NAME

canvas − Create and manipulate 'canvas' hypergraphics drawing surface widgets

SYNOPSIS

canvas *pathName* ?*options*?

STANDARD OPTIONS

−background	**−borderwidth**	**−cursor**
−highlightbackground	**−highlightcolor**	**−highlightthickness**
−insertbackground	**−insertborderwidth**	**−insertofftime**
−insertontime	**−insertwidth**	**−relief**
−selectbackground	**−selectborderwidth**	**−selectforeground**
−takefocus	**−xscrollcommand**	**−yscrollcommand**

See the **options** manual entry for details on the standard options.

WIDGET-SPECIFIC OPTIONS

Command-Line Name: **−closeenough**
Database Name: **closeEnough**
Database Class: **CloseEnough**

Specifies a floating-point value indicating how close the mouse cursor must be to an item before it is considered to be "inside" the item. Defaults to 1.0.

Command-Line Name: **−confine**
Database Name: **confine**
Database Class: **Confine**

Specifies a boolean value that indicates whether or not it should be allowable to set the canvas's view outside the region defined by the **scrollRegion** argument. Defaults to true, which means that the view will be constrained within the scroll region.

Command-Line Name: **−height**
Database Name: **height**
Database Class: **Height**

Specifies a desired window height that the canvas widget should request from its geometry manager. The value may be specified in any of the forms described in the **COORDINATES** section below.

Command-Line Name: **−scrollregion**
Database Name: **scrollRegion**
Database Class: **Scr ollRegion**

Specifies a list with four coordinates describing the left, top, right, and bottom coordinates of a rectangular region. This region is used for scrolling purposes and is considered to be the boundary of the information in the canvas. Each of the coordinates may be specified in any of the forms given in the **COORDINATES** section below.

Command-Line Name: **−state**
Database Name: **state**
Database Class: **State**

Modifies the default state of the canvas where *state* may be set to one of: **normal**, **disabled**, or **hidden**. Individual canvas objects all have their own state option which may override the default state. Many options can take separate specifications such that the appearance of the item can be different in different situations. The options that start with **active** control the appearance when the mouse pointer is over it, while the option starting with **disabled** controls the appearance when the state is disabled. Canvas items which are **disabled** will not react to canvas bindings.

Command-Line Name: **−width**

Database Name: **width**
Database Class: **width**

Specifies a desired window width that the canvas widget should request from its geometry manager. The value may be specified in any of the forms described in the **COORDINATES** section below.

Command-Line Name: **−xscrollincrement**
Database Name: **xScrollIncrement**
Database Class: **ScrollIncrement**

Specifies an increment for horizontal scrolling, in any of the usual forms permitted for screen distances. If the value of this option is greater than zero, the horizontal view in the window will be constrained so that the canvas x coordinate at the left edge of the window is always an even multiple of **xScrollIncrement**; furthermore, the units for scrolling (e.g., the change in view when the left and right arrows of a scrollbar are selected) will also be **xScrollIncrement**. If the value of this option is less than or equal to zero, then horizontal scrolling is unconstrained.

Command-Line Name: **−yscrollincrement**
Database Name: **yScrollIncrement**
Database Class: **ScrollIncrement**

Specifies an increment for vertical scrolling, in any of the usual forms permitted for screen distances. If the value of this option is greater than zero, the vertical view in the window will be constrained so that the canvas y coordinate at the top edge of the window is always an even multiple of **yScrollIncrement**; furthermore, the units for scrolling (e.g., the change in view when the top and bottom arrows of a scrollbar are selected) will also be **yScrollIncrement**. If the value of this option is less than or equal to zero, then vertical scrolling is unconstrained.

INTRODUCTION

The **canvas** command creates a new window (given by the *pathName* argument) and makes it into a canvas widget. Additional options, described above, may be specified on the command line or in the option database to configure aspects of the canvas such as its colors and 3-D relief. The **canvas** command returns its *pathName* argument. At the time this command is invoked, there must not exist a window named *pathName*, but *pathName*'s parent must exist.

Canvas widgets implement structured graphics. A canvas displays any number of *items*, which may be things like rectangles, circles, lines, and text. Items may be manipulated (e.g. moved or re-colored) and commands may be associated with items in much the same way that the **bind** command allows commands to be bound to widgets. For example, a particular command may be associated with the <Button-1> event so that the command is invoked whenever button 1 is pressed with the mouse cursor over an item. This means that items in a canvas can have behaviors defined by the Tcl scripts bound to them.

DISPLAY LIST

The items in a canvas are ordered for purposes of display, with the first item in the display list being displayed first, followed by the next item in the list, and so on. Items later in the display list obscure those that are earlier in the display list and are sometimes referred to as being "on top" of earlier items. When a new item is created it is placed at the end of the display list, on top of everything else. Widget commands may be used to re-arrange the order of the display list.

Window items are an exception to the above rules. The underlying window systems require them always to be drawn on top of other items. In addition, the stacking order of window items is not affected by any of the canvas widget commands; you must use the Tk **raise** command and **lower** command instead.

ITEM IDS AND TAGS

Items in a canvas widget may be named in either of two ways: by id or by tag. Each item has a unique identifying number, which is assigned to that item when it is created. The id of an item never changes and id numbers are never re-used within the lifetime of a canvas widget.

Each item may also have any number of *tags* associated with it. A tag is just a string of characters, and it may take any form except that of an integer. For example, "x123" is OK but "123" is not. The same tag may be associated with many different items. This is commonly done to group items in various interesting ways; for example, all selected items might be given the tag "selected".

The tag **all** is implicitly associated with every item in the canvas; it may be used to invoke operations on all the items in the canvas.

The tag **current** is managed automatically by Tk; it applies to the *current item*, which is the topmost item whose drawn area covers the position of the mouse cursor (different item types interpret this in varying ways; see the individual item type documentation for details). If the mouse is not in the canvas widget or is not over an item, then no item has the **current** tag.

When specifying items in canvas widget commands, if the specifier is an integer then it is assumed to refer to the single item with that id. If the specifier is not an integer, then it is assumed to refer to all of the items in the canvas that have a tag matching the specifier. The symbol*ta gOrId* is used below to indicate that an argument specifies either an id that selects a single item or a tag that selects zero or more items.

tagOrId may contain a logical expressions of tags by using operators: "**&&**", "||", "^", "**!**", and parenthesized subexpressions. For example:

.c find withtag {(a&&!b)||(!a&&b)}

or equivalently:

.c find withtag {a^b}

will find only those items with either "a" or "b" tags, but not both.

Some widget commands only operate on a single item at a time; if *tagOrId* is specified in a way that names multiple items, then the normal behavior is for the command to use the first (lowest) of these items in the display list that is suitable for the command. Exceptions are noted in the widget command descriptions below.

COORDINATES

All coordinates related to canvases are stored as floating-point numbers. Coordinates and distances are specified in screen units, which are floating-point numbers optionally followed by one of several letters. If no letter is supplied then the distance is in pixels. If the letter is**m** then the distance is in millimeters on the screen; if it is **c** then the distance is in centimeters; **i** means inches, and **p** means printers points (1/72 inch). Larger y-coordinates refer to points lower on the screen; larger x-coordinates refer to points farther to the right. Coordinates can be specified either as an even number of parameters, or as a single list parameter containing an even number of x and y coordinate values.

TRANSFORMATIONS

Normally the origin of the canvas coordinate system is at the upper-left corner of the window containing the canvas. It is possible to adjust the origin of the canvas coordinate system relative to the origin of the window using the **xview** and **yview** widget commands; this is typically used for scrolling. Canvases do not support scaling or rotation of the canvas coordinate system relative to the window coordinate system.

Individual items may be moved or scaled using widget commands described below, but they may not be rotated.

Note that the default origin of the canvas's visible area is coincident with the origin for the whole window as that makes bindings using the mouse position easier to work with; you only need to use the **canvasx** and **canvasy** widget commands if you adjust the origin of the visible area. However, this also means that any focus ring (as controlled by the **−highlightthickness** option) and window border (as controlled by the **−borderwidth** option) must be taken into account before you get to the visible area of the canvas.

INDICES

Text items support the notion of an *index* for identifying particular positions within the item. In a similar fashion, line and polygon items support *index* for identifying, inserting and deleting subsets of their coordinates. Indices are used for commands such as inserting or deleting a range of characters or coordinates, and setting the insertion cursor position. An index may be specified in any of a number of ways, and different

types of items may support different forms for specifying indices. Text items support the following forms for an index; if you define new types of text-like items, it would be advisable to support as many of these forms as practical. Note that it is possible to refer to the character just after the last one in the text item; this is necessary for such tasks as inserting new text at the end of the item. Lines and Polygons do not support the insertion cursor and the selection. Their indices are supposed to be even always, because coordinates always appear in pairs.

number A decimal number giving the position of the desired character within the text item. 0 refers to the first character, 1 to the next character, and so on. If indexes are odd for lines and polygons, they will be automatically decremented by one. A number less than 0 is treated as if it were zero, and a number greater than the length of the text item is treated as if it were equal to the length of the text item. For polygons, numbers less than 0 or greater than the length of the coordinate list will be adjusted by adding or subtracting the length until the result is between zero and the length, inclusive.

end Refers to the character or coordinate just after the last one in the item (same as the number of characters or coordinates in the item).

insert Refers to the character just before which the insertion cursor is drawn in this item. Not valid for lines and polygons.

sel.first Refers to the first selected character in the item. If the selection is not in this item then this form is illegal.

sel.last Refers to the last selected character in the item. If the selection is not in this item then this form is illegal.

@x,y Refers to the character or coordinate at the point given by *x* and *y*, where *x* and *y* are specified in the coordinate system of the canvas. If*x* and *y* lie outside the coordinates co vered by the text item, then they refer to the first or last character in the line that is closest to the given point.

DASH PATTERNS

Many items support the notion of a dash pattern for outlines.

The first possible syntax is a list of integers. Each element represents the number of pixels of a line segment. Only the odd segments are drawn using the "outline" color. The other segments are drawn transparent.

The second possible syntax is a character list containing only 5 possible characters "**.,-_** ". The space can be used to enlarge the space between other line elements, and cannot occur as the first position in the string. Some examples:

−dash . → −dash {2 4}
−dash - → −dash {6 4}
−dash -. → −dash {6 4 2 4}
−dash -.. → −dash {6 4 2 4 2 4}
−dash {. } → −dash {2 8}
−dash , → −dash {4 4}

The main difference of this syntax with the previous is that it is shape-conserving. This means that all values in the dash list will be multiplied by the line width before display. This assures that "." will always be displayed as a dot and "-" always as a dash regardless of the line width.

On systems which support only a limited set of dash patterns, the dash pattern will be displayed as the closest dash pattern that is available. For example, on Windows only the first 4 of the above examples are available. The last 2 examples will be displayed identically to the first one.

WIDGET COMMAND

The **canvas** 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 *arg*s determine the exact behavior of the command. The following widget commands are possible for canvas widgets:

pathName **addtag** *tag searchSpec* ?*arg arg ...*?
For each item that meets the constraints specified by *searchSpec* and the *arg*s, add *tag* to the list of tags associated with the item if it is not already present on that list. It is possible that no items will satisfy the constraints given by *searchSpec* and *arg*s, in which case the command has no effect. This command returns an empty string as result. *Sear chSpec* and *arg*'s may take any of the following forms:

above *tagOrId*
Selects the item just after (above) the one given by *tagOrId* in the display list. If *tagOrId* denotes more than one item, then the last (topmost) of these items in the display list is used.

all Selects all the items in the canvas.

below *tagOrId*
Selects 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*? ?*start*?
Selects the 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 it must be a non-negative value. Any item closer than *halo* to the point is considered to overlap it. The *start* argument may be used to step circularly through all the closest items. If *start* 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.

enclosed *x1 y1 x2 y2*
Selects all the items completely enclosed within the rectangular region given by *x1*, *y1*, *x2*, and *y2*. *X1* must be no greater than *x2* and *y1* must be no greater than *y2*.

overlapping *x1 y1 x2 y2*
Selects all the items that overlap or are enclosed within the rectangular region given by *x1*, *y1*, *x2*, and *y2*. *X1* must be no greater than *x2* and *y1* must be no greater than *y2*.

withtag *tagOrId*
Selects all the items given by *tagOrId*.

pathName **bbox** *tagOrId* ?*tagOrId tagOrId ...*?
Returns a list with four elements giving an approximate bounding box for all the items named by the *tagOrId* arguments. The list has the form "*x1 y1 x2 y2*" such that the drawn areas of all the named elements are within the region bounded by *x1* on the left, *x2* on the right, *y1* on the top, and *y2* on the bottom. The return value may overestimate the actual bounding box by a few pixels. If no items match any of the *tagOrId* arguments or if the matching items have empty bounding boxes (i.e. they have nothing to display) then an empty string is returned.

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 **bind** command except that it operates on items in a canvas rather than entire widgets. See the **bind** manual entry for complete details on the syntax of *sequence* and the substitutions performed on *command* before invoking it. 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 *command* is omitted then the command returns the *command* associated with *tagOrId* and *sequence* (an error occurs if there is no such binding). 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**, and **KeyPress**) or virtual events. The handling of events in canvases uses the current item defined in **ITEM IDS AND TAGS** above. **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** widget command below for more on this). If a virtual event is used in a binding, that binding can trigger only if the virtual event is defined by an underlying mouse-related or keyboard-related event.

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. 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 canvas window using the **bind** command, then they are invoked in addition to bindings created for the canvas'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.

pathName **canvasx** *screenx* ?*gridspacing*?

Given a window x-coordinate in the canvas *screenx*, this command returns the canvas x-coordinate that is displayed at that location. If *gridspacing* is specified, then the canvas coordinate is rounded to the nearest multiple of *gridspacing* units.

pathName **canvasy** *screeny* ?*gridspacing*?

Given a window y-coordinate in the canvas *screeny* this command returns the canvas y-coordinate that is displayed at that location. If *gridspacing* is specified, then the canvas coordinate is rounded to the nearest multiple of *gridspacing* units.

pathName **cget** *option*

Returns the current value of the configuration option given by *option*. *Option* may have any of the values accepted by the **canvas** command.

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 **canvas** command.

pathName **coords** *tagOrId* ?*x0 y0 ...*?

pathName **coords** *tagOrId* ?*coordList*?

Query or modify the coordinates that define an item. If no coordinates are specified, 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.

Note that for rectangles, ovals and arcs the returned list of coordinates has a fixed order, namely the left, top, right and bottom coordinates, which may not be the order originally given. Also the coordinates are always returned in screen units with no units (that is, in pixels). So if the original coordinates were specified for instance in centimeters or inches, the returned values will nevertheless be in pixels.

pathName **create** *type x y* ?*x y ...*? ?*option value ...*?

pathName **create** *type coordList* ?*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 subsections on individual item types below for more on the syntax of this command. This command returns the id for the new item.

pathName **dchars** *tagOrId first* ?*last*?

For each item given by *tagOrId*, delete the characters, or coordinates, in the range given by *first* and *last*, inclusive. If some of the items given by *tagOrId* do not support indexing operations then they ignore this operation. Text items interpret *first* and *last* as indices to a character, line and polygon items interpret them as indices to a coordinate (an x,y pair). Indices are described in **INDICES** above. If*last* is omitted, it def aults to *first*. This command returns an empty string.

pathName **delete** ?*tagOrId tagOrId ...*?

Delete each of the items given by each *tagOrId*, and return an empty string.

pathName **dtag** *tagOrId* ?*tagToDelete*?

For each of the items given by *tagOrId*, delete the tag given by *tagToDelete* from the list of those associated with the item. If an item does not have the tag *tagToDelete* then the item is unaffected by the command. If *tagToDelete* is omitted then it defaults to *tagOrId*. This command returns an empty string.

pathName **find** *searchCommand* ?*arg arg ...*?

This command returns a list consisting of all the items that meet the constraints specified by *searchCommand* and *arg*'s. *SearchCommand* and *args* have any of the forms accepted by the **addtag** command. The items are returned in stacking order, with the lowest item first.

pathName **focus** ?*tagOrId*?

Set the keyboard focus for the canvas widget to the item given by *tagOrId*. If*ta gOrId* refers to several items, then the focus is set to the first such item in the display list that supports the insertion cursor. If*ta gOrId* does not refer to any items, or if none of them support the insertion cursor, then the focus is not 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.

Once the focus has been set to an item, the item will display the insertion cursor and all keyboard events will be directed to that item. The focus item within a canvas and the focus window on the screen (set with the **focus** command) are totally independent: a given item does not actually have the input focus unless (a) its canvas is the focus window and (b) the item is the focus item within the canvas. In most cases it is advisable to follow the **focus** widget command with the **focus** command to set the focus window to the canvas (if it was not there already).

pathName **gettags** *tagOrId*

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

pathName **icursor** *tagOrId index*

Set the position of the insertion cursor for the item(s) given by *tagOrId* to just before the character whose position is given by *index*. If some or all of the items given by *tagOrId* do not support an insertion cursor then this command has no effect on them. See **INDICES** above for a description of the legal forms for *index*. Note: the insertion cursor is only displayed in an item if that item currently has the keyboard focus (see the **focus** widget command, above), but the cursor position may be set even when the item does not have the focus. This command returns an empty string.

pathName **imove** *tagOrId index x y*

This command causes the *index*'th coordinate of each of the items indicated by *tagOrId* to be relocated to the location (*x*,*y*). Each item interprets*inde x* independently according to the rules described in **INDICES** above. Out of the standard set of items, only line and polygon items may have their coordinates relocated this way.

pathName **index** *tagOrId index*

This command returns a decimal string giving the numerical index within *tagOrId* corresponding to *index*. *Index* gives a textual description of the desired position as described in **INDICES** above. Text items interpret *index* as an index to a character, line and polygon items interpret it as an index to a coordinate (an x,y pair). The return value is guaranteed to lie between 0 and the number of characters, or coordinates, within the item, inclusive. If*ta gOrId* refers to multiple items, then the index is processed in the first of these items that supports indexing operations (in display list order).

pathName **insert** *tagOrId beforeThis string*

For each of the items given by *tagOrId*, if the item supports text or coordinate, insertion then *string* is inserted into the item's text just before the character, or coordinate, whose index is *beforeThis*. Text items interpret *beforeThis* as an index to a character, line and polygon items interpret it as an index to a coordinate (an x,y pair). For lines and polygons the *string* must be a valid coordinate sequence. See **INDICES** above for information about the forms allowed for *beforeThis*. This command returns an empty string.

pathName **itemcget** *tagOrId option*

Returns the current value of the configuration option for the item given by *tagOrId* whose name is *option*. This command is similar to the**cget** widget command e xcept that it applies to a particular item rather than the widget as a whole. *Option* may ha ve any of the values accepted by the **create** widget command when the item was created. If *tagOrId* is a tag that refers to more than one item, the first (lowest) such item is used.

pathName **itemconfigure** *tagOrId* ?*option*? ?*value*? ?*option value ...*?

This command is similar to the **configure** widget command except that it modifies item-specific options for the items given by *tagOrId* instead of modifying options for the overall canvas widget. If no *option* is specified, returns a list describing all of the available options for the first item given by *tagOrId* (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 each of the items given by *tagOrId*; in this case the command returns an empty string. The *option*s and *value*s are the same as those permissible in the **create** widget command when the item(s) were created; see the sections describing individual item types below for details on the legal options.

pathName **lower** *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*ta gOrId* 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 (lowest) of these items in the display list is used as the destination location for

the moved items. Note: this command has no effect on window items. Window items always obscure other item types, and the stacking order of window items is determined by the **raise** command and **lower** command, not the **raise** widget command and **lower** widget command for canvases. This command returns an empty string.

pathName **move** *tagOrId xAmount yAmount*

Move each of the items given by *tagOrId* in the canvas coordinate space by adding *xAmount* to the x-coordinate of each point associated with the item and *yAmount* to the y-coordinate of each point associated with the item. This command returns an empty string.

pathName **moveto** *tagOrId xPos yPos*

Move the items given by *tagOrId* in the canvas coordinate space so that the first coordinate pair (the upper-left corner of the bounding box) of the first item (the lowest in the display list) with tag *tagOrId* is located at position (*xPos*,*yPos*). *xPos* and *yPos* may be the empty string, in which case the corresponding coordinate will be unchanged. All items matching *tagOrId* remain in the same positions relative to each other. This command returns an empty string.

pathName **postscript** ?*option value option value ...*?

Generate a Postscript representation for part or all of the canvas. If the **−file** option is specified then the Postscript is written to a file and an empty string is returned; otherwise the Postscript is returned as the result of the command. If the interpreter that owns the canvas is marked as safe, the operation will fail because safe interpreters are not allowed to write files. If the **−channel** option is specified, the argument denotes the name of a channel already opened for writing. The Postscript is written to that channel, and the channel is left open for further writing at the end of the operation. The Postscript is created in Encapsulated Postscript form using version 3.0 of the Document Structuring Conventions. Note: by default Postscript is only generated for information that appears in the canvas's window on the screen. If the canvas is freshly created it may still have its initial size of 1x1 pixel so nothing will appear in the Postscript. To get around this problem either invoke the **update** command to wait for the canvas window to reach its final size, or else use the **−width** and **−height** options to specify the area of the canvas to print. The *option−value* argument pairs provide additional information to control the generation of Postscript. The following options are supported:

−channel *channelName*

Specifies the name of the channel to which to write the Postscript. If this option and the **−file** option are not specified then the Postscript is returned as the result of the command.

−colormap *varName*

VarName must be the name of an array variable that specifies a color mapping to use in the Postscript. Each element of *varName* must consist of Postscript code to set a particular color value (e.g. "**1.0 1.0 0.0 setrgbcolor**"). When outputting color information in the Postscript, Tk checks to see if there is an element of *varName* with the same name as the color. If so, Tk uses the value of the element as the Postscript command to set the color. If this option has not been specified, or if there is no entry in*varName* for a gi ven color, then Tk uses the red, green, and blue intensities from the X color.

−colormode *mode*

Specifies how to output color information. *Mode* must be either **color** (for full color output), **gray** (convert all colors to their gray-scale equivalents) or **mono** (convert all colors to black or white).

−file *fileName*

Specifies the name of the file in which to write the Postscript. If this option and the **−channel** option are not specified then the Postscript is returned as the result of the command.

−fontmap *varName*

VarName must be the name of an array variable that specifies a font mapping to use in the Postscript. Each element of*varName* must consist of a Tcl list with tw o elements, which are the name and point size of a Postscript font. When outputting Postscript commands for a particular font, Tk checks to see if *varName* contains an element with the same name as the font. If there is such an element, then the font information contained in that element is used in the Postscript. Otherwise Tk attempts to guess what Postscript font to use. Tk's guesses generally only work for well-known fonts such as Times and Helvetica and Courier, and only if the X font name does not omit any dashes up through the point size. For example, **−*−Courier−Bold−R−Normal−−*−120−*** will work but ***Courier−Bold−R−Normal*120*** will not; Tk needs the dashes to parse the font name).

−height *size*

Specifies the height of the area of the canvas to print. Defaults to the height of the canvas window.

−pageanchor *anchor*

Specifies which point of the printed area of the canvas should appear over the positioning point on the page (which is given by the **−pagex** and **−pagey** options). For example, **−pageanchor n** means that the top center of the area of the canvas being printed (as it appears in the canvas window) should be over the positioning point. Defaults to **center**.

−pageheight *size*

Specifies that the Postscript should be scaled in both x and y so that the printed area is *size* high on the Postscript page. *Size* consists of a floating-point number followed by **c** for centimeters, **i** for inches, **m** for millimeters, or **p** or nothing for printer's points (1/72 inch). Defaults to the height of the printed area on the screen. If both **−pageheight** and **−pagewidth** are specified then the scale factor from **−pagewidth** is used (non-uniform scaling is not implemented).

−pagewidth *size*

Specifies that the Postscript should be scaled in both x and y so that the printed area is *size* wide on the Postscript page. *Size* has the same form as for **−pageheight**. Defaults to the width of the printed area on the screen. If both **−pageheight** and **−pagewidth** are specified then the scale factor from **−pagewidth** is used (non-uniform scaling is not implemented).

−pagex *position*

Position gives the x-coordinate of the positioning point on the Postscript page, using any of the forms allowed for **−pageheight**. Used in conjunction with the**−pagey** and **−pageanchor** options to determine where the printed area appears on the Postscript page. Defaults to the center of the page.

−pagey *position*

Position gives the y-coordinate of the positioning point on the Postscript page, using any of the forms allowed for **−pageheight**. Used in conjunction with the**−pagex** and **−pageanchor** options to determine where the printed area appears on the Postscript page. Defaults to the center of the page.

−rotate *boolean*

Boolean specifies whether the printed area is to be rotated 90 degrees. In non-rotated output the x-axis of the printed area runs along the short dimension of the page ("portrait" orientation); in rotated output the x-axis runs along the long dimension of the page ("landscape" orientation). Defaults to non-rotated.

−width *size*
Specifies the width of the area of the canvas to print. Defaults to the width of the canvas window.

−x *position*
Specifies the x-coordinate of the left edge of the area of the canvas that is to be printed, in canvas coordinates, not window coordinates. Defaults to the coordinate of the left edge of the window.

−y *position*
Specifies the y-coordinate of the top edge of the area of the canvas that is to be printed, in canvas coordinates, not window coordinates. Defaults to the coordinate of the top edge of the window.

pathName **raise** *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*ta gOrId* refers to more than one item then all are moved but the relative order of the moved items will not be changed. *Abo veThis* 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. This command returns an empty string.

Note: this command has no effect on window items. Window items always obscure other item types, and the stacking order of window items is determined by the **raise** command and **lower** command, not the **raise** widget command and **lower** widget command for canvases.

pathName **rchars** *tagOrId first last string*
This command causes the text or coordinates between *first* and *last* for each of the items indicated by *tagOrId* to be replaced by *string*. Each item interprets *first* and *last* independently according to the rules described in **INDICES** above. Out of the standard set of items, text items support this operation by altering their text as directed, and line and polygon items support this operation by altering their coordinate list (in which case *string* should be a list of coordinates to use as a replacement). The other items ignore this operation.

pathName **scale** *tagOrId xOrigin yOrigin xScale yScale*
Rescale the coordinates of all of the items given by *tagOrId* in canvas coordinate space. *XOrigin* and *yOrigin* identify the origin for the scaling operation and *xScale* and *yScale* identify the scale factors for x- and y-coordinates, respectively (a scale factor of 1.0 implies no change to that coordinate). For each of the points defining each item, the x-coordinate is adjusted to change the distance from *xOrigin* by a factor of *xScale*. Similarly, each y-coordinate is adjusted to change the distance from *yOrigin* by a factor of *yScale*. This command returns an empty string.

Note that some items have only a single pair of coordinates (e.g., text, images and windows) and so scaling of them by this command can only move them around.

pathName **scan** *option args*
This command is used to implement scanning on canvases. It has two forms, depending on *option*:

pathName **scan mark** *x y*
Records *x* and *y* and the canvas's current view; used in conjunction with later **scan dragto** commands. Typically this command is associated with a mouse button press in the widget and *x* and *y* are the coordinates of the mouse. It returns an empty string.

pathName **scan dragto** *x y ?gain?*
This command computes the difference between its *x* and *y* arguments (which are typically mouse coordinates) and the *x* and *y* arguments to the last **scan mark** command for the widget. It then adjusts the view by *gain* times the difference in coordinates, where *gain* defaults to 10. This command is typically associated with mouse motion events in the widget, to produce the effect of dragging the canvas at high speed through its window. The return value is an empty string.

pathName **select** *option* ?*tagOrId arg*?

Manipulates the selection in one of several ways, depending on *option*. The command may take any of the forms described below. In all of the descriptions below, *tagOrId* must refer to an item that supports indexing and selection; if it refers to multiple items then the first of these that supports indexing and the selection is used. *Inde x* gives a textual description of a position within *tagOrId*, as described in **INDICES** above.

pathName **select adjust** *tagOrId index*

Locate the end of the selection in *tagOrId* nearest to the character given by *index*, and adjust that end of the selection to be at *index* (i.e. including but not going beyond *index*). The other end of the selection is made the anchor point for future **select to** commands. If the selection is not currently in *tagOrId* then this command behaves the same as the **select to** widget command. Returns an empty string.

pathName **select clear**

Clear the selection if it is in this widget. If the selection is not in this widget then the command has no effect. Returns an empty string.

pathName **select from** *tagOrId index*

Set the selection anchor point for the widget to be just before the character given by *index* in the item given by *tagOrId*. This command does not change the selection; it just sets the fixed end of the selection for future **select to** commands. Returns an empty string.

pathName **select item**

Returns the id of the selected item, if the selection is in an item in this canvas. If the selection is not in this canvas then an empty string is returned.

pathName **select to** *tagOrId index*

Set the selection to consist of those characters of *tagOrId* between the selection anchor point and *index*. The new selection will include the character given by *index*; it will include the character given by the anchor point only if *index* is greater than or equal to the anchor point. The anchor point is determined by the most recent **select adjust** or **select from** command for this widget. If the selection anchor point for the widget is not currently in *tagOrId*, then it is set to the same character given by *index*. Returns an empty string.

pathName **type** *tagOrId*

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

pathName **xview** ?*args*?

This command is used to query and change the horizontal position of the information displayed in the canvas's window. It can take any of the following forms:

pathName **xview**

Returns a list containing two elements. Each element is a real fraction between 0 and 1; together they describe the horizontal span that is visible in the window. For example, if the first element is .2 and the second element is .6, 20% of the canvas's area (as defined by the **−scrollregion** option) is off-screen to the left, the middle 40% is visible in the window, and 40% of the canvas is off-screen to the right. These are the same values passed to scrollbars via the **−xscrollcommand** option.

pathName **xview moveto** *fraction*

Adjusts the view in the window so that *fraction* of the total width of the canvas is off-screen to the left. *Fraction* must be a fraction between 0 and 1.

pathName **xview scroll** *number what*

This command shifts the view in the window left or right according to *number* and *what*. *Number* must be an integer. *What* must be either **units** or **pages** or an abbreviation of one of these. If *what* is **units**, the view adjusts left or right in units of the **xScrollIncrement** option, if it is greater than zero, or in units of one-tenth the window's width otherwise. If *what is* **pages** then the view adjusts in units of nine-tenths the window's width. If *number* is negative then information farther to the left becomes visible; if it is positive then information farther to the right becomes visible.

pathName **yview ?***args*?

This command is used to query and change the vertical position of the information displayed in the canvas's window. It can take any of the following forms:

pathName **yview**

Returns a list containing two elements. Each element is a real fraction between 0 and 1; together they describe the vertical span that is visible in the window. For example, if the first element is .6 and the second element is 1.0, the lowest 40% of the canvas's area (as defined by the **−scrollregion** option) is visible in the window. These are the same values passed to scrollbars via the **−yscrollcommand** option.

pathName **yview moveto** *fraction*

Adjusts the view in the window so that *fraction* of the canvas's area is off-screen to the top. *Fraction* is a fraction between 0 and 1.

pathName **yview scroll** *number what*

This command adjusts the view in the window up or down according to *number* and *what*. *Number* must be an integer. *What* must be either **units** or **pages**. If*what* is **units**, the view adjusts up or down in units of the **yScrollIncrement** option, if it is greater than zero, or in units of one-tenth the window's height otherwise. If *what* is **pages** then the view adjusts in units of nine-tenths the window's height. If *number* is negative then higher information becomes visible; if it is positive then lower information becomes visible.

OVERVIEW OF ITEM TYPES

The sections below describe the various types of items supported by canvas widgets. Each item type is characterized by two things: first, the form of the **create** command used to create instances of the type; and second, a set of configuration options for items of that type, which may be used in the **create** and **itemconfigure** widget commands. Most items do not support indexing or selection or the commands related to them, such as **index** and **insert**. Where items do support these facilities, it is noted explicitly in the descriptions below. At present, text, line and polygon items provide this support. For lines and polygons the indexing facility is used to manipulate the coordinates of the item.

COMMON ITEM OPTIONS

Many items share a common set of options. These options are explained here, and then referred to be each widget type for brevity.

−anchor *anchorPos*

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

−dash *pattern*

−activedash *pattern*

−disableddash *pattern*

This option specifies dash patterns for the normal, active state, and disabled state of an item. *pattern* may have any of the forms accepted by **Tk_GetDash**. If the dash options are omitted then the

default is a solid outline. See **DASH PATTERNS** for more information.

−dashoffset *offset*
The starting *offset* in pixels into the pattern provided by the **−dash** option. **−dashoffset** is ignored if there is no **−dash** pattern. The *offset* may have any of the forms described in the **COORDINATES** section above.

−fill *color*

−activefill *color*

−disabledfill *color*
Specifies the color to be used to fill item's area. in its normal, active, and disabled states. The even-odd fill rule is used. *Color* may ha ve any of the forms accepted by **Tk_GetColor**. For the line item, it specifies the color of the line drawn. For the text item, it specifies the foreground color of the text. If*color* is an empty string (the def ault for all canvas items except line and text), then the item will not be filled.

−outline *color*

−activeoutline *color*

−disabledoutline *color*
This option specifies the color that should be used to draw the outline of the item in its normal, active and disabled states. *Color* may ha ve any of the forms accepted by **Tk_GetColor**. If*color* is specified as an empty string then no outline is drawn for the item.

−offset *offset*
Specifies the offset of stipples. The offset value can be of the form **x,y** or *side*, where side can be **n**, **ne**, **e**, **se**, **s**, **sw**, **w**, **nw**, or **center**. In the first case the origin is the origin of the toplevel of the current window. For the canvas itself and canvas objects the origin is the canvas origin, but putting **#** in front of the coordinate pair indicates using the toplevel origin instead. For canvas objects, the **−offset** option is used for stippling as well. For the line and polygon canvas items you can also specify an index as argument, which connects the stipple origin to one of the coordinate points of the line/polygon. Note that stipple offsets are *only supported on X11*; they are silently ignored on other platforms.

−outlinestipple *bitmap*

−activeoutlinestipple *bitmap*

−disabledoutlinestipple *bitmap*
This option specifies stipple patterns that should be used to draw the outline of the item in its normal, active and disabled states. Indicates that the outline for the item should be drawn with a stipple pattern; *bitmap* specifies the stipple pattern to use, in any of the forms accepted by **Tk_GetBitmap**. If the**−outline** option has not been specified then this option has no ef fect. If*bitmap* is an empty string (the default), then the outline is drawn in a solid fashion. *Note that stipples are not well supported on platforms that do not use X11 as their drawing API.*

−outlineoffset *offset*
Specifies the offset of the stipple pattern used for outlines, in the same way that the **−outline** option controls fill stipples. (See the **−outline** option for a description of the syntax of *offset*.)

−stipple *bitmap*

−activestipple *bitmap*

−disabledstipple *bitmap*
This option specifies stipple patterns that should be used to fill the item in its normal, active and disabled states. *bitmap* specifies the stipple pattern to use, in an y of the forms accepted by **Tk_GetBitmap**. If the**−fill** option has not been specified then this option has no ef fect. If*bitmap* is an empty string (the default), then filling is done in a solid fashion. For the text item, it affects

the actual text. *Note that stipples are not well supported on platforms that do not use X11 as their drawing API.*

−state *state*
This allows an item to override the canvas widget's global *state* option. It takes the same values: *normal*, *disabled* or *hidden*.

−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.

−width *outlineWidth*

−activewidth *outlineWidth*

−disabledwidth *outlineWidth*
Specifies the width of the outline to be drawn around the item's region, in its normal, active and disabled states. *outlineWidth* may be in any of the forms described in the **COORDINATES** section above. If the **−outline** option has been specified as an empty string then this option has no effect. This option defaults to 1.0. For arcs, wide outlines will be drawn centered on the edges of the arc's region.

STANDARD ITEM TYPES

ARC ITEMS

Items of type **arc** appear on the display as arc-shaped regions. An arc is a section of an oval delimited by two angles (specified by the **−start** and **−extent** options) and displayed in one of several ways (specified by the **−style** option). Arcs are created with widget commands of the following form:

pathName **create arc** *x1 y1 x2 y2* ?*option value* ...?
pathName **create arc** *coordList* ?*option value* ...?

The arguments *x1*, *y1*, *x2*, and *y2* or *coordList* give the coordinates of two diagonally opposite corners of a rectangular region enclosing the oval that defines the arc. 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. An arc item becomes the current item when the mouse pointer is over any part that is painted or (when fully transparent) that would be painted if both the **−fill** and **−outline** options were non-empty.

The following standard options are supported by arcs:

−dash
−activedash
−disableddash
−dashoffset
−fill
−activefill
−disabledfill
−offset
−outline
−activeoutline
−disabledoutline
−outlineoffset
−outlinestipple
−activeoutlinestipple
−disabledoutlinestipple
−stipple
−activestipple
−disabledstipple
−state
−tags
−width
−activewidth
−disabledwidth

The following extra options are supported for arcs:

−extent *degrees*
Specifies the size of the angular range occupied by the arc. The arc's range extends for *degrees* degrees counter-clockwise from the starting angle given by the **−start** option. *Degrees* may be negative. If it is greater than 360 or less than −360, then *degrees* modulo 360 is used as the extent.

−start *degrees*
Specifies the beginning of the angular range occupied by the arc. *De grees* is given in units of degrees measured counter-clockwise from the 3-o'clock position; it may be either positive or negative.

−style *type*
Specifies how to draw the arc. If *type* is **pieslice** (the default) then the arc's region is defined by a section of the oval's perimeter plus two line segments, one between the center of the oval and each end of the perimeter section. If *type* is **chord** then the arc's region is defined by a section of the oval's perimeter plus a single line segment connecting the two end points of the perimeter section. If *type* is **arc** then the arc's region consists of a section of the perimeter alone. In this last case the **−fill** option is ignored.

BITMAP ITEMS

Items of type **bitmap** appear on the display as images with two colors, foreground and background. Bitmaps are created with widget commands of the following form:

pathName **create bitmap** *x y* ?*option value ...*?
pathName **create bitmap** *coordList* ?*option value ...*?

The arguments *x* and *y* or *coordList* (which must have two elements) specify the coordinates of a point used to position the bitmap on the display, as controlled by the **−anchor** option. 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. A bitmap item becomes the current item when the mouse pointer is over any part of its bounding box.

The following standard options are supported by bitmaps:

−anchor **−state**
−tags

The following extra options are supported for bitmaps:

−background *color*

−activebackground *color*

−disabledbackground *color*
Specifies the color to use for each of the bitmap's "0" valued pixels in its normal, active and disabled states. *Color* may ha ve any of the forms accepted by **Tk_GetColor**. If this option is not specified, or if it is specified as an empty string, then nothing is displayed where the bitmap pixels are 0; this produces a transparent effect.

−bitmap *bitmap*

−activebitmap *bitmap*

−disabledbitmap *bitmap*
Specifies the bitmaps to display in the item in its normal, active and disabled states. *Bitmap* may have any of the forms accepted by **Tk_GetBitmap**.

−foreground *color*

−activeforeground *color*

−disabledforeground *color*
Specifies the color to use for each of the bitmap's "1" valued pixels in its normal, active and disabled states. *Color* may have any of the forms accepted by **Tk_GetColor**.

IMAGE ITEMS

Items of type **image** are used to display images on a canvas. Images are created with widget commands of the following form:

pathName **create image** *x y* ?*option value* ...?
pathName **create image** *coordList* ?*option value* ...?

The arguments *x* and *y* or *coordList* specify the coordinates of a point used to position the image on the display, as controlled by the **−anchor** option. 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. An image item becomes the current item when the mouse pointer is over any part of its bounding box.

The following standard options are supported by images:

−anchor **−state**
−tags

The following extra options are supported for images:

−image *name*

−activeimage *name*

−disabledimage *name*

Specifies the name of the images to display in the item in is normal, active and disabled states. This image must have been created previously with the **image create** command.

LINE ITEMS

Items of type **line** appear on the display as one or more connected line segments or curves. Line items support coordinate indexing operations using the **dchars**, **index** and **insert** widget commands. Lines are created with widget commands of the following form:

pathName **create line** *x1 y1... xn yn* ?*option value* ...?
pathName **create line** *coordList* ?*option value* ...?

The arguments *x1* through *yn* or *coordList* 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. A line item is the current item whenever the mouse pointer is over any segment of the line, whether drawn or not and whether or not the line is smoothed.

The following standard options are supported by lines:

−dash **−activedash**
−disableddash **−dashoffset**
−fill **−activefill**
−disabledfill **−stipple**
−activestipple **−disabledstipple**
−state **−tags**
−width **−activewidth**
−disabledwidth

The following extra options are supported for lines:

−arrow *where*

Indicates whether or not arrowheads are to be drawn at one or both ends of the line. *Wher e* 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**. When requested to draw an arrowhead, Tk internally adjusts the

corresponding line end point so that the rendered line ends at the neck of the arrowhead rather than at its tip so that the line doesn't extend past the edge of the arrowhead. This may trigger a **Leave** event if the mouse is hovering this line end. Conversely, when removing an arrowhead Tk adjusts the corresponding line point the other way round, which may trigger an **Enter** event.

−arrowshape *shape*
This option indicates how to draw arrowheads. The*shape* ar gument must be a list with three elements, each specifying a distance in any of the forms described in the **COORDINATES** section above. 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 is not specified then Tk picks a "reasonable" shape.

−capstyle *style*
Specifies the ways in which caps are to be drawn at the endpoints of the line. *Style* may ha ve any of the forms accepted by **Tk_GetCapStyle** (**butt**, **projecting**, or **round**). If this option is not specified then it defaults to **butt**. Where arrowheads are drawn the cap style is ignored.

−joinstyle *style*
Specifies the ways in which joints are to be drawn at the vertices of the line. *Style* may ha ve any of the forms accepted by **Tk_GetJoinStyle** (**bevel**, **miter**, or **round**). If this option is not specified then it defaults to **round**. If the line only contains two points then this option is irrelevant.

−smooth *smoothMethod*
smoothMethod must have one of the forms accepted by **Tcl_GetBoolean** or a line smoothing method. Only**true** and **raw** are supported in the core (with **bezier** being an alias for **true**), b ut more can be added at runtime. If a boolean false value or empty string is given, no smoothing is applied. A boolean truth value assumes **true** smoothing. If the smoothing method is **true**, this indicates that the line should be drawn as a curve, rendered as a set of quadratic splines: one spline is drawn for the first and second line segments, one for the second and third, and so on. Straight-line segments can be generated within a curve by duplicating the end-points of the desired line segment. If the smoothing method is**raw**, this indicates that the line should also be dra wn as a curve but where the list of coordinates is such that the first coordinate pair (and every third coordinate pair thereafter) is a knot point on a cubic Bezier curve, and the other coordinates are control points on the cubic Bezier curve. Straight line segments can be generated within a curve by making control points equal to their neighbouring knot points. If the last point is a control point and not a knot point, the point is repeated (one or two times) so that it also becomes a knot point.

−splinesteps *number*
Specifies the degree of smoothness desired for curves: each spline will be approximated with *number* line segments. This option is ignored unless the **−smooth** option is true or **raw**.

OVAL ITEMS

Items of type **oval** appear as circular or oval regions on the display. Each oval may have an outline, a fill, or both. Ovals are created with widget commands of the following form:

pathName **create oval** *x1 y1 x2 y2* ?*option value* ...?
pathName **create oval** *coordList* ?*option value* ...?

The arguments *x1*, *y1*, *x2*, and *y2* or *coordList* give the coordinates of two diagonally opposite corners of a rectangular region enclosing the oval. The oval will include the top and left edges of the rectangle not the lower or right edges. If the region is square then the resulting oval is circular; otherwise it is elongated in shape. 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. An oval item becomes the current item when the mouse pointer is over any part that is painted or (when fully transparent) that would be painted if both the **−fill** and **−outline** options were non-empty.

The following standard options are supported by ovals:

−dash
−disableddash
−fill
−disabledfill
−outline
−disabledoutline
−outlinestipple
−disabledoutlinestipple
−activestipple
−state
−width
−disabledwidth
−activedash
−dashoffset
−activefill
−offset
−activeoutline
−outlineoffset
−activeoutlinestipple
−stipple
−disabledstipple
−tags
−activewidth

There are no oval-specific options.

POLYGON ITEMS

Items of type **polygon** appear as polygonal or curved filled regions on the display. Polygon items support coordinate indexing operations using the **dchars**, **index** and **insert** widget commands. Polygons are created with widget commands of the following form:

pathName **create polygon** *x1 y1 ... xn yn* ?*option value ...*?
pathName **create polygon** *coordList* ?*option value ...*?

The arguments *x1* through *yn* or *coordList* specify the coordinates for three or more points that define a polygon. The first point should not be repeated as the last to close the shape; Tk will automatically close the periphery between the first and last points. 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. A polygon item is the current item whenever the mouse pointer is over any part of the polygon, whether drawn or not and whether or not the outline is smoothed.

The following standard options are supported by polygons:

−dash
−disableddash
−fill
−disabledfill
−outline
−disabledoutline
−outlinestipple
−disabledoutlinestipple
−activestipple
−state
−width
−disabledwidth
−activedash
−dashoffset
−activefill
−offset
−activeoutline
−outlineoffset
−activeoutlinestipple
−stipple
−disabledstipple
−tags
−activewidth

The following extra options are supported for polygons:

−joinstyle *style*
Specifies the ways in which joints are to be drawn at the vertices of the outline. *Style* may ha ve any of the forms accepted by **Tk_GetJoinStyle** (**bevel**, **miter**, or **round**). If this option is not specified then it defaults to **round**.

−smooth *boolean*
Boolean must have one of the forms accepted by **Tcl_GetBoolean** or a line smoothing method. Only **true** and **raw** are supported in the core (with **bezier** being an alias for **true**), but more can be added at runtime. If a boolean false value or empty string is given, no smoothing is applied. A

boolean truth value assumes **true** smoothing. If the smoothing method is **true**, this indicates that the polygon should be drawn as a curve, rendered as a set of quadratic splines: one spline is drawn for the first and second line segments, one for the second and third, and so on. Straight-line segments can be generated within a curve by duplicating the end-points of the desired line segment. If the smoothing method is **raw**, this indicates that the polygon should also be drawn as a curve but where the list of coordinates is such that the first coordinate pair (and every third coordinate pair thereafter) is a knot point on a cubic Bezier curve, and the other coordinates are control points on the cubic Bezier curve. Straight line segments can be generated within a curve by making control points equal to their neighbouring knot points. If the last point is not the second point of a pair of control points, the point is repeated (one or two times) so that it also becomes the second point of a pair of control points (the associated knot point will be the first control point).

−splinesteps *number*

Specifies the degree of smoothness desired for curves: each spline will be approximated with *number* line segments. This option is ignored unless the **−smooth** option is true or **raw**.

Polygon items are different from other items such as rectangles, ovals and arcs in that interior points are considered to be "inside" a polygon (e.g. for purposes of the **find closest** and **find overlapping** widget commands) even if it is not filled. For most other item types, an interior point is considered to be inside the item only if the item is filled or if it has neither a fill nor an outline. If you would like an unfilled polygon whose interior points are not considered to be inside the polygon, use a line item instead.

RECTANGLE ITEMS

Items of type **rectangle** appear as rectangular regions on the display. Each rectangle may have an outline, a fill, or both. Rectangles are created with widget commands of the following form:

pathName **create rectangle** *x1 y1 x2 y2* ?*option value ...*?
pathName **create rectangle** *coordList* ?*option value ...*?

The arguments *x1*, *y1*, *x2*, and *y2* or *coordList* (which must have four elements) give the coordinates of two diagonally opposite corners of the rectangle (the rectangle will include its upper and left edges but not its lower or right edges). 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. A rectangle item becomes the current item when the mouse pointer is over any part that is painted or (when fully transparent) that would be painted if both the **−fill** and **−outline** options were non-empty.

The following standard options are supported by rectangles:

−dash
−activedash
−disableddash
−dashoffset
−fill
−activefill
−disabledfill
−offset
−outline
−activeoutline
−disabledoutline
−outlineoffset
−outlinestipple
−activeoutlinestipple
−disabledoutlinestipple
−stipple
−activestipple
−disabledstipple
−state
−tags
−width
−activewidth
−disabledwidth

There are no rectangle-specific options.

TEXT ITEMS

A text item displays a string of characters on the screen in one or more lines. Text items support indexing, editing and selection through the **dchars** widget command, the **focus** widget command, the **icursor** widget command, the **index** widget command, the **insert** widget command, and the **select** widget command. Text

items are created with widget commands of the following form:

pathName **create text** *x y* ?*option value ...*?
pathName **create text** *coordList* ?*option value ...*?

The arguments *x* and *y* or *coordList* (which must have two elements) specify the coordinates of a point used to position the text on the display (see the options below for more information on how text is displayed). 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. A text item becomes the current item when the mouse pointer is over any part of its bounding box.

The following standard options are supported by text items:

−anchor **−fill**
−activefill **−disabledfill**
−stipple **−activestipple**
−disabledstipple **−state**
−tags

The following extra options are supported for text items:

−angle *rotationDegrees*
RotationDegrees tells how many degrees to rotate the text anticlockwise about the positioning point for the text; it may have any floating-point value from 0.0 to 360.0. For example, if *rotationDegrees* is **90**, then the text will be drawn vertically from bottom to top. This option defaults to **0.0**.

−font *fontName*
Specifies the font to use for the text item. *FontName* may be any string acceptable to **Tk_GetFont**. If this option is not specified, it defaults to a system-dependent font.

−justify *how*
Specifies how to justify the text within its bounding region. *How* must be one of the values **left**, **right**, or **center**. This option will only matter if the text is displayed as multiple lines. If the option is omitted, it defaults to **left**.

−text *string*
String specifies the characters to be displayed in the text item. Newline characters cause line breaks. The characters in the item may also be changed with the**insert** and **delete** widget commands. This option defaults to an empty string.

−underline
Specifies the integer index of a character within the text to be underlined. 0 corresponds to the first character of the text displayed, 1 to the next character, and so on. −1 means that no underline should be drawn (if the whole text item is to be underlined, the appropriate font should be used instead).

−width *lineLength*
Specifies a maximum line length for the text, in any of the forms described in the **COORDINATES** section above. If this option is zero (the default) the text is broken into lines only at newline characters. However, if this option is non-zero then any line that would be longer than *lineLength* is broken just before a space character to make the line shorter than *lineLength*; the space character is treated as if it were a newline character.

WINDOW ITEMS

Items of type **window** cause a particular window to be displayed at a given position on the canvas. Window items are created with widget commands of the following form:

pathName **create window** *x y* ?*option value ...*?

pathName **create window** *coordList* ?*option value* ...?

The arguments *x* and *y* or *coordList* (which must have two elements) specify the coordinates of a point used to position the window on the display, as controlled by the **−anchor** option. 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. Theoretically, a window item becomes the current item when the mouse pointer is over any part of its bounding box, but in practice this typically does not happen because the mouse pointer ceases to be over the canvas at that point.

The following standard options are supported by window items:

−anchor **−state**
−tags

The following extra options are supported for window items:

−height *pixels*
Specifies the height to assign to the item's window. *Pixels* may have any of the forms described in the **COORDINATES** section above. If this option is not specified, or if it is specified as zero, then the window is given whatever height it requests internally.

−width *pixels*
Specifies the width to assign to the item's window. *Pixels* may have any of the forms described in the **COORDINATES** section above. If this option is not specified, or if it is specified as zero, then the window is given whatever width it requests internally.

−window *pathName*
Specifies the window to associate with this item. The window specified by *pathName* must either be a child of the canvas widget or a child of some ancestor of the canvas widget. *P athName* may not refer to a top-level window.

Note: due to restrictions in the ways that windows are managed, it is not possible to draw other graphical items (such as lines and images) on top of window items. A window item always obscures any graphics that overlap it, regardless of their order in the display list. Also note that window items, unlike other canvas items, are not clipped for display by their containing canvas's border, and are instead clipped by the parent widget of the window specified by the **−window** option; when the parent widget is the canvas, this means that the window item can overlap the canvas's border.

APPLICATION-DEFINED ITEM TYPES

It is possible for individual applications to define new item types for canvas widgets using C code. See the documentation for **Tk_CreateItemType**.

BINDINGS

In the current implementation, new canvases are not given any default behavior: you will have to execute explicit Tcl commands to give the canvas its behavior.

CREDITS

Tk's canvas widget is a blatant ripoff of ideas from Joel Bartlett's *ezd* program. *Ezd* provides structured graphics in a Scheme environment and preceded canvases by a year or two. Its simple mechanisms for placing and animating graphical objects inspired the functions of canvases.

SEE ALSO

bind(n), font(n), image(n), scrollbar(n)

KEYWORDS

canvas, widget