The

VRR

Programmer's Manual

Table of Contents

1 Introduction

VRR (a Vector-based gRaphic editoR) is an application designed especially for creating illustrations of mathematical articles.

1.1 About this manual

This book does not contain instructions how to use VRR, for that purpose consult *The VRR User's Manual*. Instead, it covers the principles behind the source code and allows a programmer to get familiar with VRR sources easily. Furthermore, almost all source files and headers are heavily commented.

The Programmer's Manual is not the reference documentation of all VRR libraries and modules. For that purpose, consult the reference manual prepared from source code. The source code documentation is extracted using the Doxygen tool into HTML to allow fast navigation through function descriptions, structure index, etc. See Chapter 14 [Documentation], page 107 for details.

However, as VRR is under constant development, some of the contents of this book can easily get outdated, because there is always some delay between changing the source code and source code documentation and between changing the Programmer's Manual. Fortunately, the project design is reasonably robust and therefore we expect the design changes in the future to be only minimal.

1.2 Developers' center

The VRR project main web site is at http://vrr.ucw.cz/. Here you can find current news regarding the project, new releases, documentation and links to miscellaneous files.

The developers use a mailing list (the VRR list), so if you are interested in programming VRR or would just like to ask a few questions, feel free to contact the developers at vrr@ucw.cz. Currently, the list languages are Czech and English, as there are now no non-Czech members, but this can change in the future in favour of English-only.

At the time of writing this book, this is the list of active developers (a.k.a. The VRR Team):

- Martin "MJ" Mareš (project advisor)
- Jirka Fink – Kernel
- Pavel Charvát – GEOMLIB, parts of VRRLIB
- Eva Ondráčková – GUI
- Tomáš Valla – FONTLIB, parts of Kernel, PS/PDF export, IPE5 import, VRRLIB
- Zuzka Vlčková – parts of GUI, SVG export/import
- Ondřej "Santiago" Zajíček – VCL, Visualisation, Scheme, Save & Load

So if you wish to join us, contact our mailing list. We welcome any support.

1.3 Project background

1.3.1 The original idea

The VRR project started as a school software project at the Faculty of Mathematics & Physics, Charles University, Prague, Czech Republic. The work begun in winter 2003 and at the beginning was heavily influenced by the IPE vector editor. In 2003, IPE (version 5.0) was terribly outdated and incompatible with X Window system libraries, resulting in frequent program crashes and errors. But there were great ideas behind IPE's logical design, so we decided to create "our own IPE" enhanced by a powerful scripting language and many other useful features. So we formed the VRR Team, applied with VRR as a school software project and started to work.

However, in the spring of 2004, surprisingly the new IPE version 6.0 appeared, correcting most of its nastiest bugs. We therefore changed our mind and redesigned VRR features (fortunately not much was written at that time) into a new and original vector editor, today overwhelming IPE in most ways.

1.3.2 Development history

The development history is as follows:

- October 2003: Specification, the first assignment of tasks to programmers.
- November 2003: The choice of programming language, auxilliary development tools (CVS, Doxygen, ...). Studying of various graphical formats and the Guile library. GUI design proposition. Kernel data structures design proposition. The choice of supported graphical objects and supported operations.
- December 2003: The beginning of the actual coding. Coding style choice. Extending ideas of graphical objects and operations. Searching for suitable algorithms for propagation of transformation changes of dependent objects. We suggested to use cascade style-sheets to set object properties and style.
- January 2004: Undo history design. We decided to use transaction mechanism for change tracking or kernel data structures.
- February – March 2004: We precised the kernel data structures design, chose the representation of dependence relations between graphical objects. We also precised the GUI look and feel.
- May 2004: Programming. We implemented a large part of GEOMLIB, some kernel data structures, GUI backbone. Renderer design. We linked the Scheme library with kernel, wrote scheme console in GUI.
- June – September 2004: Holidays.
- September – October 2004: We implemented more kernel data structures. Linked renderer with GUI. Implemented R-Tree as a search data structure in GEOMLIB. We also did a massive source code cleanup. The "snarfing" system for generating Scheme function headers was introduced. A topological sorter for geometric recomputations.
- November 2004: The design of virtual properties. A DVI parser for TEX output processing. Redesign of the transaction mechanism. Context evaluations in the GUI Command Structure. Basics of PS, EPS, and PDF export.
- December 2004: Preparations for the first public release. Implementation of the GO Factory and snap. Questions about font identification. TEX text processing. The choice of the FreeType library for text rendering.
- January 2005: The first pieces of User's Manual. The first public release, version 0.5. Santiago's traneformation tool.
- February 2005 – April 2005: We added the support for colors. A new transformation tool. GO groups and paths. Property generator macros. Problems with numeric errors for transformations of many objects which caused major design changes in kernel. The choice of libpaper for export paper sizes. IPE5 import. The plugin mechanism.
- April 2005: Another public release, version 0.6.
- May – June 2005: SVG import/export. Major redesign of some parts of GUI. A CVS branch for huge kernel changes including changes in the kernel interface and thus all the dependent modules. The choice of Texinfo as a documentation tool. GEOMLIB and VCL did undergo some design changes, too.
- June 2005: Another two public releases, versions 0.7 and 0.8. A http://freshmeat.net/ announce (which was mentioned later in an article at http://root.cz/). A public presentation of the project at the Department of Applied Mathematics.

- July 2005 – August 2005: The choice of Bugzilla as a bug-tracking system. Debugging, documentation. Implementation of many minor features. The Scheme interface was replaced by a new one. Another public release, version 0.9.
- September 2005: The anticipated presentation.

VRR is a free software project developed under the GNU Public License. The authors didn't get paid even one crown for this. :) See Appendix A [License], page 110 for the distribution and usage terms.

1.3.3 The present situation

We have managed to create an editor that enables easy creation of mathematical drawings, which was our main aim. VRR offers a unique combination of these main features:

- **Geometric dependencies** which can be easily created by snap. In some well-known geometric design tools, you can create much more complex geometric constructions using projections and some other features that VRR does not have. But in VRR you can create partially dependent objects which can then be transformed, free them from dependencies or reorganize the dependency structure completely – you have much more freedom if you want to create a not very geometric image.
- **TEX texts**, a crucial feature for mathematical drawings. Unlike other editors, VRR enables you to create and view the resulting text instantly, you can even apply transformations to it. The whole image can be exported to PostScript, which is also a necessity for mathematical drawings.
- **Various graphic features** – they are not very many and could be improved vastly in VRR. VRR does not support fancy stroke and fill styles, gradients, ... as vector graphic editors usually do. However, such features are usually not used in illustrations of mathematical papers; for that, VRR's graphic capabilities are quite sufficient.

We can now see almost infinitely many possibilities how to improve VRR, which features to add ... Thus we could keep ourselves occupied for the next ten years. Regrettably, we need to submit VRR and work on some other things as well. Anyway, we are going to maintain VRR in the future; we enjoy the users' feedback that we already have and hope that our work will be useful.

1.3.4 Acknowledgement

We would like to thank the staff and students of the Department of Applied Mathematics, MFF UK, for allowing us to use the department computers and moreover for serving as very useful betatesters, and thus partially supporting our work.

We also owe many thanks to our advisor, Martin Mareš, for a very attentive supervision, numerous comments and ideas, and invaluable advice.

2 Development tools

This chapter documents the background of the source code. This means: how the source code is maintained, what standards of programming languages we used, how we report bugs, the system of source compiling and the list of external programs and libraries.

2.1 Source tree

All source files are stored in a CVS repository at the server '`atrey.karlin.mff.cuni.cz`', in the '`/akce/projekty/vrr/CVS`' directory. All developers have access to this repository. Currently, there is no public access, but we plan to change it in the future.

The current VRR stable release sources should always be available at `http://vrr.ucw.cz/` as '`.tar.gz`' or '`.tar.bz2`' files.

Every change of the repository (the CVS `commit` command) is sent into the VRR mailing list (see Section 1.2 [Developers' center], page 1) as a `diff` data and the CVS log message. This allows developers to keep active eye on other's work.

2.2 Bug tracking system

The primary way to report bug is to send e-mail to VRR mailing list (see Section 1.2 [Developers' center], page 1). Long time we used the file '`TODO`' in the root source directory to maintain the list of bugs in various status.

Then it became clear, that a '`TODO`' is not enough and we installed the Bugzilla bug tracking system, available at `http://vrr.ucw.cz/bugzilla/`. All developers are granted access and can browse list of their bugs, change their status, write comments, report new bugs, etc.

All traffic from Bugzilla is mirrored into VRR mailing list and it is possible to reply to Bugzilla messages via e-mail. Just reply to `vrrzilla@atrey.karlin.mff.cuni.cz` instead of VRR list address.

2.3 Project building system

In this section is described, how VRR is being configured and compiled.

2.3.1 Directory structure preview

After getting sources, the following directory structure should appear in the directory '`vrr`':

```
drwxr-xr-x   3 tom users    4096 Aug 28 11:59 build
-rw-r--r--   1 tom users   17992 Jan 13  2005 COPYING
drwxr-xr-x   6 tom users    4096 Aug 28 14:40 doc
drwxr-xr-x   3 tom users    4096 Jul 19 19:03 examples
drwxr-xr-x   3 tom users    4096 Aug 29 13:45 export
drwxr-xr-x   3 tom users    4096 Aug 27 11:10 font
drwxr-xr-x   4 tom users    4096 Aug 29 13:36 geomlib
drwxr-xr-x   4 tom users    4096 Aug 28 17:12 gui
drwxr-xr-x   3 tom users    4096 Aug 28 18:14 import
-rw-r--r--   1 tom users    1616 Jun 24 18:26 INSTALL
drwxr-xr-x   3 tom users    4096 Aug 29 13:43 kernel
drwxr-xr-x   3 tom users    4096 Aug 25 22:49 lib
-rw-r--r--   1 tom users    6479 Aug 28 14:39 Makefile
drwxr-xr-x   3 tom users    4096 Jul 25 14:04 plugin
-rw-r--r--   1 tom users    3873 Jun 26 11:01 README
drwxr-xr-x   3 tom users    4096 Aug 25 18:30 scheme
-rw-r--r--   1 tom users    3722 Jul 25 14:03 TODO
drwxr-xr-x   3 tom users    4096 Jul 29 18:21 vcl
```

2.3.2 Configure script

VRR uses GNU Autoconf configure script to setup build conditions. It is placed in the '`build`' directory.

Generated script is in the file '`configure`' (which should be always present in distributed sources), its Autoconf source code is in '`configure.in`'.

- The files '`config.h`' (with Autoconf source '`config.h.in`') and '`types.h`' (Autoconf source '`types.h.in`') are intended to be included by main VRR header, '`lib/lib.h`'.
- The file '`makeconfig`' (source '`makeconfig.in`') is included by root '`Makefile`' as a build configuration.
- The file '`path.h`' is there to allow project modules to include the installation path string.

Currently, these checks are implemented in '`configure.in`':

- Library availability and check for the required version. See Section 2.6 [External programs], page 7 for the list of required and optional libraries.
- Computer endianity test.
- Compiler and some of the auxiliary tools check.
- C language data types size detection. The detected types are written into '`types.h`'.

Don't forget to change VRR version in '`configure.in`' and to rebuild '`configure`' script by Autoconf with every new release.

2.3.3 Makefiles

VRR uses GNU Make and sophisticated '`Makefile`' files structure for building process. In the root directory, there is the main '`Makefile`' containing the bulk of building rules and in almost every subdirectory there is a local '`Makefile`'.

The main difference between VRR and other projects using many '`Makefile`'s is that `make` doesn't get called recursively for every subdirectory. Instead, every local '`Makefile`' is included into the main '`Makefile`' (and '`Makefile`' from subsubdirectory gets included into subdirectory '`Makefile`', and so on). Thus it is possible to use all rules and variables from the root '`Makefile`', speeding and simplifying the compilation process reasonably.

Another significant difference is that all binaries and object files are not created among the sources, but rather in the directory '`obj`' in a directory structure resembling the original source tree. Also, all binaries and other data is then installed in the '`run`' directory.

In the root '`Makefile`', there are many automating rules like handling C sources, linking of binaries and copying data files. Thus only little work is needed to compile program, link library, etc.

The typical local '`Makefile`' looks as follows:

```
DIRS+=export
ifndef POTEMKIN
PROGS+=obj/export/zpipe
endif

EXPORT_MODS = \
     pdf \
     ps \
     svg

obj/export/svg.o: CFLAGS+=$(XML_CFLAGS)
obj/export/zpipe: $(Z_LIBS)

$(LIBEXPORT):$(addsuffix .o,$(addprefix obj/export/,$(EXPORT_MODS)))
```

In every subdirectory '`Makefile`', you should add the directory into the `DIRS` variable. If there are executable programs, add them into the `PROGS` variable, with the full path inside the '`obj`'

directory. There are also three destination variables `BINDIR`, `LIBDIR` and `DATADIR`. Modifying them causes different destination directory.

The source files dependencies are handled automatically by the compiler. During compilation, the compiler dependency output is saved, processed with the `build/mergedeps` script and included into the main '`Makefile`'.

Here are the '`Makefile`' targets the programmer is encouraged to use.

- `local`: Do a local build to the directory '`run`'. This is also the default target.
- `config`: Just runs `build/configure` with default settings.
- `final`: Compile project with different directory settings for system installation.
- `dust`: Remove auxiliary files (backups, etc) created by various editors.
- `clean`: Do a cleanup by deleting '`obj`' directory, binaries in '`run/bin`' and '`build/path.h`' path setting.
- `distclean`: Do better cleanup by also deleting reference documentation and configuration.
- `totalclean`: Remove everything deletable.
- `doc`: Run `doxygen` to create reference documentation, available at '`doc/reference`'.
- `tags`: Create '`tags`' and '`TAGS`' files to be used by popular editors.

2.4 Main programming language

VRR is written in pure C language according to C99 standard. For details about C99 standard see http://vrr.ucw.cz/doc/c99.pdf.

In the Linux world, there is a wide-spread GCC compiler and the whole project has been developed using in it. The default language standard, as set in the root '`Makefile`', is C99 with GNU extensions provided by `gcc`. However, VRR conforms to C99 and the compilation in pure C99 can be enabled (see '`Makefile`' header). Thus, any compiler correctly implementing C99 should be able to compile VRR.

There is natural question, why we chose the C language instead of some object-oriented language like C++. Our answer is that C language is standard in the UNIX world, the C compilers nowadays produces more efficient code and the C language give you better control of what happens in your code. On the other hand, the C language imposes a lot of programmer's effort when implementing the object hierarchy, as we do in GEOMLIB or Kernel.

2.5 Scripting language

Scheme language is the scripting language of VRR. Also some small parts of VRR are written in Scheme, mostly initialization routines and handy shortcuts, but also save/load mechanism.

Scheme is interpreted using Guile library, which is a wide-spread Scheme library designed to be used as extension (scripting) language in other applications.

There are several reasons for using Scheme as a scripting language for a program like VRR: Scheme is a dialect of Lisp featuring simplicity and clean design, which makes it pretty easy to learn. Scheme is also often used in computer science curricula. Scheme has standard way to express structured data and integrated parser and writer of that form.

To learn Scheme see the Structure and Interpretation of Computer Programs book (available online at http://mitpress.mit.edu/sicp/full-text/book/book.html). The language standard can be found at http://www.schemers.org/Documents/Standards/R5RS/HTML/.

For more about Scheme cooperation with VRR see Chapter 13 [Scheme], page 104.

2.6 External programs

There is number of libraries VRR utilizes, as well as some auxiliary external programs used during compilation. The programs used to prepare VRR documentation are not listed here. For that purpose see Chapter 14 [Documentation], page 107.

2.6.1 Libraries

VRR utilizes the following external libraries. Make sure the development variant of packages (library headers) are installed on your system.

2.6.1.1 GTK+ library

Version at least 2.6.0 of GTK+ library is required. All fragments of GUI (see Chapter 7 [GUI], page 58) are written under GTK+, as well as some rendering routines (see Chapter 8 [VCL], page 75).

2.6.1.2 Guile library

Version at least 1.6 is required. Guile is an interpreter of the Scheme programming language. See Chapter 13 [Scheme], page 104 for details how VRR utilizes Guile.

2.6.1.3 LibKPathSea library

The library's fundamental purpose is to return a filename from a list of directories specified by the user, similar to what shells do when looking up program names to execute. The purpose in VRR is to search for TEX fonts and other files in various stages of TEX texts compilation and rendering. See Section 12.1 [DVI import], page 102 for details.

2.6.1.4 FontConfig

Version at least 2.3.1 is required. FontConfig is a library designed to provide system-wide font configuration, customization and application access. VRR uses FontConfig to search for installed fonts to be used by standard text objects (see Chapter 9 [FONTLIB], page 91) and for font substitution when requested font is not available (see Chapter 12 [Import], page 102 and Chapter 13 [Scheme], page 104).

2.6.1.5 Zlib library

The Zlib library is a general purpose data compression library. In VRR, it is used in PDF export routines to compress PDF data streams (see Section 11.2 [PDF export], page 101).

2.6.1.6 LibXML library

Version at least 2.0 is required. The LibXML library is used to parse XML files. As the SVG is an XML data format, the library is used during SVG export and import. See Section 11.3 [SVG export], page 101 and Section 12.3 [SVG import], page 102.

2.6.1.7 LibPaper library

The LibPaper library is optional, the configure script (see Section 2.3.2 [Configure script], page 5) is able to configure VRR without LibPaper. The library is used in GUI for comfortable paper format selection (see Chapter 7 [GUI], page 58).

2.6.1.8 FreeType library

FreeType is a software library that can be used by all kinds of applications to access the content of font files. Version exactly 2.1.9 is required. The FreeType library is used in somewhat nonstandard way and a copy of library sources is distributed along with VRR sources. The library is used to do font rendering and various other font manipulation. See Chapter 9 [FONTLIB], page 91.

2.6.1.9 Cairo library

The Cairo library is optional and must be explicitly enabled during installation. The Cairo library can be used as alternative drawing backend in VCL (see Chapter 8 [VCL], page 75). It brings anti-aliased lines and alpha-blending toVRR.

2.6.2 Other tools

2.6.2.1 GNU make

Project building utility. See Section 2.3.3 [Makefiles], page 5.

2.6.2.2 Autoconf

Autoconf is a tool used to build '`build/configure`' script. See Section 2.3.2 [Configure script], page 5.

2.6.2.3 GNU awk

The `gawk` is used only during compilation in the "snarfing" process to build Scheme bindings from C sources. See Chapter 13 [Scheme], page 104 for details.

2.6.2.4 Perl

Perl is used only in the '`build/mergedeps`' script to maintain Makefile dependence informations for project building. See Section 2.3 [Project building system], page 4.

2.6.2.5 pdfTEX

TEX is used to compile TEX text objects (see Chapter 6 [Kernel], page 41). *XXX: podrobnejsi odkaz* Actually, pdfTEX himself is not used, only the vector versions of TEX fonts and libkpathsea search databases are needed. However, there does not exist a separate vector TEX fonts package, so VRR requires whole pdfTEX, which includes the standard TEX by default.

3 Project structure overview

VRR consists of these main parts.

They are described in details in the following chapters. Here we supply a short overview of every part.

3.1 VRRLIB

The basic modules, intended to be used by the whole project. There are things like logging and debugging, memory allocation and general data structures.

3.2 GEOMLIB

GEOMLIB is the geometrical library of VRR project. It implements many numerical algorithms with Bézier curves, general geometrical objects, planar search data structures and many other features. From the beginning, it was designed as a standalone library with no other dependency inside VRR (with the exception of the VRRLIB).

3.3 Kernel

Kernel is the core of VRR project, implementing main VRR data structures, graphic objects, transaction mechanism, undo history and also many computations with graphic objects.

3.4 GUI

Graphical User Interface communicates with the user. GUI uses the services of Kernel to manipulate with objects, maintain data structures and perform various operations, and with VCL to render them on screen. It also calls other parts of VRR like export and import modules. GUI heavily uses the GTK+ library.

3.5 VCL

VRR Canvas Library (VCL) is the set of low-level rendering routines, as well as high-level graphical engine, containing various object expansion caching. Most of the low-level algorithms are hand-written, the rest is performed by the GDK library, part of GTK+, or optionally by the Cairo library.

3.6 FONTLIB

FONTLIB is the font rendering and manipulation library. It contains support for rendering PostScript Type1 fonts and TrueTypes, computing text bounding boxes and converting font among these formats.

3.7 Plugins

This is the VRR plugin mechanism, allowing a programmer to write separated modules (which are actually ELF dynamic libraries), that is possible to load (and sometimes also unload) in runtime. In this chapter we give the interface description, as well as some tips & tricks.

3.8 Export

VRR is able to fully export pictures in PostScript, Encapsulated PostScript, PDF (conforming to level 1.5) and SVG. PostScript and PDF export routines are hand-written, SVG export uses the LibXML library to handle the native SVG's XML format.

3.9 Import

VRR supports importing large subset of the SVG image data format. There is also experimental support for a subset of the IPE version 5.0 native image format.

3.10 Scheme

VRR has an integrated scripting language. It is based on the GUILE library, a Scheme language interpret. The connections between VRR and GUILE are described in this chapter.

4 VRRLIB

VRRLIB is the basic VRR library and is used by all VRR modules. Linking binaries and libraries with VRRLIB is automatical and you don't need to specify it manually in the Makefile. VRRLIB resides in the '`lib`' directory.

4.1 Main project header

Every C source file and header *must* include the file '`lib/lib.h`'. This file contains basic definitions, datatypes and functions, which should be common for all modules.

The programmer can be sure that he gets the following types always defined correctly (by the `build/configure` script, see Section 2.3.2 [Configure script], page 5).

- `byte`: Exactly 8 bits, unsigned.
- `u8`: Dtto.
- `sbyte`: Exactly 8 bits, signed.
- `s8`: Dtto.
- `word`: Exactly 16 bits, unsigned.
- `sword`: Exactly 16 bits, signed.
- `u16`: Exactly 16 bits, unsigned.
- `s16`: Exactly 16 bits, signed.
- `u32`: Exactly 32 bits, unsigned.
- `s32`: Exactly 32 bits, signed.
- `uns`: At least 32 bits.
- `u64`: Exactly 64 bits, unsigned.
- `s64`: Exactly 64 bits, signed.
- `addr_int_t`: For converting pointers into integers.
- `real`: Preferred floating-point type

In the header there also useful macros (for example for handling complicated `structs`), memory allocation prototypes (see Section 4.3 [Memory allocation], page 12) and logging and debugging function prototypes (see Section 4.2 [Logging and debugging], page 11).

4.2 Logging and debugging

This stuff is implemented in the '`lib/log.c`' module and by default prototyped in '`lib/lib.h`'. There are two levels of debugging. Global debugging is turned on if there is the macro `DEBUG` defined. The local debugging should be turned on and off for every module separately by defining the macro `LOCAL_DEBUG`.

These functions should be used for debugging and informative printing purposes, the format is the same as in the `printf` function.

- `msg`: Print message to stdout.
- `err`: Print message to stderr.
- `die`: Print message to stderr and abort the process.
- `DBG`: Macro doing the same job as `msg`, but only if `LOCAL_DEBUG` is defined.
- `DBGLN`: Dtto, but with source line number prepended.
- `BUG`: Abort process.

The wide usage of `ASSERT` macro is recommended.

4.3 Memory allocation

For the memory allocation purposes, the programmer *must* use the following allocator interface which is prototyped in '`lib/lib.h`' (see Section 4.1 [Main project header], page 11). The functions are protected against low memory resulting in process abort on failure and having the same meaning as the original without the `x` prepended. We mention functions like `xmalloc`, `xfree`, `xrealloc`, etc., consult '`lib/lib.h`' for details.

There is also a "small" memory allocator, intended to be used when frequently allocating small memory pieces. See the '`lib/smalloc.h`' header for interface. The implementation is in '`lib/smalloc.c`'.

4.4 Sorter

The array sorter resides in '`lib/arraysort.h`'. This is not a normal header file, it is a generator of sorting routines. Each time you include it with parameters set in the corresponding preprocessor macros, it generates an array sorter with the parameters given. Consult the '`lib/arraysort.h`' header for details. The actual sorting algorithm used is a clever modification of the well-known QuickSort.

The main reason of implementing general sorting routines like this is that it is completely callback-free, thus speeding array sorting considerably.

4.5 Data structures

VRRLIB implements several very useful general data structures.

4.5.1 Hash table

There are two universal generators of hash table routines, one in '`lib/sht.h`' and one in '`lib/hashtable.h`'. These are not normal header files, these are generators of hash tables. After the inclusion, a unique set of hashing routines is generated, with properties depending on symbols defined before the inclusion. See the files' header for details how to use it.

The reason for implementing general hashing routines like this is that it is completely callback-free, thus speeding them considerably. Moreover, it gives the programmer a control of what routines are generated and what exactly they shall do.

4.5.2 AVL-Tree

The VRRLIB implements the AVL-Tree data structure for effective manipulation with ordered sets. In each node of this binary tree, the height of the left and the right subtrees differ at most by one. It can be easily prooved that the height of such a tree is $O(\log n)$.
The following example describes a basic usage:

```
/* normal header file */
#include "lib/avltree.h"

/* node structure (1) */
struct mynode {
  struct avlnode avlnode;
  int key;
};

/* code generator */
#define AVLTREE_TYPE int
#define AVLTREE_KEY(x) ((mynode *)x)->key
#include "lib/avltreegen.h"

int main(void)
{
```

```
    struct avltree tree;
    struct mynode node;

    /* structure initialization */
    avltree_init(&tree);

    /* insertion example (2) */
    node.key = 100;
    avltree_insert(&tree, &node.avlnode);

    /* ... */
}
```

Two header files have to be included. The first (‘`lib/avltree.h`’) is a classical header file with independent structure definitions and function headers. The second one (‘`lib/avltreegen.h`’) is somewhat special. Is reads macros as parameters to generate a specialized code of some functions. The full description of supported macros can be found in the first header file.

Each inserted item must contain the `avlnode` as a substructure (1) and is accessed through its address (2) in AVL-Tree routines. For each tree, there must be also one instance of the `avltree` structure with a pointer to the root.

The implementation includes many standard functions for ordered sets like data insertion, deletion or key searching with usually logarithmic time complexity. If no key is given in the macro parameters, no searching routines are generated.

4.5.3 Cache

The caching mechanism in VRR is based on a limited amount of available memory with LRU (the least recently used) deallocation. Limits of the cache size can be set globally in the user settings.

To share the cache between several independent parts of the project, where each part can have a special optimized allocator, it is accessed through the following general interface. Every part of VRR using the global shared cache is called *cache user*. Users are identified with a unique number and a pointer to a deallocation routine. When someone cannot allocate a new block because the cache is full, the least recently used block is found and depending on the stored identification number, an appropriate deallocation routine is called. This is repeated until there is enough free space in the cache to create the new block.

All cached blocks are held in a double-linked list structure. This allows all operations to be done in a constant time. Items in the list are sorted according to the time of last access. Each access to one of the allocated blocks must be followed with a call to `cache_touch` to move it to the head of the list.

A simple self-descriptive example of the cache usage can be found in the file ‘`lib/cache_example.c`’ and a more detailed description in the file ‘`lib/cache.h`’.

4.5.4 Linked lists

A very comfortable interface for working with general linked lists is implemented. In every structure you would like to include in a linked list, define a special `node` attribute. During list operations, the structure is referenced only via this attribute. There are functions for inserting new nodes at various positions, walks through the whole linked list, deletion, etc. See ‘`lib/slists.h`’, ‘`lib/clists.h`’ (a circular modification) and ‘`lib/cclists.h`’ (a counted circular modification) for details. The usage is similar to the AVL-trees (see Section 4.5.2 [AVL-Tree], page 12).

4.5.5 Growing array

The growing array is a dynamically allocated array structure that keeps monitoring the access and when there is a request for an index beyond the actual array size, the array is automatically

resized. The file '`lib/garr.h`' can be used as a function generator. There is a support for passing growing arrays as function arguments comfortably. See '`lib/garr.h`' for details.

4.6 Miscellanea

There is also a simple prime number testing and generating defined in '`lib/prime.h`' which is primarily used by the hash table implementation (see Section 4.5.1 [Hash table], page 12).

5 GEOMLIB

5.1 Overview

GEOMLIB is the geometrical library of the VRR project.

5.1.1 Purpose

Geometrical library (GEOMLIB) consists of the following modules:

- General numerical algorithms such as polynomial solver or linear system solver.
- Simple geometrical computation with points, vectors and affine transformations.
- General geometrical data structures such as R*-Tree.
- Objective programming emulation for C99 standard.
- Hierarchy of geometrical classes, especially various curve types. Computations can be accessed through (virtual) methods.

The library is almost independent part of VRR project and uses only VRRLIB definitions. See Chapter 4 [VRRLIB], page 11 for description of VRRLIB.

5.1.2 Error handling

Many functions in GEOMLIB can fail because of floating point error or a different reason. Each error type is described by its unique error code (a negative integer constant), for example `GEOM_ERR_NUMERIC`. Full list of error codes can be found in the file '`geomlib/err.h`'.

Most of possibly failing routines follow these calling conventions:

- If a function returns integer type, zero or positive result means success, while negative result means one of defined error codes.
- If a function returns floating point type, it succeedes, when result is finite number. It fails otherwise.
- If a function returning pointer type fails, result is `NULL`.

In some cases the error code is stored in global variable `geom_errno`. To simplify long error testing and code debugging, there are many basic macros defined in the file '`geomlib/base.h`'. For historical reasons, some older parts of GEOMLIB do not return predefined integer error codes and only return undefined negative values.

5.1.3 Floating-point arithmetic

The library mainly uses `real` floating-point type defined in VRRLIB, which can be of `float` or `double` precision. Some more ambitious computations (i.e. polynomial solver) are fixed to the `double` precision. NaN (not a number) or infinities are usually considered as invalid values and produce the `GEOM_ERR_NUMERIC` error.

5.1.4 Functions input and output

If nothing else is said in function description, all parameters must contain valid data (floating-point types must contain finite numbers, pointers must not be `NULL` and structures must agree with their definitions).

Successfully finished functions always return valid or explicitly described results. Pointers to resulting structures should not overlap with other parameters. The result of failed function is undefined.

5.1.5 Header files

The main GEOMLIB header is '`geomlib/geomlib.h`'. By including this file, all structures, functions and macros can be used. It is also possible to include only a smaller subset of definitions by including the appropriate header file. There are no inclusion dependencies, because necessary headers are used recursively. Most of identifiers in GEOMLIB starts with `geom_` or `GEOM_` prefix. Some shorter internal aliases may be enabled by defining macro `GEOM_SOURCE` before the first header inclusion.

5.1.6 Self-testing code

GEOMLIB contains a script, which should help to find errors in the library implementation by applying some automated random tests. These testing routines are located in the file '`geomlib/geomlibtest.c`' and are compiled together with GEOMLIB. There are many tests of object-oriented programming emulation and correctness of numerical algorithms. Floating point results are tested to a small epsilon constant.

5.2 Numerical algorithms

5.2.1 Polynomials in power form

GEOMLIB contains a general polynomial solver which is used in most of curves computations (see Section 5.6.1 [Rational Bezier curves], page 27).

The *power form* of the general polynomial $P(t)$ is

$$P(t) = \sum_{i=0}^{n} a_i t^i,$$

where:

- $n \geq 0$ is called the *degree* of the polynomial.
- a_i are called the *coefficients* of the polynomial.
- $n = 0$ or $a_n \neq 0$.

Each nonzero polynomial of degree n has at most n real roots ($P(t_j) = 0$). The following function finds all these roots in the increasing order:

```
int geom_polynomial_solve(uns degree, double *coef,
                          uns flags, double *result);
```

To achieve a good numerical stability, all computations work in the double precision. Possible options to the algorithm can be passed in `flags` parameter as an OR combination of the following bits:

`GEOM_SOLVE_LEFT_ONLY`
: Returns only the minimal root if exists.

`GEOM_SOLVE_UNIT_INTERVAL`
: Returns only the roots in the closed interval $[0, 1]$.

`GEOM_SOLVE_MULTIPLICITY`
: Returns each root with multiplicity k as k identical entries in the resulting array.

We have implemented closed form solvers up to degree 4 (including), because they are faster then the general iterative solver designed for higher degrees. Mathematical basis of these algorithms can be found at http://mathworld.wolfram.com/ (Cubic Equation, Quartic Equation).

Roots of the degree 5 or higher polynomials can be found with Jenkins-Traub iterative algorithm. We have translated the C++ implementation from http://www.crbond.com/ to C99 standard. Detailed description of the algorithm is beyound the scope of this documentation, but some brief comments can be found in the source file '`geomlib/polynomial.c`'.

5.2.2 Polynomials in Bernstein form

The $n+1$ *Bernstein basis polynomials* of degree n are defined as

$$B_i^n(t) = \binom{n}{i} t^i (1-t)^{n-i}, \quad i = 0, \ldots, n.$$

A linear combination of Bernstein basis polynomials,

$$P(t) = \sum_{i=0}^{n} \alpha_i B_i^n(t),$$

is called a *Bernstein polynomial* or *polynomial in Bernstein form* of degree n. The coefficients α_i are called *Bernstein coefficients* or *Bézier coefficients*. Every polynomial in power form can be written in Bernstein form and vice-versa.

GEOMLIB widely use Bernstein polynomials because the base curve type (Section 5.6.1 [Rational Bezier curves], page 27) contains Bernstein basis polynomials in its definition. The file '`geomlib/bernstein.h`' defines routines similar to the routines in '`geomlib/polynomial.h`' to solve the polynomials in Bernstein form and conversion routines between power and Bernstein form. There are also another useful operators such as multiplication, addition or derivation of the polynomials.

Implementation of the Bernstein polynomial solver is very simple. It converts the polynomial to power form and then executes previously described solver (see Section 5.2.1 [Polynomials in power form], page 16). In some situations (especially in higher degrees) it would be more effective and geometrically stable to solve Bernstein polynomials and Bézier curve problems with specialized algorithms, but it is left for the future development of VRR project.

The most important functions in the header file '`geomlib/bernstein.h`' are:

```
/* Conversion of a given polynomial from Bernstein form to power form. */
int geom_bernstein_to_power(uns degree, double *bernstein, double *power);

/* Conversion of a given polynomial from power form to Bernstein form. */
int geom_power_to_bernstein(uns degree, double *power, double *bernstein);

/* Finds all roots of a given polynomial in Bernstein form.
   Meaning of flags is the same as in geom_polynomial_solve. */
int geom_bernstein_solve(uns degree, double *coef,
                         uns flags, double *result);
```

5.2.3 Matrix routines

Geometrical library implements algorithms for matrix manipulation. The main feature is a linear equations system solver, using PLU factorization. Linear systems are not used very often in the project and appears only in computations with conic sections.

PLU factorization is the factorization of rectangular matrix A to a permutation matrix P, a lower triangular matrix L with ones on diagonal and an upper trapezoidal matrix U such that $A = P \cdot L \cdot U$, U has the same size as A. P and L are square matrices with as many rows as A. The implementation of this algorithm by Gaussian elimination can be found in the file '`geomlib/matrix.c`'.

There are also routines solving the matrix kernel, rank, inversion or multiplication of two matrices.

5.2.4 Points and vectors

Planar points and vectors can be stored in similar structures:

```
struct geom_point {
  real x, y;
};

struct geom_vector {
  real dx, dy;
};
```

It is safe to type-cast between these structures any time. Some simple manipulation routines and constants are defined in '`geomlib/vector.h`'. Any finite values of coordinates are supported in GEOMLIB but in some specific situations, extremely small or large values can lead to numerical problems.

5.2.5 Affine transformations

Every planar affine transformation

$$x^* = a \cdot x + b \cdot y + c,$$
$$y^* = d \cdot x + e \cdot y + f,$$

can be expressed as a square matrix such that

$$\begin{pmatrix} x^* \\ y^* \\ 0 \end{pmatrix} = \begin{pmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 0 \end{pmatrix}.$$

5.2.5.1 Transformation structure

GEOMLIB stores transformation matrices in the structure

```
struct geom_transform {
  uns flags;            /* special flags */
  real coef[2][3];      /* matrix coefficients */
};
```

where

$$\begin{pmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \texttt{coef}[0][0] & \texttt{coef}[0][1] & \texttt{coef}[0][2] \\ \texttt{coef}[1][0] & \texttt{coef}[1][1] & \texttt{coef}[1][2] \\ 0 & 0 & 1 \end{pmatrix}.$$

To speed up some operations with special cases of transformations, the `flags` entry may have set the following bits set:

`GEOM_TRANSFORM_IDENTITY`
: Matrix expresses the identity transformation.

`GEOM_TRANSFORM_SIMILAR`
: Matrix expresses a similarity.

These flags can be set up during the structure creation or manually by the user. There is no implemented numerical algorithm to detect similarities.

The list of implemented functions with a brief description can be found in the file '`geomlib/transform.h`'. There are functions to initialize the most useful affine transformations. The following routine is used to merge affine transformations:

```
int geom_transform_merge(struct geom_transform *t1,
                         struct geom_transform *t2,
                         struct geom_transform *t);
```

Let T_1 and T_2 be matrices of some affine transformations. Then we can compute the matrix of the compound transformation by matrix multiplication $T = T_2 \cdot T_1$. The code also sets the `GEOM_TRANSFORM_IDENTITY` and `GEOM_TRANSFORM_SIMILAR` flags in the resulting structure if they can be easily determined.

Another implemented methods are for example evaluations of the inverse matrix and the rank or detection of the fixed point of affine transformation.

5.2.5.2 Two-directional transformation structure

The following structure is used to optimize performance and increase the numerical stability of mixed manipulation with compound transformations and inversions.

```
struct geom_transform2 {
  struct geom_transform primary; /* primary matrix */
  struct geom_transform inverted; /* inverse matrix */
};
```

The header file '`geomlib/transform2.h`' defines operations similar to the operations in the file '`geomlib/transform.h`', that work with this extended structure and update the inversion transformation matrix along with the primary matrix.

5.3 R*-Tree

Source: *The R*-Tree: An efficient and Robust Access Method for Points and Rectangles (1990) – Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, Bernard Seeger.*

This data structure is a popular method to localize rectangular objects in two (or generally more) dimensional space. In VRR, it is used for effective localization of geometrical objects near to a given mouse-click (see Section 7.5.4 [Snap], page 70), or to find all objects in a given rectangular area (view drawing, rectangular selection).

R*-Tree is one of the variants to data structure called R-Tree. Each leaf node represents one object within a given bounding box (rectangle overlapping the entire object). Internal nodes contain information about the smallest common bounding boxes of all their children. This arrangement allows us to walk the tree from root to leaves while ignoring large unimportant plane areas (subtrees). R-Tree variants differ by methods of grouping nodes to subtrees.

R*-Tree tree is based on a heuristic optimization and uses combination of several optimization criteria (described in [Data insertion], page 20) to arrange objects to groups and build a balanced tree over them. The data structure is fully dynamic. Insertions, deletions, updates and queries can be mixed and no periodic global reorganization is required.

5.3.1 Structures

We have used balanced (A, B)-Tree (A=`GEOM_RTREE_MIN`, B=`GEOM_RTREE_MAX`) to store the hierarchy. The user should define one instance of `geom_rtree` for each existing R*-Tree and include one `geom_rtree_obj` in each inserted object structure. Internal nodes are allocated and freed automatically. Structures were designed to minimize space requirements:

```
/* R*-Tree main structure */
struct geom_rtree {
  struct geom_rtree_node *root;
    /* pointer to root node (NULL if tree is empty) */
  clist lquery;
    /* list of dynamic rectangular queries */
};

/* R*-Tree leaf node */
struct geom_rtree_obj {
  struct geom_rectangle bbox;
    /* rectangle enclosing entire geometrical object */
  struct geom_rtree_node *parent;
    /* pointer to parent tree node */
};

/* R*-Tree internal node */
struct geom_rtree_node {
  struct geom_rectangle bbox;
    /* smallest common bounding box of node children */
  byte count;
    /* number of children */
```

```
    byte height;
      /* tree level (0 for internal nodes containing leaves) */
    struct geom_rtree_node *parent;
      /* parent node (NULL in root node) */
    struct geom_rtree_node *child[GEOM_RTREE_MAX + 1];
      /* child-pointers (internal nodes or leaves) */
};
```

5.3.2 Data insertion

In standard (A, B)-Tree, after the node is inserted, overfull nodes (i.e. the number of children exceeds B) on the trace from the newly inserted leaf to the root are split. R*-Tree uses following algorithm to find good splits. Along each axis (X and Y), the entries are first sorted by the lower value, then sorted by the upper value of their bounding boxes. For each sort all distributions of entries into two (A, B)-tree nodes are determined.

For each distribution the goodness values are computed:

area-value area[bbox(first group)] + area[bbox(second group)]

margin-value
margin[bbox(first group)] + margin[bbox(second group)]

overlap-value
area[intersection of bbox(first group) and bbox(second group)]

Depending on these goodness values the final distribution of the entries is determined.

Algorithm Split

(S1) Determine the axis, to which the split is performed.
(S1a) For each axis: Sort the entries by the lower then by the upper value of their rectangles and determine all distributions as described above. Compute S, the sum of all margin-values of the different distributions.
(S1b) Choose the axis with the minimum S as split axis.
(S2) Along the chosen split axis, choose the distribution with the minimum overlap-value. Resolve ties by choosing the distribution with minimum area-value.
(S3) Distribute the entries into two groups.

Because R*-Tree is nondeterministic in allocating the entries onto the nodes, it suffers from its old entries. Data rectangles inserted during the early growth of the structure may have introduced directory rectangles, which are not suitable to guarantee a good heuristic. Reorganization during splits is only local optimization.

To achieve better performance, some nodes are deleted and reinserted during the insertion routine. The Whole algorithm is described below. To insert a new entry, the following routine is called with the leaf level as a parameter. All used sub-algorithms are described below.

Algorithm Insert

(I1) Invoke ChooseSubtree (CS1), with the level as a parameter, to find an appropriate node N, where to place the new entry E.
(I2) If N has less than `GEOM_RTREE_MAX` entries, accommodate E in N.
If N has `GEOM_RTREE_MAX` entries, invoke OverflowTreatment (OT) with the level of N as a parameter (for Reinsertion or split).
(I3) If OverflowTreatment was called and Split was performed, propagate OverflowTreatment upwards if necessary.
If it caused a split of the root, create a new root.
(I4) Adjust all covering rectangles in the insertion path.

Algorithm Choosesubtree

(CS1) Set N to be the root.
(CS2) If N is in desired tree level, return N.
If the child-pointers in N point to leaves [determine the minimum overlap cost], choose the entry in N whose rectangle needs the least overlap enlargement to include the new data rectangle. Resolve ties primary by choosing the entry whose rectangle needs least area enlargement, secondary the entry with the rectangle of smallest area.
If the child-pointers in N do not point to leaves [determine the minimum area cost], choose the entry in N whose rectangle needs least area enlargement to include the new data rectangle. Resolve ties by choosing the entry with the rectangle of smallest area.
(CS3) Set N to be the child-node pointer of the chosen entry and repeat from (CS2).

Algorithm Overflow treatment

(OT) If the level is not the root level and this is the first call of overflow treatment in the given level during the insertion of one data rectangle, then invoke Reinsert (RI1) else invoke Split (S1).

Algorithm Reinsert

(RI1) For all `GEOM_RTREE_MAX+1` entries of a node N, compute the distance between the centers of their rectangles and the center of the bounding rectangle of N.
(RI2) Sort the entries in decreasing order of their distances computed in (RI1).
(RI3) Remove the first `GEOM_RTREE_REINSERT` entries from N and adjust the bounding rectangle of N.
(RI4) In the sort, defined in (RI2), starting with the maximum distance, invoke Insert (I1) to reinsert the entries.

5.3.3 Data deletion

Deletion of a given entry uses the same routines as the previously described insertion. If the removed leaf causes, that the number of parent child-pointers would decrease below `GEOM_RTREE_MIN`, the parent node is recursively removed and all remaining child-pointers are reinserted to the same tree level.

5.3.4 Data updates

The modification of already inserted items is implemented by simple calls of data insertion and deletion. It would be possible to improve the performance, especially for relatively small changes.

5.3.5 Rectangular queries

R*-Tree is an especially good heuristic to find objects, that fall into a given rectangular (or point) area. The algorithm is very simple. It starts in the root node and recursively descents to lower levels while cutting off whole subtrees with improper bounding boxes.

The following macros in '`geomlib/rtree.h`' implement rectangular queries for a given R*-Tree:

```
GEOM_RTREE_RECT_INTERSECT_QUERY_BEGIN(unique_prefix, rtree, rect, object) {
  /* executed for each object, whose bounding rectangle
     intersects with query rectangle */
} GEOM_RTREE_RECT_INTERSECT_QUERY_END;

GEOM_RTREE_RECT_ENCLOSE_QUERY_BEGIN(unique_prefix, rtree, rect, object) {
  /* executed for each object, whose bounding rectangle
     is entirely enclosed to query rectangle */
} GEOM_RTREE_RECT_ENCLOSE_QUERY_END;
```

```
GEOM_RTREE_POINT_QUERY_BEGIN(unique_prefix, rtree, point, object) {
  /* executed for each object, whose bounding rectangle
     contains query point */
} GEOM_RTREE_POINT_QUERY_END;
```

5.3.6 Dynamic rectangular queries

Dynamic queries can exist for an arbitrary long period of time and informs the user about changes made in a given rectangular area via callback functions.

New query may be created by `geom_rtree_query_init(query, rtree, rect, show, hide, update)` and destroyed by `geom_rtree_query_destroy(query)`. The following callbacks are supported:

`show` Informs user about every object, that appeared in the query area. This may occur during dynamic query creation, insertion of a new entry to R*-Tree or when previously invisible entry is moved to the watched area.

`hide` Informs user about every object, that disappeared from the query area. This may occur when the query is destroyed, an entry is removed from R*-Tree or previously visible entry is moved outside the query area.

`update` This callback is executed only when previously visible object changed its bounding box, but remains in the query area.

5.3.7 Center pass algorithm

The center pass algorithm enables the user to loop items stored to R*-Tree in order of increasing distance from a given center point. If the heuristic builds an effective planar hierarchy, it offers a sub-linear time complexity to find a limited number of nearest entries. This feature is used in GUI to localize geometrical objects near a given mouse-click (see Section 7.5.4 [Snap], page 70 for usage details).

The implementation is located in the VRR Kernel (see Chapter 6 [Kernel], page 41), but the description thematically belongs to this section of the Programmer's Manual.

At first, we briefly describe the algorithm, how to pass objects sorted by the distance of their bounding boxes. Let S be a set of disjoint subtrees with evaluated distance of their root's bounding box to the center point. Initially, the set S contains only one item representing the whole R*-Tree. The algorithm works in a cycle. At each step of the cycle, the first entry is removed from the set S. If it is a leaf node, the incident object is the nearest of all contained in the set S. If the entry is not a leaf node, then we split the entry (subtree) to set of its root's children subtrees and insert them back to the set S. This procedure is repeated until we have passed the required number of nearest objects or the set S is empty.

This algorithm can be easily extended to consider the exact distance of the objects to the center point instead only of bounding boxes. We only need to add a next level to the hierarchy. Every leaf node reached in the main cycle is reinserted back to set S with its exact object distance.

The set S is implemented by the heap data structure. Minimum in the heap can be found and removed in $O(\log n)$ time as well as a new item can be inserted.

Supported objects by the center pass code are all Kernel's geometrical objects. See the file '`kernel/rtree.c`' for implementation details.

5.4 Objective programming

5.4.1 Introduction

GEOMLIB uses principles of object-oriented programming to simplify hierarchy and common attributes of curve types. And because VRR is written in pure C language, the objective environment with hierarchy of classes had to be emulated.

Class is a type with defined virtual methods and data fields. Each class has its unique identification number (ID) and a table with pointers to virtual methods (VMT). Class can have one class as an ancestor. All virtual methods of ancestors are derived to descendant class and can be found at the same index of its VMT. Descendant class can replace derived pointer to virtual method or define new virtual methods. VMT also contains some useful information as class ID, instance size, class name and pointer to ancestor class VMT. Each class defines the format of its instances. The ancestor instance structure is substructure of all its descendants. The class must be directly or indirectly derived from a special class `o`.

Instance of a given class is an allocated structure with corresponding format. Structure contains header with class ID (used to determine pointer to VMT) and space to store instance data. There remain 3 bytes (to align the structure), that may be used by derived classed.

5.4.2 Definition of a new class

Each class `xyz` must define two structures:

`struct geom_xyz`
: Instance structure. Contains ancestor instance structure as the header.

`struct geom_xyz_class`
: Table of virtual methods. Must contain ancestor VMT structure as the header. There is one global variable `geom_xyz_class` of that type.

Typical example of class `xyz` definition, which is descendant of base class `o`:

```
/* definition of instance structure */
struct geom_xyz {
  struct geom_o o; /* ancestor data */
  int var; /* new data, can be used by any xyz descendants */
};

/* used to define new virtual methods in VMT structure */
#define geom_xyz_VMT geom_o_VMT \
  void (*func)(struct geom_xyz *self);

/* used to replace derived or initialize
   new pointers to virtual methods in VMT */
#define geom_xyz_INIT geom_o_INIT \
  .func = &geom_xyz_func,

/* structures definition (should be in .h file) */
GEOM_CLASS_HEAD(xyz, o);

void geom_xyz_func(struct geom_xyz *self)
{
  /* ... virtual method implementation */
}

/* global variables definition (should be in .c file) */
GEOM_CLASS_DEF(xyz);

void main() {
  /* ... */
  /* VMT initialization, unique class ID is generated */
  GEOM_CLASS_INIT(xyz);
  /* instances of xyz class can be created from now */
}
```

5.4.3 Initialization and destruction

Each instance must be initialized before a call to virtual method. During the initialization, class ID is set and the rest of the structure is filled by zeros. Some classes need to initialize

additional data in the cleared structure before most methods can work. Functions, that modify the instance from the cleared stated to the valid one, are sometimes called *constructors*.

The cleanup is done by the virtual *destructor* defined in class `o`. This method should destroy all internal structures such as additionally allocated memory.

Example:

```
/* memory allocation for a new instance */
struct geom_abc a;

/* instance initialization */
geom_instance_init(&a, GEOM_CLASS(abc));

/* now, instance data are filled by zeros */
/* ... */

/* call to virtual destructor */
geom_instance_destroy(&a);
```

5.4.4 Virtual methods

Entries in the instance VMT can be accessed by the macro `GEOM_INSTANCE_VMT(instance, class, entry)`. This macro reads the class ID from instance header, looks to table of initialized classes to retrieve address of the incident VMT and then returns the desired entry. Instance must be derived from the given `class` in the macro parameter and that class must contain the given `entry` in its VMT.

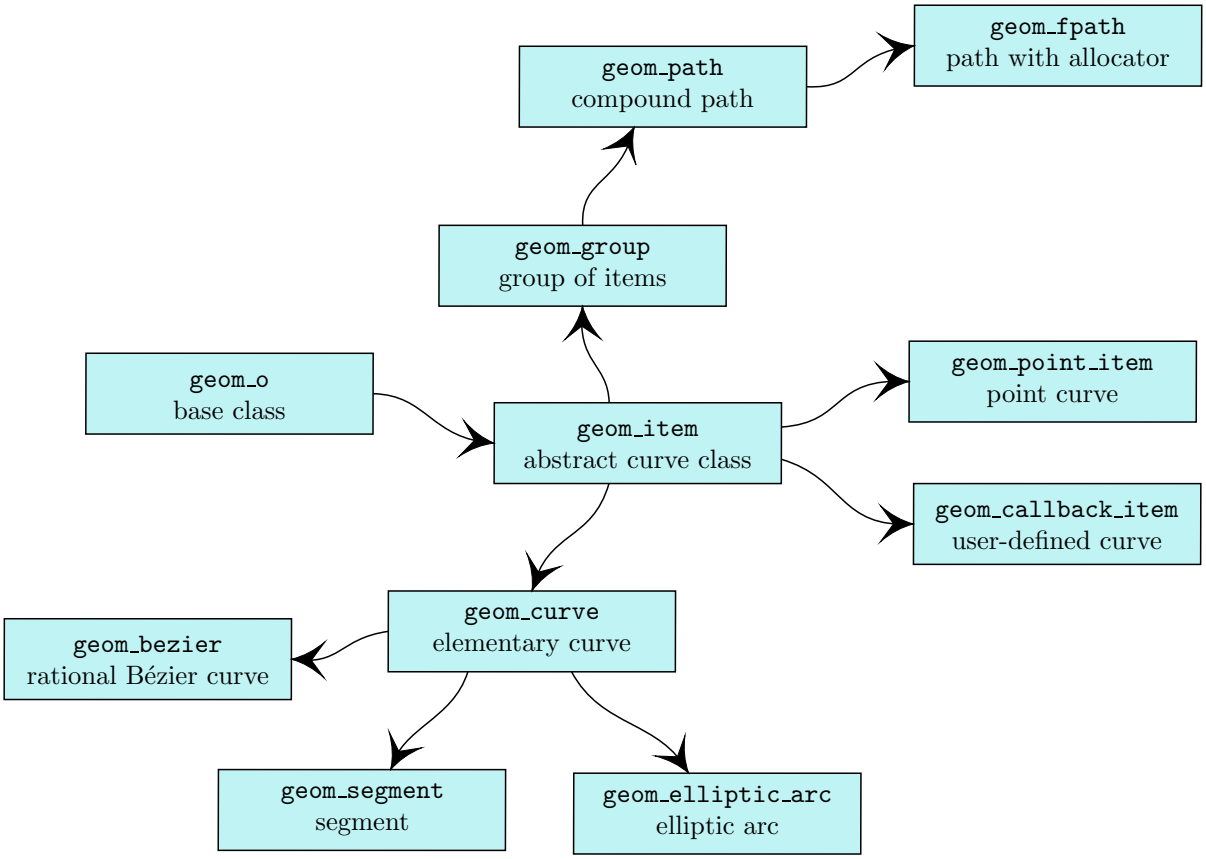
Example:

```
struct geom_abc a;
/* ... */

/* call to virtual method */
GEOM_INSTANCE_VMT(&a, abc, func)(&a);
```

5.4.5 Class hierarchy

- o
  - item
    - group
      - path
        - fpath
    - curve
      - bezier curve
      - segment
      - elliptic_arc
    - point_item
    - callback_item



Picture 1: Class hierarchy.

5.5 Common curves interface

5.5.1 Items and groups

The `item` class extends `o` class by:

- Grouping support. Each item can be inserted to any group or descendant instance (there are some exceptions).
- Debugging routines – assertions and data dumping.
- Abstract interface to planar curves.

The `group` class extends item class by ordered set of child items. There are many routines for manipulation with the set in the file '`geomlib/group.h`'. The implementation uses simple circular linked lists, but it is prepared to be replaced by a balanced tree in the future development.

Single item may be inserted to at most one group at the time, so all existing items and groups generally form a set of trees (forest). Some of that trees are used in VRR Kernel (see Chapter 6 [Kernel], page 41) to describe hierarchy of geometrical objects (GOs) in top level objects (TLOs). For more detailed description of GEOMLIB grouping usage in the kernel, see kernel documentation. Other trees can be used for example as temporary paths.

5.5.2 Curves

Items and groups define a common interface abstract to all supported curves in plane. There are many virtual methods that must or may be redefined in derived classes.

To simplify addition of new curve types and geometrical routines to GEOMLIB there is only a small subset of required methods, that must be implemented in each descendant. If all of them are written correctly, other methods can be automatically emulated with a general code.

This mechanism is achieved by the following rules:

- Each curve in GEOMLIB can be converted to a finite sequence of connected rational Bézier curves. Such sequence is called a *Bézier expansion* of the curve.
- All curve types implement conversions between their base parametrization (`TIME`) and parametrization of their Bézier expansion (`BTIME`).
- All geometrical functions are implemented for rational Bézier curves.
- All geometrical functions are implemented for compound paths that are used to store Bézier expansions.
- There is a general implementation for each geometric function, which can convert the computation to Bézier expansion.

When user calls a geometrical method without a special implementation, a general routine is executed. At first, this method computes a Bézier expansion of the curve and converts all input `TIME` parameters to `BTIME`. After that the geometrical task is solved by the code for paths and Bézier curves. If there are curve parameters in the result, they are finally converted back from `BTIME` to `TIME` parametrization.

Some geometrical problems would be impossible to solve only by Bézier expansion and parametrization conversion routines. One of them would be the splitting of the curve in a given parameter to a pair of curves of the same type. In the current version of the VRR project, there are no such fully implemented functions, but they will probably appear in future releases.

For some curve types, the computation of Bézier expansions is quite slow in the majority of geometrical methods. To increase performance of repeated requests, entire expansion paths are stored in VRR's cache. Full description of the caching mechanism can be found in the file '`geomlib/cache.c`' and Section 4.5.3 [Cache], page 13.

Implementation of the common curves interface is divided in the following files:

'`geomlib/item.?`'
: The `item` class including abstract geometrical methods.

'`geomlib/group.?`'
: The `group` class with ordered sets interface.

'`geomlib/curve.?`'
: General implementation of geometrical methods and the `curve` class definition (see Section 5.6 [Elementary curves], page 27).

'`geomlib/bezier.h`', '`geomlib/bez*.c`'
: Rational Bézier curves implementation.

'`geomlib/path.?`', '`geomlib/fpath.?`', '`geomlib/cache.?`'
: Bézier expansions.

5.5.3 Parametrizations

Each curve can be expressed with infinite number of parametric forms. This is each continuous function from a real closed interval to the plane with the curve as the image. GEOMLIB defines four parametrizations for its curve types. Some of them may be identical.

5.5.3.1 `TIME`

This is the base parametrization for each type and is used in most of computations. Therefore, its definition should allow the developer to implement a fast code. Interval of `TIME` parametrization may be generally $[0, l]$, $l \geq 0$, but for elementary curves it is restricted to unit interval $[0, 1]$. GEOMLIB includes direct conversion routines between `TIME` and each other parametrization.

5.5.3.2 `BTIME`

`BTIME` is defined as `TIME` parametrization of the Bézier expansion. The interval is $[0, n]$, where n is the number of rational Bézier curves in the expansion.

5.5.3.3 ATIME

`ATIME` represents Euclidean arc length parametrization. The interval is $[0, a]$, where a is the Euclidean arc length of the curve. Parameter t corresponds to the point at the arc distance of t from the starting point.

5.5.3.4 RATIME

`RATIME` is called relative Euclidean arc length parametrization. This is exactly the previously defined `ATIME` parametrization linearly scaled to the unit interval $[0, 1]$.

5.5.4 Geometrical methods

Full list of geometrical methods with brief description may be found in the file '`geomlib/item.h`'. If a virtual method is called with prefix `geom_item_`, the correct function address from VMT is used. Redefined virtual methods have prefixes according to the class name, for example `geom_bezier_time_to_atime`. When we know exactly the class at the time of execution, we can use a little faster direct call to the method instead of `geom_item_` interface.

5.6 Elementary curves

Class `curve` is an abstract class, that is basic class for the *elementary curves*. These curves are the only ones, that can be inserted to compound paths. The base parametrization must belong into the unit interval to allow merging into paths (see Section 5.7 [Compound paths], page 37).

5.6.1 Rational Bézier curves

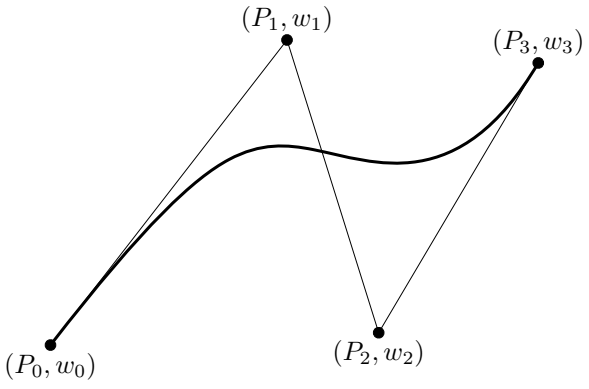
This is the most universal curve in GEOMLIB and any other supported curve can be converted to a finite sequence of rational Bézier curves. We have chosen rational Bézier curves to be the basic curve, because they can exactly represent both conic sections and non-rational Bézier curves. They are also an equivalent to `NURBS` (non-uniform rational Bézier splines) used in professional CAD systems.

5.6.1.1 Definitions

General *rational Bézier curve* of degree n is represented by

$$P(t) = \frac{\sum_{i=0}^{n} w_i B_i^n(t) P_i}{\sum_{i=0}^{n} w_i B_i^n(t)}, \quad t \in [0, 1],$$

where B_i^n are Bernstein basis polynomials, P_i are called *control points* and scalars w_i are called *weights*. This definition describes used `TIME` (see [TIME], page 26) parametrization of rational Bézier curves.



Picture 2: Example of a cubic rational Bézier curve.

When all weights are the same, the equation can be rewritten to

$$P(t) = \sum_{i=0}^{n} B_i^n(t) P_i, \quad t \in [0,1],$$

which is called *non-rational Bézier curve* or simply *Bézier curve*. This is equivalent to two-dimensional polynomial in Bernstein form (Section 5.2.2 [Polynomials in Bernstein form], page 17) with limited interval of the t parameter.

Another definition: A two-dimensional rational Bézier curve (x, y) is the three-dimensional non-rational Bézier curve (x, y, w) projected onto the plane $w = 1$ by the central projection (division by w). This representation of projected three-dimensional space is called *homogeneous coordinates*.

GEOMLIB supports only limited degree of rational Bézier curves: $1 \leq n \leq 3$. It is enough to describe only the most useful curves. All weighs must be positive and to reach a good stability in numerical computations, weights should not have extremely small or large values.

Data structures describing general rational Bézier curve are:

```
/* two-dimensional point with weight */
struct geom_point_w {
  real x, y; /* coordinates */
  real w; /* a positive weight */
};

/* rational Bézier curve */
struct geom_bezier {
  struct geom_curve curve; /* ancestor instance structure */
  struct geom_point_w pt[4]; /* control points and weights */
  struct geom_rectangle bbox; /* cached bounding box */
  real alength; /* cached Euclidean arc length */
};
```

Degree of the curve and flags are stored in the reserved bytes of `geom_o` header to minimize space requirement. Possible flags are:

`GEOM_BEZIER_NONRATIONAL`
: All weights are one. Setting up this flag can drastically speed up curve computations.

`GEOM_BEZIER_BBOX_VALID`
: Bounding box is cached.

`GEOM_BEZIER_ALENGTH_VALID`
: Euclidean arc length is cached.

5.6.1.2 Properties of rational Bézier curves

- If all weights are the same, we get a non-rational Bézier curve.
- Entire curve is in the convex hull of its control points (*convex hull property*).
- The curve starts in its first control point and ends in last control point (*endpoint interpolation*).
- The curve is tangent to the control polygon in the endpoints (*tangency condition*).
- Affine (even projective) invariance: transformed curve is equivalent to curve given by transformed control points.
- The number of intersections of the curve with a line is not larger than the number of intersection of the control polygon with the same line (*variation diminishing property*).

The geometrical meaning of some rational Bézier curves is:

non-rational linear curve
: Segment with linear parametrization.

rational linear curve
: Segment with generally nonlinear parametrization

non-rational quadratic curve
: Segment or section of parabola.

rational quadratic curve
: Conic section (segment, circular arc, elliptic arc, section of parabola or section of hyperbola).

non-rational cubic curve
rational curve curve
: No simple meaning.

5.6.1.3 Recursive subdivision

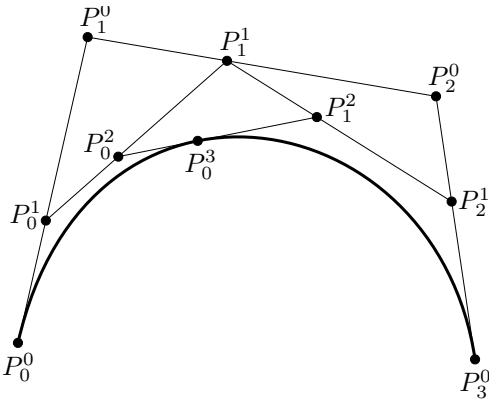
To split non-rational Bézier curves at a given parameter t, we implemented the de Casteljau algorithm. Output is the pair of non-rational Bézier curves, that correspond to subintervals $[0, t]$ and $[t, 1]$ of the original curve. The `TIME` parametrizations of resulting curves are preserved and only scaled from subintervals back to unit interval.

```
/* Splits rational Bézier curve at TIME parameter 0.5. */
int geom_bezier_split_middle(struct geom_bezier *bezier,
    struct geom_bezier *left, struct geom_bezier *right);

/* Splits rational Bézier curve at a given TIME parameter. */
int geom_bezier_split_at(struct geom_bezier *bezier,
    real time, struct geom_bezier *left, struct geom_bezier *right);
```

Let $P_0, \ldots, P_n$ are the control points of non-rational Bézier curve $\sum_{i=0}^{n} B_i^n(s) P_i$. Then we define:

$$P_i^0 = P_i,$$
$$P_i^j = (1-t) \cdot P_i^{j-1} + t \cdot P_{i+1}^{j-1}.$$



Picture 3: An application of the de Casteljau algorithm.

Resulting curve for parameter subinterval $[0, t]$ has control points P_0^i and for $[t, 1]$ control points P_i^{n-i}. Rational Bézier curves can be split in homogeneous coordinates by the same algorithm.

The described algorithm can be used multiple times to split the curve to a sequence of very small curves. By reducing length of the subintervals, the resulting curve parts convert to short

segments with linear parametrizations. This property allows us to solve problems, that would be very hard or impossible to solve directly. One of such problems is computation of Euclidean arc length, which is impossible to solve exactly for rational Bézier curves in the algebraic way.

It is better to implement iterative splits by recursive subdivision in the half, than many splits at smaller parameter values. Recursive subdivision of Bézier curves is known to be numerically stable and we can locally control number of splits, according to the current part's shape.

There are more interfaces to recursive subdivision in GEOMLIB. The following example describes one of them:

```
struct geom_bezier bezier; /* rational Bézier curve to subdivide */
struct geom_bezier_subdivision sub; /* temporarily structure */

/* ... */

/* initialize the subdivision */
geom_bezier_subdivision_init(&sub, &bezier);

/* main subdivision cycle */
while (geom_bezier_subdivision_next(&sub)) {
  /* if bezier curve sub.bezier is "small" enough... */
  if (...) { /* subdivision condition */
    /* ... then use it */
  }
  /* else split it recursively */
  else
    geom_bezier_subdivision_split(&sub);
}

/* clean up temporarily data */
geom_bezier_subdivision_destroy(&sub);
```

Information in the subdivision structure, that may be used in the condition is:

```
struct geom_bezier_subdivision {
  struct geom_bezier *bezier; /* pointer to current subdivided part */
  uns depth; /* recursion depth */
  real left; /* left limit of current interval */
  real right; /* right limit of current interval */
  /* ... internal data entries */
};
```

More detailed description of subdivision interfaces can be found in the file '`geomlib/bezier.h`'.

5.6.1.4 Evaluation of points and derivation vectors

```
int geom_bezier_point_at_time(struct geom_bezier *bezier,
    real time, struct geom_point *result);
```

The previous routine evaluates $P(t)$ from rational Bézier curve definition. Implementation calls de Casteljau algorithm to split the curve and returns common control point of split curves (endpoint interpolation property of rational Bézier curves).

```
int geom_bezier_derivation_at_time(struct geom_bezier *bezier,
    real time, struct geom_vector *result);
```

The previous function applies tangency condition of rational Bézier curves to evaluate requested derivation vector. The curve is split at the parameter and derivation in endpoint of one part scaled by its interval size is returned. Following equations are equivalent and the one with better numerical stability (larger interval) is chosen:

$$dP(t) = \frac{n \cdot w_1^R}{(1-t) \cdot w_0^R}(P_1^R - P_0^R),$$

$$dP(t) = \frac{n \cdot w_{n-1}^L}{t \cdot w_n^L}(P_n^L - P_{n-1}^L).$$

5.6.1.5 Euclidean arc length

```
/* Computes Euclidean arc length of a given rational Bézier curve. */
real geom_bezier_alength(struct geom_bezier *bezier)

/* Conversions between TIME and ATIME parametrizations. */
real geom_bezier_time_to_atime(struct geom_bezier *bezier, real time);
int geom_bezier_times_to_atimes(struct geom_bezier *bezier,
    uns count, real *times, real *atimes);
real geom_bezier_atime_to_time(struct geom_bezier *bezier, real atime);
int geom_bezier_atimes_to_times(struct geom_bezier *bezier,
    uns count, real *atimes, real *times);
```

For rational Bézier curves, it is generally impossible to express Euclidean arc length and `TIME`–`ATIME` ([ATIME], page 26) conversions by an expression. Even for elliptic arcs (a subset of quadratic rational Bézier curves), there appear nontrivial elliptic integrals that are usually solved by iterative methods. Non-rational Bézier curves cause problems from third degree.

To solve these problems, we apply recursive subdivision to approximate the curve by segments and to compute arc length of the resulting polygon. All listed routines use the same approximation to almost straight parts with almost linear parametrizations, so mixed calls should have good behaviour. If we need to convert several parameters at one time, it is much more effective to call only one routine with all the parameters (single subdivision). If the arc length is computed it is stored in Bézier structure cache for later reuse.

More details about the implementation can be found in the file '`geomlib/bezier_param.c`'.

5.6.1.6 Points with a given tangent

This function finds the `TIME` parameters, where the first derivation is parallel to a given vector. In situations with zero of infinite number of solutions, function returns no result.

```
int geom_bezier_direction_times(struct geom_bezier *bezier,
    struct geom_vector *vector, uns flags, double *result);
```

Without loss of generality we can assume, that the direction vector is parallel to x axis. If it is not, we apply a linear transformation to the vector and curve.

The curve is parallel to x axis only when the partial derivation in y is zero. For rational Bézier curve we get

$$Y(t) = \frac{\sum_{i=0}^{n} w_i B_i^n(t) Y_i}{\sum_{i=0}^{n} w_i B_i^n(t)},$$

$$0 = dY(t) = \frac{\left(\sum_{i=0}^{n} w_i dB_i^n(t) Y_i\right) \cdot \left(\sum_{i=0}^{n} w_i B_i^n(t)\right) - \left(\sum_{i=0}^{n} w_i B_i^n(t) Y_i\right) \cdot \left(\sum_{i=0}^{n} w_i dB_i^n(t)\right)}{\left(\sum_{i=0}^{n} w_i B_i^n(t)\right)^2},$$

$$0 = \left(\sum_{i=0}^{n} w_i dB_i^n(t) Y_i\right) \cdot \left(\sum_{i=0}^{n} w_i B_i^n(t)\right) - \left(\sum_{i=0}^{n} w_i B_i^n(t) Y_i\right) \cdot \left(\sum_{i=0}^{n} w_i dB_i^n(t)\right)$$

and for non-rational curves

$$Y(t) = \sum_{i=0}^{n} B_i^n(t) Y_i,$$

$$0 = dY(t) = \sum_{i=0}^{n} dB_i^n(t) Y_i.$$

All we need is derivation, multiplication and subtraction of Bernstein polynomials and a general root solver. These routines are defined in '`geomlib/bernstein.h`'. Some special cases are computed directly to increase the performance. Functions are implemented in '`geomlib/bezderiv.c`'.

5.6.1.7 Bounding box

The problem is similar to finding a bounding box of these points:

- Endpoints of the curve.
- Points, where the derivation vector is parallel to x axis.
- Points, where the derivation vector is parallel to y axis.

```
int geom_bezier_bbox(struct geom_bezier *bezier,
    struct geom_rectangle *result);
```

The previous function calls `geom_bezier_direction_times` twice to retrieve the interior points and returns their common bounding box with the first and last control point. In some singular cases, the algorithm can fail to find the correct bounding box.

5.6.1.8 Curve points in a given distance to a point

This function finds all `TIME` parameters, where the curve is at a given distance to a given point.

```
int geom_bezier_distance_times(struct geom_bezier *bezier,
    struct geom_point *point, real distance, struct garr *result);
```

Without loss of generality, we can assume that the point is located in the origin. Otherwise, we can translate the curve. Let the desired distance is D. For rational Bézier curve we get

$$
\begin{aligned}
D &= \sqrt{\left(\frac{\sum_{i=0}^{n} w_i B_i^n(t) X_i}{\sum_{i=0}^{n} w_i B_i^n(t)}\right)^2 + \left(\frac{\sum_{i=0}^{n} w_i B_i^n(t) Y_i}{\sum_{i=0}^{n} w_i B_i^n(t)}\right)^2}, \\
D^2 &= \left(\frac{\sum_{i=0}^{n} w_i B_i^n(t) X_i}{\sum_{i=0}^{n} w_i B_i^n(t)}\right)^2 + \left(\frac{\sum_{i=0}^{n} w_i B_i^n(t) Y_i}{\sum_{i=0}^{n} w_i B_i^n(t)}\right)^2, \\
0 &= -D^2 \cdot \left(\sum_{i=0}^{n} w_i B_i^n(t)\right)^2 + \left(\sum_{i=0}^{n} w_i B_i^n(t) X_i\right)^2 + \left(\sum_{i=0}^{n} w_i B_i^n(t) Y_i\right)^2.
\end{aligned}
$$

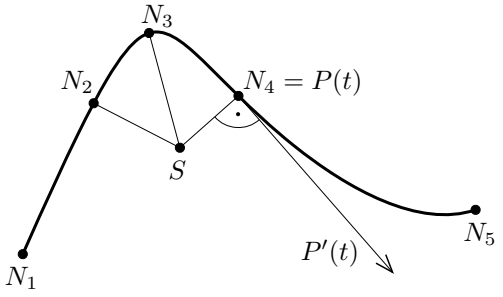
To solve roots of this polynomial, we can use routines from '`geomlib/bernstein.h`'. See '`geomlib/beznear.c`' for implementation details.

5.6.1.9 Curve point nearest to a given point

```
int geom_bezier_nearest_to_point(struct geom_bezier *bezier,
    struct geom_point *point, uns flags, struct geom_nearest *result);
```

Without loss of generality, we can assume, that the point is located in the origin. Otherwise, we can translate the curve. Problem is similar to finding the nearest points of these:

- Endpoints of the curve.
- Points, where the derivation vector and vector leading to the origin are perpendicular.



Picture 4: Candidates for the nearest point.

The second set of points can be described by

$$
\begin{aligned}
0 =& X(t) \cdot dY(t) + Y(t) \cdot dX(t), \\
0 =& \left(\frac{\sum w_i B_i^n(t) X_i}{\sum w_i B_i^n(t)} \right) \cdot \frac{(\sum w_i dB_i^n(t) Y_i) \cdot (\sum w_i B_i^n(t)) - (\sum w_i B_i^n(t) Y_i) \cdot (\sum w_i dB_i^n(t))}{(\sum w_i B_i^n(t))^2} + \\
& \left(\frac{\sum w_i B_i^n(t) Y_i}{\sum w_i B_i^n(t)} \right) \cdot \frac{(\sum w_i dB_i^n(t) X_i) \cdot (\sum w_i B_i^n(t)) - (\sum w_i B_i^n(t) X_i) \cdot (\sum w_i dB_i^n(t))}{(\sum w_i B_i^n(t))^2}, \\
0 =& \left(\sum w_i B_i^n(t) X_i \right) \cdot \left[\left(\sum w_i dB_i^n(t) Y_i \right) \cdot \left(\sum w_i B_i^n(t) \right) - \left(\sum w_i B_i^n(t) Y_i \right) \cdot \left(\sum w_i dB_i^n(t) \right) \right] + \\
& \left(\sum w_i B_i^n(t) Y_i \right) \cdot \left[\left(\sum w_i dB_i^n(t) X_i \right) \cdot \left(\sum w_i B_i^n(t) \right) - \left(\sum w_i B_i^n(t) X_i \right) \cdot \left(\sum w_i dB_i^n(t) \right) \right], \\
0 =& \left(\sum w_i B_i^n(t) X_i \right) \cdot \left(\sum w_i dB_i^n(t) Y_i \right) \cdot \left(\sum w_i B_i^n(t) \right) + \\
& \left(\sum w_i B_i^n(t) Y_i \right) \cdot \left(\sum w_i dB_i^n(t) X_i \right) \cdot \left(\sum w_i B_i^n(t) \right) + \\
& - 2 \cdot \left(\sum w_i B_i^n(t) X_i \right) \cdot \left(\sum w_i B_i^n(t) Y_i \right) \cdot \left(\sum w_i dB_i^n(t) \right).
\end{aligned}
$$

All sums are from zero to n. The resulting polynomial is of degree eight for rational cubic curves, but most cases are much easier. Main parts of the implementation can be found in the internal function `geom_bezier_center_extreme_times` in '`geomlib/beznear.c`'. The algorithm can miss the correct nearest point in some singular cases.

5.6.1.10 Intersections

```
int geom_beziers_intersections(struct geom_bezier *bezier1,
                               struct geom_bezier *bezier2,
                               uns flags, struct garr *result);
```

Computing all intersections of two rational Bézier curves is one of the most difficult tasks in GEOMLIB and it could be improved in many ways in the future. Usually, the problem is solved by recursive subdivision, polynomial solver or algorithms dealing with eigenvalues. Because there is a general polynomial solver implemented in the project, we have chosen the polynomial way.

Implemented algorithm is inspired by the article *D. Machota, J. Demmel: Algorithms for Intersecting Parametric and Algebraic Curves I: Simple Intersection*. In this text the problem of computing intersections of rational curves (in parametric or implicit form) is reduced to eigenvalue problem, but some ideas can be easily adapted for the usage of the polynomial solver.

Let $P(t)$ be the first input curve and $Q(s)$ the second one. The main idea is to convert $Q(s)$ to implicit form $F(x, y) = 0$, substitute first curve $P(s)$ to obtain an univariate polynomial and call polynomial solver to get the intersection parameters on the first curve. Finally, if it is necessary, points and parameters on the second curve are evaluated.

Detailed description of rational Bézier curve implicitization can be found in the cited article. The algorithm results in *symbolic matrix* (each entry is a linear combination of 1, x and y), whose determinant is the curve in implicit form $F(x, y) = 0$.

Before the implicitization is performed, the algorithm tries to reduce curve degree while preserving curve shape to avoid zero determinant of implicit matrix. Such curve is for example non-rational quadratic Bézier curve with the middle control point in the center of the others.

The rest of algorithm is simple, provided we have implemented polynomial solver, curve points evaluation and algorithm for finding nearest points. Details can be found in '`geomlib/bezinter.c`'. In some special cases, the algorithm fails to find the intersections.

5.6.1.11 Degree elevation

```
int geom_bezier_elevate(struct geom_bezier *bezier,
    uns target_degree, struct geom_bezier *result);
```

Degree elevation of a rational Bézier curve means to increase the number of control points while preserving curve shape and parametrization. This can be done exactly in the algebraic way. Used formulas and implementation details can be found in '`geomlib/bezier.c`'.

The previous function is extensively used while exporting images to PS, PDF and SVG (see Chapter 11 [Export], page 100) to convert lower degree curves to cubic Bézier curves.

5.6.2 Segments

Segments in GEOMLIB are stored in the following structure:

```
struct geom_segment {
  struct geom_curve curve; /* ancestor instance structure */
  struct geom_point a, b; /* endpoints */
};
```

The `TIME` parametrization is given by formula:

$$P(t) = (1 - t) \cdot A + t \cdot B.$$

The implementation of segments is very simple and intuitive. Some virtual methods are not redefined for segments, so Bézier expansion to non-rational linear curve is necessary. Details can be found in the files '`geomlib/segment.h`' and '`geomlib/segment.c`'.

5.6.3 Elliptic arcs

5.6.3.1 Definitions

Implicit form of an ellipse with center in the origin and axis parallel to plane axis is

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1,$$

where a and b are lengths of the ellipse semi-axis.

Set of all general ellipses is a subset of all planar conics

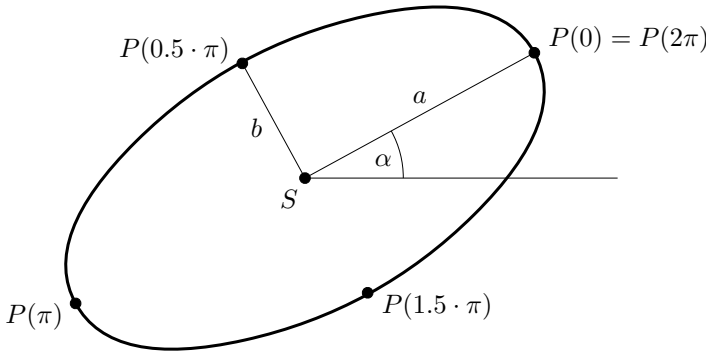
$$Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0.$$

Planar conics include circles, ellipses, parabolas, hyperbolas and some degenerate cases (lines and points). Any of their finite sections can be replaced with a finite set of quadratic rational Bézier curves. GEOMLIB implements structures and direct manipulation with ellipses, circles and their sections. Other conic types are not supported fully.

The parametric form of general ellipse is

$$x = a \cdot \cos t,$$
$$y = b \cdot \sin t,$$

rotated around the origin and moved to the ellipse center point.



Picture 5: Ellipse in parametric form P(t).

To store a section of ellipse (elliptic arc) we use this structure:

```
struct geom_elliptic_arc {
  struct geom_curve curve; /* ancestor instance structure */
  struct geom_point center; /* center point */
  real rotation; /* rotation around the center point (CCW in radians) */
  real a_radius; /* X-axis radius (a_radius >= 0) */
  real b_radius; /* Y-axis radius (b_radius >= 0) */
  real start; /* angle of arc starting point */
  real dif; /* angle between endpoints */
};
```

The `TIME` parametrization with parameter s of the arc has the previously described parametric form, where

$$S = \texttt{center},$$
$$\alpha = \texttt{rotation},$$
$$a = \texttt{a\_radius},$$
$$b = \texttt{b\_radius},$$
$$t = \texttt{start} + \texttt{dif} \cdot s.$$

5.6.3.2 Normalized form

We say that elliptic arc in parametric form is *normalized* if:

- `a_radius` $\geq$ `b_radius`
- `rotation` $\in [0, \texttt{R\_PI}] \simeq [0, \pi]$
- `start` $\in [0, \texttt{R\_2PI}] \simeq [0, 2\pi]$
- `dif` $\in [-\texttt{R\_2PI}, \texttt{R\_2PI}] \simeq [-2\pi, 2\pi]$

Every elliptic arc can be converted to this form. Almost every geometric routine assumes a normalized elliptic arc as input and returns normalized arcs as its output. Assertions of this required condition contain exact `real` constants for interval limits and no rounding errors are allowed. The following function can be used to normalize a given elliptic arc:

```
int geom_elliptic_arc_normalize(struct geom_elliptic_arc *arc);
```

5.6.3.3 Initialization

There are many routines that can be used to initialize entire ellipses in normalized form. These are: the `geom_elliptic_arc_set` and all functions starting with `geom_elliptic_arc_from_`. Full list with brief description can be found in '`geomlib/ellipse.h`'.

An elliptic arc can be created by one of these methods followed by a call to this function:

```
int geom_elliptic_arc_set_endpoints(
    struct geom_elliptic_arc *arc, struct geom_point *start_point,
    struct geom_point *end_point, struct geom_point *mid_point,
    real start_angle, real dif_angle, uns flags);
```

The previous function computes `start` and `dif` entries in `geom_elliptic_arc` structure by selected algorithm in the `flags` parameter. Meaning of remaining parameters depends on this value. Input points should be near the ellipse. Resulting elliptic arc is always normalized.
The following example shows how to create an elliptic arc with given endpoints, rotation and eccentricity that pass through a given midpoint:

```
struct geom_elliptic_arc arc;
struct geom_point *start, *end, *mid;
real rotation, eccentricity;

/* ... compute input parameters */

/* instance creation */
geom_instance_init(&arc, GEOM_CLASS(elliptic_arc));
geom_elliptic_arc_create(&arc);

/* arc initialization */
if (GEOM_ETEST(geom_elliptic_arc_from_3_points_rotation_eccentricity(
        &arc, start, end, mid, rotation, eccentricity)) &&
    GEOM_ETEST(geom_elliptic_arc_set_endpoints(
        &arc, start, end, mid, 0, 0, GEOM_CONIC_ARC_MIDDLE)) {
  /* elliptic arc has been correctly initialized */
}
else {
  /* floating point error */
}

/* instance destruction */
geom_elliptic_arc_destroy(&arc);
```

5.6.3.4 Bézier expansion

Expansion of elliptic arc to a set of quadratic rational Bézier curves is relatively simple. We will describe a formula how to a convert circular arc of angle less than π. General elliptic arc then can be expressed by at most three Bézier curves by applying an affine transformation to converted circular arcs.

Let a circular arc has angle length $\alpha < \pi$. The following Bézier curve is exactly the same curve:

- First and third control points are equal to the arcs endpoints.
- Middle control point is the intersection of tangents in the arcs endpoints.
- Weights of resulting Bézier curve are

$$w_0 = 1,$$
$$w_1 = \cos\frac{\alpha}{2},$$
$$w_2 = 1.$$

```
int geom_elliptic_arc_to_bezier(
    struct geom_elliptic_arc *arc, struct geom_bezier *bezier);
int geom_elliptic_arc_expansion_append
    (struct geom_elliptic_arc *arc, struct geom_fpath *expansion);
```

By the definition of implemented elliptic arcs, we only need to compute expansions of unit circular arcs centered to the origin (at most 3 arcs) and then apply a scale (`a_radius`, `b_radius`), `rotation` and translation (by `center`) to the result. There are some optimizations in the implementation, that can be found in source code.

```
real geom_elliptic_arc_time_to_btime(struct geom_elliptic_arc *arc,
     struct geom_expansion *expansion, real time);
```

To describe conversion from elliptic arc parametrization (t) to Bézier expansion parametrization (s), we assume, that $t \leq \frac{1}{2}$ and that there is only one Bézier curve in the expansion.

Generalization to all possible cases would be simple. The formula used for the computation is following:

$$d = 0.5 \cdot \texttt{dif},$$
$$s = \frac{\sin(d \cdot t)}{\sin(d \cdot t) + \sin(d \cdot (1 - t))}.$$

```
real geom_elliptic_arc_btime_to_time(struct geom_elliptic_arc *arc,
     struct geom_expansion *expansion, real btime);
```

The reversed conversion for $s \leq \frac{1}{2}$ is

$$t = 0.5 - \frac{\arctan \frac{\sin(d \cdot (0.5 - s))}{s - s^2 + \cos(d \cdot (0.5 - s + s^2))}}{2d}.$$

5.6.3.5 Affine transformation

```
int geom_elliptic_arc_transform(struct geom_elliptic_arc *arc,
    struct geom_transform *transform, struct geom_elliptic_arc *result);
```

Affine transformation of elliptic arc in previously defined parametric form is not an easy task. We have not discovered any direct formula for general transformation, so the computation is a little tricky. The curve is first converted to its implicit form where affine transformation can be applied. Finally, it is converted back to the resulting parametric form. These conversions can lead to numerical problems, especially for extremely thin and long ellipses or for almost singular transformations.

The long implementation of this algorithm, dealing with most of singularities, can be found in source '`geomlib/ellipse.c`'.

5.7 Compound paths

Compound path is a connected sequence of elementary curves (rational Bézier curves, segments and elliptic arcs – see Section 5.6 [Elementary curves], page 27). GEOMLIB contains two classes that can hold compound paths of arbitrary length, `path` and `fpath`. The first one only extends the `group` class by geometrical routines. The second path type can be used only in some specific situations and implements extra internal allocator to increase performance of curves inserting. All methods common to elementary curves should also work for compound paths (with differences in parametrization).

5.7.1 Class path

```
struct geom_path {
  struct geom_group group; /* ancestor instance structure */
  real alength; /* cached Euclidean arc length */
};
```

The previous structure is derived from the `group` class, which implements manipulation with ordered sets of items. Compound paths use inherited functions to store the list of connected elementary curves and cannot contain other `item` descendants. Exact continuity of neighbor curves is not checked, but it is assumed, that there are only inconsiderable errors. In the other case, behaviour of some algorithms is undefined.

The `TIME` parametrization of compound path is defined as a composition of all `TIME` parametrization intervals incident to contained elementary curves. Parameter t in the i-th curve (started from zero) corresponds to $i + t$ parameter in the compound path.

The major part of implementation is redefinition of abstract geometrical routines derived from the `group` class. Most of them are very intuitive and do not need detailed description. Usually, a specialized method is called for each elementary curve and results are merged to agree with

entire path. If it is necessary, `TIME` parameters are converted between the path and elementary curves. Full implementation with brief comments can be found in '`geomlib/path.h`' and '`geomlib/path.c`'.

Paths remember their Bézier expansions when they are generated. There is no automated mechanism how to propagate changes to the path from elementary curves or inherited `group` methods. After any change in the path (insertion, deletion or a curve modification), user should manually call `geom_path_after_change` to invalidate cached values.

Left curves in common paths are automatically freed in the path destructor. All inserted curves, that were not allocated using `xmalloc` must be removed manually before calling the destructor. Here follows a simple example of `path` structure usage:

```
struct geom_path path;
struct geom_segment *segment1, *segment2;
struct geom_elliptic_arc *arc;
void *curve;

/* ... curves allocation */

/* path instance creation */
geom_instance_init(&path, GEOM_CLASS(path));
geom_path_create(&path);

/* insertion of some preallocated elementary curves */
geom_group_add_tail(&path, segment1);
geom_group_add_tail(&path, arc);
geom_group_add_tail(&path, segment2);
geom_path_after_change(&path);

/* loop over inserted curves */
GEOM_GROUP_WALK(&path, curve) {
  /*... */
}

/* path instance destruction,
   curves are freed automatically */
geom_path_destroy(&path);
```

5.7.2 Class fpath

```
struct geom_fpath {
  struct geom_path path; /* ancestor instance structure */
  struct geom_fpath_block *block_last; /* set of allocated memory blocks */
  byte *block_ptr; /* start of unused part of last block */
  uns block_free; /* size of last block unused part in bytes */
  uns total_size; /* size of last block in bytes */
};

/* memory block of fpath allocator */
struct geom_fpath_block {
  struct geom_fpath_block *prev; /* linked list pointer */
  byte start; /* start of data buffer */
};
```

The class `fpath` is derived from `path`. The added feature is a simple greedy internal allocator for elementary curves, that is optimized for fast memory allocation. Releasing of allocated memory is impossible until the path is cleared or destroyed.

The life cycle of a `fpath` instance has two phases. In the first phase, only allocations and insertions are allowed. To get to the second phase, `geom_fpath_finish` must be called. After that, the path is fixed and no changes may not be made. Geometric routines can be called in the second phase only. This interface is more strict than it would be necessary for implemented allocator, but it allows some planned future optimizations. A typical situation where `fpath` outperforms `path` in combination with global allocator is computing of Bézier expansions.

The allocator contains a link list of memory blocks reserved by `xmalloc`. New curves are always placed at the first unused byte of the last block. If there is not enough space, a new block is allocated and added to end of the list. Sizes of new blocks increase exponentially.

The function headers and source code can be found in '`geomlib/fpath.h`' and '`geomlib/fpath.c`'.

The following example shows a typical `fpath` usage:

```
struct geom_fpath path;
struct geom_segment *segment1, *segment2;
struct geom_elliptic_arc *arc;
void *curve;

/* fpath instance creation */
geom_instance_init(&path, GEOM_CLASS(fpath));
geom_fpath_create(&path);

/* FIRST PHASE */

/* insertion of some preallocated elementary curves */
segment1 = GEOM_FPATH_APPEND_NEW(&path, segment);
/* ... segment1 initialization */
arc = GEOM_FPATH_APPEND_NEW(&path, elliptic_arc);
/* ... arc initialization */
segment2 = GEOM_FPATH_APPEND_NEW(&path, segment);
/* ... segment2 initialization */
geom_fpath_finish(&path);

/* SECOND PHASE */

/* ... from now, geometric routines are allowed */

/* fpath instance destruction */
geom_fpath_destroy(&path);
```

5.8 Special curve types

The following classes are used in Kernel to allow a similar access to special graphical objects like points and decorators as well as to other curves. These types redefine necessary virtual functions of `item` class to support their Bézier expansion. Because of the specific GEOMLIB structure, all geometrical functions such as bounding boxes computing or arc lengths are automatically supported. A short implementation can can be found in '`geomlib/special.h`' and '`geomlib/special.c`'.

5.8.1 Point item

```
struct geom_point_item {
  struct geom_item item; /* ancestor instance structure */
  struct geom_point point; /* a point in plane */
};
```

The previous simple type offers the interface to single points in plane. Most of geometrical functions could be improved by a specialized implementation working without Bézier expansion.

5.8.2 Callback-expansion item

```
struct geom_callback_item {
  struct geom_item item; /* ancestor instance structure */
  int (*func)(struct geom_callback_item *, struct geom_fpath *);
      /* expansion callback */
};
```

An instance of the `callback_item` class can express any continuous curve, that can be expanded to a sequence of rational Bézier curves. When Bézier expansion is required, callback given by

`func` pointer is executed. This routine should append the sequence of rational Bézier curves to the end of `fpath` (the path is in the first phase as described in [Class fpath], page 38). If the computation raises an error, callback may return a negative error code which is propagated to all geometrical functions results. After a change to the curve is applied, user should invalidate possibly stored expansion by `geom_item_after_change`. The `TIME` parametrization of callback-expansion items equals `BTIME` (parametrization of the expansion).

6 Kernel

6.1 Kernel overview

The purpose of VRR's kernel is to store the main data structures containing geometric objects. It performs no geometric computations directly – that is done by the GEOMLIB routines. Is uses the GEOMLIB heavily. The other modules, on which kernel does not depend, can be informed about data structure changes via the hook mechanism.

It also provides an interface for exception handling: transactions. The VRR source code contains many error checks, especially in the GEOMLIB and the kernel. To minimize the number of return value checks, we have implemented an exception handling mechanism which takes care of memory deallocation and undoing of data structure changes using the transaction logs.

The kernel consists of the following parts, described in the further sections:

- Implementation of an object hierarchy of graphic and non-graphic objects
- Transactions and undo history
- Topological sorting algorithm
- Storing of strings

6.2 Objects

6.2.1 The object hierarchy

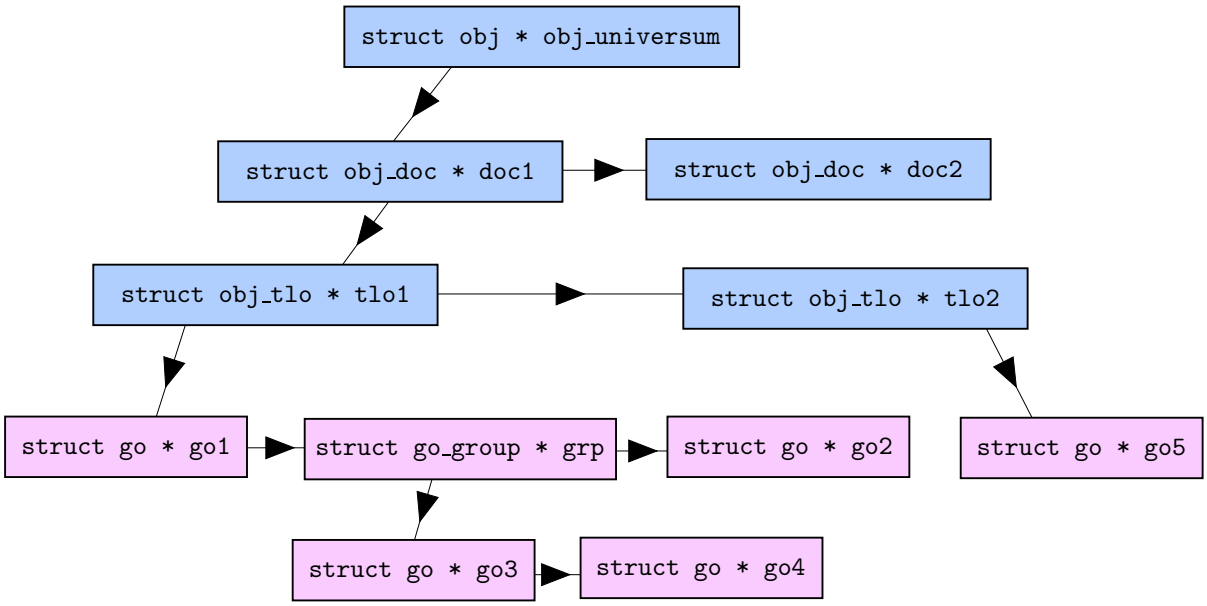
Like the rest of VRR source code, the kernel is written in pure C according to the C99 standard. Nevertheless, the kernel emulates an object hierarchy for basic kernel objects. The hierarchy looks as follows:

- `struct o` – the root of the hierarchy
  - `struct obj` – a non-graphic object
    - `struct obj_doc` – a document
    - `struct obj_tlo` – a page
  - `struct go` – a graphic object
    - `struct go_point` – a point
    - `struct go_segment` – a segment
    - ... and other graphic objects

To enable typecasts, the derived object structure always begins with the contents of the ancestor structure. These are the contents of the root object type whose meaning is then described in more detail:

```
struct o
{
    u8 kind;
    u8 type;
    u8 subtype;
    u8 _dummy;
    uns ref_count;
    struct slist prop;
    uns flags;
};
```

In the following image, you can see an example of the hierarchy of user objects hierarchy. The root of the user object hierarchy is a special object `obj_universe`.



Picture 6: An example of the universe structure.

Object kind

The *object kind* specifies if the object is a non-graphic object (`T_OBJ`) or a graphic one (`T_GO`). The graphic and non-graphic objects have very little in common. Non-graphic objects do not have any graphic nor geometric meaning, their purpose is to represent documents and pages in the object structure, and some internal special entities, too.

Object type

For graphic objects, the *type* specifies the basic object type, like point, segment, intersection, etc. Different object types have different data structures and a completely different geometric behaviour. Objects with the same type have the same hangers.

Non-graphic objects have no subtypes and are distinguished by their type only. The possible values are:

- `OT_UNIVERSE` – a special type used only for the root object of the whole user object hierarchy
- `OT_DOCUMENT` for documents
- `OT_TLO` for pages
- `OT_TEMP` for a special internal object used for storing objects temporarily during a transaction
- `OT_ZOMBIE` for a special internal object used for storing objects which no longer exist (were deleted from the universe) but cannot be deleted yet

Graphic object subtype

The graphic object subtype determines the geometric behaviour of an object more precisely. Objects with the same type have the same set of hangers (the "geometric output"), but they differ in anchors (the "geometric input"). For example, an elliptic arc can be determined by two foci and a point, or by three points on its perimeter plus rotation and eccentricity; that is specified by the subtype.

Reference count

To minimize the efforts needed for correct data deallocation, we have implemented reference counting of objects. If any data structure needs to store a pointer to an object, it should increment its reference count and decrement again to release the pointer. When the reference count of an object becomes zero, the object is destroyed.

Objects with non-zero reference count which are not linked in the universe are stored in the zombie and can be resurrected again (for example, by linking them via undo).

Property list

The objects store various data of various types and various meaning:

- geometric properties, like eccentricity of an ellipse, control point weights of a Bézier curve
- graphic properties, like stroke color, fill color, line width, invisibility
- text or TEX text source code
- any additional user-defined data

All these data can be accessed using an uniform kernel interface. Some of the properties are just regular data structure members, while the others are *virtual* with read and write callbacks which can cause complex recomputations.

6.2.2 Graphic objects

Graphic objects are the main VRR kernel objects. Each one (except for groups) represents a graphic element in the image. The geometric dependency structure is stored in the objects using *anchors and hangers*. Anchors provide "geometric input" for the objects, whereas hangers are "geometric output" devices. The set of hangers is the same for all graphic objects of the same type; the anchor set can differ for subtypes.

A hanger has a list of hangers which hang on it; an anchor contains a pointer to the one hanger it hangs on. There are several types of hangers:

- **position hangers** – hangers associated with a point position. These are the most frequent hangers. Some special hangers are called **mouse-clicks**; they do not belong to any graphic object, they belong to a page and are destroyed automatically. Mouse-clicks are used for hanging "independent" anchors which do not hang on any object's hanger and their point position can be changed arbitrarily.
- **curve hangers** – hangers associated with the object's curve, but no specific point position on it. These are used by parametric points and intersections.

A list of all the supported graphic objects follows.

6.2.2.1 Point

A point is stored in the `go_point` structure. It has one anchor and one hanger, both for the point's position. It has the `GOT_POINT` type and one subtype only – `GOST_POINT`.

6.2.2.2 Segment

A segment is stored in the `go_segment` structure. It has two anchors: the start point and the endpoint, and the corresponding hangers. Additionally, it has a curve hanger for the whole segment. It has the `GOT_SEGMENT` type and `GOST_SEGMENT` subtype.

6.2.2.3 Bézier curve

A Bézier curve has the `GOT_BEZIER` type and can be either `GOST_BEZIER_QUADRATIC` or `GOST_BEZIER_CUBIC`. Both of them have an anchor for the start point, one for the end point. The quadratic Bézier curve has another anchor for the control point, the cubic Bézier curve has two control points. For each point there is a weight property defining the rational weight of the point.

Both the quadratic and cubic Bézier curves have a curve hanger, a start point hanger and an end point hanger.

6.2.2.4 Elliptic arc

An elliptic arc is stored in the `go_elarc` structure. It has the `GOT_ELARC` type and one of the following subtypes:

- `GOST_ELARC_XYR` for arcs defined by the center point and radii
- `GOST_ELARC_FOCI` for arcs defined by the two foci and a point
- `GOST_ELARC_3SMALL` for the smallest elliptic arcs (in area) which contain the three given points
- `GOST_ELARC_3ECC` for arcs defined by three points on the perimeter, rotation and eccentricity
- `GOST_ELARC_XY1ECC` for arcs defined by the center point, a point on the perimeter, rotation and eccentricity

The anchor set differs according to the subtype. Every elliptic arc has a curve hanger, and hangers for start point, end point, and center point. In addition to the subtype, the arc has an important property called "conic". The available values of the property depend on the subtype. The values for all subtypes are:

- **start-entire** – a closed arc
- **start-dif** – an arc defined by the "start" parameter with "dif" defining the arc length

The available values for arcs with at least one point on the perimeter:

- **point-entire** – a closed arc
- **point-dif** – an arc starting from the start point with "dif" defining the arc length

The available values for arcs with three points on the perimeter:

- **ccw** – an arc connecting the three given points counterclockwise
- **cw** – an arc connecting the three given points clockwise
- **smaller** – the smaller one of the two arcs connecting the start point and the end point and having the third point on the perimeter of the whole circle/ellipse
- **bigger** – the larger one (dtto)
- **middle** – the arc connecting the start point and the end point via the center point
- **opposite** – the opposite to "middle"

6.2.2.5 Parametric point

A parametric point is stored in the `go_parametric_point` structure. It has a curve anchor and a "parameter" property which expresses the parameter of the point position on the curve. It has one hanger for the point position. The type is `GOT_PARAMETRIC_POINT` and the subtype is `GOST_PARAMETRIC_POINT`.

The parametrization is linear; the parameter value grows linearly along the curve.

6.2.2.6 Intersection point

An intersection point is stored in the `go_intersection_point` structure. It has two curve hangers and a "parameter" property which expresses the parameter of the point position on the first curve. If the geometry of the curves changes, the intersection is positioned to the intersection closest to the parameter position.

It has one hanger for the point position. The type is `GOT_INTERSECTION_POINT` and the subtype is `GOST_INTERSECTION_POINT`.

6.2.2.7 Text and TEX text

A text or TEX-text object is stored in the `go_text` or `go_tex_text` structure, respectively. Both have a position anchor for the text reference point and no hangers. They have also a transformation matrix which defined the text's transformation up to absolute movement.

The position of the text objects is defined by a reference point (hanger) and several properties. The "align" property determines the reference point position with regard to the text bounding box. The "relative-shift" and "absolute-shift" properties determine additional adjustments of the position.

Each text label has several special points useful for aligning. The left reference point is usually a point on text's baseline near the left edge of the leftmost letter and is taken from the used font. We also define the right reference point which is exactly on the right edge of text's bounding box in the current VRR's version.

First, the label is aligned according to values of "align" and "relative-shift" properties. The result is then translated by "absolute-shift" and transformed with a stored linear transformation (rotation, scale or skew) around the hanger point.

The possible values of "align-x" (the horizontal align) are:

- **refpoints-relative** – The hanger is placed horizontally between the two reference points with the x coordinate of "relative-shift" as the parameter.
- **refpoints-left** – The parameter is replaced with 0.
- **refpoints-center** – The parameter is replaced with 0.5.
- **refpoints-right** – The parameter is replaced with 1.
- **bbox-relative** – The hanger is placed horizontally between the edges of the text's bounding box.
- **bbox-left** – The parameter is replaced with 0.
- **bbox-center** – The parameter is replaced with 0.5.
- **bbox-right** – The parameter is replaced with 1.

The possible values of "align-y" (the vertical align) are:

- **baseline** – The hanger is placed vertically on the text's baseline.
- **bbox-relative** – The hanger is placed vertically between edges of the text's bounding box.
- **bbox-bottom** – The parameter is replaced with 0.
- **bbox-center** – The parameter is replaced with 0.5.
- **bbox-top** – The parameter is replaced with 1.

The ordinary text contains also the font ID and font size. The TEX text contains an array of TEX glyphs obtained from the FONTLIB.

They have the `GOT_TEXT`, `GOT_TEX_TEXT` types, and `GOST_TEXT` and `GOST_TEX_TEXT` subtypes, respectively.

6.2.2.8 Decoration point

A decoration point is stored in the `go_decorator_point` structure. It has the `GOT_DECORATION_POINT` type and `GOST_DECORATOR_POINT` subtype. It is a point with style properties including the rotation, radius and the number of vertices. The number of vertices can be:

- **zero** for a circular shape
- **one** for no shape
- **two** for a segment
- **three up to one hundred** for a polygon with the given number of vertices

- **more** for a circular shape again.

The decoration point is resistant to transformations – it always keeps the same radius and rotation.

6.2.2.9 Arrow

An arrow is stored in the `go_arrow` structure. It has the `GOT_ARROW` type and `GOST_ARROW` subtype. It is a transformation-resistant decoration, too. It should be positioned on a curve (it has a curve hanger and a parameter property) and it adjusts its direction according to the curve's tangent in the snap position. Its appearance can be adjusted by changing the properties.

Each arrow has four important points - the front point (where the hanger is located), two side points and the back point. By setting up the "arrow-alignment" property, you can specify how the arrow's rotation should be computed. We can align the arrow to the derivation in the front point or force the back point to be in the intersection of the curve and the side points' center line. The second possiblity is useful especially for a very rounded curve. The final rotation can be adjusted with "rotation".

The front shape of the arrow is controlled by the "arrow-angle" property (half of the angle between the front point and side points), "arrow-length" (distance between that points),"arrow-front" (shape type) and "front-curvature". The property "back-distance" determines the angle between the back point and side points and the "arrow-back" property determines the back shape of the arrow.

6.2.3 Groups

A group is a special case of a GO containing a list of GOs and a list of selected GOs – the selection of GOs is local in groups.

Every graphic object is always linked inside a group, if not in the universe then in a special group somewhere else. A group is a graphic object, too; but there are also some special groups called *top-level groups* which are not linked in any group. They are contained in pages instead. These groups also store the list of a page's mouse-clicks (see [Mouse clicks], page 43).

6.2.4 Paths

Paths are special cases of groups, too. In addition to groups, a path contains graphic style properties (stroke color, fill color, etc.) that override the style properties of all objects contained. The objects linked in a path must satisfy strict dependency requirements: each object's first anchor must hang on the previous object's last hanger. Moreover, only these objects can be linked into a path: a segment, Bézier curves, and elliptic arcs.

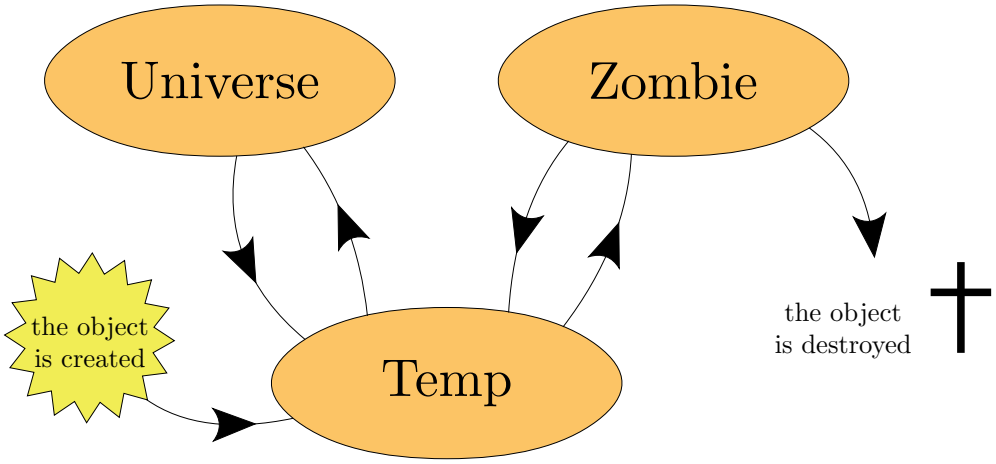
6.2.5 Pages

Every page stores a tree-like hierarchy of groups. It consists of:

- **undo history** – the list of undo history items with a pointer to the current one.
- **hook list** – hooks which were registered for the contents of this page.
- **top-level group**
- **temp** – another top-level group for objects which have been temporarily unlinked from the page (inside on a transaction).
- **tsort list** – a data structure for storing the topologically sorted contents of the page.
- **R-tree** – the GEOMLIB R-tree data structure for the page's contents. These data are freed when not needed.

Thus, every page has its independent undo history. The undo history items for undo actions preformed of non-graphic objects (which are not inside any page) are stored in a special page `tlo_universe`.

6.2.6 Linking and unlinking



Picture 7: The life cycle of an object.

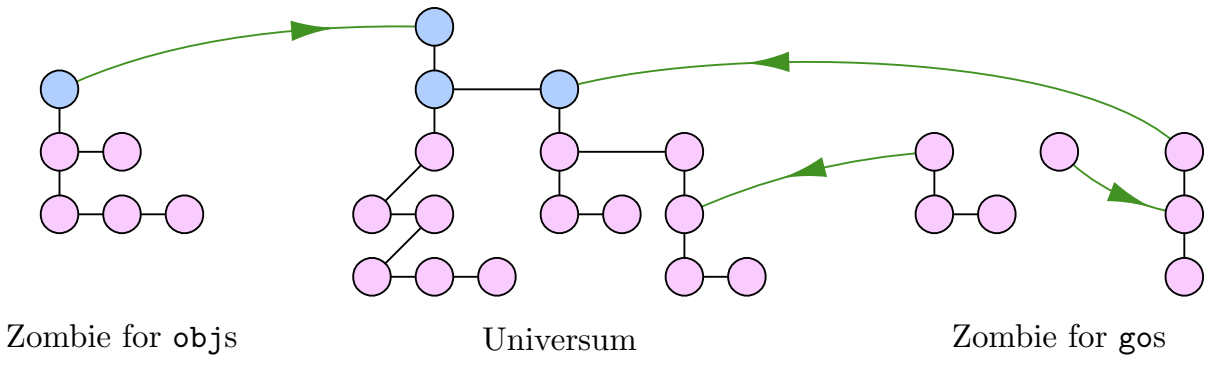
An object can "live" in the following three locations:

- **The universe** – the root of the whole object hierarchy created by the user. (Which itself is invisible for the user.)
- **The temp** – an object which stores objects which were temporarily unlinked during a transaction.
- **The zombie** – a non-graphic object which stores all objects unlinked from the universe with non-zero reference counts.

When an object is created, a transaction (see Section 6.3 [Transactions and topological sorting], page 48) is active. Every transaction uses a *temp*. Transactions working with graphic objects use the `temp` of the current transaction page, transactions working with non-graphic objects use the `obj_temp` object.

The newly created object is initialized to have the temp as its parent (we say that the object is *linked* in the temp). Then, inside the transaction, it can be relinked into an object in the universe. If not, it is relinked into the *zombie* after the end of the transaction. From there, it can be then relinked via the temp into an object in the universe.

The objects in the zombie are deleted automatically when their reference count becomes zero. If an object is relinked, its children are not – their parent pointers remain unchanged. For example, if you unlink a page, it goes to the obj-zombie, but the page's contents stay linked in the page. When you undo the last action, the page is resurrected with its contents as well.



Picture 8: An example of the zombie structure.

In the picture, you can see an example of object structure of zombie. There are two zombie objects: one for graphic objects (`gos`) and one for the non-graphic ones (`obj`). Every object linked in the zombie points to its previous parent from where it was unlinked; as observed in the picture, the previous parent might be in the zombie as well. All the pointing objects from zombie keep a reference of the parent so that the pointer does not get invalid.

The zombie and temp objects cannot be accessed directly; instead, there are these functions that do the linking:

```
obj_link(struct obj* obj, struct obj* father);
obj_unlink(struct obj* obj);
obj_relink(struct obj *obj, struct obj *father, struct obj *old_father);

go_link_after(struct go *go, struct go_group *group, struct go *after);
go_unlink(struct go *go);
go_relink(struct go *go, struct go_group *group, struct go *after);
```

and their various special cases for convenience. They use the temp internally.

6.3 Transactions and topological sorting

The VRR kernel provides an interface for exception handling: transactions. To minimize the number of return value checks, we have implemented an exception handling mechanism which takes care of memory deallocation and undoing of data structure changes using the transaction logs.

All changes to kernel data structures must be done via transactions. The transaction logs are called *undo histories*. In every page, there is one undo history for transactions manipulating the contents of the page. Additionally, for transactions manipulating the non-graphic objects, there is one undo history stored in the page `tlo_universe`.

The transactions can be nested arbitrarily. If the current transaction fails, all the necessary kernel recoveries are performed automatically using the transaction log. Some other actions can be done in the exception handler code of the transaction. The exception handler may cause another transaction fail and thus cause the exception to be propagated up in the transaction stack.

6.3.1 How to use transactions

The transaction interface is similar to those of some other programming languages:

```
TRANS_BEGIN_MAIN( transaction_tlo, undo_item_description, error_buffer )
{
    // do something

    if (something_nasty)
        trans_fail("Something nasty ocurred.");

    // ...
}
TRANS_FAILED
{
    //  Error handler.

    if (too_nasty)
        trans_fail("Something too nasty occured.");
    else
        msg("This happened: %s", error_buffer);
}
TRANS_END
```

The transactions are implemented using the `goto`, `return` and `longjmp` commands. Jumping into and out of the transaction block is forbidden. Inside of a transaction, any other functions can be called without restrictions. The only method to break the current transaction is to call

the `trans_fail` function. When this function is called, the current transaction is broken and the error handler is executed.

```
void trans_fail(const char* format, ...)
```

The `trans_fail` function leaves all nested functions and does a long jump to the beginning of the error handler. It accepts the same arguments as `printf` and the resulting message is stored in the `error_buffer` which is an array of characters. The length of the array must be `TRANS_ERR_SIZE`.

The transactions can be nested – a transaction can be called inside another transaction. When a nested transaction finishes (successfully or unsuccessfuly), the code execution continues after the `TRANS_END` of the transaction. The `trans_fail` function can be called in the error handler of a nested transaction, which causes a fail of the superior transaction. Thus the exception can be propagated.

There are three ways to start a transaction (the rest is the same):

- `TRANS_BEGIN_MAIN` – starts a new transaction with the given TLO. This should be used when you are unsure if a transaction for the desired transaction TLO is already on the transaction stack.
- `TRANS_BEGIN` – starts a new nested transaction with the current transaction tlo. This is useful if you want to prevent possible errors from causing the fail of the whole current transaction.
- `TRANS_BEGIN_ANONYMOUS` – for transactions which create no undo history items; useful when you just want to use the transaction exception mechanism.

6.3.2 Undo histories

As mentioned before, every page has its own undo history; additionally, there is a special page `tlo_universe` for storing undo history items of actions performed on `obj`s. Every undo history item corresponds to one top-level transaction, the nested transactions are included in it. Creating an undo history item for one page does in no way affect the undo histories of other pages; the undo histories are independent.

One undo item has two levels – the outer one for the GUI and the inner one for the kernel. The GUI level contains the name of the item and a list of elementary kernel operations, such as object link, anchor rehang or a property change. When undo is called, a whole GUI undo action is undone.

To manipulate the undo histories, these functions can be used:

```
trans_undo(struct obj_tlo* tlo, char *error_string);
trans_redo(struct obj_tlo* tlo, char *error_string);

trans_clear_undo(struct obj_tlo* tlo); // clear the whole undo history
trans_clear_one_redo(struct obj_tlo* tlo);
trans_clear_all_redo(struct obj_tlo* tlo);

trans_merge_undo(struct obj_tlo *tlo, struct undo_gui *first, string name);
trans_rename_undo(struct obj_tlo *tlo, struct undo_gui *ug, string name);
```

They must not be called when a transaction is active (for obvious reasons). The following functions enable the GUI to navigate through the undo histories:

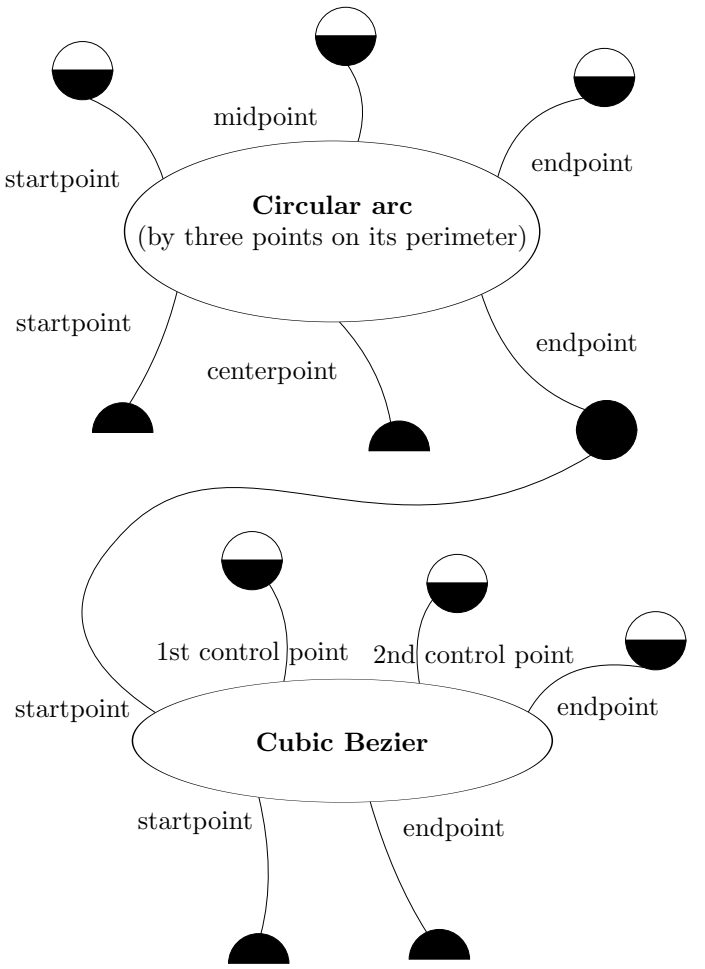
```
trans_get_first_gui(struct obj_tlo* t);
trans_get_next_gui(struct obj_tlo* t, struct undo_gui *ug);
trans_get_last_gui(struct obj_tlo* t);
trans_get_prev_gui(struct obj_tlo* t, struct undo_gui *ug);
```

6.3.3 Geometric dependencies and topological sorting

As mentioned before, every graphic object has several anchors which serve as “geometric input” and hangers which are “geometric output” devices. Every anchor is connected to exactly one

hanger, while a hanger can hold arbitrarily many anchors. This creates a dependency structure of geometric objects within a page (no anchor can hang on a hanger from a different page).

In the following image you can see a dependency diagram with two graphic objects. The half-circles represent anchors and hangers: the black half-circles pointing up are anchors, the ones pointing down are hangers and the white half-circles are mouse-clicks.



Picture 9: An example of a dependency diagram.

When any geometric property of a graphic object is changed or when an anchor of the object is rehung on some other hanger, the object must be recomputed. But all the dependent objects – objects whose anchors hang on the object's hangers – must be recomputed as well. The recomputation must be done in the right order; we use the topological order which assures that every object is dependent on the previous objects only.

We will not describe the topological sorting algorithm, as it is well known; we only describe how the algorithm is used in the VRR kernel.

Every page contains a special data structure for the topological sort: a list of graphic objects `tsort_list` and a bitmask for flags. The flags have the following meaning:

- `OF_TSORT_ACTIVE` means that the `tsort_list` is nonempty
- `OF_TSORT_PRESORT` means that the objects in the list are in topological order

- `OF_TSORT_DIRTY` means that the objects in the list have been modified, but not yet recomputed

The topological sorting algorithm is used for these activities as well:

- **Rehang anchors with acyclicity checks.** If the user tries to rehang an anchor, it must be checked that he resulting dependency graph is acyclic. For those purposes, the owner GO of the anchor is put in the sorted list with all its dependent objects and the anchor must not be rehung on any hanger whose GO is contained in the list.
- **Manage transformation of given GOs.** When performing many transformations with a certain fixed set of GOs, it is quite useful to use the same sorted list without having to rerun the sorting algorithm.

All the other editing actions must be done when the topological sorting is not active, because they could change the dependency structure (by adding new dependent GOs or removing some, for example) and make the sorted list invalid.

6.3.4 Using topological sorting

```
void tsort_start(struct obj_tlo * tlo);
void tsort_is_active(struct obj_tlo *tlo);
void tsort_end(struct obj_tlo *tlo);
```

The topological sorting is started by calling the `tsort_start` function for a page. This function sets up the `OF_TSORT_ACTIVE` flag. Calling it when the sorting is active is forbidden. The `tsort_is_active` function can be used to detect whether the sorting is active.

When the sorted list is no longer needed, the sorting should be deactivated by calling the `tsort_end` function.

```
void tsort_insert(struct go *go, struct obj_tlo *tlo);
void tsort_insert_group(struct go_group *group, struct obj_tlo *tlo);
void tsort_insert_selected(struct go_group *group, struct obj_tlo *tlo);
void tsort_insert_hanger(struct hanger *h, struct obj_tlo *tlo);
```

These functions insert an object(s) together with all the dependent ones into the sorted list. The `tsort_insert_hanger` function does not insert a hanger but the hanger's parent GO.

If the given objects are already in the list, nothing is done. Calling these functions when the sorting is active is forbidden.

```
void tsort_insert_flag(struct go *go, struct obj_tlo *tlo);
void tsort_insert_group_flag(struct go_group *group, struct obj_tlo *tlo);
void tsort_insert_selected_flag(struct go_group *group,
                                struct obj_tlo *tlo);
void tsort_insert_local_selected_flag(struct go_group *group,
                                      struct obj_tlo *tlo);
```

Some operations need a set of GOs in topological order as the input. But in the sorted list, there are the dependent objects as well; in that case it is necessary to mark all explicitly inserted GOs by a flag `GOF_TSORT`. To insert some objects with flags, use these functions instead of the previous ones.

6.4 Hooks

Hook are used to inform some other modules (on which the kernel does not depend) that something important was changed in kernel data structures. Any module can register its callbacks to some parts of the data structures; the kernel will call the callbacks on certain specified occasions.

There are four kinds of hooks: object, GO, transaction and unit. For each hook kind, there is a specific hook structure type which contains pointers to hook callbacks and a `void *` for any additional data. To use hooks, a module should allocate its own hook structures, fill the callback pointers and the additional data accordingly (or set `NULL` to those pointers for which it does not want to register any callback), and then register the hook by calling a kernel function.

When an action happens in the kernel, all hook callbacks registered for the appropriate action are called with a pointer to the registered hook structure as an argument. The hooks can then be removed, even during the call of a hook callback.

After registering, the hook structure is filled with pointers which link it in the kernel hook list; thus it is forbidden to have a hook structure registered more than once at the same time.

Object and unit hooks are global for the whole kernel, whereas GO and transaction hooks are local for pages. Now we give a description for the four hook types.

```
struct obj_hook* obj_hook_add(struct obj_hook *hook);
struct obj_hook* obj_hook_remove(struct obj_hook *hook);
struct obj_hook* obj_hook_remove_by_data(void *data);
struct go_hook* go_hook_add(struct obj_tlo *tlo, struct go_hook *hook);
struct go_hook* go_hook_remove(struct obj_tlo *tlo, struct go_hook *hook);
struct go_hook* go_hook_remove_by_data(struct obj_tlo *tlo, void * data);
```

6.4.1 Object hooks

The object hooks handle the changes of non-graphic kernel objects. They provide these callback calls:

- **Link** – an object was linked into the universe
- **Unlink** – an object was unlinked from the universe
- **Select** – an object was selected
- **Unselect** – an object was unselected
- **Property create** – a new property was created for an object
- **Properties changed** – one or more properties were changed for an object
- **Property delete** – a property was deleted from the object

The Link and Unlink hooks are not called recursively; if the object has a subtree of descendants, no hook is called for them.

6.4.2 GO hooks

- **Link** – a GO was linked into the page
- **Unlink** – a GO was unlinked from the page
- **Relink** – a GO was relinked between groups inside the page
- **Select** – a GO was selected
- **Unselect** – a GO was unselected
- **Property create** – a new property was created for a GO
- **Properties changed** – one or more properties was changed for a GO
- **Property delete** – a property was deleted from the GO
- **Change** – generic information about changes for the Visualization (see Section 7.4 [The Visualisation], page 64).

Like for the object hooks, the GO Link and Unlink hooks are not called recursively.

The Visualization hook

The Visualization needs a special hook `go_changed` which informs it about various changes. The changes can be distunguished by the data passed as a callback argument of type `const struct changed_data_generic*`. It is pointer to one of these structures:

`transformed_data`
: After a transformation, the transformation matrices are passed together with bounding boxes before and after the transformation. The change kind is `CK_TRANSFORMED`.

`altered_data`
: After a bounding box preserving change, which may be for example a change of the color, the data have the `CK_ALTERED` kind.

`changed_data`
: For any other changes the `CK_CHANGED` kind is used.

6.4.3 Transaction hooks

The transaction hooks inform about changes in transaction histories. The transaction histories are local (and independent) for pages, so transaction hooks are local for pages, too.

- **New** – a new item in the undo history was created
- **Delete** – an item of the undo history was deleted
- **Update** – un item item of the undo history changed its name
- **Undo** – one item was undone
- **Redo** – one item was redone
- **Saved** – the "saved" flag of a page (or a document) was set

6.4.4 Unit hooks

The unit hooks inform about changes in the unit list. For description of units, see Section 6.5.2 [Units], page 55.

- **Add** – a new unit was added into a slot
- **Unuse** – a unit was set as unused
- **Change** – the multiplier of a unit was changed
- **Default** – the default unit of a certain quantity was changed

6.5 Properties

The kernel objects store various data of various types and various meaning:

- geometric properties, like eccentricity of an ellipse, control point weights of a Bézier curve
- graphic properties, like stroke color, fill color, line width, invisibility
- text or TEX text source code
- any additional user-defined data

All these data can be accessed using an uniform kernel interface. Some of the properties are just regular data structure members, while the others are *virtual* with read and write callbacks which can cause complex recomputations.

One property is stored in a `struct prop` structure. Its contents are:

- `string key` – a unique identifier of the property in the given object. It contains the displayed name of the property.
- `prop_value value` – a union for storing various property values.
- `u8 type` specifies the data type of the property value stored.
- `u8 subtype` provides a more subtle differentiation of property values within the data type.
- `u8 unit` – an index into unit array See Section 6.5.2 [Units], page 55. It defines the display multiplier of the property value (used only for real number values).
- `u8 flags` – a combination of the following:
  - `PTF_VIRTUAL` means that the property is virtual. See Section 6.5.3 [Virtual properties], page 55.
  - `PTF_READ_ONLY` means that changing value of the property always fails.

- `PTF_SAVE` means that the property is saved when saving the object.
- `PTF_RECYCLABLE` means that the property is recycled by the GUI recycler (see Section 7.6.4 [Property Recycler], page 72).

6.5.1 Property types and subtypes

The `value` of a property is a union which can store an unsigned integer, a real number, a string or a pointer. The *type* of the property determines which of the possibilities is used. The *subtype* provides a more subtle differentiation of property values within the data type – it defines the semantics and allowed values for the property, which is used mainly by the GUI.

Currently, VRR supports these property types and subtypes:

- `PT_UNS`
  - `PTU_BOOLEAN` – a logical value, zero or one.
  - `PTU_FONT` – a font ID. See [Property font], page 45.
  - `PTU_CONIC_TYPE_0P`
  - `PTU_CONIC_TYPE_1P`
  - `PTU_CONIC_TYPE_2P`
  - `PTU_CONIC_TYPE_3P` – values of the "conic" property for elliptic arcs with zero, one, two or three points on the perimeter. See [Property conic], page 44.
  - `PTU_CAP_STYLE` – the line cap style, one of `PSC_BUTT`, `PSC_ROUND`, `PSC_PROJECTING`.
  - `PTU_ALIGNMENT_X`
  - `PTU_ALIGNMENT_Y` – alignment values for a text and TEX-text object. See [Texts], page 44.
  - `PTU_ARROW_FRONT`
  - `PTU_ARROW_BACK`
  - `PTU_ARROW_ALIGN` – arrow appearance values. See Section 6.2.2.9 [Arrow], page 46.
  - `PTU_COLOR` – a RGBA color coded into one 32-bit number.
  - `PTU_UNSPECIFIED` – any other unsigned integer value.
- `PT_REAL`
  - `PTR_COORDINATE` – a coordinate in millimeters.
  - `PTR_ANGLE_PI` – an angle in radians of value within $< 0; \pi >$.
  - `PTR_ANGLE_2PI` – an angle in radians of value within $< 0; 2\pi >$.
  - `PTR_ANGLE_4PI` – an angle in radians of value within $< -2\pi; 2\pi >$.
  - `PTR_NON_NEGATIVE` – any non-negative real number.
  - `PTR_REFERENCE` – a number within $< 0; 1 >$.
  - `PTR_UNSPECIFIED` – any other real number.
- `PT_STRING` is stored in our `string` data structure.
  - `PTS_LARGE_TEXT` – a large text. This is used mainly for source texts of text or TEX-text objects.
  - `PTS_FILE_NAME`
  - `PTS_UNSPECIFIED`
- `PT_POINTER` is used internally for various data which do not fit into the standard property value.
  - `PTP_TRANSFORM` – a transformation matrix.
  - `PTP_TEX_PROCESS` – data for storing TEX output.
  - `PTP_UNSPECIFIED`

An object cannot contain two or more property entries with the same key identifier. If you try to set a property with the same key, type and subtype as those of an existing one, the original property is rewritten. If the key is the same but the type or subtype does not match, the attempt causes a transaction fail.

6.5.2 Units

Property values which are real numbers have an additional feature – units. A unit defines a multiplier used for displaying thu number in the GUI. All property values stored use a canonical internal unit; for example, all coordinates are in millimeters.

The formula for displayed values is: $< native >=< displayed > * < multiplier >$.

The units are stored in an array and the properties contain their indices. When the user changes the display unit, he only modifies the property unit index and the internal property value is remains unchanged.

There are properties of the following four quantities:

- `PQ_LENGTH` for longitudinal properties
- `PQ_ANGLE` for angular properties
- `PQ_REFERENCE` for parameters within $< 0; 1 >$
- `PQ_NONE` for non-measurable properties, such as control point weights

The quantity of a property is determined by the type and subtype and can be found in the following way (for a property `prop`): `prop_subtype2quantity[prop->type][prop->subtype]`.

Default unit

Every quantity has a default unit which is stored in `uns unit_default`. If the unit of a property is `PROP_UNIT_MAX` or a deleted unit, the default unit (for the appropriate quantity) is used instead. The default unit cannot be `PROP_UNIT_MAX`.

Deleting units

Checking whether a certain unit is used in some objects would be too slow. Therefore, a unit cannot be simply deleted. Instead, it is marked as unused and it cannot be assigned to properties any more. If a property that has a deleted unit needs to be displayed, it uses the default unit instead.

The default unit cannot be deleted.

6.5.3 Virtual properties

The properties have a non-trivial overhead (because of the key, type and subtype identification, and other data). Also, the GEOMLIB cannot use the property mechanism, as it is independent of the kernel. Moreover, some properties have a special meaning and need some additional value checks (sometimes depending on other property values as well).

For these reasons, the *virtual properties* were introduced. A virtual property looks the same as a regular one; but some special callbacks handle the reading and writing of its value. The callbacks are called by the property manipulating functions.

The information about one virtual property is stored in the `struct prop_virtual` structure (common for all properties of the same key, type and subtype belonging to the same GO subtype). It contains the following callback pointers:

`prop_value (*get)(struct o* o)`
: Returns the value of the property the object o.

`void (*set)(struct o* o, prop_value new_value)`
: Tries to change the value of the property. This function may fail because of a numeric error or a forbidden value. It can also change some other variables of the object `o`.

`void (*set_low)(struct o* o, prop_value new_value)`
: This function is called by the undo to change the value – it does not trigger any callback calls.

`uns (*get_unit)(struct o* o)`
: Returns the property unit. (The property must have a unit.)

`void (*set_unit)(struct o* o, uns unit)`
: Changes the unit. (The property must have a unit.)

List of virtual properties

Every object has its own property list. In the beginning of the list, there are non-virtual properties, and the last non-virtual property points to the first virtual property. Each GO subtype has its own list of virtual properties. The last virtual property is always the "name" property.

6.6 Clipboard

To enable Copy & Paste operations, the VRR kernel implements a clipboard. The clipboard is a regular page `tlo_clipboard` which is linked in the zombie and for which a reference is kept. The GUI even enables the user to edit the clipboard contents. When some objects are copied into the clipboard, they are selected; and when the clipboard contents are to be pasted into another page, only the selected ones are pasted.

The most important clipboard functions are `clipboard_duplicate_go` and `clipboard_copy_selected_go`.

clipboard_copy_selected_go

```
void clipboard_copy_selected_go(struct go_group *source,
              struct go_group *target, struct go *after,
              uns reversed)
```

This function copies all selected GOs from the `source` group into the `target` group. The created GOs are linked after the `after` GO. If `reversed` is zero then created objects are linked in same order as they are in `source`. Otherwise, they are linked in the reversed order.

The `clipboard_copy_selected_go` function must be called in a transaction on the `target`'s page and the topological sort must be passive is both pages. The source and target pages must be different.

```
struct go* clipboard_duplicate_go(struct go *original)
```

This function receives a graphics object `original` and creates and returns its duplicate.

```
void clipboard_paste(struct go_group *target, struct go *after,
                     uns reversed)
```

This function pastes the contents of the clipboard into the *target* group. It just calls `clipboard_copy_selected_go`.

```
uns clipboard_copy(struct go_group *source, const char* name_trans,
                   char* error_string, uns reversed)
```

This function copies the selected GOs in the *source* page into the clipboard. This function only removes all the previous contents of the clipboard and calls `clipboard_copy_selected_go`. It must be called in a transaction on the *source* page.

```
void clipboard_cut(struct go_group *source, uns reversed)
```

This function moves the selected GOs from the *source* page into the clipboard. Because of the locality of undo histories, the GOs cannot be moved between pages directly; instead, the function copies the selected content of the *source* page into the clipboard and the original GOs are removed.

6.7 Strings

The string mdule is used for storing strings. It takes care of deallocations using the reference counts. It also prevents from storing the same string several times.

A string is handled using a variable of type `string`. It is a pointer to the following `string_entry` structure:

`uns length`
: The length of the string, not including the terminating '\0' character.

`uns ref_count`
: The number of references.

`struct string_entry *dead_next`
: A node of the dead list. See below.

`char text[1]`
: A null-terminated array of characters which is stored in this string.

All the strings (string structure pointers) are stored in a hash table. The key into this hash table is the array of characters. The hash table contains pointers to all used strings to avoid memory leaks.

Most referenced objects are freed when the reference count decreases to zero. This is also true for strings if there is no transaction running. Strings which lose the last reference in a transaction are freed at the end of the transaction. So if a string is used only during one transaction, referencing the string is not necessary.

7 GUI

7.1 GUI Overview

The aim of VRR's GUI is to provide a simple and easy-to-use graphic interface without lots of buttons or complicated windows. We have also tried to minimize the number of pop-up modal windows; most messages are output into a status bar instead of opening a message window.

As the whole program is written in the C language, we decided to create the GUI using the GTK. However, like the rest of VRR source code, the GUI prefers our own data types and data structures defined in VRRLIB to those of GObject and GLib (for effectivity and uniformity reasons). See Chapter 4 [VRRLIB], page 11 for description of VRRLIB.

The GUI contains several windows (mostly independent on each other) and several high-level mechanisms: the Command Structure and the GO Factory. The Command Structure defines the structure of all menus and toolbars, generates and maintains menus and toolbars and for each command it controls and evaluates the conditions under which it can be activated. The GO Factory is a mechanism similar to a finite automaton which creates new graphic objects. (They are described in more detail in the following sections.)

The source code of GUI consists mainly of callbacks for GTK signals and our kernel hooks (see Section 6.4 [Hooks], page 51). The purpose of kernel hooks is to inform (via callbacks) other parts of VRR about data structure changes and other actions. Almost no copy of kernel data structures is kept in GUI; editing actions are performed on kernel data directly. Windows displaying the same data are informed about data updates by kernel hooks as well.

7.2 Windows

VRR has several windows, mostly non-modal and independent on one another. When VRR starts, the Main Window is opened and if the user closes it, he terminates the program.

Most VRR windows have a "common ancestor" – the window structures begin with several data members common for all. These include window type identification, a Scheme proxy and a status bar with a status bar ID. Most window data structures are defined in the '`gui/main.h`' file.

```
#define WIN_O_MAGIC u8 type; SCM proxy; \
                    GtkWidget * statusbar; gint context_id

/* The ancestor */
struct window { WIN_O_MAGIC; };

/* A window */
struct wnd_univ_browser
{
    WIN_O_MAGIC;
    ...
};
```

The ancestor structure is used, for example, in the context (see Section 7.3.1 [The Context], page 61), for status bar messages and error messages. Usually, when an error occurs during an action not connected to any fixed window, the message is output into the status bar of the current context window.

In the following subsections, we describe the most important VRR windows. However, we will not describe all windows in detail, as there is nothing very interesting on the windows themselves. All the interesting features – the Command Structure, property editors, GO Factory – are described in their own sections.

VRR also provides some macros for the "Open file" and "Save file" dialogs. They store the last used directories (one for each) and update them automatically. The save dialog also asks if to overwrite an existing file. They can be found in the '`gui/dialogs.h`' and '`gui/dialogs.c`' files.

```
OPEN_DLG_START(_title, _extra_widget, _filename)
OPEN_DLG_END

SAVE_DLG_START(_title, _extra_widget, _filename)
SAVE_DLG_END
```

and a `SUGGEST_FILENAME(_o, _ext, _output)` macro which generates the suggested save filename for the given object, extension and the last used save directory. They are used like this:

```
OPEN_DLG_START( "Open file ...", NULL, NULL )
{
    // now do something with each filename selected
    // the current filename is 'filenames[i]'
}
OPEN_DLG_END

SUGGEST_FILENAME( document, "pdf", sugname );

SAVE_DLG_START( "Export PDF file ...", table, sugname )
{
    // use the one filename 'filename'
}
SAVE_DLG_END
```

7.2.1 The View

Files: '`gui/main.h`', '`gui/view.c`', '`gui/moving.c`'

The View is the most important editor window. It contains the interface of the GO Factory (see Section 7.5 [The GO Factory], page 65), a drawing area of Visualisation (see Section 7.4 [The Visualisation], page 64) (displaying the contents of a group), a toolbar and a pop-up menu with almost all available commands.

7.2.2 The Universe Browser

Files: '`gui/main.h`', '`gui/univbrowser.c`'

The Universe Browser shows the tree-like structure of all existing kernel objects in the universe. It does not keep any copy of the data; it uses a GtkTreeModel object which provides the interface between kernel structures and the GtkTreeView widget. This is described in Section 7.8.1 [The GtkTreeModel Interface for Internal Structures], page 74.

7.2.3 The Property Editor

Files: '`gui/properties.h`', '`gui/properties.c`'

The Property Editor is a window with dynamically generated contents. It displays the property editor widgets (described in Section 7.6 [Property Editor Widgets], page 70) for all properties of a certain object or for the common properties of all selected objects.

There are two different Property Editors: the context one and the non-context one. The former displays the properties of all selected objects, if there are any, or the properties of the last used context object (see Section 7.3.1 [The Context], page 61). If there is a selection, then only the editors for properties common for all selected objects are shown (with the values of the first selected object). Any change is done to all selected objects at once. The latter displays the properties of one fixed object.

The Property Window has many hook handlers for property changes, adding and removing of properties, context changes etc. After any change of property values, the widgets are updated; after any change of the property list the window's property list is rebuilt.

If the object is deleted, the non-context Property Window is closed. The context window displays the "no object to display" message when there is no context object nor any selection.

Almost all the intelligence of Property Editor is contained in the property editor widgets, see Section 7.6 [Property Editor Widgets], page 70 for more details.

7.2.4 The Text Editor

Files: '`gui/main.h`', '`gui/fonts.c`'

The Text Editor is a property editor containing several property editor widgets (see Section 7.6 [Property Editor Widgets], page 70) specific for a text or TEX-text object. It can also be used for editing a large text property of any object, in which case the specialized property widgets are hidden.

The text in the text editing area can be loaded from or saved to a file in the local character encoding (we convert it between the local charset and the UTF-8 encoding used in GTK using `iconv`). VRR also enables the user to edit the text in an external editor – it creates a child process and waits for it to terminate, which makes VRR look frozen. This also causes problems with editors which fork their process and terminate the original one, like `gvim`.

For an ordinary text object, the Text Editor displays the list of all installed fonts as obtained from FontConfig.

7.2.5 The Global Settings

Files: '`gui/main.h`', '`gui/dialogs.c`'

This window contains some property editor widgets (see Section 7.6 [Property Editor Widgets], page 70) for properties of the universe. These settings are loaded during startup and saved during exit automatically (by the kernel). The meaning of the settings is explained in the User's Manual.

7.2.6 The Undo History Window

Files: '`gui/main.h`', '`gui/undohistory.c`'

The Undo History window shows the undo history items of a tlo or the universe (the "global undo history"). Similarly to the Universe Browser window, it does not keep any copy of the data; it uses a GtkTreeModel object which provides the interface between kernel structures and the GtkTreeView widget. This is described in Section 7.8.1 [The GtkTreeModel Interface for Internal Structures], page 74.

7.2.7 The Unit Manager

Files: '`gui/main.h`', '`gui/properties.c`', '`gui/units.c`'

The Unit Manager displays the lists of all units (per quantities) and lets the user edit them. The unit lists are stored in the same GtkTreeStore objects which are used in unit lists of property editors, which makes all changes appear in the lists at once. Thus, in this case a copy of kernel structures is maintained (using the kernel unit hooks). See Section 7.6.2 [Unit Lists], page 72.

7.2.8 The Plugin Manager

Files: '`gui/main.h`', '`gui/plugins.c`'

The Plugin Manager displays the list of all loaded plugins and the registered plugin functions as obtained from kernel. It enables the user to load plugins and unload unloadable plugins. Here again, the lists are stored in GtkTreeStore objects and maintained using kernel plugin hooks.

The window does not display GUI plugin functions (see Section 10.4 [GUI Plugin Interface], page 97) which are registered to the Command Structure.

7.3 The Command Structure

Files: 'gui/cmdmgr.h', 'gui/cmdmgr.c'

The Command Structure defines the structure of all menus and toolbars, generates and maintains menus and toolbars and for each command it controls and evaluates the conditions under which it can be activated. Each menu or toolbar instance is generated according to one common template and a specified location which defines what commands should the instance contain. For example, the "File/Save" command is located in the View menu and Universe Browser menu, not in the Main Window menu; but all these are generated from the same template.

The command structure template has a stamp and each instance has a stamp, too. The stamp is incremented after every command structure change (not including changes of command states). When a menu or toolbar instance needs to be refreshed, the stamps are compared and in case that they differ, the instance is rebuilt. Otherwise, all the items inside are updated – enabled/disabled, (un)checked – only.

7.3.1 The Context

The context is a collection of several things in the GUI which are currently significant: the current window, the last used object, the group containing the last used objects, the parent document and tlo of the group. It also stores the current selection bitmasks and selected GO counts for meta selection and for selection inside the context group (counts of selected GOs per each GO type).

The context affects mainly the behaviour of the menus and toolbars – its contents define which commands can or can not be activated. But it is used for other purposes as well; for example, many user messages are written into the status bar of the context window.

The context can be changed **only** using the following function:

```
void change_context( struct window * window, struct go_group * group,
                     struct o * o );
```

These three arguments determine all the other contents of the context. After a context change, some internal GUI callbacks are called, like those of the Context Property Window. Any of the context object pointers can be `NULL` as well. A pointer may become `NULL`, for example, when the original object is unlinked or the window is closed. The selection bitmasks and counts are updated after each selection change or context group change.

7.3.2 Command Definitions

The command template is stored in the file 'gui/commands.c'. The command structure is a tree-like hierarchical structure where each node has a list of subcommands (commands with nonempty lists are called *command categories*) and a pointer to the next command in the same category. The command definition structure is this:

```
struct cmd
{
    char * title;
    char * description;
    uns type;
    uns flags;
    char * icon_path;
    GdkPixbuf * icon_pixbuf;
    int accel;
    GdkModifierType modifier;
    union
    {
        struct
        {
            uns request_mask;
            int request_cnt;
```

```
            uns request_type;
            void (*modify_state_f)(struct cmd *, GtkWidget * widget);
            void (*action_f)(void *);
        } func;
        struct
        {
            struct of_state * of_state;
        } op;
        struct
        {
            uns flags;
            struct cmd * first;
            struct cmd * selected;
        } ctg;
    } spec;
    struct cmd * parent;
    struct cmd * next;
};
```

The `type` can be one of `CT_FUNC`, `CT_FACTORY_OP`, `CT_CATEGORY`, `CT_SEPARATOR` and defines which part of the union `spec` is used.

Function commands

The function commands have type equal to `CT_FUNC`. They are the most common menu or toolbar items. They contain an action function `action_f` to be called, a context request defining when the function can be called (and the command enabled) and a `modify_state_f` function to modify the command state additionally. The command state (enabled, visible, checked, etc.) is stored in the `flags` bitmask together with the command location and other flags.

The command request consists of:

- a bitmask `request_mask` of all object types for which the command can be used. The command is enabled if no selected object has a type which is not included in the mask.
- the requested object count – `request_cnt`. Any positive value means that exactly the requested number of objects should be selected. Zero means that any count is suitable, and minus one stands for any nonzero object count.
- the request type `request_type`, one of `RT_META`, `RT_GROUP` and `RT_ANY`. The first one means that the request count applies only to meta objects (not GOs). The second one means that the count applies to the count of selected GOs in the context group and `RT_ANY` means that both selections are evaluated together.

For example, consider the following command:

```
{ "Save", "Save the file", CT_FUNC, CL_UB_MENU | CL_VIEW_MENU | CS_VIS_EN,
   0, 0, GDK_S, GDK_CONTROL_MASK,
   { .func = { VTM_DOCUMENT | VTM_CT_DOC, 1, RT_META,
   NULL, &on_file_save } },
   NULL, &file_save_as_cmd };
```

It requests exactly one document (in the meta selection or in the context), is located in the Universe Browser menu and in the View menu, and is visible and enabled before the context evaluations take effect.

Context Match Evaluation and Command Execution

When a command state needs to be evaluated, the command request is compared with the current context using the following function:

```
int context_matches( struct cmd * cmd );
```

First, the context window, group, tlo and document are tried. If any of them satisfies the request, the conditions are fulfilled and the object will be passed as the `void *` argument of the command's action function `action_f` later when the command is executed. If it is not the case, the selection is tried (meta selection or selection within a group, depending on the request type).

If neither the selection is suitable, then the context object (the last used object) is tried and if matches, it will later be passed as the `void *` argument of the action function. The selection is not passed as any argument; in case that the action function gets a `NULL` as the argument, it uses the selection.

After the evaluation of the `context_matches` function, the `modify_state_f` function is called (with the menu/tool item widget as an additional argument) to alter the command state according to some additional criteria.

To prevent possible errors, the context match is verified one more time, just before the command execution. It might happen that the menu or toolbar item has been activated but the request ceased to be fulfilled before the activation signals were distributed to our callbacks. Or the `modify_state_f` function just had violated the evaluated flags and re-enabled a disabled command. Both cases are detected with an additional context check and if the context does not match, the command execution is cancelled.

Factory Operation Commands

These commands (with type equal to `CT_FACTORY_OP`) switch the GO Factory (see Section 7.5 [The GO Factory], page 65) into the given state stored in `of_state`. They have no context requests as the states of GO Factory commands are evaluated in a special way.

Categories

Category commands (with type equal to `CT_CATEGORY`) are not actually commands, they are just branching nodes and have a list of subcommands (starting with `first`). They create submenus in menus, in toolbars, they are just expanded. In the View toolbar, some categories have their icons and can expand their contents into the right View toolbar.

The additional `flags` store the flags of the category contents.

Separators

The separators have type equal to `CT_SEPARATOR` and represent separator menu items (invisible in toolbars).

7.3.3 Command Editing Actions

The hard-coded initial command template can be modified, commands can be added and/or removed dynamically. That can be done using the function

```
void add_new_command_into( struct cmd * cmd, struct cmd * category );
```

which adds the command `cmd` to the start of the `category` category and

```
void add_new_command_after( struct cmd * cmd, struct cmd * after );
```

which adds the command `cmd` after the command `after`. To remove a command, use

```
void remove_command( struct cmd * cmd, struct cmd * from );
```

7.3.4 Plugin Menu Functions

A special menu command category `plugin_ctg` is reserved for plugin functions. A GUI plugin can register itself into the command structure, which creates a subcategory in the plugin category and assigns an ID to it.

This is done by the function

```
int plugin_menu_register( char * desc );
```

returning the plugin ID. The following functions can be used for adding and removing menu commands, adding commands in a more convenient way or unregistering the plugin. After unregistering, the plugin menu subcategory and all commands added conveniently are destroyed automatically and the plugin ID becomes invalid.

All these functions return zero iff they are successful.

```
int plugin_menu_unregister( int plugin_id );

int plugin_menu_command_register( int plugin_id, struct cmd * cmd );

int plugin_menu_command_register_conv( int plugin_id, char * title,
    char * desc, uns request_mask, int request_cnt, uns request_type,
    void (*action_f)(void *) );
```

7.4 The Visualisation

Files: 'gui/visualisation.h', 'gui/visualisation.c'

The Visualisation is an interface between the VRR kernel and VCL. There is one instance of Visualisation in each view and is responsible for displaying GOs and control objects. It consists of a VCL canvas, several VCL nodes (mainly a lazy expanding area) and some additional data. A LE-area is used for displaying all GOs. Callbacks from the LE-area are translated to questions for kernel and hooks received from kernel are translated to notifications for the LE-area. Zoom and scrolling are implemented using an affinity node before the LE-area.

There are three coordinate systems in the Visualisation. Pixel coordinates of GDK window are clear. The view coordinates (`vcoords`) are centered in the center of the View, with the x-axis parallel to window borders, y-axis flipped (in comparision to pixels) and distance one is exactly one millimeter (this is the coordinate space of the root VCL node). The image coordinates (`coords`) are coordinates in which GOs which are displayed – so they are `vcoords` after application of the chosen zoom and rotation. There are several functions for conversion between coordinate spaces (for example `visualisation_coords_to_vcoords()`).

There are three three groups of functions for manipulation with the Visualisation. The first group is for manipulating with several control objects, the second group is for manipulation with the transformation between `coords` and `vcoords` and the third group contains the aforesaid functions for conversion between coordinate spaces.

Control objecs are usually some crosses or rectangles which are used to implement several gui tools, like the transformation tool. There are the following functions for manipulation with the control objects of the transform tool:

- `visualisation_set_gadget_visible()` for enabling/disabling,
- `visualisation_set_center_gadget()`,
- `visualisation_set_xaxis_gadget()`, and
- `visualisation_set_yaxis_gadget()` for setting their position – the position of the rest is defined by the bounding box of selection),
- `visualisation_set_tf_move()`,
- `visualisation_set_tf_resize()`,
- `visualisation_set_tf_rotate()`,
- `visualisation_set_tf_skew()`,
- `visualisation_unset_tf_move()`,
- `visualisation_unset_tf_second()`, and
- `visualisation_unset_tf_skew()`) for the Santiago's transform tool,
- `visualisation_set_grid()` and
- `visualisation_unset_grid()` for grid – the `t` argument is used to specify the grid arrangement
- `visualisation_gui_rectangle_update()` and
- `visualisation_gui_rectangle_destroy()`) for the rectangular selection rectangle
- `visualisation_set_snap()` and

- `visualisation_unset_snap()` – unused functions for the Fifi
- `visualisation_gui_fifi_update()` and
- `visualisation_gui_fifi_destroy()` – used Fifi functions.

The implementation of anchor/hanger control objects is important and less trivial. It is done using another LE-area, which does not have any affinity node associated, so anchor and hanger coordinates have to be recomputed to `vcoords`. After any transformation change (scrolling) this LE-area must be flushed. The associated functions are `visualisation_set_show_anchors_mode()` and `visualisation_set_show_hangers_mode()`, their `mode` argument can be set to `VIS_SHOW_NONE`, `VIS_SHOW_ALL`, `VIS_SHOW_SELECTED` or `VIS_SHOW_SPECIFIC` based on the request for no A/H, all A/H, A/H of selected GOs or A/H of a specific go (the next argument).

Functions for manipulation with the transformation can be divided to absolute:

- `visualisation_set_orientation()`

relative:

- `visualisation_move()`
- `visualisation_scale()`
- `visualisation_rotate()`
- `visualisation_transform()`

and special:

- `visualisation_absolut_move()`
- `visualisation_center()`

7.5 The GO Factory

Files: '`gui/main.h`', '`gui/createg o.c`', '`gui/view.c`'

The GO Factory is a mechanism similar to a finite automaton which creates new graphic objects. Each state has a desired input, such as a hanger, an anchor, a property value etc. You set the starting state of an operation and then, after getting the input values from the user, proceed to the next states until the object is created. Then the GO Factory returns back to the starting state and the user can create another GO in the same way; or you set a different starting state.

The creation process can be cancelled at any time when needed; for example, when the GUI needs to respond to another user action not related to GO Factory (in that case, it has to cancel the GO Factory operation to prevent errors, which is explained in the further description).

Every state can also have a function which is called before proceeding to the next state. This function can create and link GOs, store a GO pointer in the `tmp_go` variable, and can access the input values obtained so far and perform some actions if needed. The GO Factory can create a special hanger for the mouse cursor on which some object anchors can be hung temporarily. It also enables the user to undo some partial actions by the "Step back" command which uses the undo history.

7.5.1 State definitions

The GO Factory state structure is defined like this:

```
struct of_state
{
    uns flags;
    enum of_input_kind input_kind;
    uns input_type;
    uns input_subtype;
    union
    {
```

```
            struct dist_pass_result dpr;
            struct prop prop;
        } value;
        char * description;
        void (* action_func)(struct of_state *);
        void (* cleanup_func)(struct of_state *);
        struct of_state * prev;
        struct of_state * next;
        struct undo_gui * prev_undo_item;
        struct go * tmp_go[OFR_CNT];
};
```

The `input_kind` specifies what input is wanted from the user. Its value is one of `OFIK_NONE`, `OFIK_DPR`, `OFIK_PROP`, `OFIK_SEL`, `OFIK_TRANSFORM`, and `OFIK_TF`. The last three ones are special and used in the selection and transformation tools; `OFIK_NONE` is used only in some special GO Factory states. The `OFIK_DPR` input kind means that the input is the result of a snap distance pass, and `OFIK_PROP` stands for a property value.

The `action_func` function is called before proceeding to the next GO Factory state (after the state input value has been filled). It is called inside a transaction on the current context tlo, so the function can use transaction fails for error reporting. The `of_state` function argument is a pointer to the current state.

The `cleanup_func` is called when moving back and undoing the effects of the state, and after a successful GO creation as well. It does not have to unlink any created objects; that is done by undo automatically. It should only free any additionally allocated memory or release all references that the action function created.

There are some predefined GO Factory states including the most important ones:

```
struct of_state ofs_start;
struct of_state ofs_end;
```

These two states do not take part in creation of any go directly. But the state structure is linked in such a way that the `prev` pointer of all the starting states points to the `ofs_start` state and the `next` pointer of all the last states points to `ofs_end`.

7.5.1.1 Snap result states

If the input kind is `OFIK_DPR`, then the value of `struct dist_pass_result` is expected to be filled. It contains the snap result (obtained by an R^*-Tree distance pass), which can be a hanger, an anchor, a GO with a parameter of the point position on the GO curve, an intersection represented by two GOs and a parameter, or an unsnapped point defined by its coordinates. The state's input type is the desired `dist_pass_result.type` determining which of the possibilities is the current one.

This is the structure definition:

```
struct dist_pass_result
{
    uns type;
    union
    {
        struct { struct go *go; struct geom_nearest geom; } go;
        struct { real dist; struct hanger *hanger; } hanger;
        struct { real dist; struct anchor *anchor; } anchor;
        struct { real dist; struct go *go1, *go2;
                 struct geom_intersection geom; } intersection;
        struct { real dist; struct geom_point point; } point;
    } data;
};
```

During the step-by-step creation, the input hangers are specified. But they may not yet exist. For example, if the "snap to lines" mode is set on, then a hanger can be specified by a curve GO and a parameter defining a point position on the curve. The hanger itself is created no

sooner than it is really needed, which is when an anchor of the created GO needs to be hung on it (during the execution of the state's action function). In the meantime, the GO Factory stores the snap result only; and if the effects of the action function are undone (because of Step back or transaction fail) the hanger is destroyed as well.

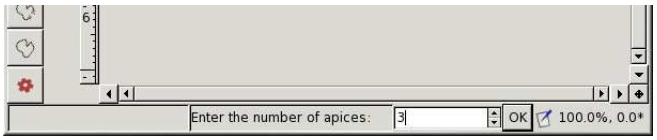
This also means that, apart from the "main" created GO, the GO Factory can create some parametric points, intersection points, or mouse-clicks as well.

7.5.1.2 Property value states

Another possible input kind is `OFIK_PROP` for property values – any property value which can be stored in VRR's kernel. The state's `type` and `subtype` express the property type and subtype, respectively, and the property value is stored in the `struct prop` structure.

To gain the property input from the user, the GO Factory creates a property editor widget (see Section 7.6 [Property Editor Widgets], page 70) in the View status bar and lets the user enter a value. The property widget does some basic value checks according to the property type and subtype; any additional checks should be done in action functions of the states. A state can refuse the obtained value using transaction fail.

So far, this feature is rarely used for getting numeric input from the user; usually, all the numbers are set to some default values and can be modified in the Property Editor when the GO is created. This makes the creation process a bit faster; on the other hand, the user has to switch between the View and the Property Editor to get the desired result.

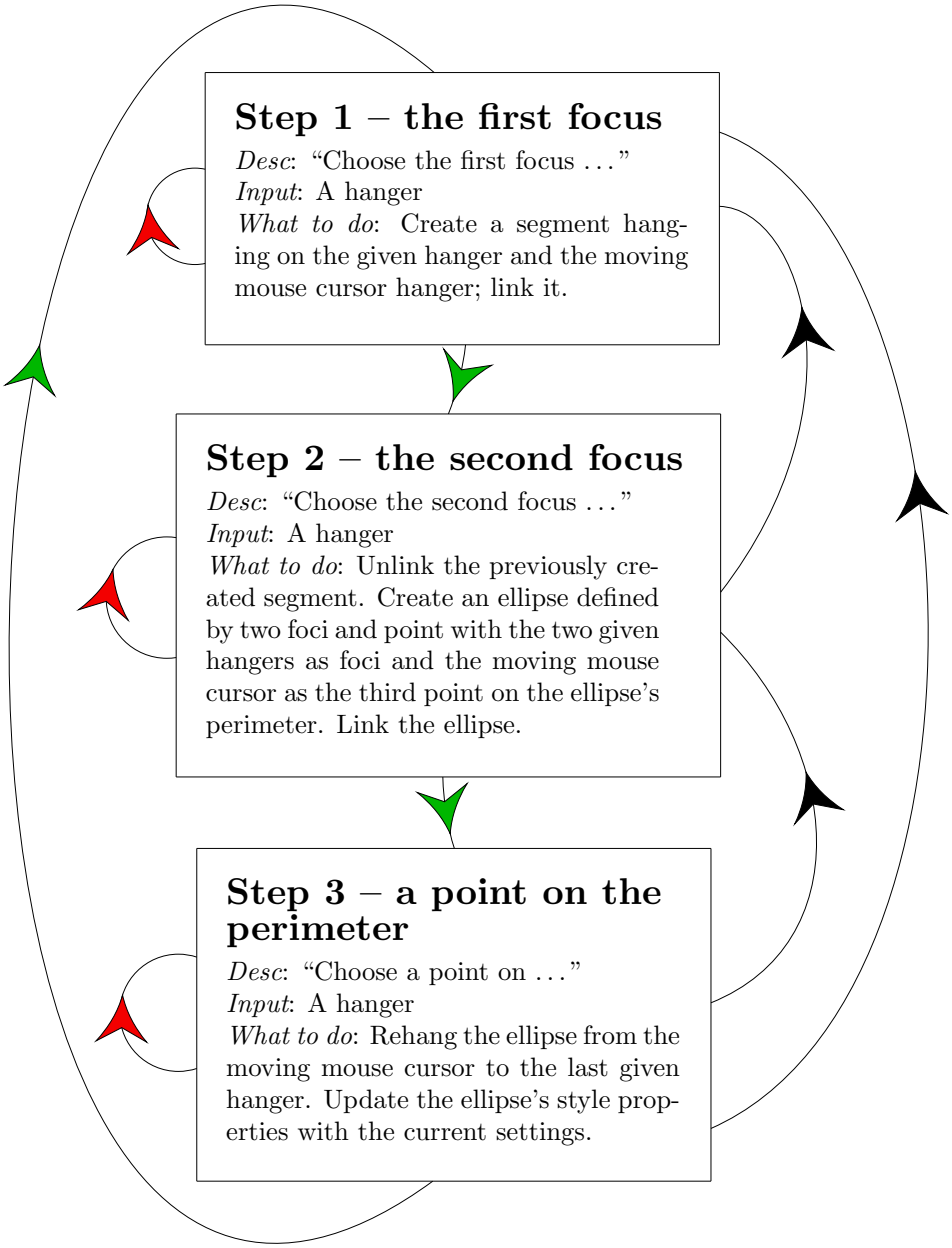


Picture 10: The View status bar showing a property value editor.

7.5.2 Transitions between states

In the following image you can see an example of transitions between GO Factory states. The starting state above is set by a user action (clicking the appropriate toolbar icon). Then the editor waits for input of the current state (in this case, a hanger), and when the input is ready, it performs some actions using the input obtained and in case of success it moves to the next state. If an error occurs, which is usually a transaction fail, the GO Factory refuses the input, the current state remains unchanged, the editor outputs an error message and waits for another input from the user. After a success in the last state, the GO Factory returns to the starting state again. Additionally, in any state except for the starting one the user can move back or cancel the whole creation at once.

The green arrows show state transitions in case of success, the red ones are for errors and the black ones show the possible steps back.



Picture 11: Transition diagram for states creating an ellipse (by two foci and a point).

The creation process is also called an *operation*. The basic operation actions are these:

```
void factory_op_start( struct go_group * group,
                       struct of_state * state, char * desc );

void factory_op_break( void );

void factory_op_step( void );

void factory_op_step_back( void );
```

The current state is accessible as `factory.state`. Here the editor can find out what input to get from the user (according to the type, subtype etc). When the input is ready, it fills the `factory.state->value` state value and calls the `factory_op_step` function. Then the GO Factory does the following:

- Checks if the filled input is really the desired one.
- Starts a new transaction in which it may process the input additionally (e.g. obtain a hanger by creating a parametric point and getting its hanger, or creating a mouse-click if

the point was not snapped to any hanger) and executes the `factory.state->action_func`, if there is any.

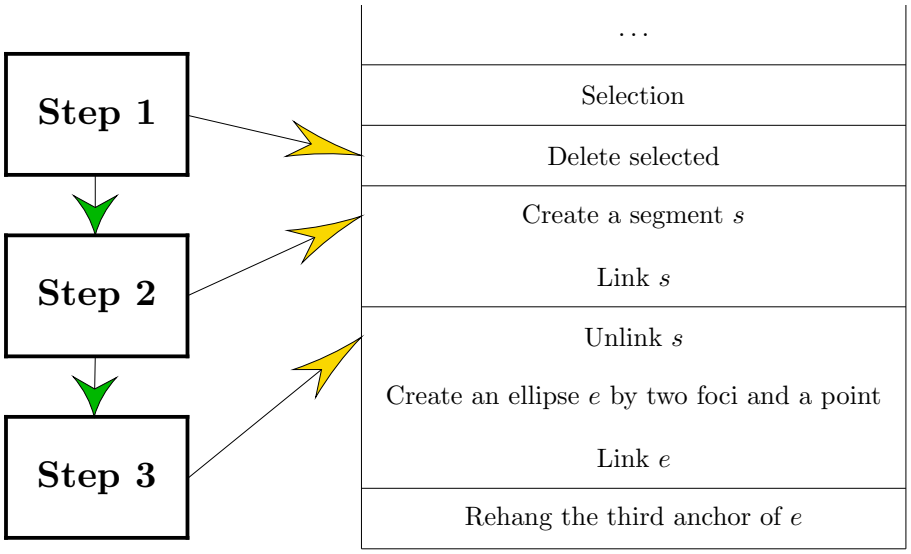
- If the transaction fails, the GO Factory does the cleanup (using the `factory.state->cleanup_func`, too), stays in the same state and waits for some other input.
- If the transaction succeeds, it proceeds to the next state and waits for more input; or, if this was the last state, returns to the starting state again.

The `factory_op_step_back` function undoes the effect of the previous state (including the cleanup) and returns the GO Factory to the previous state.

The `factory_op_break` function undoes all actions from the starting state and moves to the starting state again. This function should be called before any other editing action beyond the control of GO Factory, especially those that create undo history items. The reason is described in the following subsection.

7.5.3 Usage of Undo Items

To keep track of the actions performed in each GO Factory state, each state contains a pointer to the last undo history item which was done before the state was set as the current one. In case of cleanup, the undo history items are undone up to the one that the particular state points to.



Picture 12: States with pointers to previous undo history items.

This is the reason why the GO Factory needs to have several undo history items enabled and why extremely low undo history limits cause strange behaviour – the GO Factory cannot keep track of actions performed and if you cancel the whole creation, it undoes only the existing undo items, which may not be all items used. Also, when the GO Factory is creating an object, no other editing actions are permitted; especially the actions that create their own undo history items, or undo and redo actions. The reason for that is obvious now.

It is also possible that some GO Factory states have no action functions or generate no undo history items. In that case several states may point to the same undo history item, which does not cause any problems.

When an object creation ends successfully, all new undo items are merged into one.

7.5.4 Snap

The GO Factory has four snap modes: snap to hangers, grid, lines and intersections, and a flag which determines whether to create geometric dependencies or just modify the click position (irrelevant for snap to grid). The snap modes are independent on one another and can be combined or switched on and off arbitrarily (even during a GO Factory operation).

When several snap modes are switched on, the closest suitable snap position is chosen. If, for example, snap to grid and to lines are switched on, then a point which is either on a line **or** a grid point is chosen. Or, if there is no such near object within the snap tolerance, the clicked position is kept unchanged and a mouse-click is created. The grid has the lowest priority of all (if there are several objects with equal distance).

The snap tolerance is stored in each View window and recomputed according to the current zoom. The top-level View window sets its tolerance into the global `factory.snap_tolerance`. The bitmask of snap modes is stored in `factory.snap_set`.

The snap functions are:

```
void snap_point( struct geom_point * pos, struct dist_pass_result * dpr,
                 struct geom_transform2 * grid, struct obj_tlo * tlo,
                 real maxdist );

void snap_to_go( struct geom_point * pos, struct dist_pass_result * dpr,
                 struct obj_tlo * tlo, real maxdist );

void snap_to_anchor( struct geom_point * pos, struct dist_pass_result * dpr,
                     struct obj_tlo * tlo, real maxdist );
```

The `snap_point` function snaps the `pos` point according to the current snap settings and updates the value of `pos`. If the `dpr` pointer is not `NULL`, it copies the dist pass result into it. The distance pass algorithm is described in [Center pass algorithm], page 22.

The `grid` is a transformation matrix determining the grid points. The nearest grid point is computed in this way:

- Transform the point using the inverted grid matrix
- Round the transformed coordinates to integers
- Transform the point back using the original grid matrix

The `snap_to_go` function seeks a nearby GO regardless to the current snap settings. This is used for selection purposes, for example. `snap_to_anchor` seeks a nearby anchor – again, regardless to the current snap settings.

7.6 Property Editor Widgets

Files: '`gui/properties.h`', '`gui/properties.c`', '`gui/units.c`'

In many VRR windows, a property value needs to be displayed, edited, and updated in reaction to kernel property hooks. To do this, the windows use the property editor widgets. All property editor widgets are generated using the same code, which assures that all editor widgets for the same property type and subtype look the same and behave in the same way.

There is a data structure containing everything needed for a property editor widget:

```
struct prop_item
{
    struct o * o;
    string key;
    uns type, subtype;
    GtkWidget * value_edit;
    GtkWidget * unit_edit;
    uns flags;
    struct window * pw;
};
```

The `o` pointer represents the object to which the property belongs. `key` is the property unique identifier, `type` and `subtype` store the current property type and subtype (which might change). The two GtkWidget objects, `value_edit` and `unit_edit`, are the editor widgets of the property value and of the property unit. The unit editor does not have to be present; in that case the pointer is `NULL`. The `flags` bitmask is used only in the Property Window for property selection. `pw` points to a parent window, which is either `NULL` or the parent Property Window.

To create a property editor widget, you need to allocate a `prop_item` structure, fill it with the `o`, `key`, `type` and `subtype` values and call these functions:

```
void prop_value_edit_create( struct prop_item * pi );
void prop_unit_edit_create( struct prop_item * pi );
void prop_sync( struct prop_item * pi );
```

(where `prop_sync` updates the editor value with the current property value. This function should be called after each change of the property value announced by kernel hooks).

Or do it all in a more convenient way using

```
void prop_item_init( struct prop_item * pi, struct o * o, string key );
```

Then the created widgets can be packed into the window where needed.

7.6.1 Property Structure Definitions

Each property subtype has its own requirements on the editor. These requirements are stored in a data structure in the file '`gui/properties.c`'. The editor for a subtype is described by the following structure:

```
struct pst_data
{
    uns widget_type;
    char * description;
    union
    {
        struct { gdouble lower, upper, step, page; } spin;
        struct { uns max; string * strings; } combo;
        struct { uns dummy; } nothing;
        struct {
            void (*create_edit_func)( struct prop_item * pi );
            void (*update_edit_func)( struct prop_item * pi,
                                      struct prop * prop );
        } func;
    } data;
};
```

The widget type can be one of these:

- `PWT_BUG` – zero. This is an erroneous value. Its purpose is to prevent uninitialized subtype structures after kernel changes.
- `PWT_SPIN_UNS` – a spin button for unsigned integers. The `spin` structure then specifies the spin button settings.
- `PWT_SPIN_REAL` – the same as `PWT_SPIN_UNS` but for reals.
- `PWT_COMBO` – a combo box. The `combo` structure specifies the maximum value and the string names of the possible values from zero up to maximum minus one.
- `PWT_CHECKBOX` – a checkbox for boolean values. No additional settings are needed.
- `PWT_ENTRY` – a text entry for string properties.
- `PWT_FUNC` – a special property widget with its own creating and updating functions. This is used for example for color buttons, filename and large text editors.

The `description` is used when creating a new property and specifying its subtype. Then only some property subtypes are shown (we believe that, for example, the `PTU_CAP_STYLE` subtype for line caps is not very useful for user-defined properties) and those are subtypes with non-`NULL` description. The description is shown in the subtype list.

7.6.2 Unit Lists

File: 'gui/units.c'

The unit editor widgets are combo boxes containing the list of units for the particular property quantity. These lists are stored in GtkListStore objects and maintained using the kernel unit hooks. The same list objects are used in the Unit Manager (see Section 7.2.7 [The Unit Manager], page 60) to make all editing changes appear in all lists at once. If the user adds, deletes, or changes a unit, then the GtkListStore itself emits signals for all widgets that display its contents; we do not have to do anything more.

7.6.3 Hook Handling and Transactions

Every window containing some property editor widgets has to process the kernel property hooks and call the `prop_sync` function to update the editor value. This lowers the memory usage – one hook is set for all property editor widgets in a window. The editor widget itself handles its value changes and modifies the kernel property values accordingly. The editor value for real numbers is multiplied by the unit multiplier.

The value in the editor is always synchronized to show the current kernel values (multiplied by unit multipliers). The kernel values do not have to be equal to values that the user has set; for example, if he tries to change the start point coordinates of a geometrically dependent curve, the kernel values remain unchanged and the editor returns to the kernel values.

Not every editor value change is written to kernel – first, the kernel value is compared with the one in the editor and if they differ a little, the values are considered equal (up to rounding changes) and the kernel value remains unchanged. This, too, prevents cycling such as this:

- the user changes the value
- the value is read from the widget editor and written to kernel
- a kernel hook is called to handle property change
- the widget editor is updated by the kernel value
- the widget editor value is rounded (to a certain number of decimal places), which invokes a GTK signal later as if the user had changed the value
- ...

This does not in fact cause an infinite cycle, because the property value is locked in kernel during a property change hook call and any attempt to change the value again causes an error. So, the problem must be detected anyway; we do it by comparing the values.

The editor widgets usually use the following value-change callback:

```
void prop_value_changed( GtkWidget * w UNUSED, struct prop_item * pi );
```

which extracts the value from the widget according to the property type and subtype, compares the old and new values and changes the kernel value if needed. The widget values are updated by

```
void prop_sync( struct prop_item * pi );
void prop_sync_prop( struct prop_item * pi, struct prop * prop );
```

(the former one finds the kernel property value according to the property key identifier stored in `pi` and calls the latter).

7.6.4 Property Recycler

Files: 'gui/properties.h', 'gui/properties.c'

When the property values are set using the `TRANS_PROP_CHANGE` macro, the values of some properties (those that have the `PTF_RECYCLABLE` flag set) are stored in a special place called the *property store*. The property store is a GO – a top-level group of a tlo linked in the zombie, not in the universe, and a reference is kept for it to prevent its deletion.

To manipulate with property stores, VRR provides the following functions and macros:

```
/* the definition */
PROP_STORE_DEFINE(_id)

PROP_STORE_NEW(_id)

PROP_STORE_DESTROY(_id)

/* the actual object which stores the properties */
PROP_STORE_O(_id)

/* the transaction tlo for recycler property changes */
PROP_STORE_TLO(_id)

PROP_STORE_GET(_id, _name, _type, _union_member_name, _default)

void prop_store_set( PROP_STORE_DEFINE(ps), const char * name,
                     uns type, prop_value pv );
```

It has two property stores: `ps_global` and `ps_recycler`. The recycler stores the properties set by the user in some editor widgets, whereas the purpose of the global store is to store various settings of some dialogs which are not saved anywhere else. The `ps_recycler` has its own additional functions:

```
void gui_prop_recycler_set( string key, uns type,
                            uns subtype, uns unit, prop_value val );

void gui_prop_recycle( struct o * o );
```

The `gui_prop_recycler_set` function sets the given property to the recycler. If it already contains a property of the same key, type and subtype, then the value is changed; if the key is the same but the type or subtype does not match, the old property is deleted and a new one is created.

The `gui_prop_recycle` function goes through all the object's properties and seeks matching properties in the recycler (with the same key, type and subtype). The values of all matching properties are copied into the object. This function is used by the GO Factory when creating new objects.

7.7 Transformation Tools and Mouse Event Processing

Files: '`gui/main.h`', '`gui/moving.c`', '`gui/view.c`'

VRR has several features which react on mouse cursor movement and cause instant recomputations: the transformation tool, the GO Factory moving hanger on which created objects can be hung, Santiago's transformation tool, and the Fifi. The needed computations may be somewhat lengthy and sometimes, when a large complex image with many dependencies has to be recomputed, not all mouse motion events can be processed.

The computations are performed in idle time; if there is not enough time to process all events, the excess events are ignored. However, the mouse button release event is always processed.

7.7.1 Step-by-step Transformations

The transformation tools transform the selected objects continuously, bit by bit. Computing and merging so many transformation matrices could cause accumulation of numeric errors, especially when close to singularities. To avoid such side effects, the kernel functions apply the transformation matrices always to the original untransformed objects and GUI must supply it with such matrices. The transformations are performed using the `tsort_presorted_transform_safe` function.

7.7.2 The Experimental Fifi

The "Fifi" is a secondary cursor whose purpose is to indicate the current snap position. To update the snap position, all the possible snap objects are searched after each mouse motion event – we have done no optimizations for this so far. The computations are very slow when snap to intersections is set on, because all the intersections are computed each time the mouse cursor moves.

We plan to improve Fifi in future releases.

7.8 Special GTK Objects and Widgets Used

7.8.1 The GtkTreeModel Interface for Internal Structures

Files: '`gui/univbrowser.c`', '`gui/undohistory.c`'

To provide an interface between the VRR kernel and the GTK viewing widgets, we implement the GtkTreeModel interface for several internal structures: the main object structure of universe, and the undo history. The implementations are the "VrrKernelStore" and "VrrUndoStore" objects derived from the GObject class and are used in GtkTreeView widgets. They are used in the Universe Browser window (Section 7.2.2 [The Universe Browser], page 59) and the Undo History window (Section 7.2.6 [The Undo History Window], page 60).

These objects implement only those features needed for viewing the structure contents, they do not enable changing them directly. When any kernel data change is announced by a hook, the interface emits the appropriate signals to inform the widgets about the change.

7.8.2 Rulers

Files: '`gui/ruler.h`', '`gui/ruler.c`', '`gui/vertruler.c`', '`gui/horizruler.c`'

VRR uses special rulers created to meet all our requirements. In contrast to GTK rulers, our rulers can work with any resolution. According to the zoom, they choose a suitable decimal place as the root decimal place. This value designates proper numbers to be displayed on the rule. It can work with both small and large numbers.

The internal ruler implementation uses Pango drawing functions to display text and draws lines and marks defining the current cursor position.

The rulers support changing the lower and upper limit, moving the lower limit, changing the zoom and resolution and changing the current cursor position.

7.8.3 Color Selection Dialog

Files: '`gui/selectcolor.h`', '`gui/selectcolor.c`'

We have implemented our own Color Selection dialog. It supports the RGB, HSV, and CMYK color models and renders the resulting colors independently on the VRR image renderer – it is and independent widget which can be reused in another program without major code changes.

8 VCL

8.1 VCL Overview

8.1.1 The purpose of VCL

The VRR Canvas Library (VCL) is an implementation of a canvas widget with support for all features needed in VRR. It provides this services:

- Widget-like manipulation with drawing primitives (lines, curves, ...)
- Automatic redrawing of invalid regions (after expose-events or some changes in the canvas)
- Interface for different drawing back-ends (supports GDK background and Cairo background)

The library is an almost independent part of the project and uses only VRRLIB definitions.

8.1.2 VCL general usage

The library must be initialised by calling the `vcl_init()` function. After that, the programmer can create a new VCL canvas with the `vcl_canvas_new()` function, set its options with `vcl_canvas_setup()` and set the root VCL widget of that canvas with `vcl_canvas_set_root()`. After that, the programmer can take (from `canvas -> gtk -> widget`) a GTK DrawingArea widget containing VCL canvas and use it in a GTK application. Now you can insert, remove, or alter any children VCL widgets (in the root VCL widget) and everything is redrawn automatically. The library is not limited to one canvas, there may be several canvases at the same time.

The programmer should include exactly one of the '`vcl.h`' (for common usage) and '`vcl_internal.h`' (common usage and additional access to hidden internal function) headers.

VCL uses its own object system based on the interfaces and implementations paradigm. Interface oriented functions are dispatched in a way based on the first argument's class. Each class can implement many interfaces. The object system has some introspection abilities – an object can be asked whether it supports a specific interface, and so on. There is kind of interface hierarchy: if a class A implements interface B, then it has to implement an ancestor interface C as well.

8.1.3 Transformations

VCL widgets (called *nodes* here) create a tree-like hierarchy rooted in the canvas. On each level of tree, there is a transformation matrix from the current coordinate system to pixel coordinate system. So each node has its own coordinate system which is used to store its position and other coordinates. Transformation is defined relatively to the parent's coordinate system, so by changing of coordinate system of a non-leaf node all descendants of that node are altered as well.

8.1.4 Interface sightseeing tour

All VCL objects support the `object` interface, but this interface specifies only a destroy method. The most important interface is the `node` interface. All VCL nodes have to implement it. Every leaf node should implement either the `shape` interface or the `mask` interface (that depends on the way it describes its shape).

Every non-leaf (container) node should implement the `container` interface, and either the `composite` interface or the `enclosure` interface (that depends on the supported number of children – `enclosure` has exactly one child, `composite` has any number of children). Composite nodes often support the `placement` interface - it is used to add and remove children to a composite node, but there are composite nodes without support of this interface – they have another way to get children nodes. The node coordinate changing system supports the `transformation` interface. And the last interface – the `painter` interface – encapsulates drawing backends.

8.1.5 Propagation

There are two basic processes which happen in the VCL node tree – redrawing of regions (down-propagating, running from the root to leaves) and the propagation of a change (up-propagating, running from a leaf to the root). The first process is initiated in the canvas (by a GTK expose event) and managed by the current painter. There is a `context` structure which is internally used by painters and which can take complete control of the drawing process. Canvas calls the painter's methods and the painter walks through the tree using tree-examining methods of containers, and draws the appropriate nodes. The painter ignores uninteresting branches of the node tree. Containers are responsible for storing information about their children (they usually store the aggregate bounding box of all children) needed to cooperate with the painter so as to avoid walking through uninteresting branches. These data are updated during the second process – propagation of a change.

8.1.6 VCL Properties

A common leaf node represents just an area and its border. Its visual appearance is defined by properties. There is a special node for setting properties – the non-leaf `property` node. A property set in that node applies to all its descendants until set to another value by another `property` node.

A property is a pair (key, value), where *key* is an integer constant and *value* is a byte sequence. The list of defined property constants and byte sequence interpretations is as follows:

`VCL_PROP_FILL_COLOR`
: u32 – packed RGBA color data, used for area filling

`VCL_PROP_STROKE_COLOR`
: u32 – packed RGBA color data, used for stroke border

`VCL_PROP_STROKE_WIDTH`
: double – the width of a border in local coords

`VCL_PROP_STROKE_CAP_STYLE`
: u8 constant – cap style

`VCL_PROP_STROKE_JOIN_STYLE`
: u8 constant – join style

Values of cap and join styles are these (the meaning of cap style and join style is traditional, so we will not explain it):

- `VCL_CAP_BUTT`
- `VCL_CAP_ROUND`
- `VCL_CAP_PROJECTING`
- `VCL_JOIN_MITER`
- `VCL_JOIN_ROUND`
- `VCL_JOIN_BEVEL`

8.1.7 Alive and dead objects

Common VCL objects are *alive* objects – there are allocated dynamically on the heap and they can be used in all ways. Alive objects are created by functions that have the `_new` suffix. Apart from alive objects, there are *dead* objects. They are allocated on the stack (using functions with the `_init` suffix) and they can be used only under special circumstances (explicitly specified in the documentation). In spite of the fact that a dead object supports a certain interface, there are often misimplemented non-needed methods of that interface (methods that are not needed in some specific ways of treatment allowed for dead objects).

8.1.8 Naming, programming and documentation conventions

Function names are composed of lower-case letters with underscore used as word separator. Function and method names are always prefixed with `vcl_`, then the class name (in case of a class-specific function) or `if_` and the interface name (in case of interface method).

Pointers to unspecific VCL objects are of type `void *`. Common classes have functions with the `_new` suffix for creating objects of that class, with the `_p` suffix for predicates that test whether the argument is an object of that class, and with the `_main` suffix for internal functions used during initialisation of the library to initialise the particular class. Common interfaces have suffices `_main` and `_p` with a similar meaning.

In the next section, the functions are split into three groups – public, private, and internal. Public functions are functions which are supposed to be used by applications. Private functions are functions which are supposed to be used internally, but are not hidden and may be sometimes used by applications. Internal functions are functions similar to private, but they are only accessible in the '`vcl_internal.h`' header file.

8.2 Interface reference

8.2.1 Interface overview

`composite`
: Interface for container nodes with more children. It contains functions for enumeration of children needed for down-propagation.

`container`
: Interface for non-leaf nodes. It contains callbacks for up-propagation.

`enclosure`
: Interface for container nodes with exactly one child. It contains functions for setting and getting children.

`mask`
: Interface for leaf nodes which export their appearance as a bitmap. It contains functions for the export of appearance.

`node`
: Interface for every node. It contains private functions for connecting nodes to the node tree and helper functions for down-propagation.

`object`
: Interface for every VCL object. It contains just functions for object destroying.

`painter`
: Interface for painters – VCL backends. It contains functions needed for the drawing of nodes.

`placement`
: Interface for composite nodes with simple children replacement. It contains functions for inserting and removing children nodes.

`shape`
: Interface for leaf nodes which export their appearance as (a set of) polygons. It contains a function for the export of appearance.

`transformation`
: Interface for non-leaf nodes which change the transformation matrix for their children. It contains functions for manipulation with transformation matrices.

8.2.2 Composite interface

The composite interface is an interface for non-leaf nodes with more children. It does not have any interesting public methods.

Every object implementing the composite interface must also implement the object interface, the node interface and the container interface. There is one exception – a composite object which has only dead children does not have to support the container interface.

Tree examining methods

There are four private methods needed to implement the down-propagation. They have self-descriptive names. They are based on callbacks – the caller calls `vcl_composite_get_children`, and a composite object answers with the execution of a callback for each child. The basic order is from front to back (the first answer gives the top child, the last answer is gives the bottom child), the `_backwards` method variants use the reversed order. Basic variants return all children, `_in_bbox` method variants are allowed (but not required) to ignore some children situated outside of the `*wanted_bbox` rectangle (in the children coordinate system).

This is the only place where using dead objects is allowed – a composite node can create dead objects just before the callback answer and free it (on the stack) just after that.

```
void vcl_composite_get_children (void * obj, void (* cb)(void *, void * ),
                                 void * cb_data)

void vcl_composite_get_children_backwards (void * obj,
                                 void (* cb)(void *, void * ),
                                 void * cb_data)

void vcl_composite_get_children_in_bbox (void * obj,
                   const struct geom_rectangle * wanted_bbox,
                   void (* cb)(void *, void * ), void * cb_data)

void vcl_composite_get_children_in_bbox_backwards (void * obj,
                   const struct geom_rectangle * wanted_bbox,
                   void (* cb)(void *, void * ), void * cb_data)
```

8.2.3 Container interface

The container interface is an interface for non-leaf nodes. It does not have any interesting public methods.

Every object implementing the container interface must also implement the object interface and the node interface and one of the composite and enclosure interfaces. The canvas class is an exception for this rule. No object implementing the container interface is allowed to implement any of the mask or shape interfaces.

Change propagation methods

There are two internal methods needed to implement the up-propagation of a change. These methods inform (in the callback way) the container about changes in the child (given in the `child` arg). The `_altered` variant describes a bounding box preserving change, the `where` arg gives the bounding box of the changed part of the child. The `_changed` variant describes a major change, where `old_bbox` is the old bounding box of the child and `new_bbox` is the new bounding box of the child. All bounding boxes used here are in the children coordinate system. Common implementation of this methods is to update bounding boxes and call same method of the parent node.

```
void vcl_container_child_altered (void * obj, void * child,
                                  const struct geom_rectangle *where)

void vcl_container_child_changed (void * obj, void * child,
                                  const struct geom_rectangle *old_bbox,
                                  const struct geom_rectangle *new_bbox)
```

8.2.4 Enclosure interface

The enclosure interface is an interface for non-leaf nodes with exactly one child. It has some child-manipulating public methods.

Every object implementing the enclosure interface must also implement the object interface, the node interface and the container interface.

Child manipulating methods

Here we give the public methods with self-decriptive names for child manipulation. By using the `_set_child` method for an enclosure which already has a child, the old child is released (so it becomes a parent-free node) but not destroyed. It is not allowed to use `NULL` argument as `child`. Although newly created enclosures are usually childless, after the insertion of a child there is no way for the enclosure to become childless again.

```
void * vcl_enclosure_get_child (void * obj)

void * vcl_enclosure_set_child (void * obj, void * child)
```

8.2.5 Mask interface

The mask interface is an interface for leaf nodes which export their appearance as a bitmap. It does not have any interesting public methods.

Every object implementing the mask interface must also implement the object interface and the node interface. No object implementing the mask interface is allowed to implement any of the container or shape interfaces.

Render method

```
void vcl_mask_render (void * obj, const struct geom_transform *t,
                      const struct vcl_rectangle *bbox,
                      const struct geom_rectangle *bbox_local, u8** buff,
                      uns * buff_len, struct vcl_rectangle *retbox)
```

This method is requested for rendering the appearance of `obj` to a bitmap. The meaning of the bitmap is 1 for an occupied pixel and 0 for an unoccupied pixel. So there is only information about occupied area, not about color and so on. The `t` argument is the required transformation from the `obj`'s coordinate system to the pixel coordinate system. Arguments `bbox` and `bbox_local` are bounding boxes of the caller's area of interest – anything out of one of them is not required to be rendered correctly. `bbox` is in pixel coordinates, `bbox_local` is in `obj`'s coordinates. Arguments `buff` and `buff_len` are for bitmap buffer – if the buffer is small (or `NULL`), then the callee is responsible for reallocating it using `xmalloc` or `xrealloc` and update the value of `*buff_len`, which is the buffer size in bytes. The callee must fill `*retbox` to the (pixel) coordinates of the returned bitmap.

8.2.6 Node interface

This interface is responsible for binding objects to the VCL node tree. It contains private methods needed to connect individual objects to the tree.

Every object implementing the node interface must also implement the object interface and one of the container, mask, or shape interfaces.

Every node connected to the tree (the root node of the canvas or a node with a parent) is responsible for calling change propagation methods of its parent to participate in change propagation.

Tree binding methods

These four private methods need to be implemented for the node to be able to be connected to the tree. The last one (`_get_parent`) could be considered public. The node is required to store a pointer to its parent and some key value identifying it. `_set` functions should only be called by a container having or acquiring a node as its child. The creation of parent-child connection is done by calling appropriate public methods of the parent; and from the code of that method the `_set_parent` method with new the parent pointer is called on the child.

```
vcl_node_set_data (void *obj, int x)

vcl_node_get_data (void *obj)
```

```
vcl_node_set_parent (void *obj, void * parent)

vcl_node_get_parent (void *obj)
```

Transformation functions

There are four public functions for getting transformations from the `obj`'s children coordinate space to the pixel coordinate space (`primary` functions) and from the pixel coordinate space to the `obj`'s children coordinate space (`inverted` functions). These functions are not methods of the node interface (so a class implementing the node interface does not implement these functions) but can be called on any nodes (as the first argument). `_apply_point` applies the given transformation to one `struct geom_point`, `_apply_matrix` applies the given transformation to one `struct geom_transform`. For transforming more points it may be faster to get one `struct geom_transform` (by supplying the `_apply_matrix` function with identity) and then transform every point with it.

The return values of this functions are error values (where zero means OK) with the same meaning as standard error values of GEOMLIB.

```
int vcl_primary_apply_matrix (void * obj, struct geom_transform *dst)

int vcl_inverted_apply_matrix (void * obj, struct geom_transform *dst)

int vcl_primary_apply_point (void * obj, struct geom_point *dst)

int vcl_inverted_apply_point (void * obj, struct geom_point *dst)
```

8.2.7 Object interface

The object interface is an inteface common for all objects. It has just one function – `void vcl_object_destroy (void * obj)` . Nodes to be destroyed must be parent-free, but may have children – in that case the whole subtree is destroyed (every implementation of this interface should call `_destroy` in a recursive way).

8.2.8 Painter interface

An interface for painters – VCL backends. It does not have any interesting public methods.

Every object implementing the painter interface must also implement the object interface.

A painter should walk through the VCL node tree and draw the appropriate nodes. Common walking logic can be handled (in implementation of the painter interface) by using `struct context`.

Painting methods

These painting methods share a common structure of arguments – `node` is the current drawing node, `bbox` is the bbox to be redrawn in pixel coordinates, `bbox_local` is the bbox to be redrawn in the `node`'s coordinate system (in case of `_draw_node` or `_draw_leaf`), or the `node`'s children coordinate system (in case of `_draw_composite` – to match the coordinate system of `vcl_composite_get_children`).

A painter is supposed to be used as a sequence of calls of `_draw_begin`, `_draw_node` on the root node and `_draw_end`. The `_draw_node` function is supposed to draw the node (and, in case of composite, all its descendants, too). The remaining two functions are used in a case of implementing `_draw_node` using `struct context` and its recursive decomposing. In that case, in `_draw_node`, a dispatch and enclosure handling are done, for leaf handling `_draw_leaf` is called and for composite handling `_draw_composite`. `_draw_leaf` really draws a leaf and `_draw_composite` calls `_draw_node` on its children.

```
vcl_painter_draw_begin (void * obj, const struct vcl_rectangle *bbox)

vcl_painter_draw_end (void * obj)
```

```
vcl_painter_draw_node (void * obj, void * node,
                       const struct vcl_rectangle *bbox,
                       const struct geom_rectangle *bbox_local)

vcl_painter_draw_leaf (void * obj, void * node, const struct vcl_rectangle *bbox,
                       const struct geom_rectangle *bbox_local)

vcl_painter_draw_composite (void * obj, void * node,
                            const struct vcl_rectangle *bbox,
                            const struct geom_rectangle *bbox_local)
```

8.2.9 Placement interface

The placement interface is an interface for non-leaf nodes with simple children inserting and removing. It has public methods for children manipulating.

Every object implementing the placement interface must also implement the object interface, the node interface, the container interface and the composite interface.

Children manipulating methods

This functions should be clear, so just one remark: a removed child is released (so it becomes a parent-free node) but not destroyed.

```
vcl_placement_add_child_on_top (void * obj, void * child)

vcl_placement_remove_child (void * obj, void * child)
```

8.2.10 Shape interface

The shape interface is an interface for leaf nodes which export their appearance as (a set of) polygons. It does not have any interesting public methods.

Every object implementing the shape interface must also implement the object interface and the node interface. No object implementing the shape interface is allowed to implement any of the container or mask interfaces.

Polygonize method

```
void * vcl_shape_polygonize (void * obj, void * it, double delta,
                             const struct geom_transform *t,
                             const struct geom_rectangle *bbox,
                             u8 ** buff, uns * buff_len, uns *closed,
                             uns *points, uns hints)
```

This method is requested for expressing a part of the area occupied by `obj` by one polygon (or polyline). This function is supposed to be called in an iterative way – in every call one polygon/polyline is returned. In first call, the `it` argument should be `NULL`, and next calls should have `it` set to the return value of the previous call. If the return value is NULL, then there should be no other calls. All iterations must be done (there is no way to escape during the cycle).

The `delta` argument means the precision of the approximation. The `t` argument is the required transformation from the `obj`'s coordinate system to pixel coordinate system (the primary transformation matrix). The `bbox` argument is the bounding box of caller's area of interest – anything out of it is not required to be approximated within the given precision, but must have a correct connection to the next point inside the bounding box. Any polygons/polylines which are entirely outside `bbox` can be skipped. `bbox` is in `obj`'s coordinates.

The `buff` and `buff_len` arguments are for point buffer – if the buffer is small (or `NULL`), then the callee is responsible for reallocating it using `xmalloc` or `xrealloc` and update the value in `*buff_len`, which is the buffer size in bytes. The caller can give some hints using the `hints` argument, but the callee can ignore them. The hints are the following flags: `VCL_SHAPE_CLOSED_ONLY` – the callee can skip polylines, `VCL_SHAPE_OPEN_ONLY` – the callee can skip polygons.

Polygons/polylines are returned using an array of `struct geom_rectangle` stored in the buffer, the number of points is returned in `*points`, `*closed` indicates whether the returned object is a polygon (=1) or a polyline (=0).

8.2.11 Transformation interface

The transformation interface is an interface for container nodes which change the coordinate system of their children. It has several public methods.

Every node in the VCL node tree has its own coordinate space. A node implementing the transformation interface has two different spaces – the standard obj's coordinate space and the children coordinate space (which is equivalent to the coordinate space of its children). Between these two spaces there are transformations given by transformation matrices. The primary matrix is from children space to obj's space and the inverted matrix is from obj's space to children space.

Every object implementing the placement interface must also implement the object interface, the node interface and the container interface.

Transformation methods

The `_apply_point` methods just convert `struct geom_point` from one space to another. The `_matrix` methods are more complicated. They take a transformation as an argument and merge it with the primary or inverted transformation. Suppose that O represents the obj's space, C represents the children's space, X represents another space and $A \rightarrow B$ represents the transformation matrix from A to B. So $C \rightarrow O$ is the primary matrix and $O \rightarrow C$ is the inverted matrix. The appropriate `_matrix` methods can be described in the following way:

`_primary_apply_matrix`
: source = $X \rightarrow C$, result = $X \rightarrow O$

`_inverted_apply_matrix`
: source = $C \rightarrow X$, result = $O \rightarrow X$

`_primary_preply_matrix`
: source = $O \rightarrow X$, result = $C \rightarrow X$

`_inverted_preply_matrix`
: source = $X \rightarrow O$, result = $X \rightarrow C$

The return values of these functions are error values (0 = OK) with the same meaning as the standard error values of GEOMLIB.

```
int vcl_transformation_primary_apply_matrix (void * obj,
              const struct geom_transform *src, struct geom_transform *dst)

int vcl_transformation_inverted_apply_matrix (void * obj,
              const struct geom_transform *src, struct geom_transform *dst)

int vcl_transformation_primary_preply_matrix (void * obj,
              const struct geom_transform *src, struct geom_transform *dst)

int vcl_transformation_inverted_preply_matrix (void * obj,
              const struct geom_transform *src, struct geom_transform *dst)

int vcl_transformation_primary_apply_point (void * obj,
              const struct geom_point *src, struct geom_point *dst)

int vcl_transformation_inverted_apply_point (void * obj,
              const struct geom_point *src, struct geom_point *dst)
```

8.3 Class reference

8.3.1 Class overview

Leaf nodes:

`char` Dead mask node used by TEX-layout to draw chars.

`grid` Shape node – an equidistant rectangular infinite grid.

`path` Shape node – an open or closed path (composed of Bézier curves and segments).

`rect` Shape node – a rectangle parallel with the axes, it has a dead variant.

`segment` Shape node – one line segment.

`string` Dead mask node used by text-layout to draw strings.

Container nodes:

`affinity` Enclosure representing an affine transformation.

`group` Universal grouping node (composite, placement).

`lazy-expanding-area`
Composite node using callbacks to implement its contents.

`offset` Dead enclosure node used by TEX-layout to position chars.

`property` Enclosure for changing style properties (like color) for its descendants.

`tex-layout`
Composing node for drawing TEX expansion (or any char-positioned text).

`text-layout`
Composing node for drawing text.

The rest of the objects:

`canvas` Canvas object.

`painter-cairo`
Cairo library backend – a compile time option.

`painter-plainx`
GDK backend – default.

8.3.2 Char class

Dead node used by TEX-layout to draw chars.

Supports the node interface and the mask interface.

```
void vcl_char_init (struct vcl_char *obj,  uns code, uns charmap,
                    int font_id, real font_size,
                    struct geom_rectangle bbox)
```

The `code`, `charmap` and `font_id` arguments are fontlib values specifying a glyph. `font_size` is the size of the font and `bbox` is the bounding box of the glyph, both in local coordinates.

8.3.3 Grid class

An equidistant rectangular infinite grid. Horizontal lines are parallel with the x-axis and vertical with the y-axis, both with spacing 1 (all in the `obj`' coordinate space). If you want a different grid, use an affine transformation before.

It supports the object interface, the node interface and the shape interface.

```
vcl_grid_new (void)
```

Just creates a grid.

8.3.4 Path class

An open or closed path, composed of Bézier curves and segments. Uses `struct geom_path` as data representation, which may be external or internal.

It supports the object interface, the node interface and the shape interface.

```
struct vcl_path * vcl_path_new (struct geom_path * p)
```

Creates a path. The `p` argument is a pointer to external path-representing data (which are not freed during the destroying of the obj). If it is `NULL`, then the internal path-representing data are used (accessible as the `path -> embedded` data item).

```
void vcl_path_source_changed (struct vcl_path * obj)
```

The user can freely modify the `struct geom_path` path representation, but after that he must call `vcl_path_source_changed()` to update the path node.

8.3.5 Rect class

A rectangle parallel with the axes. It has a dead variant (names rect-s).

It supports the object interface, the node interface and the shape interface. (The object interface is not supported in the dead variant.)

```
void vcl_rect_init (struct vcl_rect *obj, struct geom_rectangle r)
```

Creates the dead variant. In `r`, there are local coordinates of the rectangle.

```
struct vcl_rect * vcl_rect_new (struct geom_rectangle r)
```

Creates the alive (normal) variant. If `r`, there are local coordinates of the rectangle.

8.3.6 Segment class

One line segment.

It supports the object interface, the node interface and the shape interface.

```
struct vcl_segment * vcl_segment_new (struct geom_point p1,
                                      struct geom_point p2)
```

Creates the path from `p1` to `p2`, in local coordinates.

8.3.7 String class

A dead node used by text-layout to draw strings.

It supports the node interface and the mask interface.

```
void vcl_string_init (struct vcl_string *obj, const char *string,
                      uns charmap, int font_id, real font_size,
                      const struct geom_rectangle *bbox)
```

`string` is a string of glyph codes (in UTF-8), `charmap` and `font_id` are fontlib values specifying the font and interpretation of glyph codes from `string`. `font_size` is the font size and `bbox` is the bounding box of the string, both in local coordinates.

8.3.8 Affinity class

A node representing an affine transformation. It changes the coordinate space of its child. It supports a set of functions for manipulation with the internal transformation.

It supports the object interface, the node interface, the container interface, the enclosure interface and the transformation interface.

```
struct vcl_affinity * vcl_affinity_new (void)
```

Just creates an affinity (with an identity matrix).

```
struct geom_transform2 * vcl_affinity_get (struct vcl_affinity *obj)
```

Gets the internal transformations.

```
void vcl_affinity_set (struct vcl_affinity *obj,
                       const struct geom_transform2 *t)
```

Sets the internal transformations.

```
void vcl_affinity_inner_merge (struct vcl_affinity *obj,
                               const struct geom_transform2 *t)
```

Merge the internal transformation with the `t` transformation; so the result is equivalent to adding a new affinity with the `t` transformation as an `obj`'s child,

If the internal primary transformation is from a space B to a space C and the primary transformation of `t` is from a space A to the space B, then the new internal primary transformation is from the space A to the space C.

```
void vcl_affinity_outer_merge (struct vcl_affinity *obj,
                               const struct geom_transform2 *t)
```

Merge the internal transformation with the `t` transformation so that the result is equivalent to adding a new affinity with the `t` transformation as the `obj`'s parent.

If ithe internal primary transformation is from a space A to a space B and the `t` primary transformation is from the space B to a space C, then the new internal primary transformation is from the space A to the space C.

```
void vcl_affinity_merge (struct vcl_affinity *obj,
                         const struct geom_transform2 *t)
```

An alias for `vcl_affinity_outer_merge`.

8.3.9 Group class

An universal grouping node. It enables to add and remove children freely.

It supports the object interface, the node interface, the container interface, the composite interface and the placement interface.

```
vcl_group * vcl_group_new (void)
```

Just creates an empty group.

8.3.10 Lazy-expanding-area class

A composite node using callbacks to implement its contents. During the creation, each instance gets three callbacks that are called by the LE-area to get information about its content. Potential children (this term is used for represented children regardless of the state of their expansion, so the potential children need not to be children of the LE-area in the VCL node tree meaning) are represented by a meaningless `void *` pointer. Children are expanded (when needed) to VCL nodes, which are cached inside the LE-area. The creator is also responsible for sending notifications (using `_notification` functions) about interesting events regarding the visualised objects.

It supports the object interface, the node interface, the container interface and the composite interface.

Callbacks

All callback headers contain the `data` argument, which is user data registered together with the callback.

```
void (* get_bbox_of_child_fn_t) (void * child_id,
                                 struct vcl_le_area * questioner,
                                 struct geom_rectangle *bbox, void * data)
```

The first callback is used by the LE-area to ask about the bbox of a child `child_id`. The callee is required to fill the `*bbox` (in local coordinates).

```
void * (* expand_child_fn_t) (void * child_id,
                              struct vcl_le_area * questioner, void * data)
```

The second callback is used by the LE-area to expand a meaningless child ID to a VCL node. The callee is required to return a (parent-free) VCL node representing the `child_id` child. The caller (LE-area) is then responsible for destroying that node.

```
void (* get_children_fn_t) (struct vcl_le_area * questioner, uns backwards,
                            const struct geom_rectangle * bbox, void * data)
```

The third callback is used by the LE-area to ask about children in the bbox. The callee is required to answer using one call `vcl_le_area_answer_child_in_bbox()` per one item inside the `*bbox`, the callee is allowed (but not required) to ignore items outside `*bbox`. The items should be answered in the order from front to back, unless `backwards` is true.

Functions

```
struct vcl_le_area * vcl_le_area_new (get_bbox_of_child_fn_t cb1,
                            void * cb1_data, expand_child_fn_t cb2,
                            void * cb2_data, get_children_fn_t cb3,
                            void * cb3_data)
```

Creates an LE-area with the appropriate callbacks.

```
vcl_le_area_flush (struct vcl_le_area * obj,
                   const struct geom_rectangle *new_bbox)
```

Flushes all cached children nodes and stored information and sets a new internal bounding box. `*new_bbox` must encompass all new potential children of the LE-area. It can be used instead of many child disappearance and child appearance notifications after a massive reorganisation of represented data.

```
void vcl_le_area_answer_child_in_bbox (struct vcl_le_area * obj,
                                       void * child_id)
```

Answers the `get_children` callback.

```
void vcl_le_area_child_altered_notification (struct vcl_le_area * obj,
                void * child_id, const struct geom_rectangle *where)
```

A notification about a bounding box preserving change in a potential child `child_id`, where `where` is the bounding box of the changed part of the potential child.

```
void vcl_le_area_child_changed_notification (struct vcl_le_area * obj,
                void * child_id, const struct geom_rectangle *old_bbox,
                const struct geom_rectangle *new_bbox)
```

A notification about a major change of a potential child, where `old_bbox` is the old bounding box of the potential child and `new_bbox` is the new bounding box of the potential child.

```
void vcl_le_area_child_transformed_notification (struct vcl_le_area * obj,
                void * child_id, const struct geom_rectangle *old_bbox,
                const struct geom_rectangle *new_bbox,
                const struct geom_transform2 * trans)
```

A notification about a major change of a potential child which has the character of an affine transformation, where `old_bbox` is old the bounding box of the potential child, `new_bbox` is the new bounding box of the potential child, and `trans` is the given transformation.

For one event, there should be only one call to one of the

- `vcl_le_area_child_altered_notification`
- `vcl_le_area_child_changed_notification`
- `vcl_le_area_child_transformed_notification`

functions.

```
void vcl_le_area_child_appeared_notification (struct vcl_le_area * obj,
                                              void * child_id)
```

A notification about a new potential child.

```
void vcl_le_area_child_disappeared_notification (struct vcl_le_area * obj,
                                                 void * child_id)
```

A notification about the vanishing of a potential child.

```
void * vcl_le_area_child_lookup (struct vcl_le_area * obj, void * child_id)
```

Does a lookup in the internal cache of children representing nodes. Returns the appropriate node of `child_id` if it is in the cache, `NULL` otherwise. The node is locked to prevent it from being destroyed by the LE-area. The callee is not responsible for destroying the returned node.

```
void vcl_le_area_child_unlock (struct vcl_le_area * obj, void * child_id)
```

Unlocks a cache item locked by `vcl_le_area_child_lookup()`.

8.3.11 Offset class

A dead node used by TEX-layout to position chars.

It supports the node interface, the container interface, the enclosure interface and the transformation interface.

```
struct vcl_offset * vcl_offset_init (struct vcl_offset * obj, real dx,
                                     real dy, void * child)
```

Creates an offset (`dx`, `dy`) with `child` as the child.

8.3.12 Property class

A node for changing style properties (like color) for its descendants. It stores a set of properties. There is no automatic change propagation after a change of the property values, so the programmer can change many properties and then call `vcl_property_changed()` to process all changes at once.

It supports the object interface, the node interface, the container interface and the enclosure interface.

```
struct vcl_property * vcl_property_new (void)
```

Creates a property node with an empty set of properties.

```
void vcl_property_set (struct vcl_property * obj, int prop_id,
                       void *prop_data)
```

Sets the property with ID `prop_id` to value `prop_data`.

```
int vcl_property_get_count (struct vcl_property * obj)
```

Returns the size of the set of properties in `obj`.

```
int vcl_property_get_nth_type (struct vcl_property * obj, int n)
```

Returns the ID of the `n`-th property.

```
void * vcl_property_get_nth_value (struct vcl_property * obj, int n)
```

Returns the value of the `n`-th property.

```
void * vcl_property_get (struct vcl_property * obj, int prop_id)
```

Returns the value of the property with ID `prop_id`.

```
void vcl_property_changed (struct vcl_property * obj)
```

Causes the change propagation (and a drawing update).

Shortcuts to set common properties

`VCL_PROP_FILL_COLOR`

```
void vcl_property_set_fill_color (void  *prop, u32 color)
```

`VCL_PROP_STROKE_COLOR`

```
void vcl_property_set_stroke_color (void  *prop, u32 color)
```

`VCL_PROP_STROKE_WIDTH`

```
void vcl_property_set_stroke_width (struct vcl_property * obj, double width)
```

`VCL_PROP_STROKE_CAP_STYLE`

```
void vcl_property_set_stroke_cap_style (void  *prop, uns cap_style)
```

`VCL_PROP_STROKE_JOIN_STYLE`

```
void vcl_property_set_stroke_join_style (void  *prop, uns join_style)
```

8.3.13 TEX-layout

A node for drawing a TEX expansion (or any char-positioned text). The data are stored as an array of `struct tex_glyph`.

It supports the object interface, the node interface and the composite interface.

```
struct vcl_tex_layout * vcl_tex_layout_new (uns copy,
                                struct tex_glyph *glyphs, uns glyphs_cnt,
                                const struct geom_rectangle *bbox)
```

Creates a TEX-layout. The glyph array is given by `glyphs` and its size is `glyph_cnt`. If `copy` is true, then the glyph array is copied, otherwise a pointer is taken (and the TEX-layout is not responsible for freeing it during the destroying). `*bbox` is the bounding box of the entire TEX-layout (in local coords).

8.3.14 Text-layout

A node for drawing text.

It supports the object interface, the node interface and the composite interface.

```
struct vcl_text_layout * vcl_text_layout_new (uns copy,
                                const char *string, int font_id,
                                real font_size,
                                const struct geom_rectangle *bbox)
```

Creates a text-layout. `string` is a string of glyph codes (in UTF-8). If `copy` is true, then the string is copied, otherwise a pointer is taken (and the text-layout is not responsible for freeing it during the destroying). `font_id` is a FONTLIB value specifying the font. `font_size` is the font size and `bbox` is the bounding box of the string, both in local coordinates.

8.3.15 Canvas class

A canvas object. It hosts the root of the VCL node tree, runs painters and handles the cooperation with GTK (events etc.).

It supports the object interface and the container interface.

```
struct vcl_canvas * vcl_canvas_new (void)
```

Just creates a canvas.

```
void vcl_canvas_setup (struct vcl_canvas *obj, int flags, real center_x,
                       real center_y, real unit_size)
```

Sets miscellaneous canvas parameters. `center_x` and `center_y` specify the relative center of the canvas – this point is fixed when resizing the canvas and it is the implicit position of $(0, 0)$. Passing the $(0.5, 0.5)$ values associates the center with the real center. `unit_size` is a multiplier for units and the flags are these:

`VCL_CANVAS_REAL_UNITS`
: the base unit is a millimeter instead of a pixel.

`VCL_CANVAS_FULL_WIDTH`
: The GTK widget allocates full width for the bounding box of the root node instead of a flexible size.

`VCL_CANVAS_FLIP_X`
: Flip the X axis (instead of X increasing in left-right direction)

`VCL_CANVAS_FLIP_Y`
: Flip the Y axis (instead of Y increasing in the up-down direction)

```
void vcl_canvas_set_root (struct vcl_canvas *obj, void * root)
```

Sets the root node. The same rules as in `vcl_enclosure_set_child` apply.

```
void * vcl_canvas_get_root (struct vcl_canvas *obj)
```

Returns the root node.

8.3.16 Painter-cairo class

A Cairo library backend. It uses alpha-blending and antialiased lines. It is a compile time option. It supports all properties.

It supports the object interface.

```
struct vcl_painter_cairo * vcl_painter_cairo_new (GtkWidget *wg)
```

Create painter-cairo on `wg` widget. (should be called internally from canvas object).

8.3.17 Painter-plainx class

A GDK backend. It does not support alpha-blending or antialiased lines. This is the default painter. It supports all properties.

It supports the object interface.

```
struct vcl_painter_plainx * vcl_painter_plainx_new (GtkWidget *wg)
```

Creates a painter-plainx on the `wg` widget. (It should be called internally from a canvas object.)

8.4 VCL Miscellanea

8.4.1 Object system implementation

The object system is implemented in the '`vcl/object.c`' and '`vcl/object.h`' files. Every interface is identified with an integer, which is also an index to the `vcl_ifaces` growing array of interface descriptors (`struct vcl_iface_dsc`). Every virtual method (generic, not an actual implementation) is identified with an integer from one to (`method_counter` - 1). Methods of one interface are assigned consecutive integer values (items `first_fn` and `fn_count` in `struct vcl_iface_dsc`).

A class is identified with `vcl_vtable`, which is an array (of length `method_counter`), in the zero slot there is a pointer `struct vcl_class_dsc`, in the next slots there are pointers to implementations of virtual functions. The check whether a given class supports a given interface is implemented by a check whether the appropriate slot for `first_fn` of that interface is non-empty. All interfaces must be defined before the first class definition, because otherwise different classes would have different sizes of `vtables` and a check whether a new interface is supported by the old class could cause invalid memory access.

8.4.2 vcl-rectangle

`struct vcl_rectangle` represents a rectangle from an included point (`lx`, `ly`) to an excluded point (`hx`, `hy`). Of course, the associated vertical and horizontal lines are included with the first point (and excluded with last one). This structure is usually used to specify a bitmap region. There are some straightforward functions in '`vcl/rectangle.h`'

8.4.3 vcl-growing-array

`struct vcl_growing_array` is a dynamic (expanding) array with a counter of used items. There are some straightforward functions in '`vcl/gas.h`'; the `vcl_ga_ensure()` function is used to ensure that a requested slot is accessible (after the call, it is accessible).

8.4.4 vcl-context

`struct vcl_context` is an internal structure used in both painters. It handles a stack of active properties and transformations, as well as the logic of tree walking during the drawing. It stores the active properties and transformations as structure items, the shaded (old) values are stored in a growing-array used as a stack. The manipulating functions can be found in '`vcl/context.h`'.

8.4.5 Packed colors

Packed colors for color properties can be accessed by the

- `uns get_r (u32 u)`
- `uns get_g (u32 u)`
- `uns get_b (u32 u)`
- `uns get_a (u32 u)`

functions, and created by the `u32 get_color (uns r, uns g, uns b, uns a)` function.

9 FONTLIB

9.1 FONTLIB overview

FONTLIB is the font rendering and manipulation library of the VRR project.

To do the most of the font manipulation and rendering, FONTLIB utilizes the FreeType library. However, FreeType is a very low-level library, does not contain everything and unfortunately it is quite buggy. Thus, a lot of handwork is implemented in FONTLIB, mostly in different font format conversions and high-level font rendering interface. We give a description how the cooperation with FreeType is implemented in Section 9.3 [FreeType library usage], page 92.

FONTLIB contains support for rendering PostScript Type1 fonts and TrueTypes, computing text bounding boxes and converting fonts among these formats. See Section 9.4 [Supported font formats], page 92 for the list and description of supported font formats.

The FONTLIB sources are located in the '`font`' directory and the public header is the file '`font/font.h`', where you can find the detailed descriptions of all functions and data structures.

9.2 FONTLIB programmers usage

All symbols defined by FONTLIB have the `font_` prefix. Before the first use of FONTLIB, the `font_init` function must be called. On the other hand, the cleanup is done by the `font_finish` function.

FONTLIB provides a powerful font server which caches the loaded fonts. Every font loaded into the fontserver gets a unique integer identification number, called the *font ID* or also the *font descriptor*. A zero or negative font ID is considered invalid. A font file can be loaded via the `font_load_file` function. The font server keeps track of loaded fonts and does not load twice the same file into the memory. Caching is managed by the FreeType library. That means that if the memory is low or there are many fonts loaded, only the recently used fonts are present in memory. The others are "swapped" and opened on demand.

Most of the communication with rendering routines, bounding box computing and other functions is done via the `struct font_ctl` structure, which is passed as an argument. Here you specify the font ID, font size (measured in millimeters), transformation, etc. Do not forget that this structure has to be properly initialized and also cleaned up by the `font_ctl_init` and `font_ctl_cleanup` functions.

The actual rendering of one glyph and a whole string is done by the functions `font_render_glyph` and `font_render_string`, respectively. These functions perform some computation, then call FreeType to do the low-level glyph rendering and return the resulting bitmap.

To compute the bounding box (in millimeters) of a given glyph or string, there are the `font_get_glyph_bbox` and `font_get_string_bbox` functions. Again, they are controlled via the `struct font_ctl` structure.

The rest of the functions does not need any special comments, just look in the reference manual or '`font/font.h`' source. We just briefly sketch the functionality. There are routines returning various font information (`font_info`, `font_t1_info`, ...). Some font formats can be converted into some other (`font_pfa_to_pfb`, `font_tt_to_type42`, `font_pfb_to_pfa`), see Section 9.6 [Font conversions], page 94 for details.

There is also the not yet fully implemented support for expanding fonts into the GEOMLIB curve representation (`font_char_decompose`). FONTLIB also includes a logic for finding the most similar font if the original one is not available (`font_search`). The FontConfig library is utilized here.

9.3 FreeType library usage

VRR uses the version 2.1.9 of the FreeType library. The homepage of the FreeType project is at http://www.freetype.org/ and FreeType is currently probably the best open source font rendering library available.

However, during our very intensive usage of FreeType in VRR we encountered an enormous amount of bugs, failures and design faults inside the FreeType library. The errors we discovered include:

- Faults during rendering fonts at small sizes.
- Lack of documentation, low quality and errors in the present documentation.
- Bugs in the new caching subsystem design.
- Heavy interface incompatibility between two very close versions (namely 2.1.7 and 2.1.9).
- The overall unstable and development character of the library.

Therefore, we chose one particular version of the library, which seemed to be the most stable one, and included its source together with the VRR source code. During the project compilation, the FreeType library is compiled, too. But this is not all, FreeType cannot be simply linked to the binaries because of the namespace conflicts with the FreeType library version used by the GTK library.

The solution is the following: the library is compiled and during the FONTLIB initialization, it is dynamically loaded into the running executable via the system function `dlopen()`. Every utilized symbol is then searched via the `dlsearch()` function and given another name, prefixed by `font_` . See the main FONTLIB source '`font/font.c`' and the internal header '`font/internal.h`' for implementation details.

We consider our solution to be a working, but quite dirty hack and we are waiting for the FreeType library to stabilize to remove it.

9.4 Supported font formats

FONTLIB accepts only scalable (vector) fonts. There is no obstacle in supporting also fixed-size (bitmapped) fonts, but since VRR is a vector editor and the fonts need to be rendered in various scales, there is no reason to support the bitmap fonts. In this section, there is a description of supported font formats.

9.4.1 PostScript Type1 fonts

The Type1 PostScript font is probably the most widely spread font format in the UNIX world.

A Type1 font program is actually a special case of a PostScript language program. The PostScript interpreter renders the font intelligently, in a device-independent manner. This allows a font developer to create one font program that can be rendered on a wide variety of devices and at many different resolutions.

A Type1 font program consists of a clear text (ASCII) portion, and an encoded and encrypted portion. The PostScript language commands used in a Type1 font program must conform to a much stricter syntax than the "normal" PostScript language programs do. Type1 font programs can include special "hints" that make their representation as exact as possible on a wide variety of devices and pixel densities.

For complete reference, see http://vrr.ucw.cz/doc/T1Format.pdf.

A Type1 font program should be a 7-bit ASCII data stream when it is sent to a PostScript interpreter. However, the programs are not always stored in this way on the host system. In environments where disk space is a concern, the files are compressed according to some scheme to reduce their size on the host system, but they need to be decompressed before they can be understood by a PostScript interpreter.

Type1 font programs are encrypted. That is, most of the actual program file has been reduced to an unreadable form that is decoded by the PostScript printer before it is executed. The encrypted data is in hexadecimal form, as a stream of digits. These digits are preceded by clear-text PostScript language code, and might have a line or two of clear-text PostScript language at the end as well.

9.4.1.1 Type1 PFA fonts

A PFA Type1 font is just Type1 font completely encoded in 7-bit ASCII.

9.4.1.2 Type1 PFB fonts

PC Type1 fonts can be stored in a compressed binary format, called PFB. These files can be unpacked as they are being downloaded to the PostScript interpreter. The file is conceptually divided into segments, each of which has a small header containing a "type" field and length information. There are three types of segments:

- TYPE 1: ASCII text. This text can be sent directly to the printer without any decompression.
- TYPE 2: This is binary data that should be converted to hexadecimal and transmitted to the printer as a stream of ASCII hex data (new lines can be inserted anywhere in hex streams, if necessary).
- TYPE 3: End-of-file indication. This is a flag that indicates that the end of the data segment has been reached.

The detailed reference can be found at http://vrr.ucw.cz/doc/Download_Fonts.pdf.

9.4.2 TrueType fonts

TrueType is the binary scalable font format originally created by Apple and nowadays is the most widely font format in the Mac and Microsoft Windows world.

A TrueType font file consists of a sequence of concatenated tables. The first of the tables is the font directory, a special table that facilitates access to the other tables in the font. The directory is followed by a sequence of tables containing the font data. These tables can appear in any order. Certain tables are required for all fonts. Others are optional depending upon the functionality expected of a particular font.

The required tables must appear in every valid TrueType font file. This is the list of the required tables:

- `cmap`: character to glyph mapping
- `glyf`: glyph data
- `head`: font header
- `hhea`: horizontal header
- `hmtx`: horizontal metrics
- `loca`: index to location
- `maxp`: maximum profile
- `name`: naming
- `post`: PostScript

The complete TrueType font reference manual can be found at http://vrr.ucw.cz/doc/TTRefMan/index.html.

9.4.3 PostScript Type42 fonts

The Type42 font is a TrueType font coded in a special way and wrapped in a PostScript font envelope. This can be used to download TrueType fonts to PostScript printers (or PostScript compatible printers) that contain a TrueType rasterizer. This method yields better print quality than can be achieved by converting a TrueType font to a Type1 one.

The complete Type42 font format specification is available at http://vrr.ucw.cz/doc/Type42_Spec.pdf.

9.5 Font rendering

FONTLIB utilizes the FreeType library for the purpose of low-level rendering and bounding box computation. When we omit the technical details of FreeType usage, the rendering is straightforward: a font is loaded, scaled, a transformation is applied and the glyph is rendered into a bitmap based on the bounding box size. The same applies to the bounding box computation.

The only exception is string rendering. Here, the rendering routine has two passes. In the first one, the exact bitmap size is computed, and in the second one all the string glyphs are actually rendered.

Rendering routines are located in the '`font/render.c`' source file.

9.6 Font conversions

FONTLIB provides some routines for font format conversions. This is mostly needed in the exports (see Chapter 11 [Export], page 100). For example, in order to include a TrueType font inside a PostScript file, it must be converted into a Type42 font. Similar rules apply to PFB Type1 fonts.

The conversion routines are realized in the '`font/convert.c`' source file. The converters are also available as standalone utilities, each with its simple wrapper around the FONTLIB.

9.6.1 PFA to PFB conversion

This conversion works by encoding the source ASCII PFA format into the binary PFB format. We perform a somewhat heuristic PFA parsing to find the three sections (see PFB file description), and we encode the middle one (the encrypted font data) into binary representation.

9.6.2 PFB to PFA conversion

The PFB to PFA conversion is straightforward, we follow the strict PFB definition, we parse the source data and convert the middle section (encrypted font data) into ASCII representation.

9.6.3 TrueType to Type42 conversion

The TrueType to Type42 conversion is tricky. The FreeType support is almost useless here and we have to implement everything manually. It is a work of black magic and intensive hacking.

First, the Type42 header is constructed according to information provided by FreeType. We then load the important TrueType font tables by manually parsing the TrueType file. Then we construct a modified TrueType file copy, changed for the purpose of Type42 enveloping. This includes building a new TrueType directory and encoding everything in ASCII.

9.7 Other FONTLIB functionality

FONTLIB ability to search for the most suitable available font is a straightforward application of the FontConfig library.

The experimental font expansion into GEOMLIB curves is realized using FreeType. FreeType includes routines for walking through the decoded font curves and we simply read these curves and convert them into GEOMLIB objects.

The informational functions simply call the appropriate FreeType function to get information about the font or just copy them from the right FreeType data structures.

10 Plugins

VRR has a support for *plugins*. A plugin is a standalone binary file that contains various functions which are distributed separately from VRR.

The plugin interface is defined in '`kernel/plugin.h`', the implementation is in '`kernel/plugin.c`'.

10.1 Plugin mechanism implementation

Each plugin is actually an ELF shared library file.

Shared libraries are libraries that are loaded by programs when they start. When a shared library is installed properly, all programs that start afterwards automatically use the new shared library. It is actually much more flexible and sophisticated than this, because the approach used by Linux permits you to:

- update libraries and still support programs that want to use older, non-backward-compatible versions of those libraries;
- override specific libraries or even specific functions in a library when executing a particular program.
- do all this while programs are running using the existing libraries.

For more informations about shared libraries, see for example http://www.linux.org/docs/ldp/howto/Program-Library-HOWTO/.

However, shared libraries can be loaded not only during program startup, but also manually, by the `dlopen()` system function. The function loads the library file and returns a handler (or just returns a handler if the library is already loaded). The library symbols can then be accessed via the `dlsym` function. Thus, the plugin can export functions and variables to the program. Sometimes, the plugin cannot be also unloaded, because it may change the VRR behavior in an irreversible way.

The VRR plugin interface maintains the list of currently loaded plugins, and for each loaded plugin the list of exported functions. The loaded plugin list is maintained as a linked list of `struct plugin_rec` structures, similarly the function lists. The functions are exported by the plugin itself during initialization. See Section 10.2 [Rules for writing plugins], page 96 for programmers usage informations.

The arguments of a plugin function are passed via the `union plugin_arg` union. The return type of the function must be either `void` or `union plugin_arg`. Only a subset of C and VRR data types is allowed in function prototypes, see `enum plugin_prototypes` for the allowed list. The function arguments are passed as an array of `union plugin_arg` unions. The function prototype is described to the VRR plugin interface by arguments of the registration `plugin_function_register` function.

When registering a function via `plugin_function_register`, the function is exported also to the Scheme interface and it is possible to use it from the Scheme console. See Chapter 13 [Scheme], page 104 for implementation details.

10.2 Rules for writing plugins

The plugin interface must be properly initialized and cleaned up by functions `plugin_init` and `plugin_finish`. A plugin is loaded via the function `plugin_load` and (possibly) unloaded by `plugin_unload`.

Every valid VRR plugin must contain the function `u32 plugin_start(void)`. Make sure the prototype is exactly this. `plugin_start` is called immediately after the successful plugin loading and its purpose is to perform initialization routines as well as plugin function registering. The

plugin flags are described in the return value. If you wish your plugin to be unloadable, make sure you set the flag `PLUGIN_UNLOADABLE`.

Every exported plugin function must of prototype `union plugin_arg func(union plugin_arg *arg)` or `void func(union plugin_arg *arg)`. The exported plugin functions are registered into the interface via the function `plugin_function_register`.

This is an example (taken from '`plugin/hell.c`'):

```
u32 plugin_start(void)
{
  plugin_function_register("tsunami", NULL, PLUGIN_T_VOID,
                           2, PLUGIN_T_REAL, PLUGIN_T_REAL);
  plugin_function_register("hilbert_curve", NULL, PLUGIN_T_VOID,
                           2, PLUGIN_T_OBJ_TLO, PLUGIN_T_INT);
  plugin_function_register("random_pastes", NULL, PLUGIN_T_VOID,
                           2, PLUGIN_T_OBJ_TLO, PLUGIN_T_INT);
  plugin_function_register("show_horizontal_points", NULL, PLUGIN_T_VOID,
                           1, PLUGIN_T_OBJ_TLO);
  plugin_function_register("test_solve_y", NULL, PLUGIN_T_VOID,
                           2, PLUGIN_T_OBJ_TLO, PLUGIN_T_REAL);
  plugin_function_register("plus", "adds two integer numbers", PLUGIN_T_INT,
                           2, PLUGIN_T_INT, PLUGIN_T_INT);
  return 0;
}
```

Similarly, when unloading an (unloadable) plugin, the function `plugin_stop` is executed, if present in the plugin.

In general, it is possible (as the plugin is a solid part of VRR after loading) to change every single variable and execute every function VRR has for its internal purposes, but doing that is discouraged. Try to write the plugins in the cleanest possible way.

10.3 Implemented plugins

Plugins, implemented for VRR, are stored in the '`plugin`' directory. They have mostly experimental and demonstration purpose, but we expect that soon in the future numberous plugins will arise.

We implemented these plugins:

- '`plugin/hell.c`': An experimental plugin used to test the VRR plugin mechanism. It contains functions for various funny picture transformations (`tsunami`, `random_pastes`, ...).
- '`plugin/heaven.c`': An experimental GUI plugin used to demonstrate the GUI plugin features. The description can be found in Section 10.4.3 [An Example of a GUI Plugin], page 98.

10.4 GUI Plugin Interface

VRR's GUI enables the plugins to register new functions and add new items in menus and toolbars. The command set of the Command Structure can be simply extended by other commands (see Section 7.3.3 [Command Editing Actions], page 63) and provides an additional interface for command editing actions for plugins (see Section 7.3.4 [Plugin Menu Functions], page 63). Moreover, the plugins can register their own GO Factory states by adding the GO Factory commands into the Command Structure (see Section 7.3.2 [Command Definitions], page 61).

10.4.1 Basic Features for Plugins

File: '`gui/cmdmgr.h`'

As an author of a GUI plugin, you can use any VRR functions you like. However, a basic interface for registering plugin functions into the GUI are provided in addition to functions for

plugin loading and unloading. During plugin load, you register the plugin with a description and obtain a unique plugin menu ID. Then you can add your functions into a specially created command category conveniently, as described in Section 7.3.4 [Plugin Menu Functions], page 63.

Or, you can create your menu commands "manually" and then feed them into the command structure to any place you like. The `plugin_ctg` category is the one reserved for plugin functions and you can access it directly. Remember, however, that you should not change or remove any commands you have not created yourself.

The plugin functions registered in the GUI do not appear in the function list of the Plugin Manager; instead, you can find them in the View toolbar and View pop-up menu.

10.4.2 How to Avoid Plugin Problems

At present, the user can load a plugin several times, which could cause some problems. For example, you store the plugin menu ID in a static variable and after another load of the plugin, the value of the variable is changed. When unregistering any of the plugin instances, the same ID is used and the old value is forgotten – you unregister one plugin instance several times and the other ones remain registered.

When adding a command "manually" into the Command structure, another problem can occur: the Command Structure fills the `next` and `parent` pointers in the added command so that it is linked in the internal structures. After adding the same command several times, the pointer structure is broken and the Command Structure gets confused and might get lost in an infinite cycle.

Thus, you should prevent the user from causing problems by loading your plugin multiple times. A simple example how to do that is:

```
int my_menu_id = -1;
int stopped;

u32 plugin_start(void)
{
    if (my_menu_id != -1)
        return 0;

    my_menu_id = plugin_menu_register("My plugin");

    // register some functions, ...

    return MY_PLUGIN_FLAGS;
}

void plugin_stop(void)
{
    if (stopped)
        return;

    plugin_menu_unregister(my_menu_id);

    stopped = 1;
}
```

You should also try to avoid collisions of variable names with other plugins' variables, for example, by prefixing the names with the name of your plugin.

10.4.3 An Example of a GUI Plugin

An example of a GUI plugin can be found in the file '`plugin/heaven.c`'. This plugin uses both the kernel and the GUI interface and demonstrates the most useful features. It is heavily commented and quite self-descriptive.

It includes registering commands via the convenient plugin interface and manually created commands as well; it also shows how to create GO Factory states and use them. It demonstrates some additional GUI features, such as property recycling (see Section 7.6.4 [Property Recycler], page 72) and context changes (see Section 7.3.1 [The Context], page 61). Moreover, it shows a complex kernel transaction which creates graphic objects dependent on one another.

11 Export

VRR supports exporting images into the PostScript, PDF and SVG formats. Export modules are located in the 'export' directory. All modules share the common public header 'export/export.h' which contains export function prototypes.

11.1 PostScript export

PostScript is the worldwide printing and imaging standard. It is used by print service providers, publishers, corporations, and government agencies around the globe. In short, PostScript is a complex programming language designed especially for printing graphics. See http://vrr.ucw.cz/doc/PLRM.pdf for complete reference. The structuring information is maintained in the form of the DSC comments inside the PostScript code, see http://vrr.ucw.cz/doc/DSC.pdf for complete definition. Without these additional informations, the document structure is nearly unrecognizable. VRR PostScript output is fully conforming to DSC conventions.

There are libraries which allow the programmer to output valid PostScript code. However, we decided to write our exporter by hand, which gives us more control on what happens in the code.

In the beginning of the output, the exporter stores the DSC header and defines its own set of command shortcuts to minimize the output size. Then the font data are stored. PostScript requires special font formats. The Type1 fonts are supported but the TrueTypes are not and must be converted into a Type42 font, which is done by FONTLIB. See Chapter 9 [FONTLIB], page 91. Every used font is dumped only once. The exporter is able to omit the font files and write DSC commands instead, which should cause loading of the specified fonts by the PostScript viewer and interpreter.

The export itself is quite straightforward. The exporter walks through the GO list, outputs the graphical environment setup (stroke color, fill color, line caps, etc.) and then the appropriate command with arguments. The document is exported as one PostScript file and every TLO object is exported as a separate document page. Every object in the output is surrounded by the `gsave` and `grestore` PostScript commands, so that the programmer can freely change graphical output properties without bothering with restoring them. The same applies to every page, which is surrouned by `save` and `restore` commands.

We should also mention that sometimes an object we are exporting is not supported by PostScript object set and we approximate it with Bézier curves, which is done by GEOMLIB (see Chapter 5 [GEOMLIB], page 15).

The exporter source code is in the file 'export/ps.c'.

11.1.1 Encapsulated PostScript

An encapsulated PostScript file is a PostScript language program describing the appearance of a single page. Typically, the purpose of the EPS file is to be included, or "encapsulated", in another PostScript language page description. The EPS file can contain any combination of text, graphics, and images, and it is the same as any other PostScript language page description with only a few restrictions. See http://vrr.ucw.cz/doc/EPSF_Spec.pdf for complete reference.

The VRR PostScript exporter supports an EPS export variant which of course exports valid EPS file containing only one page. Consult the exporter source code for details how the PS and EPS outputs differs.

11.2 PDF export

The Portable Document Format is another graphical document format published by Adobe. PDF is not a general-purpose programming language as the PostScript (see Section 11.1 [PostScript export], page 100). Instead, it a binary data format intended for interactive viewing. To improve performance for interactive viewing, PDF defines a more structured format than that used by most PostScript language programs. PDF also includes objects, such as annotations and hypertext links, that are not part of the page itself but are useful for interactive viewing and document interchange. See http://vrr.ucw.cz/doc/PDFReference16.pdf for complete reference.

The PDF file is organized into *streams*, where each stream contains some part of the document. There are cross-references among these streams. The PDF export is a little bit more complicated when compared to PostScript export (see Section 11.1 [PostScript export], page 100). The table of contents is located at the end of the file and there is a lot of indirect references, which means that the exporter has to keep track of the positions of all exported streams.

Not counting the complications caused by references, the PDF export is also straightforward. After writing the PDF header, the fonts are exported, together with the corresponding font headers. TrueTypes are fully supported, the Type1 fonts are required to be stored in the PFB format and in a special way, which is done by FONTLIB, see Chapter 9 [FONTLIB], page 91. Then the exporter walks through the GO list and exports the objects one by one by outputting the corresponding commands and arguments, preceded by graphical environment (colors, line widths, etc.) setup and surrounded by graphical stack save and restore commands. At the end, the PDF content table is stored.

The source is in the file '`export/pdf.c`', consult it for details.

11.3 SVG export

SVG (Scalable Vector Graphics) is a language for describing two-dimensional graphics and graphical applications in XML. VRR supports SVG 1.1, which is a W3C Recommendation.

SVG makes it possible to do high-resolution printing, animation, drill down, rollover and pop up text along with other special effects. It is an open standard.

More information about the SVG format is available at the Adobe's website at http://www.adobe.com/svg/, for specification, see http://www.w3.org/TR/SVG/.

SVG is based on the XML (eXtensible Markup Language). We use libxml to write valid XML code. Following the SVG DTD, the exported file contains all recommended tags and attributes.

Each graphic object is exported into a corresponding SVG graphic object (line, text, Bézier curve etc.) or into a group of cubic non-rational Bézier curves. This approximation is computed by GEOMLIB. TEX texts are expanded into single characters and approximated to common SVG text.

Except for graphic object specific parameters, we export all supported object attributes like fill color, fill opacity, stroke line cap, stroke join, visibility, stroke color, stroke width, and opacity.

In case of any error while exporting the graphic object, the export fails. At the end, all data are flushed into the file and the export finishes successfully.

In future releases, we would like to improve the TEX text importing, in the recent version it is limited (we export only printable 7-bit characters).

For more details, see '`export/svg.c`'.

12 Import

VRR is able to import subsets of the SVG and IPE v5.0 image formats. The import modules are located in the '`import`' directory. All modules share the common public header '`import/import.h`', which contains import function prototypes.

12.1 DVI import

DeVice Independent (DVI) is the TEX output file format. Its format is in detail documented in the book *Donald E. Knuth: TEX: The Program*. VRR has its own DVI file parser (located in '`import/dvi.c`') which is used mainly for TEX text handling (see Chapter 6 [Kernel], page 41 how). However, the parser is designed to be general enough to be used alone, without Kernel. The parser uses the the LibKPathSea library (see Section 2.6 [External programs], page 7) for font file lookups during DVI parsing.

The DVI parser itself is not only a parser, it is also an interpreter of the DVI simple "machine code". Consult the DVI reference or parser source for details. The output of the DVI parser consists of a list of *TEX glyphs*. See '`import/import.h`' for the definition of `struct tex_glyph` data structure.

A TEX glyph can be either a rule (which is a black-filled rectangle), in which case the placement and dimensions are given, or a character code, in which case the code, font ID and placement are given. After object scanning, the parser recomputes units into millimeters and changes the TEX placement coordinates into the Cartesian coordinates.

12.2 IPE import

The current version (6.0 at the time being) of the IPE editor is available at `http://ipe.compgeom.org/`. As in the time we were developing VRR there was installed the outdated version 5.0, we decided to write a really simple and basic import module to reuse the vast amount of IPE v5.0 pictures.

Writing the importer (located in '`import/ipe5.c`') was the work of trial and error, as the IPE native format is not documented. We mention that it is a 3-in-1 polyglot. When processed by TEX it produces the TEX writings and when processed by a PostScript interpreter, the graphics is printed. Moreover, some internal IPE information is saved in the comments.

The importer is a simple text file parser, see the source for details on IPE format. The following features are not implemented:

- splines
- font sizes
- arrows
- stroke and fill color
- line width and pattern

The discovered IPE objects are inserted into the supplied TLO.

12.3 SVG import

SVG (Scalable Vector Graphics) is a language for describing two-dimensional graphics and graphical applications in XML. VRR supports SVG 1.1, which is a W3C Recommendation.

SVG makes it possible to do high-resolution printing, animation, drill down, rollover and pop up text along with other special effects. It is an open standard.

More information about the SVG format is available at the Adobe's website at `http://www.adobe.com/svg/`, for specification, see `http://www.w3.org/TR/SVG/`.

SVG is XML based, so we use the libxml library to read tags and attributes from the imported file. We do not support all SVG features, especially groups, cascading styles, triggers, filters, some text transformations, patterns and because of our different internal arc representation, we do not support SVG arcs.

According to the imported graphic object, we read some of its attributes, which are advanced in VRR. In case of any error, the import finishes with an error status. If all needed (and eventually some optional) attributes of the imported graphic object are read, the appropriate VRR graphic object is created in the VRR kernel and these attributes are set as its properties.

We expect SVG import to support more features in future releases.

13 Scheme

VRR supports an integrated scripting language using the GUILE library. The glue code connecting VRR with GUILE as well as the source code in scheme are located in the '`scheme`' directory.

13.1 Scheme kernel data types

Files: '`scheme/glt_kernel.c`', '`scheme/glt_kernel.h`'

For accessing VRR objects from Scheme, it is needed to create Scheme objects for VRR objects. We call these Scheme objects proxies. Proxy data types are defined during kernel initialization, in the function `glt_kernel_init()`. There are three types of proxies: `o`, `anchor` and `hanger` proxies. From the user's point of view there a is different division based on the object kinds – in this division `o` proxies are in two categories: `obj` and `go` proxies, but their implementation is the same. Proxies are implemented using the GUILE's mechanism of smobs - small objects with type information and one pointer. This pointer is used to store a pointer on the target structure. So, having a proxy, it is simple to get the VRR structure (functions `scm2o()`, `scm2anchor()`, `scm2hanger()`). We wanted not to have more different proxies for one kernel structure, so we use a hash table to convert pointers to kernel structures to their proxies (functions `o2scm()`, `anchor2scm()`, `hanger2scm()`). If anyone wants a proxy to a kernel structure, it is taken from the hash table or created (and put to the hash table). A proxy increases the reference counter of the appropriate kernel structure. If garbage collector finds a proxy as unusable, then the reference is freed and the proxy is removed from the hash table.

13.2 Scheme GUI data types

Files: '`scheme/glt_gui.c`', '`scheme/glt_gui.h`'

For accessing GUI objects (i.e. windows), there are also proxies, but these proxies are completely independent of kernel proxies. GUI proxies are initialized during GUI initialization, in the function `guilelink_gui_init()`. These proxies are also implemented using smobs, but the reverse mechanism is simpler. Each window has a slot in its structure for storing a proxy which is initially empty. After a request for the proxy (function `window2scm`), a newly created proxy is stored in this slot. If the garbage collector finds the proxy as unusable, then this slot is reset to empty. GUI proxies do not increase reference counters of windows (because windows do not have reference counters), so it is possible that window is freed sooner than its proxy. So, the destroying method of each window calls `window_proxy_clean()` on the stored proxy and this function clears the content (stored pointer) of that proxy so that the cleared proxy is not considered valid (Scheme functions fail on that proxy).

13.3 Scheme bindings for VRR functions

Files: '`scheme/gl_misc.c`',

There are are three ways how to add a new function into the VRR Scheme interface – write it in Scheme, write it in C with special regard to Scheme (manual conversion of arguments from Scheme shape, call the registration function on it) and automatically generate from a common C function. A small number of functions are created in the second way – usually functions with complicated argument patterns or with another complication. They are defined in files in the '`scheme`' directory, particularly '`scheme/gl_misc.c`'. They usually have the `gl_` name prefix and arguments of types `SCM`. The vast majority of functions are generated automatically using the `snarf` script (written in GNU AWK, located at '`build/snarf`').

13.4 Scheme snarfing

Files: 'build/snarf', 'scheme/glt_common.h', 'scheme/glt_kernel.h', 'scheme/glt_gui.h', 'scheme/scheme_def.h', 'kernel/guilelink.h', 'gui/guilelink.h'

The snarf script scans the selected source headers for function prototypes written in a special manner, and for each such function it creates an encapsulation function (written to the generated source file) which is responsible for conversion of arguments and return value and which can be connected to Scheme. What does a specific header look like? snarf is looking for lines starting with SCHEME, the rest of each such line is searched for tokens – a token is a word starting with S, continuing with non-space symbols and terminated by a space symbol (which is not counted as part of the token). The string starting with the first non-space symbol after the first token and terminating with the first open parenthesis is considered to be the name of the function. The first (output) token specifies the return value of the function, the remaining (input) tokens specify arguments (in the given order).

Valid tokens are:

SINT	integer in C, exact number in Scheme
SUNS	unsigned in C, exact number in Scheme
SREAL	real in C, real in Scheme
SBOOL	unsigned in C, boolean in Scheme
SSTRINGC	const char * in C, string in Scheme
SSYMBOLC	const char * in C, symbol in Scheme
SSTRINGS	string (special kernel type) in C, string in Scheme
SSYMBOLS	string (special kernel type) in C, symbol in Scheme
SANCHOR	pointer to struct anchor in C, anchor proxy in Scheme
SHANGER	pointer to struct hanger in C, hanger proxy in Scheme
SO_*type*	pointer to struct *type*, which is descendant of struct o in VRR object hierarchy, in C, go or obj proxy in Scheme.
SW_*type*	function accepts a pointer to a struct *type*, which is a descendant of struct window in VRR GUI object hierarchy; in C, a window proxy in Scheme.

There is the SVOID token representing void return value, which is valid only as an output token. The last token may be STRANS or SMETATRANS, which signalises that the given function must be called in an appropriate transaction. All these tokens are preprocessor macros (defined in 'scheme/scheme_def.h'), so they are converted by the preprocessor to correct C source. for example:

```
SCHEME SVOID group_relink_selected_go(SO_go_group source,
                                      SO_go_group target, SO_go after); STRANS
```

is converted by the preprocessor to:

```
void group_relink_selected_go(struct go_group *source,
                              struct go_group *target, struct go *after);
```

and the generated encapsulation is:

```
static SCM sh_group_relink_selected_go (SCM ar0, SCM ar1, SCM ar2)
{
     volatile SCM retval = SCM_UNSPECIFIED;

     assert_SO (ar0, SCM_ARG1, "group-relink-selected-go", ID_go_group);
     assert_SO (ar1, SCM_ARG2, "group-relink-selected-go", ID_go_group);
     assert_SO (ar2, SCM_ARG3, "group-relink-selected-go", ID_go);
     ASSERT_GO_TRANS_SCHEME ("group-relink-selected-go");
```

```
        TRANS_BEGIN(err_buf)
              group_relink_selected_go((struct go_group *) in_SO (ar0),
                                       (struct go_group *) in_SO (ar1),
                                       (struct go *) in_SO (ar2));
        TRANS_FAILED
              throw_transaction_failed (err_buf);
        TRANS_END

        return retval;
    }
```

The header files '`kernel/guilelink.h`' and '`gui/guilelink.h`' are standard places to write simple functions accessible only from Scheme.

13.5 Scheme modules

Scheme code is separated to several modules to prevent namespace clutter. Most of them are not accessible by default (for example in the console). Module names look like `(vrr name)`. Most modules are stored in files, but there is one which is created completely in C code: `(vrr low)`. This module contains all snarfed functions from kernel.

'`misc.scm`'
: Module `(vrr misc)` – miscellaneous VRR-independent functions.

'`support.scm`'
: Module `(vrr console)` – internal functions and structures.

'`console.scm`'
: Module `(vrr console)` – internal functions for the console.

'`property.scm`'
: Module `(vrr property)` – property handling routines.

'`save.scm`'
: Module `(vrr save)` – save implementation.

'`load.scm`'
: Module `(vrr load)` – load implementation.

'`high.scm`'
: Module `(vrr high)` – high level interface, accessible to users.

'`gui.scm`' Module `(vrr gui)` – high level GUI interface, accessible to users, contains all functions snarfed from GUI.

13.6 Scheme exceptions and transactions

In VRR there may be an arbitrary depth of scheme/C switches on stack. GUILE Scheme uses exceptions for error signalization. Both exceptions and failed transactions contain some form of long jump. It is necessary to ensure that a failed transaction does not jump over some scheme sections on stack (and vice versa). It is accomplished by wrapping each C code executed from Scheme code by an anonymous transaction; and each Scheme code executed from the C code by the function `scm_call_with_dynamic_root()` which disallows any indirect returns from the Scheme code. A convenient way to call Scheme code from C is using the function `call_guile_fn_from_c` which does all the steps needed. Exceptions generated in Scheme code are translated to trans-fails in C code, and vice versa. But this conversion is not ideal, because VRR trans-fails contain just strings whereas GUILE exceptions contain arbitrary data; then an exception is early converted to a string for trans-fail and the next exception in a row is just a **trans-fail** exception.

14 Documentation

The documentation of the VRR project is split into these three parts:

- **The VRR User's Manual** – It contains a detailed user description how to control the program to use all VRR's features.
- **The VRR Programmers' Manual** – It covers the principles behind the source code and allows a programmer to get familiar with VRR sources easily. See Section 1.1 [About this manual], page 1.
- **The source code documentation** – A detailed description of implemented functions, macros and data structures which is directly written to the source code. This part of documentation is extracted using the Doxygen tool into HTML to allow fast navigation through function descriptions, structure index, etc.

14.1 Building manuals

The source code of *The User's Manual* and *The Programmer's Manual* in GNU Makeinfo format (see `http://www.gnu.org/software/texinfo/texinfo.html` for details) is located in the subdirectory '`doc/manual`' of the VRR's project tree.

Both manuals can be compiled by executing the `make` command (see Section 2.3.3 [Makefiles], page 5) in '`doc/manual`' subdirectory or by `make manual` in the root directory. The script generates printable books in DVI, PS and PDF format and also a cross-referenced HTML documentation.

Additional necessary tools beside the GNU Makeinfo Documentation Project to successfully compile the books are TEX, pdfTEX and the `bmeps` utility, available at `http://bmeps.sourceforge.net/`.

The results are generated in '`doc/manual`' to files with the prefix '`manual`' for The User's Manual and the '`progman`' for The Programmer's Manual. HTML documentations are located in '`manual_html`' and '`progman_html`' with '`index.html`' as the main file.

14.2 Building source code documentation

Source files in the project follow a syntax that can be processed with the Doxygen tool to generate an on-line documentation. Doxygen is able to automatically recognize all definitions if C source and process their description from special comments, beginning with "`/**`" or "`///`". Details about the syntax can be found at `http://www.doxygen.org/`.

The source code documentation can be built with `make doc` command executed in the VRR's root directory. Resulting HTML documentation is generated to the subdirectory '`doc/reference/html`'.

15 Future plans

This chapter documents the most important changes we plan in the future releases. Some of them are partially implemented, some of them are only designed. For the complete list of bugs, errors and planned features look at VRR Bugzilla (see Section 2.2 [Bug tracking system], page 4).

15.1 VRRLIB

The VRRLIB is quite complete for our needs.

15.2 GEOMLIB

There are many possible ways, how to improve functionality of GEOMLIB. The main plans for later versions of VRR are:

- Improve the effectiveness and geometrical stability of some computations for Bézier curves. Some current algorithms may in some situations to fail find the correct solution. When we try for example to find the intersection of a self-crossing cubic curve (directly or indirectly with a split to a pair of curves), the implemented algorithm does not find anything because of a zero resulting polynomial.
- Implement specialized geometrical methods for segments and other supported curves to increase their performance. Now, almost every routine invokes an expansion to Bézier curves.
- Implement NURBS as a new class. Rational Bézier curves are an equivalent to NURBS, but it would be nice to allow creation of this popular curve type with its many editing features.
- Add a direct support for parabolic and hyperbolic arcs. There are already some unfinished functions, like creation of general conics from 5 points. The solver of liner equation systems could be used for many useful features.
- Finish the implementation of connected and unconnected sets in the plane with compound paths as a border (*path sets*). Algorithm to compute planar arrangement of paths is already under construction.
- Implement expansion of a path with a given line style (width, ...) to outline with path set as the result. At first, the algorithms should be computed the arrangement and then for each face create set of cyclic paths in a given offset.
- Implement a support for dashed curves (or general repeated patters).
- Improve the interface for VCL and write optimized functions for expanding curves to visible segments.
- And many more features ...

15.3 Kernel

- First, we would like to implement the support of paths in the kernel. There are some beginnings of the path support now, but almost everything needs to be done yet. For example, we would like to do the following: path operations, such as: join, split, subdivide, merge, union, ... Heuristic functions which create a path from a given set of objects.
- The save/load mechanism is written in Scheme currently. That showed up to be exteremely impractical, so we need to rewrite the code to C.
- We plan to extend the set of supported objects by various geometric projections (translation, rotation, homothety, ...) and angular objects to improve the geometric contruction capabilities of VRR.

15.4 GUI

The GUI feature plans depend on the needs of other VRR modules. Namely, we plan to do the following:

- The path editing support. Now, the path manipulation is quite awkward.
- Redesign the GO Factory to be more powerful and elastic; its GO creating capabilities are somewhat limited. We also plan to change the property value entering mechanism to be more user friendly.
- Enable a better and intuitive manipulation with the geometric dependency structure so that the user can keep track about the dependencies he has created.

15.5 VCL

- Speed up rendering of some curve types (circles, etc.).
- Better support for line styles.
- Draw edges with anti-aliasing support.
- Alpha blending.

15.6 FONTLIB

- Implement our own font cache instead of the one provided by FreeType.
- Get rid of FreeType dynamic loading, as documented in Section 9.3 [FreeType library usage], page 92.
- Finish the font decomposition into GEOMLIB curves.

15.7 Plugins

- Implement the failure mechanism during plugin loading.
- Write more plugins.

15.8 Export and Import

In future versions, we would like to:

- SVG Import – support more SVG features like patterns, groups and cascading styles.
- SVG Import – SVG elliptic arc is defined by start point, end point, two radii and x-axis rotation. GEOMLIB in recent state is not able to work with this type of arc and we expect to extend GEOMLIB functions to be able to import this graphic objects.
- SVG Import and Export – improve the way we work with texts, correct the text positions.
- follow the future GEOMLIB functions and graphic object attributes, like extended line styles (dashing, . . .)
- SVG Export – improve the TEX text importing which is limited in the recent version (we export only printable 7-bit characters).

15.9 Scheme

- Simplify the snarfing process.
- Test input parameters for invalid values (for example infinite floating-point numbers).

15.10 Other

We would like to satisfy all incentive suggestions reported by users and remain in developing VRR.

Appendix A License

A.1 GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Lesser General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with

modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:
   a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
   b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
   c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

   Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

   In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:
   a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.
9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.
10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Z