E U R O P E A N S O U T H E R N O B S E R V A T O R Y

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 2.2

Date 5/16/99

Prepared	A. Brighton	5/16/99	
	Name	Date	Signature
Approved	M. Albrecht		
	Name	Date	Signature
Released	P. Quinn		
	Name	Date	Signature

Issue/Rev.	Date	Section/Page affected	Reason/Initiation/Document/Remarks
1.0	**19/01/98**	**All**	**Created**
2.0	**01/08/98**	**Installation, Reference**	**Updated.**
2.1	**11/09/98**	**All**	**Updated, setup HTML, cross refs.**

Table of Contents

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.

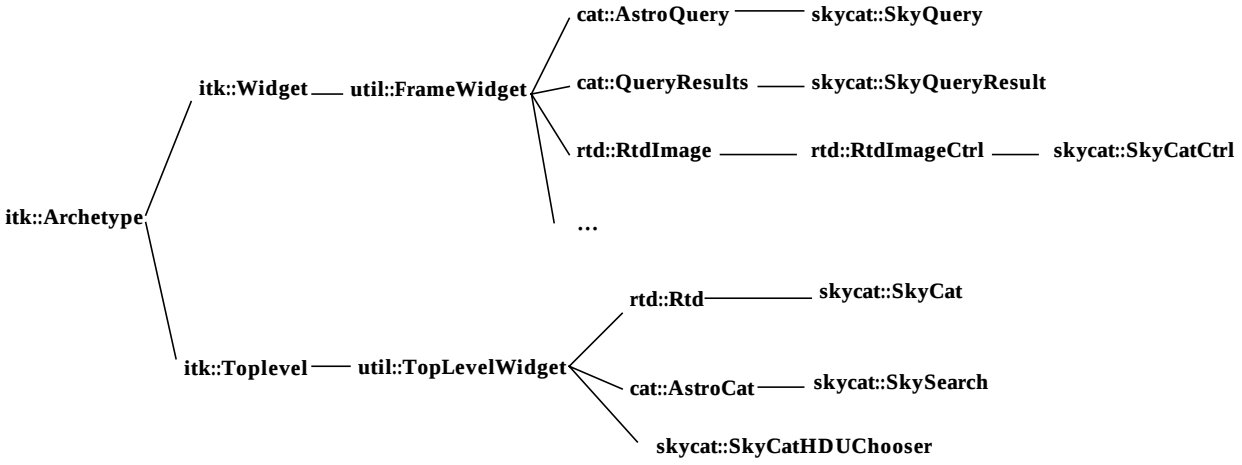
2.1 Skycat Classes

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.
SkyCatHDUChooser	**util::TopLevelWidget**	This is a pop up window that displays a list of available HDUs (FITS header/data units). This includes a listing of any FITS tables and image extensions. A small version of each image extension is also displayed.
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 be-

low.



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 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.

> **Note:** The rtd and cat package tar files each contain the tclutil and astrotcl packages, so you won't normally need to get them separately. For compatibility with previous releases they are treated as internal packages.

The above packages are also dependent on a Tcl/Tk environment, including the *TclX*, *BLT and Itcl*, 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.

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.

For Tcl7.6/Tk4.2, 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. 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.

For Tcl8.0/Tk8.0, the *Scriptics TclPro prowrap*[2] application is used, if available. The ET version does not currently work with Tcl8.0.

1. The ET code had to be extended to support Itcl and Itk.
2. See http://www.scriptics.com/tclpro/ for information about TclPro and prowrap.

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. The option value if the basename of the file.
-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.

Skycat Command Line Options

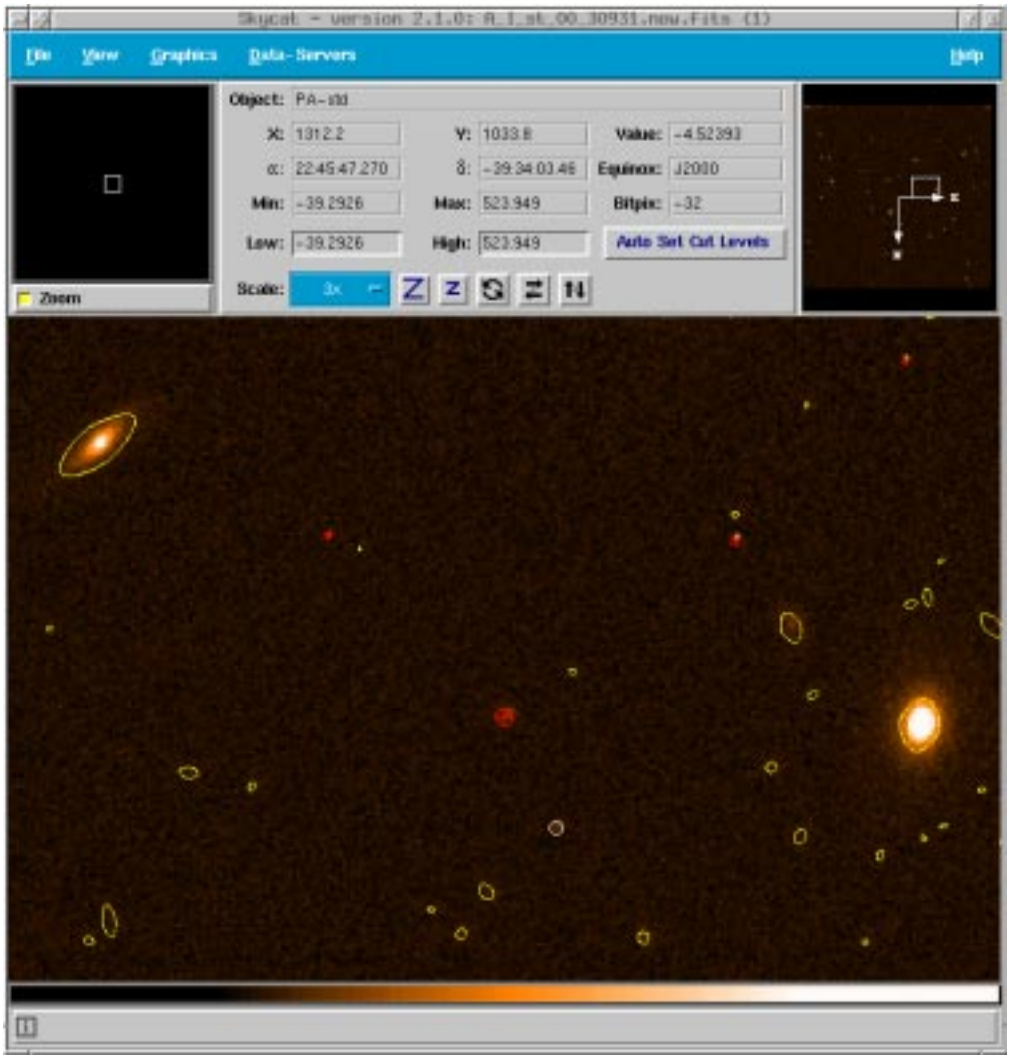
Option	Value	Description
-file	file name	Specify a FITS file to load (‘-’ for stdin). This is the default option, so you can leave off the “-file” part and just specify a file to load.
-float_panel	boolean	If the option value is 1, the skycat info panel is put in a separate popup window, leaving more space for the image window (The default is off).
-min_scale -max_scale	number (*2 = zoom 2x* *-2 = zoom 1/2x*)	Specify the min and max scale values for the *Magnification* Menu. Negative values shrink the image, positive values zoom in closer. The default values are -10 and 20.
-panel_layout	layout	With this option, you can change the order of the zoom and pan windows in the layout. The default layout is: *zoom window* on the left, *info panel* in the center and *pan window* right. If the layout is specified as *saoimage*, a layout similar to saoimage is used (*info panel*, *pan window*, *zoom window*). If *reverse* is specified, the order of the windows is the reverse of the default.
-pickobjectorient	vertical, horizontal	Specify the orientation of the “Pick Object” window. The default is *horizontal* (panel left, image right). *vertical* puts the image at top and panel underneath.
-port	port number	Listen for remote commands on the given port (default: 0, which means to choose a port).
-remote	boolean	If *-remote 1* is specified and a skycat process is already running, the existing skycat process is sent a message and asked to open a new window and the new skycat process exits immediately. This has the advantage of sharing the image colormap and using fewer system resources, however it depends on being able to use the Tcl send mechanism. For security reasons, Tcl send will not work if you are using *xhost* based X security. You need to use *Xauth* security. See the *Tcl/Tk Tools* book from O’Reilly for more on this topic.
-rtd	boolean	If true, include ESO/VLT Real-Time Features in the Skycat menubar (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.

Skycat Command Line Options

Option	Value	Description
-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 The Main Skycat 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(n)* widget class, 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(n)* 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 The 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(n).*

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.

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

The top widget, *SkyQuery(n)*, 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(n).* 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

Plugins are defined here as Tcl procedures that are called to extend skycat and add new features. Skycat supports two types of plugins: one at the *widget* level, which is called for each instance of a top level widget, after it has been constructed, and one at the *application* level, called once for the application, before any widgets are created. The Tcl plugin procedure can do something simple, such as add a new menu item with a new feature, or something quite complex, including replacing the main application class with a derived class and dynamically loading Tcl packages from shared library files.

3.3.1.1 Widget Level 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.

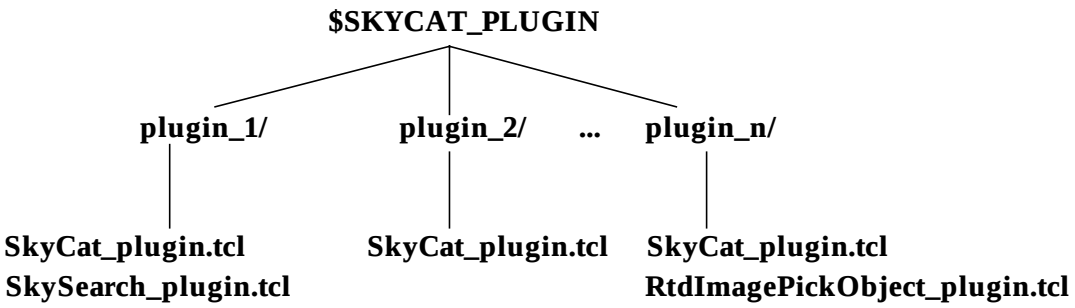
The plugin source files are located as follows: If the environment variable *FOO*`_PLUGIN` is defined (replace *FOO* with the Itcl *class* or *application* name in upper case), it is assumed to be a colon separated list of plugin source file names, or the names of directories containing plugin files, or containing subdirectories with plugin files (see the example source tree below).

In the case of skycat, you could define the environment variable `SKYCAT_PLUGIN`, since skycat is the name of the application and also the name of the main class. You could also define other environment variables, such as `SKYSEARCH_PLUGIN`, for other toplevel Itcl classes, if you want to have a Tcl proc be called for each instance.

Example:

```
setenv SKYCAT_PLUGIN "$dir1/myplugin1.tcl:$dir2/myplugin2.tcl:$dir3"
```

In the above example, the two source files (*$dir1/myplugin1.tcl* and *$dir2/myplugin2.tcl*) will be loaded as plugins, as well as any file named *$dir3/${classname}_plugin.tcl* or *$dir3/*/${classname}_plugin.tcl*. Probably the simplest way to organize the plugins is to define a single top level plugin directory and create a subdirectory for each plugin.



Typical Skycat plugin directory structure.

You can put plugin procedures for more than one class in a single plugin file, or 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.

3.3.1.2 Example Widget Level 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, for example:

```
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 main Itcl class name is `SkyCat`. The argument `$this` is the name of an instance of the `SkyCat` class and is needed in order to be able to call 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(n)*) 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`.

3.3.1.3 Application Level Plugins

Some tasks might require more than just adding a new menu item to the existing window. You might need to dynamically load shared libraries for new Tcl packages or even replace the main application widget (*SkyCat(n)*) with a new derived class widget, which modifies the default behavior.

An application plugin is defined in the same way as the widget plugins described above, by defining the environment variable `SKYCAT_PLUGIN` as a colon separated list of files or directories. However in this case, the plugin source file does not only (or necessarily) define the `SkyCat_plugin` procedure, but also includes Tcl commands to execute at the global level. This works because the plugin files for the main Itcl application class (`SkyCat`) are sourced before any widgets are created (other plugin files are loaded on demand as needed).

Although the plugin procedure is only called after a widget is created, the file for the main widget is sourced before the first instance is created, giving you a chance to execute code, where you can, among other things, redefine the definition of the `rtdimage` Tk image type to include new commands defined in a C++ subclass of `Skycat` (which is a subclass of `RtdImage`).

Before creating the first instance of the main skycat window, the application plugin files are *sourced*. In the plugin code, you can set the global Tcl variable `mainclass` to the name of a new Itcl class derived from the `SkyCat` widget class. The code that creates the main window will then use that class in place of the `SkyCat` class. This is one way to gain full control of the application without modifying it and also allows you to add new command line options, since these are the same as the options for the main application class widget.

3.3.1.4 Example Application Level Plugin

Here is a very simple example of an application plugin, not for *Skycat*, but for *Rtd*. Just to demonstrate how it works, we could make *Skycat* be a plugin for *Rtd*, so that when we define the environment variable RTD_PLUGIN to point to this file (or a directory containing the file Rtd_plugin.tcl) and then start "*rtd*", the window that actually comes up will be the *skycat* main window:

```
# assumes these environment variables are defined
lappend auto_path $env(CAT_LIBRARY) $env(SKYCAT_LIBRARY)

# load the required packages
foreach pkg {Cat Skycat} {
    if {[catch {package require $pkg} msg]} {
        puts "error loading $pkg package: $msg"
        return
    }
}

# use this class for the main window
set mainclass SkyCat
```

A similar plugin can also be defined for skycat. There is one small problem though with redefining the main application class as we did above, since this will not work for more than one plugin. We could achieve a similar result by simply loading the Cat and Skycat Tcl packages and then adding the Data-Servers menubutton to the menubar:

```
# assumes these environment variables are defined
lappend auto_path $env(CAT_LIBRARY) $env(SKYCAT_LIBRARY)

# load the required packages
foreach pkg {Cat Skycat} {
    if {[catch {package require $pkg} msg]} {
        puts "error loading $pkg package: $msg"
        return
    }
}

proc Rtd_plugin {this} {
    set w [info namespace tail $this]
    set image [$w component image]

    # set X defaults
    cat::setXdefaults
    skycat::setXdefaults

    # add the catalog menu
    AstroCat::add_catalog_menu $this $image ::skycat::SkySearch 0
}
```

We would have to add a little more code to get everything just right (for example, loading the skycat Xdefaults cause the new menu button to have a different color than the others), but this should give you an idea of how it works.

3.3.2 Subclassing

If you are planning on making changes or additions to skycat, you may want to define subclasses of the existing Itcl or C++ classes and redefine selected methods to do what you want. This is the nor-

mal way of doing things within skycat and related packages and can be used for new applications as well as for skycat plugins.

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.

Another thing to watch out for, when redefining methods, is to keep them compatible with the original versions. It is best to call the parent class version of a method first, and then add your own code, unless you are sure that that is not what you want.

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. These procedures are well suited for use by a remote application, since they do not require any widget names. Of course this interface can also be used from within plugins or other Tcl code.

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 Class Member Procs for use with Tcl send

Name	Description
`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). To get the top level window for an image, you can use the command [winfo toplevel $skycatImage]. The main windows for skycat have names such as .skycat1, .skycat2, etc., although you should not use the hard coded names. Use `SkyCat::get_skycat_images` and check that the return value actually still exists with [winfo exists $skycatImage].

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.

See the `rtdimage` subcommand `mmap`, which supports this.

3.5 Skycat Public Interfaces

The plugin and remote interfaces open up many possibilities for extending skycat with new features, however they also open the door to backward compatibility problems in future releases. While we can't guarantee that future changes will not break existing plugins, we can define a *public interface* and try to avoid making changes that will not be backward compatible to it. This section attempts to define that interface.

The interface as a whole is described in the documentation for the packages used: *tclutil*, *astrotcl*, *rtd*, *cat* and *skycat*, and will not be repeated here. Instead, we simply list some basic guidelines for developing code that can more easily be supported in future releases.

3.5.1 Extended Tcl Commands

Any Tcl commands that are implemented in C or C++ are considered part of the public interface. This makes sense, since the private part is hidden in the C or C++ code. This includes central commands such as *rtdimage* (an extended Tk image type and also a Tcl command) and *astrocat*, the Tcl interface to the catalog library, and also any of the commands provided by the Tcl, Tk, TclX, Itcl, and BLT Toolkits.

3.5.2 C++ Classes

The public methods of any C++ classes are naturally part of the public interface.

Derived classes also have access to protected methods and member variables. You might need this access in order to derive a subclass of, for example, the RtdImage or Skycat classes, for adding new image features.

Derived classes may redefine base class methods, however if the derived class version does not also call the parent class version, we can not guarantee future compatibility.

3.5.3 C Libraries

The C interfaces used by skycat and rtd all have C++ class wrappers, so none of the C libraries belong to the "supported" public interface. Use the C++ classes instead. In this way, we can replace a C library, while still keeping the same C++ class interface. This is also very likely to happen, especially for classes, such as FitsIO and WCS.

3.5.4 Itcl Classes, Itk Widgets

The Itcl classes used by skycat use the keywords *public*, *private*, and *protected* to indicate which parts of the class belong to the public interface, which parts are private, and which parts may only be used in derived classes. Itcl is not as strict as C++ at enforcing these restrictions, since you can get around them, for example, by calling a method with the full "scope" path name. Still, this helps to document the public interface. The man pages for the classes also list the public and protected methods as well as the *Itk components*, which are the symbolic names of the internal widget components.

Any Itk components are public, so you can use the Itk *component* command to access internal widgets. Thus, it is okay to use "`$w component canvas`", but not "`$w.canvas`", which might change to something else in the future, such as "`$w.top.left.canvas`".

Any Itk options or public class variables are also public. You can access these values with the built-

in *cget* and *configure* methods.

Derived classes may need to also access the protected member variables and methods. For example, nearly all of the Itk widgets used by skycat are derived from either `TopLevelWidget` or `FrameWidget` and refer to their widget frames as `$w_`, which is a protected variable in the base class. Protected variables for widget names are okay to use in derived classes, however hard coded widget path names (even relative path names) should be avoided, if possible.

As with the C++ interface, derived classes may redefine base class methods, however if the derived class version does not also call the parent class version, we can not guarantee future compatibility.

Most of the Itcl classes used in skycat define Itk widgets that interact with the user. There are not normally many methods that are meant to be called from outside the class hierarchy. These include any callback methods for button presses or other events and any methods that add new menus or other widgets to the display or return the names of existing widgets. For example, the following methods, which are defined in the `TopLevelWidget` base class, are part of the public interface:

`start`	Member proc to start an application from the main class.
`busy`	Display the busy cursor over the window while evaluating some commands.
`add_menubar`	Add a menu bar to the window.
`add_menubutton`	Add a menubutton to the menu bar.
`add_menuitem`	Add a menu item to a menubutton.
`configure_menubutton`	Configure a menubutton.
`get_menu`	Get the widget name of a menu from the menubutton label.
`add_help_button`	Add a help button to the menubar.
`list_windows`	Return a list of top level windows, besides this one.
`hide_windows`	Hide all top level windows besides this one.
`add_short_help`	Add a short help window to the main window.
`quit`	Quit the window and exit the application, if there are no more windows.

3.5.5 Tcl Procs

There are a number of utility procs defined in the tclutil package (in tkutil.tcl and tclutil.tcl). These are part of the public interface and are not likely to change.

Also, all of the dialog procs in the tclutil package (in udialog.tcl) are public. These include procs such as `error_dialog`, `info_dialog`, and `warning_dialog` (see the plugin example earlier in this chapter).

4 Reference

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

4.1 COMMANDS

4.1.1 skycat(1)

NAME

skycat - A tool for displaying astronomical images and catalogs

SYNOPSIS

skycat ?filename? ?-option value ...?

OPTIONS

?-file filename?
Specify a FITS file to display. '-' means read the file from the standard input. The '-file' part is optional, so you can also simply specify a file name. Image compression and decompression is done automatically, based on the file name suffix: .gzfits or .gfits for GZIP compression, .hfits for H-compress, and .cfits for UNIX compression.

-cat bool
If bool is 1, include 'Data-Servers' menu in the menubar (This is the default). The 'Data-Servers' menu gives you access to the ESO Archive extensions for browsing astronomical catalogs, plotting objects in the image window and getting images over the network from the image servers, such as the Digitized Sky server.

-rtd bool
If bool is 1, include the Real-Time menu in the menubar (default is 0). The Real-Time menu gives you access to the VLT Real-Time Display features, such as camera control and rapid frames. To use these features, the rtdServer daemon must be running on the local host. A client application, linked with the Rtd image event library can then send images via shared memory to be displayed in rapid succession.

-float_panel bool
If the option value is 1, the skycat info panel is put in a separate popup window, leaving more space for the image window (The default is off).

-panel_layout <saoimage | reverse | default>
With this option you can change the order of the zoom and pan windows in the layout. The default layout is: zoom window on the left, info panel in the center and pan window right. If "-panel_layout saoimage" is specified, a layout similar to saoimage is used (info panel, pan window, zoom window). If "-panel_layout reverse" is specified, the order of the windows is the reverse of the default.

-remote bool
If "-remote 1" is specified and a skycat process is already running, the existing skycat process is sent a message and asked to open a new window and the new skycat process exits immediately. This has the advantage of sharing the image colormap and using fewer system resources, however it depends on being able to use the Tcl send mechanism. For security reasons, Tcl send will not work if you are using "xhost" based X security. You need to use X-auth security. See the "Tcl/Tk Tools" book from O'Reilly for more on this topic.

-min_scale n

```
-max_scale n
        Specify the min and max scale values for the Magnification
        menu.  Negative values shrink the image, positive values zoom
        in closer. The default values are -10 and 20.

-port portnum
        Specify a port number to use for the remote RTD socket
        interface. See the Rtd User's Guide for details on this socket
        based interface. By default, a port number is chosen
        automatically and written to the file ~/.rtd-remote.

-disp_image_icon bool
        If bool is 1 (default), display a miniature version of the
        image in the tool's icon window.

-default_cmap <cmap>
        Specify the default colormap. This should be one of the names
        listed in the 'Colors' popup window (default is 'real').

-default_itt <itt>
        Specify the default intensity transfer table. This should be
        one of the names listed in the 'Colors' popup window (default
        is 'ramp').

-colorramp_height <pixels>
        This option can be used to change the height of color bar (the
        widget at the bottom of the screen displaying the image
        colors).

-with_colorramp bool
        If bool is true, display the color bar (default).

-with_zoom_window bool
        If bool is true, display the zoom window (default).

-with_pan_window bool
        If bool is true, display the pan window (default).

-dozoom bool
        If bool is true, turn the zoom window on automatically
        (default).
```

DESCRIPTION

```
The ESO Skycat tool combines the image display capabilities of the RTD
(Real-Time Display) with a set of classes for accessing astronomical
catalogs locally and over the network using HTTP. The tool allows you
to view FITS images from files or from the Digitized Sky Survey (DSS).

MENU ITEMS:

File menu
        Open...
                Open and display a (FITS) image file.
        Reopen...
                Reload the image display after the image has changed
                on disk.
        Save as...
                Save the current image to a file.
        Save region as...
                Save a section of the current image to a file.
        Print...
                Print the current image to a file or printer.
```

```
        Clear
                Clear the image display.

        New Window
                Display up a new main window.
        Close
                Close the main window and exit if there are no more
                windows.

        Exit
                Exit the application.

View menu
        Colors...
                Display a window for manipulating the image colormap.
        Cut Levels...
                Display a window for manipulating the image cut levels.
        Cuts...
                Display a graph of pixel values along a line drawn
                interactively over the image.
        Pick Object...
                Select an object or star in the image and display
                statistics.
        Fits Header...
                Display the FITS header for the current image.
        Pixel Table...
                Display a table of pixel values surrounding the mouse
                cursor.
        Magnification
                Set the magnification factor of the image display.

        Hide Control Panel
                Toggle the visibility of the upper control panel
        Hide Popup Windows
                Toggle the visibility of the popup windows.


Graphics menu
        Toolbox
                Display the line graphics toolbox.
        Mode =>
                Select the drawing mode.
        Width =>
                Set the line width for drawing.
        Arrow =>
                Select the arrow mode for lines.
        ArrorShape =>
                Select the arrow shape for lines.
        Fill =>
                Select the fill color for drawing.
        Outline =>
                Select the outline color for drawing.
        Stipple =>
                Select the stipple pattern for filling objects.
        Font =>
                Select the font to use for labels.
        Smooth =>
                Set the smooth option for drawing polygons

        Clear =>
                Delete graphic objects.
        Delete =>
                Delete selected graphic objects.
```

```
        Hide Graphics
                Toggle the visibility of the image line graphics

    Data-Servers
        Catalogs =>
                Select a catalog from the menu.
        Image Servers =>
                Select an image server from the menu.
        Archives =>
                Select an archive from the menu.
        Local Catalogs =>
                Select a local catalog from the menu.


    Real-time menu (displayed when -rtd 1 is specified)
        Attach Camera
                Attach the real-time camera - start receiving images.
        Detach Camera
                Detach the real-time camera - stop receiving images.
        Set Camera...
                Set the real-time camera name.

        Rapid Frame
                Create a rapid frame by interactively drawing a
                rectangle on the image.

    Help menu
        About Skycat...
                Display a window with information about this Skycat
                version.
        Help...
                Display information about Skycat in netscape (if
                netscape is available).
```

ENVIRONMENT VARIABLES

```
    $SKYCAT_CONFIG
        If set, this is used as the URL to access the skycat
        configuration file, which contains the list of available
        catalogs and how to query them. By default, the configuration
        file is also searched for in $HOME/.skycat/skycat.cfg, and if
        that is not found, in the ESO default URL:
        http://archive.eso.org/skycat/skycat2.0.cfg.

    $SKYCAT_PLUGIN
        If set, this variable should be a colon separated list of
        files or directories containing skycat plugins. A skycat
        plugin is a Tcl script that defines a Tcl proc to be called
        for each instance of the main window. The script is sourced
        before any windows are created and can also load shared
        libraries dynamically to add new features.  See the Skycat
        User's Guide (ftp://ftp.archive.eso.org/pub/skycat/docs) for
        more information.
```

FILES

```
        http://archive.eso.org/skycat/skycat2.0.cfg - default
        configuration file.
```

SEE ALSO

```
      SkyCat(n), Skycat(3), rtd(1), RtdImage(3), AstroCat(n)


- - - - - -
Last change: 07 May 99
```

4.2 C++ CLASSES, C ROUTINES

4.2.1 Skycat(3)

NAME

Skycat - A C++ class that extends the rtdimage Tk image type

PARENT CLASS

RtdImage

SYNOPSIS

```
#include "Skycat.h"

class Skycat : public RtdImage {
...
public:
    Skycat(Tcl_Interp*, const char* instname, int argc, char** argv,
             Tk_ImageMaster master, const char* imageType,
             Tk_ConfigSpec* specs = (Tk_ConfigSpec*)NULL,
             RtdImageOptions* options = (RtdImageOptions*)NULL);

    virtual ~Skycat() {}

    virtual int call(const char* name, int len, int argc, char* argv[]);

    static int CreateImage(Tcl_Interp*, char *name, int argc, char **argv,
                    Tk_ImageType*, Tk_ImageMaster, ClientData*);

    static Skycat* getInstance(char* name);


    int get_compass(double x, double y, const char* xy_units,
                    double radius, const char* radius_units,
                    double ratio, double angle,
                    double& cx, double& cy, double& nx, double& ny,
                    double& ex, double& ey);

    int rotate_point(double& x, double& y, double cx, double cy, double angle);

    int make_label(ostream& os, const char* label, double x, double y,
                   const char* tags, const char* color,
                   const char* font = "-*-courier-medium-r-*-*-*-120-*-*-*-*-*-*");

    int draw_symbol(const char* shape,
                    double x, double y, const char* xy_units,
                    double radius, const char* radius_units,
                    const char* bg, const char* fg,
                    const char* symbol_tags,
                    double ratio = 1., double angle = 0.,
                    const char* label = NULL, const char* label_tags = NULL);

    int draw_circle(double x, double y, const char* xy_units,
                    double radius, const char* radius_units,
                    const char* bg, const char* fg,
                    const char* symbol_tags,
                    double ratio = 1., double angle = 0.,
                    const char* label = NULL, const char* label_tags = NULL);

    int draw_square(double x, double y, const char* xy_units,
                    double radius, const char* radius_units,
                    const char* bg, const char* fg,
                    const char* symbol_tags,
```

```
                    double ratio = 1., double angle = 0.,
                    const char* label = NULL, const char* label_tags = NULL);

    int draw_plus(double x, double y, const char* xy_units,
                  double radius, const char* radius_units,
                  const char* bg, const char* fg,
                  const char* symbol_tags,
                  double ratio = 1., double angle = 0.,
                  const char* label = NULL, const char* label_tags = NULL);

    int draw_cross(double x, double y, const char* xy_units,
                   double radius, const char* radius_units,
                   const char* bg, const char* fg,
                   const char* symbol_tags,
                   double ratio = 1., double angle = 0.,
                   const char* label = NULL, const char* label_tags = NULL);

    int draw_triangle(double x, double y, const char* xy_units,
                      double radius, const char* radius_units,
                      const char* bg, const char* fg,
                      const char* symbol_tags,
                      double ratio = 1., double angle = 0.,
                      const char* label = NULL, const char* label_tags = NULL);

    int draw_diamond(double x, double y, const char* xy_units,
                     double radius, const char* radius_units,
                     const char* bg, const char* fg,
                     const char* symbol_tags,
                     double ratio = 1., double angle = 0.,
                     const char* label = NULL, const char* label_tags = NULL);

    int draw_ellipse(double x, double y, const char* xy_units,
                     double radius, const char* radius_units,
                     const char* bg, const char* fg,
                     const char* symbol_tags,
                     double ratio = 1., double angle = 0.,
                     const char* label = NULL, const char* label_tags = NULL);

    int draw_compass(double x, double y, const char* xy_units,
                     double radius, const char* radius_units,
                     const char* bg, const char* fg,
                     const char* symbol_tags,
                     double ratio = 1., double angle = 0.,
                     const char* label = NULL, const char* label_tags = NULL);

    int draw_line(double x, double y, const char* xy_units,
                  double radius, const char* radius_units,
                  const char* bg, const char* fg,
                  const char* symbol_tags,
                  double ratio = 1., double angle = 0.,
                  const char* label = NULL, const char* label_tags = NULL);

    int draw_arrow(double x, double y, const char* xy_units,
                   double radius, const char* radius_units,
                   const char* bg, const char* fg,
                   const char* symbol_tags,
                   double ratio = 1., double angle = 0.,
                   const char* label = NULL, const char* label_tags = NULL);

    int symbolCmd(int argc, char* argv[]);
    int hduCmd(int argc, char* argv[]);

};
```

DESCRIPTION

Class Skycat extends the RtdImage C++ class by adding methods for drawing symbols in an image based in world or image coordinates and by adding support for FITS tables and multiple FITS HDUs. Since the RtdImage class implements the rtdimage Tk image type, this class adds features to the rtdimage command as well.

The symbol drawing methods defined here were originally implemented in Itcl and were later moved here to improve performance when plotting large numbers of symbols in an image.

CONSTRUCTOR

Skycat(interp, instname, argc, argv, master, imageType, specs, options)

Create a new skycat extended rtdimage object with the given name and arguments. The optional arguments "specs" and "options" allow derived classes to add new configuration options. See RtdImage(3) for hints on how to add new subcommand and options. The imageType argument is normally "rtdimage", but could be set to a different name, if you do not want to redefine the rtdimage type, but add a new one instead. The "master" argument is a Tk struct that contains pointers to the image handling routines.

METHODS

call(name, len, argc, argv)

This virtual method is defined at every level in the class hierarchy and is used to call a member function by specifying the name as a string. This is used to implement rtdimage subcommands by passing control from Tcl to C++. All of the methods that implement subcommands take the same arguments: argc and argv, the Tcl command line arguments.

CreateImage(interp, name, argc, argv, type, master, clientData)

This is the entry point from tcl to create a image.

getInstance(name)

Return a pointer to the Skycat class object for the given tcl rtdimage instance name, or NULL if the name is not an rtdimage.

get_compass(x, y, xy_units, radius, radius_units, ratio, angle, cx, cy, nx, ny, ex, ey)

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 arguments {cx cy nx ny ex ey} are for the 3 points center, north and east.

rotate_point(x, y, cx, cy, angle)

Rotate the point x,y around the center point cx,cy by the given angle in degrees.

make_label(os, label, x, y, tags, color, font)

Write a Tcl canvas command to the given stream to add a label to the image at the given canvas coordinates with the given

```
        label text, color and canvas tags.

draw_symbol(shape, x, y, xy_units, radius, radius_units, bg, fg,
            symbol_tags, ratio, angle, label, label_tags)
        Draw a symbol on the image with the given shape at the given
        coordinates (in the given x,y units), with the given radius
        (in radius_units), bg and fg color, canvas tags list, x/y
        ratio and rotation angle.

        shape may be one of "circle", "square", "plus", "cross",
        "triangle", "diamond", "ellipse", "compass", "line", "arrow".

        x and y are the coordinates in "xy_units", which is one of the
        units accepted by the Rtd commands (canvas, image, screen,
        "wcs $equinox", "deg $equinox").

        The radius value is interpreted in radius_units.

        bg and fg are X color names for the symbol (may be the same).

        symbol_tags should be a Tcl list of canvas tags for the
        symbol.

        ratio and angle are used to stretch/shrink and rotate the
        symbol.

        label is an optional text for a label to place near the
        symbol.

        label_tags should be a Tcl list of canvas tags for the label,
        or an empty or null string, if there is no label.

        Returns an error if the coordinates or part of the symbol are
        off the image.

        This method uses world coordinates, if available, for the
        rotation and orientation, for symbols that support it (i.e.:
        rotation is relative to WCS north).

draw_square(x, y, xy_units, radius, radius_units, bg, fg, symbol_tags,
            ratio, angle, label, label_tags)
draw_circle(...)
draw_plus(...)
draw_cross(...)
draw_triangle(...)
draw_diamond(...)
draw_ellipse(...)
draw_compass(...)
draw_line(...)
draw_arrow(...)
        These methods each draw one type of symbol. They are called by
        the draw_symbol method and have the same arguments (but no
        shape argument, of course).

symbolCmd(argc, argv)
        This method implements a the Tcl symbol subcommand (a new
        rtdimage subcommand added in this subclass):

        Usage:

          $instName symbol $shape $x $y $xy_units $radius $radius_units \
                 $bg $fg $symbol_tags ?$ratio $angle $label $label_tags?
```

```
        Draw a symbol on the image with the given shape at the given
        coordinates (in the given x,y units), with the given radius
        (in radius_units), bg and fg color, canvas tags list, x/y
        ratio and rotation angle.

        shape may be one of "circle", "square", "plus", "cross",
        "triangle", "diamond", "ellipse", "compass", "line", "arrow".

        x and y are the coordinates in "xy_units", which is one of the
        units accepted by the Rtd commands (canvas, image, screen,
        "wcs $equinox", "deg $equinox").

        The radius value is interpreted in radius_units.

        bg and fg are X color names for the symbol (may be the same).

        symbol_tags should be a Tcl list of canvas tags for the
        symbol.

        ratio and angle are optional and used to stretch/shrink and
        rotate the symbol. The default ratio is 1, default angle 0.

        label is an optional text for a label to place near the
        symbol.

        label_tags should be a Tcl list of canvas tags for the label,
        or an empty or null string, if there is no label.

        Returns an error if the coordinates or part of the symbol are
        off the image.

        Uses world coordinates, if available, for the rotation and
        orientation, for symbols that support it (i.e.: rotation is
        relative to WCS north).


hduCmd(argc, argv)
        This method implements the "hdu" subcommand, to access
        different FITS HDUs (header data units). Each HDU may be of
        type "image", "binary" table or "ascii" table.

         Usage: <path> hdu count
         or:    <path> hdu list
         or:    <path> hdu listheadings
         or:    <path> hdu type ?number?
         or:    <path> hdu headings ?$number?
         or:    <path> hdu get ?$number? ?$filename? ?$entry?
         or:    <path> hdu create $type $extname $headings $tform $data
         or:    <path> hdu delete $number
         or:    <path> hdu set $number
         or:    <path> hdu ?$number?

        If the "hdu count" subcommand is specified, it returns the
        number of HDUs in the current image.

        The "hdu type" subcommand returns the type of the current or
        given HDU as a string "ascii", "binary" or "image".

        If the "hdu list" subcommand is specified, it returns a Tcl
        list of FITS HDU information of the form:

          {{number type extname naxis naxis1 naxis2 naxis3 crpix1
            crpix2} ...}
```

```
Where:

  - number is the HDU number

  - type is the HDU type: one of "image", "binary table",
    "ascii table".

  - extname is the value of the EXTNAME keyword, if set

  - naxis, naxis1, naxis2, naxis3 match the FITS keyword
    values.

The "hdu listheadings" subcommand returns a list of the column
names returned by the "hdu list" subcommand. This can be used
to set the title of a table listing of the HDUs in a FITS
file.

The "hdu headings" subcommand returns a list of the column
names in the current or given FITS table.

The "hdu get" subcommand with no arguments returns the
contents of the current ASCII or binary table as a Tcl list (a
list of rows, where each row is a list of column values). If
the HDU number is given, the contents of the given HDU are
returned. If a filename argument is given, the FITS table is
written to the given file in the form of a local (tab
separated) catalog.  If optional "entry" argument is given, it
specifies the catalog config entry as a list of {{keyword
value} {keyword value} ...}, as defined in the catalog config
file (~/.skycat/skycat.cfg). The entry is written to the
header of the local catalog file and is used mainly to specify
plot symbol information for the catalog.

The "hdu create" command creates a new FITS table in the
current image file. $type maye be "ascii" for an ASCII table
or "binary" for a binary FITS table. The name of the table is
given by extname. The table headings and data correspond to
the catalog headings and data. The tform argument is a list of
FITS storage formats, one for each column, of the form {16A 2D
12A ...} (similar to FORTRAN formats, see the FITS docs).

The "hdu delete" command deletes the given HDU. The argument
is the HDU number. The other HDUs in the file following the
deleted one are moved to fill the gap.

If the "hdu" subcommand is specified with no arguments, it
returns the current HDU number. If a number argument is given,
the current HDU is set to that number.

The "hdu set" subcommand sets the current HDU to the given
number.  The keyword "set" is optional (see below).

An optional numerical argument may be passed to the "hdu"
subcommand, in which case the "current HDU" is set to the
given number.
```

SEE ALSO

SkyCat(n), SkySearch(3), RtdImage(3), TkImage(3), FitsIO(3)

```
- - - - - -
Last change: 07 May 99
```

4.2.2 SkySearch(3)

NAME

SkySearch - C++ class to extend the "astrocat" Tcl command

PARENT CLASS

TclAstroCat

SYNOPSIS

```
#include "SkySearch.h"

class SkySearch : public TclAstroCat {
...
public:
    SkySearch(Tcl_Interp* interp, const char* cmdname, const char* instname);

    static int astroCatCmd(ClientData, Tcl_Interp* interp, int argc, char* argv[]);

    virtual int imgplotCmd(int argc, char* argv[]);
};
```

DESCRIPTION

The SkySearch class extends the "astrocat" Tcl command (class TclAstroCat) with image plotting capabilities. The plot method was originally implemented as an Itcl method (see SkySearch(n)), but this turned out to be slow for large numbers of plot symbols. This class improves the plotting performace by making use of C++ symbol drawing methods defined in the Skycat class. This class adds a "plot" subcommand to the astrocat Tcl command.

PUBLIC METHODS

astroCatCmd(clientData, interp, argc, argv)

This is the entry point from Tcl. This static method is called when the astrocat command is used. It creates a new Tcl command with the same name as its first argument, that can be used to access the astrocat and skysearch subcommands. the object can be deleted with the "delete" subcommand.

imgplotCmd(argc, argv)

This method implements the "plot" subcommand:

usage: $instName imgplot $image ?$data? ?$equinox? ?$headings?

This subcommand is used to plot catalog objects on the skycat image and was reimplemented here in C++ code to improve performance for large complicated catalogs.

$image is the name of the image object ("rtdimage" object, implemented by the RtdImage C++ class and extended by the Skycat C++ class).

If $data is specified, it should be a Tcl list of rows to be plotted, in the format returned by the query command.

If $equinox is specified, it is the equinox of the ra and dec columns in the data (the first 3 columns are assumed to be id, ra and dec, unless otherwise defined in the catalog config entry or header).

If $headings is given, it is used as a Tcl list of column headings. Otherwise the catalog headings are used, if there is a current catalog.

Note: normally you will need to specify all the arguments, since the querries are done in the background (See AstroCat(n) (cat package) and Batch(n) (tclutil package)). The information for the previous query is lost when the background process exits. This might change if queries were done using threads or if the background/interrupt handling were done in the C++ code rather than in the Tcl code, as it is done now.

SEE ALSO

astrocat(n), AstroCat(n), SkyCat(n), SkySearch(n), RtdImage(3), TkImage(3), TclCommand(3), Batch(n)

- - - - - -

Last change: 07 May 99

4.2.3 ITCL CLASSES

4.2.4 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.

ITK COMPONENTS

image

SkyCatCtrl(n) widget (derived from RtdImageCtrl), for displaying image and control panel.

WIDGET OPTIONS

-cat

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

-catalog

Specify a catalog (may be a local file) to load on startup.

-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.

-remote

If another skycat application is running on this display, use it rather than this process (saves memory and colors in the colormap).

-rtd

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

PUBLIC METHODS

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

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

PROTECTED METHODS

```
add_go_menu {}
       Add a "Go" menu with shortcuts to view images previously viewed.

add_graphics_save_menu_item {}
       Add a menu item to the Graphics menu for saving the line graphics
       in a FITS table in the image.

add_help_menu {}
       Add a menubutton with help items.

add_olaf_menu {}
       Add a menubutton with OLAF items.

add_realtime_menu {}
       Add the Real-time menubutton and menu, if the -rtd option was
       given.

add_view_hdu_menu_item {}
       Add a menu item to the View menu for selecting the current HDU.

init {}
       Called after the options have been evaluated.

load_toplevel_geometry {}
       Restore the position of the top level window from the previous
       session.

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).

resize {w h}
       Called when the main window is resized:  Check the geometry to
       make sure it fits on the screen.

save_toplevel_geometry {}
       Save the position of the top level window so we can reload it the
       next time.

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.
```

```
        start_remote {}
               This method is called for the -remote option. If another skycat is
               running, use it to display the image and exit, otherwise do it in
               this process.  Try Tk send, and if that fails, fall back on the
               RTD socket interface.
```

PROTECTED VARIABLES

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

COMMON CLASS VARIABLES

```
        toplevel_geometry_
               Name of the file used to save the positions of the top level
               windows.
```

SEE ALSO

```
        Rtd(n)
```

```
- - - - - -
Last change: 07 May 99
```

4.2.5 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.

WIDGET OPTIONS

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

PUBLIC METHODS

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

add_history {filename}
: Add the current image to the history catalog under the given filename. The current FITS header is used to extract information about the image to put in the catalog. The file basename is assumed to be the unique id.

apply_history {filename}
: Check if the given filename is in the history catalog, and if so, apply the cut levels and color settings for the file.

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

convert_coords {coords from_units to_units}
: Convert the given coordinates from $from_units to $to_units and return the result. $coords may be a list of an even number of values {x1 y1 x2 y2 x3 y3 ...}. The result is the same list, converted to the output coordinates.

display_fits_hdus {}
: Display a popup window listing the HDUs in the current image, if any.

draw_symbol {shape x y xy_units radius radius_units bg fg symbol_tags {ratio 1} {angle 0} {label ""} {label_tags ""}}
: Draw a symbol on the image with the given shape at the given

```
        coordinates (in the given x,y units), with the given radius (in
        radius_units), bg and fg color, canvas tags list, x/y ratio and
        rotation angle.

        shape may be one of "circle", "square", "plus", "cross",
        "triangle", "diamond", "ellipse", "compass", "line", "arrow".

        x and y are the coordinates in "xy_units", which is one of the
        units accepted by the Rtd commands (canvas, image, screen, "wcs
        $equinox", "deg $equinox").

        The radius value is interpreted in radius_units.

        bg and fg are X color names for the symbol (may be the same).

        symbol_tags should be a Tcl list of canvas tags for the symbol.

        ratio and angle are optional and used to stretch/shrink and rotate
        the symbol. The default ratio is 1, default angle 0.

        label is an optional text for a label to place near the symbol.

        label_tags should be a Tcl list of canvas tags for the label, or
        an empty or null string, if there is no label.

        Returns an error if the coordinates or part of the symbol are off
        the image.

        Uses world coordinates, if available, for the rotation and
        orientation, for symbols that support it (i.e.: rotation is
        relative to WCS north).

forward_image {}
        Go forward again to the next image.

load_graphics_from_image {}
        Check if there is a FITS table with the same name as the current
        image extension, but with ".GRAPHICS" appended.  If found, restore
        the previously saved line graphics from the table.  This method is
        called automatically when a new image extension is loaded.

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.

previous_image {}
        Go back to the previous image.

save_as {{dir "."} {pattern "*"} {x0 ""} {y0 ""} {x1 ""} {y1 ""}}
        Save the current image or a section of the current image to a file
        in FITS format chosen from a file name dialog. If dir and pattern
        are specified, they are used as defaults for the file selection
        dialog.  If x0, y0, x1 and y1 are specified (canvas coordinates),
        then a section of the image is saved.

        The return value is the name of the new file, if any, or an empty
        string.  (redefined from parent class to set filename_, used in
        check_save).

save_graphics_with_image {}
        Save the current line graphics in a FITS binary table in the
        image.  The table has 3 columns: "type", "coords", and "config".
        "type" gives the shape and is one of the Tk canvas item types.
```

```
        "coords" is a list of coordinates for the item.  "config" is a Tcl
        list of configuration options for the item.  There can be one
        graphics table for each image extension. For each image extension,
        the graphics table is called "${extname}.GRAPHICS".

    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}
        This method is called when the user has selected an area of the
        image.  The results are in canvas coordinates, clipped to the 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_fits_hdus {}
        Update the popup window listing the HDUs in the current image.

    update_history_menu {w m}
        Update the given menu with image history items. $w is the
        TopLevelWidget containing the menubar.
```

PROTECTED METHODS

```
    check_save {}
        If the current image is not saved in a file (came from a server,
        ...) check if it should be saved before loading a new image. Then,
        if the file exists, add image info for it to the history catalog.

    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.

    init {}
        This method is called from the base class (TopLevelWidget) after
        all the options have been evaluated.

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

    new_image_cmd {}
        This method is called by the image code whenever a new image is
        loaded.  It is redefined here to handle multiple HDUs in FITS
        files.
```

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

PROTECTED VARIABLES

```
        back_list_
               Used for the Go=>Back/Forward menu itemes.

        filename_
               The name of the image file, if any.

        pi_
               Const PI.

        rad_
               Const PI/180.

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

SEE ALSO

```
        RtdImageCtrl(n)
```

```
- - - - - -
Last change: 07 May 99
```

4.2.6 SkyCatHduChooser(n)

NAME

SkyCatHduChooser - Itcl widget for displaying FITS extensions

NAMESPACE

skycat

PARENT CLASS

util::TopLevelWidget

SYNOPSIS

SkyCatHduChooser <path> ?options?

DESCRIPTION

This class defines a widget for displaying the HDUs in the current FITS image. The user can select a FITS table or image extension to display by clicking on an entry the list or on one of the small images displayed in a table.

ITK COMPONENTS

buttons
: Button frame at bottom of window.

image_table
: Frame (BLT table) used to display images in FITS extensions.

table
: TableList(n) widget for displaying the list of HDUs.

WIDGET OPTIONS

-catinfo
: Name of the FITS table containing catalog config info.

-image
: Target SkyCatCtrl itcl class object.

PUBLIC METHODS

show_hdu_list {}
: Update the list of HDUs and the image displays, if needed.

PROTECTED METHODS

add_image_bindings {im hdu name}
: Add bindings to the given RtdImage itcl class object and set it to display the given HDU when clicked on. The image's extension name is also given.

delete_hdu {}
: Select an HDU to display. This makes it the current HDU (XXX TO BE DONE: should also delete entry from $catinfo table).

display_fits_table {name hdu}
: Display the current FITS table.

```
get_config_entry_from_fits_table {extname filename}
       Return the catalog config entry for the named FITS table, if
       available, or a default entry. If the current FITS file contains
       an HDU named $catinfo, with an entry for the named catalog
       ($extname), then extract and return that entry as a Tcl keyed
       list.

init {}
       This method is called after the options have been evaluated.

make_buttons {}
       Add a row of buttons at bottom.

make_image_table {}
       Make a subwindow for displaying miniature versions of image
       extensions.

make_short_help {}
       Add a short help window.

make_table {}
       Make the table component for displaying the HDU info.

resize {im new_width new_height}
       This method is called when the image window is resized.  The
       rtdImage widget and the new width and height are given.

select_hdu {}
       This method is called when a line in the HDU list is selected.
       Update the states of the buttons depending on the selection.

select_image_hdu {hdu}
       This method is called when the user clicks on an image HDU icon.
       Display the selected image and disable the delete button.

set_hdu {}
       Set the HDU to display. Makes the currently selected HDU the
       current HDU.
```

PROTECTED VARIABLES

```
ext_
       Array(HDUIndex,keyword) of image keyword and widget info.

filename_
       Name of image file.

image_
       Internal rtdimage object.

imagetab_
       Table displaying image extensions.

num_images_
       Number of image HDUs in the current FITS file.

table_
       Table displaying the HDUs.
```

COMMON CLASS VARIABLES

```
astrocat_
```

C++ astrocat object use here to access catalog entries.

SEE ALSO

TopLevelWidget(n)

- - - - - -
Last change: 07 May 99

4.2.7 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.

WIDGET OPTIONS

-skycat
Name of SkyCat itcl widget.

PUBLIC METHODS

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.

get_image_center_width_height {wcs_flag}
Return a list of values indicating the center coordinates and the width and height of the current image. If wcs_flag is 1, the return list is {ra dec equinox width height}, width and height in arcmin, otherwise {x y width height} 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. If we are dealing with an image server, the radius value is replaced by width and height, i.e.: {ra dec equinox width height}, where width and height are in

```
            arcmin for wcs or pixels otherwise.  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 search values to the center position and radius of
            the image, (for catalogs) of width and height of image (for image
            servers).
```

PROTECTED METHODS

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

PROTECTED VARIABLES

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

SEE ALSO

```
        AstroQuery(n)



- - - - - -
Last change: 07 May 99
```

4.2.8 SkyQueryResult(n)

NAME

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

NAMESPACE

skycat

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.

WIDGET OPTIONS

-catinfo
: Name of the FITS table containing catalog config info.

-skycat
: Name of SkyCatCtrl itcl widget.

PUBLIC 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).

save_with_image {entry}
: Save the current data as a FITS table in the current image file. The argument is the catalog config entry.

PROTECTED METHODS

save_config_info_to_fits_table {extname entry}
: Save the given catalog config entry in a FITS table with the name $catinfo. The hdu arg gives the HDU number of the $catinfo table, or 0 if it does not exist.

SEE ALSO

QueryResult(n)

```
- - - - - -
Last change: 07 May 99
```

4.2.9 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.

ITK COMPONENTS

results
: SkyQueryResult(n) widget to display the results of a catalog query.

searchopts
: SkyQuery(n) widget (derived from AstroQuery(n)) for displaying search options.

WIDGET OPTIONS

-canvasfont
: Font used in canvas to mark objects.

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

PUBLIC METHODS

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 {}
: Delete any graphic objects in the image belonging to this 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.

filter_query_results {}
       Remove any items in the query result list that have not been
       plotted because they were not in the image (circular
       search/rectangualr image).

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.

get_table_row {id}
       Return the table row index corresponding the given symbol canvas
       id.  Note: The plot subcommand in SkySearch.C adds a canvas tag
       "row#$rownum" that we can use here.  Also: cat$id is first tag in
       the tag list for each object.

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).

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"}.

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.

    save_with_image {}
        Save the current data as a FITS table in the current image file.

    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_symbol {id toggle {rownum -1}}
        Select a symbol, given the canvas id and optional row number in
        the table listing. 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 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).

    init {}
        Called after options have been evaluated.

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

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

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

PROTECTED VARIABLES

```
    canvas_
        Canvas window containing main image.

    draw_
        CanvasDraw object for drawing on image.

    image_
        Internal rtdimage image for main image.

    label_tag_
        Canvas tag used to identify all labels for this instance.
```

```
object_tag_
       Canvas tag used to identify all objects for this instance.

skycat_
       SkyCatCtrl widget instance.

symbols_
       Array containing supported symbol names.

tag_
       Canvas tag used to identify all symbols for this instance.

wcs_
       Name of wcs object for converting between hh:mm:ss and double
       deg.
```

COMMON CLASS VARIABLES

```
history_catalog_
       Name of the history catalog.

history_cols_
       List of columns in the history catalog.
```

SEE ALSO

```
AstroCat(n)
```

```
- - - - - -
Last change: 07 May 99
```

5 Installation

The configure script and Makefile in this directory can be used to build any or all of the following packages:

- tclutil - Tcl and C++ Utilities Package
- astrotcl - Astronomical Tcl and C++ Utilities
- rtd - Real-Time Display
- cat - Catalog library
- et - Patched version of Embedded Tk
- skycat - The ESO Skycat Tool
- plugins/gaia - A skycat plugin that adds photometry related features

5.1 Requirements

One of the following Tcl environments should be installed:

For Tcl7.6:

- itcl2.2 - [Incr Tcl], includes tcl7.6, tk4.2
- BLT2.1 - BLT toolkit, graphs and other widgets
- tclX7.6.0 - Extended Tcl

Or for Tcl8.0 (or newer, TclPro also supported - see below):

- Tcl8.0.3 - Tcl Shell
- Tk8.0.3 - Tk X Toolkit
- itcl3.0.1 - [Incr Tcl]
- BLT2.4f - BLT toolkit, graphs and other widgets
- tclX8.0.3 - Extended Tcl

These packages are available from the TCL archives.

- http://www.tcltk.com/ for Itcl.
- http://www.NeoSoft.com/tcl/ for TclX and other contributed Tcl software
- http://www.scriptics.com/tclpro/ for Tcl8.x, TclPro and the latest releases.

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).

NOTE: If you are building with *Scriptics TclPro*, you also need a *normal* Tcl8.x installation (installed from the source). The Skycat configure scripts depend on the `$prefix/lib/tclConfig.sh` and related files being installed. These files are not currently part of the TclPro installation. Also, the names of some of the libraries are different in TclPro. Skycat uses the "prowrap" application and the Scriptics libraries to build a single binary version for tcl8.x, if they are found.

NOTE: In order to build a "single binary" version of skycat correctly, the static ".a" Tcl/Tk libraries must be installed.

5.2 Building the Software

Optionally define the environment variable TCLTK_ROOT to point to the top level install directory for Tcl (for example, /usr/local):

```
% setenv TCLTK_ROOT /usr/local              # for csh, tcsh
```

or:

```
% TCLTK_ROOT=/usr/local; export TCLTK_ROOT  # for sh, ksh, bash
```

To compile the sources, type the following from the top level directory:

```
% configure -prefix $INSTALLDIR --with-gcc
% make all
% make install
```

$INSTALLDIR should be the name of the top level directory (such as /usr/local or $PWD/install) in which to install the software. The default install directory is /usr/local.

The configure option --with-gcc says to use GNU gcc and g++. If you want to use CC and cc instead, specify the --with-cc option:

```
% configure -prefix $INSTALLDIR --with-cc
```

If you prefer using shared libraries and loadable Tcl modules add the option:

```
--enable-shared
```

Note that if you are using g++, you must also have libg++ compiled as a shared library for this to work (libg++-2.7.2.x also has the "--enable-shared" option). Also, if you use --enable-shared, both static and shared libraries will be generated, otherwise only the static versions.

5.3 If you run into Problems...

The top level configure script actually runs "configure" in each of the package subdirectories found (tclutil, astrotcl, rtd, cat, et, and skycat). If there are errors, you might try to run configure and make manually in a subdirectory. As an alternative to configure, you can also use one of the scripts config.debug, config.shared, etc, which do the same thing, but with different options. You might want to copy one of the scripts and edit it first, to set the install directory or other options. See the */README and */INSTALL files for more information.

Solaris cc:

On Solaris, make sure the correct C compiler is being used (not /usr/ucb/cc). We have tested with Solaris SunPRO CC/cc and gcc (2.7.2.3).

HP-UX message "out of memory":

On HP-UX: If you get the message "not enough memory" when running skycat with shared libraries, it doesn't necessarily mean that you have to buy more memory. This is the informative message that HP prints when there is an undefined symbol in a shared library....

One way to find out what the symbol is, is to compile and use this little program with the C++ compiler: (I got this from Peter Biereichel):

```
#include <dl.h>
#include <errno.h>
#include <stdio.h>
main(int argc, char *argv[])
{
  shl_t handle;
  handle = shl_load(argv[1], BIND_IMMEDIATE | BIND_VERBOSE, 0L);
  if (handle == 0)
    {
    printf("shl_load failed %s\n", argv[1]);
    perror("");
    }
```

```
      else
        printf("shl_load ok\n");
    }
```

For example:

```
g++ shlload.C -o g++ shlload
% shlload librtd.sl
% shlload libcat.sl
```

Shlload should report the name of the undefined symbol in the given shared library.

Please report any problems to me:

Allan Brighton

abrighto@eso.org