EUROPEAN SOUTHERN OBSERVATORY

Organisation Européenne pour des Recherches Astronomiques dans l'Hémisphère Austral

Europäische Organisation für astronomische Forschung in der südlichen Hemisphäre

VERY LARGE TELESCOPE

Data Management Division

The ESO SkyCat Tool

Astronomical Image and Catalog Browser

Programmer's Manual

Doc.No. VLT-MAN-ESO-19400-1552

Issue 1.0, Version 2.0.4

Date 19/01/98

	Name	Date	Signature
Prepared	A. Brighton	19/01/98	
Approved	M. Albrecht		
Released	P. Quinn		

Change Record

Issue/Rev.	Date	Section/Page affected	Reason/Initiation/Document/Remarks
1.0	19/01/98	All	Created

1 INTRODUCTION

This manual describes the implementation of *skycat*, a tool that combines visualization of images with access to catalogs and archive data for astronomy.

The skycat application consists mainly of a small collection of Itcl classes based on the *rtd* (Real-Time Display) and *cat* (Astronomical Catalog Library) packages. Skycat uses inheritance to add catalog features to the *rtd* application, and to add image support to the catalog classes.

In addition, the skycat package contains support for generating a single binary executable that can be more easily distributed on the net, since it does not require any special Tcl environment. The skycat features are also available as a Tcl package or shared library that can be dynamically loaded in a Tcl application.

1.1 Getting Skycat Software, Binaries and Documentation

The latest versions of the skycat sources, binaries, and documentation may always be found under the following URL:

ftp://ftp.archive.eso.org/pub/skycat/

The documentation is in the *doc* subdirectory of the above URL.

1.2 Skycat mailing list

A mailing list has been setup to support a wide collaboration on the *skycat*, *rtd* and *cat* projects. Please see the following URL for more information:

http://archive.eso.org/skycat/

1.3 Distribution and support

The *skycat* binaries are freely available to any users who want to download and use the software at their own risk. Users who wish to modify the source code should contact malbrech@eso.org.

1.4 Purpose

The purpose of this manual is to describe the implementation of the *skycat* application.

1.5 Scope

This document is primarily aimed at software developers who would like to add new features to *skycat* or modify existing features.

1.6 Applicable Documents

This document is based on the following documents:

[1] VLT-PRO-ESO-10000-0228, 1.0 10/03/93 -- VLT Software Programming Standards

1.7 Reference Documents

The following documents are referenced in this document:

[1] VLT-MAN-ESO-19400-1550 1.0 19/01/98 -- Tcl and C++ Utilities, Programmer's Manual

[2] VLT-MAN-ESO-19400-1551 1.0 19/01/98 -- Astronomical Tcl and C++ Utilities

[3] GEN-SPE-ESO-19400-0949 3.1 16/01/98 -- Astronomical Catalog Library, User Manual

[4] VLT-MAN-ESO-17240-0866 1.0 15/01/98 -- Real Time Display, User's Manual

2 OVERVIEW

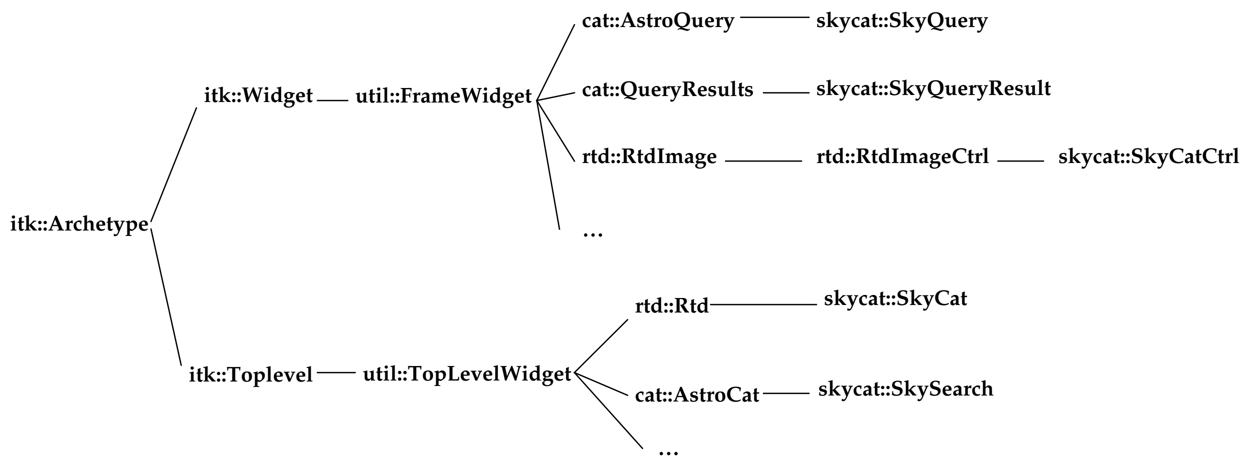
The *skycat* application adds catalog searching features to the *rtd* (real-time image display) by defining subclasses of key Itcl classes in the *rtd* and *cat* packages. This section gives a short overview of the skycat classes and the source code package. The next section goes into more details on the individual widget classes and features for extending skycat, such as plugins.

2.1 Skycat Classes

The following table gives an overview of the Itcl classes defined by *skycat*:

Class Name	Parent Class	Description
SkyCat	**rtd::Rtd**	This class defines skycat's main top level image window and defines some supporting code for use by remote applications using Tcl *send*.
SkyCatCtrl	**rtd::RtdImageCtrl**	Defines the image frame and control panel within the main skycat window and also provides methods for drawing symbols in the image window in world or image coordinates.
SkySearch	**cat::AstroCat**	This is the top level window for searching catalogs. It adds image and symbol plotting support to the *AstroCat* parent class.
SkyQuery	**cat::AstroQuery**	Defines the frame in the catalog window for entering query parameters and adds buttons for setting the default search area from the image or a selected region in the image.
SkyQueryResult	**cat::QueryResult**	This class defines the frame in the catalog window use for displaying and editing catalog query results. The derived class adds a hook to the *rtd pick object* feature for selecting an object in an image to add to a local catalog.

The *Reference* section contains man pages for each of the skycat classes. The parent classes are described in the documentation for the *cat* and *rtd* packages. Part of the class hierarchy is shown below.



Skycat Itcl Class Hierarchy

2.2 Package Organization

The source code for the skycat application consists of the following packages:

Package Name	Namespace	Description
tclutil	**util**	General purpose Tcl and C++ utility classes and widgets.
astrotcl	**astro**	A collection of Tcl and C++ classes for astronomical software (image I/O, compression, world coordinates).
rtd	**rtd**	Real-time image display widget and application.
cat	**cat**	Astronomical catalog library and widgets for searching catalogs and browsing catalog directories.
skycat	**skycat**	Combines the *rtd* image display features with catalog searching features for the *skycat* application.

These packages are available separately or all in a single gzipped tar file from the skycat ftp directory: ftp://ftp.archive.eso.org/pub/skycat/. See the installation instructions at the end of this document for instructions on compiling and installing this package. The documentation for all of the packages is also available in ftp://ftp.archive.eso.org/pub/skycat/doc.

The above packages are also dependent on a Tcl/Tk environment, including the *TclX*, *BLT*, *Itcl* and *Tix* extensions, which are also available from the above URL.

Each of the packages, including the skycat package, can be dynamically loaded in a Tcl application, if compiled with shared library support. Unlike the other packages, the skycat package does not currently contain much C or C++ code, only a few files defining in-line bitmaps and the necessary Tcl package code. The rest of the code is in Tcl (Itcl classes).

2.3 Single Binary Versions of Skycat

Skycat can also be compiled as a single binary that includes all of the necessary bitmaps, colormaps and Tcl source files. This is done using a slightly modified version of ET[1] (*Embedded Tk*), a public domain package designed for this purpose. This works by creating a single Tcl script out of all of the Tcl sources (including all dependent packages). The generated Tcl script is then "*compiled*" by ET , creating a C source file, in which the Tcl code is declared as a string.[2] ET defines the necessary environment so that Tcl scripts are found in the string at run time. The bitmaps and colormap files are compiled in, in all cases, so that there is no problem there.

Earlier versions of skycat used a different method of generating a single binary: emacs *unexec*. In this method, a binary was generated from a running skycat application, in the same way that it is done in GNU Emacs. This method works on Solaris and HP-UX, but is less portable to other platforms and more error prone. Both versions are included in the current source releases, although the ET version is used by default.

1. The ET code had to be extended to support Itcl and Itk.
2. When we move to Tcl-8.x, we will investigate the possibility of using the new Tcl compiler.

3 USER'S GUIDE

This section first describes the skycat Itcl widget classes and then describes a number of ways to extend skycat and add new features by using *plugins*, Tcl *send*, a remote socket interface, and subclassing.

3.1 The Skycat Application

The skycat application comes in two basic versions:

- An interpreted version
- A single binary version

Actually both versions are interpreting Tcl source code. The difference is in the external environment required.

The single binary version has all of the Tcl source files included as strings in the binary, and so does not require any local Tcl installation or special environment to run. This version is easier to distribute over the net.

The interpreted version, on the other hand, requires a complete Tcl environment, with all of the necessary extensions. This version is practical to use during development or when you want to dynamically load Tcl packages at run-time. This version is started using a symbolic link to a shell script *skycat.sh* that sets the necessary environment variables and starts *skycat_wish*, passing it the main *skycat.tcl* source file. The shell script is generated by the skycat configure script and includes the package path names found at the time configure was run.

In both versions, the application is started by calling *SkyCat::startSkyCat* (a class procedure). This proc sets up the necessary Tcl environment and creates an instance of the top level *SkyCat* class. The command line arguments are passed unchanged to the class as options. See Rtd(n) and SkyCat(n) for a detailed description of the options.

Usage:

```
skycat ?fitsFile? ?-option value ...?
```

Skycat Command Line Options

Option	Value	Description
-cat	boolean(1 or 0)	Include ESO/Archive catalog extensions (default: 1).
-colorramp_height	height in pixles	Height of colorramp window (default: 12).
-debug	boolean	Debug flag: If true, run background processes in the foreground to ease debugging.
-default_cmap	basename of colormap file	Specify a different default colormap (default: “real”). See the $RTD_LIBRARY/colormap directory for a list of the available colormaps.
-default_itt	basename of itt file.	Set the default intensity transfer table. (default: “ramp”). See the $RTD_LIBRARY/colormap directory for a list of the available files.
-file	file name	Specify a FITS file to load (‘-’ for stdin). This is the default option, so you can leave off the “-file” part and just specify a file to load.

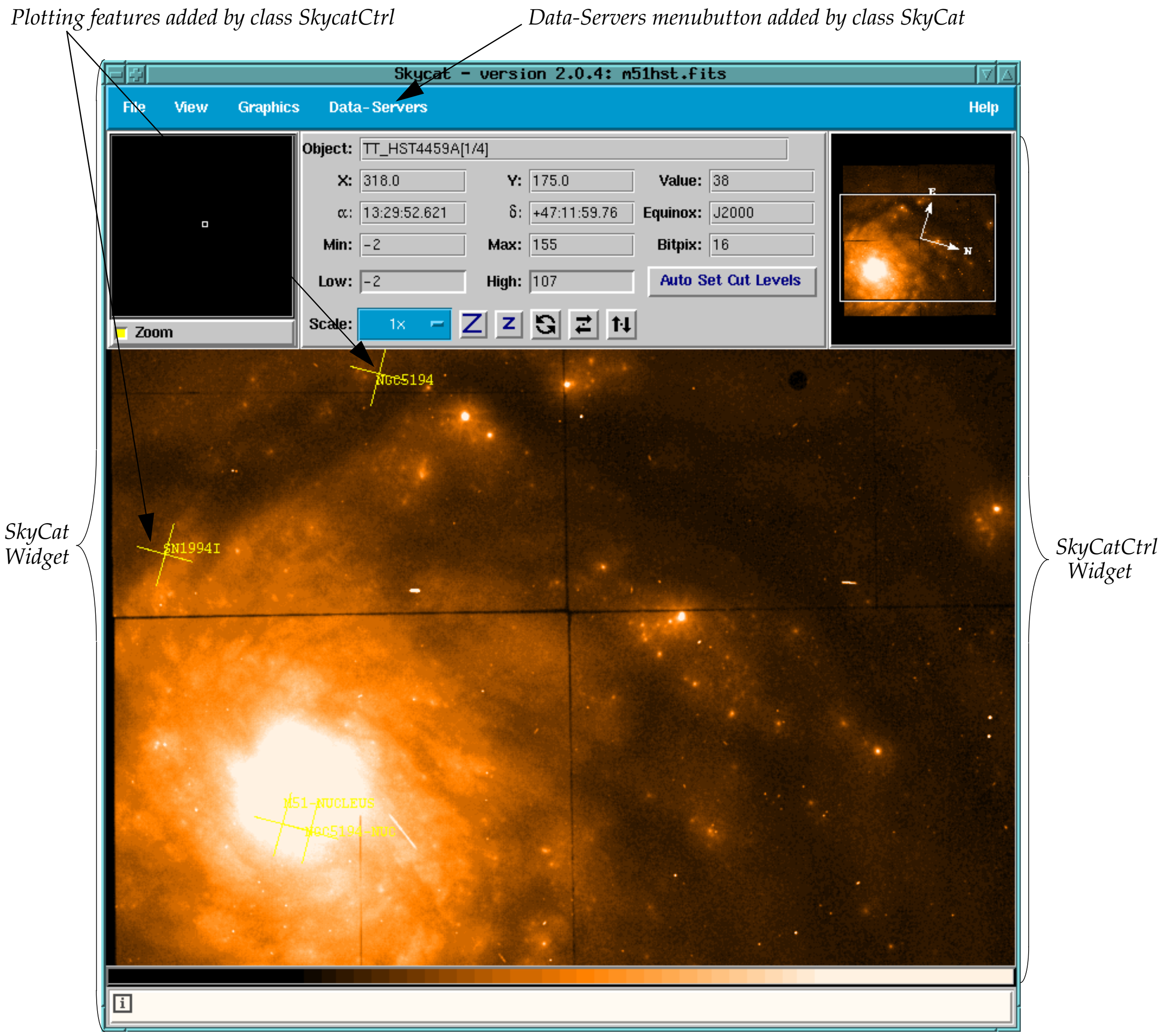
Skycat Command Line Options

Option	Value	Description
-port	port number	Listen for remote commands on the given port (default: 0, which means to choose a port).
-rtd	boolean	If true, include ESO/VLT Real-Time Features in the Skycat menu-bar (default: not included).
-scrollbars	boolean	If true, Display horizontal and vertical scrollbars for the image (not displayed by default).
-shm_data	boolean	If true, put image data in sysV shared memory. By default image files are mapped with mmap, not with sysV shared memory.
-shm_header	boolean	If true, put the image header in sysV shared memory (default: 0).
-usexshm	boolean	If true (default), use X shared memory, if available.
-use_zoom_view	boolean	If true (default), use a "view" of the image for the zoom window.
-verbose	boolean	If true, print diagnostic messages for debugging (default: 0).
-with_colorramp	boolean	If true (default), display the color bar.
-with_grid	boolean	If true, include a WCS grid button (default: 0).
-with_pan_window	boolean	If true (default), display the pan window.
-with_zoom_window	boolean	If true (default), display the zoom window.
-zoom_factor	number	Set the zooming factor for the zoom window (default: 4 x).

3.2 Skycat Widget Classes

3.2.1 Image Window

Skycat defines subclasses of some *rtd* and *cat* classes and combines the image display features of the *rtd* with the catalog features of the *cat* package. The main window of the skycat application is inherited from *rtd* and adds only the *Data-Servers* menu, which is generated automatically based on the list of available catalogs. Skycat's top level window is implemented by the *SkyCat* widget class,



The Skycat Main Window

which is a subclass of the *Rtd* class. The *SkyCat* class adds the *Data-Servers* menu and defines an external *proc* interface, so that other Tcl applications can use the Tcl *send* command to send commands to *skycat*.

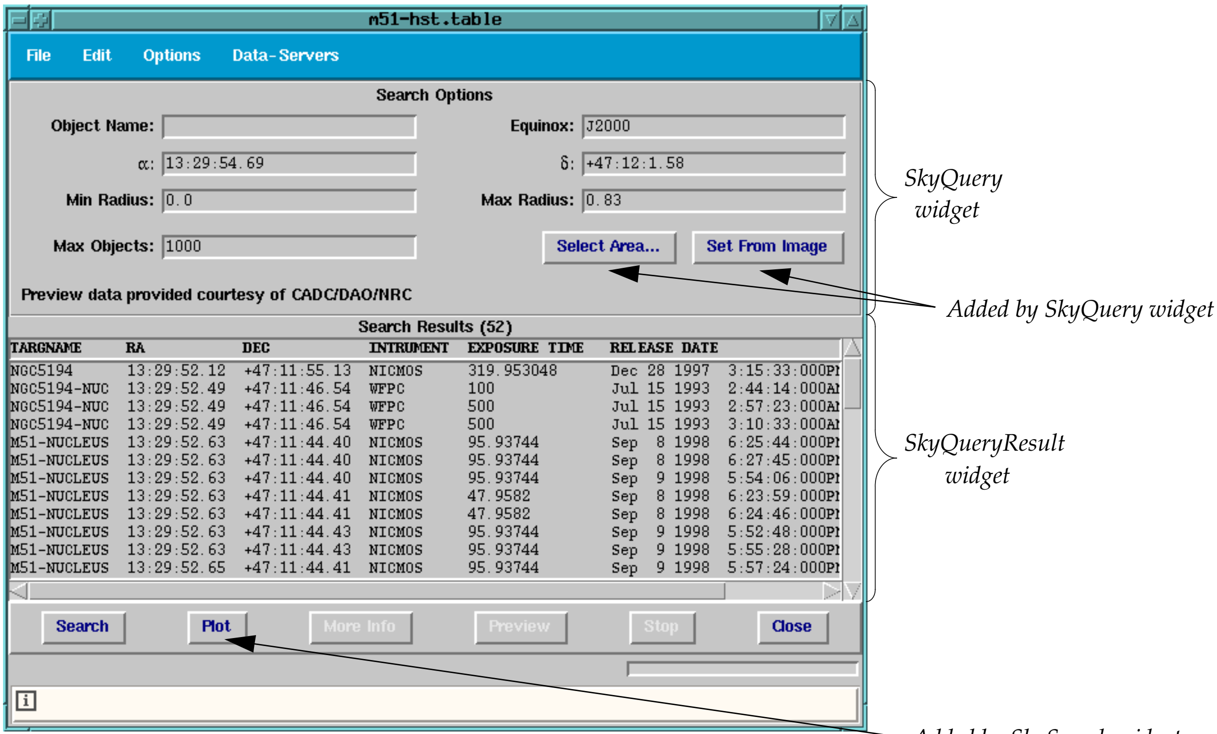
The *SkyCatCtrl* widget defines the inner frame, containing the image and the upper panel. This is a subclass of the *RtdImageCtrl* class and adds methods for drawing plot symbols.

3.2.2 Catalog Window

When a catalog is selected from the *Data-Servers* menu, the window that is displayed is a subclass of the catalog class *cat::AstroCat* called *SkySearch*.

The top level *SkySearch* widget class adds image and plotting support to the *cat::AstroCat* widget. It adds the *Plot* button and redefines the *plot* method to draw symbols in the image for each query result object.

:



The SkySearchWidget

Skycat redefines the internal widgets that make up the *AstroCat* widget here to deal with images and plotting.

The top widget, *SkyQuery*, is a subclass of *cat::AstroQuery* and adds two buttons for setting the area of the image to search:

- "*Select Area*", to interactively select a region of the image to search, and
- "*Set From Image*" to set the range to include the entire image.

The results of the query are displayed in a subclass of *cat::QueryResult* called *SkyQueryResult*. This class redefines the methods for adding an object to a local catalog to include automatic selection of object coordinates using the *RtdImagePick (pick object)* widget.

3.3 Extending Skycat

There is an almost endless list of features that you could add to an application like skycat. Some of the features are of general use, while many others are specific to a certain project or telescope. This section describes some of the ways you can add new features or modify existing features without having to make any changes in the skycat source code. This is important, since it saves a lot of work merging source code after every new version comes out.

3.3.1 Plugins

Nearly all of the top level widgets used in skycat support *plugins*. This is a simple feature inherited from the *TopLevelWidget* base class.

For any given widget class *Foo* that is a subclass of *TopLevelWidget*, a Tcl proc named *Foo_plugin* many be defined. The plugin proc is called for each instance of that class, after the class construction is complete (after calling the *init* method), with the name of the class instance as an argument. If the environment variable *FOO_PLUGIN* is defined (replace *FOO* with the class name in upper case), it is assumed to be the name of the Tcl source file containing the plugin code and defining the Tcl proc *Foo_plugin*. The file is sourced and the file's directory is added to the Tcl *auto_path* variable. You can put plugin procedures for more than one class in a single plugin file, or you can define environment variables and create files for each one separately. If the plugin procedure is defined, it will be called once for each instance of the class. The directory containing the plugin file is automatically appended the tcl *auto_path* variable, so that you can easily split the plugin into different source files in that directory, as long as it contains a *tclIndex* file.

The plugin proc can be used to add features to widgets, such as additional menus or buttons. Once you have the *handle* for the top level widget, it is usually easy to access other internal widgets, if necessary, to make any changes or additions you want. The Tcl language is also very flexible and will allow you to redefine procedures and methods at run time.

3.3.2 Example Skycat Plugin

Below is an example plugin file for the *SkyCat* class. To use this plugin, you have to define the environment variable `SKYCAT_PLUGIN` first:

```
setenv SKYCAT_PLUGIN your-pathname/SkyCat_plugin.tcl
```

The source code for SkyCat_plugin.tcl is shown below:

```
proc SkyCat_plugin {this} {
      set w [info namespace tail $this]
      add_graphics_features $w
}
```

This plugin is used to add some menu items to the *Graphics* menu for saving line graphics to a file in world or image coordinates so that they can be reloaded again later. The proc has to be called *SkyCat_plugin*, since the class name is *SkyCat*. The argument *$this* is the name of an instance of the *SkyCat* Itcl class and is needed in order to be able to call SkyCat methods. The variable *w* is set to the name of the top level window, which is the same string, but without the Itcl namespace information. You can usually use *$w* in place of *$this*, since the namespaces we are using (*skycat*, *cat*, *rtd*, etc.) are *imported* by default. *$w* has the advantage that it can be used to refer to both the Tk widget and the Itcl class instance.

The code for the *add_graphics_features* proc is shown below:

```
proc add_graphics_features {w} {
      set m [$w get_menu Graphics]
      $m add separator

      $w add_menuitem $m command "Save graphics..." \
            {Save line graphics to a file} \
            -command [code save_graphics $w]

      $w add_menuitem $m command "Load graphics..." \
            {Load line graphics from a file} \
            -command [code load_graphics $w]
}
```

The methods used here to add menu items are described in the man page for *TopLevelWidget* in the *tclutil* package documentation. We first get the handle for the *Graphics* menu, and then add the two items to it, including a short help text and a Tcl command to be called to do the work.

We might have wanted to create a new menubutton, "*Annotations*", rather than adding items to the existing one. The code to do that would look something like this:

```
proc add_graphics_features {w} {
      set m [$w add_menubutton Annotations]
```

```
        $w add_menuitem $m command "Save annotations..." \
              {Save line graphics to a file} \
              -command [code save_annotations $w]

        $w add_menuitem $m command "Load annotations..." \
              {Load line graphics from a file} \
              -command [code load_annotations $w]
  }
```

When the user selects the "*Save graphics*" (or "*Save Annotations*") menu item, a tcl proc *save_graphics* is called and that is where the real work begins.

We could also have put this code in an Itcl class, however this plugin does not have its own window (it only adds the menu items), so it makes sense to use plain Tcl procs here. It is usually easiest to define one Itcl class per widget (frame or toplevel).

The save_graphics proc is a bit long, so we'll take it a step at a time as an example. The first thing we have to do is find out the name of the file in which to save the graphics:

```
set filename [filename_dialog]
if {"$filename" == ""} {
      return
}
```

The Tcl proc *filename_dialog* is one of a collection of simple dialog procedures provided by the *tclutil* package. It pops up a file browser and returns the user's selection, or an empty string if no file was selected. So now we know the file. We can check if it exists already, and ask if we should overwrite it:

```
if {[file exists $filename]} {
      if {! [confirm_dialog "File `$filename' exists. Overwrite it?"]} {
            return
      }
}
```

confirm_dialog is another *tclutil* dialog that displays a message and gets a yes or no answer (*OK*, *Cancel*) from the user. Before we go any further, here is a short overview of the simple dialogs that are available in this environment:

Dialogs defined in the Tclutil Package

filename_dialog	Choose a file with a file browser.
confirm_dialog	Ask for confirmation before doing something.
error_dialog	Report an error message.
info_dialog	Display a message.
choice_dialog	Ask the user to make a choice form a number of items.
input_dialog	Ask the user to type something in.

When we save the graphics, we have to ask whether to save the coordinates as world coordinates or image pixel coordinates.

```
set choice [choice_dialog \
      "Please select the type of coordinates to save the graphics in:" \
            {{World Coordinates} {Image Coordinates} Cancel} \
            {World Coordinates} $w]
if {$choice == "Cancel"} {
```

```
        return
    } elseif {$choice == "World Coordinates"} {
        set units {deg J2000}
    } else {
        set units image
    }
```

Here we set *units* to either "*deg J2000*" for world coordinate degrees in J2000 or "*image*" for image pixel coordinates. This is the syntax that is supported by the *rtd*, along with other types, such as "*canvas*" and "*screen*". We will use *$units* below to convert the coordinates of the graphic items from *canvas* to *$units* coordinates. But first, we need to open the output file and write the first line of output indicating the units of the coordinates.

```
    if {[catch {set fd [open $filename w]} msg]} {
        error_dialog $msg
        return
    }
    puts $fd "set units \"$units\""
```

Now we are almost ready to do some work. We still need access to the *canvas* widget holding the image and graphics, the *rtdimage* object that displays the image in the canvas, and the graphics editor object, *draw*, of class *CanvasDraw*, that is normally used to create graphic items and setup bindings for moving and resizing the objects.

```
    set canvas [$w component image component canvas]
    set image [$w component image get_image]
    set draw [$w component image component draw]
    $draw deselect_objects
```

Here we used the Itk *component* method to access the *canvas* component of the *SkyCat* image window. Actually we are going two levels down here, that is why there are two "*component*" references in the first line. The above example could also be coded, perhaps a little more efficiently as follows:

```
    set im [$w component image]
    set canvas [$im component canvas]
    set image [$im get_image]
    set draw [$im component draw]
    ...
```

The second line gets the handle of the internal *rtdimage* object, which we want to use for coordinate conversion. This is not a widget, but the Tk image type called *rtdimage*, which is implemented in C++. We could use the Itcl class object (of type *SkyCatCtrl*) returned by `[$w component image]`, since it implements the same methods as the internal *rtdimage* object (by forwarding them), however, it is more efficient to use the object directly when possible.

The third line in the example gets the handle of the *CanvasDraw* object used to manage the line graphics and uses it to make sure no objects are selected, since we don't want to save the selection handles along with the graphics.

Now we are ready to save the canvas graphics to the file. We can get the list of graphic objects and all of the information we need about them from the *canvas* widget and use the *rtdimage* object to convert the coordinates to the required units.

In the loop below, we write one line for each canvas item (except for image items, which we ignore here). Each line has the format of a Tcl list of the form {*type coordinates configOptions*}, where:

- *type* is the item type (line, rect, etc.)
- *coordinates* are the converted coordinates (a list of floating point numbers). The coordinates are converted by a procedure *convert_coords* shown later.
- *configOptions* is a list of configuration options for the item, such as *{{-width 2} {-fill red} ...}*.

```
foreach item [$canvas find all] {
      set type [$canvas type $item]
      if {"$type" == "image"} {
            continue
      }
      set coords [convert_coords [$canvas coords $item] canvas $units $image]
      set config {}
      foreach cfg [$canvas itemconfigure $item] {
            lappend config [list [lindex $cfg 0] [lindex $cfg 4]]
      }
      puts $fd [list $type $coords $config]
}
```

The *convert_coords* proc converts a list of coordinates from one units to another using an *rtdimage* object, which provides subcommands for this based on the image's FITS header, assuming the image supports world coordinates. The list of coordinates (*coords*) might contain only two values *{x y}* or multiple points *{x0 y0 x1 y1 x2 y2 ... xn yn}* and the return value is a list with the new units.

```
proc convert_coords {coords from_units to_units image} {
      set result {}
      set len [llength $coords]
      for {set i 0} {$i < $len} {incr i 2} {
            set ix [lindex $coords $i]
            set iy [lindex $coords [expr $i+1]]
            $image convert coords $ix $iy $from_units x y $to_units
            lappend result $x $y
      }
      return $result
}
```

The procedure for loading the graphics back is not shown here, but can be found in the source distribution in the *skycat/demos* directory.

Note that you can also define plugin procedures for other top level windows. For example, for the catalog window, the proc would be called *SkySearch_plugin*.

3.3.3 Subclassing

If you are planning on making many changes or additions to skycat, an alternative to using plugins is to define subclasses of the existing classes and redefine selected methods to do what you want. This is the normal way of doing things within skycat and related packages.

For example, skycat's main window is a subclass of the *rtd* main window. The catalog window is a subclass of the *AstroCat* class. By defining a derived class, you can easily add buttons or menu items to existing classes. By looking at the source code and/or documentation and redefining selected methods, you can also change or add to the existing behavior.

One important thing to note here is the role of the Itcl class *constructor* and the *init* method. You should not make any assumptions about the state of a widget or the options while in the constructor. The base classes *TopLevelWidget* and *FrameWidget* both call the *init* method, using an "*after idle*" handler, after all constructors in the class hierarchy have completed.

3.4 Remote Interfaces

3.4.1 Tcl send

By using the Tcl *send* command, a second Tcl application can control skycat and do basically anything. However, skycat offers an interface for using *send* that consists of a number of member procs in the *SkyCat* class.

SkyCat Class Member Procs for use with Tcl send

Name	Description
SkyCat::get_catalog	This proc returns the instance name of a catalog widget. If more than one catalog window is open, it asks the user to select one from a list. If it can't find one, it reports an error and returns an empty string.
SkyCat::get_imagesvr	Same as above, but for the image server (DSS) window.
SkyCat::get_catalog_info	Returns the contents of the catalog window as a Tcl list. The format of the return value is {{*selected_row*} {{*row1*} {*row2*} ...}} where each row is a list of column values. The selected_row is empty if there is no selection, otherwise it is a list of column values in the selected row.
SkyCat::display_image	Fetch and display an image (from DSS), given the world coordinates (ra, dec, equinox) and a width and height in arcmin.
SkyCat::mark_image	Display a rectangle on the image at the given world coordinates center coordinates with the given width and height in arcmin and return the item's canvas tag or id.
SkyCat::unmark_image	Remove the given mark from the image, by specifying the id returned from SkyCat::mark_image.
SkyCat::load_image	Load a FITS image for viewing.
SkyCat::pick_object	Pop up a window and ask the user to select an object in the image. Wait for the selection and return the information for it in the form: *{x y ra dec equinox fwhmX fwhmY angle object background}* An optional Tcl command may be specifed to be called whenever a new object is selected. The command can include a "*send* ..." prefix to call a proc in another application.
SkyCat::get_skycat_images	Return a list of *SkyCatCtrl* class instances in this process (there might be multiple cloned instances of the image window, created via the "*New Window*" menu item).

3.4.2 Remote Socket Interface

The RTD (*Real-Time Display*) documentation has a section in the *User's Guide* describing a remote socket interface. This interface applies to Skycat as well. See the RTD documentation for more details. The socket interface works in basically the same way as Tcl *send*, except that you can use it from a non-Tcl based application. You can send any rtdimage commands over the socket to be evaluated. One useful rtdimage command is *remotetcl*, which evaluates its argument in RTD's Tcl interpreter.

3.4.3 SysV Shared Memory

The RTD documentation also describes the use of sysV shared memory. You can keep the image data in shared memory and use Tcl send or the RTD socket interface to tell skycat when to update the image.

3.4.4 Real-Time Server

rtdServer, the real-time server daemon used in VLT software, can be used with skycat or rtd to display images rapidly from shared memory. See the RTD documentation for a detailed description of this.

3.4.5 Mmap

Skycat and rtd use *mmap* to map image files to memory, since this is more efficient, especially for very large images. Applications can take advantage of this fact by modifying the image data in the file and then signaling skycat or rtd to redisplay the image.

There is currently no provision for specifying that the image header and data should be in separate files, so you need to be careful about changing the size of the FITS header in the file and thus the position of the image data.

We are working on methods to deal more flexibly with mmapped image files.

4 REFERENCE

Following is a list of man pages for the various classes and utilities provided in the *skycat* package.

4.0.1 SkyCat(n)

NAME

    SkyCat - image display application class with catalog extensions

NAMESPACE

    skycat

PARENT CLASS

    rtd::Rtd

SYNOPSIS

    SkyCat <path> ?options?

DESCRIPTION

```
This class defines a top level window for the skycat application.  The
easiest way to create an instance this class is via the "startSkyCat"
proc. It sets up the environment, creates an instance of the class and
then waits for the application to exit.

The SkyCat widget supports the same options as the Rtd widget, its base
class, and adds some of its own options.  The widget options are the
same as the skycat command line options, since these are passed
unchanged to the widget.
```

OPTIONS

```
-cat
        Flag: if true, display the data-servers menu (catalog features).

-dhsdata
        Directory used to hold image files from OLAF/DHS.

-dhshost
        For OLAF (On-Line Archive Facility): name of DHS host machine.

-help_url
        Url to use for the help menu - link to skycat WWW page.

-rtd
        Flag: if true, display the real-time menu (VLT features).
```

METHODS

```
add_help_menu {}
        Add a menubutton with help items.

add_olaf_menu {}
        Add a menubutton with OLAF items.

clone {}
        Make a new main window (redefined from parent class).

feedback {msg}
        This method is redefined here to get feedback during startup.

init {}
        Called after the options have been evaluated.
```

```
make_init_window {}
        Display a window while the application is starting up.

make_rtdimage {}
        Create the rtd image widget with catalog extensions (redefined
        from parent class to use class with catalog features added).

select_region {x0 y0 x1 y1}
        This method is called when a region of the image has been selected
        (via -regioncommand option when creating image above).  The
        arguments are the bounding box of the region in canvas coords.
        pass it on to any catalog windows to select any catalog symbols in
        the region.

setXdefaults {}
        Set default X resources for colors and fonts, and set some default
        key bindings. This method is called from the parent class and
        overridden here.  These are built-in defaults that the user can
        also override in the ~/.Xdefaults file.
```

PROCS

```
display_image {ra dec width height {equinox 2000}}
        This proc can be called via send from another application to
        display an image given the coordinates and a width and height in
        arcmin.

get_catalog {{what "catalog"}}
        This proc returns the instance name of the catalog (or image
        server) widget If more than one is open, it asks the user to
        select which one.  If it can't find one, it reports an error and
        returns "".

get_catalog_info {}
        This proc can be called via send from another application to
        return the contents of the catalog window as a Tcl list.

        The format of the return value is {{selected_row} {{row1} {row2}
        ...}} where each row is a list of column values. The selected_row
        is empty if there is no selection, otherwise it is a list of
        column values in the selected row.

get_imagesvr {}
        This proc returns the instance name of the image server widget If
        more than one is open, it asks the user to select which one.  If
        it can't find one, it reports an error and returns "".

get_skycat_images {}
        Return a list of SkyCatCtrl class instances in this process (there
        might be multiple cloned instances...).

load_image {filename}
        This proc can be called via send to load a fits image for
        viewing.

mark_image {ra dec width height}
        This proc can be called via send from another application to
        display a rectangle on the image at the given center coords with
        the given width and height and return the item's canvas tag or
        id.

pick_object {{cmd ""}}
```

```
        Pop up a window, ask the user to select an object in the image,
        wait for the selection and return the info for it in the form:
        {$x $y $ra $dec $equinox $fwhmX $fwhmY $angle $object $background}

        An optional Tcl command may be specifed to be called whenever a
        new object is selected. The command can include a "send ..."
        prefix to call a proc in another application.

startSkyCat {}
        Start the application with the above class as the main window This
        proc is called from tkAppInit.c when we are running the single
        binary version (dumped with emacs unexec) Note that the binary
        version doesn't need to set auto_path or look for Tcl sources or
        colormaps at run-time, since they are already loaded in the
        binary.

unmark_image {id}
        Remove the given mark from the image (id returned from
        mark_image).
```

PROTECTED VARIABLES

```
percent_done_
        Used in startup dialog .
```

SEE ALSO

```
Rtd(n)
```

\- - - - - -
Last change: 02 Feb 98

4.0.2 SkyCatCtrl(n)

NAME

```
SkyCatCtrl - image display widget with catalog extensions
```

NAMESPACE

```
skycat
```

PARENT CLASS

```
rtd::RtdImageCtrl
```

SYNOPSIS

```
SkyCatCtrl <path> ?options?
```

DESCRIPTION

```
This class extends the RtdImageCtrl class (see RtdImageCtrl(n)) by
adding image catalog extensions and user interface dialogs for use with
astronomical catalogs.
```

OPTIONS

```
-debug
        Flag: if true, run queries in the foreground for better
        debugging.
```

METHODS

```
about {}
        Display a popup window with information about this application.

clear {}
        This method is also redefined from the parent class to set the
        window header info.

convert_coords {in_x in_y in_units out_units}
        Convert the given input coordinates in the given input units to
        the given output units and return a list {x y} with the new
        values.  The units may be one of {canvas image wcs deg "wcs
        $equinox", "deg $equinox"}.

convert_dist {in_x in_y in_units out_units}
        Convert the given input distance in the given input units to the
        given output units and return a list {x y} with the new values.
        The units may be one of {canvas image wcs deg "wcs $equinox", "deg
        $equinox"}.

display_logo {}
        Display the skycat logo in the center of the image window with
        some copywrite text Note that we assume here that the logo was
        created previously and the name of the image is in the global var
        skycat_logo.

draw_arrow {x y xy_units radius radius_units bg fg tags {ratio 1}
            {angle 0}}
        Draw an arrow symbol at the given coords See "draw_symbol" for a
        description of the arguments.

draw_circle {x y xy_units radius radius_units bg fg tags args}
```

```
        Draw a circle at the given coords See "draw_symbol" for a
        description of the arguments.

draw_compass {x y xy_units radius radius_units bg fg tags {ratio 1}
            {angle 0}}
        Draw a compass symbol at the given coords See "draw_symbol" for a
        description of the arguments.

draw_cross {x y xy_units radius radius_units bg fg tags args}
        Draw a cross symbol at the given coords See "draw_symbol" for a
        description of the arguments.

draw_diamond {x y xy_units radius radius_units bg fg tags args}
        Draw a diamond at the given coords See "draw_symbol" for a
        description of the arguments.

draw_ellipse {x y xy_units radius radius_units bg fg tags {ratio 1}
            {angle 0}}
        Draw an ellipse of the given size at the given coords.  See
        "draw_symbol" for a description of the arguments.

draw_line {x y xy_units radius radius_units bg fg tags {ratio 1} {angle
            0}}
        Draw a line through the center of the object at the given coords
        along the north/south axis, then rotated by the given angle.  See
        "draw_symbol" for a description of the arguments.

draw_plus {x y xy_units radius radius_units bg fg tags {ratio 1} {angle
            0}}
        Draw a plus symbol at the given coords See "draw_symbol" for a
        description of the arguments.

draw_square {x y xy_units radius radius_units bg fg tags args}
        Draw a square at the given coords See "draw_symbol" for a
        description of the arguments.

draw_symbol {symbol x y xy_units radius radius_units bg fg tags {ratio
            1} {angle 0}}
        Draw the given symbol in the image.

        $symbol is the name of a symbol and may be one of {circle square
        plus cross triangle diamond ellipse arrow line compass}

        $x and $y are the coordinates of the center of the object to draw
        in the given units ($xy_units), one of {image canvas "wcs
        $equinox" "deg $equinox"}.

        $radius is the radius for the symbol in the given units
        ($radius_units)

        $bg and $fg are the background and foreground colors for drawing.
        If they are different, 2 lines are drawn, otherwise 1.

        tags is a list of canvas tags for the symbol.

        $ratio and $angle are optional and used only where applicable.
        They indicate the size ratio (width/height) and the rotation angle
        in degrees (relative to north) . If x and y are in world
        coordinates, the angle takes that into account.

draw_triangle {x y xy_units radius radius_units bg fg tags args}
        Draw a triangle at the given coords See "draw_symbol" for a
        description of the arguments.
```

```
get_compass {x y xy_units radius radius_units {ratio 1} {angle 0}}
        Return the canvas coordinates of the 3 points: center, north and
        east, given the center point and radius in the given units, an
        optional rotation angle, and an x/y ellipticity ratio.  If the
        image supports world coordinates, that is taken into account (the
        calculations are done in RA,DEC before converting to canvas
        coords).  The conversion to canvas coords automatically takes the
        current zoom and rotate settings into account. The return value is
        a list {x y nx ny ex ey} for the 3 points center, north and east.

list_windows {}
        Return a list of top level windows that this class might create.

load_fits_ {}
        This method is redefined here from the base class to include the
        file name in the window header.

mark_image {ra dec width height}
        Display a rectangle on the image at the given center coords with
        the given width and height and return the items tag or id.

rotate_point {x y cx cy angle}
        Rotate the point x,y around the center point cx,cy by the given
        angle in deg.

select_area {{shape rectangle}}
        Ask the user to select an area of the image by dragging out a
        region and return as a result a list of the form {x0 y0 x1 y1} in
        pixels.

selected_area {id x0 y0 x1 y1}
        Called when the user has selected an area of the image.

send_to_netscape {url}
        Send a URL to be displayed by netscape.

subscribe {variable dhshost dhsdata}
        Subscribe to (or unsubscribe from) the OLAF DHS server images
        (ESO/Archive On-Line Archive Facility project: use the -dhshost
        and -dhsdata options to add this feature.) The first argument is
        the name of the trace variable used in the checkbutton menuitem.
        dhshost is the name of the host running the DHS server, to which
        we subscribe.  dhsdata is the directory to use to hold the images
        files (temporary files).

unmark_image {id}
        Remove the given mark from the image.

update_title {}
        Update the toplevel window header and icon name to include the
        name of the file being displayed.
```

PROTECTED VARIABLES

```
pi_
        Const PI.

rad_
        Const PI/180.

subscribe_pid_
        Pid of rtdSubscribe process (OLAF).
```

SEE ALSO

`RtdImageCtrl(n)`

\- - - - - -

Last change: 02 Feb 98

4.0.3 SkyQuery(n)

NAME

SkyQuery - Widget for searching catalogs and plotting the results in an image.

NAMESPACE

skycat

PARENT CLASS

cat::AstroQuery

SYNOPSIS

SkyQuery <path> ?options?

DESCRIPTION

A SkyQuery widget is a frame containing entries for search options (inherited from class AstroQuery) with added support for plotting objects in an image.

OPTIONS

-skycat

Name of SkyCat itcl widget.

METHODS

add_search_options {}

Add (or update) the search options panel (redefined from parent class AstroCat to add buttons).

convert_coords {in_x in_y in_units out_units}

Convert the given input coordinates in the given input units to the given output units and return a list {x y} with the new values. The units may be one of {canvas image wcs deg "wcs $equinox", "deg $equinox"}.

get_image_center_radius {wcs_flag}

Return a list of values indicating the center coordinates and radius of the current image. If wcs_flag is 1, the return list is {ra dec equinox radius-in-arcmin}, otherwise {x y radius-in-pixels}. If no image is loaded, an empty string is returned.

select_area {}

Ask the user to select an area to search interactively and insert the resulting radius and center pos in the catalog window.

select_image_area {wcs_flag}

Ask the user to select an area of the image by dragging out a region on the image return the resulting center pos and radius as a list of {x y radius-pixels}, or {ra dec equinox radius-in-arcmin} if wcs_flag is 1. An empty string is returned if there is no image or the user cancels the operation.

set_default_values {}

Set the default values for the search panel entries: (redefined from parent class AstroCat to set values from the image).

```
set_from_image {}
        Set the default values for position and radius from the image, if
        there is one.
```

PROTECTED VARIABLES

```
image_
        Internal rtdimage image for main image.
```

SEE ALSO

```
AstroQuery(n)
```

\- - - - - -
Last change: 02 Feb 98

4.0.4 SkyQueryResult(n)

NAME

SkyQueryResult - Widget for viewing query results with skycat image support.

NAMESPACE

cat

PARENT CLASS

cat::QueryResult

SYNOPSIS

```
SkyQueryResult <path> ?options?
```

DESCRIPTION

A SkyQueryResult widget is defined as a QueryResult (see cat package) with some added support for skycat image access, used for selecting objects to add to a local catalog.

OPTIONS

-skycat

Name of SkyCat itcl widget.

METHODS

edit_selected_object {{command ""}}

Pop up a window so that the user can edit the selected object(s) The optional command is evaluated with no args if the object is changed. (redefined from parent class AstroCat to add image support).

enter_new_object {{command ""}}

Pop up a dialog to enter the data for a new object for a local catalog The command is evaluated after the users enters the new data. (redefined from parent class to add image support).

SEE ALSO

QueryResult(n)

\- - - - - -

Last change: 02 Feb 98

4.0.5 SkySearch(n)

NAME

SkySearch - Widget for searching a catalog and plotting the results

NAMESPACE

skycat

PARENT CLASS

cat::AstroCat

SYNOPSIS

SkySearch <path> ?options?

DESCRIPTION

This class extends the AstroCat catalog widget browser class (see AstroCat(n) to add support for plotting objects and displaying images.

OPTIONS

-canvasfont

Font used in canvas to mark objects.

-id

Optional unique id, used in searching for already existing catalog widgets.

METHODS

add_dialog_buttons {}

Add the dialog button frame (redefined from parent class AstroCat to add Plot button).

add_result_table {}

Add the table for displaying the query results (redefined from parent class AstroCat to add image support).

add_search_options {}

Add the search options panel (redefined from parent class AstroCat to add image support).

bind_symbol {id keysym command}

Add the given binding to the given symbol id is the unique id for the symbol, seq and command are the parameters to bind.

clear {}

Clear the table listing (done in base class) and remove any plot symbols from the display.

convert_coords {in_x in_y in_units out_units}

Convert the given input coordinates in the given input units to the given output units and return a list {x y} with the new values. The units may be one of {canvas image wcs deg "wcs $equinox", "deg $equinox"}.

delete_objects {name}

Delete any graphic objects in the image belonging to the given catalog.

```
deselect_objects {}
        Deselect any objects in the image.

deselect_symbol {tag}
        Deselect the given symbol, given its canvas tag or id.

display_image_file {filename}
        Display the given image file.

gen_blank_image {}
        Generate a dummy blank image for the purpose of plotting catalog
        objects on it.  Return 0 if OK, otherwise 1.

gen_pix_image {radius}
        Generate a blank image without WCS. radius is radius of the image
        in pixels, Returns "0" if all is OK.

gen_wcs_image {ra_deg dec_deg equinox radius}
        Generate a blank image that supports world coordinates for the
        purpose of plotting catalog objects.  ra_deg, dec_deg and equinox
        give the center of the image (in deg), radius the radius in
        arcmin.  Returns "0" if all is OK.

get_display_height {}
        Retun the height of the image display canvas.

get_display_width {}
        Retun the width of the image display canvas.

get_image_name {}
        Return the name (file or object name) of the currently loaded
        image, or empty if no image is loaded.

label_object_in_image {id name}
        Insert the id for the given object in the image near the object
        and return a string containing status info. name identifies the
        source catalog (short_name).

label_selected_object {}
        Insert the Id for the object selected in the Table in the image
        near the object.

make_label {name id x y units text color}
        Add a label to the image at the given coordinates (in the given
        units) with the given text and color. The id arg should be a
        unique id for the label in the catalog and $name should be the
        short name of the catalog.  $units may be any of the units
        supported by the RTD {image canvas screen "wcs $equinox" "deg
        $equinox"}.

make_short_help {}
        Add a short help window and set the help texts (redefined from
        parent class AstroCat).

picked_wcs_object {x y units}
        This method is called when the user clicks on a graphic symbol for
        a star.  The user might be selecting this star, so call the
        RtdImage method to do that.

plot {}
        Plot the stars/objects found in the previous search in the image
        window.  The symbols to use are taken from the config file.
```

```
plot_again {}
        Re-plot the listed objects.

plot_objects {cols symbol expr}
        Plot the current objects using the given plot info from the
        catalog config file.

        cols should be a list of column names that may be used in $symbol
        or $expr.

        $symbol may be a simple symbol name, such as "circle" or a list
        such as {ellipse color ratio angle label condition}.

        The last 4 args may be expressions using the column names given.
        ratio gives the ratio of x/y, angle is the rotation angle, label
        is an optional text expr and condition may evaluate to 1 or 0 to
        indicate whether or not the symbol should be displayed for an
        object.

        $expr should be an expression that evaluates to the size of the
        radius of the object in pixes, or a list {expr units} where  units
        is one of {image deg ...} as supported by rtd.

plot_row {rownum row id x y pos_units cols symbol bg fg ratio angle
            label cond size size_units name}
        Plot the object for the given row ($row).  $rownum is the row
        number for error reporting.

        id is a unique id for the row.

        $x and $y are the coords of the center of the object in the given
        units (pos_units).

        $cols is a list of column names for columns whose values are used
        in $expr.

        $symbol is the plot symbol

        $bg and $fg are the background and foreground colors for drawing.
        If they are different, 2 lines are drawn, otherwise 1.

        $ratio and $angle are optional arguments to the plot proc (for
        ellipse, plus).

        $label is an expr to use to label the object.

        $cond should evaluate to a boolean expr. If true, plot the object,
        otherwise not.

        $size should be an expr and evaluate to the radius on the object
        in the given units (size_units), using the column names as tcl
        variables.

        $name is the short name of the catalog, used to identify all
        catalog objects for this catalog.

        This method returns the string "error" if there were errors and
        plotting should stop.

plot_symbol {symbol id name x y xy_units radius radius_units bg fg
            ratio angle label}
        Plot the given symbol in the image.
```

```
        $symbol is the plot symbol (circle, ellipse, etc...)

        id is a unique id for the row.

        $name is the short name of the catalog, used to identify all
        catalog objects for this catalog.

        $x and $y are the coordinates of the center of the object to plot
        in the given units ($xy_units).

        $radius is the radius for the symbol in the given units
        ($radius_units)

        $bg and $fg are the background and foreground colors for drawing.
        If they are different, 2 lines are drawn, otherwise 1.

        $ratio and $angle are optional arguments to the plot proc (for
        ellipse, plus).

        $label is an expr to use to label the object.

select_region {x0 y0 x1 y1}
        This method is called when a region of the image has been selected
        (From class SkyCat, via -regioncommand option when creating the
        image).  The arguments are the bounding box of the region in
        canvas coords.  Select any catalog symbols in the region.

select_result_row {}
        Called when a row in the table is selected. Redefined from parent
        clas to also select the plot symbol.

select_symbol {id toggle rownum}
        Select the given symbol, given its id and optional row number in
        the list. If $toggle is 0, deselect all other symbols first,
        otherwise toggle the selection of the items given by $id.

set_state {state}
        Set/reset widget states while busy (redefined from parent class
        AstroCat).
```

PROTECTED VARIABLES

```
canvas_
         canvas window containing main image.

draw_
        CanvasDraw object for drawing on image.

image_
        Internal rtdimage image for main image.

skycat_
         SkyCatCtrl widget instance.
```

SEE ALSO

```
AstroCat(n)
```

```
- - - - - -
Last change: 02 Feb 98
```

4.0.6

5 INSTALLATION

5.1 Before you build skycat

The skycat package depends on the following packages being already installed. These are available from ftp://ftp.archive.eso.org/pub/skycat:

- tclutil - Tcl and C++ Utilities Package
- astrotcl - Astronomical Tcl and C++ Utilities Package
- rtd - Real-Time Image Display
- cat - Astronomical Catalog Library

These packages also assume the following standard Tcl packages are installed:

- itcl-2.2 - [Incr Tcl], includes tcl7.6, tk4.2
- BLT-2.1 - BLT toolkit, graphs and other widgets
- tclX-7.6.0 - Extended Tcl

These packages are available from the TCL archives.

See: http://www.tcltk.com/ for Itcl.

and http://www.NeoSoft.com/tcl/ for TclX and other contributed Tcl software

You can also find a tar file on ftp://ftp.archive.eso.org/pub/skycat that contains all of the Tcl packages used, with any required patches installed.

In addition, skycat requires the *Tix* package, which is used for the catalog directory browser:

- Tix4.1.0 - Tix widget set

Tix is also avilable from the above URLs.

You can install the standard Tcl/Tk packages anywhere, however it is easiest to install them all in the same directory (using the same -prefix argument to configure) and define the `TCLTK_ROOT` environment variable to point to the top level install directory. This variable is used, if defined, by the *tclutil*, *astrotcl, rtd, cat, and skycat* package configure scripts, to locate the Tcl installation.

If you are building the *tclutil*, *astrotcl, rtd, cat* or other related packages also, it is best to place these packages under the same parent directory, so that you have $dir/tclutil, $dir/astrotcl, and $dir/cat (so there is no problem finding the local include files and libraries).

5.2 Building Skycat

To build the software, type:

```
configure
make
make install
```

The default install dir is /usr/local. You can specify the -prefix argument to configure to change this:

```
configure -prefix $INSTALLDIR
```

The compilers used are the same as those used to build the *tclutil* package. The file `tclutilConfig.sh` is read by configure to get the necessary information.

If you specify the configure option --enable-shared:

```
configure -prefix $INSTALLDIR --enable-shared
```

you can build a shared library and load the Tcl commands with the Tcl *load* command or load it as a

Tcl package under the name *Skycat*. The Tcl code to do this would look something like this, assuming the pkgIndex.tcl file for the *skycat* package is in your Tcl *auto_path* variable:

```
if {[catch {package require Skycat} msg]} {
 puts "error loading Skycat package $msg"
 exit 1
}
```

Note: Note: If you want to build a stand-alone binary version of skycat, you must also build static libraries (.a archives) for all of the required packages (tclutil, astrotcl, rtd, skycat, itcl, tclx, blt and tix).

5.3 VLT Make Procedure

As an alternative to running configure and make, you can also do this:

```
cd src
make
make install
```

The Makefile in the skycat/src directory runs configure and then make as described above. You can also specify options to that Makefile, for example:

```
cd src
make PREFIX=$INSTALLDIR CONFIGURE_FLAGS=--with-gcc
```

The PREFIX variable defaults to /usr/local and is the prefix of the directory in which to install the software.

5.4 Running

The skycat application may be started in the following ways:

1. Interpreted version:
   a. The interpreted version, in the source tree:
      ```
      % cd skycat/interp/demos
      % skycat
      ```
   b. The interpreted version, installed with *-prefix $INSTALLDIR*:
      ```
      % $INSTALLDIR/bin/skycat
      ```
2. The single binary version has the OS version appended to the name, for example skycat-SunOS-5.5.1, and can be found in the following directories:
   a. In the source tree:

      `skycat/et/bin` *(default)*

      `skycat/unexec/bin` *(no longer supported)*
   b. In the install tree:

      `$INSTALLDIR/bin`