interpreted code for rendering quickly. In fact, Kpl has a byte-compiler which makes it faster. Some simple experiments have shown it to be roughly 15 times slower than C for simple math (compared to tcl which is typically about 1,000 times slower than C). Because Kpl is a general-purpose language, it can be used for on-the-fly calculations as well as render calls. Pad++ supplies several render that available through Kpl that allow a Kpl object to render fairly complex objects.

Kpl is a stack-based post-fix language (much like PostScript). Some basic documentation is available with the Pad++ release in *doc/kpl.troff*. See the section in this document on the KPL-PAD++ INTERFACE for a description of how to access Kpl through Pad++, and what Pad++ routines are available from Kpl.

Kpl items are created with widget commands of the following form:

```
pathName create kpl [option value option value ...]
```

There may be any number of option-value pairs, each of which sets one of the configuration options for the item. These same option-value pairs may be used in itemconfigure widget commands to change the item's configuration. The following special options are supported for kpl objects:

`-bb [9]`	A KPL script that gets evaluated to specify the bounding box of an item

Note that all coordinates in Kpl are specified in pixels, and not in the current Pad++ units. An example follows that creates a Kpl item that draws a brown triangle. In this case, the Kpl code is stored in the file triangle.kpl.

```
         # Tcl code to load Kpl code and to create
         # Pad++ Kpl item that draws a brown triangle
kpl eval 'triangle.kpl source
set pen [.pad alloccolor brown]
.pad create kpl -bb {-10:-10 110:110} -renderscript {draw_triangle}


         /* Kpl code (in a separate file)
            to draw a brown triangle */
{
'pen tcl_get -> Pen
Pen setcolor
3 setlinewidth
newpath
  0:0 moveto
  100:0 lineto
  50:100 lineto
  0:0 lineto
stroke
} -> draw_triangle
```

Label Items

Label items are widgets that simply display some text with a background color. They have no behavior. Labels are created with widget commands of the following form:

```
pathName create label [option value option value ...]
```

Labels are one of several widgets that are designed to mirror the functionality and usage of the standard widgets in Java’s Abstract Windowing Toolkit (AWT). When labels are created, they automatically get the tag "Label".

The following options are supported for labels:

`-fill` [23]	Specifies fill color of label
`-font` [24]	Specifies font to use for text
`-pen` [45]	Specifies pen color of label
`-text [56]`	The text of the label

Line Items

Items of type line appear on the display as one or more connected line segments. Lines are created with widget commands of the following form:

pathName create line [x_1 y_1... x_n y_n [option value option value ...]]

The arguments x_1 through y_n give the coordinates for a series of two or more points that describe a series of connected line segments. After the coordinates there may be any number of option-value pairs, each of which sets one of the configuration options for the item. These same option-value pairs may be used in itemconfigure widget commands to change the item's configuration. If a line is created without any points, it will not be rendered until some points are added with the **coords** command. The following options are supported for lines:

`-arrow` [7]	Whether to draw arrow heads with this item
`-arrowshape [8]`	The shape of drawn arrow heads
`-capstyle [12]`	Specifies how to draw line ends
`-joinstyle [32]`	Specifies how to draw the joints within multi-point lines
`-noisedata` [42]	Specifies parameters to render item with noise
`-pen` [45]	Specifies pen color of item
`-penwidth` [46]	Specifies width of pen

Menu Items

Menu items are widgets that are elements of menubar and are used to implement a pull-down menu. They contain a list of menuitems or checkboxmenuitems. When they are pressed, the member menuitems and checkboxmenuitems are displayed and may be selected. Menus are created with widget commands of the following form:

```
pathName create menu [option value option value ...]
```

Menus are one of several widgets that are designed to mirror the functionality and usage of the standard widgets in Java's Abstract Windowing Toolkit (AWT). When menus are created, they automatically get the tag "Menu". Menus have default event handlers which define their behavior. These event handlers are defined on the tag "Menus" for the "Run" event mode. See the section on **Default Bindings** for more details about the event bindings.

The following options are supported for menus:

`-fill` [23]	Specifies fill color of menu
`-font` [24]	Specifies font to use for text
`-members [38]`	The list of members of a menu
`-pen` [45]	Specifies pen color of menu
`-relief [49]`	Specifies how border should be rendered (raised, flat, sunken, ridge, groove)
`-state [53]`	State of the menu (normal, active, or disabled)
`-text [56]`	The text of the menu

The following example shows how several pull-down menus can be created that consist of menuitems and checkboxmenuitems. The menus that are created are then put in a menubar.

```
set f1 [.pad create menuitem -text "New..."]
set f2 [.pad create menuitem -text "Open..."]
set f3 [.pad create menuitem -text "Save"]
set f4 [.pad create menuitem -text "Save As"]
set f5 [.pad create menuitem -text "Exit"]

set e1 [.pad create checkboxmenuitem -text "Cut"]
set e2 [.pad create menuitem -text "Copy"]
set e3 [.pad create menuitem -text "Paste"]

set g1 [.pad create menuitem -text "Content"]
set g2 [.pad create menuitem -text "Index"]
set g3 [.pad create menuitem -text "About"]

set m1 [.pad create menu -members "$f1 $f2 $f3 $f4 $f5" -text "File"]
set m2 [.pad create menu -members "$e1 $e2 $e3" -text "Edit"]
set m3 [.pad create menu -members "$g1 $g2 $g3" -text "Help"]
```

```
.pad create menubar -members "$m1 $m2 $m3" -height 30
```

Menubar Items

Menubar items are widgets that define a pull-down menu. They contain a list of menus. When the consituent menus are pressed, their member menuitems and checkboxmenuitems are displayed and may be selected. Menubars are created with widget commands of the following form:

```
pathName create menubar [option value option value ...]
```

Menubars are one of several widgets that are designed to mirror the functionality and usage of the standard widgets in Java's Abstract Windowing Toolkit (AWT). When menubars are created, they automatically get the tag "MenuBar". Menubars have no default event handlers which define their behavior.

The following options are supported for menubars:

`-fill` [23]	Specifies fill color of menu
`-members [38]`	The list of members of a menu

See the documention for Menu items for some example code that uses menubars.

Menuitem Items

Menutem items are widgets that are elements of pull-down or pop-up menus. When they are a member of a menu, they can be activated by moving the mouse over them and letting go. When a menuitem is activated, an associated script will be fired. Menuitems are created with widget commands of the following form:

```
pathName create menuitem [option value option value ...]
```

Menuitems are one of several widgets that are designed to mirror the functionality and usage of the standard widgets in Java's Abstract Windowing Toolkit (AWT). When menuitems are created, they automatically get the tag "MenuItem". Menuitems have default event handlers which define their behavior. These event handlers are defined on the tag "MenuItem" for the "Run" event mode. See the section on **Default Bindings** for more details about the event bindings.

The following options are supported for menuitems:

`-command [14]`	Callback that is executed when menuitem is pressed
`-fill` [23]	Specifies fill color of menuitem
`-font` [24]	Specifies font to use for text
`-pen` [45]	Specifies pen color of menuitem
`-relief [49]`	Specifies how border should be rendered (raised, flat, sunken, ridge, groove)
`-state [53]`	State of the menuitem normal, active, or disabled)
`-text [56]`	The text of the menuitem

See the documention for Menu and Choicemenu items for some example code that uses menuitems.

Oval Items

Items of type oval appear as ovals on the display. Each oval may have an outline (pen color), a fill, or both. Ovals are created with widget commands of the following form:

```
pathName create oval [x1 y1 x2 y2 [option value option value ...]]
```

The arguments x_1, y_1, x_2, and y_2 give the coordinates of two diagonally opposite corners of the oval. If an oval is created without any points, it will not be rendered until some points are added with the **coords** command. After the coordinates there may be any number of option-value pairs, each of which sets one of the configuration options for the item. These same option-value pairs may be used in itemconfigure widget commands to change the item's configuration. The following options are supported for ovals:

`-fill` [23]	Specifies fill color of item
`-joinstyle [32]`	Specifies how to draw the joints within multi-point lines
`-pen` [45]	Specifies pen color of item
`-penwidth` [46]	Specifies width of pen

Pad Items

Each pad widget implicitly defines a special "pad" item which always has the id "1". This is a special item which can get events and has a few itemconfigure options. It may not be explicitly created or deleted. The valid options are:

`-view [65]`	Specifies the view this item sees
`-visiblelayers [67]`	The layers that are visible within this view (just for portals and surface, item #1)

Panel Items

Panel items are widgets that are used to group a collection of items. They are similar to groups, except they have a background color, and they are fixed size. Panels are created with widget commands of the following form:

pathName create panel [[x_1 y_1 x_2 y_2] option value option value ...]

Panels are one of several widgets that are designed to mirror the functionality and usage of the standard widgets in Java’s Abstract Windowing Toolkit (AWT). When panels are created, they automatically get the tag "Panel".

Unlike group items, panels do not set their size based on their contents. Rather, they are fixed size as specified by the command line coordinates, or by the **-width** [68] and **-height** [26] itemconfigure options. Panels have their own coordinate system where (0, 0) specifies the panels lower left corner. Adding items to a panel adds them relative to the panel’s coordinate system.

The following options are supported for panels:

`-divisible` [16]	True if events go through the panel to its members
`-fill` [23]	Specifies fill color
`-members [38]`	The list of members of the panel

Also, see the **addgroupmember [2]** and **removegroupmember [72]** commands that can be used to add and remove items from the panel.

Polygon Items

Items of type polygon appear as polygonal regions on the display. Each polygon may have an outline (pen color), a fill, or both. Polygon are created with widget commands of the following form:

pathName create polygon [x_1 y_1... x_n y_n [option value option value ...]]

The arguments x_1, y_1, ..., x_n, and y_n specify the coordinates of the vertices of the polygon. If a polygon is created without any points, it will not be rendered until some points are added with the **coords** command. After the coordinates there may be any number of option-value pairs, each of which sets one of the configuration options for the item. These same option-value pairs may be used in itemconfigure widget commands to change the item's configuration. The following options are supported for polygons:

`-fill` [23]	Specifies fill color of item
`-joinstyle [32]`	Specifies how to draw the joints within multi-point lines
`-pen` [45]	Specifies pen color of item
`-penwidth` [46]	Specifies width of pen

Portal Items

Portals are a special type of item in Pad++ that sit on the Pad++ surface with a view onto a different location. Because each portal has its own view, a surface might be visible at several locations, each at a different magnification, through various portals. In addition, portals can look onto surfaces of other Pad++ widgets. The surface that the portal is looking onto is called that portal's *lookon*. Portal items are created with widget commands of the following form:

```
pathName create portal [x1 y1 x2 y2 ... [option value option value ...]]
```

If two points are specified, then the portal will be rectangular where those two points specify the lower left and upper right coordinates of the portal. If more than two points are specified, then the portal will be polygonal shaped by those points. If a portal is created without any points, it will not be rendered until some points are added with the **`coords`** command. There may be any number of option-value pairs, each of which sets one of the configuration options for the item. These same option-value pairs may be used in itemconfigure widget commands to change the item's configuration. The following options are supported for text items:

`-border [10]`	Specifies border color of item
`-borderwidth` [11]	Specifies width of border
`-fill` [23]	Specifies fill color of item
`-font` [24]	Specifies font to use for text
`-lookon [36]`	Specifies the pad widget this item sees
`-pen` [45]	Specifies pen color of item
`-relief [49]`	Specifies how border should be rendered (raised, flat, sunken, ridge, groove)
`-title [59]`	Some items only: Title of an item
`-view [65]`	Specifies the view this item sees
`-visiblelayers [67]`	The layers that are visible within this view (just for portals and surface, item #1)

Note that it is impossible to directly change an item's parameters when it is viewed within a portal. That is, you can not have an object that has a -minsize of 20% in the top-level view, but a `-minsize` of 0% within a portal. One (inelegant) workaround to this is to use an alias. You could fmake an alias of the original object and put it in a different place. Put what ever min/maxsize you want on the alias, and have the portal look onto the alias.

Rectangle Items

Items of type rectangle appear as rectangular regions on the display. Each rectangle may have an outline (pen color), a fill, or both. Rectangles are created with widget commands of the following form:

```
pathName create rectangle [x1 y1 x2 y2 [option value option value ...]]
```

The arguments x_1, y_1, x_2, and y_2 give the coordinates of two diagonally opposite corners of the rectangle If a rectangle is created without any points, it will not be rendered until some points are added with the **`coords`** command. After the coordinates there may be any number of option-value pairs, each of which sets one of the configuration options for the item. These same option-value pairs may be used in itemconfigure widget commands to change the item's configuration. The following options are supported for rectangles:

`-fill` [23]	Specifies fill color of item
`-joinstyle [32]`	Specifies how to draw the joints within multi-point lines
`-pen` [45]	Specifies pen color of item

`-penwidth` [46]	Specifies width of pen

Scrollbar Items

Scrollbar items are widgets that are used to interactively select a numeric value within a range. Whenever the value is changed, an associated script will be fired. Scrollbars are created with widget commands of the following form:

```
pathName create scrollbar [option value option value ...]
```

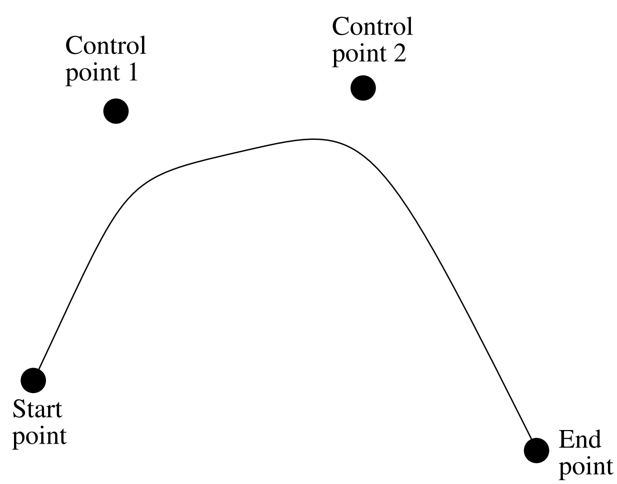
Scrollbars are one of several widgets that are designed to mirror the functionality and usage of the standard widgets in Java's Abstract Windowing Toolkit (AWT). When scrollbars are created, they automatically get the tag "Scrollbar". Scrollbars have default event handlers which define their behavior. These event handlers are defined on the tag "Scrollbar" for the "Run" event mode. See the section on **Default Bindings** for more details about the event bindings.

The following options are supported for scrollbars:

`-command [14]`	Callback that is executed when scrollbar value is changed
`-fill` [23]	Specifies fill color of scrollbar
`-from [25]`	Smallest value that scrollbar takes
`-linesize [34]`	Amount scrollbar should change to represent a line change
`-orientation [43]`	Orientation of scrollbar (horizontal or vertical.)
`-pagesize [44]`	Amount scrollbar should change to represent a page change
`-to [60]`	Largest value that scrollbar takes
`-value [64]`	Current value of scrollbar

Spline Items

Items of type spline appear on the display as one or more bezier curves joined end to end, so the last point of the one curve is used as the first point of the next. Splines are displayed as smooth curves at any magnification. They are rendered in more detail when they are larger. It is possible to create a fixed approximation to a spline with the **`spline2line`** command. In addition, it is possible to generate a spline that approximates a multi-segmented line with the **`line2spline`** command. A bezier curve is defined using four points - the start and end point for the curve, and two control points that indicate the path that the curve follows. For example:



For a spline made from a single bezier segment, the points are given as follows:

<start-x> <start-y> <c1-x> <c1-y> <c2-x> <c2-y> <end-x> <end-y>

That is, first the start point is given, followed by the first control point, followed by the second control point and finishing with the end point for the curve. For example, you can create a simple spline using:

.pad create spline 0 0 10 10 20 10 30 0

here (0, 0) defines the start of the curve. (10, 10) is the first control point, (20, 10) is the second control point, and the curve ends at (30, 0).

Splines are created with widget commands of the following form:

`pathName create spline` x_1 y_1... x_n y_n `[option value option value ...]`

The arguments x_1 through y_n give the coordinates for a series of one or more splines. Each point is specified by two coordinates. When specifying a spline made from two or more bezier curves, the end point of the first curve is used as the start point for the second, so the second curve only requires an additional three points (two control points and an end point). In general a spline of N bezier curves requires 3N+1 points (6N+2 coordinates). This represents a start point and then three points for each curve.

For convenience, if the end point of the last curve segment in a spline is omitted, Pad++ assumes that the curve should be 'closed' - it uses the start point of the first curve as the end point for the last curve, creating a closed shape. For closed shapes, therefore, you should provide 3N points (6N coordinates).

After the coordinates there may be any number of option-value pairs, each of which sets one of the configuration options for the item. These same option-value pairs may be used in itemconfigure widget commands to change the item's configuration. The following options are supported for lines:

`-arrow` [7]	Whether to draw arrow heads with this item
`-arrowshape [8]`	The shape of drawn arrow heads
`-capstyle [12]`	Specifies how to draw line ends
`-joinstyle [32]`	Specifies how to draw the joints within multi-point lines
`-pen` [45]	Specifies pen color of item
`-penwidth` [46]	Specifies width of pen

TCL Items

Items of type tcl are really a simple of way of having user-describable item. A Tcl item really consists of two Tcl scripts to render an item procedurally (one to render, and the other to compute the bounding box.) The render script can render by calling the pad widget with the various drawing routines (see **`drawline`**, **`drawtext`**, **`setcolor`**, **`setlinewidth`**.) Tcl's are created with widget commands of the following form:

`pathName create tcl [option value option value ...]`

There may be any number of option-value pairs, each of which sets one of the configuration options for the item. These same option-value pairs may be used in itemconfigure widget commands to change the item's configuration. The following options are supported for tcl objects:

`-bb [9]`	A script that gets evaluated to specify the bounding box of an item

Text Items

A text item displays a string of characters on the screen in one or more lines. There is a single custom "vector" font. Text items are created at a default size of one pixel high. Their size can be changed with the **`scale`** command or the *-position* itemconfigure option.

INDICES

Many of the commands for text take one or more indices as arguments. An index is a string used to indicate a particular place within a text, such as a place to insert characters or one endpoint of a range of characters to delete. Indices have the syntax:

```
base modifier modifier modifier ...
```

Where *base* gives a starting point and the modifiers adjust the index from the starting point (e.g. move forward or backward one character). Every index must contain a base, but the modifiers are optional.

The base for an index must have one of the following forms:

`line.char`

Indicates *char*'th character on line *line*. Lines are numbered from 0. Notice that this is different than the Tk text widget. Within a line, characters are numbered from 0.

`line.end`

Indicates the last character on line *line*. Lines are numbered from 0.

`char`

Indicates the *char*'th character from the beginning of the file (starting at 0).

`@x,y`

Indicates the character that covers the pixel whose *x* and *y* coordinates within the text's window are *x* and *y*.

`end`

Indicates the last character in the text.

`mark`

Indicates the character just after the mark whose name is *mark*.

If modifiers follow the base index, each one of them must have one of the forms listed below. Keywords such as chars and wordend may be abbreviated as long as the abbreviation is unambiguous. Modifiers must have one of the following forms:

`+ count chars`

Adjust the index forward by count characters, moving to later lines in the text if necessary. If there are fewer than count characters in the text after the current index, then set the index to the last character in the text. Spaces on either side of count are optional.

`- count chars`

Adjust the index backward by count characters, moving to earlier lines in the text if necessary. If there are fewer than count characters in the text before the current index, then set the index to the first character in the text. Spaces on either side of count are optional.

`+ count lines`

Adjust the index forward by count lines, retaining the same character position within the line. If there are fewer than count lines after the line containing the current index, then set the index to refer to the same character position on the last line of the text. Then, if the line is not long enough to contain a character at the indicated character position, adjust the character position to refer to the last character of the line. Spaces on either side of count are optional.

`- count lines`

Adjust the index backward by count lines, retaining the same character position within the line. If there are fewer than count lines before the line containing the current index, then set the index to refer to the same character position on the first line of the text. Then, if the line is not long enough to contain a character at the indicated character position, adjust the character position to refer to the last character of the line. Spaces on either side of count are optional.

`linestart`

Adjust the index to refer to the first character on the line.

`lineend`

Adjust the index to refer to the last character on the line.

`wordstart`

Adjust the index to refer to the first character of the word containing the current index. A word consists of any number of adjacent characters that are letters, digits, or underscores, or a single character that is not one of these.

`wordend`

Adjust the index to refer to the character just after the last one of the word containing the current index. If the current index refers to the last character of the text then it is not modified.

If more than one modifier is present then they are applied in left-to-right order. For example, the index `"end - 1 chars"` refers to the next-to-last character in the text and `"insert wordstart - 1 c"` refers to the character just before the first one in the word containing the insertion cursor.

MARKS

The second form of annotation in text widgets is a mark. Marks are used for remembering particular places in a text. They have names and they refer to places in the file, but a mark isn't associated with particular characters. Instead, a mark is associated with the gap between two characters. Only a single position may be associated with a mark at any given time. If the characters around a mark are deleted the mark will still remain; it will just have new neighbor characters. In contrast, if the characters containing a tag are deleted then the tag will no longer have an association with characters in the file. Marks may be manipulated with the `mark` sub-command, and their current locations may be determined by using the mark name as an index in widget commands.

One mark has special significance. The mark *insert* is associated with the insertion cursor. The mark *point* is an synonym for insert. This special mark may not be unset.

USAGE

Text items are supported by the Pad++ text command. Text items are created with widget commands of the following form:

```
pathName create text [option value option value ...]
```

There may be any number of option-value pairs, each of which sets one of the configuration options for the item. These same option-value pairs may be used in itemconfigure widget commands to change the item's configuration. The following options are supported for text items:

`-editable [18]`	True if text item is editable with default event handlers (default is false)
`-font` [24]	Specifies font to use for text
`-pen` [45]	Specifies pen color of item
`-text [56]`	The text of any item containing text

Note that when the -width or -height of a text item is set, the text item is clipped to those dimensions rather than being squashed or stretched as most items are.

Also, see the **`text` [91]** command that can be used to manipulate text items.

Text items have default event bindings which can be used for emacs-style editing of them. See the section on **Default Bindings** for more info.

Textfile Items

A textfile item displays a string of characters on the screen in one or more lines as with text items, but the text is loaded in from a file. Textfile items are supported by the Pad++ text command. Textfile items are created with widget commands of the following form:

```
pathName create textfile [option value option value ...]
```

There may be any number of option-value pairs, each of which sets one of the configuration options for the item. These same option-value pairs may be used in itemconfigure widget commands to change the item's configuration. The following options are supported for text items:

`-file [22]`	File an item should be defined by
`-font` [24]	Specifies font to use for text
`-pen` [45]	Specifies pen color of item
`-text [56]`	(Read-only) The text of any item containing text
`-writeformat [69]`	Controls whether disk-based item is written out by copy or reference

Note that when the -width or -height of a textfile item is set, the textfile item is clipped to those dimensions rather than being squashed or stretched as most items are.

Also, see the **`text` [91]** command that can be used to manipulate text items.

Text items have default event bindings which can be used for emacs-style editing of them. See the section on **Default Bindings** for more info.

Textarea Items

Textarea items are widgets that are used to enter a free form multi-line block of text. They can be edited with emacs-style keys, and copied from and paste to with the mouse. They have horizontal and vertical scrollbars that can be used to edit a larger block of text than can fit in the window. Textareas are created with widget commands of the following form:

```
pathName create textarea [option value option value ...]
```

Textareas are one of several widgets that are designed to mirror the functionality and usage of the standard widgets in Java's Abstract Windowing Toolkit (AWT). When textareas are created, they automatically get the tag "Textarea". Textareas have default event handlers which define their behavior. These event handlers are defined on the tag "Textarea" for the "Run" event mode. See the section on **Default Bindings** for more details about the event bindings.

The following options are supported for textareas:

`-editable [18]`	True if textarea is editable (default is true)
`-fill` [23]	Specifies fill color of textarea
`-font` [24]	Specifies font to use for text
`-pen` [45]	Specifies pen color of textarea
`-text [56]`	The whole text within the textarea

Also, see the **`text` [91]** command that can be used to manipulate textarea items.

Text items have default event bindings which can be used for emacs-style editing of them. See the section on **Default Bindings** for more info.

Warning: The scrollbars on textareas are currently not hooked up to the text within the textarea.

Textfield Items

Textfield items are widgets that are used to enter a free-form single line of text. They can be edited with emacs-style keys, and copied from and paste to with the mouse. Textfields are much like textareas, but are limited to one line. Textfields are created with widget commands of the following form:

```
pathName create textfield [option value option value ...]
```

Textfields are one of several widgets that are designed to mirror the functionality and usage of the standard widgets in Java's Abstract Windowing Toolkit (AWT). When textfields are created, they automatically get the tag "Textfield". Textfields have default event handlers which define their behavior. These event handlers are defined on the tag "Textfield" for the "Run" event mode. See the section on **Default Bindings** for more details about the event bindings.

The following options are supported for textfields:

`-editable [18]`	True if textfield is editable (default is true)
`-fill` [23]	Specifies fill color of textfield
`-font` [24]	Specifies font to use for text
`-pen` [45]	Specifies pen color of textfield
`-text [56]`	The whole text within the textfield

Also, see the **`text` [91]** command that can be used to manipulate textfield items.

Text items have default event bindings which can be used for emacs-style editing of them. See the section on **Default Bindings** for more info.

Window Items

Window items are widgets that act like top-level windows within Pad++, but with no window dressing. They are used to group a collection of items. They are similar to frames, except they have no window dressing and no default event handlers. Windows are created with widget commands of the following form:

pathName create window [[x_1 y_1 x_2 y_2] option value option value ...]

Windows are one of several widgets that are designed to mirror the functionality and usage of the standard widgets in Java's Abstract Windowing Toolkit (AWT). When windows are created, they automatically get the tag "Window". Windows have no default event handlers which define their behavior. Windows are basically the essence of a Frame item type without the window dressing and without the event handlers.

Unlike group items, windows do not set their size based on their contents. Rather, they are fixed size as specified by the command line coordinates, or by the **-width** [68] and **-height** [26] itemconfigure options. Windows have their own coordinate system where (0, 0) specifies the panels lower left corner. Adding items to a window adds them relative to the window's coordinate system.

The following options are supported for windows:

`-divisible` [16]	True if events go through the window to its members
`-fill` [23]	Specifies fill color
`-members` [38]	The list of members of the window

Also, see the **addgroupmember [2]** and **removegroupmember [72]** commands that can be used to add and remove items from the window.

Default Bindings

There are several default event bindings in Pad++ written in C++. In addition, the PadDraw sample application has many event bindings defined in Tcl that may be useful. There are two classes of default event bindings, navigation and widget bindings.

The navigation bindings allow panning on button 1, and zooming in and out on buttons 2 and 3, respectively. These bindings are very simple versions and a serious application may want to redefine them. They can be turned on and off the with **-defaultEventHandlers** widget configuration option. By default, they are off. The bindings are:

- Pan with button 1:
  - *<ButtonPress-1> on* "`all`"
  - *<B1-Motion> on* "`all`"
  - *<ButtonRelease-1> on* "`all`"
- Zoom in/out with buttons 2/3:
  - *<ButtonPress-2> on* "`all`"
  - *<B2-Motion> on* "`all`"
  - *<ButtonRelease-2> on* "`all`"
  - *<ButtonPress-3> on* "`all`"
  - *<B3-Motion> on* "`all`"
  - *<ButtonRelease-3> on* "`all`"

The widget bindings allow standard interaction with the user interface widgets. These bindings get created the first time a widget of each type is created. The event bindings are defined on tags of the name of the widget. Widgets are created with these tags by default, and so these bindings are defined by default. To disable these bindings, just remove the tag from the widget. The event bindings are defined in the "Run" mode, and so for them to be active, the Run modifier must be set. This can be done with:

```
.pad modifier set "Run"
```

For key bindings to work, the system focus must be set to the pad widget. You can do this with:

```
focus .pad
```

The default event bindings are:

- Button widgets:
  - *<Run-ButtonPress-1> on* "`Button`"
  - *<Run-B1-Motion> on* "`Button`"

- *<Run-ButtonRelease-1> on* "`Button`"

- Scrollbar widgets:
  - *<Run-ButtonPress-1> on* "`Scrollbar`"
  - *<Run-B1-Motion> on* "`Scrollbar`"
  - *<Run-ButtonRelease-1> on* "`Scrollbar`"

- TextArea widgets:
  - *<Run-KeyPress> on* "`Textarea`"
  - *<Run-ButtonPress-1> on* "`Textarea`"
  - *<Run-B1-Motion> on* "`Textarea`"
  - *<Run-ButtonRelease-1> on* "`Textarea`"
  - *<Run-ButtonPress-2> on* "`Textarea`"
  - *<Run-B2-Motion> on* "`Textarea`"
  - *<Run-ButtonRelease-2> on* "`Textarea`"

- TextField widgets:
  - *<Run-KeyPress> on* "`Textfield`"
  - *<Run-ButtonPress-1> on* "`Textfield`"
  - *<Run-B1-Motion> on* "`Textfield`"
  - *<Run-ButtonRelease-1> on* "`Textfield`"
  - *<Run-ButtonPress-2> on* "`Textfield`"
  - *<Run-B2-Motion> on* "`Textfield`"
  - *<Run-ButtonRelease-2> on* "`Textfield`"

- Frame widgets:
  - *<Motion> on* "`Frame`"
  - *<Leave> on* "`Frame`"
  - *<Run-ButtonPress-1> on* "`Frame`"
  - *<Run-B1-Motion> on* "`Frame`"
  - *<Run-ButtonRelease-1> on* "`Frame`"
  - *<ButtonPress-2> on* "`Frame`"
  - *<B2-Motion> on* "`Frame`"
  - *<ButtonRelease-2> on* "`Frame`"
  - *<ButtonPress-3> on* "`Frame`"
  - *<B3-Motion> on* "`Frame`"
  - *<ButtonRelease-3> on* "`Frame`"

Finally, the basic Text item has default event bindings that can be used to edit the text with emacs-style keys. To use these bindings, the text item must be made editable and given the tag "Text". In addition, the Run mode must be set, and the focus must be set to the Pad++ widget. An example creation of a text item that uses the handlers is:

```
.pad create text -text Hello -editable 1 -tags "Text" -anchor nw
.pad modifier set "Run"
focus .pad
```

- Text items:
  - *<Run-KeyPress> on* "`Text`"
  - *<Run-ButtonPress-1> on* "`Text`"
  - *<Run-B1-Motion> on* "`Text`"
  - *<Run-ButtonRelease-1> on* "`Text`"
  - *<Run-ButtonPress-2> on* "`Text`"
  - *<Run-B2-Motion> on* "`Text`"
  - *<Run-ButtonRelease-2> on* "`Text`"

Global TCL Variables

Pad++ defines several global Tcl variables that are available for use by Tcl applications. They are:

- **Pad_Error** True during Pad++ background errors.
- **Pad_Version** Current version of this Pad++ software
- **Pad_Write** Used in the **<Write>** event for an application to specify if the system should write out a specific object or not. (See the **write** command and the **<Write>** event in the **bind** command.)

KPL-Pad++ Interface

As described in the section above on KPL ITEMS, Kpl is a byte-compiled language that comes with Pad++ that is typically used for creating new objects. It is a general-purpose language, and has the ability to call certain Pad++ rendering routines. Some basic documentation is available with the Pad++ release in *doc/kpl.troff*.

There are two ways to interact with Kpl. The first is to make a Pad++ Kpl item with a Kpl renderscript (described above). In this case, every time the item is rendered, the Kpl script will be executed. The second method is to use the **`kpl`** command available directly from Tcl. The **`kpl`** command has the following format:

`kpl` `subcommand [args ...]`

Where *subcommand* must be one of the following:

`eval` string
Byte-compiles and evaluates *string* as a Kpl script.

`push` value
Pushes *value* onto the top of the Kpl stack.

`pop`
Pops the top element off of the Kpl stack and returns it.

`get` name
Returns the current value of the Kpl variable, *name*.

`set` name value
Sets the Kpl variable *name* to *value*.

There are several Kpl commands available for interacting with the Tcl environment, and for rendering directly onto the Pad++ surface (when within a render callback). They are organized into a few groups as follows:

These commands provide a mechanism for accessing Tcl variables from Kpl.

`tclset` `name value`
Sets the global Tcl variable name to *value*.

`tclset2` `array_name element value`
Sets the global Tcl array *array_name*(*element*) to *value*.

`tclget` `name`
Returns the value of the global Tcl variable *name*.

tclget2 `array_name element`

Returns the value of the global Tcl array *array_name*(*element*).

tcleval `tcl_string`

Evaluates the Tcl string *tcl_string*.

These commands provide basic drawing capability.

drawborder `llcorner urcorner width border relief`

Draws a 3D border within the rectangle specified by *llcorner* and *urcorner* (where each of those are 2D vectors). *Width* specifies the zoomable width of the border. *Border* specifies the border color and must have been previously allocated with the Pad++ **allocborder** command. *Relief* specifies the style of border, and must be one of: "`raised`", "`flat`", "`sunken`", "`groove`", "`ridge`", "`barup`", or "`bardown`".

drawline `vector`

Draws a line specified by *vector*. As Kpl vectors may be up to 16-dimensional, this vector can specify up to 8 (x, y) points. This routine will draw a line connecting as many points as are specified within *vector*.

drawimage `imagetoken x y`

Draws the image specified by imagetoken at the point (x, y). (Also see **image** commands as well as the description of **image** items). This command can only be called within a render callback.

drawpolygon `vector`

Draws a polygon specified by *vector*. As Kpl vectors may be up to 16-dimensional, this vector can specify up to 8 (x, y) points. This routine will draw a closed polygon connecting as many points as are specified within *vector*.

drawtext `text position`

Draws text. *Text* specifies the text to be drawn. *Position* specifies the where the text gets drawn. *Position* is a two-dimensional vector specifying the (x, y) position. (Also see the KPL **setcolor**, **setfont**, and **setfontheight** commands.)

getlevel

Returns the current refinement level.

getsize

Returns the current size of the object, where size is the larger of the width and height.

renderitem `tagOrId`

During a render callback triggered by the *-renderscript* option, this function actually renders the object. During a *-renderscript* callback, all the items specified by *tagOrId* are rendered (and the current item is not rendered unless it is in *tagOrId*). This function may only be called during a render callback.

setabslinewidth `width`

Sets the current drawing with to an absolute width. All lines will be drawn with this width. This is an absolute width, so this specifies the width independent of the current view. I.e., the line width will not change as the view changes.

setcapstyle `capstyle`

Sets the capstyle of lines for drawing. *Capstyle* may be any of: "butt", "projecting", or "round".

setcolor `color`

Sets the current drawing color to *color*. Note that *color* must have been previously allocated by the **alloccolor** Pad++ command.

setfont `font`

Specifies the font to be used for rendering text for this item. *Font* must specify a filename which contains an Adobe Type 1 font, or the string "`Line`" which causes the Pad++ line-font to be used. Defaults to "`Times-12`". (Also see the **setfontheight** command.)

setfontheight `height`

Sets the height of the font for future drawing with render callbacks. *Height* is specified in pixels. (Also see the **setfont** command).

setjoinstyle `joinstyle`

Sets the joinstyle of lines for drawing. *Joinstyle* may be any of: "bevel", "miter", or "round".

setlinewidth `width`

Sets the current drawing width to a zoomable width. All lines will be drawn with this width. This is a zoomable width, so this specifies the width as it will look when the view has a magnification of 1.0.

These commands provide drawing commands in a style much like postscript.

closepath

Specifies the end of a path.

curveto `vector`

Draws a bezier curve. Here, *vector* is a six-dimensional vector. The current point plus these three points specify four points which control the bezier curve.

fill

Fills the current path.

lineto `vector`

Specifies a straight line in the current path from the current point to (x, y) specified by *vector*. Makes (x, y) the current point.

moveto `vector`

Moves the current point within the current path to (x, y) specified by *vector*.

newpath

Specifies the beginning of a new path.

stroke

Draws the current path with an outline only - the path is not filled.

These commands provide control over refinement.

interrupted

Returns true (1) if there has been an event during this render to interrupt it. It is up to objects that take very long to render themselves to check this flag during the rendering. If it is true (i.e., the render has been interrupted), then the Kpl render routine should return immediately - without completing the render. Generally, renders at refinement level 0 should always be quite fast, but further refinement levels can take an arbitrarily long time to render as long as they are interruptible.

refine

Specifies that this item wants to be refined. Pad++ will schedule a refinement, and at some point in the near future, the item will be re-rendered at the next higher refinement level. An item can use the current level in conjunction with this command to render itself simply at first, and then fill in more and more detail when it is refined.

Here is an example that creates a Kpl item with a renderscript that exercises some of the commands described here.

```
        # Tcl code to load Kpl code and to create
        # Pad++ Kpl item.
kpl eval 'triangle.kpl source
set pen [.pad alloccolor brown]
.pad create kpl -bb {-10:-10 110:110} -renderscript {test_drawing}

        /* Kpl code (in a separate file)
           to test the drawing commands */
{
           /* Draw a looping bezier curve */
3 setlinewidth
'pen1 tclget setcolor
newpath
  0:0 moveto
  200:75:-100:75:100:0 curveto
stroke

           /* Draw a filled square */
'pen2 tclget setcolor
newpath
  0:0 moveto
  50:0 lineto
  50:50 lineto
  0:50 lineto
fill

           /* Draw a square outline */
'pen3 tclget setcolor
newpath
  0:0 moveto
  50:0 lineto
  50:50 lineto
  0:50 lineto
  0:0 lineto
stroke

           /* Draw a square outline
           with an absolute width */
1 setabslinewidth
'pen4 tclget setcolor
newpath
  0:0 moveto
  50:0 lineto
  50:50 lineto
```

```
   0:50 lineto
   0:0 lineto
stroke

            /* Cause one level of refinement.
            Notice the bezier curve is rendered
            at low-resolution at first,
            and then improves with refinement. */
getlevel => i
i 1 < (
   refine
)
} -> test_drawing
```