
# User and Programming Guide

Version 0.9
30th May 2006

# Contents

# 1 Short overview

### What's all about?

iceWing, an **I**ntegrated **C**ommunication **E**nvironment **W**hich **I**s **N**ot **G**esten (This is a reference to an older program, the predecessor of iceWing.) is a graphical plugin shell. It is optimized for image processing and vision system development. But its use in totally different fields of research is as well possible, e.g. audio-stream processing. Predefined or self-written plugins operate hierarchically on data provided by other plugins and can also generate new data-streams, allowing flexible communication and interaction between these plugins. An important predefined plugin is the grabbing plugin, which can read images from the disk in various image formats, from grabber-hardware, e.g. V4L2-devices or FireWire, and also from external, network wide processes via DACS streams.

Not being only a batch-plugin-shell iceWing is also a highly customizable GUI platform for the plugins: It has a list of given GUI-elements and allows the plugins to simply make use of them. So plugins can show the user their current status and can let the user change parameters on the fly. Moreover methods of easy visualization of plugin results are available. The plugins can open any number of windows and display in these windows any data in a graphical form.

### Where to get iceWing?

The Homepage for iceWing can be found at

http://icewing.sourceforge.net

There are two mailing lists dedicated to iceWing user and development discussion. More infos about these mailing lists can be found at

http://sourceforge.net/mail/?group_id=151242

Additionally, the main author can be reached at

floemker@techfak.uni-bielefeld.de

### Who did iceWing?

Program:

Frank Lömker, floemker@techfak.uni-bielefeld.de

This documentation:

Frank Lömker, floemker@techfak.uni-bielefeld.de

Initial installation and user guide: Andreas Hüwel, andreas.huewel@gmx.de

Programming guide translation of V0.8.1: Ilker Savas

## License

iceWing is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

iceWing is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA.

# 2 Installation

## 2.1 Requirements

**Libraries needed** iceWing is targeted at Unix-like operating systems and was actually tested on Linux, Alpha/True64, Solaris, and Mac OSX. For compiling and using it some programs and libraries must be installed:

- Basic commands like bunzip2, tar, make, makedepend, C compiler (probably gcc) - nothing that a normal Unix installation does not provide.
- X11
- GTK V1.2, tested for version >=1.2.5, <2.0

Additional *optional* libraries extend the functionality of iceWing:

- gdk-pixbuf: Loading additional image-formats, tested for version >= 0.8.
- libjpeg: JPEG- and AVI-MJPEG-saving.
- libpng: PNG-saving and loading of PNG images with a depth of 16 bit.
- libz – used by the above optional PNG saving package.
- libraw1394, libdc1394: Grabbing from firewire cameras supporting the "Digital Camera Specification".
- unicap: Grabbing from different hardware devices.
- FFmpeg: Loading a wide variety of video formats. At least a version from the 18. of November, 2005 or later is needed, version 0.4.9-pre1 is too old.

*Attention:* You will always need the development files, especially header files, for the different libraries. We will show for RedHat packages (rpm) how to verify that everything is installed. Debian packages are quite similar. You can easily check with the following shell command

```
> rpm -qa |grep gtk
```

which packages are installed and which version that libraries have. iceWing needs

the headers, too, and the given developer packets provide them. If you compile your own libraries from sources, you have to add the headers into the default-include path, so iceWing will find them. After verifying all this, you only need to get the iceWing tarball "icewing-version.tar.bz2".

**Further stuff** You can use DACS for your own iceWing plugins to communicate with other external net wide processes. It works quite similar to Corba. You do *not* need DACS for any iceWing internal communication (iceWing - plugin, plugin - plugin) or to e.g. access files of the system. Additionally, iceWing has integrated support for reading/publishing images to/from other programs and for remote control of plugin sliders via DACS. If DACS is not available, these features can be disabled during compiling.

More information about DACS can be found in the Web under this address:

http://www.techfak.uni-bielefeld.de/ags/ni/projects/dacs/

Many details of DACS are also in the dissertation of Niels Jungclaus [Jun98].

## 2.2 Installation

Lets assume you have the tarball in "./". Then simply

```
> bunzip2 -c icewing-version.tar.bz2 | tar -xv
> cd icewing-version
```

where version is the particular iceWing version number you are using.

**Adopt the Makefile to your system** Now you have to edit the Makefile to adopt it to your system and in- or exclude the support for additional packages. Mostly it will be uncommenting some few lines that you probably will not need on your system.

**PREFIX:** It must be set to your installation place, for example "/usr/local"

**FLAGS:** Eventually you must adopt the compiler flags to your hardware (E.g.: You might not have a Pentium/Athlon processor?)

**WITH_AV, WITH_FIRE, WITH_UNICAP:** If included, iceWing will have support for grabbing images from a camera. On Alpha/True64 systems an external library – the AVlib – is needed. This library is written by the AG-AI and supports grabbing of images via the Multi Media Extension MME. Composit and SVideo cameras are supported. The value given to WITH_AV specifies the location of the AVlib.

On Linux support for image grabbing is directly integrated into iceWing. This gets activated if WITH_AV is defined. Composite and SVideo cameras are supported with the help of the "Video for Linux Two" interface (see "http://linux.bytesex.org/v4l2/"). If additionally WITH_FIRE is defined FireWire (IEEE1394) cameras supporting the "Digital Camera Specification" are supported as well. Here, the external libraries libraw1394.a and libdc1394_control.a are needed (see "http://www.linux1394.org/" and "http://sourceforge.net/projects/libdc1394/"). WITH_FIRE gives the location of these two libraries.

By defining WITH_UNICAP additionally to WITH_AV support for the unicap library gets integrated. Unicap provides a uniform API for different kinds of video capture devices, e.g. IEEE1394, Video for Linux, and some other. See "http://www.datafloater.de/unicap/" for details about this library. WITH_UNICAP gives the location of the unicap library.

**WITH_DACS:** If included, iceWing will have support for DACS, which is like Corba for network wide interprocess communication. Especially, iceWing will be able to send and receive images and widget configurations via DACS communication channels.

**WITH_GPB:** Enables loading of images of other formats than "PNM" and "PNG". For this, the gdk-pixbuf library is used.

**WITH_JPG, WITH_PNG, WITH_ZLIB:** Enables further image saving formats. When all are commented, iceWing can only save the various "PNM" formats. Moreover, iceWing will not be able to load 16 bit PNG images and movie files in AVI containers encoded with the motion jpeg codec.

**WITH_FFMPEG:** Enables loading of a wide variety of video formats by using the FFmpeg library. See "http://www.ffmpeg.org" for further details about this library. Without the FFmpeg library, iceWing can only handle AVI files with the motion jpeg codec if WITH_JPG is enabled.

During development of iceWing a CVS snapshot of the FFmpeg library from the 18. of November, 2005 and a SVN snapshot from the 26. of May, 2006 were used. You can find the FFmpeg SVN snapshot version from May at the iceWing homepage. The last release version at that date, version 0.4.9-pre1, is too old and will *not* work. WITH_FFMPEG gives the location of the extracted FFmpeg archive, where the compiled sources of the library are expected.

So to add FFmpeg support, extract the FFmpeg archive, change to the extracted directory, configure the library, and compile it. Installing the library is not necessary. iceWing will use the include files and the static libraries directly from this source directory.

**"make" it all** After adopting the Makefile, you can build the installation files from the sources:

```
> cd {wherever your sources are}
> make depend
> make
```

You must now login as admin/root, if $(PREFIX) is not writable for the current user. The Makefile expects the directory that is named in $(PREFIX) to be existing - if not:

```
> mkdir $(PREFIX)
```

The installation is now fully prepared. Now the time for installation has come!

```
> cd {wherever your sources are}
> make install
```

To all the cautious admins: You eventually want to check the groups and rights of the new dirs now.

*That's it!* For further details about what was done, just have insight into the installation log file, which is located at "$(PREFIX)/share/log/iceWing.log".

If the new "icewing"-executable is in the execute-path, you can right away start "icewing", the executable (see section 3.1 Quicktour). If you are interested in what is where in this installation, have a look at section 6.1 Filesystem.

**Troubleshooting** Check the output for errors, also the installation-logfile. You are sure, that you installed the needed libraries properly, but

```
> make
```

produced errors like:

/bin/sh: gdk-pixbuf-config: command not found

Let's again assume you used the RedHat package named "gdk-pixbuf-devel" (well, Debian packages are treated similar). Then check via

```
> rpm -ql gdk-pixbuf-devel |grep gdk-pixbuf-config
```

or with *full* search

```
> find / -name gdk-pixbuf-config
```

*where* the needed packed-files got installed. Maybe they are simply not in default-execute path!?

Assume you find "gdk-pixbuf-config" in "/opt/gnome/bin/gdk-pixbuf-config". With the bash-shell you can replace the compiling

```
> make
```

command by:

```
> PATH=$PATH:/opt/gnome/bin make
```

Or you change the Makefile entry “GDK_PIXBUF = gdk-pixbuf-config” to “GDK_PIXBUF = /opt/gnome/bin/gdk-pixbuf-config”.

# User guide

# 3 Introduction

## 3.1 Quick "on the fly" tour

In this Quicktour you will get a short overview over iceWing and experiment with it's GUI. For this you will start a session with a small video as input and then play a bit with a demo plugin. With the sources of iceWing you got a small video, which you can use during this tour. It can be found in the docs directory under the name "quicktour.avi". Alternatively, you can choose any other image or video (of a format, that iceWing supports).

**Compiling the plugin** During installation of iceWing the demo plugin was not compiled and installed. So let's do that now.

```
> cd {wherever your iceWing-sources are}
> cd plugins/demo
```

Now make sure that you have $(PREFIX)/bin in the default-execute path or, alternatively, change the Makefile in the current directory. The Makefile needs to find "icewing-config", which was installed in $(PREFIX)/bin. Then simply

```
> make depend
> make
> make install
```

This compiles the demo plugin and installs it under the name libdemo.so in the directory $(PREFIX)/lib/iceWing, where iceWing can find it automatically.

**The quick tour** You tell by command line parameters from where iceWing will take it's images. There are several kinds of sources:

- file-loading from disk (like you do for this session)
- grabbing from camera
  (only available, if iceWing was compiled with AVlib support)
- DACS streams
  (only available, if iceWing was compiled with DACS support)

So let's jump in and start iceWing with the example plugin "demo": You launch iceWing in the shell-command line with (as one single line, after changing to the right directory)

```
> cd {wherever your iceWing-sources are}
> icewing -sp docs/quicktour.avi -l demo
```

After this, iceWing opens it's main window. At the same time on the starting console iceWing and the plugin write constantly their status.

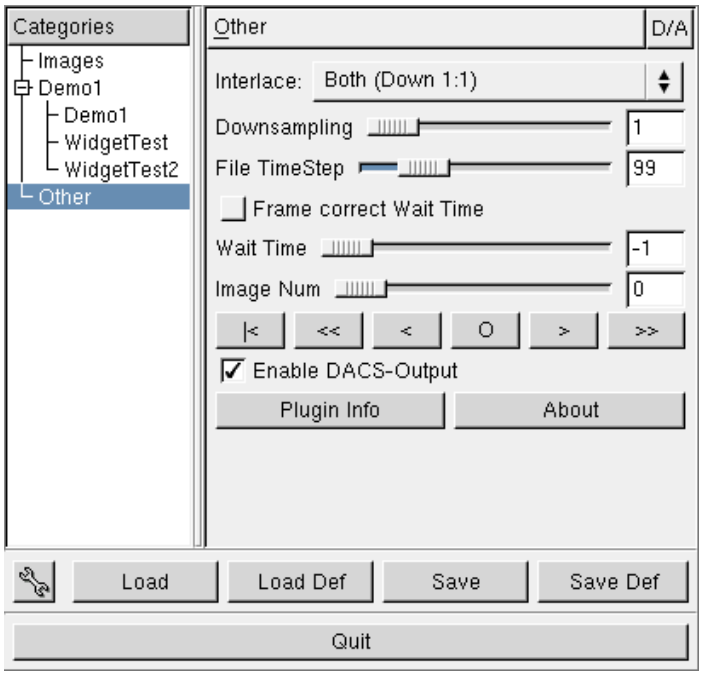
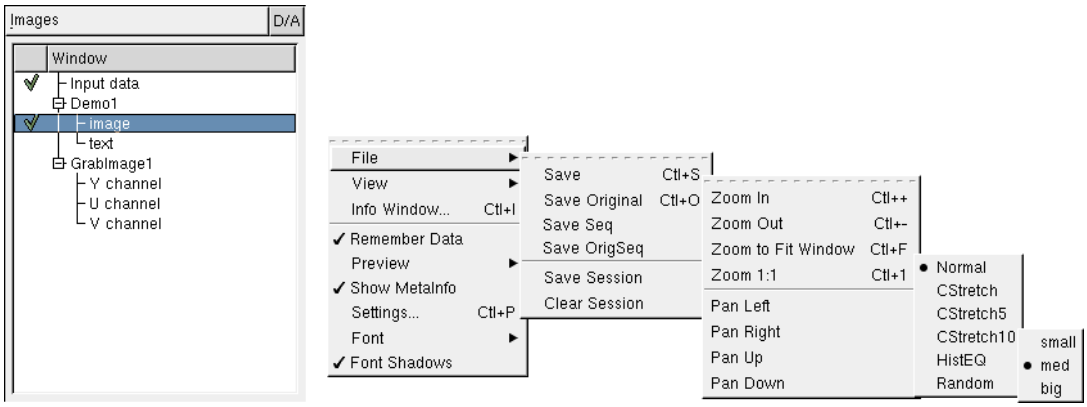



Figure 3.1: The main window of iceWing.

In figure 3.1 you see the iceWing main GUI, so lets explore some things. On the left side you see the "Categories" area, which has now "Images" and "Other" and some plugin pages.

**The category "Other"** Click on "Other" to activate this special page. You can see several sliders on the right side. If you have during a session as data source not a video stream but one single image, "Image Num" will always be set to 0. None the less iceWing has still the option to (re)acquire that single image on a timely basis: The slider "Wait Time" sets the delay (in ms), after that this (for video-streams: next) image shall be acquired automatically. If set to -1, iceWing waits until you click manually the read buttons below. So now set the wait time to 200ms, and you see the mass of console output greatly reduced.

**The plugin pages** You also launched with the command line parameter "-l" the plugin "demo", which is inside libdemo.so. Every plugin can generate none or more page-entries on the left "Category" side. By selecting the pages you can view plugin status or change the plugin parameters on the right side. Page "Demo1" has some options, that directly affect the plugin. "Demo1 WidgetTest" and "Demo1 WidgetTest2" not *do* anything useful. But they show you all the available GUI-elements of iceWing, that can also be easily used by your own plugins.




(a) (b)

Figure 3.2: (a) The page "Images". (b) The context menu of window "Demo1 image" with all it's submenus.

**How to display plugin results?** With click on category "Images" you see on the page on the right side several "Window" entries coming up (see figure 3.2(a)). They show you all windows that the plugins wish to display. Double-click (de)activates the window of that entry and you can see what the plugins are doing.

You see one window "Input data" for the current input image, one separate window for each YUV channel of the source image and two windows for the output of the plugin "Demo1".

Lets now activate the first demo window called "Demo1.image". Inside this new window press the right mouse button and verify that "Remember Data" is active (see figure 3.2(b)). This flag makes all coming zoom commands operate on the currently loaded image data, not only on *new* acquired/grabbed images. If you deselect this option, and e.g. try to zoom around, the zoom commands will still be remembered - but the window content will still display (unchanged until next image reading) the last acquired image without any of the zoom commands visible. When you set "Wait Time" = -1, you see the "Remember Data" flag's functionality very clear: When

"Remember Data" is inactive, only reloading the image source will refresh the window content and show your until then done zooms.

**Zooming the window content** Inside the window shift/crtl/alt-keys plus middle mouse button zoom in/out/reset the image. When you just hold the middle button and move the mouse around, you can pan inside the image, if you have zoomed into it. Additionally, you can use the scroll whell and the shift and ctrl-keys to pan and zoom in the window, or, as a third option, the "View" context menu. If you have problems with the zooming, look at the section 5.2.6 "Panning/Zooming the image windows".

Now play a bit with the "Remember Data" flag and "Wait Time" and zooming to work it out.

**Manipulating image colors** In the context menu (right mouse button) you can see the "Preview->Random" option. This tool randomizes the color assignments. This can be useful to verify the output of a color separating plugin, because very similar/neighbored colors now get very distinguishable.

**Saving the zoomed part into a separate file** After you played around a bit, you can safe the frame content into a file with context menu entry "File/Save". The picture will be the window content and it's name be "Gsnap_0.ppm". If you wish a different name/format, you can change this (and more) by clicking the wrench icon of the main Window (see figure 3.1) to get the preferences window. There you can change the image name to e.g. "quicktour_zoomed_%d.ppm" (the %d is a placeholder for the image number).

**More about this plugin via "plugin info"** On the page "Other" in the main window (figure 3.1) click "Plugin Info". Here you see how the plugin "Demo1" is integrated into the iceWing system, You see the sheet and the only two active plugins are "Demo1" and "grab". The plugin "grab" is needed to acquire the image from hard disk (command line parameter "-sp"). All other plugins ("record" etc.) are inbuilt ones, but they are not registered into the current session of iceWing and thus are not active.

More of this all later in the section 5.2.4 Plugin Info...

## 3.2 The special plugin "grab"

**The plugin "grab"** is a very basic and thus inbuilt plugin, that allows to acquire images or video streams from grabber hardware, disk or via DACS from external processes. This plugin will be used by nearly all other image processing plugins. But

iceWing not *needs* this plugin! But then you need your own plugin as "data-provider" for e.g. audio streams or likewise.

**Some comments on up-/downsampling** iceWing provides the images via the plugin "grab" (or other data-provider plugins). And the plugin "grab" provides two separate queues of image histories: downsampled and upsampled (i.e. full/original sized) images. With a downsample factor of 1 the two queues have the same images, while a factor of 3 makes the downsampled images consist of every 3rd pixel of the original sized image. Both queues have a history size - how many images will be stored for plugin access to older images. Excellent - but what for?

Well, this feature is useful when both reduced and detailed image size versions are needed for the global task. E.g. a continously operating plugin "trajectory tracking" uses the faster downsampled images to swiftly keep track of the hand-position. Meanwhile, but much less frequently, another very time-intensive plugin "object detection" separates an object in that hand. For this job, it may register for the more detailed queue of upsampled images.

# 4 The command line interface

## 4.1 The command line parameters in detail

```
> icewing -h
```

gives you the list of command line parameters. Now they will be explained more detailed, arranged to subject.

### General options

**@**<**file**> This allows to store arguments in files. Option "@" replaces it's argument <file> with the content of <file>. Any lines in <file> starting with '#' are ignored, the remaining lines are treated as further options.

**-h** | **–help** Besides showing all options and there meaning iceWing writes the names of all plugin instances, that are created by parameter "-l" or "-lg" and then terminates. You need the precise names of the plugin instances if you wish to send options to specific plugin instances with option "-a".

**–version** Shows version information and exits the program.

**-n** <**name**> When you use DACS, the launched instances of iceWing must be somehow addressable. This option specifies the process name of this instance of iceWing - the default name is "icewing". If there are several instances of iceWing (network wide), you must care: Give at least those iceWing processes unambiguous names, that are used with DACS.

**-p** <**width**>**x**<**height**> Sets the size of preview windows, default: 378x278.

**-rc** <**config-file—config-setting**> If the argument to this option contains a '=', the argument is interpreted as a gui setting and the referenced gui element is modified accordingly. Otherwise, the given config file <config-file> is loaded *additionally* to the standard file "$(HOME)/.icewing/values" (which is read first). This option can be given multiple times.

**-ses** <**session-file**> Load the session file session-file *instead* of the standard file "$(HOME)/.icewing/session" and use this file for any session related operations.

**-time <cnt—plugins—all>...** "cnt" specifies after how many main loop iterations time measurements are given out. If "cnt"<=0 all time measurements are disabled. The default is 50. The other arguments allow to automatically create timers for measuring the execution time of the process() call for single plugin instances. If "all" is given, all plugin instances are measured. For example

```
> icewing -time ‘‘5 backpro imgclass’’
```

outputs time measurements all 5 main loop runs and creates timers for the plugin instances backpro and imgclass. This option can be given multiple times.

**-iconic** Start the main iceWing window iconified.

**-t <talklevel>** iceWing outputs debug messages only if their level is below <talklevel>, default: 5, used range of levels: 0..4.

### Input options

*Remember*: All this options, that are related to image input (-sg, -sp, -sp1, -sd, -c, -f, -r, -stereo, -bayer -crop, -rot, and -oi) are passed to the special plugin "grab". If you use another plugin as data-source, that plugin will have it's own input options (passed via "-a"). Neither "grab" knows of the other plugins options, nor does the other plugin see this input options.

So *this* options could also be thought as "input options for plugin grab". You can use multiple instances of the plugin grab and thus multiple images at the same time. If you want to do that, you have to pass these options via the iceWing option "-a" to the additional grab instances. Thus every instance of grab can get it's very own options. The multiple instances are created by loading the plugin via the option "-l" multiple times.

**-sg <inputDrv> <drvOptions>** *S*ource of *G*rabber: If you use a grabber camera for your images, you must include the AVlib in installation. The AVlib supports several camera systems, and here you select which you use. iceWing simply passes the given parameters on.

<inputDrv> can be one of PAL_COMPOSITE, PAL_S_VIDEO, V4L2, FIREWIRE, or UNICAP (or abbreviated "C", "S", "V", "F", and "U"). See section 4.2 for more details about the different drivers.

<drvOptions> is an option string, which specifies in more detail how the driver you selected with <inputDrv> should behave. The different options of the drivers are described in section 4.2.

The different drivers provide help for it's driver options: If you are not sure, which options your selected driver, e.g. "F", the firewire driver, has, try

```
> icewing -sg F help
```

and an overview of the options will be printed to the console.

If you only give -sg without anything special, the default setting is PAL_S_VIDEO with no special options.

**-sp <fileset>** *S*ource of *P*ictures: What you specify as <fileset> will be the picture(s) data-source for the plugin "grab". Reaching the last picture iceWing loops, beginning again with the first picture.

iceWing natively supports the *pnm* image format. Depending on your version (or if installed at all) of the gdk-pixbuf library, the range of supported image formats is greatly enhanced. Version 0.16 provides e.g. this formats: *bmp*, *gif*, *ico*, *jpeg*, *png*, *ras*, *tiff*, and *xbm*. *pnm* images are read in bit depths from 8 to 32 and in special variants float and double images can be read, too. *png* images are read in 8 bit and 16 bit depths. All other formats are 8 bit only.

<fileset> specifies the list of file names. It has the following format:
fileset = fileset | 'y' | 'r' | 'e' | 'E' | 'f' | 'F' | 'file'

**Single Pictures** You can simply give single pictures as files

```
> icewing -sp image.ppm image2.gif
```

**'y', 'r' → YUV or RGB** The pictures can be stored as color model YUV (which is the default) or RGB. With a 'y' or 'r' in front of the name you can specify the color model of the coming files. So this example names one YUV, two RGB and another YUV picture as data source sequence:

```
> icewing -sp imageYUV1.ppm r imageRGB1.ppm imageRGB2.ppm
y imageYUV2.ppm
```

**'file' with %d or any int based printf() conversion specifier** As further option you can name a whole series of pictures: with e.g. %d the plugin "grab" replaces %d by integer numbers beginning from 0 and tries to open the file. As another example "%04d" matches all numbers, starting with 4 leading zeros. As soon as iceWing cannot match the current number, it moves on to the next fileset. E.g.

```
> icewing -sp image%03d.ppm picture%d.ppm
```

makes the plugin "grab" increasingly scan for (and if found: load) files named with "image000.ppm", "image001.ppm"... If no further file is found, it scans for pictures named "picture0.ppm", "picture1.ppm"...

**'file' with at least %t or %T, or a combination of %t, %T, and %d** An alternative method to specify a series of pictures, one example would be

'/tmp/image%T_%t.png'. If %t or %T is inside the file name part of one 'file' to the -sp option, iceWing scans the complete directory, in the example '/tmp', for files matching the file name part with any numbers replacing %d, %T, and %t. It loads the found files sorted by the numbers replacing %d, %T, and %t in that direction. I.e. the coarsest order is given by %d and the finest by %t.

In this case no printf() style format specifiers are allowed, as files with any number format are used at the same time.

**'e', 'E' → Open files/Check file extension** iceWing must know the number of images you specified on the command line. To verify if a file is a movie file and then get its frame count, iceWing opens every file during startup. With an 'E' in front of the file names iceWing opens only these files which have a known movie extension (e.g. '.avi' or '.ogm'). This speeds up the program start. With an 'e' in front of the file names you switch back to opening all files. The default is to open all files.

**'f', 'F' → Duplicate/No duplicate frames in movies** Movie files store the number of frames to display in one second (FPS value) and the to be displayed frames. Normally, if less or more frames are stored in the movie at one point in time than the number of frames, which had to be available according to the FPS value, single frames get duplicated or removed. If 'F' is specified, this duplication and removal does not happen. In this case, the "Image Num" slider in the user interface on the "Other" page (see page 32) can only be used for seeking if the read continuous buttons ('¡¡' and '¿¿') are not pressed. The default is to comply with the FPS value.

**-sp1 <fileset>** This is just the same as -sp. But after the last picture is reached and every registered plugin has finished it's work on that picture, the iceWing process ends instead of looping back to the first picture.

**-sd <stream> [synclev]** Use a DACS stream as input of images (for plugin "grab").

An external process creates that stream of images somewhere in the network. It has published it via DACS, and this iceWing process can order that stream as input of images.

The stream can have some synchronize-tokens of several hierarchical levels integrated. This SYNC-tokens of increasing level create substructures of the incoming data (e.g. letters, words, sentences...). You can register the stream at a given synclevel. Level 0 means *every* single image will be delivered to this iceWing instance. Higher levels lead to fewer images, depending strongly on the SYNC-level philosophy of the stream creating process. You need to get this information about the stream creating process to choose the appropriate SYNC-level, with that you register the DACS stream. If the stream delivering

process separates the images with SYNC-level 2 and you order this stream at this level, you will get always the latest image. Any older images get dropped off the stream, as soon as the new image arrives the stream. So with this strategy iceWing gets always up to date images - simply by ordering at the appropriate level.

*Caution:* SYNC-level of 0 is *special* and means, that this instance of iceWing gets *every* single image, that is put into the stream (no loss of any image). You may need this, e.g. because plugins sometimes need access to older, but still unreceived images. But the DACS process must store all of the undelivered images. If iceWing consumes the images at a slower rate than the images are put into the stream (in average), this will definitely lead sooner or later to a *huge* size of the DACS process – and finally (when the storage limit is exceeded) the process gets killed by the system.

If you wish write your own image-creation process to send images to iceWing via DACS: There is already the SFB-360 internal data type "struct Bild_t" (which is declared in the file "sfb.h"). It encodes the image, and you must use it as the type to be passed to DACS. Then iceWing can receive images from your own process.

For further details about DACS see the dissertation of Nils Jungclaus [Jun98].

**-stereo** Expect gray images containing interlaced stereo images as input and decompose them by putting them one after the other. This option is similar to the "stereo=deinter" option of the firewire driver (see section 4.2 for more details).

**-bayer [mode] [pattern]** Expect a gray image with an embedded bayer pattern as the input image. Use the specified method and the specified bayer pattern to decompose it. If no method or pattern is specified, downsampling and RGGB are used. The supported methods are:

**down** Downsampling of the input image by a factor of 2.

**neighbor** Nearest neighbor interpolation.

**bilinear** Bilinear interpolation.

**hue** Smooth hue transition interpolation.

**edge** Edge sensing interpolation.

Supported bayer patterns are: RGGB | BGGR | GRBG | GBRG.

This option is similar to the "bayer" and "pattern" options of the firewire driver (see section 4.2 for more details).

**-crop x y width height** Crop a rectangle starting at position (x,y) of size width x height from the input image. If width or height are smaller than zero or zero,

the values are measured from the right or bottom side. E.g. "-crop 5 10 -5 -10" would crop a border of 5 pixel from the left and right sides and a border of 10 pixels from the top and bottom sides of the input image.

**-rot** {**0** | **90** | **180** | **270**} Rotate the input image by 0°, 90°, 180°, or 270°. The default is 0°.

### Options for up-/downsampling behavior

**-c** <**cnt**> iceWing internally manages a queue of downsampled images. With this option you can specify the length of this queue.

Default value is 2.

**-f [cnt]** If you do not specify this option, iceWing has only the downsampled queue of images. With "-f" iceWing activates the *F*ull sized (i.e. the upsampled) queue of images.

The optional [cnt] sets the queue size, the default is 1.

**-r** <**factor**> *R*emember downsample factor of input images.

If you use already downsampled images as input, unfortunately iceWing does not know this without further notifying. <factor> tells the interested plugins, what factor the source images got downsampled.

Remember: With downsampled image sources, plugin grab will *additionally* downsample the input into the downsample queue. So when the input images already have downsample factor of 2, and the current iceWing instance has downsample factor of 3, the images in the downsample queue will have a *true* downsample factor of 6 (compared to the original image), while the images inside the upsampled queue have downsample factor of 2. The only way to solve this and e.g. simulate the original size of the image is to use this option "-r": You tell iceWing, what downsample factor the input images already have. And the plugins can (but not need to) make use of this additional information.

And also remember: Even the commando "Save Original" saves the original (=unrendered) image, but *including* the given downsample factor (set in figure 3.1 page "other" or in paragraph Downsampling 5.2.3).

### Output/remote control options

The "-o" options have several different purposes regarding the communication to other programs and with the sub options you specify, what to output and what interfaces to enable.

**-of** With option "-of", this instance of iceWing can be fully remote controlled via DACS including nearly every single GUI-widget element.

This uses the capability of iceWing to save and load all it's current status into the config file (while session file stores the window properties). Remote control via DACS works quite similar: With this option "-of" iceWing publishes DACS wide the functions void <icewing>_control(char[]) and char[] <icewing>_getSettings(void). <icewing> is the name of this instance of iceWing (that you specified with option -n). The "char[]" means, that you send a normal c-string as parameter to the _control() function. The content of that string can be any lines of the iceWing configuration file and iceWing accepts the new settings. Similar, the _getSettings() function returns a string with the current settings of all widgets in the format of the configuration file.

Additionally this option publishes to DACS the function struct Bild_t <icewing>_getImg(imgspec). With this function, external processes can receive an image from this instance of iceWing via DACS. The images are send encoded in the SFB-360 "struct Bild_t" data type. There are further options, to allow the external process to select precisely which image it receives, and in which downsample format.

The format of "imgspec":

```
['PLUG' <plugnum>] ('NUM' <imgnum>|'TIME' <sec> <usec>|
'FTIME' <sec> <usec>) down
```

**PLUG** If multiple instances of the plugin grab are running, multiple images are available at the same time. With PLUG you can select the image from the instance <plugnum>. The default is 1, i.e. the image from the first grab instance.

**NUM** Every single image in iceWing has a continuous number, starting with 0. iceWing returns the image with the number <imgnum>. If <imgnum><0, return a full size image, see option "-f". If <imgnum>==0, return the current full size image.

So the sign determines, from which queue iceWing takes the image from: the upsampled or the downsampled queue (See also options "-c" and "-f").

If the upsampled queue is not existing, you will always get the downsampled version of the image.

**TIME** iceWing returns that image with a grabbing time most similar to (<sec> <usec>). The image is taken from the downsampled queue.

If TIME is in the future, the nearest image is taken - and that will always be the most recent image.

**FTIME** iceWing returns a full size image with a grabbing time most similar to (<sec> <usec>).

Again, if the upsampled queue is not existing, you will always get the downsampled version of the image.

<**down**> iceWing will downsample the returned image by factor <down>. This factor is applied *additionally* to the iceWing downsample factor (adjustable in page "Other", see figure 3.1)!

As example: iceWing has set a downsample factor of 2 and this option <down> is e.g. set to 3. Now the image will be delivered to DACS with a true downsample factor of 6 (well, with FTIME it is 3).

**-oi [interval]** Output images on DACS stream <icewing>_images and provide a function void <icewing>_setCrop("x1 y1 x2 y2") to crop the streamed images. <icewing> stands for the DACS name of this iceWing process (see option "-n"). With the optional [interval] iceWing will send only every nth image to the stream. If the upsampled queue exists, you will get an upsampled image, otherwise a downsampled image. The images are sent in the "struct Bild_t" format.

With the function <icewing>_setCrop("x1 y1 x2 y2") a freely defined rectangle of the image can be dumped to the stream. The parameter string defines the rectangle. The four coordinates refer to the full size, i.e. not downsampled image. iceWing adapts them internally to the real image size.

**-os** Output some (currently very few) status informations on DACS stream <icewing>_status.

The function "iw_output_status (const char *msg)" declared in output.h sends on this stream.

### Plugin options

**-l <plugin libraries>** Each plugin lives inside a library. This option loads the given plugin libraries into iceWing. The library names must be separated by ' ', ',', or ';'. Additionally, this option can be given multiple times. If you wish to have several instances of a plugin, repeat the name of the relevant library. But be cautious: not all plugins can operate as several instances (e.g. plugin "min")!

Search order: First the library is searched as specified with this option. If the library was not found, iceWing searches again in $(PREFIX)/lib/iceWing/ for the library. If still not found, the name is expanded by "lib[...].so" and again searched in $(PREFIX)/lib/iceWing/.

**-lg** *Not* for Alpha machines! Similar to "-l", but with a *very* impacting difference: while dlopen()'ing the libraries, the flag RTLD_GLOBAL is set (makes all not-static objects of the library global for the whole iceWing process)! So this is more a linker option, than an iceWing feature!

Use only, when you *really* know what you are doing (e.g. *great* danger of name-clashes...)!

**-a** <**plugin instance**> <**option**> Send command line arguments to a plugin instance. This option can be given multiple times.

Every plugin should provide help on it's options. To find those help messages, use parameter '-a pluginName "-h" '.

**-d** <**plugins**> Disables the given plugin instances. This option can be given multiple times.

Normally all loaded plugins get activated. You can toggle plugin activation also via GUI, but sometimes you may wish to start an iceWing session with an initially disabled plugin instance.

E.g. '-d "backpro imgclass" '

## 4.2 Parameters of the grabber driver

If you want to use a grabber camera as a source for your images, you have to pass the option "-sg" with one of the drivers PAL_COMPOSITE, PAL_S_VIDEO, V4L2, FIREWIRE, or UNICAP (or abbreviated "C", "S", "V", "F", and "U") to the grabbing plugin. You can additionally pass different options to the different grabber drivers. If you pass "help" as an option, you get an overview of all available options, e.g. for the firewire driver:

```
> icewing -sg F help
```

The help message will be printed to the console. The different options are separated by ":". Parameters of an option are separated by a "=" from the option name, e.g. "camera=2:bayer=hue" would be an allowed option string. In detail the different options are:

### 4.2.1 Drivers on OSF Alpha systems

#### MME driver for Composite or S_Video devices

**help** Shows the help page of the driver.

**camera=val** Up to two cameras can be connected to the computer. Here you can select with a number of 0 or 1 which of these cameras should be used. The default is 0, the first one.

**fps=val** You can grab the images with different speeds. This option sets the frame rate the camera should operate in. The default is 25.

### 4.2.2 Drivers on Linux systems

#### V4L2 driver for Composite, S_Video, and other devices

**help** Shows the help page of the driver.

**debug** If given, different debugging information about the camera, it's capabilities, and the current driver status is printed to the console.

**device=name** Specifies the device, the driver will use to grab images from. If this option is not specified, "/dev/video" or, if this is not available, "/dev/video0" is used.

**input=num** The video input to use. If the driver was called as "C" or PAL_COMPOSITE, the first composite video input is the default for this option. If called as "S" or PAL_S_VIDEO, the first S-Video video input is the default. And finally, if the driver is called as "V" or V4L2, 0 is the default.

**format=num** Most devices support different image formats, e.g. YUV or RGB formats in various pixel depths. Here you can select which one to use. If this option is not specified, the driver uses a YUV format with a depth as big as possible.

**propX=val** V4L2 devices have several properties, e.g. brightness, hue, contrast, and other. With this option you can set them. E.g. `"prop0=0.64"` will set the first property to 0.64. Information about the available properties and their allowed values is shown if the option "debug" is given.

**buffer=cnt** V4L2 can use several intermediate image buffers to compensate for an intermediate slowness of the program before grabbing the next frame. This option sets the number of buffers, the default is 4.

#### Firewire driver for DV-cameras connected via firewire

**help** Shows the help page of the driver.

**debug** If given, different debugging information about the camera, it's capabilities, and the current driver status is printed to the console.

**device=name** Specifies the device, the driver will use to grab images from. If this option is not specified, "/dev/video1394" or "/dev/video1394/0" is used, whichever is available.

**camera=val** Multiple cameras can be connected to the computer via one device. Here you can select with a number starting with 1 which of these cameras should be used. The default is 1, the first one.

**fps=val** DV-cameras normally support different speeds in which they can deliver the images. This option sets the frame rate the camera should operate in. Supported frame rates are: 1.875, 3.75, 7.5, 15, 30, and 60. The default is 15.

**mode=yuvXXX|rgbXXX|monoXXX|16monoXXX** DV-cameras can support different color spaces and different image sizes for the images. With this option you can select which mode the camera should operate in if an image without any downsampling should be grabbed. If the desired mode is not supported by the camera, the driver falls back to a supported mode. The "XXX" specifies the desired width, e.g. "mono1024" would be a possible mode specifier. The default is yuv640x480.

**bayer=down|neighbor|bilinear|hue|edge** Some cameras support color image grabbing, but deliver the image not decomposed but as one gray image plane with an embedded bayer pattern. In a bayer pattern a square of size 2 by 2 pixels holds information about all three RGB channels. To decompose this information, different interpolation methods exist. If this option is given, a gray image of depth 8 or 16 bit with an embedded bayer pattern is expected and decomposed with the specified method. The supported interpolation methods:

**down** The 2x2 bayer square is used to only get one color pixel, the destination image gets downsampled by a factor of 2.

**neighbor** Nearest neighbor interpolation, where each interpolated output pixel gets the value of the nearest pixel in the input image, is used.

**bilinear** Bilinear interpolation, where each interpolated output pixel gets the average value of the two or four nearest pixels in the input image, is used.

**hue** Smooth hue transition interpolation. Here the green channel is gained by bilinear interpolation. For the blue and the red channel a "hue value" gets defined as $B/G$ or $R/G$. The neighboring hue values are then used to estimate a color pixel. E.g. if a blue pixel is located on the left and on the right side of a pixel, the blue pixel in the middle gets estimated by:
$B_M = \frac{G_M}{2} * \left(\frac{B_L}{G_L} + \frac{B_R}{G_R}\right)$

**edge** Edge sensing interpolation. The blue and red channels are computed identical to the "smooth hue transition interpolation" method. For the

green channel horizontal and vertical gradient magnitudes are calculated. A green pixel gets interpolated by the horizontal neighbors, if the horizontal gradient is smaller than the vertical one. Otherwise the vertical pixels are used.

**pattern=RGGB|BGGR|GRBG|GBRG** The information in a 2 by 2 bayer square can be ordered in different ways. This option specifies how it is ordered. The default is RGGB, red in the first pixel, green in the second pixel and the first pixel on the second row, and blue in the second column on the second row.

**stereo=raw|deinter** If given, an image of type YUV422 is expected. However, this image is not interpreted as a normal color image, but as two interlaced gray scale images. If stereo=raw is given, the image gets interpreted as a gray image without decoding the interlacing. If stereo=deinter is given, the image gets decoded and saved one after the other. For example the Videre stereo camera saves its two images in this way.

#### Unicap driver for various devices

This driver uses the unicap library to access various devices. Unicap provides a uniform API for different kinds of video capture devices, e.g. IEEE1394, Video for Linux, and some other. See "`http://www.datafloater.de/unicap/`" for details about this library.

**help** Shows the help page of the driver.

**debug** If given, different debugging information about the devices, it's capabilities, and the current driver status is printed to the console.

**device=val** Unicap supports multiple devices. Here you can select which one to use. The default is 0, the first one.

**format=val** Most devices support different image formats. Here you can select which one to use. The default is 0, the first one.

**propX=val** The different devices have several properties, e.g. brightness, hue, video source, and other. With this option you can set them. E.g. `"prop0=11738"` will set the first property to 11738. Information about the available properties and their allowed values is shown if the option "debug" is given.

## 4.3 Configuration files

iceWing uses two configuration files, which get loaded and stored during runtime:

**.icewing/session** stores the window properties of the current session. Default wise it's "$(HOME)/.icewing/session", but you can use alternative files via command line option "-ses". In the preferences window or with the context menu you can save your current session into an alternative file.

The content is simple - for each active window there is an entry like

"Name of the window"= "x" win-x "y" win-y "w" width "h" height
"zoom" zoom "dx" pan-x "dy" pan-y

where win-x and win-y specify the window position, zoom specifies the zoom factor of the window (0 means fit-to-window) and pan-x and pan-y specifies the panning position for the content of the window. The zoom and panning values are only stored if "Save pan/zoom values" in the preferences window is active, see section 5.2.2. Lines starting with "#" are treated as comments and get ignored.

**.icewing/values** stores all settings of every single GUI value of the plugins and the iceWing system. Additionally, the hotkeys for the context menu of the image windows get stored in these files. Default wise it's "$(HOME)/.icewing/values", but you can use *additional* files via command line option "-rc" or in the main window with the Load/Save buttons.

If you wish to remote control the iceWing process via DACS, you must know the structure of the content of this config file. The best will be to save the current settings and have a look at the file.

Every line holds for one iceWing widget it's setting. Each widget is unambiguously addressed (it's path) via it's window name or it's category name and it's widget name. It's entries look like this: "windowname.widgetname" = value

Each widget type has it's own kind of values (e.g. for booleans: true=1, false=0), the most complex widget surely is "list". More details about the different widgets can be found in the Programming Guide in section 7.3.1.

# 5 The Graphical User Interface

## 5.1 The iceWing render chain

Why should you know this details? Well, you will need to understand the basics of the underlying render mechanism to fully understand how the GUI works and what the many GUI-commands are used for! In iceWing rendering data to image windows is done in several steps. Figure 5.1 shows an overview of the complete rendering process. By using the different GUI commands you can manipulate and inspect the data at different positions of the render chain.

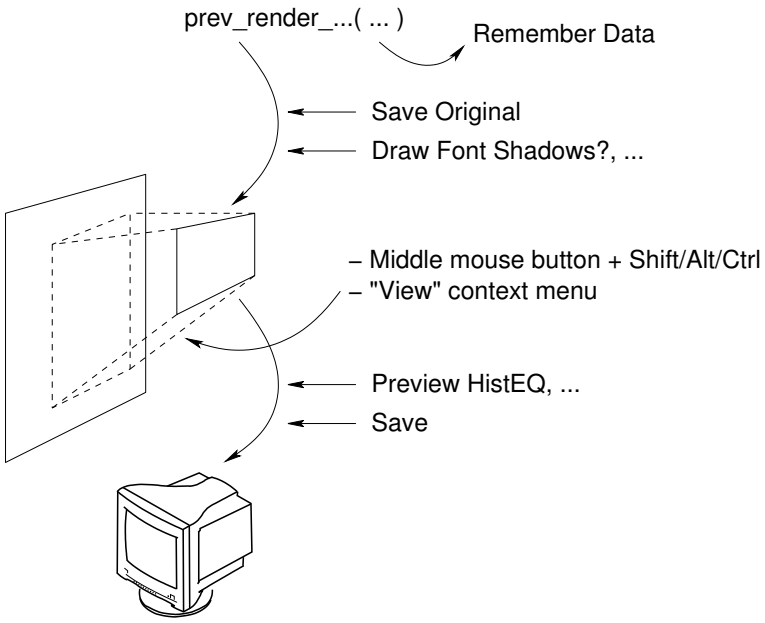



Figure 5.1: The render chain of iceWing.

If a plugin wants to display any data, it calls one of the different `prev_render_xxx()` functions (see section 7.3.2 for an in deep description of these functions). An example is the plugin "grab", which displays the loaded or grabbed image in the window "Input data".

The first thing these render functions do is to check the "Remember Data" flag, which is associated to every image window. If the flag is set, the complete data which

is passed to the render functions is copied for later use. This allows to re-render the complete image without the help of the plugin. If any of the remaining parameter of the render chain are changed, e.g. the displayed part of the data or the zoom level, the data can be immediately redisplayed. This means, that you can instantly see the effect on your current image. Otherwise, if the flag "Remember Data" is not set, the effect of the changed parameter gets only visible after the next call of the plugin to the `prev_render_xxx()` functions. For the "grab" plugin for example this means that you see the effect not until the next image is loaded or grabbed.

If an image should be displayed, the data of the complete image can now be saved with the help of the "Save Original" GUI function. *Attention:* This saves only the first image, not any text, which may be rendered on the image, nor any lines, circles or anything else besides the first image.

The next step in the render chain is the rendering of the data into an internal buffer. During this the data can be modified. E.g. you can add a drop shadow to all displayed text via the context menu of the image windows. Moreover the displayed region and the size/zoom factor can be changed interactively with the mouse. Thus the coordinate system for the data as specified by the plugin (the "world coordinates") and the coordinate system for the rendered image as displayed on the screen (the "screen coordinates") are not identical.

The last step in the render chain is the display of the buffer on the screen. During this some color changes, e.g. a histogram equalization, can be applied to the buffer. Besides displaying the result on screen the result can be saved to a file with the help of the "Save" GUI function.

## 5.2 The GUI commands

### 5.2.1 iceWing main window

The main window is divided into two main parts: In the upper area you see "Categories" and to it's right the page content of the selected category. Most plugins will create at least one entry (called "page") into the categories-list. It allows you to check/change that plugins parameters. Please see the respective plugin's documentation for any plugin specific information. In this documentation only the somewhat special categories "Images" and "Other" will be described in more detail (see sections 5.2.5 and 5.2.3).

Besides these categories you see some global buttons in the iceWing main window:

**Detach/attach symbol (D/A)** Clicking this symbol at the top right corner will detach this page into a separate window. Clicking again will put it back into the iceWing main window.

You will use this feature, if you repeatedly wish to flip quickly from one page to another. Or you change the slider on plugin page 1 and want to see the effects on plugin page 2. If you e.g. work on a plugin sheet and wish to single step (wait-delay=-1) through the next images and watch the plugins doing, you surely wish to detach the "other" page.

**Preferences button** The wrench icon opens the preference window, where you can configure different settings for the iceWing main program. See section 5.2.2 for a complete description.

**Load/Save buttons** Loads/saves all made settings from all widgets inside iceWing from/into the file "$(HOME)/.icewing/values", if you selected the "Def" (meaning "default") variant of the buttons. The Load/Save buttons allow to select the file name. But be aware: this is different to the command "load/save session", which stores the window size and position of all open windows, not the widget settings.

### 5.2.2 Preferences button

The wrench icon opens the preference window, where you can configure different settings for the iceWing main program. "Image Saving" specifies settings for the save functions in the context menu of image windows. In detail these are

**Image format** Specifies the file format used for saving images. If "By Extension" is selected, the format gets selected based on the extension of the file name.

Saving in a vector format, i.e. the "SVG" format, requires that "Remember Data" in the context of the image who's data should be saved is activated. In this case the complete image must be rerendered, which needs the saved data. See section 5.1 for more details about "Remember Data". Additionally, some features of the iceWing render functions are not supported by SVG. So the exported images may not be completely identical to the displayed ones.

**Image name** The file name, under which the images will be saved. You can embed different information in the name by using the following modifiers:

%d: The consecutive image saving counter, starting at 0.
%t: The milliseconds part of the time the image gets saved.
%T: The seconds part of the time the image gets saved.
%b: Name of the current user.
%h: The system's host name.
%w: The name of the window, from which the image gets saved.

Any of the above modifiers can be changed by printf() style format specifiers. E.g. “image%03d.ppm” would result in “image000.ppm”, “image001.ppm” and so on as file names.

**AVI framerate** If AVI files get saved, this value will be entered in the file for the frame rate. That is this value will not really change the saved data, only this one setting in the header of the saved AVI file is changed.

**Quality** The quality and thus the compression value used when saving jpeg images or AVI files. 100 means a small compression and thus a good quality.

**Show SaveMessage** If activated a dialog is shown after every image saving which confirms the successful saving.

**Save full window** If activated and “Save” or “Save Seq” is used as the saving command, an image of the size of the complete image window will be saved. If deactivated a black border around the image will not be saved.

**Reset FilenameCounter** By using %d in the file name, a consecutive image saving counter is embedded in the image file name. This button resets this value to zero.

“Other” has settings for the session handling and the GUI. In detail these are

**Save As / Save / Clear** These buttons are similar to the menu entries in the context menu of the image windows. “Save As” saves the current session in a file and allows at the same time to change the current session file name, “Save” saves the current session in the current session file, and “Clear” physically deletes the current session file and thus clears this session. The default session file is “$(HOME)/.icewing/session”.

A session file stores a list of windows and their configuration, their position, their size, and optionally their zoom and panning values. On the next start of iceWing a session file can be loaded, which then will restore the windows and the window configuration.

**Auto-save at exit** If activated, at program exit the iceWing window configuration will be saved in the current session file.

**Save pan/zoom values** If activated the current panning position and the current zoom value will be saved additionally in session files. Otherwise only the position and size of all open windows will be saved.

**Use tree for** If set to “Categories”, the categories list will use a tree widget, i.e. categories like “Demo1 WidgetTest” will be shown in a tree like strukture. Otherwise a list widget will be used. If this widget is set to “Images”, the window

list in the category "Images" will use a tree widget. If you change this setting, you must save the current settings (by using the "Save" or "Save Def" buttons in the main window) and restart iceWing.

**Scrollbars in categorie pages** If activated, the iceWing main window and any detached pages can be resized to any height. If needed, a scrollbar is displayed in the different pages. If deactivated, the windows height can not be made smaller than what is needed for all widgets.

**Auto add render widgets** The rendering in image windows can be modified by using commands in the context menu of the image windows. By default these commands are only visible if the plugin which performs the rendering has added the commands. If this button is activated, all commands which belong to the kind of rendering used in an image window are added automatically by iceWing.

### 5.2.3 Commands in category "Other"

The category "Other" has widgets, which specify settings for the plugin "grab", and some widgets for the iceWing main program:

**Interlace** Different grabber camera systems have different methods of sending the video-stream. Here you can select what of the grabbed data should be used. "Both" selects the complete image, "Even" selects only the even field of an interlaced image. "Even + Aspect" grabs only the even field and afterwards halves the image in the horizontal direction to get square pixels again. "Down 2:1/Virtual 2:2" adjusts for grabbed halve field images by halving the image in the horizontal direction and afterwards telling other plugins that the image was downsampled in the vertical direction, too (see also command line parameter "-r").

Attention: Only some grabbing drivers and additionally only some kernel camera drivers support the grabbing of half fields. So if you specified "-sg" for using a grabber and you set this to something different than "Both" it might well be that the expected does not happen. In this case half field grabbing is not supported with the used configuration of the grabber driver, its configuration and the selected downsampling factor.

**Downsampling** A ratio of 1 will grab and deliver the image 1:1. But if you e.g. set it to 3, then only every 3rd pixel in both horizontal and vertical direction of the full sized image will be delivered to other plugins - it will become scaled down by a factor of 3. There can be two queues, where the images are stored – one for the original size image and another queue for the downsampled images (see command line parameter "-c" and "-f").

**File TimeStep** Every image the plugin "grab" provides to other plugins is marked with a time stamp. E.g. if you use the grabber, the time stamp marks the time the image was grabbed. With this slider you can select the behavior if files from disk should be loaded.

-1 sets the time stamp to the time the image was loaded. Values above 0 specify an increase of the time stamp in ms. E.g. if set to 40 the first image gets a time stamp of 0ms, the second of 40ms, than 80ms .... If this value is set to 0, the frame rate of video files and the values scanned during processing %t and %T in file names specified with option "-sp" are used. See page 16 for further information about %t and %T.

**Frame correct Wait Time** If this button is not selected, iceWing waits exactly the time slice specified with the next slider before the next image gets acquired. If this button is selected, iceWing adapts this time slice. For example if the processing of the last image took 50ms and the 'Wait Time' slider is set to 200ms, iceWing will only wait 150ms.

**Wait Time and positioning buttons** Sets the delay (in milliseconds), until the next image shall be acquired from disk/grabber. If you set it to -1, iceWing waits until you manually change the image by pressing one of the positioning/acquiring buttons. This -1 works like a "pause-mode".

**Image Num** If you work with a video stream on disk, which you have specified with the command line parameter "-sp", this slider will appear. It shows the current position inside the video stream and allows to seek to an other position.

**Enable DACS Output** If iceWing outputs any data via DACS, e.g. if you have used the command line parameter "-oi", toggling this button disables/enables this outputting.

**Plugin Info** Opens the "Plugin Info" window, which shows different information about all loaded plugins. This window is described in more detail in the section 5.2.4.

**About** Opens a window showing some information about iceWing, e.g. the version number and the copyright.

### 5.2.4 The "Plugin Info" window

The "Plugin Info" window, which can be opened with a button in the category "Other", shows different information about plugins iceWing knows about and about

the communication between them. Figure 5.2.4 shows all the different pages of this window.

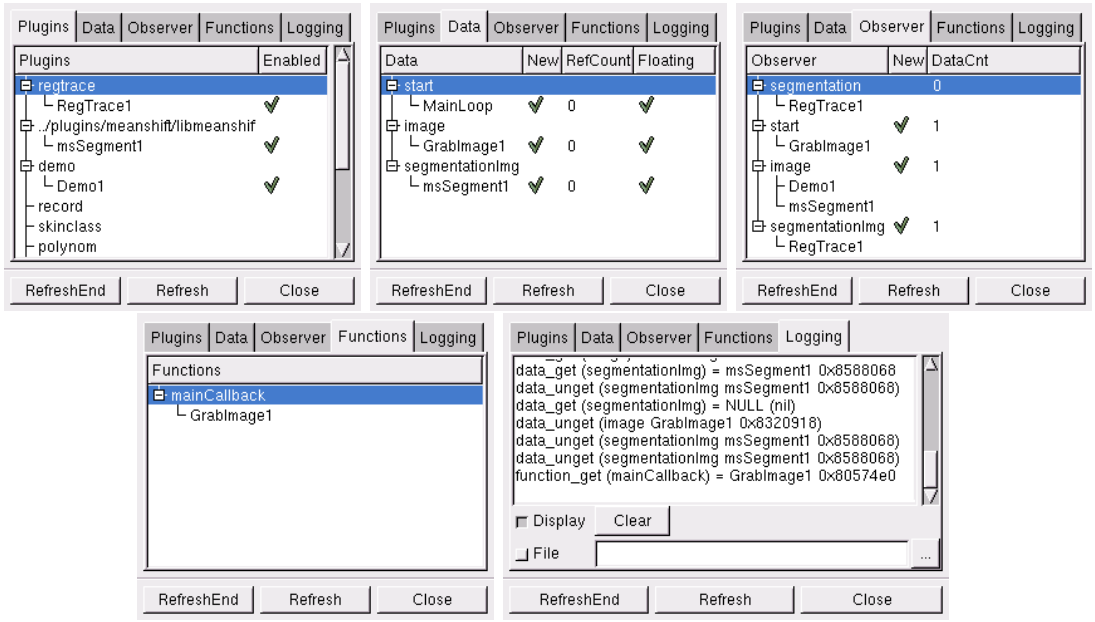



Figure 5.2: All pages of the window "Plugin Info", which shows information about loaded plugins.

Inside a main loop iceWing calls the registered plugin instances repeatedly. The order in which the instances get called is defined by the plugins. For this and for the communication between plugins iceWing offers different functionality. Plugin instances can interchange data, they can observe the provision of data, and they can call functions of other plugin instances. Details about these communication possibilities is displayed on the remaining pages. The displayed information is not continously updated. By pressing "Refresh" the current internal state about the communication information is displayed immediately. "RefreshEnd" defers the display shortly before the end of the main loop, directly before any floating data with a reference count of zero gets deleted (see below). So by pressing "RefreshEnd" you will get in a lot of cases information about all data, which was provided during the last main loop iteration.

**Plugins** On the first page *"Plugins"* a list of all plugins iceWing knows about and all instances of these plugins is displayed. A double-click on one of the plugin instances (de)activates the instance. If an instance is deactivated no functions of this plugin instance are called. This is similar in effect to the command line parameter "-d". Additionally there is a context menu with two entries. The first entry allows to (de)activate an instance of a plugin. With the second a new

instance of a plugin gets created. This is similar to specifying "-l pluginName" on the command line.

**Data** The *"Data"* page shows information about all data the plugins have created. In iceWing an identifier, a string, is always associated with all data. This identifier allows the plugins to access the data. In the first column the identifier and the instance name of the plugin, which has made the data available, is displayed. All data inside iceWing is reference counted and gets automatically deleted if the reference count drops to zero. If the data is marked as "floating" the deletion is deferred until the end of one main loop iteration. Data is marked as "New" as long one main loop iteration is not finished since the data was provided by a plugin instance.

**Observer** The *"Observer"* page shows information about plugin instances, which observe the provision of data. If new data with an identifier, which is observed, gets provided by any plugins the observer of this data are called with the new data as an argument. Thus the order in which instances get called is defined by the provision of data and by the observation of these data elements. There is one special data element. The data "start" is not provided by a plugin, but by iceWing at the start of every main loop. Plugin instances can observe this data and thus get called at the start of the main loop. The "Observer" page shows the different registered observer and information about currently known data elements with a corresponding identifier. New data is available, if the data was provided during the current main loop iteration. Additionally the amount of available data elements with the observed identifier is displayed.

**Functions** The *"Functions"* page shows information about all functions, which plugin instances have published. The function identifier and the plugin instance names, which have provided the function, are displayed.

**Logging** The *"Logging"* page allows to get more detailed real time information about the communication between the plugins. By activating the "Display" toggle button calls to the different iceWing functions, which deal with the communication between plugins, are shown in the text widget above. For a description of the different functions please see section 7.2. The "Clear" button clears the text widget. Deactivating the toggle button stops the logging. Activating the "File" toggle button records this information in the file, whose name is given in the string widget next to the toggle button. The file is closed if the toggle button is deactivated.

### 5.2.5 Category "Images" and image windows

The category "Images" shows a list of all image windows that the plugins wish to display. Double-clicking an entry of the list opens or closes the window of that entry.

Every of these image windows iceWing or any plugin creates will have different standard menu entries in a context menu. You can access this menu by clicking with the right mouse button in an open image window. Depending on the things a plugin renders in the image, there might be additional plugin specific entries.

Here is a list of entries you will find in all or, partly, a lot of image windows:

**File/Save** Saves the actual visible window content, including all active rendering manipulations. In the preference window of iceWing you can specify different parameters for the saving process, e.g. the file name and the file format.

**File/Save Original** Saves the underlying original "world coordinate" image, without any active rendering manipulations, in the original color space (for example YUV). But if the downsample factor (page "other") is >1, you will still save downsampled images. This downsampling happens inside the plugin "grab" *before* the original images are anyhow rendered in a image window or passed to further iceWing plugins. Moreover this function does not save any texts, lines, regions, or anything else besides images the plugin may display in the image. Only the first image is saved.

**File/Save Seq** Once activated, every new acquired image (the visible window content, including all active rendering manipulations) is saved continuously until deactivated. Normally, if the active image format is for single images, a series of image files gets stored. Otherwise, if the AVI file format is selected, a single video-stream file is stored. You can change this format in the preferences window, see section 5.2.2 for further details.

*Caution!* If you have selected the AVI format in the preference window, you must remember: To prevent corrupt AVI files, you must end this "Save Seq" by (de)selecting this command in the context menu again and continue with at least one new image to close the saved AVI file. Alternatively while recording you can close the whole image window or end iceWing with the "Quit" button, and iceWing sends the close file command itself. Changing the Image format in the preference window will do the same. If the file is not closed the AVI will miss some finalizing code and thus will be corrupt.

**File/Save OrigSeq** Same as "Save Seq", but the original "world coordinate" images are used. For the data, which gets saved by this function the same as already stated under "Save Original" applies.

**File/Save Session** Saves the current iceWing window configuration under the currently active session name. On the next start of iceWing every currently open window will be remembered as it is. Without any special parameters the next time iceWing is launched, the default configuration-file in "$(HOME)/.icewing/session" will be used to restore any windows. Alternatively, the command line parameter "-ses <session-file>" can be used to switch to non-default session files.

The Save/Clear Session menu entries are the same commands as the ones in the preferences-window.

**File/Clear Session** This physically deletes the current session file and thus clears the session. At the next launch the window layout of iceWing will look like the inbuilt default.

**View** The View submenu contains different entries to zoom and pan inside the image windows. So these entries are alternatives to the middle mouse button and the scroll whell. See section 5.2.6 for more details. The menu is especially handy if the entries are called via hotkeys. Besides using predefined hotkeys, the hotkeys of all menu entries can be dynamically changed. If a menu entry has currently the focus of the mouse a new hotkey can be set by simply pressing the desired hotkey. All hotkeys are saved in the configuration file.

**Info Window** Opens a small window, which displays the coordinates and color (in several color spaces) of the pixel at the mouse position if the mouse is inside any image window. Additionally, if the mouse is over any rendered images, the "Original" tab shows information about the data at the mouse as it was passed to the iceWing render functions. For example iceWing can display images containing float values, which are converted to 8 bit integer values during display. Possibly, these values are further changed by any special rendering filters or any drawings, which are shown above the image. These final 8 bit values are then displayed as the color values. In contrast, the "Original" tab shows the float values from the inital image.

Additionally, this window contains two toggle button for two special functions. If "Grab Values" is pressed, the info window waits on a press with the left mouse button inside one of the image windows. The values at the time the button was pressed are then additionally displayed in the info window. Thus two positions can be easily compared. If "Measure" is pressed, distances and angles in the image windows can be measured. If the left mouse button is pressed inside an image window and then moved to a second position, the distance of these two positions and the angle between a horizontal line and the marked line are displayed in the info window. Afterwards, to change the initially selected line, the end points of the marked line can be dragged around.

**Remember Data** (De-)activates the "Remember Data" mode, that was already discussed in detail in section 5.1.

**Settings** Opens a window, where you can change some further image window related options. As one point different options for the "Show Meta info" feature and the image rendering can be specified. Moreover, all plugins can add any widgets to this window. For a description of these options please refer to the special plugin documentation. The other options are:

**Histogram** "Show Meta info" displays histograms of all rendered images. Here you can change the kind of this histograms. "Use lines" changes the visualization form of the histograms (filled or single lines). "Include min/max" changes the method used for scaling the histogram in the vertical axis. If activated, the biggest value of the histogram is used to scale the histogram. If deactivated, the first and the last histogram entries are not considered during determining the biggest value of the histogram. Useful, if a small object is located on an otherwise black or white image.

**scaleMin/Max** Images in iceWing can have any data type ranging from unsigned chars to doubles. For displaying these images must be converted to a range of 0 to 255. With these sliders you can influence the conversion process. With "scaleMin = -2" the image values are simply shifted in the range 0..255 without locking further at the image values, e.g. for unsigned short images 65535 is displayed with a value of 255. "scaleMin = -1" clamps the image values to a range of 0..255.

All other values for the sliders consider the minimal and maximal values of the to be displayed image. The Min slider specifies the amount, the minimal value is shifted towards the maximal value, in percent of the difference of the minimal value and the maximal value. The Max slider shifts the maximal value towards the minimal value. All pixels darker then the shifted minimal value are displayed as black, all pixels lighter then the maximum are displayed as white, and everything in between is stretched linearly.

### Entries of the submenu Preview

**Normal** Shows the image in the window with the original colors as they were specified during rendering.

**CStretch** The color histogram of the image is stretched: The minimum (and maximum) of all used colors is set to 0 (255) and all colors are stretched linearly to the full range 0 - 255 again.

**CStretch5** 5%, beginning with the nearest to black (white) pixels are set to black (white). All other colors are stretched linearly.

**CStretch10** Same as CStretch5, but with 10% of all pixels to black/white.

**HistEQ** Equalize the histogram of the image.

Equalization is used to repair images that have too much contrast or are too light or dark. Equalization attempts to flatten the histogram of the image.

**Random** This randomly shuffles the color map.

So colors of the original image, that look very similar, change to very distinguishable colors. This is very useful to e.g. verify some color-separation processes.

**Show MetaInfo** This menu entry appears only if images are displayed in the window. "Show MetaInfo" (de-)activates the display of some additional information of the original image as passed to the render functions, for example the color-space of the input image, the size of the image in pixels and the color histogram of the image.

**Font** This menu entry appears only if text may be displayed in the window. Selects the font and thus the size of any text, which get rendered inside the image window (for example the text shown if the meta info is activated).

**Font Shadows** This menu entry appears only if text may be displayed in the window. Shows the rendered text with shadowed fonts. Helps sometimes to make the text more visible.

### 5.2.6 Panning/Zooming the image windows

If a new image window is opened the rendered image is normally displayed in such a zoom level that it is always completely visible. By pressing the shift key and the middle mouse button inside the window you can zoom into the image, by pressing the ctrl key and the middle mouse button you can zoom out. Alt plus the middle mouse button resets to the initial fit to window displaying mode. If fit to window is *not* active you can hold the middle mouse button and move around inside the window to move the clipping frame for the "world coordinate" picture, i.e. to pan inside the image.

*Attention:*
If your window manager is already using one or more of that combinations for itself, then that signal can not reach iceWing! Then you have to change your window manager setting: Disable that mouse signal there, so it will no longer get caught by your window manager and can reach iceWing.

Besides using the middle mouse button for zooming and panning, there are two additional posibilities to reach these functions: by using the mouse wheel or the context menu. For panning the window vertical, again if fit to window is *not* active, you can use the mouse wheel. Panning horizontally can be done with the mouse wheel while pressing the shift key and zooming by using the mouse whell while the ctrl key is pressed. Additionally, the image context menu contains a submenu "View" with entries for all these functions.

## 5.3 The GUI widgets

(TODO: FULL list, or only short summary?) Widget "List" can have a context menu. And it can be reordered via drag'n drop...

in Goptions.h opts_xxx_create()

# Programming guide

# 6 iceWing Files

## 6.1 Filesystem hierarchy

Let's see, what the installation consists of. Below the installation prefix, for example "/usr/local", you have this structure-content:

**bin/** icewing - the executable itself

icewing-config

A shell script similar to e.g. gtk-config which makes compiling of own plugins easy. It generates compiler-flags, extracts system-paths, and more. "`icewing-config --help`" shows all available options of the script.

icewing-docgen

A shell script which collects help messages of all plugins, which are installed under $(PREFIX)/lib/iceWing and which are integrated in iceWing. The script calls all these plugins with the option "-h" and stores the output in the files "Readme.txt" and "Readme.html" in the current directory.

icewing-plugingen

A plugin template generator. The script generates in the current directory a basic C or C++ plugin including a Makefile to compile it. "`icewing-plugingen --help`" gives more information.

`'icewing-config --exec-prefix'` gives this directory, whereas `'icewing-config --prefix'` gives the installation directory, i.e. this directory without the "/bin" part.

**include/** - All the headers for your own plugin programming

iceWing - headers from the iceWing system

Besides other options, `'icewing-config --cflags'` contains this directory.

iwPlugins - plugin headers

If your plugin publishes new structures to be used by other plugins, you probably want to put them here.

`'icewing-config --pincludedir'` gives this directory.

**lib/** iceWing/ - Here is the default place for iceWing to search for plugin libraries. Normally you give iceWing by command line parameter "-l" library names to load. If iceWing is not able to load them directly, it automatically tries to load them from this directory.

`'icewing-config --libdir'` gives this directory.

**man/** - The manpage of iceWing

**share/** iceWing/ - Place where plugins can store additional data files. E.g. the plugin "Face" has here it's config-file placed, and the polynom-classificator plugin stores it's data here, too.

`'icewing-config --datadir'` gives this directory.

log/icewing.log - The installation log file. It contains all actions performed during the "make install" phase.

## 6.2 Headerfiles overview

TODO

# 7 iceWing – A CASE Tool

**Originally taken from [Löm04, Anhang B]. Since then translated by Ilker Savas and updated according to changes in iceWing.**

During developing software in science there are certain extensive tasks to do, which occur every time but do not belong to the real task directly. Very often one needs a way to influence the parameters of an algorithm easily and quickly. Similarly important is the possibility to easily visualize any data. It must be easy to examine and save the data at any time. In greater integrated systems it is advantageous, if the different components can be developed separately from each other without the loss of flexible and fast interaction with each other afterwards.

In science an ergonomically sophisticated graphical user interface, which can also be easily handled by a person not familiar with the program, is mostly not needed. In most cases it is important that a small team which is familiar with the task can easily develop and optimize its special algorithm. This team of specialists must be able to handle the user interface in an easy way. Thereby it is mostly not intended to develop a complete program for an end user. Several systems already offer functionality in these directions. For example the commercial program MATLAB offers comprehensive options for visualization and also for generation of graphical user interfaces, which can be used easily via the provided scripting language [The03]. The open script language Tcl with its graphical tool Tk offers also great facilities to generate user interfaces [Ous94].

But existing systems usually are not optimized for the specific needs during developing scientific software. iceWing, *the Integrated Communication Environment Which Is Not Gesten*[1] was developed to account for this lack. iceWing is a graphical shell, which offers the above mentioned functionalities to dynamically loadable plugins in an easy way. In the next sections iceWing will be introduced more closely. First an overview of the structure of plugins is given. The following chapters then give more details of the various fields of iceWing in an exemplary manner. So this is not a complete reference manual which would describe all functions and types of iceWing. Further details not mentioned here can be found in the header files and example plugins of iceWing.

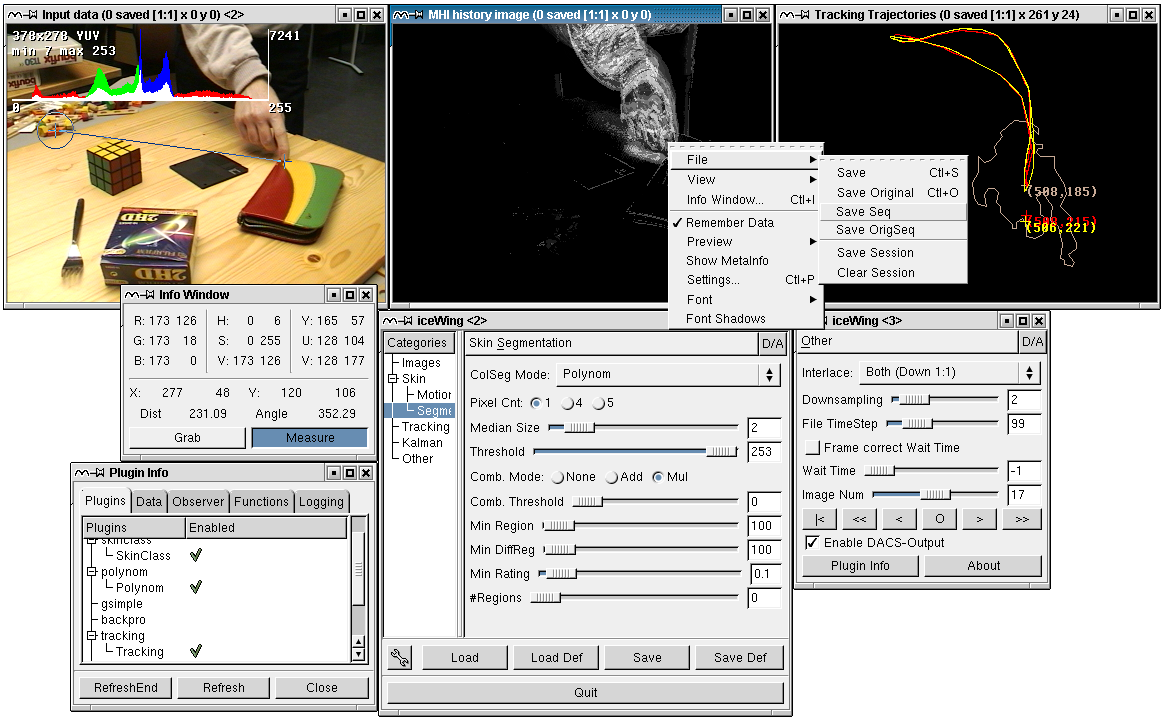



Figure 7.1: A typical session with iceWing. Various intermediate results are being displayed and could be examined. Various parameters can be influenced interactively.

## 7.1 Overview

iceWing is a program written in the C language, which can dynamically load plugins realized as shared libraries . It was tested on i386 Linux with the compilers GCC Version 2.95 up to Version 4.0[2] and on Alpha's with OSF 4.0f with the compilers DEC C Version 5.9 and GCC Version 3.1. For graphical outputs as well as the user interface generation the GTK toolkit[3] Version 1.2 is used. With the help of other external libraries the functionality can be extended, for example towards supported graphical formats and supported cameras for grabbing images. Figure 7.1 demonstrates a typical session with iceWing in which hands in a sequence of images are segmented and tracked.

### Plugins in iceWing

[1]This is a reference to an older program, the predecessor of iceWing.
[2]see http://gcc.gnu.org/
[3]see http://www.gtk.org/

iceWing the program only provides an initial user interface and miscellaneous auxiliary routines. The real functionality is realized by the various plugins that may be plugged into iceWing. Plugins for iceWing must implement the interface shown in Figure 7.2. This is illustrated in Figure 7.3 with the help of a minimal plugin. The source code of this plugin as well as a Makefile to compile it can be found in the iceWing source distribution in the directory "plugins/min/". When the plugin, i.e. the shared library, is first loaded, a function named `plug_get_info()` is invoked. This is also the only predetermined entry point in the library. This function is used to create a new instance of the plugin, so it acts as a factory function. It returns a pointer to a filled structure of type `plugDefinition`.

```
typedef struct plugDefinition {
    char *name;
    int abi_version;
    void (*init) (struct plugDefinition *plug,
                  grabParameter *para, int argc, char **argv);
    int  (*init_options) (struct plugDefinition *plug);
    void (*cleanup) (struct plugDefinition *plug);
    BOOL (*process) (struct plugDefinition *plug,
                     char *id, struct plugData *data);
} plugDefinition;
```

Figure 7.2: The structure plugDefinition, which every plugin must implement.

The structure `plugDefinition` contains all the information iceWing needs for a new plugin. The function pointers `init()`, `init_options()`, `cleanup()`, and `process()` define entry points for the instance of the plugin. `init()` is used for the general initialization of an instance. For example it processes command line arguments for the plugin instance. In `init_options()` the graphical user interface is initialized. More details about the user interface initialization can be found in section 7.3. `cleanup()` is invoked at the end of the program to release any resources. Finally `process()` is called if the real functionality of the plugin is to be executed. When this invocation happens can be determined with the functions for the communication between plugins. For more details see section 7.2. The variable `name` declares the name of the instance. As the name serves as the identification of the instance it must be unique. `abi_version` should always be set to the constant `PLUG_ABI_VERSION`. During runtime it is used to check if the plugin was compiled against the correct iceWing version.

As the plugin in Figure 7.3 always gives a fixed name back it can be instantiated only one time. To change this the structure of type `plugDefinition` has to be allocated dynamically and the plugin name contained inside the structure must be made unique for each invocation. This can be achieved by integrating the instance number `cnt` into the name. `cnt` contains the number of calls to the function `plug_get_info()`. Thus `plug_get_info()` is modified to:

```
#include "main/plugin.h"

static void min_init (plugDefinition *plug,
                      grabParameter *para, int argc, char **argv)
{
    ...
}

...

static plugDefinition plug_min = {
    "Min",
    PLUG_ABI_VERSION,
    min_init,
    min_init_options,
    min_cleanup,
    min_process
};

plugDefinition *plug_get_info (int cnt, BOOL *append)
{
    *append = TRUE;
    return &plug_min;
}
```

Figure 7.3: The principal structure of a minimal plugin.

```
*append = TRUE;
plugDefinition *def = calloc (1, sizeof(plugDefinition));
*def = plug_min;
def->name = g_strdup_printf ("Min%d", cnt);
return def;
```

Now any number of instances of the plugin are possible.

Plugins in C++ can be realized in the same way as described above. Alternatively one can use the C++ class shown in figure 7.4. By deriving from this class the creation of a plugin instance is also possible. In this case the factory function `plug_get_info()` is modified to

```
*append = TRUE;
ICEWING::Plugin* newPlugin =
        new ICEWING::MinPlugin (g_strdup_printf("C++Min%d", cnt));
return newPlugin;
```

where `ICEWING::MinPlugin` is a class derived from `ICEWING::Plugin`. A C++ variant of the "min" plugin can be found in the iceWing source distribution in the diretory "plugins/min_cxx/".

```
namespace ICEWING {
    class Plugin : public plugDefinition {
    public:
        Plugin (char *name);
        virtual ~Plugin() {};

        virtual void Init (grabParameter *para, int argc, char **argv) = 0;
        virtual int  InitOptions () = 0;
        virtual bool Process (char *ident, plugData *data) = 0;
    };
}
```

Figure 7.4: The class Plugin, which allows the creation of plugins in C++ in a manner suitable for C++.

For simplified creation of new plugins a plugin generator is available: *icewing-plugingen*. icewing-plugingen is a shell script which expects up to three arguments:

```
> icewing-plugingen [-c|-cxx|-cpp] plugin-name short-name
```

If "-c" is given, which is as well the default, a new C-plugin of name "plugin-name" is generated in the current directory. If "-cxx" or "-cpp" is given, a C++-Plugin deriving from the class Plugin is created. For the C-Version function and type names in the generated source start with "short-name". Besides the needed sources for the plugin a Makefile is generated and the plugin is directly compiled for immediate testing. The plugin is kept short and simple, but shows already data observation, easy user interface generation, and rendering of data in a window.

## 7.2 Communication between plugins

Within a main loop iceWing continuously invokes the loaded plugins. The order of the invocation can be determined by the plugins themselves. For this and for further communication of the plugins iceWing offers several facilities. Plugins can exchange data between each other. They can observe the storing of data by other plugins and they can make functions available to other plugins and invoke functions made available. Details about these communication possibilities will now be given.

### Data exchange

In iceWing data consists of a string as an identifier, a reference counter and a pointer to the concrete data. This data element is deposited and made available for other plugins by the function

```
typedef void (*plugDataDestroyFunc) (void *data);
```

```
void plug_data_set (plugDefinition *plug, const char *ident,
                    void *data, plugDataDestroyFunc destroy);
```

The function `destroy()` is invoked when the reference counter of the data has reached zero at the end of a main loop run. For access to data provided by other plugins there are the functions

```
typedef struct plugData {
    plugDefinition *plug;   /* plugin which stored the data */
    char *ident;            /* ident under which the data was stored */
    void *data;             /* the stored data */
} plugData;

plugData* plug_data_get (const char *ident, plugData *data);
plugData* plug_data_get_new (const char *ident, plugData *data);
plugData* plug_data_get_full (const char *ident, plugData *data,
                              BOOL onlynew, const char *plug_name);
```

With `plug_data_set()` one can store several data elements attached to one identifier. With the parameter `data` of `plug_data_get()` one can access them successively. If the value of this parameter is `NULL` then the data element first stored under the identifier is returned. When invoked again with the previously returned pointer the following data element is returned. If `plug_data_get_new()` is used only data elements stored since the start of the current main loop run are returned. Data elements stored during previous runs, which where not freed because of increased reference counts, are skipped. If `plug_data_get_full()` is used, the returned data elements can be additionally restricted to these elements, which were stored by a special plugin.

Every invocation of `plug_data_get()` or one of its variants increments the reference counter of the returned data element. Increasing the reference counter of data, to which a pointer is already available, can be done with the function

```
void plug_data_ref (plugData *data);
```

The references can be released with the function

```
void plug_data_unget (plugData *data);
```

Every successful call to `plug_data_get()` and every call to `plug_data_ref()` requires a succeeding call to `plug_data_unget()` to let the reference count drop again. Finally this leads to an automatic call to the `destroy()` function for releasing the data.

### Observing data

So far it is not clear when iceWing invokes the `process()` functions of the plugins. This is determined by the observation of data provided by other plugins. With the function

```
void plug_observ_data (plugDefinition *plug, const char *ident);
```

a plugin can observe the storage of data with the identifier `ident`. When *new* data with this identifier gets stored, the `process()` function of the observing plugin is invoked.

Data is considered new, if it is stored within the current main loop run. At the start of a main loop run iceWing stores pseudo data under the identifier `"start"`. Every plugin that observes this data will be invoked at the start of every main loop run. These plugins can now store data themselves using their own identifiers to initiate the call of other plugins observing these identifiers. If there are no more plugins registered for the identifiers of newly stored data, the next main loop run is initiated by again storing pseudo data under the identifier `"start"`. The different plugins are invoked sequentially. Only if the `process()` function of the previous plugin is finished the `process()` function of the next plugin is invoked. Even if there were multiple data elements stored under the same identifier the plugins are invoked only once. Plugins that should process all data elements stored under the same identifier must fetch them sequentially with `plug_data_get()`. An alternative approach can be realized with the function `plug_add_default_page()`. See page 53 for more details. By invoking

```
void plug_observ_data_remove (plugDefinition *plug, const char *ident);
```

a plugin can stop its observation of data with a certain identifier.

### Function exchange

The communication method described in the last paragraph is purely data driven. Additionally there is the possibility for plugins to provide their functions to other plugins. This can be achieved by the function

```
typedef void (*plugFunc) ();

void plug_function_register (plugDefinition *plug,
                             const char *ident, plugFunc func);
```

Again with the help of the identifier `ident` a plugin can access the registered function by means of the function

```
typedef struct plugDataFunc {
    plugDefinition *plug; /* plugin which registered the function */
    char *ident;          /* ident under which the function was registered */
    plugFunc func;        /* the registered function */
} plugDataFunc;

plugDataFunc* plug_function_get (const char *ident, plugDataFunc *func);
```

Similar to data storing many functions can be registered under the same identifier. By setting `func` to `NULL` the first function registered under an identifier can be accessed. Consecutive calls to `plug_function_get()` with a previously returned pointer yields the successively registered function. With the function

```
void plug_function_unregister (plugDefinition *plug,
                               const char *ident);
```

a provided function can be withdrawn by the plugin which provided it.

## 7.3 Graphical abilities

The graphical abilities of iceWing can be divided into three groups. The first group incorporates functions for generating a user interface consisting of widgets. The second group contains functions for the display of various data. Furthermore there are functions not classifiable into one of these groups directly. The generation of the graphical interface mainly takes place in the `init_options()` function of each plugin. However, every function described in this chapter can be invoked at any later time as the user wishes. In the following sections these abilities will be now introduced.

### 7.3.1 Generating a user interface

Every widget that could be generated with iceWing functions follows the same philosophy. During the generation of the widget the address of a variable is passed to the generating function. This variable is modified in the background by iceWing, without the need of any help by the plugin. Indeed the plugin has no means to modify the variable directly. Besides this, the screen layout for the widgets is predetermined for the most part. This limitation yields two benefits:

- The creation and management of widgets gets very simple for the plugins.
- There is the possibility of automatically loading and saving widget values. This functionality is completely independent of any support coming from the plugin. Moreover, it is even possible to remotely set the widget values via DACS again without the help of the plugin.

Figure 7.5 gives an overview of all widget types iceWing can create. The "demo" plugin, which was already used during the Quicktour (see section 3.1), uses all these widgets and additionally gives an overview over different render capabilities of iceWing. It can be found in the iceWing source distribution in the diretory "plugins/demo/".

#### Graphical user interface

Widgets can be created on every page on the main window of iceWing, in the context menu of display windows and in the "Settings" window of display windows. Figure 7.1 shows all these possible positions. A new page in the main window of iceWing can be created with the function

```
int opts_page_append (const char *title);
```

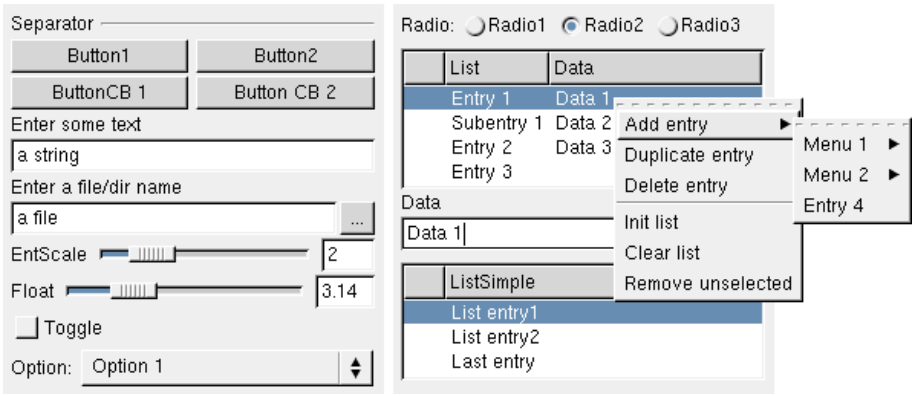



Figure 7.5: All widgets iceWing provides for user interface generation.

`title` denotes the name to be displayed in the "Categories" list. If `title` contains a period or a space (`"."`, `" "`), the "Categories" list will be displayed in a tree structure. The return value of the function `opts_page_append()` is an index that has to be specified during the creation of a widget. The function

```
int prev_get_page (prevBuffer *b);
```

returns the index of the "Settings" window of display windows. This enables the creation of widgets in these windows.

Widgets can be created with different functions for the different widget types. The functions are all named according the pattern `opts_<widgetname>_create()`. Examples of four widget types are

```
void opts_separator_create (long page, const char *title);
void opts_button_create (long page,
                         const char *title, const char *ttip,
                         gint *value);
void opts_entscale_create (long page,
                           const char *title, const char *ttip,
                           gint *value, gint left, gint right);
void opts_float_create (long page,
                        const char *title, const char *ttip,
                        gfloat *value, gfloat left, gfloat right);
```

The parameter list of the functions again follows always the same pattern. `page` denotes the page the widget should appear on. New widgets are always added at the bottom of the specified page. `title` denotes the name of the widget. In the examples in figure 7.5 it was for example `"Enter some text"` for the string widget and `"EntScale"` for the integer slider. This name together with the page name has to be unique throughout the whole program as it is also used as the identifier of the widget itself. The resulting identifier is `"pagetitle.widgettitle"`. With this identifier the

widget is addressable at any later time. Alternatively, to create the same widget, `page` can be set to `-1` and `title` to the complete identifier, i.e. `"pagetitle.widgettitle"`.

If a widget with the identifier `"pagetitle.widgettitle"` already exists, the newly created widget replaces the old one. This allows to change any widget parameter at any time, for example the allowed range of values for a slider.

`ttip` specifies the tool tip of the widget. In `value` the address of a variable has to be given. This variable is modified by iceWing in the background in a dedicated thread. The plugin does not need to care about querying the widget and setting the variable. It can simply use the variable. The remaining parameters specify the valid values of the variable `value`.

Additionally it is possible to set the value of a widget at a later time and to delete a created widget with the two functions

```
long opts_value_set (const char *title, void *value);
gboolean opts_widget_remove (const char *title);
```

`title` denotes the identifier of a widget. In case of an integer the value to be set is passed directly in `value`, otherwise it is a pointer to the value to be set. For example to set the value of the widget `"EntScale"` on page `"demo"` in figure 7.5 to 3 one can use

```
opts_value_set ("demo.EntScale", GINT_TO_POINTER(3));
```

whereas to set the `"float"` widget

```
float newval = 3.0;
opts_value_set ("demo.Float", &newval);
```

must be used.

The current settings of all the different widgets created with one of the `opts_<widgetname>_create()` functions can be loaded and saved to/from files without any plugin interaction in the graphical user interface. See paragraph 4.3 for more details about these files. Sometimes some of these settings should not be saved or loaded. This can be achieved with the functions

```
void opts_defvalue_remove (const char *title);
void opts_save_remove (const char *title);
```

For both functions `title` denotes the identifier of the to be affected widget. The function `opts_defvalue_remove()` prevents, that settings from the configuration files, which are automatically loaded during the start of iceWing, are applied to the given widget. This function must be called before the corresponding `opts_<widgetname>_create()` call. Otherwise the settings were already applied. `opts_save_remove()` prevents that the settings are saved to any configuration files in the first place.

### Nongraphical interface

The automated treatment of variables of widgets for loading and saving can be extended to variables that don't have a widget assigned. This is provided by the functions

```
typedef enum {
    OPTS_BOOL, OPTS_INT, OPTS_LONG, OPTS_FLOAT, OPTS_DOUBLE, OPTS_STRING
} optsType;

typedef void (*optsSetFunc) (void *value, void *new_value, void *data);

void opts_variable_add (const char *title,
                        optsSetFunc setval, void *data,
                        optsType type, void *value);
void opts_varstring_add (const char *title,
                         optsSetFunc setval, void *data,
                         void *value, int length);
```

`title` corresponds to the `title` variable of the widget functions. `value` specifies the address of the variable to be loaded and saved. With `type` their type is declared. If `func` is set to `NULL` then in addition to be saved automatically in the background the variable is also loaded and set automatically. Otherwise the function `setval` with the additional argument `data` gets invoked, if the variable should be modified. This function is then responsible for the modification of the variable. With the function `opts_varstring_add()` one can additionally specify a maximal length for a string, which will be not exceeded during the setting of the string.

### Plugin support

There are certain standardized functions which are (a) controllable by widgets and (b) interesting for a lot of plugins. In addition most plugins need at least one page to place their widgets on. To simplify this there is the following function which creates a new page with two special widgets:

```
typedef enum {
    PLUG_PAGE_NOPLUG        = 1 << 0,
    PLUG_PAGE_NODISABLE     = 1 << 1
} plugPageFlags;

int plug_add_default_page (plugDefinition *plugDef,
                           const char *suffix,
                           plugPageFlags flags);
```

This function creates a new page in the main window of iceWing under the name '`plugDef->name" "suffix`' and returns its index. Furthermore up to two widgets are

created whose functionalities are completely realized by iceWing. The first widget toggles the invocation of the `process()` function of the plugin. If `PLUG_PAGE_NODISABLE` is given in `flags`, this widget is suppressed.

The second widget is created if `PLUG_PAGE_NOPLUG` is not set in `flags`. With this widget one can control which data elements of which other plugins should be passed to the plugin `plugDef`. Normally the function `process()` of the plugin `plugDef` is invoked as soon as data with an identifier observed by `plugDef` is provided by any other plugin. Even though there could be more than one data element available under the observed identifier at the time the plugin is invoked only once. The plugin can get the remaining data elements by using the function `plug_data_get()`.

With the function `plug_add_default_page()` this behavior is changed. If in this case nothing is entered in the second widget, the plugin is invoked separately for all available data elements of all plugin, who's identifier the plugin is observing. When there are names of plugins in the widget, the plugin is invoked only for these plugins. The plugin is not invoked for data elements from other plugins.

Since this widget supplies iceWing with additional information about the interdependencies between the registered plugins, the order of invocation of the plugins can be changed accordingly if necessary. If multiple plugins observe the same identifier `"ident"` normally the order of their invocation is not determined. However, if in one plugin's `plug_add_default_page()` widget is specified that the plugin want's to get `"ident"` from one of these other plugins, then this other plugin is always invoked first. As an example suppose that plugins "A" and "B" both observe `"image"`. If now the user specified in the widget of plugin "A" that "A" should get `"image"` from plugin "B", plugin "B" is always invoked before plugin "A" if data of type `"image"` is available.

For easy creation of names of the form '`plugDef->name`"`suffix`' there is the function

```
int plug_name (plugDefinition *plugDef, const char *suffix);
```

which returns a pointer to a per-plugin-instance string of the form '`plugDef->name`"`suffix`'. This often comes handy for accessing widgets on the page created for example by `plug_add_default_page()` but as well to create pages with `opts_page_append()` or to access widgets on these pages. All widget names in iceWing have to be unique. If these pages use this naming scheme as well, the names are easily made unique. The same is true for the function `prev_new_window()`, where this nameing scheme should be used as well.

### 7.3.2 Graphical display of data

iceWing offers various possibilities for data visualization. Plugins can create any number of data display windows. The user may at any time open and close these windows, scroll and zoom in them, save their contents or display meta informations

about their content. During these actions vector objects like lines and ellipses get correctly redrawn at the desired zoom level. All these actions do not involve the plugins, since the display windows are managed completely by iceWing.

### Window management

New windows can be created by the function

```
typedef struct prevBuffer {
    ...
    guchar *buffer;       /* buffer for the image */
    int width, height;    /* width, height of the buffer */
    GtkWidget *window;    /* window in which the buffer is displayed */
    ...
} prevBuffer;

prevBuffer *prev_new_window (const char *title, int width, int height,
                             gboolean gray, gboolean show);
```

`title` is on the one hand the name of the window and on the other hand an identifier. This identifier together with the names of the pages created with `opts_page_append()` in the iceWing main window has to be unique throughout the whole program. If the name contains a period (`"."`) the windows in the "Images" list in the main window will be displayed in a tree structure. `width` and `height` specify the initial size of the window. The user can resize the window at any time. Specifying `-1` means that the default values for width and height are taken. `gray` determines if the content of the window is displayed in gray or in color. If `show = TRUE` the window is opened instantly after creation. Otherwise the user has to double-click on the window title in the "Images" list to open the window.

Bigger amounts of memory are used not before the window is opened the first time. Every drawing function of iceWing first tests if a window is actually open before drawing and returns immediately if this is not the case. So the consumption of computer resources stays low even if many windows are created and outputs in them are produced without any further checks. Sometimes it is useful to check whether a window is open before any complex computations of output data for the window is carried out. This can be done with `(buffer->window != NULL)`. With

```
void prev_free_window (prevBuffer *b);
```

one can remove and free a window previously created with `prev_new_window()`.

The user can scroll and zoom in the windows at any time. Additionally these actions can be performed by the plugin using the function

```
void prev_pan_zoom (prevBuffer *b, int x, int y, float zoom);
```

The content of window `b` is then displayed at location `(x,y)` with a zoom value of `zoom`. If any of the parameters are below zero, the respective old values are retained.

Some plugins need informations about mouse actions and key presses in a particular window. They can be retrieved with the functions

```
typedef enum {
    PREV_BUTTON_PRESS               = 1 << 0,
    PREV_BUTTON_RELEASE             = 1 << 1,
    PREV_BUTTON_MOTION              = 1 << 2
    PREV_KEY_PRESS                  = 1 << 3,
    PREV_KEY_RELEASE                = 1 << 4
} prevEvent;
typedef void (*prevButtonFunc) (prevBuffer *b, prevEvent signal,
                                int x, int y, void *data);
typedef void (*prevSignalFunc) (prevBuffer *b, prevEventData *event,
                                void *data);

void prev_signal_connect (prevBuffer *b, prevEvent sigset,
                          prevButtonFunc cback, void *data);
void prev_signal_connect2 (prevBuffer *b, prevEvent sigset,
                           prevSignalFunc cback, void *data);
```

If one of the events specified by `sigset` occurs, which is triggered by a mouse button or a key press in window `b`, the function `cback()` with the additional argument `data` is invoked. `cback()` is additionally provided with the event occurred and further information about the event. The function `prev_signal_connect()` only enables the handling of mouse events whereas `prev_signal_connect2()` permits the handling of both mouse events as well as key press events. The effects of zooming and scrolling are completely removed from the passed mouse coordinates.

### The display of graphical objects

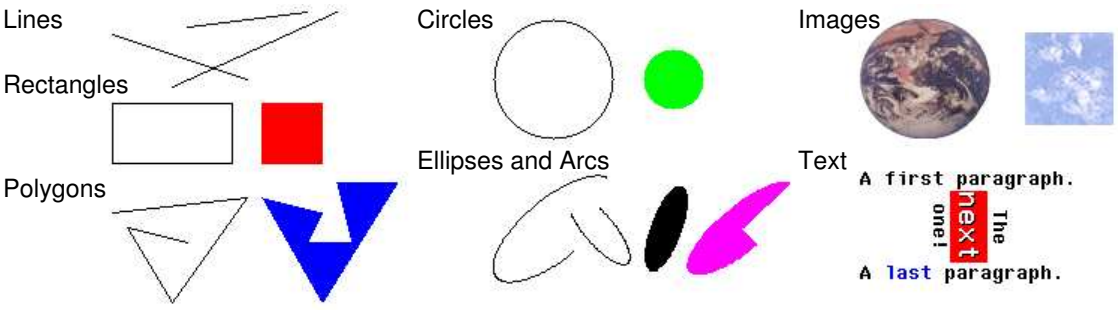



Figure 7.6: All graphical elements iceWing can render.

iceWing provides the possibility to display miscellaneous graphical primitives in the windows created by `prev_new_window()`. Figure 7.6 gives a summary of all primitives iceWing can create. The display of the objects is accomplished in multiple stages. The first step is optional and, if enabled, copies all of the original data which is needed for the display of an object. With this data iceWing can redraw the image without interaction with the plugin – scrolling and zooming in high quality gets possible without the plugins help. Subsequently the objects are displayed in an off-screen buffer using the current scroll and zoom values. This buffer contains the precise section of the image which will be seen in the window and hence can be used for redrawing the window - again without the involvement of the plugin. In a final step the buffer is then drawn into the window.

All objects can be displayed using different functions with a common interface. Two examples of these functions are

```
typedef enum {
    PREV_IMAGE,
    PREV_TEXT,
    PREV_LINE,
    ...
    PREV_NEW = 30
} prevType;

#define RENDER_THICK    (1<<30)  /* use thickness for vector objects */
#define RENDER_CLEAR    (1<<31)  /* clear the buffer */

void prev_render_data (prevBuffer *b, prevType type, const void *data,
                       int disp_mode, int width, int height);
void prev_render_list (prevBuffer *b, prevType type, const void *data,
                       int size, int cnt,
                       int disp_mode, int width, int height);
```

Every render function gets different standard arguments. Window `b` is the window in which the object is displayed. With `width` and `height` one can specify the total size of all objects which are still inside the window. These latter two values are needed to compute the correct zoom factor for the display of the object if the "Fit to window" display mode for the window is chosen. If these values are -1 or `RENDER_CLEAR` is *not* set in `disp_mode`, the last values for the buffer `b` are used instead. With `disp_mode` the rendering can be influenced. If `RENDER_CLEAR` is specified the contents of the whole window is cleared and the internal copies of all parameters of former drawing actions are freed before drawing. `RENDER_THICK` enables the rendering of thicker lines by means of a special setting. See the function `prev_set_thickness()` for more details. The remaining arguments specify the particular object, which should be displayed in the window. With `prev_render_data()` an arbitrary object can be displayed and `prev_render_list()` displays an array of objects of the same type. `type` specifies the type of the objects. `data` is in the first case a pointer to the data of one object and in

the second case a pointer to the data of an array of objects. In the second case `cnt` gives the length of the array and `size` denotes the size of an array element.

Data for objects to be displayed must be specified via structures. For example for lines it is

```
typedef struct {
    iwColtab ctab;
    int r, g, b;
    int x1, y1, x2, y2;
} prevDataLine;
```

The structures for the other available objects are similar. `ctab` determines how `r`, `g` and `b` should be interpreted. For example if `ctab` is set to `IW_RGB` then they are interpreted as a point in the RGB color space, with `IW_YUV` as a point in the YUV color space, or with a pointer to a color table `r` would be interpreted as an index in this color table. Specifying `-1` for `r`, `g`, or `b` has a special meaning. In this case the corresponding color channel is not modified during the rendering. Finally `x1`, `y1`, `x2` and `y2` are the coordinates of the endpoints of the line. To enable subpixel accurate displaying of objects there are `float` data type variants of every vector object structure, for example `prevDataLineF` for lines. For displaying these data elements there are corresponding prevType values like `PREV_LINE_F` for lines.

To simplify the call to `prev_render_list()` there is a wrapper for every object type. For example arrays of lines or images can be displayed also using the functions

```
void prev_render_lines (prevBuffer *b,
                        const prevDataLine *lines, int cnt,
                        int disp_mode, int width, int height);
void prev_render_imgs (prevBuffer *b,
                       const prevDataImage *imgs, int cnt,
                       int disp_mode, int width, int height);
```

In image processing very often one needs to display an image. For the frequent case of 8 bit images there is the function

```
void prev_render (prevBuffer *b, guchar **planes,
                  int width, int height, iwColtab ctab);
```

with which no structure has to be filled in this case. Additionally by setting `RENDER_CLEAR` this function clears the complete window before rendering the image. To render other types of images – iceWing also supports images of type 16 bit int, 32 bit int, float, and double – or to support other forms of flexibility the previously introduced function `prev_render_imgs()` or the different general render functions must be used.

For simplified text output there is additionally the function

```
void prev_render_text (prevBuffer *b,
                       int disp_mode, int width, int height,
                       int x, int y, const char *format, ...);
```

This function invokes `sprintf()` internally and and renders the returned string. The string can contain additional formatting instructions to modify color, type and alignment of the rendering. For example by embedding `<fg="255 0 0" bg="0 0 0" font=big>` it can be switched to a big red font on a black background. Further details about the formatting instructions can be found in the header "Grender.h".

The rendering of objects can be influenced further by the two functions:

```
void prev_set_bg_color (prevBuffer *buf, uchar r, uchar g, uchar b);
void prev_set_thickness (prevBuffer *buf, int thickness);
```

`prev_set_bg_color()` determines with which color the window is cleared if `RENDER_CLEAR` is set. With `prev_set_thickness()` lines are drawn with the specified thickness if `RENDER_THICK` was set during the output of objects.

All functions for drawing objects introduced so far draw the objects incrementally in an off-screen buffer assigned to the appropriate window. In a final step this buffer can be rendered in the associated window using the function

```
void prev_draw_buffer (prevBuffer *b);
```

The steps to follow to render something in a window using iceWing can be summarized as:

| |
|---|
| Create a new window `b` with `prev_new_window()`. This is the necessary first step and can be well placed in the `init_options()` function of the plugin. But any later executed code paths are as well possible. |
| LOOP |
| Render various objects in the off-screen buffer of window `b` using several of the `prev_render_xxx()` functions. `RENDER_CLEAR` must be set before the first function is invoked to clear the window beforehand. |
| Render the off-screen buffer in window `b` by invoking the function `prev_draw_buffer()`. |

Besides this described interface with the `prev_render_xxx()` functions to render objects there is also another interface consisting of `prev_drawXxx()` functions. This is an interface with lower abstraction which for example ignores scroll positions as well as zoom adjustments. Because of this it should not be used most of the time. If this low level interface is necessary nevertheless, further details about it's usage can be found in the according header files.

### 7.3.3 Further graphical functionalities

If the program has to be terminated this should never be managed using the regular ANSI function `exit()`. For this one should always use the function

```
void gui_exit (int status);
```

The direct call to `exit()` can lead to a segmentation violation as well as to a freeze of the complete program. As various graphical abilities are realized in a dedicated seperate thread, an accurate synchronization with this thread has to be performed before termination. This synchronization is ensured by `gui_exit()`.

In iceWing images are managed by the structure

```
typedef enum {
    IW_8U, IW_16U, IW_32S, IW_FLOAT, IW_DOUBLE
} iwType;

typedef struct iwImage {
    guchar **data;      /* the real image data */
    int planes;         /* number of planes available in data */
    iwType type;        /* type of data */
    int width, height;  /* width, height of the image in pixel */
    int rowstride;      /* distance in bytes between two lines */
                        /* if >0: color images are interleaved in data[0] */
    iwColtab ctab;      /* color space used in data */
} iwImage;
```

In this structure image data can be stored in various types and arrangements. With `iwType` the type of `data` is specified ranging from 8 bit unsigned to double. The arrangement of the image data can be *planed* as well as *interleaved*. In a planed arrangement the color planes are provided separately in the fields `data[0]`, `data[1]`, ..., `data[planes-1]`. In an interleaved arrangement `data[0]` contains all the color information where the particular color values of a pixel are juxtaposed in the array. For example for `planes=3` in the RGB color space first the red color value of the first pixel is stored followed by the green and blue values. Subsequently the values of the second pixel are stored in the same way until finally the values of pixel `width*height` are reached and stored in `data[0]`. Images of all these types and arrangements can be created, released, loaded, saved, and also displayed.

There are several functions for managing images of which some of the important ones are:

```
gboolean iw_img_allocate (iwImage *img);
iwImage* iw_img_new (void);
iwImage* iw_img_new_alloc (int width, int height,
                           int planes, iwType type);
void     iw_img_free (iwImage *img, iwImgFree what);
iwImage* iw_img_load (const char *fname, iwImgStatus *status);
iwImgStatus iw_img_save (const iwImage *img, iwImgFormat format,
                         const char *fname, const iwImgFileData *data);
```

Further details about these and several other functions for managing images can be found in the header file “Gimage.h”.

## 7.4 Further abilities

Besides the so far explained abilities iceWing has further functionalities in the graphics as well as in some other areas. In this section some of them will be introduced more closely where some others will not be detailed much. For example in the header file "output.h" there are functions to ease the use of DACS. These include functions for image output, for the output of status messages and for the output of general data via streams. Furthermore, it is possible to make functions available for RPC communication via DACS. All these functions manage the registration with DACS, the error handling and the generation of unique stream names and function names.

Further examples of useful functions include "session management" or functions that allow the registration of new graphical primitives that can then be used with the general rendering functions. Further details of these as well as the so far presented functions can be found in the appropriate header files of iceWing.

### The "grab" plugin

A central plugin is the in iceWing integrated plugin "grab". It enables the import and the delivering of a series of images for further processing to other plugins. The images can be imported from a grabber (controlled by Video4Linux 2 or FireWire under Linux and MME on Alphas), from DACS, if encoded in the `Bild_t` format, or from files of various pixel formats. Once "grab" is invoked by iceWing it imports the next image and makes it available for other plugins under the ident "image" using the function `plug_data_set()`. For the image the YUV color space is used. The structure that "grab" uses to provide the data is derived from iwImage, i.e. it first contains the image and afterwards different additional data. In detail, the structure is:

```
typedef struct grabImageData {
    iwImage img;         /* the image data */
    struct timeval time; /* time the image was grabbed */
    int img_number;      /* consecutive number of the grabbed image */
    char *fname;         /* image read from a file? -> name of the file */
    int frame_number;    /* image from video file -> frame number, else 0 */
    float downw, downh;  /* down sampling, which was applied to the image */
} grabImageData;
```

The image is allways encoded in the planed format, i.e. the color planes are provided separately in the fields `data[0]`, `data[1]`, ..., `data[planes-1]` of the `iwImage` structure. Additionally the plugin can provide the imported images via a DACS stream in the `Bild_t` format. By means of a DACS-function the current or optionally also past images can be fetched from other programs in the `Bild_t` format. If there is an observer for the identifier `''imageRGB''`, then additionally the current image is provided in the RGB color space under this identifier using the function `plug_data_set()`. For the storing of the data associated with "imageRGB" again the `grabImageData` structure is used.

### Auxiliary functions

"tools.h" provides several smaller auxiliary routines that are frequently needed during program development. These include functions for the output of errors, warnings, debug messages and functions for testing assertions:

```
void iw_debug (int level, const char *str, ...);
void iw_debug_1 (int level, const char *str);
void iw_debug_2 (int level, const char *str, ARG1);
...
void iw_warning (const char *str, ...);
void iw_warning_1 (const char *str);
void iw_warning_2 (const char *str, ARG1);
...
void iw_error (const char *str, ...);
void iw_error_1 (const char *str);
void iw_error_2 (const char *str, ARG1);
...
void iw_assert (scalar expression, const char *str, ...);
void iw_assert_1 (scalar expression, const char *str);
void iw_assert_2 (scalar expression, const char *str, ARG1);
...
```

All these functions invoke `sprintf()` internally. The functions `iw_debug_xxx()` and `assert_xxx()` only produce code if the macro DEBUG is set during compilation. `iw_debug_xxx()` only produces output if the value of `level` is smaller than `talklevel` which can be chosen via the command line of iceWing (option "-t", see page 4.1). `iw_warning_xxx()` always outputs its message, whereas `iw_error_xxx()` additionally terminates the execution of the program. The function variants without a number, i.e. `iw_debug()`, `iw_warning()`, `iw_error()`, and `iw_assert()`, allow an arbitrary number of arguments, but for full functionality they need either GCC or another compiler compatible with ANSI-C99. Otherwise these variants are implemented as functions instead of macros, which provide less information than the macro variants.

Additionally there are several functions for meassuring the execution time of program parts. Some of them are:

```
int  iw_time_add (const char *name);
void iw_time_start (int nr);
long iw_time_stop (int nr, BOOL show);

#define iw_time_add_static(number,name) ...
#define iw_time_add_static2(number,name,number2,name2) ...
```

`iw_time_add()` creates a new timer and returns an index to it. Using this index it can be started with the function `iw_time_start()` and stopped with `iw_time_stop()`. With `show = TRUE` the passed time is printed to `stdout` immediately. Otherwise this is done after a certain number of iceWing main loop runs. With

iw_time_add_static() the initialization can be simplified. With this function a new static variable named number is defined. This variable can be initialized by invoking iw_time_add() one time. An example usage scheme is:

```
/* Definition of other variables */
...
iw_time_add_static (time_demo, "Demo Messung");

iw_time_start (time_demo);
/* Execution of the program part to be measured */
...
iw_time_stop (time_demo, FALSE);
```

The analysis of command line arguments can be simplified by the function:

```
char iw_parse_args (int argc, char **argv, int *nr, void **arg,
                    const char *pattern);
```

This function tests if argv[*nr] occurs in pattern where pattern is scanned without case sensitivity. Subsequently *nr is incremented appropriately so that with the next invocation of iw_parse_args() the next argument can be analysed. The EBNF format of pattern is { "-" token ":" ch ["r"|"ro"|"i"|"io"|"f"|"fo"|"c"] " " }, where token is an arbitrary string without " " and ":" and ch is an arbitrary character. If -token is found, ch is returned by the function. The remaining optional characters in pattern define some modifiers. "r" specifies that an additional argument is needed which will be returned using the variable arg. "i" and "f" mean that an additional integer argument or float argument is needed, respectively. "c" specifies that token can be continued arbitrarily. This continuation is returned using the variable arg. Finally "o" denotes, that the string, integer, or float argument is optional. The application of iw_parse_args() is best explained using an example:

```
void *arg;
char ch, *str_arg;
int nr = 0, int_arg;

str_arg = NULL;
int_arg = 0;
while (nr < argc) {
    ch = iw_parse_args (argc, argv, &nr, &arg,
                        "-I:Ii -S:Sr -H:H -HELP:H --HELP:H");
    switch (ch) {
        case 'I':
            int_arg = (int)(long)arg;
            break;
        case 'S':
            str_arg = (char*)arg;
            break;
        case 'H':
        case '\0':
```

```
            help();
        default:
            fprintf (stderr, "Unknown character %c!\n", ch);
            help();
    }
}
```

## 7.5 Using external libraries

To ease interfacing with the OpenCV computer vision library [Int05] there are some auxiliary functions in the header file "opencv.h" in the "tools" directory. The functions from this file only work if the plugin which calls these functions links against the OpenCV libraries. Otherwise a symbol from the OpenCV libraries can not be resolved.

OpenCV uses the `IplImage` structure for image representation. To convert to and from the iceWing `iwImage` type the three functions

```
IplImage* iw_opencv_from_img (const iwImage *img, BOOL swapRB);
iwImage*  iw_opencv_to_img (const IplImage *img, BOOL swapRB);
iwImage*  iw_opencv_to_img_interleaved (const IplImage *img, BOOL swapRB);
```

can be used. All these functions first allocate a new image and then copy the provided image to the newly allocated image. To free the returned image the functions `cvReleaseImage()` for `IplImage` images and `iw_img_free (image, IW_IMG_FREE_ALL)` for `iwImage` images must be used. With the flag `swapRB` on the fly conversion between RGB and BGR can be performed. If this flag is set to `TRUE`, the planes 0 and 2 are swapped during conversion.

To display `IplImage` images in a preview window the function

```
void iw_opencv_render (prevBuffer *b, const IplImage *img, iwColtab ctab);
```

is available. The function clears the buffer by using `RENDER_CLEAR` and subsequently displays `img` in `b` with the color transformation `ctab`. This function does not create a new copy of the image and thus is as fast as the iceWing "native" render functions `prev_render()` and `prev_render_imgs()`. Remember that you must use `prev_draw_buffer()` to finally see the result on the screen.

# Bibliography

[Int05] Intel. OpenCV – Open Source Computer Vision Library, 2005. http://www.intel.com/research/mrl/research/opencv/.

[Jun98] Nils Jungclaus. *Integration verteilter Systeme zur Mensch-Maschine-Kommunikation.* Dissertation, Universität Bielefeld, Technische Fakultät, 1998.

[Löm04] Frank Lömker. *Lernen von Objektbenennungen mit visuellen Prozessen.* Dissertation, Universität Bielefeld, Technische Fakultät, 2004. http://bieson.ub.uni-bielefeld.de/volltexte/2004/549/.

[Ous94] John K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, March 1994.

[The03] The Mathworks. Matlab, 2003. http://www.mathworks.com.

# Index