Pad++ Reference Manual
(Version 0.2.7)

URL: http://www.cs.unm.edu

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
- Item Types

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.

Items

`create`	Create new items
`itemconfigure`	Configure existing items
`delete`	Delete existing items
`find`	Search for items by various keys
`setid`	Change the id of an item
`type`	Get the type of an item
`raise`	Bring an item higher in the drawing order
`lower`	Push an item lower in the drawing order
`pick`	Find the item under a point
`text`	Modify text item
`allocimage`	Allocate data for an image item
`freeimage`	Free data from an image item
`pushcoordframe`	Add a new relative coordinate frame to the stack
`popcoordframe`	Pop a relative coordinate frame off of the stack
`resetcoordrame`	Reset coordinate frame stack to empty
`-sticky`	Specifies if an item should stay put when the view changes
`-lock`	Locks an item so it can not be modified or deleted
`-width`	Width of an item. Normally computed automatically, but can be set
`-height`	Height of an item. Normally computed automatically, but can be set
`-info`	A place to store application-specific information with an item
`-zoomaction`	A script that gets evaluated when an item is scaled larger or smaller than a set size
`-htmlanchors`	Some items only: The anchors associated with an HTML page
`-url`	Some items only: The URL associated with an item
`-html`	Some items only: The HTML item associated with an htmlanchor
`-ismap`	Some items only: True if an htmlanchor is an image map
`-state`	Some items only: State of an item (such as visited, unvisited, etc.)
`-dither`	Some items only: Render with dithering
`-image`	Some items only: Image data associated with item (allocated by allocimage)

`-arrow`	Some items only: Whether to draw arrow heads with this item
`-arrowshape`	Some items only: The shape of drawn arrow heads
`-title`	Some items only: Title of an item
`-file`	Some items only: File an item should be defined by

Item Transformations

`slide`	Move an item relatively in (x, y)
`scale`	Change the size of an item relatively
`coords`	Change the coordinates of an item
`bbox`	Get the bounding box of an item
`getsize`	Get the size of an item (possibly within portals)
`-place`	Transformation of an item - Translation (x, y), and magnification (z)
`-x`	X componenent of -place transformation
`-y`	Y componenent of -place transformation
`-z`	Z componenent of -place transformation
`-anchor`	The part of the item that -place refers to

View Transformations

`getview`	Get the current view (possibly within portals)
`moveto`	Change the view (possibly within portals)
`zoom`	Zoom the view around a specified point
`center`	Change the view so as to center an item
`centerbbox`	Change the view so as to center a bounding box
`-viewscript`	A script that gets evaluated whenever the view is changed
`-view`	Some items only: Specifies the view this item sees
`-lookon`	Some items only: Specifies the pad widget this item sees

Tags

`addtag`	Add a tag to an item
`deletetag`	Delete a tag from an item
`dtag`	Synonym for deletetag
`gettags`	Get the tags an item has
`hastag`	Determine if an item has a particular tag
`-tags`	List of tags associated with an item

Events

`bind`	Create, modify, access, or delete event bindings
`focus`	Set the focus for keyboard events
`bindtags`	Specify whether events should go to the most-specific or most-general description first
`addmodifier`	Add a new user-defined modifier for future use
`deletemodifier`	Delete a user-defined modifier
`setmodifier`	Make the specified user-defined modifier the current one
`getmodifier`	Get the current user-defined modifier
`-events`	True if item receives events, false otherwise

Groups

`addgroupmember`	Add an item to a group
`removegroupmember`	Remove an item from a group
`getgroup`	Get the group an item belongs to
`-members`	The list of members of a group
`-divisible`	True if events go through a group to its members

Layout

`grid`	Layout pad items in a grid as with the Tk grid command

`tree`	Layout pad items with a dynamic graphical-fisheye view tree

Rendering

`damage`	Specify that a group of items needs to be redrawn
`update`	Force any requested render requests to occur immediately
`-alwaysrender`	True if the item must be rendered, even if the system is slow and the item is small
`-minsize`	The minimum size an item is rendered it (absolute or relative to window size)
`-maxsize`	The maximum size an item is rendered it (absolute or relative to window size)
`-faderange`	Range over which an item fades in or out
`-transparency`	Transparency of an item. 0 is completely transparent, 1 is completely opaque
`-layer`	The layer an item is on
`-visiblelayers`	The layers that are visible within this view (just for portals and pad surface, item #1)
`-fill`	Some items only: Specifies fill color of item
`-pen`	Some items only: Specifies pen color of item
`-penwidth`	Some items only: Specifies width of pen
`-border`	Some items only: Specifies border color of item
`-borderwidth`	Some items only: Specifies width of border
`-relief`	Some items only: Specifies how a border should be rendered
`-font`	Some items only: Specifies font to use for text
`-capstyle`	Some items only: Specifies how to draw line ends
`-joinstyle`	Some items only: Specifies how to draw the joints within multi-point lines
`-noisedata`	Some items only: Specifies parameters to render item with noise

File I/O

`read`	Read a .pad file
`write`	Write a .pad file (all the items on a widget)

Miscellaneous

`info`	Get type-specific information about an item
`configure`	Modify the pad widget
`windowshape`	Modify the shape of the top-level window that a pad widget is in
`islinked`	Determine if the top-level window that a pad widget is in has been mapped yet
`setlanguage`	Set the language to be used for future calback scripts
`settoplevel`	Set the language to be used by the top-level interpreter

Utilities

`line2spline`	Generate points for a spline that approximate a line
`spline2line`	Generate points for a line that approximate a spline
`clock`	Create a clock to measure elapsed milliseconds
`getdate`	Get the current date in unix format
`getpads`	Get a list of all pad widgets currently defined
`noise`	Generate 'perlin' noise
`padxy`	Convert a window point (x, y) to pad coordinates
`urlfetch`	Retrieve a URL over the internet in the background
`-updatescript`	Some items only: A script to evaluate when a background action has made progress
`-donescript`	Some items only: A script to evaluate when a background action has completed
`-errorscript`	Some items only: A script to evaluate when a background action has an error

Renderscripts

`renderitem`	Render an item in a render callback
`allocborder`	Allocate a border for future rendering
`alloccolor`	Allocate a color for future rendering
`allocimage`	Allocate a image for future rendering
`freeborder`	Free a border previously allocated

`freecolor`	Free a color previously allocated
`freeimage`	Free an image previously allocated
`setcapstyle`	Specify how the end caps of lines should be drawn
`setjoinstyle`	Specify how the joints within multi-point lines should be drawn
`setlinewidth`	Specify the penwidth of lines when they are drawn
`setfont`	Specify the font to be used for renderscript drawing
`setfontheight`	Specify the font height to be used for renderscript drawing
`drawline`	Draw a line within a renderscript
`drawpolygon`	Draw a polygon within a renderscript
`drawtext`	Draw text within a renderscript
`drawimage`	Draw an image within a renderscript
`getlevel`	Get the render level within a renderscript
`getmag`	Get the current magnification within a renderscript
`getportals`	Get the list of portals being rendered within during a renderscript
`gettextbbox`	Get the bounding box of a text string
`-renderscript`	A script that gets evaluated every time an item is rendered
`-bb`	A script that gets evaluated to specify the bounding box of an item

Debugging

`printtree`	Print all the items on the pad surface in their internal tree structure

Extensions

`addtype`	Create a new item type
`addoption`	Create a new option for an existing type

Note that change bars appear wherever this document differs from the previous version.

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. PadDraw is started by running the *pad* script which automatically runs *padwish* and starts the Tcl program. When running PadDraw by executing *pad*, the Tcl interpreter is not available.

PADWISH SYNOPSIS

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

Valid options are:

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

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

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

`-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.

`-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).

`-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.

`-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'.

`-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).

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

`--`
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 only with Tcl7.5/Tk4.1.

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: **colorCubeSize**
Class: **ColorCubeSize**
Command-Line Switch: **-colorCubeSize**

Specifies how many colors to allocate for images. Whenever images are displayed, the system tries to allocate colorCubeSize^3 colors. For example, if colorCubeSize is 5, then 5*5*5 or 125 colors will allocated. If unsuccessful, smaller color cubes are tried successively. Default is 5.

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: **debugstat**
Class: **DebugStat**
Command-Line Switch: **-debugstat**

Turns on and off status line on the Pad (for debugging). Default is 0. The status line shows the total number of items on the pad surface, the number of items checked for rendering, and the number of items actually rendered during the most recent render.

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: **enableOpaque**
Class: **EnableOpaque**
Command-Line Switch: **-enableOpaque**

Normally, objects which are completely behind opaque objects are not rendered. Turn this flag off to turn off this efficiency method. 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: **refinementDelay**
Class: **RefinementDelay**
Command-Line Switch: **-refinementDelay**

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

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:

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

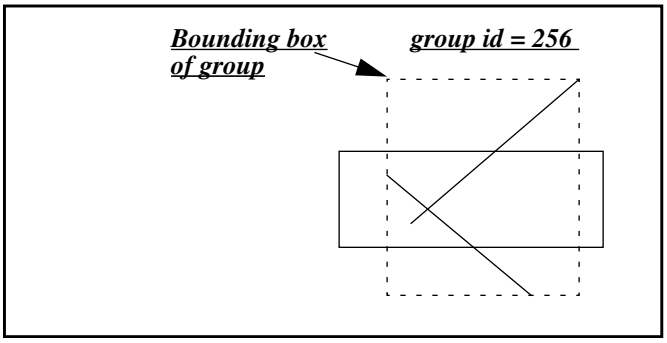
Add all items specified by *tagOrId* to the group specified by *groupTagOrId*. If *groupTagOrId* specifies more than one item, the first one is used. The items are added to the end of the group in the order specified by *tagOrId*. Groups automatically update their bounding boxes to enclose all of their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

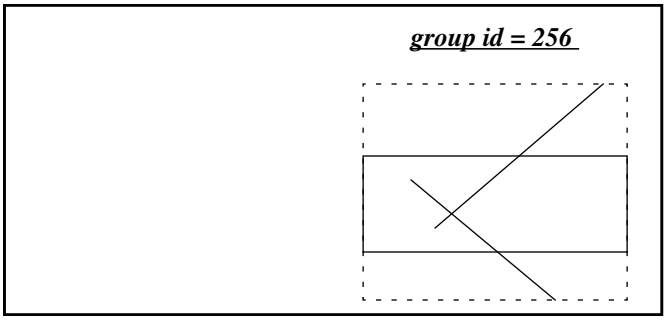
Example :

```
set id0 [.pad create line 0 0 100 100]
254
set id1 [.pad create line -10 20 80 -60]
```

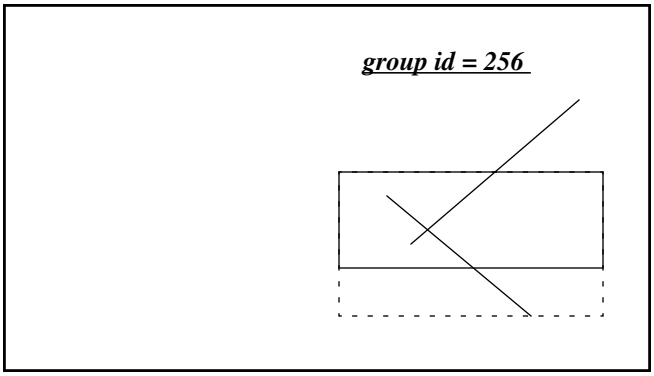
```
255
set gid [.pad create group -members "$id0 $id1"]
256

.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
```

`pathName` **`addmodifier`** `modifier`

Define *modifier* to be a user-defined modifier that can be used in future event bindings. (Also see the **`deletemodifier`**, **`setmodifier`**, **`getmodifier`**, and **`bind`** commands).

`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.)

`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]`

`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.)

`pathName` **`allocborder`** `color`

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. (Also see the **`freeborder`** and **`drawborder`** commands).

`pathName` **`alloccolor`** `color`

Allocates a color for future use by render callbacks. *Color* may have any of the forms accepted by Tk_GetColor. (Also see the **`freecolor`** and **`setcolor`** commands).

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

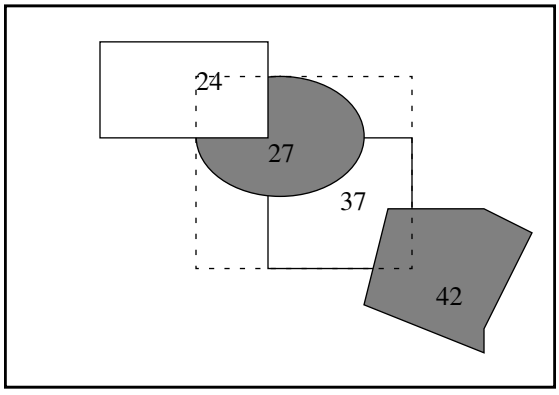
Allocates an image for future use by image objects and render callbacks. *file* specifies the name of a file containing an image. **allocimage** can always read gif file formats. In addition, if Pad++ is compiled with the appropriate libraries, **allocimage** can also read jpeg and tiff image file formats, and will automatically determine the file type. Normally, images 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. The image may have transparent pixels. This returns an image token which can be used by related commands. (Also see the **freeimage**, **drawimage**, and **info** commands, and the description of **image** items.).

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
```

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**. 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
```

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*).

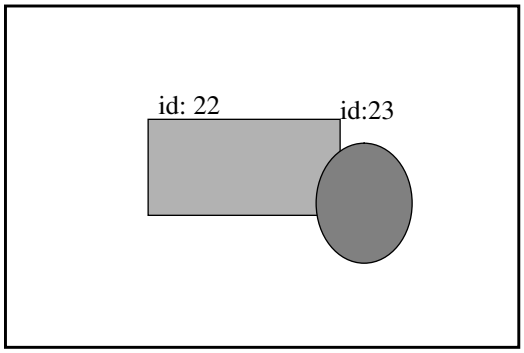
If *tagOrId* is *pathName*, then it does not change the ordering of any objects, but controls the default ordering of objects created in the future.

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*.

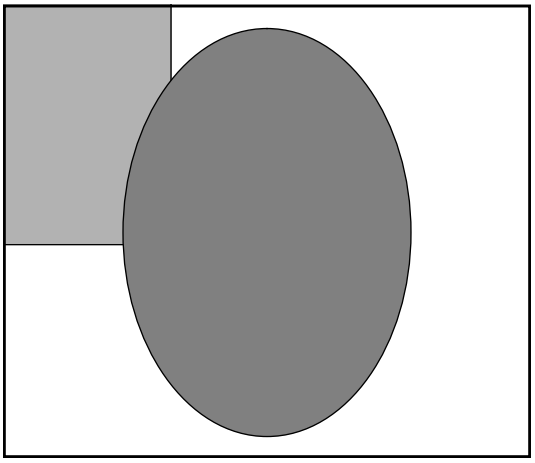
pathName **center** [-twostep] tagOrId [time x y [z [portalID ...]]]

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's center appears at the position on the screen specified by x and y, both 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 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



`pathName` **`centerbbox`** `[-twostep] x1 y1 x2 y2 [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 item's center appears at the position on the screen specified by x and y, both 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 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.

`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
```

`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.

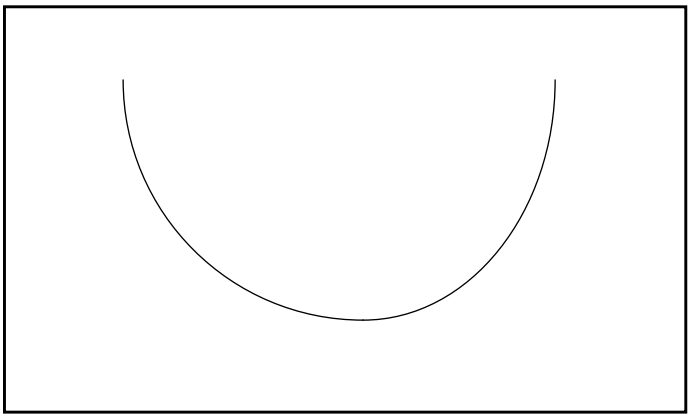
`pathName` **`coords`** `[-objectcoords] [-append] [-nooutput] tagOrId [`x_0 y_0 `...]`

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
   }
```

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 subsection below for detail on the syntax of this command. This command returns the id for the new item.

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.

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).

pathName **deletemodifier** modifier

Delete *modifier* from the list of valid user-defined modifiers. Any event bindings that are defined with this modifier become invalid. (Also see the **addmodifier**, **setmodifier**, **getmodifier**, and **bind** commands).

pathName **deletetag** tagToDelete tagOrId [tagOrId ...]

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. Note that **dtag** is an acceptable synonym for deletetag. 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 deletetag foo [.pad find withtag bar]`

pathName **drawborder** border type width x_1 y_1 x_2 y_2

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

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

`pathName` **`drawimage`** `imagetoken x y`

Draws the image specified by imagetoken at the point (x, y). (Also see **`allocimage`**, **`freeimage`,** and **`info`** commands as well as the description of **`image`** items). This command can only be called within a render callback.

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

Draws a multi-segment line connecting the specified coordinates. (See **`setcolor`**, **`setlinewidth`,** **`setcapstyle`,** and **`setjoinstyle`** commands). This command can only be called within a render callback.

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

Draws a closed polygon connecting the specified coordinates. (See **`setcolor`** and **`setlinewidth`**). This command can only be called within a render callback.

`pathName` **`drawtext`** `string xloc yloc`

Draws the specified text at the specified location. This command can only be called within a render callback. (Also see the **`setcolor`**, **`setfont`**, and **`setfontheight`** commands.)

`pathName` **`find`** `[-groupmembers] searchCommand [arg arg ...]`

This command returns a list consisting of all the items that meet the constraints specified by *searchCommand* and *arg*'s. The objects are returned in display list order, and if *-groupmembers* is specified, then group members are returned, otherwise, they are not. Note that this command does not return the pad surface (id #1). *SearchCommand* may take any of these forms:

`all`
Returns all the items on the pad.

`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] [startTagOrId]`
Returns the single item closest to the point given by *x* and *y*. If more than one item is at the same closest distance (e.g. two items overlap the point), then the top-most of these items (the last one in the display list) is used. If *halo* is specified, then any item closer than *halo* to the point is considered to overlap it. (*Halo* must be a non-negative number.) If *halo* is not specified, then only items at the point (x, y) will be found.

The *startTagOrId* argument may be used to step circularly through all the closest items. If *startTagOrId* is specified, it names an item using a tag or id (if by tag, it selects the first item in the display list with the given tag). Instead of selecting the topmost closest item, this form will select the topmost closest item that is below start in the display list; if no such item exists, then the selection behaves as if the start argument had not been specified.

`withinfo info`
Returns all the items containing the string *info* in their info itemconfigure option.

`withlayer layer`
Returns all the items on the layer *layer*.

`withname name`
Returns all the items having *name*.

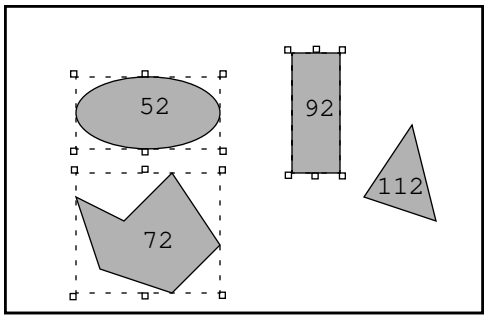
`withtag tagOrId`
Returns all the items given by *tagOrId*.

`withtext text`
Returns all the items containing *text*.

`withtype type`
Returns all the items of type *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
```

`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.

`pathName` **`freeborder`** `border`

Frees the *border* previously allocated by **`allocborder`**. (Also see the **`allocborder`** and **`drawborder`** commands).

`pathName` **`freecolor`** `color`

Frees the *color* previously allocated by **`alloccolor`**. (Also see the **`alloccolor`** and **`setcolor`** commands).

`pathName` **`freeimage`** `imagetoken`

Frees the *image* previously allocated by **`allocimage`**. (Also see the **`allocimage`** and **`drawimage`** commands, as well as the description of **`image`** items).

`pathName` **`getdate`**

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

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

`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).

`pathName` **`getlevel`**

Returns the current render level This command can only be called within a render callback. (See the sections on *Refinement* and *Region Management and Screen Updating* in the Programmer's Guide for more information about render levels).

`pathName` **`getmag`** `tagOrId`

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 *-place* itemconfigure option). This command can only be called within a render callback.

`pathName` **`getmodifier`**

Return the current active modifier. (Also see the **`addmodifier`**, **`deletemodifier`**, **`setmodifier`**, and **`bind`** commands).

`pathName` **`getpads`**

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

`pathName` **`getportals`**

Returns the list of the portals the current object is being rendered within. This command can only be called within a render callback.

`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.

`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.

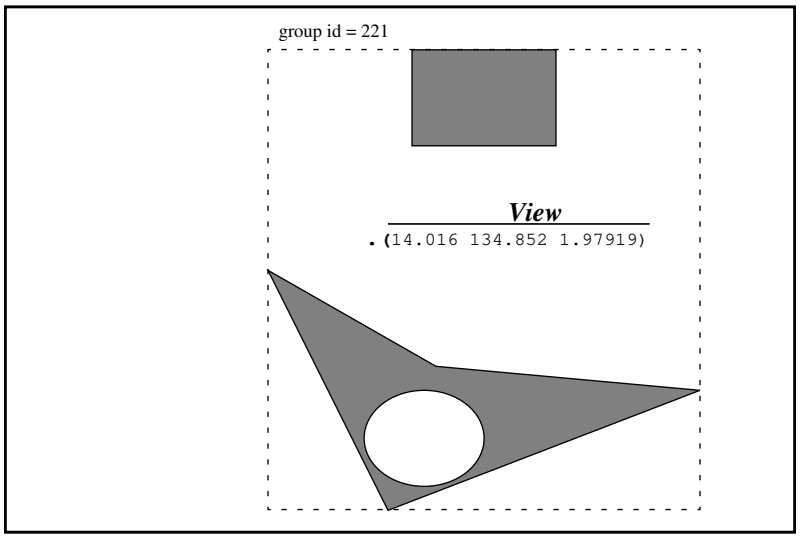
`pathName` **`gettextbbox`** `string`

Returns a list with four elements giving the bounding box of *string* if it is drawn with the **`drawtext`** 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 **`setfont`** and **`setfontheight`** commands.

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

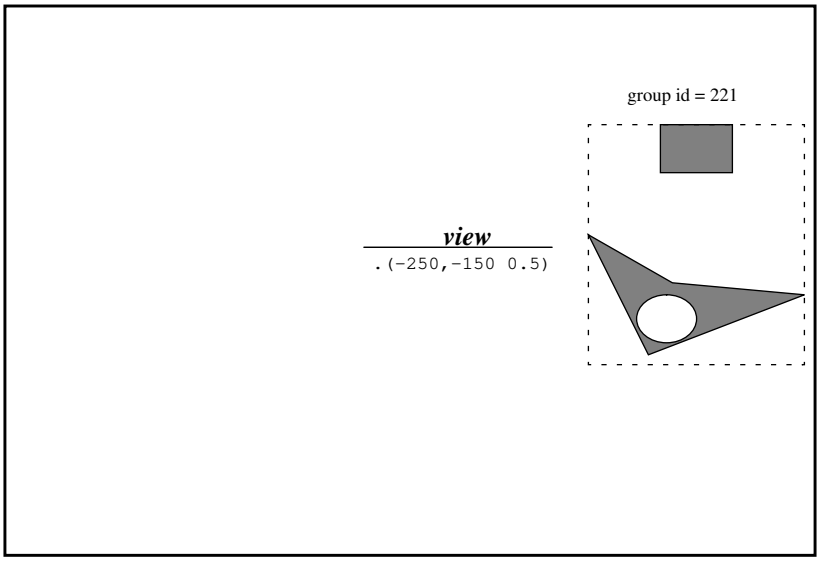
Returns the current view of the main window in "*xview yview zoom*" form. Here, (*xview*, *yview*) specifies the point at the center of the window, and *zoom* specifies the magnification. If a list of *portalID's* is specified, than the view of the last portal is returned instead of the view of the main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

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




```
.pad moveto -250 -150 0.5
.pad getview
```

```
-250 -150 0.5
.pad ic 221 -place
8.1125 118.753 1
```



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.

pathName **hastag** tagOrId tag

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.

pathName **info** subcommand

A general command for accessing information about pad and items on the pad surface. *subcommand* may be any of the following: *html* or *image*. Each subcommand may have sub-subcommands and options. All the subcommands and their options follow:

`html getlastchangedate <tagOrId>`
Returns the last date this page was modified as specified by the server.

`html getlength <tagOrId>`
Returns length of this page in bytes.

`html getsource <tagOrId>`
Returns HTML source of this page.

`html gettype <tagOrId>`
Returns Mime type of this page as specified by the server.

`image getdim <imagetoken>`
Returns dimensions {x y} of this image in pixels.
`image getname <imagetoken>`
Returns filename this image was loaded from.

`pathName` **`islinked`**

WARNING: **`islinked`** is an obsolete command and will be removed in the next release. Replace all uses of **`islinked`** with the Tk 'winfo ismapped' command.

Returns a flag specifying if *pathName* has been mapped to the display yet.

`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. **`ic`** is an allowed synonym for `itemconfigure`. 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).

`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`**.)

`pathName` **`lower`** `[-one] 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.

`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).

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
```

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.

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

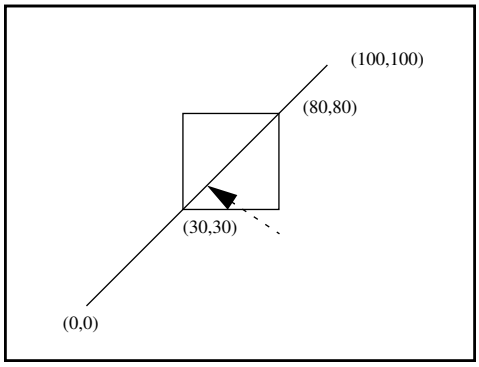
Given a window coordinate (*winx*, *winy*), it returns the visible object underneath that point. If the point should pass through any portals, a **<PortalIntercept>** event will be fired which will determine if the event will pass through that portal. By default, the **pick** command uses the divisibility 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
```

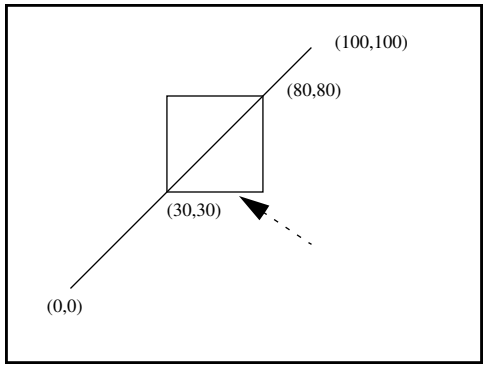
Now, group the line and rectangle:

```
% .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)
```

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.

pathName **printtree**

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

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 *-place* or *-z* itemconfigure options).

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

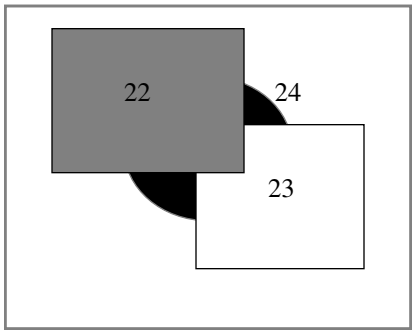
```
for {set i 0} {$i < 50} {incr i} {
   set id [.pad create rectangle 10 10 80 80 -penwidth 2]
   .pad pushcoordframe $id
}
.pad resetcoordframe
```

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

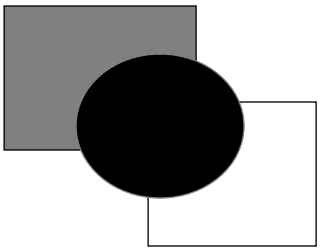
pathName **raise** [-one] tagOrId [aboveThis]

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.

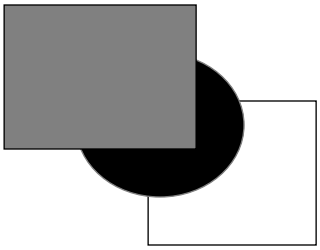


.pad raise 24



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

The original position turns to be:



`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.

`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.

`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.

`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.

`pathName` **`scale`** `tagOrId [scaleAmount [padX padY]]`

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 (padX, padY), 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).

`pathName` **`setcapstyle`** `capstyle`

Sets the capstyle of lines for drawing within render callbacks. *Capstyle* may be any of: "butt", "projecting", or "round". This command can only be called within a render callback.

`pathName` **`setcolor`** `color`

Sets the color for future drawing with render callbacks. *Color* must have previously been allocated by **alloccolor**. This command can only be called within a render callback. (Also see the **alloccolor** and **freecolor** commands).

`pathname` **`setfont`** `fontname`

Sets the font for future drawing with render callbacks. This affects the result of the **gettextbbox** command. *Fontname* must specify a filename which contains an Adobe Type 1 font, or the string "`System`" which causes the Pad++ line-font to be used. Defaults to "`System`". (Also see the **setfontheight** command).

`pathname` **`setfontheight`** `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 **gettextbbox** command. (Also see the **setfont** command).

`pathname` **`setid`** `tagorid id`

Sets the id of an existing item to *id*. If tagord 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.

`pathName` **`setjoinstyle`** `joinstyle`

Sets the joinstyle of lines for drawing within render callbacks. *Joinstyle* may be any of: "bevel", "miter", or "round". This command can only be called within a render callback.

`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.)

`pathName` **`setlinewidth`** `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. This command can only be called within a render callback.

`pathName` **`setmodifier`** `modifier`

Make *modifier* be the current active modifier for this pad widget. *modifier* must have been previously defined with the setmodifier command. (Also see the **`addmodifier`**, **`deletemodifier`**, **`getmodifier`**, and **`bind`** commands).

`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.)

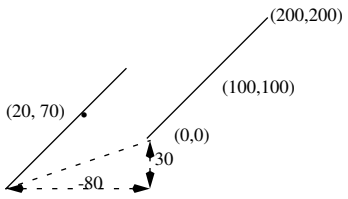
`pathName` **`shape`** `[innercoords outercoords]`

WARNING: **`shape`** has been renamed to windowshape, and will be removed in the next release. Replace all uses of **`shape`** with the **`windowshape`** command.

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

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 *-place* 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).

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




```
.pad slide $id -80 30
20.000000 70.000000
```

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 apprxoimates 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**.)

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

Controls all interaction with a text item. 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.

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 focii, 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 togethe under an invisible root node. When the layout function is called on the root node, both hierarchied will be laid out according to the layout object which resides at the root node.

A dynamic tree supports an abitrary 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 animte 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.

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.

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.

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.

```
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.

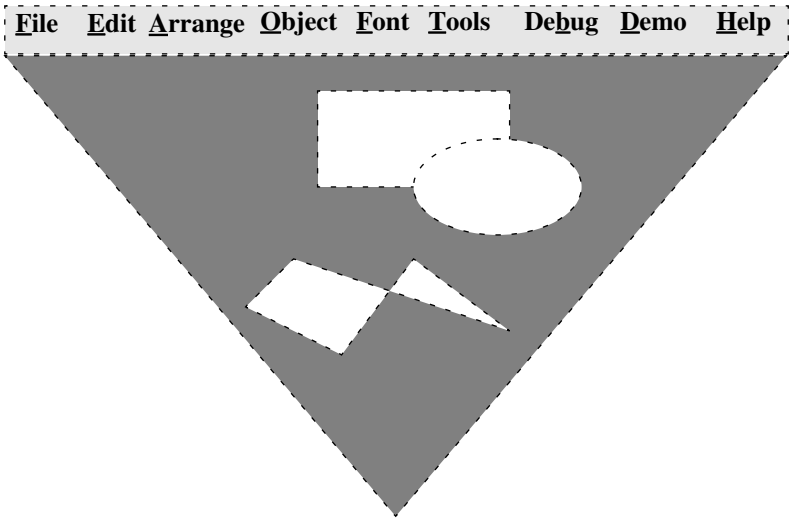
```
#
# urlfetch example using a file
#
proc done {filename token} {
   set file [open $filename "r"]
   ...  # handle file
}
set file "foo"
.pad urlfetch http://www.cs.unm.edu -file $file \
   -donescript "done $file"

#
# 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"
```

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.

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}
```



`pathName` **`write`** `filename [tagOrId tagOrId ...]`

Writes the Tcl commands necessary to recreate the items on the Pad++ surface into *filename*. 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 *filename* is an empty string, than this command returns the string 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.

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.

`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++ (grid, group, handle, html, image, kpl, line, polygon, portal, rectangle, spline, tcl, text, and textfile). 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.

These are the options that are supported by all item types:

`-place`: Place sets the anchor position of the object.

Every object has a `-place` which specifies the anchor point of that object (see `-anchor`). The place specifies the object's position and size. The size is multiplicitive. Place can be one of:

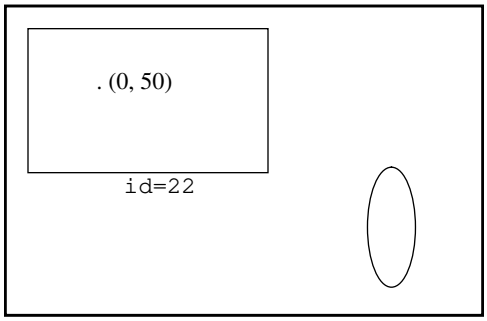
- *"x y size"*

Specifies (x, y, size) where 'x y size' is a single string specifying anchor point and item size. Items that have coordinates (lines, rectangles, polygons, and portals) have a default -place which depends on the coordinates of the item. For a "center" anchor (the default), the place will be the center of the coordinates. Other items (that don't have coordinates) have a default of "0 0 1".

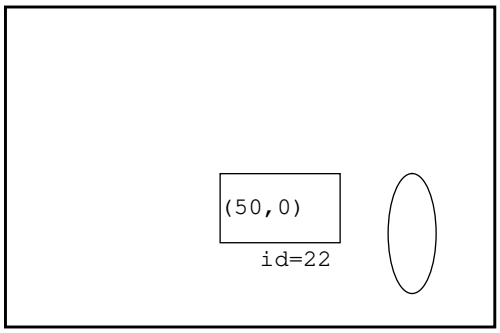
- *"center"*

  Center of screen. The object is positioned and sized so that its biggest dimension fills up 75% of the window, and it is centered. (This is dependent on the current view, and the current window dimensions.)

```
.pad ic 22 -place -150 150 1
```




```
.pad ic 22 -place "-50 20 0.5"
```



`-x x`

A synonym for the first (x) component of `-place`.

`-y y`

A synonym for the second (y) component of `-place`.

`-z z`

A synonym for the third (z) component of `-place`.

`-alwaysrender boolean`

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).

`-anchor anchorPos`

AnchorPos tells how to position the object relative to the positioning point for the item (see `-place`); 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.

`-clipping boolean`

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.

`-events boolean`

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

`-faderange value`

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.)

`-height height`

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. Items are centered within the specified *height*. If the dimensions are specified as smaller than the default values, the item is clipped to those dimensions. (Also see the `-width` itemconfigure option.)

`-info info`

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

`-layer layer`

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".

`-lock lock`

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`**.

`-minsize size`

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 minium 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.

`-maxsize size`

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.

`-renderscript TclScript`

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.

`-sticky boolean`

Specifies if this item should be "sticky". Sticky items are rendered independent of the current view. That is, as the view pans and zooms, sticky items appear effectively stuck to the screen. All sticky items are rendered after non-sticky items, thus sticky items always are on top of non-sticky items. (See the **getview** and **moveto** commands.) Defaults to 0 (false).

`-tags tagList`

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.

`-timerscript TclScript`

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*).

`-timerrate rate`

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`).

`-transparency value`

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.

`-viewscript TclScript`

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.

`-visible boolean`

WARNING: -visible is an obsolete option and will be removed in the next release. Replace all uses of the *-visible* option with *-transparency* which is more general.

Specifies whether this item is visible. Note that invisible items receive events and respond to commands such as **find**. Defaults to true.

`-width width`

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. Items are centered within the specified *width*. If the dimensions are specified as smaller than the default values, the item is clipped to those dimensions. (Also see the `-height` itemconfigure option.)

`-zoomaction {size growScript shrinkScript}`

Specifies a pair of Tcl scripts that gets evaluated when an item grows or shrinks so that its size crosses size. This is a simple way of making "semantically zoomable" objects - that is, objects that look different

when the 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.

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 -visible 0
   .pad pushcoordframe rect
   set image_token [.pad allocimage images/unm_logo_orig.gif]
   .pad create image -image $image_token -anchor sw -tags "image"
   .pad popcoordframe
   .pad renderitem
}

proc shrink {} {
   .pad ic rect -visible 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"
}
```

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 boolean`

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).

The syntax of the grid sub-command is:

```
pathNname 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 immidiately. 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 forcolumn, 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

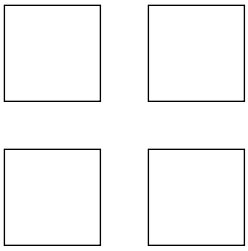
- The *-ipadx* and *-ipady* grid item configuration options are not available in pad.
- Master window geometry propagation flag is not available in pad.
- The parent-child and stacking restrictions and rules for master and slave items are not supported in pad (slaves can only be in the master group).
- 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 boolean`

> Specifies whether events should go to group members. If *-divisible* is 1 (true), events never go to the group 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 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).

`-members members`

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*.

HANDLE ITEMS

Items of type handle are special items that are designed to be used for selection handles around items when manipulating them within an application. Handles are similar to rectangles, except unlike every other item in Pad++, handles do *not* zoom. Handles are always rendered at the same size. Handles, however, do slide around like other items. They just don't zoom. Handles default to being five by five pixels, but this can be changed with the `-width` and `-height` itemconfigure options.

Handles are created with widget commands of the following form:

```
pathName create handle [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 handles:

`-fill color`

Fill the background of the 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 red.

`-pen color`

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

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 color`

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.

`-borderwidth width`

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.

`-divisible boolean`

If true, then events go through the HTML object to its anchors. If false, events stop at the HTML object, and never go through to the anchors. Defaults to true.

`-donescript script`

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.

`-errorscript script`

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.

`-fill color`

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.

`-font fontname`

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 "`System`" which causes the Pad++ line-font to be used. Defaults to "`System`".

`-htmlanchors`

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

`-members`

Because an HTML item is a group, it may contain other members in addition to its anchors. This allows setting and retrieving of all members that are part of this HTML item.

`-updatescript script`

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.

`-url urlname`

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".

`-width width`

Specifies the width (in the current units) of the html page. The page will be re-laid out according to the new width, and the length of the page may change dependent on the new width.

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`

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

`-ismap`

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

`-state state`

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".

`-url`

Returns the URL that this anchor addresses. This is a read-only option.

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 dithermode`

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).

`-image imagetoken`

Specifies the image that this item will render. (Also see the **`allocimage`** and **`freeimage`** commands.)

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:

`-renderscript kpl_script`

This is like the standard `-renderscript` option available to all items, but in this case, the string specifies a Kpl script instead of a Tcl script.

`-bb boundingboxScript`

A Kpl script that will be evaluated to compute the bounding box of this item. It should return two-element vectors that specify (x_1, y_1), (x_2, y_2) which are the lower left and upper right corners of this items bounding box.

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
```

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. The following options are supported for lines:

`-arrow where`

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`".

`-arrowshape shape`

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.

`-capstyle cap`

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.

`-joinstyle join`

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.

`-noisedata noisedata`

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.

`-pen color`

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.

`-penwidth width`

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.

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:

`-visiblelayers layers`

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". (See the *-layer* itemconfigure option that all items have.)

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. 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 color`

Fill the area of the polygon with color, which may be specified in any of the forms accepted by Tk_GetColor. If color is "none", (the default), then the polygon will not be filled.

`-joinstyle join`

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.

`-pen color`

Draw an outline around the edge of the polygon in color. Color may have any of the forms accepted by Tk_GetColor. If color is "none", then no outline will be drawn for the rectangle. This option defaults to black.

`-penwidth width`

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.

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. 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 color`

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.

`-borderwidth width`

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.

`-fill color`

Fill the background of the portal 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.

`-font fontname`

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 "`System`" which causes the Pad++ line-font to be used. Defaults to "`System`".

`-lookon surface`

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.

`-pen color`

color specifies the text color of the title. If color is "none", then no outline will be drawn for the rectangle. This option defaults to either black or white - whichever contrasts the most with the fill color.

`-relief relief`

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*"

`-title title`

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.

`-view place`

Specifies the (x, y, zoom) location this portal looks onto. Like a Pad++ widget, place specifies the point rendered at the center of the portal and the magnification. Defaults to directly under the location the portal was created at.

`-visiblelayers layers`

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". (See the *-layer* itemconfigure option that all items have.)

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` x_1 y_1 x_2 y_2 `[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 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 color`

Fill the area of the rectangle with color, which may be specified in any of the forms accepted by Tk_GetColor. If *color* is "none" (the default), then the rectangle will not be filled.

`-joinstyle join`

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.

`-pen color`

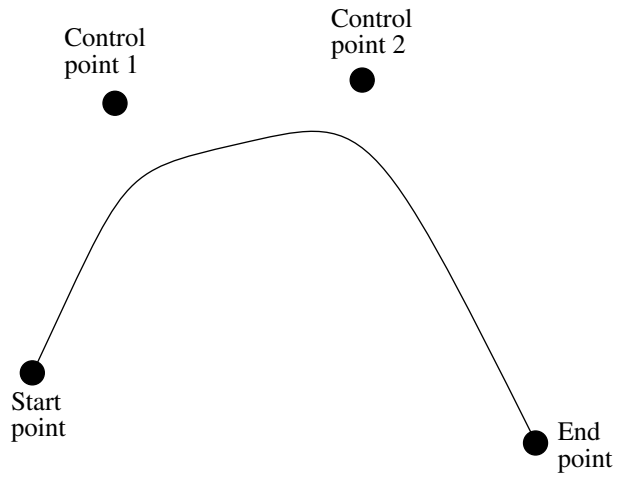
Draw an outline around the edge of the rectangle in color. *Color* may have any of the forms accepted by Tk_GetColor. If *color* is "none", then no outline will be drawn for the rectangle. This option defaults to black.

`-penwidth width`

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.

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 x1 y1... xn yn [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 where
```

Indicates whether or not arrowheads are to be drawn at one or both ends of the spline. *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`".

`-arrowshape shape`

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 spline from the neck of the arrowhead to its tip. The second element gives the distance along the spline from the trailing points of the arrowhead to the tip, and the third element gives the distance from the outside edge of the spline to the trailing points. If this option isn't specified then Pad++ picks a "reasonable" shape.

`-capstyle cap`

Specifies how the ends of the spline 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.

`-joinstyle join`

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.

`-pen color`

Color specifies a color to use for drawing the spline; 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.

`-penwidth width`

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.

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 boundingboxScript`

A Tcl script that will be evaluated to compute the bounding box of this item. 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.

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 *-place* 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:

`-font fontname`

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 "`System`" which causes the Pad++ line-font to be used. Defaults to "`System`".

`-pen color`

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

`-text string`

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.

The **`text`** command is described above with the other Pad++ commands under WIDGET COMMANDS.

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 fileName`

fileName specifies the filename to read a text file from.

`-font fontname`

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 "`System`" which causes the Pad++ line-font to be used.

Defaults to "System".

`-pen color`

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

`-text`

Returns the text in this textfile item. This is a read-only option and can not be set.

DEFAULT BINDINGS

In the current implementation, new Pad++ widgets are not given any default behavior: all behavior has to be described explicitly. However, the PadDraw sample application has many event bindings that may be useful.

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 **allocimage**, **freeimage,** and **info** 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 "`System`" which causes the Pad++ line-font to be used. Defaults to "`System`". (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
```