Zinc, an advanced scriptable Canvas.
The pre 3.3 Reference Manual.

[CENA technical Note NT03-532]

Patrick Lecoanet, Christophe Mertz

1 Septembre 2004

Contents

Chapter 1

Introduction

1.1 What is TkZinc ?

TkZinc widgets are very similar to Tk Canvases in that they support structured graphics. Like the Canvas, TkZinc implements items used to display graphical entities. Those items can be manipulated and bindings can be associated with them to implement interaction behaviors. But unlike the Canvas, TkZinc can structure the items in a hierarchy (with the use of group items), has support for affine 2D transforms (i.e. translation, scaling, and rotation), clipping can be set for sub-trees of the item hierarchy, the item set is quite more powerful including field specific items for Air Traffic systems and new rendering techniques such as transparency and gradients. If needed, it is also possible to extend the item set in an additionnal dynamic library through the use of a C api.

Since the 3.2.2 version, TkZinc also offers as a runtime option, the support for openGL rendering, giving access to features such as antialiasing, transparency, color gradients and even a new, openGL oriented, item type : `triangles` . In order to use the openGL features, you need the support of the GLX extension on your X11 server. Of course, performances will be dependant of your graphic card. At the time of writing, NVidia drivers for XFree86 R4.1 are doing a nice job. A laptop with a GeForce GO graphic card works nice for non trivial applications. We also succeeded in using TkZinc with openGL on the Exceed X11 server (running on windows and developped by Hummingbird) with the 3D extension.

As an example of TkZinc capabilities when combined with openGL, we implemented the TkZinc logo as a Perl module (available as a goodie in `LogoZinc.pm`). This logo (see below) was designed with Adobe Illustrator and then programmed in Perl.

Figure 1.1: Zinc Logo written as a Perl/Tk module

Like the canvas TkZinc focuses on the notion of script language. We strongly believe that the script environments are very powerful for rapid prototyping and for developping small to medium scale field specific applications. In these cases developper know-how and time are a scarce resource and the application either has few clients or is short lived. It is important to grant non-specialists an access

to the powerful tools that are available today for HMI building, through a rather simple product.

The TkZinc widget is available for the Tcl/Tk and the Perl/Tk scripting environments. A binding over Tcl/Tk is also provided for Python. It should be easy to do the same for Ruby, a binding for Tk is provided in the standard distribution of Ruby. Other scripting languages may be used as well depending on the availability of a Tk interface.

This document is Tcl/Tk and Perl/Tk oriented but it should be easy for Python or Ruby programmers to adapt. Every time a TkZinc command is described in this document, it is given first in Tcl/Tk idiom and then in Perl/Tk idiom.

This document is also referenced as CENA technical note NT03-532.

1.2 Differences with previous versions

1.2.1 Differences between 3.3 and 3.2.97 release

This release has been mainly focused on producing a stable code base that compile and run on all three supported platforms with as little effort as possible.

The only functional change is the integration of the fieldbbox command into the bbox command.

1.2.2 Differences between 3.2.97 and 3.2.6 release

- TkZinc now works with Perl ptk 8.4 and utf8; However there remains some serious performance hit at launch time, when combinig openGL and ptk8.4, due to utf8 fonts management.
- text and icons can now be scaled and rotated in X (i.e. without openGL)
- translate method accepts an additionnal argument 'absolute'
- scale method accepts additionnal arguments: unit and rotation center
- find and addtag methods can now search inside atomic group
- the bbox method accepts into account the clipping area of a group
- TkZinc option -trackmanagehistory is replaced by -trackvisiblehistorysize and in for track items the attribute -visiblehistorysize has been removed and replaced by -historyvisible. **Beware: Incompatible change**
- Default value of -composescale and -composerotation of texts and icons is now false. This is coherent with the default behavior of these items (being rigids). The impact of this change is greatly minored by the new processing of the -position attribute.
- transformation of items with a -position has been slightly modified. The point described by -position is no longer considered in the coordinate space of the item but in the coordinate space of its parent group. The item is always located in 0,0 of its own coordinate space. This is so to make use of -composescale and -composerotation a lot more useful (and compatible).
- The fieldbbox method has been added to get item filed bounding box.

- Four new methods are now available for managing the transforms: tcompose, tget, tset and skew. A predefined named transformation is also available 'identity' to be used with tsave. A predefined tag 'device' can be used to convert coordinates in or from device coordinates (with transform method).

- TkZinc with Tcl/Tk now works on windows and MacOS X (with X11 and fink).

- compilation on Linux works fine now, and TkZinc for Perl is on the CPAN

- A powerful perl module Tk::Zinc::Graphics has been added to help creating complex curves. French man pages are available. A port in Tcl is also available.

- png images with transparencies can now be displayed (requires openGL rendering)

- bezier items have been suppressed; they can now be easily replaced by curve items.

- curve items support now a higher level of description: they may be composed of line segments, and bezier segments. In the future they may also support other kinds of segments (such as arcs...).

- the coords method accepts a list of arrays as well as flat list of coordinates. When coords returns more than one point it is always a list of arrays. (and no more a flat list of x y x y ...). **Beware this is a small incompatible change in the API**.

- operators of the contour command have been replaced by a flag which indicates if the contour must be taken as counterclockwise, clockwise or unchanged. Contours ids are now predictable. The GPC "not-so-free" library is no more used. It has been replaced by the GLU library. So TkZinc is again fully free software.

- curve item have a new -fillrule attribute.

- the syntax of gradient has been changed, mainly to accomodate with any color specification defined for X. **Beware that old gradient are no more compatible**

- conical gradient type has been added; gradient paramaters has been extended.

- Perl modules ZincText, ZincDebug, ZincTrace and ZincTraceErrors have been renamed Tk::Zinc::Text Tk::Zinc::Debug Tk::Zinc::Trace and Tk::Zinc::Trace.

- TkZinc comes now with a ZincTrace.pm module to trace every TkZinc method call

- the hierarchical view in ZincDebug.pm can now display some choosen attributes in a choosen format.

- 6 new Perl demos in zinc-demos: "testGraphics", "magic Lens", "pathTags", "tiger" and "curve with bezier control points" and "fillrule". Many Perl/Tk demos have been ported to Tcl/Tk.

- pathTags introduced in 3.2.6 have been documented. Label and label format documentation has been enhanced.

1.3 Where can I find TkZinc and documentation ?

`Zinc` is available as source in tar.gz format or as Debian or RedHat/Mandrake packages at

- http://www.tkzinc.org/ or
- http://freshmeat.net/projects/zincisnotcanvas/

For people from CENA, its possible to get the TkZinc source code through a private CVS server. Please contact directly `lecoanet@cena.fr` for more informations.

This documentation is available as part of the TkZinc software. It is also available separately on the web sites. This document is formatted with LaTeX and is distributed as either html pages or a pdf file.

As a complement to this reference manual, small Perl/Tk demos of TkZinc are also available through a small application named `zinc-demos` , highly inspired from the *widget* application included in Tk. The aim of these demos are both to demonstrates the power of TkZinc and to help newcomers start using Zinc with small examples.

1.4 What is this document about ?

This reference manual describes the TkZinc widget interface. It shows how to create and configure a TkZinc widget, and how to use the commands it provides to create and manipulate items. The next chapter `Widget creation and options` describes how to create a new widget and which options and resources are available to configure it. The chapter `Groups, Display List and Transformations` describes the use of groups and coordinates transformations. The chapter `Item ids, tags and indices` describes the item tags along with their main purposes. Also introduced is the concept of part name used by some items (`track` and `waypoint`). Finally, this chapter provides a description of textual indices.

The chapter `Widget commands` describes the commands which apply to a Zinc widget. They are used for creating, modifying or deleting objects, applying transforms ... The chapter `Item types` describes all the items provided by TkZinc along with their attributes. The chapter `Labels, fields and labelformat` describes the use of labels, the possible attributes of fields and finally the labelformat syntax. The chapter `Attributes types` describes the legal form of all item attributes. The chapter `The mapinfo commands` introduces the mapinfo, a simple map description structure, and describes the commands used to create and manipulate mapinfos. Finally the chapter `Other resources provided by the widget` describes some resources provided by or with TkZinc.

1.5 Copyright and Licence

1.6 Authors and credits

Zinc has been developed by Patrick Lecoanet. He also developed two previous version called *Radar Widget* which share some characteristics with this version. The *Radar Widget* was heavily used at CENA for many projects over nearly 10 years. The release 2 is still in use. It was enhanced and then used for actual radar displays in two main French Air Traffic Control Centres 24 hours a day. Dominique Ruiz, Frederic Lepied helped a lot in the developement of these earlier versions.

Zinc benefited greatly from the close interaction and the needs expressed by Jean-Luc Vinot. Jean-Luc has a background of Graphic Designer and is now an HMI developer at CENA. He envisions many, many new ideas for advanced HMI. Many of them would have been difficult to implement if at all possible with similar widgets. Zinc would have been less interesting without his ideas.

Didier Pavet and his team as well as Daniel Etienne and Herve Damiano were the first users and helped a lot either by reporting bugs, problems or solutions. Thanks to all these people and to the CENA for supporting this work.

The core of this documentation has been written by Patrick Lecoanet, the main author of TkZinc. This documentation has been enriched by Christophe Mertz.

1.7 How can I find help with TkZinc

If you are stuck with a feature you don't understand. If you don't know how to do something with TkZinc. If you think you have found a bug or a mismatch between the documentation and the behavior of the widget. Please feel free to contact us. Mail either `lecoanet@cena.fr` or the TkZinc mailing list. To subscribe to the mailing list, please consult the site http://www.tkzinc.org/ .

1.8 How may I contribute to TkZinc development

If you think TkZinc is an interesting tool, they are many ways to help with TkZinc development. First of all, subscribe to the TkZinc mailing list and get in touch with us. To subscribe, please consult the site http://www.tkzinc.org/ .

- The very first way to contribute is to use TkZinc and to report any bug or problem you may experiment. Of course, if you send a script that exhibits the problem or even better a patch,

your problem will have more chance to find a solution.

- The second way to contribute is by commenting on and proposing enhancement to this reference manual. As it has been written by french writers, english readers may really help in making this document easier to use. If you really feel ambitious, you may even try to write a tutorial, but that may be quite an undertaking! Some documentation (currently Tk::Zinc::Graphics) are in French only. Translating it in english would be great.

- The third way to contribute, and may be the funniest way, is to enrich the set of demos (see chapter `Other resources provided by the widget`). Feel free to send us your productions. They may be simple but demonstrative or more complex. It is up to you! They will be integrated in the next release of TkZinc if they are worth it.

- The fourth way to contribute, and may be the most difficult, is to enrich the set of items (see section `C api for adding new items`) in an separate dynamic library. Then send us source code (with appropriate copyright and licence) if you want them to be integrated in a future release of TkZinc.

Chapter 2

Widget creation and options

The TkZinc command creates a new TkZinc widget, the general form are in Tcl and Perl:

```
zinc
```

```
$version = $mainwindow->zinc();
```

```
$Tk::Zinc::VERSION;
```

These expressions can be used to get the version of TkZinc. The string returned by the last expression also details the graphic head available. For example : "zinc-version-3205d X11 GL".

```
zinc pathname ?options?
```

```
$mainwindow->Zinc(?option=>value?, ..., ?option=>value?);
```

`pathname` name the new widget and specifies where in the widget hierarchy it will be located.

You can set the `$ZINC_GLX_INFO` environment variable in order to display some information about the OpenGL instance used by TkZinc *(New since TkZinc v3.2.6i)*.

Any new TkZinc widget comes with a root group item, always identified by the item id 1. This group will contain all other items, either directly or through groups created themselves in the root group. Together the items form a tree rooted at the root group, hence its name. The chapter `Groups, Display List and Transformations` describes the use of groups. The chapter `Item ids, tags and indices` describes the item ids and item tags, used as argument in most commands.

The options are used to configure how the newly created widget will behave. They can be changed later by using the `configure` and `itemconfigure` Tk commands.

Options apply only to the widget itself. They are a Tk supported concept and benefit from the option database an other mechanisms used to externally adapt the application to different environments. Attributes are a similar concept available for items and other TkZinc objects. But they are private to TkZinc and do not benefit from Tk support. They have been named differently to avoid confusion.

Any number of options may be specified on the command line or in the option database to modify the global behavior of the widget. Available options are described below.

Command line switch:	`-backcolor`
Database name:	`backColor`
Database class:	`BackColor`

This is the color that will be used to fill the TkZinc window. It is also used as a default color for some item color attributes. See each color attribute for the actual source of the default color. Its default value is `#c3c3c3`, a light grey.

Command line switch: `-borderwidth`
Database name: `borderWidth`
Database class: `BorderWidth`

Specifies the width of the 3d border that should be displayed around the widget window. This border does overlap the active TkZinc display area. The area requested from the geometry manager (or the window manager if applicable) is the area defined by `-width` and `-height` , the border is not taken into account. This value can be given in any of the forms valid for coordinates (See `TkGet_Pixels`). The default value is `2`.

Command line switch: `-confine`
Database name: `confine`
Database class: `Confine`

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

Command line switch: `-cursor`
Database name: `cursor`
Database class: `Cursor`

Specifies the cursor to use when the pointer is in the TkZinc window. The default value is set to preserve the cursor inherited at widget creation.

Command line switch: `-font`
Database name: `font`
Database class: `Font`

The font specified by this option is used as a default font for item attributes of type font. Its default value is `-adobe-helvetica-bold-r-normal--*-120-*-*-*-*-*-*`.

Command line switch: `-forecolor`
Database name: `foreColor`
Database class: `ForeColor`

The color specified by this option is used as a default color for many item color attributes. See each each color attribute for the actual source of the default color. Its default value is `black`.

Command line switch: `-fullreshape`
Database name: `fullReshape`
Database class: `FullReshape`

If this option is True, the shape applied to the TkZinc window will propagate up the window hierarchy to the toplevel window. The result will be a shaped toplevel. See also the `-reshape` option, it controls whether a shape is applied to the TkZinc window or not. The default is `true`.

Command line switch: `-height`
Database name: `height`
Database class: `Height`

Specifies the height of the TkZinc window. This value can be given in any of the forms valid for coordinates (See `Tk_GetPixels`). The default is `100` pixels.

Command line switch: `-highlightbackground`
Database name: `highlightBackground`
Database class: `HighlightBackground`

Specifies the color to display in the traversal highlight region when the widget does not have the input focus. The default value is `#c3c3c3`.

Command line switch: `-highlightcolor`
Database name: `highlightColor`
Database class: `HighlightColor`

Specifies the color to use for the traversal highlight rectangle that is drawn around the widget when it has the input focus. The default value is `black`.

Command line switch: `-highlightthickness`
Database name: `highlightThickness`
Database class: `HighlightThickness`

Specifies a non-negative value indicating the width of the highlight rectangle drawn around the outside of the widget when it has the input focus. The value may have any of the forms acceptable to `Tk_GetPixels`. If the value is zero, no focus highlight is drawn around the widget. The default value is `2`.

Command line switch: `-insertbackground`
Database name: `insertBackground`
Database class: `InsertBackground`

Specifies the color to use as background in the area covered by the insertion cursor. This color will normally override either the normal background for the widget (or the selection background if the insertion cursor happens to fall in the selection). The default value is `black`.

Command line switch: `-insertofftime`
Database name: `insertOffTime`
Database class: `InsertOffTime`

Specifies a non-negative integer value indicating the number of milliseconds the insertion cursor should remain off in each blink cycle. If this option is zero then the cursor is on all the time. The default value is `300`.

Command line switch: `-insertontime`
Database name: `insertOnTime`
Database class: `InsertOnTime`

Specifies a non-negative integer value indicating the number of milliseconds the insertion cursor should remain on in each blink cycle. The default value is `600`.

Command line switch: `-insertwidth`
Database name: `insertWidth`
Database class: `InsertWidth`

Specifies a value indicating the width of the insertion cursor. The value may have any of the forms acceptable to `Tk_GetPixels`. The default value is `2`.

Command line switch: `-lightangle`
Database name: `lightAngle`
Database class: `LightAngle`

Specifies the lighting angle in degre used when displaying relief. The default value is `120`.

Command line switch: `-mapdistancesymbol`
Database name: `mapDistanceSymbol`
Database class: `MapDistanceSymbol`

This option specifies the symbol to be used as a milestone along map lines. This option can be given any Tk bitmap which can be obtained by `Tk_GetBitmap`. The spacing between markers is 10 nautic miles. The default value is `AtcSymbol19` (see Other resources provided by the widget).

Command line switch: `-maptextfont`
Database name: `mapTextFont`
Database class: `MapTextFont`

Specifies the font used to draw the texts contained in maps. The default is `-adobe-helvetica-bold-r`

Command line switch: `-overlapmanager`
Database name: `overlapManager`
Database class: `OverlapManager`

This option accepts an item id. It specifies if the label overlapping avoidance algorithm should be allowed to do its work on the track labels and which group should be considered to look for tracks. The default is to enable the avoidance algorithm in the root group (id 1). To disable the algorithm this option should be set to `0`.

Command line switch: `-pickaperture`
Database name: `pickAperture`
Database class: `PickAperture`

Specifies the size of an area around the pointer that is used to tell if the pointer is inside an item. This is useful to lessen the precision required when picking graphical elements. This value must be a positive integer. It defaults to `1`.

Command line switch: `-relief`
Database name: `relief`
Database class: `Relief`

Specifies the border relief. This option can be given any legal value for a relief (See `relief` for a description of possible values). The default value is `flat`.

Command line switch: `-render`
Database name: `render`
Database class: `Render`

Specifies whether to use or not the openGL rendering. When True, requires the GLX extension to the X server. Must be defined at widget creation time. This option is readonly and can be used to ask if the widget is drawing with the GLX extension or in plain X (to adapt the application code for example). The default value is `false`.

Command line switch: `-reshape`
Database name: `reshape`
Database class: `Reshape`

Specifies if the clipping shape that can be set in the root group item should clip the root group children or be used to reshape the TkZinc window. This option can be used with the fullreshape option to reshape the toplevel window as well. The default value is `true`.

Command line switch: `-scrollregion`
Database name: `scrollRegion`
Database class: `ScrollRegion`

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 TkZinc.

Command line switch: `-selectbackground`
Database name: `selectBackground`
Database class: `SelectBackground`

Specifies the background color to use for displaying the selection in text items. The default value is `#a0a0a0`.

Command line switch: `-speedvectorlength`
Database name: `speedVectorLength`
Database class: `SpeedVectorLength`

Specifies the duration of track speed vectors. This option is expressed using a time unit that should be chosen by the application (usually minutes) and kept coherent with the unit of the track attribute `-speedvector` (usually nautic mile / minute). The default value is `3`.

Command line switch: `-takefocus`
Database name: `takeFocus`
Database class: `TakeFocus`

(Slightly adapted from the Tk options manpage).

Determines whether the window accepts the focus during keyboard traversal (e.g., Tab and Shift-Tab). Before setting the focus to a window, the traversal scripts consult the value of the takeFocus option. A value of 0 means that the window should be skipped entirely during keyboard traversal. 1 means that the window should receive the input focus as long as it is viewable (it and all of its ancestors are mapped). An empty value for the option means that the traversal scripts make the decision about whether or not to focus on the window: the current algorithm is to skip the window if it is disabled, if it has no key bindings, or if it is not viewable. If the value has any other form, then the traversal scripts take the value, append the name of the window to it (with a separator space), and evaluate the resulting string as a callback. The script must return 0, 1, or an empty string: a 0 or 1 value specifies whether the window will receive the input focus, and an empty string results in the default decision described above. *Note: this interpretation of the option is defined entirely by the callbacks (part of the keyboard traversal scripts) that implement traversal; the widget implementations ignore the option entirely, so you can change its meaning if you redefine the keyboard traversal scripts.* The default value is empty.

Command line switch: `-tile`
Database name: `tile`
Database class: `Tile`

Specifies an image name to be used as a tile for painting the TkZinc window background. The default value is `""` (the empty string).

Command line switch: `-trackmanagedhistorysize`
Database name: `trackManagedHistorySize`
Database class: `TrackManagedHistorySize`

This option accepts only positive integers. It specifies the number of positions collected in the history list by the track items. When this many positions have been collected, the oldest is dropped to make room for a new one on a first-in first-out basis. See also the `-trackvisibehistorysize` option and the `-historyvisible` track attribute. The default value is `6`.

Command line switch: `-trackvisiblehistorysize`
Database name: `trackVisibleHistorySize`
Database class: `TrackVisibleHistorySize`

This option accepts only positive integers. It specifies the number of past positions to display for tracks. It is a widget wide control. Users of previous releases used the `-visiblehistorysize` track attribute for the same effect. The number of past positions displayed can not exceed the accumulated positions controlled by the option `-trackmanagedhistorysize` . The track `-historyvisible` attribute controls whether a track should display its history. The default value is `6`.

Command line switch: `-tracksymbol`
Database name: `trackSymbol`
Database class: `TrackSymbol`

Specifies the symbol displayed at the current position of a track. This option accepts a `bitmap` . The default value is `AtcSymbol15`.

Command line switch: `-xscrollcommand`
Database name: `xScrollCommand`
Database class: `ScrollCommand`

Specifies a callback used to communicate with horizontal scrollbars. When the view in the widget's window changes (or whenever anything else occurs that could change the display in a scrollbar, such as a change in the total size of the widget's contents), the widget will make a callback passing two numeric arguments in addition to any specified in the callback. Each of the numbers is a fraction between 0 and 1, which indicates a position in the document. 0 indicates the beginning of the document, 1 indicates the end, .333 indicates a position one third the way through the document, and so on. The first fraction indicates the first information in the document that is visible in the window, and the second fraction indicates the information just after the last portion that is visible. Typically the xScrollCommand option consists of the scrollbar widget object and the method "set" i.e. [set =¿ $sb]: this will cause the scrollbar to be updated whenever the view in the window changes. If this option is not specified, then no command will be executed.

Command line switch: `-xscrollincrement`
Database name: `scrollincrement`
Database class: `xScrollIncrement`

Specifies an increment for horizontal scrolling. If the value of this option is greater than zero, the horizontal view in the window will be constrained so that the TkZinc 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 switch: `-yscrollcommand`
Database name: `yScrollCommand`
Database class: `ScrollCommand`

Specifies a callback used to communicate with vertical scrollbars. This option is treated in the same way as the xScrollCommand option, except that it is used for vertical scrollbars and is provided by widgets that support vertical scrolling. See the description of xScrollCommand for details on how this option is used.

Command line switch: `-yscrollincrement`
Database name: `scrollincrement`
Database class: `yScrollIncrement`

Specifies an increment for vertical scrolling. If the value of this option is greater than zero, the vertical view in the window will be constrained so that the TkZinc y coordinate at the left 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.

Command line switch: `-width`
Database name: `width`
Database class: `Width`

Specifies the width of the TkZinc window. This value can be given in any of the forms valid for coordinates (See `Tk_GetPixels`). The default is `100` pixels.

Chapter 3

Groups, Display lists, Clipping and Transformations

Groups are very powerful items. They have no graphics of their own but are used to bundle items together so that they can be manipulated easily as a whole. Groups can modify in several way how items are displayed and how they react to events. They have many uses in TkZinc and we will describe them in this chapter. The main usages are:

- to bundle items together so they can be cloned, destroyed, hidden, moved and more as a whole,
- to bundle several items together so that they form a new single item composed of several simpler one. This is done by modifying the way events are associated with items (see `-atomic`),
- to interpose a new coordinate system in a hierarchy of items. This can be very useful to manage panning, zooming and other kind of viewing transformation. See below for an explanation of the transformation system
- to compose some specific attributes such as transparency, sensitivity, visibility, ... with those of their children items,
- to apply a clipping to their children items,
- to manage display ordering between items and to do the display lists housekeeping.

3.1 The root group and the item tree

An item, be it simple like a rectangle or more complex like a group, is always created relative to a group which is known as its parent, the group's items are its children. The items form a tree whose nodes are the group items. The top-most node is known as the root group, of id 1, which is automatically created with TkZinc. By convention, the root group is its own parent. It is not possible to change the parent of the root group and it is not possible to delete it. However, it is possible to change the group of all other items after creation, and thus modify the item tree at any time. This is the use of the `chggroup` command.

3.2 Attributes composed with children

The following attributes are composed down the item tree to form the resulting attribute value in the leaf items:

- `-sensitive`: the sensitivity (to keyboards or mouse event) of an item is the result of and-ing together the `-sensitive` attributes found when descending from the root group to a specific leaf item.
- `-visible`: the visibility of an item is the result of and-ing together the `-visible` attributes found when descending from the root group to a specific leaf item.
- `-alpha`: the transparency of an item is the result of combining the `-alpha` attributes of the groups found when descending from the root group to a specific item with the alpha channel found in a given color of this item. The transparency is a percentage between 0 and 100, two transparencies are combined by multiplying both and then dividing by 100. The transparency can be used only if the environment support openGL and if the widget was created with the `-render` option set to True.

3.3 Atomic groups

It may seem at first that there is a contradiction in this title, but there is not. It is possible to built complex objects from simple items simply by assembling those items together in a group (using other intervening groups if the need arise). Once this is done, it would be convenient if the whole acted as a single item, the top assembling group. It is already so for many commands that act on a group, it is possible to move, resize, rotate, restack, clone, hide, change the transparency, delete the group as a whole without knowing anything of its children. But when it comes to event dispatching, the group is completly transparent so far. So the event dispatch mecanism will try to locate the smallest most visible item containing the pointer and will trigger the associated bindings. Not exactly what we meant. So groups have a feature, the `-atomic` attribute, that is used to seal a group so that events cannot propagate past it downward. If an item part of an atomic group is under the pointer, TkZinc will try to trigger bindings associated with the atomic group not with the item under the pointer. This improves greatly the metaphor of an indivisible item.

It must also be noted that commands such as `find 'enclosed'/'overlapping'` or `addtag 'enclosed'/'` **act differently on an atomic group**. Such search command will not traverse an atomic group. So if a part of the atomic group is enclosed or overlapping, the search command will return the atomic group and not its part.

A small program, `Atomic groups` is available as part of `zinc-demos` to demonstrate the atomic groups behaviour.

3.4 Display order and display lists

The items are displayed in a specific order which determines how they stack. This order is also important for associating events with items. The items are arranged in a display list for each group. The display list imposes a total ordering among its items. The group display lists are connected in a tree identical to the group tree and form a hierarchical display list. The items are drawn by traversing the display list from the least visible item to the most visible one. Each time a group is

encoutered the traversal proceed with this group display list before resuming the upper display list traversal. The search to find the item that should receive an event is done in the opposite direction. In this way, items are drawn according to their relative stacking order and events are dispatched to the top-most item at a given location.

It is important to note as a consequence of this structuring, that items of a group are stacked between the items that are under the group and the items that are on top of the group. Thus, items of two groups cannot be intertwinned, they stack exactly as their groups stack, that is items of the underneath group a drawn then the items of the other group are drawn on top.

The item ordering imposed by the display lists can be ajusted in three ways. The two first are local to a group's display list. The third can be used to rearrange between groups.

- The attribute `-priority` can be used to give an absolute stacking position to an item amongst the items of its group. When a new item is added to a group with a priority matching the priority of existing items of the group, the new item is placed on top of the those items. Last comer is most visible. The same rule is followed when changing the priority of an item and when moving an item to a new group. `map` and `reticle` default priority is 0. `window` item attribute is meaningless. All other items default priority is 1.

- The commands `raise` and `lower` adjust the relative order of an item in its group. These commands can be used to bring an item in front or to the back of its group. It is also possible to place an item before or after a given item of its group. These commands act in such a way as to preserve the absolute relationship set by the `-priority` attribute. To do so they may adjust the priority of the moved item to match the priority of the item just below (raise) or just above (lower). If the priority of the moved item is not in conflict with its new neighborhood, it is not affected.

- It is also possible to move the item to another group. This has an effect on the item stacking as it will be forced to the stacking location of its new group.

3.5 Event sensitivity

An item will catch an event if all the following conditions are satisfied:

- the item `-sensitive` must be set to true (this is the default).

- the item must be under the pointer location.

- the item must be on top of the display list (at the pointer location). Beware that an other item with its `-visible` set to false DOES catch event before any underneath items.

- the item must not be clipped (at the pointer location)

- the item must not belong to an atomic group, since an atomic group catchs the event instead of the item.

An item satisfying all the above conditions can have its `-visible` set to false, or can be fully transparent (when using openGL). It will still catch the events.

3.6 Transformations

In TkZinc each item is geometrically defined in its own coordinate space. So each time a new item is created, a new coordinate system is attached to it. This coordinate system must be related to the coordinate systems of the other items to place the items with respect to each other. This relationship is defined by an affine transformation associated with the item. This transformation establishes the relationship between an item and its group. The items being arranged in a tree by their groups, its possible via the transformations to place all the items in an absolute coordinate system known as the window space.

Just after item creation, the item transformation is set to identity, i.e the item coordinate system maps exactly on the system of its group. The commands `translate` , `scale` , `rotate` , `skew` can be used to modify this relationship to the effect of translating, enlarging, shrinking, rotating or skewing the item. It must be emphasized that those commands act on the relation between two coordinate spaces, *not* on the item geometry itself. If the goal is to change the item (except for groups, see next paragraph) geometry, the command `coords` may be more appropriate (but see below the command `tapply`).

As it should be clear, groups are like any other items, they are defined in their own coordinate space and are assembled with their parents by transformations. This is a very powerful tool to manage the geometry of clusters of items. One must not refrain from using groups only to assign them a transformation task such as panning a whole set of items or scaling a set while another is kept in place in another group. For the developper convenience, the `coords` method on a group change its transformation. It defines the absolute translation applied to the group.

Another very interesting use of a group as a transformation tool is to manage a window coordinate space where the origin is not in the top left corner and where the Y axis goes from bottom to top. It is quite simple to write a function that is triggerred on the window resize event whose only goal is to compute a new transformation for the group. Other parts of the application and the other items are not aware of this happening. A good factorization example.

In fact, transformation are so useful that a whole set of functions are available to help use them in full. Apart from the already mentioned `translate` , `scale` , `rotate` , and `skew` commands, it is also possible to restore a transformation to its initial state, identity, with the `treset` command. It is also possible to compose a transformation with another name transform with the `tcompose` command.

An item transformation can be saved under a name, in fact creating a named transformation which can be manipulated just as an item transformation (i.e using translate, scale, rotate, treset). Once a transformation has been named it can be used to set the transformation of any item using with the command `trestore` . And it can be disposed of with the command `tdelete` .

An item can be physically modified by applying its own transformation to itself. This is the goal of the `tapply` command. It applies the item transformation to its own coordinates an then reset the item transformation. Visually nothing has changed but in fact the item is irrevocably modified. Be aware that if it is quite easy to undo a change in a transformation by using treset or by saving and then restoring a transformation, it is not so easy to revert a physical modification on an item. The exact order of the operations must be recorded and even then there is no shield against round off errors that will probably occur. This command may be used together with the translate, scale and rotate commands if someone really want, even after reading this paragraph, to implement the canvas move, scale and even rotate commands.

A predefined named transformation exists. Its name is `identity` and refers to the identity transform.

it can not be modified by the user.

When dealing with mouse events and other sources of window coordinates, it is often useful to map the window coordinates to an appropriate coordinate space. The command `transform` is just what is needed to do so. It is powerful enough to be able to convert coordinates from any coordinate space to any other. A special provision has been made to facilitate conversion from window space to another space. The opposite is not impossible but rely on a small trick: the root group transformation must be left as identity (the default at creation time). In this way, it is possible to use the root group space, which is then the same as the window space, as the target space of the `transform` command.

If you need to manage many different transformations independently, it is a good practice to apply these transformations to different groups. For example, a group can be used for translation and an other group (father or son) for scaling.

When a rotation or a scale appear in a transformation, all items do not behave exactly in the same manner. For example text items do not scale or rotate. Only their position moves according to the rotation or the scaling factor. Here is how items react to the scale and rotation factors of the transformation.

- `group` They have no graphical shape by themselves;
- `track` The position, past positions, speedvector are fully transformed. The circular marker is transformed by the X scale factor, it always remains circular. The label position is computed relative to the new position and speedvector direction but is otherwise rigid and its distance (in pixels) to the position is unchanged.
- `waypoint` The position is fully transformed. The label position is computed relative to the new position and new rotation but is otherwise rigid and its distance (in pixels) to the position is unchanged.
- `tabular` , `text` Only the position (relative to the anchor) is affected.
- `icon` With openGL, icon items can be rotated and zoomed.
- `reticle` Only the center and the spacing between circles are affected.
- `map` lines and arcs are fully transformed. For texts and symbols only the position is affected.
- `rectangle` , `arc` , and `curve` are fully transformed.
- `window` Only the position (relative to the anchor) is affected.

However, every item has a couple of attributes `-composescale` and `-composerotation` that can be used to control how the scale and rotation factors are inherited from the parents' transformations. These attributes default to `true` (i.e. rotation and scale from parents are meaningful, except for `icon` where these attributes defaulted to `false`). When one of these attributes is set to false the corresponding factor is reset from the inherited transformation. Scale factors are reset to 1.0 and rotation is reset to 0. Be careful that this applies to the inherited transformation, *not* to the item transformation itself which is composed *after* taking into account the composition attributes.

As you can see, the transformation process is quite powerful but complex. A small program, `Tranformation testbed` is available as part of `zinc-demos` to demonstrate the transformation capabilities of TkZinc. This is also a great resource to understand how it works and to tame its complexity. It is possible to use this program to test one's idea on a given transformation problem before coding it as part of a complex application.

3.7 Clipping and groups

Groups can set a clip boundary before drawing their children. Thought of this feature as if a group can be made to act as a window on its children. Except that the window can have any shape you like to give it. Each group has a `-clip` attribute which can be set to an item of the group. This item, known as the clipper of the group, defines the shape of the clipping. All item types except `group` , `track` , `waypoint` , `reticle` and `map` can be used as clippers but the clipper must be a direct child of the clipped group. The clipper defines the shape of the clipping but is also drawn as a regular group item. It is typical to either mask explicitly the clipper by turning off its `-visible` attribute or to fill and lower it so it can act as the background. Of course, other creative uses can be found but be warned that the clipper outline will never be easthetically drawn due to round off or quantization errors, it is better to turn off borders or outlines in this case.

It is also possible to clip the root group (only on X11 system. Does not work on windows systems). Depending on the value of TkZinc options `-reshape` and `-fullreshape` , the clipping form can be used either to clip all items in the TkZinc widget, or reshape the TkZinc widget, or to propagate the TkZinc widget shape to the parent windows. In the latter case, this allows to build non-rectangulaire applications. This requires both the SHAPE X11 extension and a compliant Window Manager (fvwm is known to support non rectangulaire top windows). The clipping form should have a bounding box with the same ratio as the topwindow or some normalisation will occur. Example:

```
use Tk::Zinc;


my $mw = MainWindow->new();
my $zinc = $mw->Zinc(-reshape => 1, -fullreshape => 1)->pack;


# creating a triangulaire curve
my $triangle= $zinc->add('curve',1, [ [0,0], [100,0], [50,100] ], -closed => 1);
# using the triangulaire curve to reshape both TkZinc and Mainwindow widgets
$zinc->itemconfigure(1, -clip => $triangle);


$zinc->add('arc',1, [ [0,0], [100,100] ], -filled => 1, -fillcolor => 'darkblue'); ...
&Tk::MainLoop;
```

Chapter 4

Item ids, tags and indices

4.1 Item ids

Each item is associated with a unique numerical id which is returned by the `add` or `clone` commands. All commands on items accept those ids as (often first) parameter in order to uniquely identify on which item they should operate. When an id has been allocated to an item, it is never collected even after the item has been destroyed, in a TkZinc session two items cannot have the same id. This property can be quite useful when used in conjonction with tags, which are described below.

4.2 Tags

Apart from an id, an item can be associated with as many symbolic names as it may be needed by an application. Those names are called tags and can be any string which does not form a valid id (an integer). However the following characters may not be used to form a tag: `. * ! ( ) & | :`. Tags exists, and may be used in commands, even if no item are associated with them. In contrast an item id doesn't exist if its item is no longer around and thus it is illegal to use it. Tags can be used to group items to do some action, or to mark an item that has a special function. Many other tasks can be solved with tags once one gets used to them.

Two special tags are implicitly managed by TkZinc. The tag `all` is associated with all items in TkZinc. The tag `current` is always associated with the topmost item that lies under the mouse pointer. If no such item exists, no item has this tag.

In commands, tags can be used almost anywhere an item id would be legal. In the command descriptions, the expression `tagOrId` means that it is legal to provide either a tag or an item id. This means that virtually all actions can be either performed on a specific item by using its id or on a whole set of items by using a tag. In order for this collective behavior to be useful, if a command or an attribute does not apply to an item named by the tag, it is simply ignored, no error will be reported (This may yet not be the case with all commands, please report infringements).

Everywhere a `tagOrId` can be specified as a target for some action, it is possible to give a logical expression of tags and ids. The available boolean operators include logical and `&&`, logical or `||`, logical xor `^`, logical not `!` and subexpression grouping `()`. Here is an example of a `bbox` command called on a set of items defined by a logical expression. Note that tags and ids can be mixed. For example:

```
($xo, $yo, $xc, $yc) = $zinc->bbox("(red && black)||(pink && !$thisitem)");
```

Many methods only operate on a single item at a time; if `tagOrId` is specified in a way that names multiple items, then the normal behavior for these methods is to use the first of these items in the display list (most visible) that is suitable for the method. Exceptions are noted in the method descriptions below.

Tags can be associated with items by giving a tag list to the `-tags` attribute or by using the more powerful `addtag` command. A tag can be removed by the `dtag` command, by setting the `-tags` attribute to the empty list, all tags are remove from an item at once (except the implicit ones). Tags can be read with the `gettags` or by querying the `-tags` attribute. The items named by a tag are returned in a list by the `find` command which as exactly the same capabilities as `addtag` .

4.3 PathTags

A special form of tag called a pathTag can be used as a tagOrId argument in all commands except `bind` . This special tag describes an item or a group of items in the absolute item hierarchy. The pathTag consists in a path down the group hierarchy followed by an (optionnal) effective tag, in the usual sense.

The path is an ordered list of tags set up on groups that drives the search from the root group down the group hierarchy. The path starts with either a dot or a star, and the tags in the path are separated by dots or stars. The dot means that the next tag selects a group item that is a direct child of the current group, starting with the root group. The star selects a group item that is a possibly indirect child of the current group, the candidate is found in display list order. The first tag in the path, the one just after the first dot or star, can be a group id; This is useful in order to limit the search in a specific sub-hierarchy.

The last tag of a pathTag, the one not followed by a dot or a star, is the effective tag searched for. It can be omitted, in this case the search proceed with the tag all. The dot or star just before the effective tag, even if the tag is implied, controls how the tag is searched. If a dot is present, the search is limited to the current group level. If a star is present, the search proceed from the current group level down the whole group subtree.

A demo called “Using pathTags” in zinc-demos may help you better understand pathTags. Here are some commonly used pathTags idioms:

`.group1Tag.group2Tag.aTag`
: Selects all **direct** children with the tag `aTag` in the group obtained by following the path `.group1Tag.group2Tag` from the root group. The search proceed from the root group to the first direct child group in display list order with tag `group1Tag`. Then it searches for the first direct child group with tag `group2Tag`. Finally in this group, the search ends by finding all direct children items (including groups) with tag `aTag`. If a tag is not found the whole search is aborted and no item is selected.

`.group1Tag.group2Tag*aTag`
: Selects all children, **direct or indirect**, with the tag `aTag` in the group obtained by following the path `.group1Tag.group2Tag` from the root group.

`.group1Tag*group2Tag.aTag`
: Selects all **direct** children with the tag `aTag` in the group obtained by following the path

`.group1Tag*group2Tag` from the root group. The search proceed from the root group to the first direct child group in display list order with tag `group1Tag`. Then it searches in display list order down the hierearchy for the first group with tag `group2Tag` . Finally in this group, the search ends by finding all direct children items (including groups) with tag `aTag`. If a tag is not found the whole search is aborted and no item is selected.

`.group1Tag*group2Tag*aTag`
: Selects all items with the tag `aTag` in the hierarchy of the group obtained by following the path `.group1Tag*group2Tag` from the root group. The search proceed from the root group to the first direct child group in display list order with tag `group1Tag`. Then it searches in display list order down the hierearchy for the first group with tag `group2Tag` . Finally in this group, the search ends by finding all direct children items (including groups) with tag `aTag`. If a tag is not found the whole search is aborted and no item is selected.

`.group1Tag.group2Tag.`
: Selects all **direct** children of the group obtained by following the path `.group1Tag.group2Tag` from the root group. If a tag is not found the whole search is aborted and no item is selected.

`.group1Tag.group2Tag*`
: Selects all items in the hierarchy of the group obtained by following the path `.group1Tag.group2Tag` from the root group. If a tag is not found the whole search is aborted and no item is selected.

`.groupId.aTag`
: Selects all **direct** children with tag `aTag` of the group with id `groupId`. If `groupId` is not found or is not a group id, the search is aborted and no item is selected. This form together with the next is specially useful with cloned items hierarchies where only the topmost group item is known after cloning. Using pathTags it is now possible to make use of designs using components with named sub-components. It is possible to clone a component and afterward to change the behavior of named sub-components with pathTags of the form `.componentId.subComponentName` or perhaps better .componentId*subComponentName . Some care is needed in order to avoid sub-component name clash. Remember that the search for tags proceed in **display list order**, not in hierarchy order. In other more technical words the search walks the item tree **depth first** not breadth first.

`.groupId*aTag`
: Selects all items with tag `aTag` in the hierarchy of the group with id `groupId`. If `groupId` is not found or is not a group id, the search is aborted and no item is selected.

`.groupId.`
: Selects all **direct** children of the group with id `groupId`. It is the only way to get direct children of a group.

`.groupId*`
: Selects all items in the hierarchy of the group with id `groupId`.

`.`
: Selects all **direct** children of the root group.

`*`
: Selects all items in the hierarchy (not counting the root group itself).

`.aTag`
: Selects all **direct** children of the root group with the tag `aTag`.

`*aTag`
: Selects all items in the whole hierarchy (starting a the root group) with the tag `aTag`. It is the same as using the simple tag `aTag`.

4.4 Tags and bindings

Tags are also very useful to associate scripts with events. The `bind` command is used to specify a script to be invoqued when an event sequence is associated with a tag.

The event dispatch mecanism in TkZinc collects which tags are related to a given event and then use the bindings established by `bind` to activate the related scripts. Event dispatching operates on three event sources: mouse events, keyboard events and internally generated enter/leave events. Mouse events are dispatched to the item under the mouse pointer, if any; keyboard events are dispatched to the focus item, if any; leave events are dispatched to the item previously under the pointer, enter events to the item newly under the pointer. Tags are collected from the item found.

Special tags are managed for items with fields or parts (e.g. a `track` has both, a `tabular` has only `fields` and a `rectangle` has none). They are built from a tag or an id followed by a `:` followed by a (zero based) field index or by the name of a part. Those tags can only be used in event bindings.

Here is the complete list of tags, either real or implicit, that are tried to find bindings. They are listed in the order they are processed.

1. the implicit tag `all` (associated with all items),
2. the other tags (in some not reliable order),
3. the item id,
4. the implicit tags build from the tags and the current part name or field index, if any,,
5. the implicit tag build from the item id and the current part name or field index, if any.

An exception is made for the `Leave` event when dispatched to an item with parts or fields. This is needed to process the exit of a field/part before the exit of the corresponding item. In this case the order is shown by the following list.

1. the implicit tags build from the tags and the current part name or field index,
2. the implicit tag build from the item id and the current part name or field index,
3. the implicit tag `all`,
4. the other tags,
5. the item id.

Here are examples of possible bindings.

1. `$zinc->bind($id, '<1>', \&acallback);`
   will call `acallback` if mouse button 1 is clicked anywhere over item `$id`;

2. `$zinc->bind('selected', '<1>', \&acallback);`
   the click must be anywhere over any item associated with the `selected` tag;

3. `$zinc->bind('foo:0', '<1>', \&acallback);`
   the click must occurs over field 0 of an item with tag `foo`;

4. `$zinc->bind("$id:3", '<1>', \&acallback);`
   the click must be over field 3 of item `$id`, and the field must exists in the item;

5. `$zinc->bind("$id:speedvector", '<1>', \&acallback);`
   the click must be over a part named `speedvector` (item track) in item `$id`, the part must exists in the item.

4.5 Text indices

Indices are used to specify a character position in textual items such as the `text` item. Indices are accepted as parameters by commands managing text: `cursor` , `index` , `insert` , `dchars` and `select` .

`number` This should be an integer giving the character position within the text of the item. The indices are zero based. A number less than zero is treated as zero and a number greater than the text length is rounded to the text length. A number equal to the text length refers to the position past the last character in the text.

`end` Refers to the position past the last character in the text. This is the same as specifying a number equal to the text length.

`insert` Refers to the character just before the insertion cursor in the item.

`sel.first` Refers to the first character of the selection in the item. If the selection is not in the item, this form returns an error.

`sel.last` Refers to the last character of the selection in the item. If the selection is not in the item, this form returns an error.

`@x,y` Refers to the character at the point given by `x` and `y`, `x` and `y` are interpreted as window coordinates. If the point lies outside of the area corvered by the item, they refer to the first or last character in the line that is closest to the point.

`bol` refers to the beginning of line

`eol` refers to the end of line

`bow` refers to the beginning of word

`eow` refers to the end of word

`up` refers to previous line

`down` refers to next line

Chapter 5

Widget commands

In this chapter, we first list all commands by categories, then we details each command, by alphabetical order.

5.1 Categories of commands

- Items creation and deletion : `add becomes clone remove`
- Accessing options and attributes : `chggroup configure coords gettags group itemcget itemconfigure`
- Accessing field attributes of track, waypoint and tabular : `currentpart hasfields itemcget itemconfigure numparts`
- Transformations : `coords rotate scale skew tapply tcompose tdelete tget transform translate treset tsave tset`
- Groups, Display list and Priorities : `chggroup find` ('ancestors') `group lower raise`
- Tag management : `addtag dtag find gettags hastag`
- Text management (text item and text fields : `cursor dchars focus index insert select`
- Bindings : `bind focus`
- Coordinates : `anchorxy bbox coords contour fit hasanchor smooth transform vertexat`
- Named resources : `gname gdelete tsave`
- Micellanous : `monitor postscript type`

5.2 Commands by alphabetical order

The available commands are listed in alphabetical order.

The command set for the TkZinc widget is much inspired by the Canvas command set. Someone comfortable with the Canvas should not have much trouble using the TkZinc's commands. Eventually,

the command set will be a superset of the Canvas command set. Anyway some commands depart radically from the equivalent in the Canvas. For example, the user should be aware that `scale` and `translate` do not behave in the same way at all; `find` and `addtag` have been extended to support groups, etc. So use the available knowledge with some care.

In the Perl/Tk version, the commands returning a list, return a Perl array (not a reference) and all list parameters are given as array references.

`pathname` **add** `?type group? ?initargs? ?option value? ... ?option value?`

```
@types = $zinc->add();
$id = $zinc->add(type, group);
$id = $zinc->add(type, group, initargs);
$id = $zinc->add(type, group, initargs, option=>value, ..., ?option=>value?);
```

This command is used to create new items in a TkZinc widget. It can be called with no parameters to return the list of all item types currently known by TkZinc. It can also be called with a valid item type as first parameter and a group item as second parameter to create a new item of this type in the given group.

After these first two parameters come some item type specific arguments. Here is detailed description of these arguments by type:

`arc`
: The arc type expects a list of four floating point numbers `xo yo xc yc`, giving the coordinates of the origin and the corner of the enclosing rectangle. The origin should be the top left vertex of the enclosing rectangle and the corner the bottom right vertex of the rectangle.

`curve`
: The curve type expects either a flat list or a list of lists. In the first case, the flat list must be a list of floating point numbers `x0 y0 x1 y1 ... xn yn`, giving the coordinates of the curve vertices. The number of values should be even (or the last value will be discarded) but the list can be empty to build an empty invisible curve. In the second case, the list must contain lists of 2 or 3 elements: xi, yi and and an optionnal point type. Currently, the only available point type is 'c' for a cubic bezier control point. For example in Perl/Tk, the following list is an example of 2 beziers segments with a straight segment in-between:

  `( [x0, y0], [x1, y1, 'c'], [x2, y2, 'c'], [x3, y3], [x4, y4, 'c'], [x5, y5] )`

  As there is only on control point, `[x4, y4, 'c']` , for the second cubic bezier, the omitted second control point will be defaulted to the same point. A curve can be defined later with the `contour` or `coords` commands. As a side effect of the curve behavior, a one vertex curve is essentially the same as an empty curve, it only waste some more memory.

`rectangle`
: The rectangle type expects a list of four floating point numbers `xo yo xc yc`, giving the coordinates of the origin and the corner of the rectangle.

`triangles`
: The triangles type expects a list of at least 6 floating point numbers `x0 y0 x1 y1 ... xn yn`, giving the coordinates of the vertices of the triangles composing this item. The triangles layout is defined by the attribute `-fan` . If `-fan` is true, the

triangles are arranged in a fan with the first point being the center and the other points defining the perimeter. If -fan is false, the triangles are arranged in a strip.

tabular , track , waypoint
: These types expect the number of fields they will manage in the label or tabular form. This number must be greater or equal to zero.

group , icon , map , reticle , text , window
: These types do not expect type specific arguments.

Following the creation args the command accept any number of attributes - values pairs to configure the newly created item. All the configurable item type attributes are valid in this context. The command returns the item id.

`pathname addtag tag searchSpec ?arg arg ...?`

`$zinc->addtag(tag, searchSpec);`

This command add the given tag to all items matching the search specification. If the tag is already present on some item, nothing is done for that item. The command has no effect if no item satisfy the given criteria. The command returns an empty string.

Many commands take a group as a starting point for the search. If no group is given, the root group is assumed. In any cases, the starting group will not be reported in the search result. This means that the root group will never be reported in a search and that tags cannot be attached to it except in specifying its id.

The search specification and the associated arguments can take the following forms:

- `pathname addtag tag above tagOrId`
  `$zinc->addtag(tag, 'above', tagOrId);`
  Selects the item just above the one given by `tagOrId`. If `tagOrId` names more than one item, the topmost of these items in the display list will be used. If `tagOrId` does not refer to any item then nothing happen.
- `pathname addtag tag all ?inGroup? ?recursive?`
  `$zinc->addtag(tag, 'all', ?inGroup?, ?recursive?);`
  This form is no more allowed since version 3.2.6 of TkZinc. Please use a form '"withtag", "all"' as documented below.
- `pathname addtag tag atpriority priority ?tagOrId?`
  `$zinc->addtag(tag, 'atpriority', priority, ?tagOrId?);`
  Selects all the items at the given priority. The tagOrId optional parameter can be specified to restrict the search. It specifies a group to start with instead of the root group and it can be used to control if the search should be recursive or not (see PathTags for more on this subject).
- `pathname addtag tag ancestors tagOrId ?tagOrId2?`
  `$zinc->addtag(tag, 'ancestors', tagOrId, ?tagOrId2?);`
  Selects all ancestors (i.e. parent groups) of tagOrId. If `tagOrId` names more than one item, the first, (or the topmost) of these items in the display list will be used. If ?tagOrId2? is specified, only parent groups with this tag are selected.
- `pathname addtag tag below tagOrId`
  `$zinc->addtag(tag, 'below', tagOrId);`
  Selects the item just below the one given by `tagOrId`. If `tagOrId` names more than one item, the lowest of these items in the display list will be used. If `tagOrId` does not refer to any item then nothing happen.

- `pathname` **addtag** `tag closest x y ?halo? ?startItem?, ?recursive?`
  `$zinc->`**addtag**`(tag, 'closest', x, y, ?halo?, ?startItem?, ?recursive?);`
  Selects the item closest to the point `x - y`. Any item overlapping the point is considered as closest and the topmost is selected. If `halo` is given, it defines the size of the point `x - y`. `halo` must be a non negative integer. If `start` is specified, it must be an item tag or id. If it names a group and this group is not atomic, the search starts with the first item of this group. If it names a non group item or an atomic group (for a tag, the lowest item with the tag is considered), the search starts with the item below `start` instead of the first item in the display order. If `startItem` does not name a valid item, it is ignored. If `recursive` is false the search is limited to the starting group. If set to true, not specified or "override", the search proceed from the starting group down the hierarchy rooted at this group. "override" forces the search to explore atomic groups and report the most specific item instead of the group itself.
- `pathname` **addtag** `tag enclosed xo yo xc yc ?inGroup? ?recursive?`
  `$zinc->`**addtag**`(tag, 'enclosed', xo, yo, xc, yc, ?inGroup?, ?recursive?);`
  Selects all the items completely enclosed in the rectangle whose origin is at `xo - yo` and corner at `xc - yc`. `xc` must be no greater than `xo` and `yo` must be no greater than `yc`. All coordinates must be integers. `inGroup` specifies a group to start with instead of the root group. If `recursive` is false the search is limited to the starting group. If set to true, not specified or "override", the search proceed from the starting group down the hierarchy rooted at this group. "override" forces the search to explore atomic groups and report the most specific item instead of the group itself.
  It may be necessary to update the TkZinc internal geometry with a call to `update` if the current state is not stable (i.e before calling the main loop or in a callback after modifying the transform or doing something else affecting the geometry of items).
- `pathname` **addtag** `tag overlapping xo yo xc yc ?inGroup? ?recursive?`
  `$zinc->`**addtag**`(tag, 'overlapping', xo, yo, xc, yc, ?inGroup?, ?recursive?)`
  Selects all the items that overlaps or are enclosed in the rectangle whose origin is at `xo - yo` and corner at `xc - yc`. `xc` must be no greater than `xo` and `yo` must be no greater than `yc`. All coordinates must be integers. `inGroup` specifies a group to start with instead of the root group. If `recursive` is false, the search is limited to the starting group. If set to true, not specified or "override", the search proceed from the starting group down the hierarchy rooted at this group. "override" forces the search to explore atomic groups and report the most specific item instead of the group itself.
  See also the `enclosed` variant above for a discussion on updating the geometry.
- `pathname` **addtag** `tag withtag tagOrId`
  `$zinc->`**addtag**`(tag, 'withtag', tagOrId);`
  Selects all the items given by `tagOrId`.
- `pathname` **addtag** `tag withtype type ?tagOrId?`
  `$zinc->`**addtag**`(tag, 'withtype', type, ?tagOrId?);`
  Selects all the items of type `type`. The tagOrId optional parameter can be specified to restrict the search. It specifies a group to start with instead of the root group and it can be used to control if the search should be recursive or not (see PathTags for more on this subject).

`pathname` **anchorxy** `tagOrId anchor`

```
($x, $y) = $zinc->anchorxy(tagOrId, anchor);
```

Returns the *window coordinates* of an item anchor. If no item is named by `tagOrId` or if the item doesn't support anchors, an error is raised. If more than one item match `tagOrId`, the topmost in display list order is used.

```
# for example to get the lower right corner of a text item
($x ,$y) = $zinc->anchorxy('myTextTag', 'se');
```

```
pathname bbox ?-field fieldNo?  ?-label?  tagOrId ?tagOrId ...?
```

```
($xo, $yo, $xc, $yc) =
$zinc->bbox(?-field fieldNo?  ?-label?  tagOrId, ?tagOrId ...?);
```

Returns a list of 4 numbers describing the *window coordinates* of the origin and corner of a rectangle bounding all the items named by the `tagOrId` arguments. If no items are named by `tagOrId` or if the matching items have an empty bounding box, an empty string is returned. The `-field` and `-label` options can be used to query the bounding box of an item's field or an item's label. These two options are mutually exclusives. When one of these options is specified, the bbox command is applied to the first item selected by the tagOrId arguments. If no labelformat has been set for the item, bbox will return an empty array.

```
pathname becomes
```

```
$zinc->becomes();
```

Not yet implemented.

```
pathname bind tagOrId ?part?  ?sequence?  ?command?
```

```
@bindings = $zinc->bind(tagOrId, ?part?);
@binding = $zinc->bind(tagOrId, ?fpart?, sequence);
$zinc->bind(tagOrId, ?part?, sequence, '');
$zinc->bind(tagOrId, ?part?, sequence, command);
```

This command associates `command` with the item tag, item id, part tag `tagOrId`. If an event sequence matching `sequence` occurs for an item, or an item part, the command will be invoked. If all parameters are specified a new binding between `sequence` and `command` is established, overriding any existing binding for the sequence. If the first character of `command` is "+", then `command` augments the existing binding instead of replacing it. In this case the command returns an empty string. If the `command` parameter is omitted, the command returns the `command` associated with `tagOrId` and `sequence` or an error is raised if there is no such binding. If only `tagOrId` is specified the command returns a list of all the sequences for which there are bindings for `tagOrId`.

This widget command is similar to the Tk `bind` command except that it operates on TkZinc items instead of widgets. Another difference with the `bind` command is that only mouse and keyboard related events can be specified (such as `Enter`, `Leave`, `ButtonPress`, `ButtonRelease`, `Motion`, `KeyPress`, `KeyRelease`). The `bind` manual page is the most accurate place to look for a definition of `sequence` and `command` and for a general understanding of how the binding mecanism works.

The handling of events in the widget is done with respect to the current item and when applicable the current item part (see Item ids, tags and indices for a discussion of the `current` tag and the special tags used in bindings). `Enter` and `Leave` events

are trigerred for an item when it becomes or cease to be the current item. Mouse related events are reported with respect to the current item. Keyboard related events are reported with respect to the focus item if it exists (See the `focus` command for more on this).

It is possible that several bindings match a particular event sequence. When this occurs, all matching bindings are triggered. The order of invocation is as follow: the binding associated with the tag `all` is invoked first, followed by the bindings associated with the item tags in order, followed by the binding associated with the item id, followed by bindings associated with the item part tags if relevant, followed by the binding associated with the item part if relevant. If there are more than one binding for a single tag or id, only the most specific is triggered.

Two syntaxes are available for addressing item parts (for items having part ie. `track` `waypoint` and `tabular`): either an Id of the following form: `id:part` or by using the optionnal `part` argument.

If bindings have been registered for the widget window using the `bind` command, they are invoked in addition to bindings registered for the items using this widget command. The bindings for items will be invoked before the bindings for the window.

`pathname` **cget** `option`

```
$val = $zinc->cget(option);
```

Returns the current value of the widget option given by `option`. `option` may be any of the options described in the chapter `Widget options` .

`pathname` **chggroup** `tagOrId group ?adjustTransform?`

```
$zinc->chggroup(tagOrId, group, ?adjustTransform.?);
```

Move the item described by `tagOrId` in the group described by `group`. If `tagOrId` or `group` describes more than one item, the first in display list order will be used. If `adjustTransform` is specified, it will be interpreted as a boolean. A true value will lead to an adjustment of the item transform in order to maintain an identical display rendering of the item regardless of its new position in the display hierarchy. If `adjustTransform` is omitted, it defaults to false.

`pathname` **clone** `tagOrId ?attr value ...?`

```
$id = $zinc->clone(tagOrId, ?attr=>value, ...?);
```

Create an exact copy of all the items described by `tagOrId`. The copy goes recursively for group items (deep copy). After copying the pairs `attr value` are used, if any, to reconfigure the items. Any attribute that as no meaning in the context of some item is ignored. The items down the hierarchy of group items are not concerned by the configuration phase. The command returns the list of cloned items id in creation order (display list order of the models). No item id will be returned for items cloned in the hierarchy of cloned groups.

`pathname` **configure** `?option? ?value? ?option value ...?`

```
@options = $zinc->configure();
@option = $zinc->configure(option);
$zinc->configure(option=>value, ?option=>value,...?);
```

Query or modify the options of the widget. If no `option` is given, returns a list describing all the supported options in the standard format for Tk options (see the chapter Widget options for a list of available options). If an `option` is specified without a `value`, the command returns a list describing the named option in the standard Tk format. If some `option - value` pairs are given, then the corresponding options are changed and the command returns an empty string.

`pathname` **contour** `tagOrId ?operatorSpec coordListOrTagOrId?`

```
$contourNum = $zinc->contour(tagOrId);
$contourNum = $zinc->contour(tagOrId, operatorAndFlag, coordListOrTagOrId);
```

Manipulate contours on items that can handle multiples geometric contours. Currently only curve items can do this.

`tagOrId` specifies the item whose contours will be modified. If `tagOrId` describes more than one item, the first in display list order will be used.

If the command is invoked with only the tagOrId parameter, it returns the number of contours composing the item. In fact it always returns the number of contours after a command has been conducted, it happens that with this reduced form nothing is done except returning the number of contours.

On items that do not support multiple contours, the returned value is 0 or 1 depending on the item having a contour or not. track , waypoint , reticle , group , map , and yield 0 while rectangle , tabular , arc , icon , triangles , text , and window yield 1.

`coordListOrTagOrId` specifies a list of coordinates (either a flat list of X and Y, or a list similar to the one allowed for a curve item) or an item describing a contour. If a flat list is specified it should contain an even number of floating point values specifying the contour vertices X and Y in order. If a tag or an id is specified, it is should be from one of these classes: arc, curve, icon, rectangle, tabular, text, window. The external shape of the item will be used as the contour. If `coordListOrTagOrId` describes more than one item, the first in display list order will be used.

`operatorAndFlag` specifies the operation that will be carried. This can be:

`add` to extend the surface of the curve. In this case there is a mandatory flag describing the way the contour will be added. It may take the following values:

- 0 the list of points is taken unchanged. In this case, the `coordListOrTagOrId` parameter cannot be a tag and must be an explicit list of points.
- 1 the list of points is reverted, if needed, so that the points defines a **counterclockwise** contour.
- -1 the list of points is reverted,if needed, so that the points defines a **clockwise** contour.

The exact graphical effect depends on the value of the `-fillrule` as described in the `fillrule` type description.

`remove` to remove an existing contour

The following operations are no more available: `diff`, `inter`, `union`, `xor`; An error is generated if the items are not of a correct type or if the coordinate list is malformed.

`pathname` **coords** `tagOrId ?add/remove? ?contour? ?index? ?coordList?`

```
$zinc->coords(tagOrId, ?add/remove?, ?contour?, ?index?, ?coordList?);
```

Query or changes the coordinates of the item described by `tagOrId`. If the item is a group, query or set the translation applied to the group. If `tagOrId` describes more than one item, the first in display list order is used. The optional `contour` gives the contour, if available, that should be operated. The default contour is 0. The optional `index` gives the vertex index that should be operated in the given contour. The optional `coordList` is either a flat list (of one or more vertices described as X, Y floating point values) or a list of lists [X Y ?type?] such as described in the curve item. The coordinates will be used to replace or add coordinates to the current contour.

Almost all items can be manipulated by this command, the map item is the only current exception. The effect of the command can be quite different depending on the item. For icons, texts, windows, tabulars, the coordinates of the anchor can be modified or read. For groups, the coordinates of the origin of the transformation can be set or read. For tracks and waypoints, the coordinates of the current position can be set or read. For tracks setting the current position this way will make the previous position shift into the history. For reticles, the coordinates of the center can be set or read. For arcs and rectangles, the coordinates of the origin and corner can be set or read. For curves and triangles, coordinates of all points defining the item can be set or read. For all items that do not support multiple contours (currently all except curves) the `contour` parameter should be omitted or specified as zero.

When `coords` is used to get potentially more than one point, it returns **always** a list of lists. Each sub list contains X, Y, and 'c' if the point is a bezier control point.

When `coords` is used to get exactly one point (either because it is used to get the nth point of an item or because the item always has exactly one point (e.g. the item is a group, track, waypoint, map, or reticle)), it returns a flat list of 2 (or may be 3 for control points of curve items) values.

The optional parameters must be combined to produce a given behavior. Here are the various form recognized by the command:

- `pathname` **`coords`** `tagOrId contourIndex`
  `@coords = $zinc->`**`coords`**`(tagOrId, contourIndex);`

  Get all coordinates of contour at contourIndex. All items can answer if contourIndex is zero. Curves can handle other contours.

- `pathname` **`coords`** `tagOrId contourIndex coordList`
  `$zinc->`**`coords`**`(tagOrId, contourIndex, coordList);`

  Set all coordinates of contour at contourIndex. All items can do it if contourIndex is zero. Curves can handle other contours. For groups, icons, texts, windows, tabulars, reticles, tracks, waypoints, only the first vertex will be used. For rectangles and arcs, only the first two vertices will be used. Curves can handle any number of vertices.

- `pathname` **`coords`** `tagOrId contourIndex coordIndex`
  `($x, $y) = $zinc->`**`coords`**`(tagOrId, contourIndex, coordIndex);`

  Get coordinate at coordIndex in contour at contourIndex. All items can answer if contourIndex is zero. Curves can handle other contours. For groups, icons, texts, windows, tabulars, reticles, tracks, waypoints, coordIndex must be zero. For rectangles and arcs, coordIndex must zero or one.

- `pathname` **`coords`** `tagOrId contourIndex coordIndex coordList`
  `$zinc->`**`coords`**`(tagOrId, contourIndex, coordIndex, coordList);`

Set coordinate at coordIndex in contour at contourIndex. All items can do it if contourIndex is zero. Curves can handle other contours. For groups, icons, texts, windows, tabulars, reticles, tracks, waypoints, coordIndex must be zero. For rectangles and arcs, coordIndex must zero or one. **WARNING:** coordList must be a list of one list describing x, y and optionnaly the point type.

- `pathname` **coords** `tagOrId remove contourIndex coordIndex`
  `$zinc->`**coords**`(tagOrId, 'remove', contourIndex, coordIndex);`
  Remove coordinate at coordIndex in contour at contourIndex. Can only be handled by curves. Only curves can handle contourIndex other than zero.
- `pathname` **coords** `tagOrId add contourIndex coordList`
  `$zinc->`**coords**`(tagOrId, 'add', contourIndex, coordList);`
  Add coordinates at the end of contour at contourIndex. Can only be handled by curves. Only curves can handle contourIndex other than zero.
- `pathname` **coords** `tagOrId add contourIndex coordIndex coordList`
  `$zinc->`**coords**`(tagOrId, 'add', contourIndex, coordIndex, coordList);`
  Add coordinates at coordIndex in contour at contourIndex. Can only be handled by curves. Only curves can handle contourIndex other than zero.

And the slightly abbreviated forms:

- `pathname` **coords** `tagOrId`
  `@coords = $zinc->`**coords**`(tagOrId);`
  Get all coordinates of contour 0. See first form.
- `pathname` **coords** `tagOrId coordList`
  `$zinc->`**coords**`(tagOrId, coordList);`
  Set all coordinates of contour 0. See second form.
- `pathname` **coords** `tagOrId remove coordIndex`
  `$zinc->`**coords**`(tagOrId, 'remove', coordIndex);`
  Remove coordinate at coordIndex in contour 0. See fifth form.
- `pathname` **coords** `tagOrId add coordList`
  `$zinc->`**coords**`(tagOrId, 'add', coordList);`
  Add coordinates at the end of contour 0. See sixth form.

`pathname` **currentpart**

`$num = $zinc->`**currentpart**`();`

Returns a string specifying the item part that has the pointer. If the current item doesn't have parts or if the pointer is not over an item (no item has the `current` tag) the command returns `""`. The string can be either an integer describing a field index or the name of a special part of the item. Consult each item description to find out which part names can be reported.

`pathname` **cursor** `tagOrId index`

`$zinc->`**cursor**`(tagOrId, index);`

Set the position of the insertion cursor for the items described by `tagOrId` to be just before the character at `index`. If some of the items described by `tagOrId` don't support an insertion cursor, the command doesn't change them. The possible values for `index` are described in the section `Text indices` . The command returns an empty string.

`pathname` **dchars** `tagOrId first ?last?`

```
$zinc->dchars(tagOrId, first);
$zinc->dchars(tagOrId, first, last);
```

Delete the character range defined by the parameters `first` and `last` inclusive in all the items described by `tagOrId`. Items that doesn't support text indexing are skipped by the command. If `last` is not specified, the command deletes the character located at `first`. The command returns an empty string. `first` and `last` are indices as described in Text indices .

`pathname` **dtag** `tagOrId ?tagToDelete?`

```
$zinc->dtag(tagOrId);
$zinc->dtag(tagOrId, tagToDelete);
```

Delete the tag `tagToDelete` from the list of tags associated with each item named by `tagOrId`. If an item doesn't have the tag then it is leaved unaffected. If `tagToDelete` is omitted, `tagOrId` is used instead. The command returns an empty string as result.

`pathname` **find** `searchCommand ?arg arg ...?`

```
@items = $zinc->find(searchCommand, ?arg?, ...);
```

This command returns the list of all items selected by `searchCommand` and the `args`. See the addtag command for an explanation of `searchCommand` and the various `args`. The items are sorted in drawing order, topmost first. For example:

```
# to get the item under the mouse cursor:
$item = $zinc->find('withtag', 'current');


# to get the closest item of a point:
$closest = $zinc->find ('closest', $x, $y);


# to get direct children of an atomic group with a pathTag:
@children = $zinc->find('withtag', ".atomicGroup.");


# to get all groups recursively contained in a group
@groups = $zinc->find('withtype', 'group', ".aGroup*");
```

As detailled in addtag command the following searchCommands are possible:

- `pathname` **find** `above tagOrId`
  `$zinc->find('above', tagOrId);`
- `pathname` **find** `atpriority priority ?tagOrId?`
  `$zinc->find('atpriority', priority, ?tagOrId?);`
- `pathname` **find** `ancestors tagOrId ?tagOrId2?`
  `$zinc->find('ancestors', tagOrId, ?tagOrId2?);`
- `pathname` **find** `below tagOrId`
  `$zinc->find('below', tagOrId);`
- `pathname` **find** `closest x y ?halo?  ?startItem?, ?recursive?`
  `$zinc->find('closest', x, y, ?halo?, ?startItem?, ?recursive?);`
- `pathname` **find** `enclosed xo yo xc yc ?inGroup?  ?recursive?`
  `$zinc->find('enclosed', xo, yo, xc, yc, ?inGroup?, ?recursive?);`

- `pathname` **find** `overlapping xo yo xc yc ?inGroup? ?recursive?`
  `$zinc->`**find**`('overlapping', xo, yo, xc, yc, ?inGroup?, ?recursive?);`
- `pathname` **find** `withtag tagOrId`
  `$zinc->`**find**`('withtag', tagOrId);`
- `pathname` **find** `withtype type ?tagOrId?`
  `$zinc->`**find**`('withtype', type, ?tagOrId?);`

`pathname` **fit** `coordList error`

`@controls = $zinc->`**fit**`(coordList, error);`

This command fits a sequence of Bezier segments on the curve described by the vertices in `coordList` and returns a list of lists describing the points and control points for the generated segments. All the points on the fitted segments will be within `error` distance from the given curve. `coordList` should be either a flat list of an even number of coordinates in x, y order or a list of lists of point coordinates X, Y. The returned list can be directly used to create or change a curve item contour.

`pathname` **focus** `?tagOrId? ?itemPart?`

`($item, $part) = $zinc->`**focus**`();`
`$zinc->`**focus**`(tagOrId, ?itemPart?);`

Set the keyboard focus to the item described by `tagOrId`, and to its `itemPart` if the item has parts. If `tagOrId` describes more than one item, the first item in display list order that accept the focus is used. If no such item exists, the command has no effect. If `tagOrId` is an empty string the focus is reset and no item has the focus. If `tagOrId` is not specified, the command returns a list of two elements. The first is the id of the item with the focus or an empty string if no item has the focus. The second is the item part or an empty string if not appliable.

When the focus has been set to an item that support an insertion cursor, the item will display its cursor and the keyboard events will be directed to that item.

The widget receive keyboard events only if it has the window focus. It may be necessary to use the Tk `focus` command to force the focus to the widget window.

`pathname` **gdelete** `gradientName`

`$zinc->`**gdelete**`('fading');`

This command breaks the binding between the given gradient name and the named gradient. When the gradient will be no longer used it will be deallocated.

`pathname` **gettags** `tagOrId`

`@tags = $zinc->`**gettags**`(tagOrId);`

This command returns the list of all the tags associated with the item specified by `tagOrId`. If more than one item is named by `tagOrId`, then the topmost in display list order is used to return the result. If no item is named by `tagOrId`, then the empty list is returned.

`pathname` **gname** `?gradientDesc? gradientName`

`$zinc->`**gname**`('black:100|white:0/0', 'fading');`
`$exist = $zinc->`**gname**`('nameOrGradient');`

This command sets a name binding between the given gradient description and the given name. The name can be used in the same way the gradient description would be. The gradient will not be deallocated until the gdelete command is invoqued on the name (and no item use the gradient). This feature can be a big performance gain when using many gradients in an animation, the name acts here as a caching mecanism.

When gname is called with only one argument, it returns iehter false or the name of a named gradient if the argument is either a named gradient or the name of a named gradient.

`pathname` **group** `tagOrId`

```
$group = $zinc->group(tagOrId);
```

Returns the group containing the item described by `tagOrId`. If more than one item is named by `tagOrId`, then the topmost in display list order is used to return the result.

`pathname` **hasanchors** `tagOrId`

```
$bool = $zinc->hasanchor(tagOrId);
```

This command returns a boolean telling if the item specified by `tagOrId` supports anchors. If more than one item is named by `tagOrId`, then the topmost in display list order is used to return the result. If no items are named by `tagOrId`, an error is raised.

`pathname` **hasfields** `tagOrId`

```
$bool = $zinc->hasfields(tagOrId);
```

This command returns a boolean telling if the item specified by `tagOrId` supports fields. If more than one item is named by `tagOrId`, then the topmost in display list order is used to return the result. If no items are named by `tagOrId`, an error is raised.

`pathname` **hastag** `tagOrId tag`

```
$bool = $zinc->hastag(tagOrId, tag);
```

This command returns a boolean telling if the item specified by `tagOrId` has the specified tag. If more than one item is named by `tagOrId`, then the topmost in display list order is used to return the result. If no items are named by `tagOrId`, an error is raised.

`pathname` **index** `tagOrId index`

```
$num = $zinc->index(tagOrId, index);
```

This command returns a number which is the numerical index in the item described by `tagOrId` corresponding to `index`. The possible forms for `index` are described in Text indices . The command returns a value between 0 and the number of character in the item. If `tagOrId` describes more than one item, the index is processed in the first item supporting text indexing in display list order.

`pathname` **insert** `tagOrId before string`

```
$zinc->insert(tagOrId, before, string);
```

This command inserts `string` in each item described by `tagOrId` just before the text position described by `before`. The possible values for `before` are described in Text indices . Items that doesn't support text indexing are skipped by the command. The command returns an empty string.

`pathname` **itemcget** `tagOrId ?fieldId? attr`

```
$val = $zinc->itemcget(tagOrId, attr);
$val = $zinc->itemcget(tagOrId, field, attr);
```

Returns the current value of the attribute given by `attr` for the item named by `tagOrId`. If `tagOrId` name more than one item, the topmost in display list order is used. If `field` is given, it must be a valid field index for the item or an error will be reported. If a field index is given, the command will interpret `attr` as a field attribute (see `field`), otherwise it will be interpreted as an item attribute (see the chapter `Item types`). If the attribute is not available for the field or item type, an error is reported.

`pathname` **itemconfigure** `tagOrId ?fieldId? ?attr? ?value? ?attr value ...?`

```
@attribs = $zinc->itemconfigure(tagOrId);
@attrib = $zinc->itemconfigure(tagOrId, attrib);
$zinc->itemconfigure(tagOrId, attrib=>value, ?attrib=>value?, ...);
@attrib = $zinc->itemconfigure(tagOrId, fieldId, attrib);
$zinc->itemconfigure(tagOrId, fieldId, attrib=>value, ?attrib=>value?, ...);
```

Query or modify the attributes of an item or field. If no attribute is given, returns a list of lists describing all the supported attributes in the same format as for a single attribute, as described next. If a single attribute is specified without a value, the command returns a list describing the named attribute. Each attribute is described by a list with the following content: the attribute name, the attribute type, a boolean telling if the attribute is read-only, an empty string, and the current value of the attribute. In the two querying forms of the command the topmost item described by `tagOrId` is used.

If at least one attribute - value pair is given, then the corresponding attributes are changed for all the items described by `tagOrId` and the command returns an empty string. If `field` is given, it must be a valid field index for the item or an error will be reported. If a field index is given, the command will interpret the given attributes as field attributes, otherwise they will be interpreted as item attributes. If an attribute does not belong to the item or field, an error is reported. When configuring a set of item defiend by a tag, all items must then accept these attributes.

`pathname` **lower** `tagOrId ?belowThis?`

```
$zinc->lower(tagOrId);
$zinc->lower(tagOrId, belowThis);
```

Reorder all the items given by `tagOrId` so that they will be under the item given by `belowThis`. If `tagOrId` name more than one item, their relative order will be preserved. If `tagOrId` doesn't name an item, an error is raised. If `belowThis` name more than one item, the bottom most them is used. If `belowThis` doesn't name an item, an error is raised. If `belowThis` is omitted the items are put at the bottom most position of their respective groups. The command ignore all items named by `tagOrId` that are not in the same group than `belowThis` or, if not specified, in the same group than the first item named by `tagOrId`. The command returns an empty string. As a side affect of this command, the `-priority` attribute of all the reordered items is ajusted to match the priority of the `belowThis` item (or the priority of the bottom most item).

`pathname` **monitor** `?onOff?`

```
$bool = $zinc->monitor();
$zinc->monitor(onOff);
```

This command controls the gathering of performance data. The data gathering is inited and turned on when the command is called with a boolean true parameter. The gathering is stopped if the command is called with a boolean false parameter. If the command is called with no parameters or with a boolean false parameter, it returns a string describing the currently collected data. The other form of the command returns the empty string.

`pathname` **numparts** `tagOrId`

```
$num = $zinc->numparts(tagOrId);
```

This command tells how many fieldId are available for event bindings or for field configuration commands in the item specified by `tagOrId`. If more than one item is named by `tagOrId`, the topmost in display list order is used to return the result. If no items are named by `tagOrId`, an error is raised. This command returns always 0 for items which do not support fields. The command `hasfields` may be used to decide whether an item has fields.

`pathname` **postscript**

```
$zinc->postscript();
```

Not yet implemented.

`pathname` **raise** `tagOrId ?aboveThis?`

```
$zinc->raise(tagOrId);
$zinc->raise(tagOrId, aboveThis);
```

Reorder all the items given by `tagOrId` so that they will be above the item given by `aboveThis`. If `tagOrId` name more than one item, their relative order will be preserved. If `tagOrId` doesn't name an item, an error is raised. If `aboveThis` name more than one item, the topmost in display list order is used. If `aboveThis` doesn't name an item, an error is raised. If `aboveThis` is omitted the items are put at the top most position of their respective groups. The command ignore all items named by `tagOrId` that are not in the same group than `aboveThis` or, if not specified, in the same group than the first item named by `tagOrId`. The command returns an empty string. As a side affect of this command, the `-priority` attribute of all the reordered items is ajusted to match the priority of the `aboveThis` item (or the priority of the topmost item).

`pathname` **remove** `tagOrId ?tagOrId ...?`

```
$zinc->remove(tagOrId, ?tagOrId?, ...);
```

Delete all the items named by each `tagOrId`. The command returns an empty string.

`pathname` **rotate** `tagOrId angle ?degree?  ?centerX centerY?`

```
$zinc->rotate(tagOrId, angle);
$zinc->rotate(tagOrId, angle, centerX, centerY);
```

Add a rotation to the items or the transform described by `tagOrId`. If `tagOrId` describes a named transform then this transform is used to do the operation. If `tagOrId` describes more than one item then all the items are affected by the operation. If `tagOrId` describes neither a named transform nor an item, an error is raised. The angle is given in radian if `degree` is omitted or false, if it is specified as true, the angle is in degrees. The last two optional parameters describe the center of rotation, which defaults to the origin.

`pathname` **scale** `tagOrIdOrTName xFactor yFactor ?centerX centerY?`

```
$zinc->scale(tagOrIdOrTName, xFactor, yFactor);
$zinc->scale(tagOrIdOrTName, xFactor, yFactor, centerX, centerY);
```

Add a scale factor to the items or the transform described by `tagOrId`. If `tagOrId` describes a named transform then this transform is used to do the operation. If `tagOrId` describes more than one item then all the items are affected by the operation. If `tagOrId` describes neither a named transform nor an item, an error is raised. A separate factor is specified for X and Y. The optional parameters describe the center of scaling, which defaults to the origin.

`pathname` **select** `option ?tagOrId? ?arg?`

```
$zinc->select(option, ?tagOrId?, ?arg?);
```

Manipulates the selection as requested by `option`. `tagOrId` describes the target item. This item must support text indexing and selection. If more than one item is referred to by `tagOrId`, the first in display list order that support both text indexing and selection will be used. Some forms of the command include an `index` parameter, this parameter describes a textual position within the item and should be a valid index as described in Text indices . The valid forms of the command are :

- `pathname` **select** `adjust tagOrId index`
  `$zinc->select('adjust', tagOrdId, index);`
  Adjust the end of the selection in `tagOrId` that is nearest to the character given by `index` so that it is at `index`. The other end of the selection is made the anchor for future select to commands. If the selection is not currently in `tagOrId`, this command behaves as the select to command. The command returns an empty string.
- `pathname` **select** `clear`
  `$zinc->select('clear');`
  Clear the selection if it is in the widget. If the selection is not in the widget, the command has no effect. Return an empty string.
- `pathname` **select** `from tagOrId index`
  `$zinc->select('from', tagOrdId, index);`
  Set the selection anchor point for the widget to be just before the character given by `index` in the item described by `tagOrId`. The command has no effect on the selection, it sets one end of the selection so that future select to can actually set the selection. The command returns an empty string.
- `pathname` **select** `item`
  `($item, $part) = $zinc->select('item');`
  Returns a list of two elements. The first is the id of the selected item if the selection is in an item on this widget; Otherwise the first element is an empty string. The second element is the part of the item (track, waypoint or tabular item only) or the empty string.
- `pathname` **select** `to tagOrId index`
  `$zinc->select('to', tagOrdId, index);`
  Set the selection to be the characters that lies between the selection anchor and `index` in the item described by `tagOrId`. The selection includes the character given by `index` and includes the character given by the anchor point if `index` is greater or

equal to the anchor point. The anchor point is set by the most recent select adjust or select from command issued for this widget. If the selection anchor point for the widget is not currently in `tagOrId`, it is set to the character given by index. The command returns an empty string.

`pathname` **skew** `tagOrIdOrTName xSkewAngle ySkewAngle`

```
$zinc->skew(tagOrIdOrTName, xSkewAngle, ySkewAngle);
```

Add a skew (or shear) transform to the to the items or the transform described by `tagOrIdOrTName`. If `tagOrId` describes a named transform then this transform is used to do the operation. If `tagOrId` describes more than one item then all the items are affected by the operation. If `tagOrId` describes neither a named transform nor an item, an error is raised. The angles are given in radian.

`pathname` **smooth** `coordList`

```
@coords = $zinc->smooth(coordList);
```

This command computes a sequence of segments that will smooth the polygon described by the vertices in `coordList` and returns a list of lists describing points of the generated segments. These segments are approximating a Bezier curve. `coordList` should be either a flat list of an even number of coordinates in x, y order, or a list of lists of point coordinates X, Y. The returned list can be used to create or change the contour of a curve item.

`pathname` **tapply**

```
$zinc->tapply();
```

Not yet implemented.

`pathname` **tcompose** `tagOrIdOrTName tName ?invert?`

```
$zinc->tcompose(tagOrIdOrTName, tName);
$zinc->tcompose(tagOrIdOrTName, tName, invert);
```

Modify either the named transform `tagOrIdOrTName` or the corresponding item by composing the `tName` transform. The `invert` boolean, if specified, causes the `tName` transform to be inverted prior composition.

If `tagOrIdOrTName` describes neither a named transform nor an item, an error is raised. If `tName` does not describe a named transform an error is raised.

`pathname` **tdelete** `tName`

```
$zinc->tdelete(tName);
```

Destroy a named transform. If the given name is not found among the named transforms, an error is raised.

`pathname` **tget** `tagOrIdOrTName ?selector?`

```
($m00, $m01, $m10, $m11, $m20, $m21) = $zinc->tget(tagOrIdOrTName);
($xtranslate, $ytranslate, $xscale, $yscale, $angle, $xskew) =
$zinc->tget(tagOrIdOrTName, 'all');
($xtranslate, $ytranslate) = $zinc->tget(tagOrIdOrTName, 'translate');
($xscale, $yscale) = $zinc->tget(tagOrIdOrTName, 'scale');
($angle) = $zinc->tget(tagOrIdOrTName, 'rotate');
($xskew) = $zinc->tget(tagOrIdOrTName, 'skew');
```

With only one argument, get the six elements of the 3x4 matrix used in affine transformation for `tagOrIdOrTName`. The result is compatible with the tset method. With optional second parameter 'all' returns the transform decomposed in translation, scale, rotation, skew and return the list in this order, With 'translation', 'scale', 'rotation', 'skew' optional second parameter, returns the corresponding values.

`pathname` **transform** `?tagOrIdFrom? tagOrIdTo coordList`

```
@coords = $zinc->transform(tagOrIdTo, coordList);
@coords = $zinc->transform(tagOrIdFrom, tagOrIdTo, coordList);
```

This command returns a list of coordinates obtained by transforming the coordinates given in `coordList` from the coordinate space of the transform or item described by `tagOrIdFrom` to the coordinate space of the transform or item described by `tagOrIdTo`. If `tagOrIdFrom` is omitted it defaults to the window coordinate space. If either `tagOrIdFrom` or `tagOrIdTo` describes more than one item, the topmost in display list order is used. If either `tagOrIdFrom` or `tagOrIdTo` doesn't describe either a transform or an item, an error is raised. The `coordList` should either be a flat list containing an even number of coordinates each point having two coordinates, or a list of lists each sublist of the form [X Y ?pointtype?]. The returned coordinates list will be isomorphic to the list given as argument.

It is possible to convert from window coordinate space to the coordinate space of any item. This is done by omitting `?tagOrIdFrom?` and specifying in `tagOrIdTo`, the id of the item. It can also be done by using the predefined tag 'device' as first argument.

It is also possible to convert from the coordinate space of an item to the window coordinate space by using the predefined tag 'device' as second argument.

For example:

- `($x, $y) = $zinc->transform('device', $mygroup, [$xdev, $ydev]);` transforms the point described by `$xdev,$ydev` in window coordinates, to `$mygroup` coordinates in `$x,$y`;
- `($xdev, $ydev) = $zinc->transform($mygroup, 'device', [$x, $y]);` transforms the point described by `$x,$y` in `$mygroup` coordinates, to window coordinates in `$xdev,$ydev`
- `($x2, $y2) = $zinc->transform($group1, $group2, [$x1, $y1]);` transforms the point described by `$x1,$y1` in `$group1` coordinates, to `$group2` coordinates in `$x2,$y2`;

`pathname` **translate** `tagOrIdOrTName xAmount yAmount ?absolute?`

```
$zinc->translate(tagOrdIdOrTName, xAmount, yAmount, ?absolute?);
```

Add a translation to the items or the transform described by `tagOrIdOrTName`. If `tagOrIdOrTName` describes a named transform then this transform is used to do the operation. If `tagOrIdOrTName` describes more than one item then all the items are affected by the opration. If `tagOrIdOrTName` describes neither a named transform nor an item, an error is raised. A separate value is specified for X and Y. If the optionnal `?absolute?` parameter is true, it will set an absolute translation to the `tagOrIdOrTName`

`pathname` **treset** `tagOrIdOrTName`

```
$zinc->treset(tagOrdIdOrTName);
```

Set the named transform or the transform for the items described by `tagOrIdOrTName` to identity. If `tagOrIdOrTName` describes neither a named transform nor an item, an error is raised.

`pathname` **trestore** `tagOrId tName`

`$zinc->`**trestore**`(tagOrdId, tName);`

Set the transform for the items described by `tagOrId` to the transform named by `tName`. If `tagOrId` doesn't describe any item or if the transform named `tName` doesn't exist, an error is raised.

`pathname` **tsave** `?tagOrIdOrTName? tName ?invert?`

```
$zinc->tsave(tName);
$zinc->tsave(tagOrdIdOrTName, tName);
$zinc->tsave(tagOrdIdOrTName, tName, invert);
```

Create (or reset) a transform associated with the name `tName` with initial value the transform associated with the item `tagOrIdOrTName`. If `tagOrIdOrTName` describes more than one item, the topmost in display list order is used. If `tagOrIdOrTName` doesn't describe any item or named transformation, an error is raised. If `tName` already exists, the transform is set to the new value. This command is the only way to create a named transform. If `tagOrIdOrTName` is not specified, the command returns a boolean telling if the name is already in use. The `invert` boolean, if specified, cause the transform to be inverted prior to be saved.

It is possible to create a new named transformation from the identity by using the predefined tag 'identity': `$zinc->tsave('identity', 'myTransfo');`

`pathname` **tset** `tagOrIdOrTName m00 m01 m10 m11 m20 m21`

`$zinc->`**tset**`(tagOrIdOrTName, m00, m01, m10, m11, m20, m21);`

Set the six elements of the 3x4 matrix used in affine transformation for `tagOrIdOrTName`. **BEWARE that depending on mij values, it is possible to define a not inversible matrix which will end up in core dump. This method must BE USED CAUTIOUSLY.**

`pathname` **type** `tagOrId`

`$name = $zinc->`**type**`(tagOrdId);`

This command returns the type of the item specified by `tagOrId`. If more than one item is named by `tagOrId`, then the type of the topmost item in display list order is returned. If no items are named by `tagOrId`, an error is raised.

`pathname` **vertexat** `tagOrId x y`

`($contour, $vertex, $edgevertex) = $zinc->`**vertexat**`(tagOrdId, x, y);`

Return a list of values describing the vertex and edge closest to the *window coordinates* `x` and `y` in the item described by `tagOrId`. If `tagOrId` describes more than one item, the first item in display list order that supports vertex picking is used. The list consists of the index of the contour containing the returned vertices, the index of the closest vertex and the index of a vertex next to the closest vertex that identify the closest edge (located between the two returned vertices).

Chapter 6

Item types

This chapter introduces the item types that can be used in TkZinc. Each item type provides a set of options that may be used to query or change the item behavior. Some item types cannot be used with some widget commands, or use special parameters with some commands. Those cases are noted in the description of the item.

6.1 Group items

Group items are used for grouping objects together. Their usage is very powerfull and their use is best described in the previous chapter Groups, Display List, Clipping and Transformations .

Applicable attributes for `group` are:

`-alpha` alpha
: Specifies the transparency to compose with the children transparencies. Needs the openGL extension.

`-atomic` boolean
: Specifies if the group should report itself or its components during a search or for binding related operations. This attribute enable the use of a group as a single complex object build from smaller parts. It is possible to search for this item or use it in bindings without dealing with its smaller parts. The defaut value is `false`.

`-clip` item
: The item used to clip the children of the group. The shape of this item define an area that is used as a clipping shape when drawing the children of the group. Most items can be used here but notable exceptions are the `reticle` and `map` items. The default value is `""` which means that no clipping will be performed.

`-composealpha` boolean
: Specifies if the alpha value inherited from the parent group must be composed with the alpha of this group. The default value is `true`.

`-composerotation` boolean
: Specifies if the current rotation should be composed with the local transform. The default value is `true`.

`-composescale` boolean
: Specifies if the current scale should be composed with the local transform. The default value is `true`.

`-priority` priority
: The absolute position in the stacking order among siblings of the same parent group. The default value is `1`.

`-sensitive` boolean
: Specifies if the item and all its children should react to events. The defaut value is `true`.

`-tags` taglist
: The list of tags associated with the item. The default value is `""`.

`-visible` boolean
: Specifies if the item and all its children is displayed. The defaut value is `true`.

6.2 Track items

Track items have been designed for figuring out typical radar information for Air Traffic Control. However they may certainly be used by other kinds of radar view and surely by other kind of plan view with many moving objects and associated textual information.

A track is composed of two main parts:

- The first one is purely graphic and is composed of many parts, some of them being identified by their "partName":
  - the **current position** of the object. Its partName is `position`.
  - a **speed vector** which size depends on the attribute `-speedvector` for the track and the option `-speedvectorlength` . This speed vector may be set visible or not, sensitive or other attributes can be set such as color, width, ticks, mark at the end... Its partName is `speedvector`.
  - a **leader** which links the current position to the label. The leader may be visible or not, sensitive or not, and other graphic characteristics can be modified. Its partName is `leader`.
  - **past positions** which are previous position after the track has been moved by the `coords` command. The number of such past positions, their visibility and other graphic characteristics can be be modified. This part is never sensitive.
  - a **marker**, which is a circle around the current position. This marker can be visible or not and other graphic characteristics can be configured. The marker is never sensitive.
  - a **connection**, which is a link with another track or waypoint item; links are drawn between their **current position**. This connection may be visible or not, sensitive or not, and other graphic characteristics can be be modified. Its partName is `connection`.
- the second part is a block of texts described by a labelformat (see chapter `Labels, labelformats, and fields` . Each text can have its graphic decorations (alignment, background, images, borders...). These attributes are listed in the chapter `Labels, label formats and fields` and can be changed by the command `itemconfigure` .

The following picture shows a simple `track` with a label of 5 fields and 5 past positions. This track also shows a marker, the circle around the current position.

An other very important feature of `track` item is that TkZinc offers an anti-overlap manager. This manager tries to avoid any overlap of tracks labels. It also avoids that the label overlap the speed-vector. This manager is stable over time: there should be few cases where labels are moved to a very

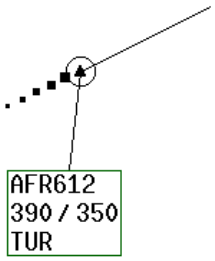

Figure 6.1: A track with a label composed of 5 fields

different position. This manager applies to all tracks included in a group (by default the group 1). It can be enabled/disabled with the TkZinc attribute `-overlapmanager` . New labels positions are computed by the overlap manager every time a track is moved, a track is created or destroyed and every time the TkZinc widget is resized. Due to software licence limitation, TkZinc *do not include* the very last version of this anti-overlap manager. If you are interested in this anti-overlap manager, please contact Didier Pavet at `pavet@cena.fr`.

Track items can be linked together or to waypoint items. The line figuring the link is configurable.

Applicable attributes for `track` are :

`-circlehistory boolean`
: If set to true the track history will be plotted as circles otherwise it will be plotted as squares. The default value is `false`.

`-composealpha boolean`
: Specifies if the alpha value inherited from the parent group must be composed with the alpha of this item. The default value is `true`.

`-composerotation boolean`
: Specifies if the current rotation should be composed with the local transform. The default value is `true`.

`-composescale boolean`
: Specifies if the current scale should be composed with the local transform. The default value is `true`.

`-connecteditem item`
: The `track` or `waypoint` item at the other end of the connection link. The default value is `""` which means that no connection link will be drawn.

`-connectioncolor gradient`
: The uniform (possibly transparent) color of the connection link. The first color of a real gradient color will be used. The default value is the current value of the widget option `-forecolor` .

`-connectionsensitive boolean`
: Specifies if the connection link is sensitive. The actual sensitivity is the logical `and` of this attribute and of the item `-sensitive` attribute. The default value is `true`.

`-connectionstyle linestyle`
: The line style of the connection link. The default value is `simple`.

`-connectionwidth dimension`
: The width of the connection link. The default value is `1`.

`-filledhistory boolean`
If set to true the track history will be filled otherwise it will be outlined. The default value is `true`.

`-filledmarker boolean`
If set to true the circular marker will be filled otherwise it will be outlined. The default value is `false`.

`-frozenlabel boolean`
Specifies if the label should be frozen at its current location to prevent the anti overlapping system from moving it. The default value is `false`.

`-historycolor gradient`
The uniform (possibly transparent) color of the track history. The first color of a real gradient color will be used. The default value is the current value of the widget option `-forecolor` .

`-labelanchor anchor`
The anchor used in positionning the label. The default value is `center`.

`-labelangle angle`
The angle in degrees between the label anchor and the normal to the speed vector. This attribute works with the `-labeldistance` attribute to specify a position for the label anchor with respect to the item origin. There is another alternative method for label positioning which is implemented with the `-labeldx` and `-labeldy` methods. Simultaneous use of the two methods should be done with care as there is no automatic update of values from the `-labeldx` , `-labeldy` set to the `-labeldistance` , `-labelangle` set. The default value is `20`.

`-labelconvergencestyle dimension`
XXX New. To be documented. The default value is ??.

`-labeldistance dimension`
The minimum distance in pixels between the track position and the label anchor. See the explanation of the `-labelangle` attribute for some more details. The default value is 50.

`-labeldx dimension`
The X offset between the track position and the label anchor. The default value is computed from the values in the `-labeldistance` and `-labelangle` attributes.

`-labeldy dimension`
The Y offset between the track position and the label anchor. The default value is computed from the values in the `-labeldistance` and `-labelangle` attributes.

`-labelformat labelformat`
Geometry of the label fields. The default value is `""` which means that no label will be displayed.

`-labelpreferredangle angle`
XXX New. To be documented. The default value is ??.

`-lastasfirst boolean`
If set to true, the last position in the history will be drawn in the same color as the current position instead of being drawn in the history color. The default value is `false`.

`-leaderanchors leaderanchors`
The attachment of the leader on the label left or right side (whether the label is on the right or left of the current position). The default value is `""` which means that the leader anchor is at the label center, whatever the label position.

`-leadercolor gradient`
: The uniform (possibly transparent) color of the label leader. The first color of a real gradient color will be used. The default value is the current value of the widget option `-forecolor` .

`-leaderfirstend lineend`
: Describes the arrow shape at the current position end of the leader. The default value is `""`.

`-leaderlastend lineend`
: Describes the arrow shape at the label end of the leader. The default value is `""`.

`-leadersensitive boolean`
: Specifies if the label leader is sensitive. The actual sensitivity is the logical `and` of this attribute and of the item `-sensitive` attribute. The default value is `true`.

`-leadershape lineshape`
: The shape of the label leader. The default value is `straight`.

`-leaderstyle linestyle`
: The line style of the label leader. The default value is `simple`.

`-leaderwidth dimension`
: The width of the label leader. The default value is `1`.

`-markercolor gradient`
: The uniform (possibly transparent) color of the circular marker. The first color of a real gradient color will be used. The default value is the current value of the widget option `-forecolor` .

`-markerfillpattern bitmap`
: The pattern to use when filling the circular marker. The default value is `""`.

`-markersize dimension`
: The (scale sensitive) size of the circular marker. The default value is `0` which turn off the display of the marker.

`-markerstyle linestyle`
: The line style of the marker outline. The default value is `simple`.

`-mixedhistory boolean`
: If true the track history will be plotted with dots every other position. The default value is `false`.

`-numfields unsignedint`
: Gives the number of fields available for the label. This attribute is read only.

`-position point`
: The current location of the track. The default value is `"0 0"`.

`-priority priority`
: The absolute position in the stacking order among siblings of the same parent group. The default value is `1`.

`-sensitive boolean`
: Specifies if the item should react to events. The default value is `true`.

`-speedvector point`
: The speed vector Δx and Δy in unit / minute. The default value is `"0 0"` which results in no speed vector displayed.

`-speedvectorcolor gradient`

The uniform (possibly transparent) color of the track's speed vector. The first color of a real gradient color will be used. The default value is the current value of the widget option -forecolor .

-speedvectormark boolean
If set a small point is drawn at the end of the speed vector. The point is drawn with the speed vector color. The default is false.Not yet available without openGL

-speedvectorsensitive boolean
Specifies if the track's speed vector is sensitive. The actual sensitivity is the logical and of this attribute and of the item -sensitive attribute. The default value is true.

-speedvectorticks boolean
If set a mark is drawn at each minute position. The default is false. Not yet available without openGL

-speedvectorwidth dimension
New. XXX To be documented. The default value is 1.

-symbol bitmap
The symbol displayed at the current position. The default value is the current value of the widget option -tracksymbol .

-symbolcolor gradient
The uniforme (possibly transparent) color of the symbol displayed at the current position. The first color of a real gradient color will be used. The default value is the current value of the widget option -forecolor .

-symbolsensitive boolean
Specifies if the current position's symbol is sensitive to events. The actual sensitivity is the logical and of this attribute and of the item -sensitive attribute. The default value is true.

-tags taglist
The list of tags associated with the item. The default value is "".

-visible boolean
Specifies if the item is displayed. The default value is true.

-historyvisible boolean
Specifies whether the item should display its history according to the options -trackvisiblehistor
and -trackmanagedhistorysize . The default value is true.

6.3 WayPoint items

Waypoints items have been initially designed for figuring out typical fixed position objects (i.e. beacons or fixes in the ATC vocabulary) with associated block of texts on a radar display for Air Traffic Control. They supports mouse event handling and interactions. However they may certainly be used by other kinds of radar view or even by other kind of plan view with many geographical objects and associated textual information.

A waypoint is composed of the following parts:

- the **position** of the waypoint. Its partName is position.
- a **leader** which links the current position to the label. The leader may be visible or not, sensitive or not, and other graphic characteristics can be be modified. Its partName is leader.

- a **label** which is a block of texts described by a labelformat (see chapter Labels, labelformat, and fields . Each text can have its graphic decorations (alignment, background, images, borders...). These attributes are listed in the chapter Labels, label formats and fields and can be changed by the command itemconfigure .
- a **connection**, which is a link with another `waypoint` or `track` item. This connection may be visible or not, sensitive or not, and other graphic characteristics can be be modified. Its partName is `connection`.

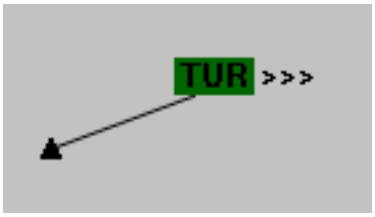

Figure 6.2: A waypoint with a label composed of five fields; fields have borders

Applicable attributes for `waypoint` are:

`-composealpha` boolean
: Specifies if the alpha value inherited from the parent group must be composed with the alpha of this item. The default value is `true`.

`-composerotation` boolean
: Specifies if the current rotation should be composed with the local transform. The default value is `true`.

`-composescale` boolean
: Specifies if the current scale should be composed with the local transform. The default value is `true`.

`-connecteditem` item
: The `track` or `waypoint` item at the other end of the connection link. The default value is `""` which means that no connection link will be drawn.

`-connectioncolor` gradient
: The uniform (possibly transparent) color of the connection link. The first color of a real gradient color will be used. The default value is the current value of the widget option `-forecolor` .

`-connectionsensitive` boolean
: Specifies if the connection link is sensitive. The actual sensitivity is the logical `and` of this attribute and of the item `-sensitive` attribute. The default value is `true`.

`-connectionstyle` linestyle
: The line style of the connection link. The default value is `simple`.

`-connectionwidth` dimension
: The width of the connection link. The default value is `1`.

`-filledmarker` boolean
: If set to true the circular marker will be filled otherwise it will be outlined. The default value is `false`.

`-labelanchor` anchor
: The anchor used in positionning the label. The default value is `center`.

`-labelangle` *angle*
: The angle in degrees between the label anchor and the normal to the speed vector. This attribute works with the `-labeldistance` attribute to specify a position for the label anchor with respect to the item origin. There is another alternative method for label positioning which is implemented with the `-labeldx` and `-labeldy` methods. Simultaneous use of the two methods should be done with care as there is no automatic update of values from the `-labeldx` , `-labeldy` set to the `-labeldistance` , `-labelangle` set. The default value is `20`.

`-labeldistance` *dimension*
: The minimum distance in pixels between the way point position and the label anchor. See the explanation of the `-labelangle` attribute for some more details. The default value is `50`.

`-labeldx` *dimension*
: The X offset between the way point position and the label anchor. The default value is computed from the values in the `-labeldistance` and `-labelangle` attributes.

`-labeldy` *dimension*
: The Y offset between the way point position and the label anchor. The default value is computed from the values in the `-labeldistance` and `-labelangle` attributes.

`-labelformat` *labelformat*
: Geometry of the label fields. The default value is `""` which means that no label will be displayed.

`-leaderanchors` *leaderanchors*
: The attachment of the leader on the label left or right side (whether the label is on the right or left of the current position). The default value is `""` which means that the leader anchor is at the label center, whatever the label position.

`-leadercolor` *gradient*
: The uniform (possibly transparent) color of the label leader. The first color of a real gradient color will be used. The default value is the current value of the widget option `-forecolor` .

`-leaderfirstend` *lineend*
: Describes the arrow shape at the current position end of the leader. The default value is `""`.

`-leaderlastend` *lineend*
: Describes the arrow shape at the label end of the leader. The default value is `""`.

`-leadersensitive` *boolean*
: Specifies if the label leader is sensitive. The actual sensitivity is the logical `and` of this attribute and of the item `-sensitive` attribute. The default value is `true`.

`-leadershape` *lineshape*
: The shape of the label leader. The default value is `straight`.

`-leaderstyle` *linestyle*
: The line style of the label leader. The default value is `simple`.

`-leaderwidth` *dimension*
: The width of the label leader. The default value is `1`.

`-markercolor` *gradient*
: The uniform (possibly transparent) color of the circular marker. The first color of a real gradient color will be used. The default value is the current value of the widget option `-forecolor` .

`-markerfillpattern` bitmap
: The pattern to use when filling the circular marker. The default value is `""`.

`-markersize` dimension
: The (scale sensitive) size of the circular marker. The default value is `0` which turn off the display of the marker.

`-markerstyle` linestyle
: The line style of the marker outline. The default value is `simple`.

`-numfields` unsignedint
: Gives the number of fields available for the label. This attribute is read only.

`-position` point
: The current location of the way point. The default value is `"0 0"`.

`-priority` priority
: The absolute position in the stacking order among siblings of the same parent group. The default value is `1`.

`-sensitive` boolean
: Specifies if the item should react to events. The default value is `true`.

`-symbol` bitmap
: The symbol displayed at the current position. The default value is `AtcSymbol15`.

`-symbolcolor` gradient
: The uniform (possibly transparent) color of the symbol displayed at the current position. The first color of a real gradient color will be used. The default value is the current value of the widget option `-forecolor` .

`-symbolsensitive` boolean
: Specifies if the current position's symbol is sensitive to events. The actual sensitivity is the logical `and` of this attribute and of the item `-sensitive` attribute. The default value is `true`.

`-tags` taglist
: The list of tags associated with the item. The default value is `""`.

`-visible` boolean
: Specifies if the item is displayed. The default value is `true`.

6.4 Tabular items

Tabular items have been initially designed for displaying block of textual information, organised in lists or spread out on a radar display.

A tabular item is mainly composed of a *label* which is a block of texts described by a labelformat (see chapter `Labels, labelformats and fields` . Each text can have its graphic decorations (alignment, background, images, borders...). This attributes are listed in the chapter `Labels, label formats and fields` and can be changed by the command `itemconfigure` . A tabular can be attached with the `-connecteditem` attribute to the label of a `track` , `waypoint` or another `tabular` .

Applicable attributes for `tabular` are:

`-anchor` anchor
: The anchor used in positionning the item. The default value is `nw`.

`-composealpha` boolean

Specifies if the alpha value inherited from the parent group should be composed with the alpha of this item. The default value is `true`.

`-composerotation boolean`
Specifies if the current rotation should be composed with the local transform. The default value is `true`.

`-composescale boolean`
Specifies if the current scale should be composed with the local transform. The default value is `true`.

`-connecteditem item`
Specifies the The `track`, `waypoint` or `tabular` item relative to which this item is placed. Connected item should be in the same group. The default value is `""`.

`-connectionanchor anchor`
Specifies the anchor on the connected item. The default value is `sw`.

`-labelformat labelformat`
Geometry of the label fields. The default value is `""` which means that nothing will be displayed.

`-numfields unsignedint`
Gives the number of fields available for the label. This attribute is read only.

`-position point`
The item's position relative to the anchor (if no connected item specified). The default value is `"0 0"`.

`-priority priority`
The absolute position in the stacking order among siblings of the same parent group. The default value is `1`.

`-sensitive boolean`
Specifies if the item should react to events. The default value is `true`.

`-tags taglist`
The list of tags associated with the item. The default value is `""`.

`-visible boolean`
Specifies if the item is displayed. The default value is `true`.

6.5 Text items

Text items are used for displaying text. They can also be used for text input. In this case, they must get the focus for keyboards events with the command `focus` . Many TkZinc options (see chapter `Widget options` can be used for configuring the text input (for example : `-insertbackground` , `-insertofftime` `-insertontime` , `-insertwidth`).

With and without openGL, text items can be rotated or scaled. However, attributes `-composerotation` and `-composescale` must be set before rotation and scaling.

A Tcl module, zincText is available, it provides simple bindings for interactive text input. For enabling interactive text editing on an item, the item should be sensitive and should have the tag "text".

A Perl module, called Tk::Zinc::Text (see the section `Tk::Zinc::Text`) is provided for easing text input in text items (it can also be used for text input in labelled items such as `track` , `waypoint` or `tabular`).

Applicable attributes for `text` are:

`-alignment alignment`

Specifies the horizontal alignment of the lines in the item. The default value is `left`.

`-anchor` *anchor*
: The anchor used in positionning the item. The default value is `nw`.

`-color` *gradient*
: Specifies the uniform (possibly transparent) color for drawing the text characters, the overstrike and underline lines. The first color of a real gradient color will be used. The default value is the current value of the widget option `-forecolor` .

`-composealpha` *boolean*
: Specifies if the alpha value inherited from the parent group should be composed with the alpha of this item. The default value is `true`.

`-composerotation` *boolean*
: Specifies if the current rotation should be composed with the local transform. The default value is `false`.

`-composescale` *boolean*
: Specifies if the current scale should be composed with the local transform. The default value is `false`.

`-connecteditem` *item*
: Specifies the item relative to which this item is placed. Connected item should be in the same group. The default value is `""`.

`-connectionanchor` *anchor*
: Specifies the anchor on the connected item. The default value is `sw`.

`-fillpattern` *bitmap*
: Specifies the pattern used to draw the text characters, the overstrike and underline lines. The default value is `""`.

`-font` *font*
: Specifies the font for the text. The default value is the current value of the widget option `-font` .

`-overstriked` *boolean*
: If true, a thin line will be drawn horizontally across the text characters. The default value is `false`.

`-position` *point*
: The item's position relative to the anchor (if no connected item specified). The default value is `"0 0"` (Tcl/Tk) or `[0,0]` (Perl/Tk).

`-priority` *priority*
: The absolute position in the stacking order among siblings of the same parent group. The default value is `1`.

`-sensitive` *boolean*
: Specifies if the item should react to events. The default value is `true`.

`-spacing` *dimension*
: Specifies a pixel value that will be added to the inter-line spacing specified in the font. The value can be positive to increase the spacing or negative to reduce it. The default value is `0`.

`-tags` *taglist*
: The list of tags associated with the item. The default value is `""`.

`-text` *string*
: Specifies the text characters. Newline characters can be embedded to force line ends. The default value is `""`.

`-underlined` *boolean*
: If true, a thin line will be drawn under the text characters. The default value is `false`.

`-visible` *boolean*
: Specifies if the item is displayed. The default value is `true`.

`-width` *dimension*
: Specifies the maximum pixel width of the text, a line break will be automatically inserted at the closest character position to match this constraint. If the value is zero, the width is not under the item control and line breaks must be inserted in the text to have multiple lines. The default value is `0`.

6.6 Icon items

Icon items are used for displaying bitmap images. Any bitmap file format supported by Tk can be used. If the bitmap file supports transparency (not alpha-blending, only full transparency), TkZinc will render this transparent area. With and without openGL, icons can be rotated or scaled. However, attributes `-composerotation` and `-composescale` must be set before rotation and scaling.

Applicable attributes for `icon` are:

`-anchor anchor`
: The anchor used in positionning the item. The default value is `nw`.

`-color gradient`
: Specifies the uniform (possibly transparent) fill color used for drawing the bitmap. The first color of a real gradient color will be used. If The icon contains an image, only the transparency of the color is used and defines the alpha transparency of the image when `-render` is set to true. The default value is the current value of the widget option `-forecolor` .

`-composealpha boolean`
: Specifies if the alpha value inherited from the parent group should be composed with the alpha of this item. The default value is `true`.

`-composerotation boolean`
: Specifies if the current rotation should be composed with the local transform. The default value is `false`.

`-composescale boolean`
: Specifies if the current scale should be composed with the local transform. The default value is `false`.

`-connecteditem item`
: Specifies the item relative to which this item is placed. Connected item should be in the same group. The default value is `""`.

`-connectionanchor anchor`
: Specifies the anchor on the connected item. The default value is `sw`.

`-image image`
: Specifies a Tk image that will be displayed by the item. The image may have a mask (depend on the image format) that clip some parts. This option has precedence over the `mask` option if both are specified. The default value is `""`.

`-mask bitmap`
: Specifies a Tk bitmap that will be displayed by the item. The bitmap is filled with the color specified with the `color` option. This option is inactive if an image has been specified with the `image` option. The default value is `""`.

`-position point`
: The item's position relative to the anchor (if no connected item specified). The default value is `"0 0"` (Tcl/Tk) or `[0,0]` (Perl/Tk.

`-priority priority`
: The absolute position in the stacking order among siblings of the same parent group. The default value is `1`.

`-sensitive boolean`
: Specifies if the item should react to events. The default value is `true`.

`-tags taglist`
: The list of tags associated with the item. The default value is `""`.

`-visible boolean`
: Specifies if the item is displayed. The default value is `true`.

6.7 Reticle items

Reticle items are set of concentric circles. The number of circles can be either finite or not. Some periodic circles may be different, they are called bright circles; they can be configured differently from other circles. This item has mainly be designed for radar display images, to help user evaluationg distances from the central point. Reticle cannot handle events.

Applicable attributes for `reticle` are:

`-brightlinecolor gradient`
: This is the uniform (possibly transparent) color of highlighted circles. The first color of a real gradient color will be used. The default value is the current value of the widget option `-forecolor` .

`-brightlinestyle linestyle`
: This is the line style of the highlighted circles. The default value is `simple`.

`-composealpha boolean`
: Specifies if the alpha value inherited from the parent group should be composed with the alpha of this item. The default value is `true`.

`-composerotation boolean`
: Specifies if the current rotation should be composed with the local transform. The default value is `true`.

`-composescale boolean`
: Specifies if the current scale should be composed with the local transform. The default value is `true`.

`-firstradius dimension`
: This is the radius of the innermost circle of the reticle. The default value is `80`.

`-linecolor gradient`
: This is the uniform (possibly transparent) color of regular (not highlighted) circles. The first color of a real gradient color will be used. The default value is the current value of the widget option `-forecolor` .

`-linestyle linestyle`
: This is the line style of the regular (not highlighted) circles. The default value is `simple`.

`-numcircles unsignedint`
: Specifies how many circles should be drawn. The default value is `-1` which means draw as many circles as needed to encompass the current widget window. This does not take into account any possible clipping that can mask part of the reticle. The idea behind this trick is to draw an infinite reticle that is optimized for the current scale.

`-period unsignedint`
: Specifies the recurrence of the bright circles over the regulars. The default value is `5` which means that a bright circle is drawn then 4 regulars, etc.

`-position point`
: Location of the center of the reticle. The default value is `"0 0"`.

`-priority priority`
: The absolute position in the stacking order among siblings of the same parent group. The default value is `0`.

`-sensitive boolean`
: Specifies if the item should react to events. The default value is `false` as the item cannot handle events.

`-stepsize dimension`
: The (scale sensitive) size of the step between two consecutive circles. The default value is `80`.

`-tags taglist`
: The list of tags associated with the item. The default value is `""`.

`-visible boolean`
: Specifies if the item is displayed. The default value is `true`.

6.8 Map items

Map items are typically used for displaying maps on a radar display view. Maps are not be sensitive to mouse or keyboard events, but have been designed to efficiently display large set of points, segments, arcs, and simple texts. A map item is associated to a mapinfo. This mapinfo entity can be either initialized with the videomap command or more generally created and edited with a set of commands described in the The mapinfo related commands section.

Applicable attributes for map are:

-color gradient
: Specifies the uniform (possibly transparent) color used to draw or fill the map. The texts and symbols that are part of the map are also drawn in this color. The first color of a real gradient color will be used. The default value is the current value of the widget option -forecolor .

-composealpha boolean
: Specifies if the alpha value inherited from the parent group should be composed with the alpha of this item. The default value is true.

-composerotation boolean
: Specifies if the current rotation should be composed with the local transform. The default value is true.

-composescale boolean
: Specifies if the current scale should be composed with the local transform. The default value is true.

-filled boolean
: If set to true the map wil be filled otherwise it will be drawn as thin lines. The default is false.

-fillpattern bitmap
: Specifies the pattern to be used when filling the map. The value should be a legal Tk bitmap. The default value is "".

-font font
: Specifies the font that will be used to drawn the texts of the map. The default value is the current value of the widget option -maptextfont .

-mapinfo mapinfo
: Specifies the lines, texts, symbols and other various graphical components that should be displayed by the map item. All these graphical components will share the graphical attributes (color, font, etc) of the item and its coordinate system. The default value is "" which means that nothing will be displayed by the map.

-priority priority
: The absolute position in the stacking order among siblings of the same parent group. The default value is 0.

-sensitive boolean
: Specifies if the item should react to events. The default value is false as the item cannot handle events.

-symbols bitmaplist
: XXX to be detailed. The default value is ??.

-tags taglist
: The list of tags associated with the item. The default value is "".

-visible boolean
: Specifies if the item is displayed. The default value is true.

6.9 Rectangle items

Items of type `rectangle` display a rectangular shape, optionally filled. The rectangle is described by its bottom-left and top right corners.

It is possible to use this item as a clip item for its group. It is also possible to use the rectangle in a `contour` command to build a complex shape in a `curve` item. The two points describing the rectangle can be read and modified with the `coords` command.

Applicable attributes for `rectangle` are:

`-composealpha boolean`
: Specifies if the alpha value inherited from the parent group should be composed with the alpha of this item. The default value is `true`.

`-composerotation boolean`
: Specifies if the current rotation should be composed with the local transform. The default value is `true`.

`-composescale boolean`
: Specifies if the current scale should be composed with the local transform. The default value is `true`.

`-fillcolor gradient`
: Specifies the color that will be used to fill the rectangle if requested by the `-filled` attribute. The default value is a one color gradient based on the current value of the widget option `-forecolor` .

`-filled boolean`
: Specifies if the item should be filled. The default value is `false`.

`-fillpattern bitmap`
: Specifies the pattern to use when filling the item. The default value is `""`.

`-linecolor gradient`
: Specifies the uniform (possibly transparent) color used to draw the item outline. The first color of a real gradient color will be used. The default value is the current value of the widget option `-forecolor` .

`-linepattern bitmap`
: Specifies the pattern to use when drawing the outline. The default value is `""`.

`-linestyle linestyle`
: Specifies the line style to use when drawing the outline. The default value is `simple`.

`-linewidth dimension`
: Specifies the width of the item outline (not scalable). The default value is `1`.

`-priority priority`
: The absolute position in the stacking order among siblings of the same parent group. The default value is `1`.

`-relief relief`
: Specifies the relief used to drawn the rectangle outline. This attribute has priority over the `-linepattern` and `-linestyle` attributes. The color of the relief is derived from the color in `-linecolor` . The default value is `flat`.

`-sensitive boolean`
: Specifies if the item should react to events. The default value is `true`.

`-tags taglist`
: The list of tags associated with the item. The default value is `""`.

`-tile image`
: Specifies an image used for filling the item with tiles. This will be done only if filling is requested by the `-filled` attribute. This attribute has priority over the `-fillcolor` attribute and the `-fillpattern` attribute. The default value is `""`.

`-visible boolean`
: Specifies if the item is displayed. The default value is `true`.

6.10 Arc items

Items of type `arc` display an oval section, optionally filled, delimited by two angles. The oval is described by its enclosing rectangle. The arc can be closed either by a straight line joining its end points or by two segments going throught the center to form a pie-slice.

It is possible to use this item as a clip item for its group, the clip shape will be the polygon obtained by closing the arc. It is also possible to use this polygon in a `contour` command to build a complex shape in a `curve` item. The two points describing the enclosing rectangle can be read and modified with the `coords` command. The first point should be the top left vertex of the rectangle and the second should be the bottom right.

Applicable attributes for `arc` are:

`-closed boolean`
: Specifies if the outline of the arc should be closed. This is only pertinent if the arc extent is less than 360 degrees. The default value is `false`.

`-composealpha boolean`
: Specifies if the alpha value inherited from the parent group should be composed with the alpha of this item. The default value is `true`.

`-composerotation boolean`
: Specifies if the current rotation should be composed with the local transform. The default value is `true`.

`-composescale boolean`
: Specifies if the current scale should be composed with the local transform. The default value is `true`.

`-extent angle`
: Specifies the angular extent of the arc relative to the start angle. The angle is expressed in degrees in the trigonometric system. The default value is `360`.

`-fillcolor gradient`
: Specifies the color used to fill the arc if requested by the `-filled` attribute. The default value is a one color gradient based on the current value of the widget option `-backcolor` .

`-filled boolean`
: Specifies if the item should be filled. The default value is `false`.

`-fillpattern bitmap`
: Specifies the pattern to use when filling the item. The default value is `""`.

`-firstend lineend`
: Describes the arrow shape at the start end of the arc. This attribute is applicable only if the item is not closed and not filled. The default value is `""`.

`-lastend lineend`
: Describes the arrow shape at the extent end of the arc. This attribute is applicable only if the item is not closed and not filled. The default value is `""`.

`-linecolor gradient`
: Specifies the uniform (possibly transparent) color used to draw the item outline. The first color of a real gradient color will be used. The default value is the current value of the widget option `-forecolor` .

`-linepattern bitmap`
: Specifies the pattern to use when drawing the outline. The default value is `""`.

`-linestyle linestyle`
: Specifies the line style to use when drawing the outline. The default value is `simple`.

`-linewidth dimension`
: Specifies the with of the item outline (not scalable). The default value is `1`.

`-pieslice boolean`
: This attribute tells how to draw an arc whose extent is less than 360 degrees. If this attribute is true the arc open end will be drawn as a pie slice otherwise it will be drawn as a chord. The default value is `false`.

`-priority priority`
: The absolute position in the stacking order among siblings of the same parent group. The default value is `1`.

`-sensitive boolean`
: Specifies if the item should react to events. The default value is `true`.

`-startangle angle`
: Specifies the arc starting angle. The angle is expressed in degrees in the trigonometric system. The default value is `0`.

`-tags taglist`
: The list of tags associated with the item. The default value is `""`.

`-tile image`
: Specifies an image used for filling the item with tiles. This will be done only if filling is requested by the `-filled` attribute. This attribute has priority over the `-fillcolor` attribute and the `-fillpattern` attribute. The default value is `""`.

`-visible boolean`
: Specifies if the item is displayed. The default value is `true`.

6.11 Curve items

Items of type `curve` display pathes of line segments and/or cubic bezier connected by their end points. A cubic Bezier is defined by four points. The first and last ones are the extremities of the cubic Bezier. The second and the third ones are control point (i.e. they must have a third "coordinate" with the value 'c'). If both control points are identical, one may be omitted. As a consequence, it is an error to have more than two succcessive control points or to start or finish a curve with a control point.

The polygon delimited by the path can optionally be filled. It is possible to build curve items with more than one path to describe complex shapes with the `contour` command. This command can be used to perform boolean operations between a curve and almost any other item available in TkZinc including another curve. The exact appearance of a multi-contour curve (i.e. which parts are filled and which are holes) depends on the value of an attribute, called `-fillrule` . In the following figure (a snapshot of `zinc-demos`) two curves with four holes each are in front of a text. You can partially see the text through the holes.

It is possible to use this item as a clip item for its group, the clip shape will be the polygon obtained by closing the path. The vertices can be read, modified, added or removed with the `coords` command.

Applicable attributes for `curve` are:

`-capstyle capstyle`
: Specifies the form of the outline ends. This attribute is only applicable if the curve is not closed and the outline relief is flat. The default value is `round`.

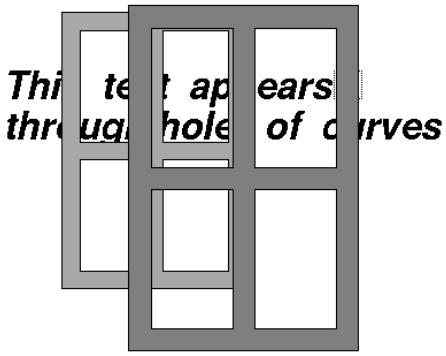

Figure 6.3: Two curves with 4 holes each. A text is visible behind

`-closed` boolean
: Specifies if the curve outline should be drawn between the first and last vertex or not. The default value is `false`.

`-composealpha` boolean
: Specifies if the alpha value inherited from the parent group should be composed with the alpha of this item. The default value is `true`.

`-composerotation` boolean
: Specifies if the current rotation should be composed with the local transform. The default value is `true`.

`-composescale` boolean
: Specifies if the current scale should be composed with the local transform. The default value is `true`.

`-fillcolor` gradient
: Specifies the color used to fill the curve if requested by the `-filled` attribute. The default value is a one color gradient based on the current value of the widget option `-backcolor` .

`-filled` boolean
: Specifies if the item should be filled. The default value is `false`.

`-fillpattern` bitmap
: Specifies the pattern to use when filling the item. The default value is `""`.

`-fillrule` fillrule
: Specifies the way contours are combined together to specify complex surfaces, with holes and disjoint surfaces. The default value is `"odd"`. This means that a point of the space is considered inside the curve surface if an odd number of contours are surrounding the point. This attribute should only be modified for curves with multiple or complicated contours.

`-firstend` lineend
: Describes the arrow shape at the start of the curve. This attribute is applicable only if the item is not closed, not filled and the relief of the outline is flat. The default value is `""`.

`-joinstyle` joinstyle

Specifies the form of the joint between the curve segments. This attribute is only applicable if the curve outline relief is flat. The default value is `round`.

`-lastend lineend`

Describes the arrow shape at the end of the curve. This attribute is applicable only if the item is not closed, not filled and the relief of the outline is flat. The default value is `""`.

`-linecolor gradient`

Specifies the uniform (possibly transparent) color used to draw the item outline. The first color of a real gradient color will be used. The default value is the current value of the widget option `-forecolor` .

`-linepattern bitmap`

Specifies the pattern to use when drawing the outline. The default value is `""`.

`-linestyle linestyle`

Specifies the line style to use when drawing the outline. The default value is `simple`.

`-linewidth dimension`

Specifies the with of the item outline (not scalable). The default value is `1`.

`-marker bitmap`

Specifies a bitmap that will be used to draw a mark at each vertex of the curve. This attribute is not applicable if the outline relief is not flat. The default value is `""` which means do not draw markers.

`-markercolor gradient`

Specifies the uniform (possibly transparent) color of the markers. The first color of a real gradient color will be used. The default value is the current value of the widget option `-forecolor` .

`-priority priority`

The absolute position in the stacking order among siblings of the same parent group. The default value is `1`.

`-relief relief`

Specifies the relief used to drawn the curve outline. This attribute has priority over the `-linepattern` and `-linestyle` attributes. The color of the relief is derived from the color in `-linecolor` . The default value is `flat`.

`-sensitive boolean`

Specifies if the item should react to events. The default value is `true`.

`-smoothrelief boolean`

Specifies if the relief should be smoothed along the curve. This is useful to obtain smooth curved reliefs instead of facets The default value is `false`.

`-tags taglist`

The list of tags associated with the item. The default value is `""`.

`-tile image`

Specifies an image used for filling the item with tiles. This will be done only if filling is requested by the `-filled` attribute. This attribute has priority over the `-fillcolor` attribute and the `-fillpattern` attribute. The default value is `""`.

`-visible boolean`

Specifies if the item is displayed. The default value is `true`.

6.12 Triangles items

Triangles items are used for displaying complexe surfaces with variables colors and transparencies. For example, it can be used to create a circular color selector displaying a range of colors. The way triangles composing a triangle item are arranged is defined by the `-fan` option.

This item has been added to provide access to a basic openGL geometric construction but it is also available in the X environment albeit with less features.

Applicable attributes for `triangles` are:

`-colors gradientlist`
: Specifies the colors of each vertex of the triangles.

`-composealpha boolean`
: Specifies if the alpha value inherited from the parent group should be composed with the alpha of this item. The default value is `true`.

`-composerotation boolean`
: Specifies if the current rotation should be composed with the local transform. The default value is `true`.

`-composescale boolean`
: Specifies if the current scale should be composed with the local transform. The default value is `true`.

`-fan boolean`
: If true, triangles are created with a fan like layout. Otherwise triangles are arranged like a strip. The default value is `true`.

`-priority priority`
: The absolute position in the stacking order among siblings of the same parent group. The default value is `1`.

`-sensitive boolean`
: Specifies if the item should react to events. The default value is `true`.

`-tags taglist`
: The list of tags associated with the item. The default value is `""`.

`-visible boolean`
: Specifies if the item is displayed. The default value is `true`.

6.13 Window items

Items of type `window` display an X11 window at a given position in the widget.

It is possible to use this item as a clip item for its group, the clip shape will be the window rectangle. It is also possible to use the rectangular shape of the window item in a `contour` command to build a complex shape in a `curve` item. The position of the window, relative to the anchor, can be set or read with the `coords` command (i.e. if no connected item is specified).

One of the most frequent use of this item is to embed any Tk widget into TkZinc, including, of course, another TkZinc instance. Another less obvious use is to embed a whole Tk application into TkZinc, here is how to do it: The embedding application should create a frame with the `-container` option set to true; Add a window item to the relevant TkZinc widget with the `-window` attribute set to the id of the container frame; The embedded application should create its toplevel with the `-use` option set to the id of the container frame; Or, as an alternative, the embedded `wish` can be launched with the `-use` option set to the container frame id.

Applicable attributes for `window` items are:

`-anchor anchor`

The anchor used in positionning the item. The default value is `nw`.

`-composealpha boolean`
Specifies if the alpha value inherited from the parent group should be composed with the alpha of this item. The default value is `true`.

`-composerotation boolean`
Specifies if the current rotation should be composed with the local transform. The default value is `true`.

`-composescale boolean`
Specifies if the current scale should be composed with the local transform. The default value is `true`.

`-connecteditem item`
Specifies the item relative to which this item is placed. Connected item should be in the same group. The default value is `""`.

`-connectionanchor anchor`
Specifies the anchor on the connected item used for the placement. The default value is `sw`.

`-height dimension`
Specifies the height of the item window in screen units. The default value is `0`.

`-position point`
The item's position relative to the anchor (if no connected item specified). The default value is `"0 0"` (Tcl/Tk) or `[0,0]` (Perl/Tk).

`-priority priority`
Constraints of the underlying window system dictate the stacking order of window items. They can't be lowered under the other items. Additionally, to manipulate their stacking order, you must use the raise and lower Tk commands on the associated Tk window. The value of this attribute is meaningless.

`-sensitive boolean`
This option has no effect on window items. The default value is `false`.

`-tags taglist`
The list of tags associated with the item. The default value is `""`.

`-visible boolean`
Specifies if the item is displayed. The default value is `true`.

`-width dimension`
Specifies the width of the item window in screen units. The default value is `0`.

`-window window`
Specifies the X id of the window that is displayed by the item. This id can be obtained by the Tk command `winfo id widgetname`. The default value is `""`.

Chapter 7

Labels, label formats and fields

TkZinc was initially developed for building interactive radar image working on X server. This requires very good performances, for displaying many hundred tracks and moving them every few second. Tracks are typically composed of some geometric parts and some textual parts. These two parts are connected together with a leader. The geometric parts are subject to scaling. For example the speed vector length in pixel depends on the scale. But the textual part must not be zoomed. Managing parts which are scaled and others which are not, can be a real challenge. Usual toolkits or widget are not suited to such behaviours, but TkZinc is.

To be able to manage many items mixing geometric parts and non-geometric parts, TkZinc introduces the concepts of label, labelformat, fields and fields attributes.

7.1 Labels and labelformats

A label is a set of rectangular fields attached to the following item types : `track` , `waypoint` and `tabular` . The fields of a label may contain either text or bitmaps or images. A label cannot be identified or manipulated by itself; There is no function nor method to get or manipulate a label as an object or an item. A label is always associated to an item and is manipulated through this item.

Some label global characteristics are set/get at the item level:

- The maximum number of fields is defined at item creation, as the second argument of the `add` method. The field number can not be changed after creation. These fields will be indexed from 0 to n-1. The number of fields can be read with the command `numparts` . For example:

```
$track = $zinc->add('track',1, 4, ....);
# this creates a track item in root group with 4 fields, indexed from 0 to 3
```

- The rectangular geometries of displayable fields are defined through the item attribut `-labelformat`. The value is a string following the syntax of the `labelformat` type. This attribute can be set at any time; thus modifying its value is a way to quickly change the geometry (or the visibility) of some fields. Fields may overlap. They are drawn according to the index order: field 0 is drawn before (thus under) field 1. The labelformat also optionnaly describes a clipping rectangle. For example:

```
$zinc->itemconfigure($track, -labelformat => 'a12a0+0+ x20x100>0 a2a0>1>0');
```

```
                                        ##       ^        ^         ^
                                        ##       field0   field1    field2
    # the labelformat indicates that only the first 3 fields will be displayed:
    #  field 0 expands for the size of the text + 12 pixels.
    #       It starts at the top left point
    #  field 1 has a size of 20x10 pixels.
    #       It is left aligned with field 0, just under field 0
    #  field 2 expands for the size of the text + 2 pixels.
    #       It is adjacent to the right of field 1, just under field 0
```

Characteristics of each individual field are called field attributes. They are all described in next section Attributes for fields . They may be set or get with the itemcget and itemconfigure command. These commands require as a second argument the field number. By configuring field attributes you can modify :

- the field content : -text , -image , -tile , -fillpattern ,
- the field colors : -backcolor , -bordercolor , -color ,
- the text general appearance : -alignment , -autoalignment , -font ,
- the field border : -border , -relief , -reliefthickness ,
- and the field visibility and sensitivity: -sensitive , -visible .

As an example:

```
$zinc->itemconfigure($track, 0, -text => 'Hello World',
                                -color => 'white',
                                -backcolor => 'black',
                                -filled => 1);
# this should display "Hello World" in white on black in field 0
```

It is possible to bind callbacks to fields, with the command bind and special tags (described Tags and bindings). As an example:

```
$zinc->bind("$track:1", '<1>', \&acallback);
# this binds &acallback to field 1
```

Inside a callback function, it is possible to know which field the mouse cursor is over with the command currentpart .

A Perl module, called Tk::Zinc::Text (see the section Tk::Zinc::Text) is provided for easing text input in text fields (it can also be used for easing text input in text item).

7.2 Attributes for fields

Fields are item parts of items supporting labelformat (i.e. track , waypoint and tabular). They can be configured in a similar way of items themselves, with the command itemconfigure , but this command requires an additionnal parameter (in second position) the fieldId. To get the value

of a field attribute, you can use the command itemcget with the `fieldId` as an additionnal second parameter.
NB: Field attributes cannot be configured at item creation with the command `add` .

Applicable attributes for fields are:

`-alignment alignment`
: The horizontal alignment of both the text and the image. The default value is `left`.

`-autoalignment autoalignment`
: The dynamic horizontal alignments used depending on the label orientation. The default value is `"-"` which means do not use dynamic alignment.

`-backcolor gradient`
: The field background color. The default value is the current value of the widget option `-backcolor` .

`-border edgelist`
: The border description edge by edge. The border is a one pixel wide outline that is drawn around the field outside the relief. Some border edges can be omitted, this attribute describes the edges that should be displayed as part of the border. The default value is `""`.

`-bordercolor gradient`
: The border uniform (possibly transparent) color. The first color of a real gradient color will be used. The default value is the current value of the widget option `-forecolor` .

`-color gradient`
: The text uniform (possibly transparent) color. The first color of a real gradient color will be used. The default value is the current value of the widget option `-forecolor` .

`-filled boolean`
: Specifies if the field background should be filled. The default value is `false`.

`-fillpattern bitmap`
: The fill pattern used when filling the background. This attribute is overrided by the tile attribute. The default value is `""`.

`-font font`
: The text font. The default value is the current value of the widget option `-font` .

`-image image`
: An image to be displayed in the field. The image will be centered vertically in the field. The default value is `""`.

`-relief relief`
: Specifies the relief to be drawn around the field, inside the border. The color of the relief is derived from the color in `-backcolor` . The default value is `flat`.

`-reliefthickness dimension`
: Width of the relief drawn around the field. The default value is `0` which means that no relief should be drawn around the field.

`-sensitive boolean`
: Specifies if the field should react to input events. The default value is `true`.

`-text string`
: A line of text to be displayed in the field. The text will be centered vertically in the field. The default value is `""`.

`-tile image`
: Specifies an image that will be tiled over the field background is the field is filled. This attribute has precedence over the `-fillpattern` attribute. The default value is `""`.

`-visible boolean`
: Specifies if the field is displayed. The default value is `true`.

Chapter 8

Attribute types

We describe in this chapter all the available types in TkZinc. They are listed by alphabetical order.

NB: Two types are very important and their existence should be known by any new user of TkZinc: `gradient` *and* `labelformat` .

alignment

Specifies the horizontal alignment of an entity. The legal values are: `left`, `right`, `center`.

alpha

Specifies the transparency of an item. The value must be an integer from 0 (fully transparent) to 100 (fully opaque).

anchor

Specifies one of the nine characteristic points of a rectangle or a bounding box that will be used to position the object. These points include the four corners, the four edge centers, and the center of the rectangle. The possible values are: `nw`, `n`, `ne`, `e`, `se`, `s`, `sw`, `w`, `center`.

angle

Specifies an angle in degrees, the value must be an integer from 0 to 360 inclusive.

autoalignment

Specifies the horizontal alignments that should be used for track or way point fields depending on the label position relative to the position of the item. The attribute may have two forms: a single dash `-` means turning of the automatic alignment feature for the field; The other form consists in three letters which describe in order: the alignment to be used when the label is to the left of the item position, above or below the item position and to the right of the item position. The possible values for each letter is: `l` for left alignment, `c` for center alignment and `r` for right alignment. Here is an example: `rll` means right align the field if the label is on the left side of the item, and left align if the label is above, below or on the right of the item.

bitmap

This should be a string naming a valid Tk bitmap. The bitmap should be known to Tk prior to its use. TkZinc registers a set of bitmaps that can be used for any bitmap valued attribute (see section `Bitmaps`). Extensions to Tk are available to create or manipulate bitmaps from a script. The value may also name a file containing a valid X11 bitmap description. The syntax in this case is `@filename`.

bitmaplist

This is an extension of the `bitmap` attribute type. It describes a list of bitmaps that will be the value of the attribute.

boolean

This is the description of a standard Tcl boolean value. The possible values are `0`, `false`, `no` or `off` for the false value and `1`, `true`, `yes` or `on` for the true value.

capstyle

This the description of a line cap. The possible values are `butt`, `projecting` and `round`.

color

This is a string that describes a color. The description may have one of two forms, a colorname such as `green` or `LemonChiffon` or an rgb specification in one of the following formats, `#rgb`, `#rrggbb`, `#rrrgggbbb` or `#rrrrggggbbbb`. If less than four digits are provided for a color component, they represent the most significant bits of the component. For example `#3a7` is equivalent to `#3000a0007000`.

dimension

This is a string that represent distance. The string consists in a floating point signed number.

edgelist

This is a list describing the edges of a border that should be considered for processing (e.g for drawing). The possible values are `left`, `right`, `top`, `bottom`, `contour`, `oblique` and `counteroblique`. The `contour` value is the same as the `"left top right bottom"` list. The `oblique` and `counteroblique` values describe diagonal segments from top-left to bottom-right and from top-right to bottom-left respectively. The following picture gives some edges examples.

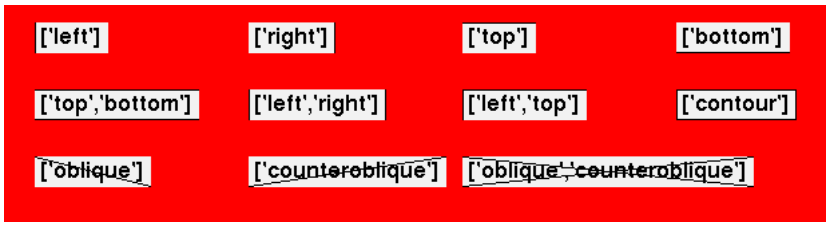

Figure 8.1: edgelist examples

fillrule

This is a string describing the rule used to compose the different contours or pathes of a curve. The allowed values are directly inspired from the openGL GLU tesselators as described for example in the chapter 11 of the "The OpenGL Programming Guide 3rd Edition The Official Guide to Learning OpenGL Version 1.2", ISBN 0201604582. You can also refer to the example fillrule provided with TkZinc in `zinc-demos` . The allowed values are `odd`, `nonzero`, `positive`, `negative`, and `abs_geq_2`. The following figure shows the effect of fillrule value on curves with multiple contours:

font

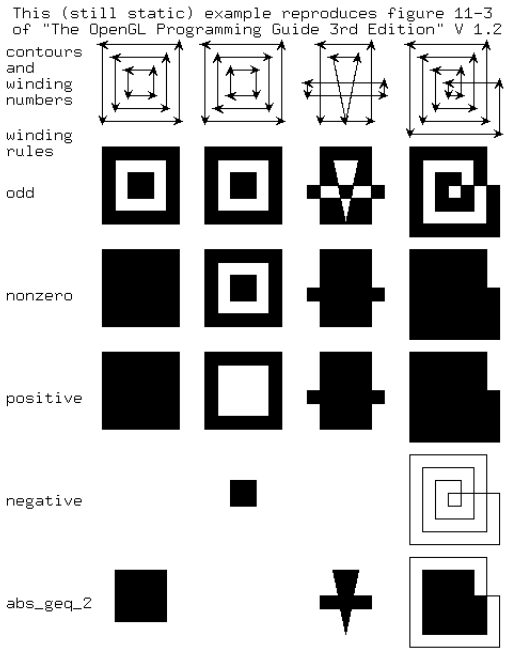

Figure 8.2: Examples of fillrule on curves

This is a string describing a font. For an exhaustive description of what is legal as a font description, refer to the Tk `font` command man page. Just to mention to popular methods, it is possible to specify a font by it's X11 font name or by a list whose elements are the font family, the font size and then zero or more styles including `normal`, `bold`, `roman`, `italic`, `underline`, `overstrike`.

Please note, that some font data are cached by TkZinc, on the application level. This is specially usefull with openGL. To avoid breaking the cache mecanism, you should avoid using a font once on only one item, then modifiy this item font and repeat this again and again.

gradient

This is a string describing a color gradient to be used for example to fill a surface. Gradient are also used to describe color of lines, even if in this case the lines are limited to one color with and optionnal alpha percentage.

The string may consist in a single color specification that will be used to paint a solid surface or a color with an alpha value or a list of gradient steps separated by `|` characters.

- The general pattern for an axial gradient is :
  `"=axial degre | gradient_step1 | ... | gradient_stepn"` or
  `"=axial x1 y1 x2 y2 | gradient_step1 | ... | gradient_stepn"`
  The `degre` parameter defines the angle of the axe in the usual trigonometric sense. It defaults to 0. The `x1 y1 x2 y2` parameters define both the angle and the extension of the axe.
- The general pattern for a radial gradient is :
  `"=radial x y | gradient_step1 | ... | gradient_stepn"` or
  `"=radial x1 y1 x2 y2 | gradient_step1 | ... | gradient_stepn"`

The `x y` parameters define the center of the radial. The `x1 y1 x2 y2` parameters define both the center and the extension of the radial.

- The general pattern for a path gradient is :
  `"=path x y | gradient_step1 | ... | gradient_stepn"`
  The `x y` parameters define the center of the gradient.
- The general pattern for a conical gradient is :
  `"=conical degre | gradient_step1 | ... | gradient_stepn"` or
  `"=conical degre x y | gradient_step1 | ... | gradient_stepn"` or
  `"=conical x1 y1 x2 y2 | gradient_step1 | ... | gradient_stepn"`
  The `degre` parameter defines the angle of the cone in the usual trigonometric sense. The optional `x y` parameters define the center of the cone. By default, it is the center of the bounding-box. The `x1 y1 x2 y2` parameters define the center and the angle of the cone.

All x and y coordinates are expressed in percentage of the bounding box, relatively to the center of the bounding box. So `0 0` means the center while `-50 -50` means the lower left corner of the bounding box.

If none of the above gradient type specification is given, the gradient will be drawn as an axial gradient with a null angle.

Each gradient segment section has the general form:

`color;alpha color_position mid_span_position`

Each color can be specified as a valid X color : either a named color or #rrggbb value or any valid X color specification such as standard device-independent color specification (e.g. `CIEuvY:<u>/<v>/<Y>` as defined in the X man page). An alpha value can be applied to the color using the optional `;alpha` parameter whose value should be in the 0..100 intervalle.

The color position tells where in the gradient surface, measured as a percentage of the total gradient distance, the color should start. The first gradient segment has its position set to 0 and the last segment has its position set to 100, regardless of the specification. The position can thus be safely omitted for these segments. The in between segments must have a position explicitly set. If not given, the position will default to 0.

The mid span position tells where in the current gradient segment should be the median color. The position is given in percentage of the current gradient segment distance. The mid span position can be used to obtain a non linear gradient segment, this is useful to describe relief shapes. This parameter can be omitted in which case it defaults to 50 and the gradient segment is perfectly linear.

A gradient segment can be specified as a single color. In this case a flat uniform fill will result.

The following picture gives many examples of gradients. They correspond to the following values:

```
axial 1 :  '=axial 0 | black|white'  :=  'black|white'
axial 2 :  '=axial 90 | black|white'
axial 3 :  '=axial 30 |black|white'
axial 4 :  '=axial 30|black|black;0'
```

```
radial 1 : '=radial -14 -20|white|black'
radial 2 : '=radial 0 0 | white;50 0 70|black 50|white 100'
path 1 :   '=path -14 -20|white|black;80'
path 2 :   '=path -14 -20 |white|white 30|black;80'
```

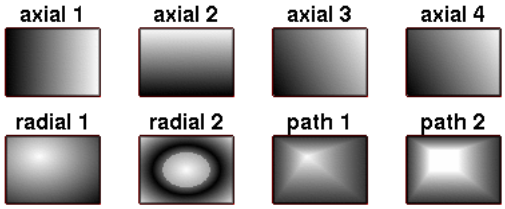

Figure 8.3: Examples of axial, radial and path gradients

gradientlist

This is an extension of the `gradient` attribute type. It describes a list of gradients that will be the value of the attribute.

image

This should be the name of a previously registered Tk image.

In pure Tcl-Tk only GIF, PPM and bitmap formats are available as source for images. With the Img extension many others popular formats are added including JPEG, XPM and PNG.

In Perl/Tk most image formats can be used, specially with Tk::JPEG or Tk::PNG modules.

Please note, that some image data are cached by TkZinc, on the application level. This is specially usefull with openGL. To avoid breaking the cache mecanism, you should avoid using an image once on only one item option, then modifiy this item option and repeat this again and again.

item

Describes an item id or a tag. If a tag is provided an item will be searched for the tag and the first matching in display list order will be used.

joinstyle

Describes a join style. The possible values are `bevel`, `miter` and `round`.

labelformat

The format is as follow. Parameters between `[]` are optional and take default values when omitted. Spaces can appear between blocks but not inside.

```
[WidthxHeight] [<field0Spec>] [<field1Spec>] ... [<fieldnSpec>]
```

`Width` and `Height` are strictly positive integers. They set the size of the clipping box surrounding the label. If not specified, there will be no clipping. If specified alone, they specify the size of the only displayed field which index is 0.

```
<fieldiSpec> ::= <fieldiSize>[<fieldiPos>]
```

Each fieldiSpec specify the size and position of the field numbered i.

`<fieldiSize> ::= <sChar><fieldWidth><sChar><fieldHeight>`

`<sChar> ::= x|f|i|a|l`

`<sChar>` specifies the meaning of the following `<fieldWidth>` or `<fieldHeight>`. Those are positive integers. Values for `<sChar>` have the following meaning :

- `'x'` : the corresponding dimension (either width or height) is in pixel, according to the value of the `<fieldWidth>` or `<fieldHeight>`
- `'f'` : the corresponding dimension is in percentage of the mean width/height of a character (in the field font). The following `<fieldWidth>` or `<fieldHeight>` gives the percentage. The value must be an integer between 0 and 100.
- `'i'` : the corresponding dimension is in percentage of the size of the image in the field. The following `<fieldWidth>` or `<fieldHeight>` gives the percentage. The value must be an integer between 0 and 100. If the field contains no image, the dimension is 0.
- `'a'` : the corresponding dimension is automatically adjusted to match the field's content plus the given value in pixels.
- `'l'` : the corresponding dimension is adjusted to match the global size of the label (not counting fields with `'l'` size specs). The corresponding integer parameter is not used with this size specification. The global size of the label is considered when the labelformat is set. If some fields sizes change afterwards, you should set again the labelformat so that fields using a `'l'` specification are re-computed. It is not possible to reference the field in another `<fieldiPos>` (see below).

`<fieldiPos> ::= <pChar><fieldX><pChar><fieldY>.`

`<pChar> ::= +|<|>|^|$`

`<fieldX>` and `<fieldY>` are either integer or index refering an other field of the labelformat.

Values for `pChar` have the following meaning :

- `'+'` : the position, either on the X or Y axis, is in pixel, possibly negative. XXX what does it mean if negative? The value is given by the corresponding `<fieldX>` or `<fieldY>`.
- `'<'` : The field will be at the left (or top) of the field refered by the corresponding index `<fieldX>` (or `<fieldY>`)
- `'>'` : The field will be at the right (or bottom) of the field refered by the corresponding index `<fieldX>` (or `<fieldY>`)
- `'^'` : The field will be left (or top) aligned with the field refered by the corresponding index `<fieldX>` (or `<fieldY>`).
- `'$'` : The field will be right (or bottom) aligned with the field refered by the corresponding index `<fieldX>` (or `<fieldY>`).

`<fieldiPos>` can be omitted if there is only one field.

leaderanchors

Describes where to attach the label leader on the label. Two positions can be defined: one when the label is at the right of current position and the other when the label is at the left of current position. **Not to be confused with the regular rectangular anchors**.

The format is: `lChar leftLeaderAnchor [lChar rightLeaderAnchor]`

If `lChar` is a `|`, `leftLeaderAnchor` and `rightLeaderAnchor` are the indices of the field that serve to anchor the label's leader. More specifically the bottom right corner is used when `leftLeaderAnchor` is active and the bottom left corner is used when `rightLeaderAnchor` is active.

If `lChar` is `%`, `leftLeaderAnchor` and `rightLeaderAnchor` should be specified as `widthPercentxheight` each value being a percentage (between 1 and 100) of the width or height of the label bounding box. If `rightLeaderAnchor` is not specified it defaults to `leftLeaderAnchor`. If neither are specified, the center of the label is used as an anchor.

lineend

Describes the shape of the arrow at the beginning or end of a path. This is a list of three numbers describing the arrow shape in the following order: distance along the axis from neck to tip of the arrowhead, distance from trailing points to tip and distance from outside edge of the line to the trailing points (see canvas). If an empty list is given, there is no arrow.

lineshape

Describes the shape of a path connecting two points. The possible values are `straight`, `rightlightning`, `leftlightning`, `rightcorner`, `leftcorner`, `doublerightcorner` and `doubleleftcorner`. The following figure shows these different line shapes:

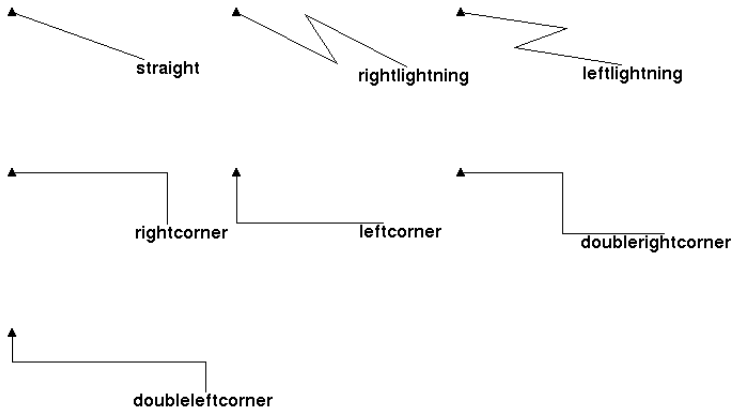

Figure 8.4: Examples of all available line shapes

linestyle

Describes the style of the dashes that should be used to draw a line. The possible values are `simple`, `dashed`, `mixed` and `dotted`.

mapinfo

This is the name of a previously registered mapinfo object (see the chapter The mapinfo related commands) that will define the lines, arcs, symbols, and texts displayed in a map item.

point

This is a list of two floating point values that describes a point position or some two dimensional delta (used for example to describe the speed vector of a track item).

priority

A strictly positive integer value for the display priority.

relief

Describes a border relief. The possible values, illustrated in the following figure are `flat`, `raised`, `sunken`, `ridge`, `groove`, `roundraised`, `roundsunken`, `roundridge`, `roundgroove`, `raisedrule`, `sunkenrule`.

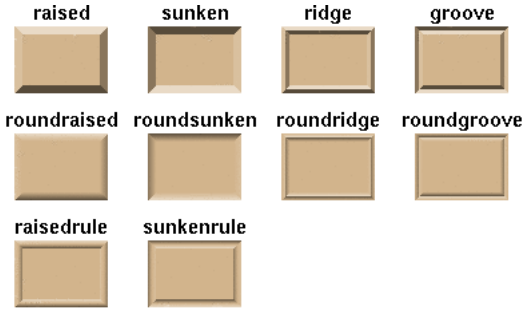

Figure 8.5: Examples of all available non-flat reliefs

string

Just what its name implies, a string.

taglist

This should be a list of strings describing the tags that are set for an item.

unsignedint

Describes an unsigned integer value.

window

A string describing an X window id. This id can be returned by the `winfo id a-widget-path` command.

Chapter 9

The mapinfo related commands

MapInfo objects are used to describe graphical primitives that will be displayed in map items. It is possible to describe lines, arcs, symbols and texts as part of a MapInfo. The `mapinfo` and `videomap` commands are provided to create and manipulate the mapinfo objects.

9.1 The mapinfo command

`mapinfo name` **create**
`$mainwindow->mapinfo(name,` **create,** `)`

Create a new empty map description. The new mapinfo object named `name`.

`mapinfo mapInfoName` **delete**
`$mainwindow->mapinfo(mapInfoName,` **delete,** `)`

Delete the mapinfo object named by `mapInfoName`. All maps that refer to the deleted mapinfo are updated to reflect the change.

`mapinfo mapInfoName` **duplicate** `newName`
`$mainwindow->mapinfo(mapInfoName,` **duplicate,** `newName)`

Create a new mapinfo that is a exact copy of the mapinfo named `mapInfoName`. The new mapinfo object will be named `newName`.

`mapinfo name` **add** `type args`
`$mainwindow->mapinfo(name,` **add,** `type args)`

Add a new graphical element to the mapinfo object named by `name`. The `type` parameter select which element should be added while the `args` arguments provide some type specific values such as coordinates. Here is a description of recognized types and their associated parameters.

line
: This element describes a line segment. Its parameters consists in a line style (`simple`, `dashed`, `dotted`, `mixed`, `marked`), an integer value setting the line width in pixels and four integer values setting the X and Y coordinates of the two end vertices.

arc
: This element describes an arc segment. Its parameters consists in a line style (`simple`, `dashed`, `dotted`, `mixed`, `marked`), an integer value setting the line width in pixels,

two integer values setting the X and Y of the arc center, integer value setting the arc radius and two integer values setting the start angle (in degree) and the angular extent of the arc (in degree).

symbol
: This element describes a symbol. Its parameters consists in two integer values setting the X and Y of the symbol position and an integer setting the symbol index in the `-symbols` list of the map item.

text
: This element describes a line of text. Its parameters consists in a text style (`normal`, `underlined`), a line style (`simple`, `dashed`, `dotted`, `mixed`, `marked`) to be used for the underline, two integer values setting the X and Y of the text position and a string describing the text.

`mapinfo name` **count** `type`
`$mainwindow->mapinfo(name,` **count**`, type)`

Return an integer value that is the number of elements matching `type` in the mapinfo named `name`. `type` may be one the legal element types as described in the `mapinfo add` command.

`mapinfo name` **get** `type index`
`$mainwindow->mapinfo(name,` **get**`, type index)`

Return the parameters of the element at `index` with type `type` in the mapinfo named `name`. The returned value is a list. The exact number of parameters in the list and their meaning depend on `type` and is accurately described in `mapinfo add`. `type` may be one the legal element types as described in the `mapinfo add` command. Indices are zero based and elements are listed by type.

`mapinfo name` **replace** `type index args`
`$mainwindow->mapinfo(name,` **replace**`, type index args)`

Replace all parameters for the element at `index` with type `type` in the mapinfo named `name`. The exact number and content for `args` depend on `type` and is accurately described in `mapinfo add`. `type` may be one the legal element types as described in the `mapinfo add` command. Indices are zero based and elements are listed by type.

`mapinfo name` **remove** `type index`
`$mainwindow->mapinfo(name,` **remove**`, type index)`

Remove the element at `index` with type `type` in the mapinfo named `name`. `type` may be one the legal element types as described in the `mapinfo add` command. Indices are zero based and elements are listed by type.

`mapinfo name` **scale** `factor`
`$mainwindow->mapinfo(name,` **scale**`, factor)`

Scale all coordinates of all the elements described in the mapinfo named `name` by `factor`. The same value is used for X and Y axes.

`mapinfo name` **translate** `xAmount yAmount`
`$mainwindow->mapinfo(name,` **translate**`, xAmount yAmount)`

Translate all coordinates of all the elements described in the mapinfo named `name`. The `xAmount` value is used for the X axis and the `yAmount` value is used for the Y axis.

9.2 The videomap command

This section describes the videomap command, used to create a mapinfo from a proprietary file format for simple maps, in use in french Air Traffic Control Centres. The format is the binary cautra4 (with x and y in 1/8nm units)

`videomap` **ids** `fileName`

Return all sub-map ids that are described in the videomap file described by `fileName`. The ids are listed in file order. This command makes possible to iterate through a videomap file one sub-map at a time, to know how much sub-maps are there and to sort them according to their ids.

`videomap` **load** `fileName index mapInfoName`

Load the videomap sub-map located at position `index` in the file named `fileName` into a mapinfo object named `mapInfoName`. It is possible, if needed, to use the `videomap ids` command to help translate a sub-map id into a sub-map file index.

Chapter 10

Other resources provided by the widget

In this chapter we describe resources included in TkZinc widget. This include bitmaps sets (used as symbols for some items or used as stipples), Perl modules goodies and TkZinc simple demonstrations.

10.1 Bitmaps

TkZinc creates two sets of bitmaps.

The first set contains symbols for ATC tracks position, waypoints position and maps symbols. These bitmaps are named AtcSymbol1 to AtcSymbol22.

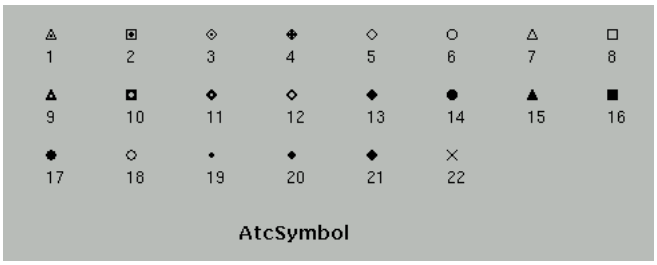

Figure 10.1: Bitmaps available for position of tracks, waypoints, and maps

The second set provides stipples that can be used to implement transparency, they are named AlphaStipple0 to AlphaStipple15, AlphaStipple0 being the most transparent.

10.2 Tk::Zinc::Debug Perl module

`Tk::Zinc::Debug.pm` is a Perl module useful for debugging purpose. It can be used in a Perl application using TkZinc to display the hierarchical tree of items, to display items selected by their id or tags, to grab items with the mouse and to get the list of items enclosed or overlapped by a rectangle designated by the mouse. You will be presented a list of items, with many interesting attributes such as position, priority, visibility, group... and even more information on request. Much of the selected items attributes can be interactively modified. When an application uses `Tk::Zinc::Debug.pm`,

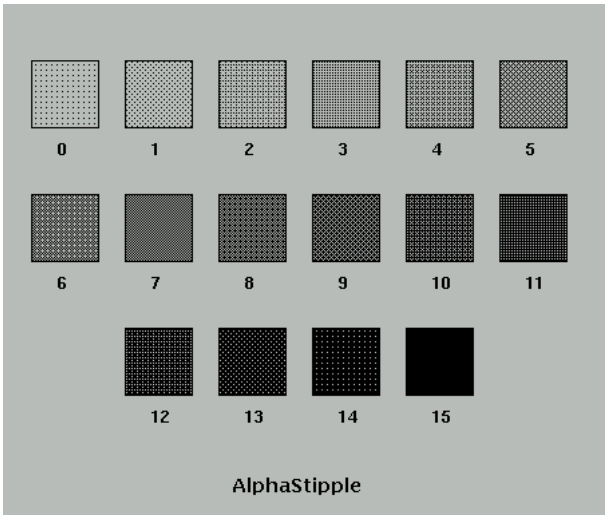

Figure 10.2: Bitmaps available for creating stipples

you can get a short reminder by depressing the `Esc` key in the main window of this application. For more information, please refer to the `Tk::Zinc::Debug.pm` man pages with the classical command `man Tk::Zinc::Debug`

To use this module, you can import it either by adding, for example, the following statements in your source code:

```
use Tk::Zinc::Debug;
finditems($zinc);
tree($zinc, -optionsToDisplay => '-tags', -optionsFormat => 'row');
```

or simply by using the -M option of Perl:

```
perl -MTk::Zinc::Debug yourscript.pl
```

10.3 Tracing TkZinc methods call in Perl/Tk

TkZinc package includes two tools for helping you debugging your Perl/Tk scripts or tracking some nasty segfault which should never occure since TkZinc is (almost) totally bugfree.

10.3.1 Tracking Perl/Tk script errors

Because you sometime get some errors inside `TkZinc` with a cryptic message like `"....  errors in Tk.pm line 228"`, it may be usefull to know where exactly in your code is the error. There is a simple and convenient mean to do this, just by using a small module called `Tk::Zinc::TraceErrors`, released with `TkZinc`. It traces every call of a TkZinc method inducing a Tk error. It prints on the standard output the following informations:

- the filename where the method has been invoked
- the line number in the source file
- the TkZinc method name

- the list of arguments in a human-readable form
- the error message

To use this module you can import it either by adding the following statement in your source code:
`use Tk::Zinc::TraceErrors;`
or better, by using the -M option of Perl:
`perl -MTk::Zinc::TraceErrors yourscript.pl`

10.3.2 Tracking TkZinc segfaults in Perl/Tk

If you encounters a segfault in one Perl/Tk script and you suspects that TkZinc might be responsible, you should use a small module called `Tk::Zinc::Trace`, released with `TkZinc`. It traces every call of a TkZinc method. The method call is printed on the standard output before the effective call, and the result of the invokation is printed after the call. To be sure to identify a segfault at the proper time, it forces an update of TkZinc widget. Thus, this might slow down your script, but should dramatically speed up the identification of the call which makes TkZinc segfaulting. It prints on the standard output the following informations:

- the filename where the method has been invoked
- the line number in the source file
- the TkZinc method name
- the list of arguments in a human-readable form
- the returned value

To use this module you can import it either by adding the following statement in your source code:
`use Tk::Zinc::Trace;`
or better, by using the -M option of Perl:
`perl -MTk::Zinc::Trace yourscript.pl`

10.4 zinc-demos

Starting at version 3.2.4 of TkZinc small applications are included as demos. They are all accessible through an application called `zinc-demos`. These numerous (about 30) tiny demos are useful for newcomers and as starting points for developing real applications. They consists in toy applications, graphically advanced examples or even a TkZinc port of `TkTetris` from Slaven Rezic.

10.5 Tk::Zinc::Graphics Perl module

The Tk::Zinc::Graphics Perl module implements many high level functions for building high quality graphic objects with TkZinc.

Please read the man page for more details: `man Tk::Zinc::Graphics`, french version only currently. Any volontear to translate it in English?

NB: There is also a tcl version of this module.

10.6 Tk::Zinc::Text Perl module

The Tk::Zinc::Text Perl module implements bindings for text input 'a la emacs'. It works for text item or for text fields of track, waypoint or tabular items. The item which requires text input must just be tagged with the 'text' tag.

Please read the man page for more details: `man Tk::Zinc::Text`

10.7 C api for adding new items

The C function AddItemClass provided with the source code of TkZinc, can be used to extent the default set of items in TkZinc in an additionnal dynamic library. The AddItemClass C function is extensively used for implementing the core item set. So please refer to the source code for examples or send email for more information on precise problems.

We will try to further document this feature in the future.

List of Figures

Index