

BALL Biochemical
algorithms library

A Tutorial

Oliver Kohlbacher, Heiko Klein, Andreas Kerzmann,
Andreas Moll, Andreas Kämper, Anna Katharina Dehof, Andreas Hildebrandt

Release 1.4.1
July 21, 2016

Contents

1	Introduction	1
1.1	Overview	2
2	Getting Started	5
2.1	System Requirements	6
2.1.1	External Software and Libraries	6
2.2	Installation	7
2.2.1	Configuring BALL	7
2.2.2	Building the Libraries	10
2.2.3	Installing the Libraries	11
2.3	Testing the Library	12
2.3.1	Unit Tests	12
2.3.2	Benchmarks	12
3	First Steps With BALL	13
3.1	Building Molecules	14
3.2	Handling Proteins	21
3.3	A Simple AMBER Calculation	23
4	Foundation Classes	27
4.1	The BALL File Class	29
4.2	BALL Strings	31
4.2.1	String Operations	31
4.2.2	Conversion	31
4.2.3	Predicates	32
4.2.4	Comparing Strings	33
4.2.5	Stream and Field Operations	33
5	Kernel	35
5.1	Kernel Classes	36
5.1.1	Molecular Framework	37

CONTENTS

5.1.2	Protein Framework	37
5.1.3	Nucleic Acid Framework	38
5.2	Kernel Iterators	38
6	VIEW Programming	40
6.1	Modularity	41
6.2	Messaging System	50
6.3	Design of the Visualization Classes	51
6.4	Creating Dialogs	54
6.5	User Defined Settings	55
6.6	Multithreading	57
6.7	How to Create a Geometric Primitive	59
7	Python Extensions	62
7.1	Overview	63
7.2	Installation	64
8	FAQ	66
	Acknowledgments	69
	Index	70
	References	77

1

Introduction

1. INTRODUCTION

BALL (Biochemical ALgorithms Library) is an application framework for rapid software prototyping in the field of Molecular Modeling. It provides an extensive C++ framework of data structures and algorithms for structural bioinformatics and cheminformatics. BALL is available for all major operating systems, including Linux, Windows, MacOS X. It is available free of charge under the Lesser GNU Public License (LGPL). Parts of the code are distributed under the GNU Public License (GPL). This tutorial shall help new users to make their first steps with BALL. It is *not* a full documentation. For a full documentation, please refer to the *Reference Manual* [1] which can be obtained in HTML, PDF, or postscript format.

There are also several papers and technical reports published on BALL [3, 4, 16, 15, 17, 19, 20] that give a cursory overview of its design principles and illustrate its use in Rapid Software Prototyping. If you publish results that were obtained using BALL, please cite it as follows:

O. Kohlbacher, H.-P. Lenhof: BALL – Rapid Software Prototyping in Computational Molecular Biology, *Bioinformatics* 2000, 16(9):815–824.

The latest version of BALL, bug fixes, and updates are available from our website

`http://www.ball-project.org`.

Further documentation, a Wiki and a bug tracker, can be found on

`http://ball-trac.bioinf.uni-sb.de/`

providing a Code Library with Code snippets, further tutorials, Release Notes, and FAQs.

The next section of this tutorial gives a short overview of the structure and contents of BALL. Chapter 2 describes the installation and related issues. In Chapter 3, some fundamental concepts and classes are explained. After that we will show how to use more advanced features of BALL. We will have a general overview of BALL (Chapters 4 and 5) and the usage of the visualization parts of BALL (Chapter 6). Chapter 7 will introduce the mechanisms and the usage of BALL's Python bindings. Finally, Chapter 8 tries to answer some of the most frequently asked questions.

1.1 Overview

The basis of all BALL classes is an extensive set of *foundation classes*. They provide generic implementations of advanced data structures (*e.g.*, trees, hash associative containers, hash grids, *etc.*), mathematical objects (*e.g.*, matrices, vectors, geometric objects), implementations of design patterns, object persistence, and access to the operating system (*e.g.*, networking support, file handling). An overview of the overall structure of BALL is given in Fig. 1.1.

1.1. OVERVIEW

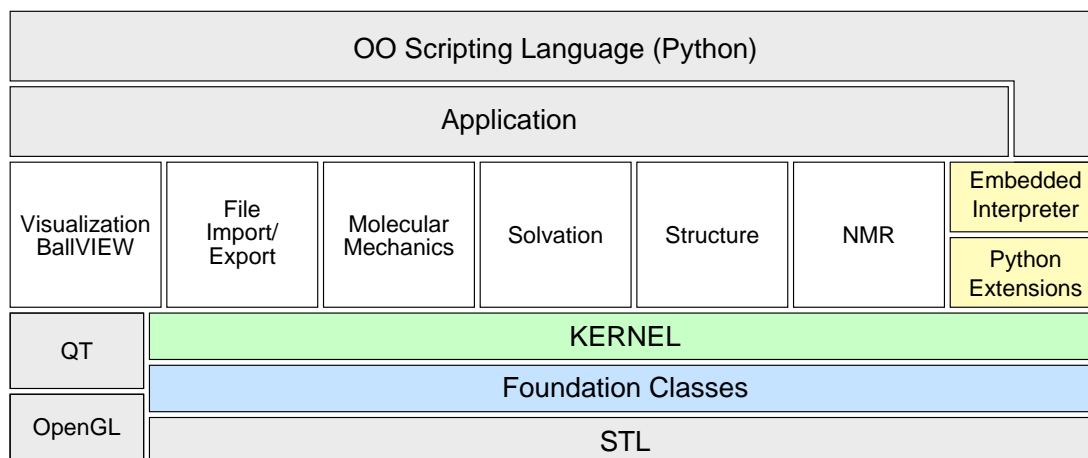


Figure 1.1: Overview of the structure of BALL

The BALL *kernel* contains the molecular data structures for the representation of atoms, bonds, molecules, proteins, *etc.* The kernel classes are implemented with the foundation classes and are carefully designed to provide maximum extensibility and flexibility.

The third layer of classes (on top of the foundation classes and the kernel) provides the functionality required for applications. We call them *basic components*. Each of these basic components is independent of the other components. We have implemented five basic components that cover Molecular Mechanics, file import/export, advanced solvation methods, structure analysis/comparison and visualization. These areas were selected, because our main interest stems from the field of protein docking. The Molecular Mechanics component does not only provide standard force fields (AMBER [6], CHARMM [5], MMFF94 [9, 10, 11, 13, 12]), but also a generic force field. This generic force field is a base class for all force fields. It implements fundamental methods to manage parameter files, parameter assignment, and atom type assignment, and defines a common interface for all force fields. Thus, large portions of code may be reused for the implementation of new force fields. The file import/export component implements general methods for efficient file handling, but also methods to read and write the most common file formats used for molecular structures (*e.g.*, PDB, MOL, MOL2, HIN, XYZ, KCF, SD). The solvation component provides methods for calculating solvation effects. The first method that accounts for solvation is the atomic contact energy by Zhang *et al.* [26]. The second approach that we implemented is a numerical solver for the Poisson-Boltzmann equation. The structure component contains algorithms to search for common structural motifs in proteins, to map molecules onto each other, and to calculate solvent accessible and solvent excluded surfaces of molecules. For the visualization of the results, we designed VIEW, a class

1. INTRODUCTION

hierarchy that visualizes BALL kernel objects with standard representations (ball and stick, van der Waals, ribbons, surfaces, *etc.*). The visualization was implemented using OpenGL [21] and Qt [22]. Hence, it is highly portable and enables the user to produce state-of-the-art graphical user interfaces with a minimum effort.

The fourth layer in BALL consists of Python bindings. Python [25] is an object-oriented scripting language that is easy to learn and very readable. BALL provides Python bindings for most of its classes, which allow the use of the BALL C++ classes from Python. This approach drastically reduces the turn-around times (no compiling and linking phase is required) and adds scripting capabilities to BALL without introducing a new and exotic scripting language. The Python bindings use the same class names as the BALL classes, therefore after the initial rapid development of a methodology, the existing Python code can be easily ported back to C++ for production purposes. Chapter 7 gives a short introduction to this feature.

2

Getting Started

2.1 System Requirements

BALL has been tested on all major platforms including recent Linux distributions, MacOS X, and Windows.

2.1.1 External Software and Libraries

BALL needs `flex` and `bison` for automatically generating parsers for various purposes. These utilities are standard software and should be installed on every contemporary UN*X machine. The newest versions are downloadable from <http://www.gnu.org/software/>.

The usage of GNU `make` is recommended, although BALL will also build with other versions of `make`. It is available from <ftp://ftp.gnu.org/gnu/> and easy to install.

The following packages can be found in our contrib package. You only need to install them manually when not using the autobuild script.

BALL uses the boost library [23] for various classes and internal tasks. A version greater or equal 1.35.0 is required.

The compilation of the visualization component VIEW also requires the Qt library (at least version 4.3), which is available from <http://qt.nokia.com/products/>.

If Qt was not installed in one of the standard library paths or the Qt header files were not installed in one of the compiler's default include directories, use `--with-qt-libs=DIR` and `--with-qt-incl=DIR` as options to `configure` (see Section 2.2) to specify the paths Qt was installed to. `configure` will also honor the `QTDIR` environment variable.

Qt also requires OpenGL [21]. On platforms that do not provide OpenGL, Mesa [18] can be used. Mesa is a 3-D graphics library with an API which is very similar to that of OpenGL. It can be obtained from <http://www.mesa3d.org>.

BALL uses GLEW, the OpenGL Extension Wrangler Library for access to the OpenGL extensions. If the GLEW library is not installed, BALLView will still work, but the advanced OpenGL features like volume rendering, vertex buffers and so on will not work. GLEW is freely available from <http://glew.sourceforge.net>.

If your machine has a graphics accelerator from Intel, ATI-AMD or NVidia, you can use the vendor-provided OpenGL drivers. The respective download locations are <http://www.intel.com/support/graphics> (Intel), <http://ati.amd.com/support/driver.html> (ATI-AMD), and <http://www.nvidia.com/content/drivers> (NVidia).

To use the Python extensions of BALL (Python is an object oriented scripting language), you will also need Python (version 2.5 or above) installed (<http://www.python.org>) and a recent version of SIP (version 4.6.0 or above,

available at <http://www.riverbankcomputing.co.uk/sip/>). SIP [24] is a tool for generating Python bindings for C++ class libraries.

Additionally you might need the FFTW [7] package for fast Fourier transformations (version 3.1.2), available from <http://www.fftw.org/>.

Please make sure that the required external C++ libraries (*i.e.* Qt and SIP) have been compiled with the same compiler (and compiler version!) as the BALL libraries. Otherwise you will most likely see a plethora of strange error messages, either while linking applications or at runtime.

2.2 Installation

We provide two different ways to build the libraries and applications:

The first one is the `autobuild` script. The build process is started by simply typing

```
|| ./autobuild
```

in the directory `BALL/source`. This procedure is especially suited for inexperienced users, since it automatically downloads a contrib source package from our website. This package contains almost all dependencies and prevents any clashes with already installed libraries, by compiling and installing them in a sub folder in the BALL directory. Afterwards, the BALL library is configured, build and linked against the downloaded libraries. We recommend to use the `autobuild` script for installations which should run out of the box on most platforms. This kind of installation will include all of BALL and BALLView's features. If this is not desired or you want to configure the library with different settings, please proceed like stated below:

As an alternative BALL can be configured with a classical `configure/make run`. If all requirements stated in Section 2.1 are met, BALL is built by issuing the following commands in the directory `BALL/source`:

```
|| ./configure
|| make
|| make install
```

The following sections give further details on the configuration of the library, on the library files created, how to test the library, and how to build BALL applications.

2.2.1 Configuring BALL

"configure" tries to gather as much information on your system as possible and then creates the necessary configuration files (`config.h`, `config.mak`, `common.mak`, and `Makefile`). The configuration of BALL may be adapted to your needs and to your system configuration from the command line by adding one or more

2. GETTING STARTED

of the options from Table 2.1. An overview of these options can also be obtained by executing "configure --help"

<code>--x-includes=DIR</code>	X include files are in DIR
<code>--x-libraries=DIR</code>	X library files are in DIR
<code>--disable-optimization</code>	do not optimize the library for speed
<code>--enable-debuginfo</code>	create debug information
<code>--enable-gsl</code>	enable the build of BALL with support for GSL (GNU Scientific Library)
<code>--disable-VIEW</code>	disable the compilation of the visualization classes
<code>--enable-64</code>	build 64 bit objects (if allowed by the compiler)
<code>--with-compiler=CXX</code>	use CXX to compile BALL
<code>--with-cxxflags=FLAGS</code>	add FLAGS to the C++ compiler flags (commas are converted to blanks)
<code>--with-ldflags=FLAGS</code>	add FLAGS to the linker flags (commas are converted to blanks)
<code>--with-arflags=FLAGS</code>	add FLAGS to the flags for the creation of the static libraries
<code>--with-dynarflags=FLAGS</code>	add FLAGS to the flags for the creation of the shared libraries
<code>--with-qt=QTDIR</code>	Qt installation is in QTDIR. This option is equivalent to setting the QTDIR environment variable. Using this option should be preferred over the next two options, which are only necessary if the Qt installation has non-standard paths for libraries and headers.
<code>--with-qt-incl=DIR</code>	Qt header files are in DIR
<code>--with-qt-libs=DIR</code>	Qt libraries are in DIR

2.2. INSTALLATION

<code>--with-moc=MOC</code>	The absolute path to the Qt meta object compiler (moc, typically found in \$QTDIR/bin/moc)
<code>--with-uic=UIC</code>	The absolute path to the Qt user interface compiler (uic, typically found in \$QTDIR/bin/uic)
<code>--with-opengl-incl=DIR</code>	OpenGL/Mesa header files are in DIR/GL
<code>--with-opengl-libs=DIR</code>	OpenGL/Mesa libraries are in DIR/GL
<code>--with-mesa</code>	use MESA instead of OpenGL
<code>--with-glew-incl=DIR</code>	GLEW header files are in DIR/GL
<code>--with-glew-libs=DIR</code>	GLEW libraries are in DIR
<code>--without-libxnet</code>	use libsocket/libnsl for linking rather than libxnet (under Solaris)
<code>--with-python=EXE</code>	Enable Python support using the Python executable in EXE. If no executable is specified (<code>--with-python</code> only), <code>configure</code> looks for an installed python in the current PATH. Typically, from the executable <code>configure</code> can figure out where the headers and the library are hidden, so that the following two options are usually not required.
<code>--with-python-incl=DIR</code>	Python includes (Python.h) is in DIR
<code>--with-python-libs=DIR</code>	Python library (libpython*.a) is in DIR
<code>--with-python-ldopts=X</code>	Use additional options X when linking with the Python library
<code>--with-sip-lib=DIR</code>	the SIP library resides in DIR
<code>--with-sip-incl=DIR</code>	the SIP header file resides in DIR
<code>--with-sip=DIR</code>	the SIP executable resides in DIR
<code>--without-xdr</code>	no RPC/XDR headers available - do not build portable binary persistence support

2. GETTING STARTED

<code>--with-fftw-lib=DIR</code>	Enable support for the FFTW library and search for the library in <code>DIR</code> .
<code>--with-fftw-incl=DIR</code>	Header files for FFTW are in <code>DIR</code> . Required for non-standard include paths only.
<code>--help</code>	display help information

Table 2.1: *Options for configure*

For example, to compile BALL without the visualization component, specify

```
|| configure --disable-VIEW
```

To compile the visualization classes using the Qt installation in `/opt/misc/qt`, specify the path to the Qt directory as follows:

```
|| configure --with-qt=/opt/misc/qt
```

If the headers or libraries are installed in non-standard directories, you can also specify them separately:

```
|| configure --with-qt-libs=/opt/lib  
|| --with-qt-incl=/opt/qt/include
```

If Mesa should be used (when compiling under Linux), the correct options might look like this:

```
|| configure --with-qt-libs=/opt/qt/lib  
|| --with-qt-incl=/opt/qt/include  
|| --with-opengl-libs=/opt/mesa/lib  
|| --with-opengl-incl=/opt/mesa/include
```

2.2.2 Building the Libraries

After the successful termination of `configure`, issuing `"make"` will build the shared libraries. Two different libraries will be built:

<code>libBALL.so</code>	the main BALL library
<code>libVIEW.so</code>	the visualization classes

The latter library is not built if `"--disable-VIEW"` is specified or `configure` cannot find X libraries, OpenGL libraries, or Qt libraries (and the respective headers).

It is also possible (although not recommended) to build the corresponding static libraries `libBALL.a` and `libVIEW.a` using `"make staticlibs"`. Please note that statically linked binaries are huge.

2.2.3 Installing the Libraries

After compiling the libraries, they are installed in `BALL/lib/$BINFMT/` when calling `"make install"` where `$BINFMT` is the binary format as determined by `configure`. Currently, the only way to install the libraries somewhere else is by moving them by hand to the desired destination. Wherever you install the shared libraries, please make sure to include their location in the `LD_LIBRARY_PATH` environment variable.

If you are using `csh`, `tcsh`, or similar shells, use the command

```
|| setenv LD_LIBRARY_PATH DIR
```

to set the library path. If you are using `sh`, `bash`, or related shells, try

```
|| LD_LIBRARY_PATH=DIR  
|| export LD_LIBRARY_PATH
```

If you installed the shared libraries in a directory that the dynamic linker `ld` searches by default, it is not necessary to set `LD_LIBRARY_PATH`.

2.3 Testing the Library

2.3.1 Unit Tests

BALL provides an extensive suite of test programs to ensure the correctness of the code on all platforms. This test suite requires a lot of patience since the compilation takes quite some time. However, we recommend to run these tests to ensure that the library is fully operational. The test programs are located in the directory `BALL/source/TEST`. To compile and run the test suite, use `"make test"`. Please make sure that `LD_LIBRARY_PATH` is correctly set, otherwise the execution of the test programs will fail.

Each of the test programs tests one or more classes of BALL. When a test program (e.g. `Atom_test`) is run, the program prints either "OK" (if all tests passed) or "FAILED" if any of the tests failed. `"make test"` runs all tests and complains if a certain test fails.

If this happens, please let report a bug through our online bug tracking system at <http://ball-trac.bioinf.uni-sb.de>.

For all bug reports, please include your system configuration, the file `config.log` (which contains the results of tests configure performed), and the file `BALL/include/BALL/config.h` (which contains the compiler defines used by BALL).

2.3.2 Benchmarks

If you want to know how fast the version of BALL is compared to other systems, you might want to run the benchmark suite in `BALL/source/BENCHMARKS`. You can compile the benchmark suite by changing to that directory and running `"make"`. After that, run the benchmarks with `"make bench"`. Depending on your hardware and whether you compiled BALL with or without optimization, running the benchmarks will take up to several minutes. Upon termination, it will print an overall number, the BALLStones. This number is a crude estimate of the performance you can expect for a mix of typical BALL applications. The benchmark suite currently includes benchmarks for the BALL kernel data structures, file I/O, the AMBER force field, and the Poisson-Boltzmann solver. The BALL web site contains a list of benchmark results for different platforms, please feel free to submit your results for inclusion.

3

First Steps With BALL

3.1 Building Molecules

Since BALL is intended for Molecular Modeling, the classes representing atoms, bonds, and molecules are of central importance. In this first example, we will construct a water molecule “by hand” to illustrate the use of these classes. Typical applications would read molecular structures from a file. This will be shown in the second example. The whole source code for the first example is given in Listing 3.1. To start the construction of a water molecule, we first create an (empty) atom. Atoms are represented by the class `Atom`.

```
|| Atom* oxygen = new Atom;
```

Atoms have a number of attributes. As we created this atom without specifying any of its properties, these attributes are set to their defaults. Table 3.1 lists the attributes of an atom along with its *accessors* (methods to access or modify an attribute) and default values.

Attribute	Type	Accessors	Default value
name	String	setName getName	" "
element	Element	setElement getElement	Element::UNKNOWN
charge	float	setCharge getCharge	0.0
radius	float	setRadius getRadius	0.0
type name	String	setTypeNames getTypeNames	" "
type	Atom::Type	setType getType	Atom::INVALID_TYPE
position	Vector3	setPosition getPosition	(0, 0, 0)
velocity	Vector3	setVelocity getVelocity	(0, 0, 0)
force	Vector3	setForce getForce	(0, 0, 0)

Table 3.1: *Attributes of an atom along with its accessors and default values.*

For example, we can assign the element for our new atom:

```
|| oxygen->setElement(PTE[Element::O]);
```

The expression `PTE[Element::O]` returns an instance of class `Element`. It is assigned to our atom using the `setElement` method. We leave our new atom at the

3.1. BUILDING MOLECULES

default position (0, 0, 0) and create two new atoms, which will be the still missing hydrogen atoms:

```
Atom* hydrogen1 = new Atom;
Atom* hydrogen2 = new Atom;
hydrogen1->setElement(PTE[Element::H]);
hydrogen2->setElement(PTE[Element::H]);
```

Now we have to assign the correct coordinates to the two hydrogen atoms. The method `setPosition` takes an instance of `Vector3` as an argument. This class is used to represent coordinates and vectors in three-dimensional space. An object of type `Vector3` can be constructed from three floating point numbers which represent the *x*, *y*, and *z* coordinates. Thus, we can assign the coordinates as follows:

```
hydrogen1->setPosition(Vector3(-0.95, 0.00, 0.0));
hydrogen2->setPosition(Vector3( 0.25, 0.87, 0.0));
```

Now, our three atoms are of the right type and at the right positions. However, we do not yet have a molecule, so let's create one:

```
Molecule* water = new Molecule;
```

Molecules are represented by the `Molecule` class. Each instance of this class may contain an arbitrary number of atoms. Using the `insert` method, we can construct a molecule from our atoms:

```
water->insert(*oxygen);
water->insert(*hydrogen1);
water->insert(*hydrogen2);
```

For a complete water molecule, we still need two bonds. This can be achieved with the method `createBond`:

```
oxygen->createBond(*hydrogen1);
oxygen->createBond(*hydrogen2);
```

or

```
hydrogen2->createBond(*oxygen);
```

To verify that everything worked as expected, we might print the number of atoms in the molecule or the number of bonds for each atom:

```
cout << "#_of_atoms_in_water:_"
      << water->countAtoms() << endl;
cout << "#_of_bonds_of_oxygen:_"
      << oxygen->countBonds() << endl;
cout << "#_of_bonds_of_hydrogen1:_"
      << hydrogen1->countBonds() << endl;
cout << "#_of_bonds_of_hydrogen2:_"
      << hydrogen2->countBonds() << endl;
```

3. FIRST STEPS WITH BALL

The method `countAtoms` is available for all kernel classes that might contain atoms and returns the total number of atoms for this object. The method `countBonds` returns the number of bonds the atom shares. An atom can have at most eight bonds.

We can also verify the bond distances:

```
float distance = oxygen->getDistance(*hydrogen1);  
  
cout << "bond_length:_ " << distance << endl;
```

`getPosition` is the complementary method of `setPosition`: it returns the current position of an atom. The return value is again of type `Vector3`. The length of this vector is then returned by the `getLength` method.

Water molecules rarely occur alone, so we are going to create further water molecules. All BALL kernel classes are container classes and support *deep copying*, *i.e.* when assigned or copy constructed their contents are copied as well. So, we can easily create a new water molecule:

```
Molecule* water2 = new Molecule(*water);
```

This molecule is an exact copy of our original water molecule. Especially, the atoms have the same position as in the original. So we want to shift the whole molecule to another position. In principle, we could access all atoms in the copy and add a constant translation vector to their position. However, there is a simpler way. BALL provides so-called *processors*. These processors may be applied to any of the kernel objects and perform an operation on any objects they encounter. For example the `TranslationProcessor` performs a simple translation on every atom it finds. The use of the processors is very simple. All kernel classes define an `apply` method which takes a processor as an argument. In order to translate our second water molecule by a certain distance, we first create a `TranslationProcessor`:

```
TranslationProcessor translation(Vector3(5, 0, 0));
```

The translation vector is specified as the argument of the constructor. Now we may translate the atoms of our water molecule by a simple call to `apply`

```
water2->apply(translation);
```

Another important kernel class besides atoms and molecules is `System`. A system is a collection of atoms, molecules, or any other kernel objects. For example, we can store our two water molecules in a system object:

```
System S;  
S.insert(*water);  
S.insert(*water2);
```

We can now manipulate the two water molecules simultaneously. For example a further application of the translation processor to the system would apply the translation to both molecules. But now we would like to have a look at what we built. So we might write our system to a file and inspect it with a molecule viewer (*e.g.* `BALLView`).

3.1. BUILDING MOLECULES

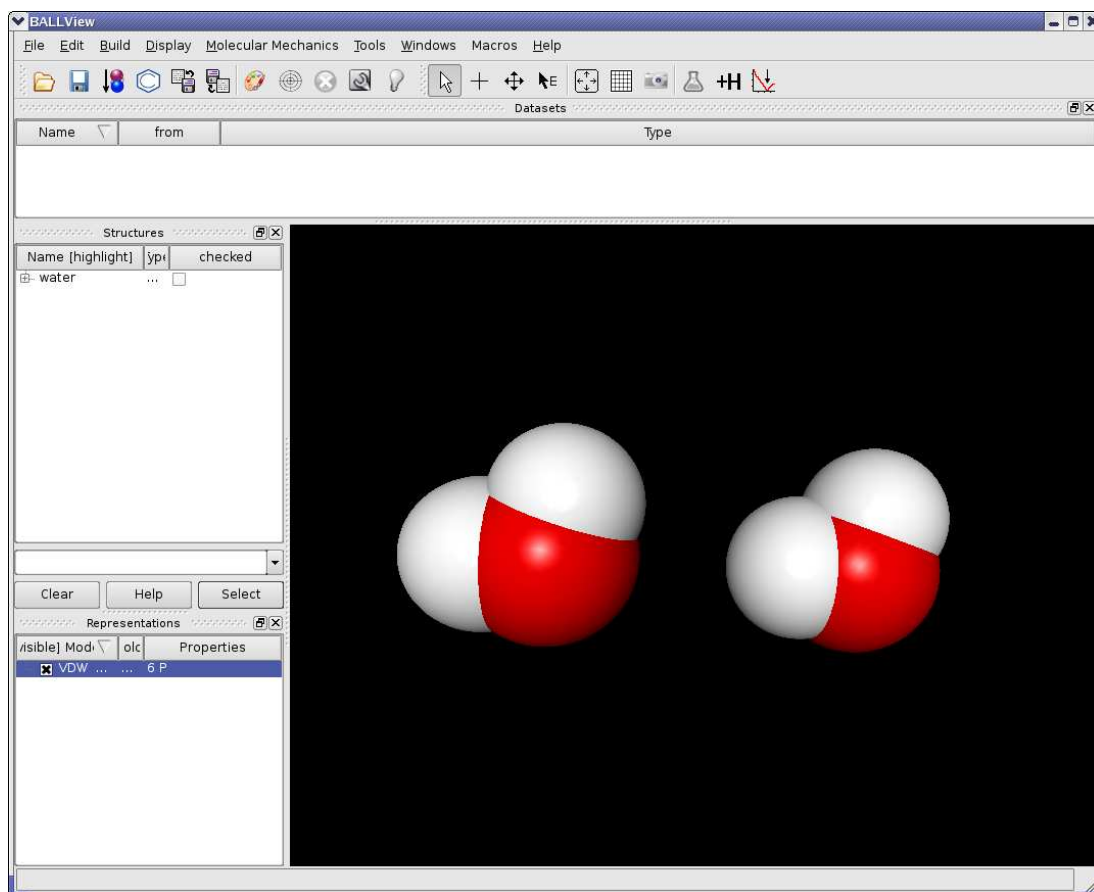


Figure 3.1: Screenshot of BALLView showing the result of the first example (tutorial1.C)

BALL supports a variety of file formats. For example we could write the system to a HyperChem HIN file [14]:

```
HINFile outfile("water.hin", std::ios::out);
outfile << S;
outfile.close();
```

These three lines of code create a `HINFile` object which is used to read and write HyperChem files. The first line opens a file named `water.hin` for output (`std::ios::out`; if the second argument is omitted the file is opened for reading only). The second line uses the stream operator `<<` to write the contents of system `S` to this file. The file is closed in the third line.

As all molecules and atoms we created are inside the system, they are deleted automatically as soon as the system is deleted.

If this short program is run it creates the following output:

```
# of atoms in water: 3
```

3. FIRST STEPS WITH BALL

```
|| # of bonds of oxygen: 2  
|| # of bonds of hydrogen1: 1  
|| # of bonds of hydrogen2: 1  
|| bond distance: 0.95
```

It also creates a file named `water.hin` which contains the two water molecules. Fig. 3.1 shows the contents of this file in the BALLView viewer.

To compile this small example, we still have to include a few header files. The complete file is shown in Listing 3.1 on Page 19 and can be found in `BALL/source/TUTORIAL/tutorial1.C`.

First, we have to include `iostream` as we use `cout` and `endl` to print some text. Then, we have to include the headers for the BALL kernel classes `Atom`, `Bond`, `Molecule`, `System`, and `PTE`. The headers for the HyperChem file support are found in the directory `BALL/FORMAT` and the headers for the `TranslationProcessor` class are found in `BALL/STRUCTURE/geometricProperties.h`.

As all BALL classes are hidden in the BALL namespace, we have to give access to this namespace with the command

```
|| using namespace BALL;
```

Similarly, we also use the namespace `std` which contains `cout` and `endl`.

If you have BALL installed, you might now try to compile this example. Simply change to the directory `BALL/source/TUTORIAL` which contains all the examples from this tutorial and type

```
|| make tutorial1  
|| ./tutorial1
```

This should build and execute the example. Please remember to set the environment variable `LD_LIBRARY_PATH` to the directory where your libraries are installed.

After the successful execution of the example, a file named `water.hin` should appear in the current directory. This file contains the two water molecules. You might inspect it with the molecule viewer BALLView. Fig. 3.1 shows a screenshot of BALLView displaying the two water molecules.

3.1. BUILDING MOLECULES

Listing 3.1: *The complete source code of the first example.*

```
// tutorial example 1
// -----
// build two water molecules and write them to a file

// needed for cout
#include <iostream>

// the BALL kernel classes
#include <BALL/KERNEL/atom.h>
#include <BALL/KERNEL/bond.h>
#include <BALL/KERNEL/molecule.h>
#include <BALL/KERNEL/system.h>
#include <BALL/KERNEL/PTE.h>

// reading and writing of HyperChem files
#include <BALL/FORMAT/HINFile.h>

// the TranslationProcessor
#include <BALL/STRUCTURE/geometricTransformations.h>

// we use the BALL namespace and the std namespace
using namespace BALL;
using namespace std;

int main()
{
    // we create a new atom called oxygen
    // and set its element to oxygen (PTE[Element::O])
    Atom* oxygen = new Atom;
    oxygen->setElement(PTE[Element::O]);

    // now we create two hydrogen atoms...
    Atom* hydrogen1 = new Atom;
    Atom* hydrogen2 = new Atom;
    hydrogen1->setElement(PTE[Element::H]);
    hydrogen2->setElement(PTE[Element::H]);

    // ...and move them to approximately correct positions
    hydrogen1->setPosition(Vector3(-0.95, 0.00, 0.0));
    hydrogen2->setPosition(Vector3( 0.25, 0.87, 0.0));

    // We create our water molecule...
    Molecule* water = new Molecule;

    // ...and insert the three atoms into the molecule.
    water->insert(*oxygen);
    water->insert(*hydrogen1);
```

3. FIRST STEPS WITH BALL

```
water->insert(*hydrogen2);

// Then we build the two O-H bonds
oxygen->createBond(*hydrogen1);
oxygen->createBond(*hydrogen2);

// Some statistics: Molecule::countAtoms()
// returns the number of atoms, Atom::countBonds()
// the number of bonds the atom shares
cout << "#_of_atoms_in_water:_\n"
      << water->countAtoms() << endl;
cout << "#_of_bonds_in_oxygen:_\n"
      << oxygen->countBonds() << endl;
cout << "#_of_bonds_of_hydrogen1:_\n"
      << hydrogen1->countBonds() << endl;
cout << "#_of_bonds_of_hydrogen2:_\n"
      << hydrogen2->countBonds() << endl;

// compute the bond length
float distance = oxygen->getDistance(*hydrogen1);

cout << "bond_length:_\n" << distance << endl;

// Now we copy our molecule using a copy constructor.
Molecule* water2 = new Molecule(*water);

// A translation processor moves the second molecule
// 5 Angstrom along the x axis
TranslationProcessor translation(Vector3(5, 0, 0));
water2->apply(translation);

// We insert our two molecules into a system
System* S = new System;
S->insert(*water);
S->insert(*water2);

// and write this system to a HyperChem file
HINFile outfile("water.hin", std::ios::out);
outfile << *S;
outfile.close();

// We delete the system. This also deletes
// the molecules and the atoms therein
delete S;
}
```


3.2 Handling Proteins

In the second example we take a step towards real world applications. Instead of constructing our own molecules, we now read a protein from a *PDB file*. The PDB format [2] is a standardized file format for molecular structure data. In our case, we will read BPTI (bovine pancreatic trypsin inhibitor), a small protein:

```
PDBFile infile("bpti.pdb");
System S;
infile >> S;
infile.close();
```

In the first line, we create a `PDBFile` object and assign it to a file named `bpti.pdb`. Then we create an empty system `S` and read the contents of the PDB file into the system using the stream operator `>>`. This use of the stream operators is possible for all file formats in BALL (see also the first example on Page 17). Hence, you can easily switch file formats by changing just the type of the `infile` object (*e.g.* replace it by `HINFile` to read HyperChem files).

We can now verify whether the file was correctly read. BPTI should have 454 atoms. As mentioned before, for each kernel object containing atoms (*i.e.* classes that are derived from `AtomContainer`), we can obtain the actual number of atoms by calling `countAtoms`:

```
cout << "#_of_atoms_in_BPTI:_ " << S.countAtoms() << endl;
```

Now, we are interested in the sequence of BPTI. Since BPTI contains only a single chain, we might just traverse all residues of this chain and write their names to `cout`. This is done by *iterators*. Iterators are objects that can be used to traverse container objects (*e.g.* lists, arrays, or – in our case – proteins). Iterators “point” to an element of a container object. They may be incremented to get the next element and they may be dereferenced similar to pointers by using `*` or `->`. In fact, C pointers can be thought of as iterators. The use of iterators in BALL is similar to the use of iterators in the *Standard Template Library* (STL). But there is a significant difference. STL containers usually contain objects of one single type (*e.g.* a list of strings). In BALL kernel objects, this is different. A system may contain atoms, proteins, molecules, residues, *etc.* This leads to a difference in the interface. A typical `for` loop using STL iterators to access all elements of a list looks as follows:

```
list<string> string_list = ...;
list<string>::iterator list_it;

for (list_it = string_list.begin();
     list_it != string_list.end();
     ++list_it)
{
    cout << *list_it << endl;
}
```

3. FIRST STEPS WITH BALL

Here, we use a list iterator (`list_it`) to traverse the whole list (`string_list`). The method `begin()` returns an iterator that points to the first element of the list. In the `for` loop we increment the iterator until it equals the iterator returned by `end()`. This method returns a past-the-end iterator, *i.e.* the iterator points beyond the last element of the container. In the body of the `for` loop, we access the list element the iterator points to using the `*` operator.

Traversing a BALL kernel data structure is as simple as traversing a list with STL. But since kernel objects may contain objects of a variety of types, we have to define over which objects we intend to iterate. For example iterating over all residues of our system requires a `ResidueIterator`. Clearly, we also need different `begin()` and `end()` methods for all data types. Hence, the loop that prints the sequence of our protein reads as follows:

```
ResidueIterator res_it;
for (res_it = S.beginResidue();
     res_it != S.endResidue();
     ++res_it)
{
    cout << res_it->getName() << "_";
}
cout << endl;
```

This loop iterates over all residues in `S` and uses the method `Residue::getName()` to access the residues name. All proteins and residues are traversed in the order in which they appear in the PDB file. Since the PDB format requires the residues to start with the N-terminus, the sequence is printed in the correct order (N-terminus to C-terminus).

The source code for this example can be found as `tutorial2.C` in `BALL/source/TUTORIAL`. The PDB file `pbt1.pdb` can also be found in this directory.

3.3 A Simple AMBER Calculation

Having introduced the basics of handling proteins in the last chapter, we now turn towards real-life examples: performing a molecular dynamics simulation with a protein structure from PDB. Again, we will be using BPTI, reading it from a PDB file as in the previous example:

```
|| PDBFile infile("pdb4pti.ent");  
|| System S;  
|| infile >> S;  
|| infile.close();
```

The file we chose to read here is the original file as obtained from the PDB, therefore it contains neither hydrogen atoms, nor bonds. The BALL class `FragmentDB` provides a convenient way to solve both problems. `FragmentDB` is an extendible database of residue structures. By comparing the residues in the PDB file with the reference templates in the `FragmentDB`, we can identify the missing bonds and hydrogen atoms. A matching between atoms is computed based on the names, so the atom names have to adhere to the PDB naming convention. For deviating naming schemes, `FragmentDB` provides a member instance `normalize_names`, which tries to convert the names to the PDB naming convention. `normalize_names` is a processor, so we can apply it to any given kernel data structure:

```
|| FragmentDB db("fragments/Fragments.db");  
|| S.apply(db.normalize_names);
```

Instantiating `FragmentDB` usually takes a few seconds to parse the fragment database and the naming conversion tables stored in `BALL/data/Fragments`. The resulting data may then be used by two very handy processors: `add_hydrogens` and `build_bonds`. As their names suggest, those processors add the missing hydrogen atoms and rebuild missing bonds. We will now add the missing atoms and bonds of BPTI through simple application of the two processors:

```
|| S.apply(db.add_hydrogens);  
|| S.apply(db.build_bonds);
```

Now we have constructed a complete protein structure of BPTI. We can verify this by applying a `ResidueChecker` to the system. `ResidueChecker` is a processor that performs a number of consistency checks on a given kernel data structure:

- check for missing atoms
- check for overlapping atoms (closer than 0.5 Å)
- check for integrality of residue charges (not relevant here)
- check bond lengths (should be within 15% of the template bond lengths)

3. FIRST STEPS WITH BALL

The information on missing atoms and bond lengths is taken from an instance of `FragmentDB`:

```
|| ResidueChecker rc(db);  
|| S.apply(rc);
```

If the `ResidueChecker` notices a problem with the structure, it will print a warning. In our case, it was (hopefully) correct, so nothing will happen.

Although our protein structure is correct, the positions of the added hydrogen atoms are only approximations of the true positions. `AddHydrogensProcessor` tries to set the positions based on the positions given in the `FragmentDB` templates, which usually deviate from their optimal position in the given structure. We can relax those hydrogen positions using a molecular mechanics calculation. We will use the AMBER force field [6] and optimize the hydrogen positions while keeping the heavy atoms rigid. The AMBER force field is implemented in the `AmberFF` class. Instantiating a force field and setting up a calculation is very simple:

```
|| AmberFF amber(S);
```

This constructor-call creates a new `AmberFF`, reads the parameter file (the default one, `amber94.ini`, which corresponds to the AMBER file `parm94.dat`), and constructs some internal data structures. This particular instance of the force field is now *bound* to the system and all its actions will apply to that system, unless a different system is specified in a call to `setup`.

We now want to optimize the hydrogens only. This can be achieved through the *selection* concept of the kernel classes. Whether an atom is selected or not has different meanings at different stages of the force field calculation. When calling `setup` (as the above constructor does implicitly), the force field will ignore all unselected atoms and only selected atoms will be part of the computation. There is one special case: If the whole system is unselected, it will be treated as if it had been selected completely (this is just for convenience).

After `setup` has been called, the selection gets a different meaning. It now indicates which atoms are to be optimized (in an energy minimization), moved (in an MD simulation) or considered for the force and energy evaluations at all. Thus by deselecting the whole system (the default) we ensure that all atoms are considered by the force field. If we want to optimize the hydrogen atoms only, we have to select them. One way to do that is the `Selector` class. Given a BALL *kernel expression* (see the documentation for `Expression` for details), it will select all atoms for which the expression evaluates to true. In our case, the expression `"element(H)"` describes the set of atoms we are interested in:

```
|| Selector hydrogen_selector("element(H)");  
|| S.apply(hydrogen_selector);
```

An energy minimization of those hydrogens is done using the `Conjugate-GradientMinimizer`. It is not hard to figure out that this class indeed implements

3.3. A SIMPLE AMBER CALCULATION

a conjugate gradient energy minimization. In a similar fashion as the force field is bound to a system, the energy minimizer instances are bound to the force field. Again, we can use a constructor to do most of the work:

```
ConjugateGradientMinimizer cgm(amber);
cgm.setEnergyOutputFrequency(1);
cgm.setMaxGradient(5.0);
std::cout << "Minimizer_options:" << std::endl;
cgm.options.dump();
cgm.minimize(100);
```

The minimizer object `cgm` has a variety of options controlling its behavior (please consult the reference manual for all possible options). After instantiating it, we first adjust its *energy output frequency*, i.e. the number of iterations performed before a status message is printed. This status message contains information on the current energy and the gradient. `setMaxGradient` adjusts the convergence criterion for the minimizer: as soon as the RMS gradient of the system falls below 5 kJ/(mol Å), the minimization is aborted. Please note that all energies in BALL are in kJ/mol, all positions and distances in Å and therefore the gradient in kJ/(mol Å). The current settings of the minimizer are all stored in the member `options` (a public instance of `Options`), so the internal state of the minimizer is readily obtained by dumping `options` to `cout`.

Finally, a call to `minimize` with argument 100 will cause the minimizer to run for (at most) 100 iterations. The result is a structure of BPTI containing all the atoms at optimized positions, so now we can perform an MD simulation of the whole protein. Obviously, we now have to deselect the hydrogen atoms. Selection is recursive, so deselecting or selecting a residue will select/deselect all its atoms, selecting a system will select all its molecules, and so on. For a more detailed description, please read Section 5.1. The easiest way to deselect all atoms is therefore to deselect the whole system:

```
S.deselect();
```

Similar to energy minimization, molecular dynamics (MD) simulation is also implemented as its own class, `MolecularDynamics`. There are two derived classes: `CanonicalMD` and `MicroCanonicalMD`, implementing an MD simulation in the canonical ensemble (isothermal) and the microcanonical ensemble (adiabatic).

For a protein immersed in water, the canonical ensemble is the obvious choice. We will furthermore run the simulation in a cubic box with periodic boundary conditions. So the first step is to set up that box and fill it with water. Luckily, water is the default solvent in BALL, so all you have to do is let the force field know that you want to set up a box with water around the current system:

```
amber.options[PeriodicBoundary::Option::PERIODIC_BOX_ENABLED]
    = true;
amber.options[PeriodicBoundary::Option::PERIODIC_BOX_ADD_SOLVENT]
```

3. FIRST STEPS WITH BALL

```
    = true;
    amber.setup(S);
```

Setting the option `PERIODIC_BOX_ENABLED` will cause the force field to enable periodic boundary conditions at the next call to `setup`. In analogy, setting the option `PERIODIC_BOX_ADD_SOLVENT` will add the default solvent to the box. By default, the box is defined as the bounding box of the system augmented by 5 Å in each direction. You can also specify arbitrary bounding boxes or different solvents. Please refer to the documentation of `PeriodicBoundary` for a more detailed description.

We can now instantiate the molecular dynamics object, set up a run at 300 K and perform a few MD simulation steps:

```
CanonicalMD md(amber);
md.setReferenceTemperature(300);
md.simulate(10);
std::cout << "Simulation_settings:" << std::endl;
md.options.dump();
```

As in the force field and the energy minimizer, the MD simulation object stores all its settings in an `options` object. The `simulate` method simulates a given number of MD steps.

The source code for the complete example can be found as `tutorial3.C` in `BALL/source/TUTORIAL`. Further documentation is available for all classes in the **BALL Reference Manual**. The header files required for molecular mechanics reside in `BALL/include/BALL/MOLMEC` and its subdirectories.

4

Foundation Classes

4. FOUNDATION CLASSES

The BALL foundation classes are a collection of useful classes used throughout the whole BALL kernel. The following sections shall help you to get acquainted with some of the more important ones. The foundation classes can be broken down into several categories:

- general data structures (*e.g.* strings, portable data types, hash containers)
- concepts (*e.g.* management of properties, object persistence)
- mathematical data structures (*e.g.* vectors, matrices, geometric primitives)
- system classes (file I/O, networking, logging)
- miscellaneous other stuff (plugins)

These classes implement a plethora of useful things, so you should just browse through the reference manual to figure out which ones suit your needs.

Some of the classes are of fundamental importance, so we will give a short overview of their basics in the following sections. However, the total number of classes is too large to cover here. Please have a look at the headers in `BALL/include/BALL/COMMON|CONCEPT|SYSTEM|MATHS|DATATYPE` and the BALL reference manual – it's worth the time!

4.1 The BALL File Class

All classes handling file I/O in BALL are derived from a common base class, `File`. This base class provides a lot of functionality that applies to all derived classes as well. One of the most useful features is on-the-fly file transformation. For example, we can open a file (*e.g.* a PDB file using the `PDBFile` class) that is not stored locally on a disc, but in the internet:

```
PDBFile
infile("http://www.ball-project.org/Downloads/Examples/pdb4pti.ent");
System S;
infile >> S;
infile.close();
```

This command retrieves the file `pdb4pti.ent` from its location at the BALL project site using the HTTP protocol and reads the contents of that file into a `System`. The retrieval and the expansion of the URL into something meaningful is performed by the classes `TransformationManager` and `TCPTransfer`. Any filename is first handed to the static instance of `TransformationManager` which `File` possesses, which then applies all matching rules from a predefined rule set to that file name. The resulting expanded file names are then checked for special prefixes by `File`:

- a file name starting with `exec:` will execute the command after the colon, redirect the output of the command to a temporary file, which is then opened and returned,
- a file name starting with `http:` or `ftp:` will initiate an HTTP or FTP transfer from the given URL to a temporary file, which is opened and returned.

The `exec:` prefix can be used to filter existing files through a command, *e.g.* the compressed (GZIPped) file `test.txt.gz` can be achieved through the filename `exec:gunzip -c test.txt.gz`, assuming that the `gunzip` executable is in your path. You can also automate this process using the `TransformationManager`. By simply defining a rule for all files ending in `.gz`, `gunzip` will be called automatically:

```
File::registerTransformation
(".", ".gz", "exec:/usr/local/bin/gunzip_-c_%s");
```

Similarly, if you store a local copy of the PDB in `/local/PDB`, you might want to define a short-hand for the path to the PDB through the following rule:

```
File::registerTransformation
("PDB:.*", "/local/PDB/structures/all/pdb/pdb%b.ent");
```

This rule would then expand the name `PDB://4pti` to the local copy in `/local/PDB/structures/all/pdb/pdb4pti.ent`. In the above rule, `4pti` corresponds to the basename of the file, *i.e.* the name without path and without

4. FOUNDATION CLASSES

the file type extension (everything after the last dot). For details on the formulation of rules, please refer to the section covering `TransformationManager` in the BALL Reference Manual.

Some restrictions apply when using transformed file names. First, they may be used for *reading* files only – there is no distinct rule set for writing files. Second, the transformation manager will always apply the first rule it finds. Therefore chaining of rules is not possible and the user is responsible for avoiding ambiguities between multiply defined rules.

4.2 BALL Strings

BALL provides a heavy-weight `String` class that has been designed to provide a wealth of functionality using a simple and consistent syntax. In general, you should avoid using `const char*` or STL string when using BALL, although they are compatible to each other. You can easily convert BALL strings to char pointers (using the `c_str()` method) and automatically convert char pointers to BALL strings.

This part of the tutorial will give a short introduction to the wealthy functionality of BALL strings. For complete information refer to the BALL Reference Manual [1].

4.2.1 String Operations

There are useful operations possible with BALL strings. Let us start with a very basic one, concatenation. The following code snippet will concatenate two BALL strings:

```
String A("Concat");
String B("enate");
String C = A + B;
```

But concatenation is also defined with STL strings and even standard C strings, *i.e.* `char*`, as operands:

```
string A("Concat");
char* B = "enate";
String C = A + B;
```

Another very useful operation is swapping two strings:

```
String A("Swap");
String B("swaP");
A.swap(B);
```

Something we might also use very often is reversing a string:

```
String A("Swap");
A.reverse();
```

And finally, it is even possible to substitute parts of a string with another `String` by using the `substitute` method:

```
String A("Please_replace_REPLACE_with_something_else.");
String B("REPLACE");
String C("SOMETHING_ELSE");
A.substitute(B, C);
```

4.2.2 Conversion

BALL strings are featured with many conversion mechanisms. Converting other types to a BALL string is done by using a constructor. Let us first construct a string from some basic C types:

4. FOUNDATION CLASSES

```
char c_char = 'B';
int c_int = 1;
float c_float = 2.99792458;

String A(c_char);
String B(c_int);
String C(c_float);
```

There are many other simple types supported, like unsigned int, double, etc. Refer to the reference manual for further information.

How do we make an int out of a BALL string? Or a char? That's equally easy. We only need to call the explicit conversion method. All those methods are named toType, where Type is the type you want to convert to. Have a look at the following example:

```
String i_wanna_be_an_int("4711");
String i_wanna_be_a_char("A");
String i_wanna_be_a_double("6.0221e23");

int i_am_an_int = i_wanna_be_an_int.toInt();
char i_am_a_char = i_wanna_be_a_char.toChar();
double i_am_a_double = i_wanna_be_a_double.toDouble();
```

4.2.3 Predicates

BALL strings provide many predicates that can be used for determining special properties. One can find out whether a String contains a certain substring, starts with a special prefix, ends with a suffix, consists only of letters or is simply a floating point number. The following code snippet will give you some idea of the power of the predicates.

```
String T("This_STRING_does_not_start_with_PREFIX");
cout << "String_is_empty?_" << T.isEmpty()
    << endl;
cout << "Has_prefix_" << T.hasPrefix("PREFIX")
    << endl;
cout << "Contains_" << T.hasSubstring("PREFIX")
    << endl;
cout << "Contains_only_letters?_" << T.isAlpha()
    << endl;
```

We will get the following output:

```
String is empty? 0
Has prefix "PREFIX"? 0
Contains substring "PREFIX"? 1
Contains only letters? 1
```

4.2.4 Comparing Strings

Commonly one often wants to compare strings, which is a pain with C type character fields. BALL strings provide you with a simple interface and rich functionality. Let's first have a look at equality tests. Note that you are not limited to BALL strings for those comparisons, but can use C strings as arguments to all those operations:

```
String test_string("Compare_me.");
String another_test_string("Blah.");
char* test_C_string = "No_match.";

cout << test_string.compare(another_test_string) << endl;
cout << test_string.compare(test_C_string) << endl;
cout << test_string == test_string << endl;
cout << test_string != "No,_this_is_not_equal." << endl;
```

You can also check whether a string is lexicographically less than another one:

```
cout << test_string < another_test_string << endl;
```

And finally it is even possible to limit the comparison to a certain area of a string by defining the start index and the length of the segment:

```
Index start_index = 9;
Size length = 2;
cout << test_string.compare("me", start_index, length) << endl;
```

4.2.5 Stream and Field Operations

Everyone familiar with measurement data processing encountered the problem of getting data fields out of lines containing several values of data. Quite often interpreter languages like awk or perl are used for such tasks. The disadvantage of using such tools obviously is that you cannot integrate such languages easily into a C++ program. The BALL development team was also frequently confronted with such problems. Resultingly, BALL strings provide methods to extract fields from strings.

```
String data = "1_2_3_4.567_8_blah";
cout << "Line_contains_" << data.countFields()
    << "_values" << endl;
cout << "The_data_at_index_5_is_" << data.getField(5) << endl;
```

The code above should print the number 8. Sometimes log files contain quoted data. You can even handle such lines by using the field functions for quoted entries:

```
String data = "1_2_3_4.567\"8_blah\"";
cout << "Line_contains_" << data.countFieldsQuoted()
    << "_values" << endl;
cout << "The_data_at_index_5_is_" << data.getFieldQuoted(5)
    << endl;
```

4. FOUNDATION CLASSES

The code above should print the number "8 blah".

Additionally BALL strings know how to get single lines from a stream. So, if you want to read and analyze a log file, open it, read the single lines and get the values you want:

```
istream is;  
Index index = 5;  
String line = getline(is)  
String value = line.getField(index)
```

5

Kernel

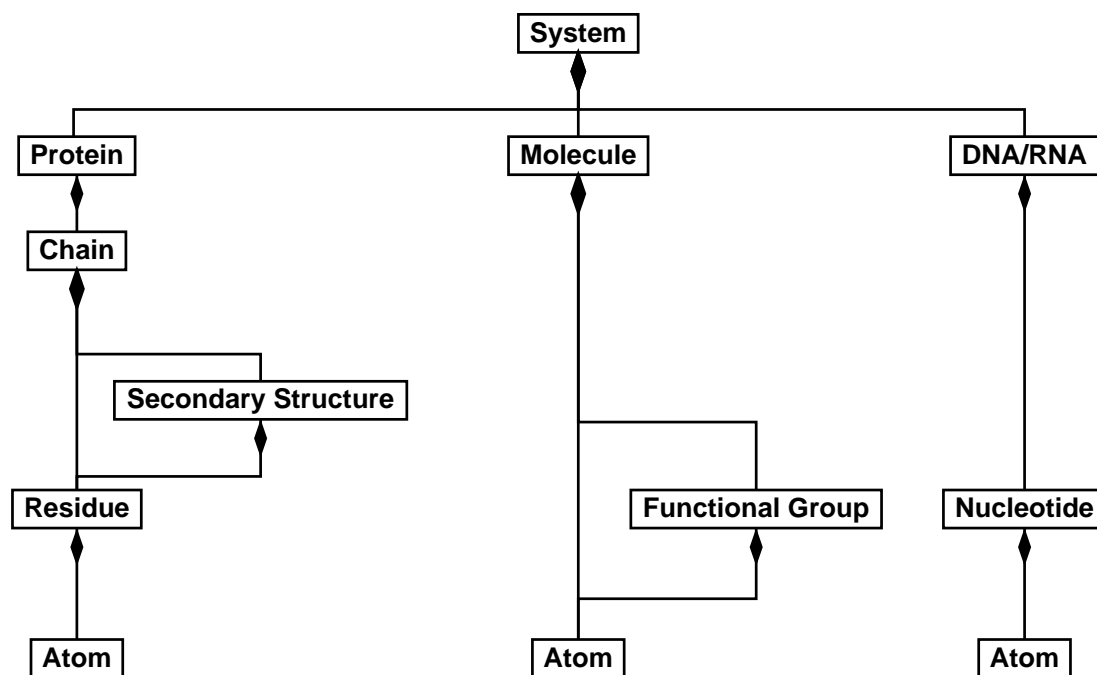


Figure 5.1: A model of the biochemical problem domain. BALL tries to model these entities as closely as possible with its kernel classes.

5.1 Kernel Classes

The BALL kernel data structures have been designed to model the problem domain (*i.e.* well-known biochemical entities) as closely as possible. Although some of the terms in biochemistry are rather fuzzy, there is a clear hierarchical relationship (Fig. 5.1). BALL tries to model this hierarchical relationship as a tree structure. The design pattern used to implement this tree is the *composite pattern* [8], which is implemented in the `Composite` class. Derived from `Composite` is the `AtomContainer` class, the base class of all classes handling atoms. Also the `Atom` class is derived from `Composite`. The typical user will probably use only derived classes of `AtomContainer`: `Atom` and `Molecule`. The kernel classes decompose into three frameworks: the general molecular framework, the protein framework, and the nucleic acid framework (Fig. 5.2). Each of these frameworks contains a few classes which try to model the respective problem domain as closely as possible. We will briefly discuss the roles of each of these classes (Sections 5.1.1 to 5.1.3) before describing some of the general features of the kernel classes.

By deriving all kernel classes from the common base class `Composite`, they share all the features implemented there. The following Section 5.2 will briefly discuss some of these features.

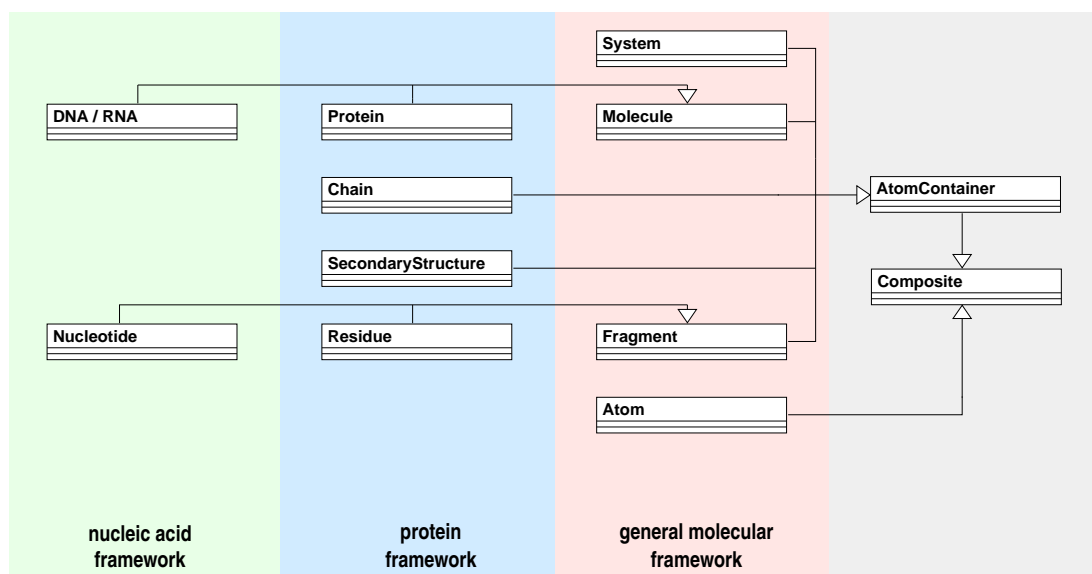


Figure 5.2: The *BALL* kernel classes consist of three main frameworks: the molecular framework, the protein framework, and the nucleic acid framework.

5.1.1 Molecular Framework

The molecular framework contains the classes `System`, `Molecule`, `Fragment`, `Bond`, and `Atom`. Molecules can contain an arbitrary number of atoms or fragments. A fragment can be used to define distinct groups in a molecule, *e.g.* functional groups or charge groups. Fragments can be nested, so you may want to define several functional groups within one larger fragment in a molecule. The atoms are then contained in the fragments. In contrast to other systems, the atoms of a molecule are not necessarily connected to each other (or in graph-theoretical terms: they do not have to represent a connected component of the graph formed by atoms and bonds). Systems are nothing but collections of molecules.

5.1.2 Protein Framework

The protein framework is a specialization of the general molecular framework and describes the structures encountered in proteins. Proteins are (more or less) molecules, so `Protein` is a subclass of `Molecule` and proteins can be handled like molecules and stored along with them in systems. Proteins can often be decomposed into several chains. These chains in turn can contain secondary structure elements, which then contain residues. Residues usually describe the amino acids of a protein. The sequence information of a protein is encoded implicitly in the order of the tree: it can be obtained by reading all instances of `Residue` in the order they are contained in a protein. So the

protein, the chains, and secondary structure elements should contain the residues in the correct order, from N-terminal to C-terminal, if you construct them by hand. The first and last residue of a chain are considered to be terminal (according to the member function `isTerminal`). This is also the reason why a protein should always contain at least one chain. Secondary structure elements, defined by `SecondaryStructure`, are optional. Wherever this information is known (*e.g.* when read from PDB files), instances of `SecondaryStructure` are created to store it. Each secondary structure element has a property (*e.g.* helix, β -sheet) describing its type.

5.1.3 Nucleic Acid Framework

The nucleic acid framework contains the classes `NucleicAcid` and `Nucleotide` and is used to represent structures of nucleic acids.

5.2 Kernel Iterators

Iteration over kernel data structures is a key concept in BALL. The BALL kernel iterators are STL-like iterators. Since most BALL kernel classes are so-called *multi-containers*, *i.e.* they can contain different objects, we cannot use the typical STL `begin()/end()` methods. For example, in a protein, you might not only want to iterate over all chains, but also over all residues or all atoms contained therein. BALL offers the methods `beginChain()/endChain()`, `beginAtom()/endAtom()`, and so on for all kernel classes. Similarly, the iterators are not `typedef`d within the class, but are independent classes, because an `AtomIterator` could be defined for any kernel class containing atoms. The only exception from that rule is `Atom::BondIterator`, since an atom is the only container having bonds.

We can use those iterators in an STL-like fashion

```
Molecule m = ...;
AtomIterator ai;
for (ai = m.beginAtom(); ai != m.endAtom(); ++ai)
{
    ...
}
```

but there is also a convenient shorthand: the operator `+` for all iterators. In contrast to STL iterators, BALL iterators are tightly bound to their containers and are thus aware of the container's end. The plus operator returns a boolean value determining the validity of the operator. It will return **false** as soon as the iterator reaches the end of the container. So we can rewrite the above code as:

```
Molecule m = ...;
AtomIterator ai(m.beginAtom());
for (; +ai; ++ai)
```

```
{  
    ...  
}
```

There exist several variants of the kernel iterators: the standard forward iterators (*e.g.* `AtomIterator`) and reverse iterators (`AtomReverseIterator`). For both, there are also `const` versions of these iterators, which are required when iterating over `const` instances of kernel classes (`AtomConstIterator`, `AtomConstReverseIterator`). Iteration is possible for all kernel classes, if they can contain the respective instances. So it is possible to iterate over all atoms in a molecule, but it is not possible to iterate over all chains of a residue, because `Residue` does not provide the `beginChain/endChain` methods.

Analogously to the STL, iterators may be dereferenced using both the star operator, and the arrow operator:

```
AtomIterator ai = molecule.beginAtom();  
cout << "first_atom:_" << (*ai).getName()  
    << "_@" << ai->getPosition() << std::endl;
```

6

VIEW Programming

Visualization not only covers the graphical representation of things (*e.g.* molecules, measurement data, ...) but also the graphical user interface (GUI). Both jobs can be done using `VIEW` functionality. Both areas are discussed in the next few sections.

6.1 Modularity

In general most of the time in GUI programming is spent on implementing the interactions of the individual elements of the user interface with each other (*e.g.* defining menu entries, button actions, etc.). In order to reduce these efforts, the functionality in the `VIEW` library has been bundled into different modules (*e.g.* OpenGL rendering, force field methods, the scripting language interface, etc.), which can be freely combined to an application. These modules automatically connect to each other and thus allow the user to add further functionality with as little as a single line of code. To this end, we have designed a set of base classes describing the interactions of the interface elements. The two most important components in this design are `MainControl`, the application's main window, and `ModularWidget`, the base class for all modules. The next pages will describe the modeling and implementation of this approach.

MainControl

The class `MainControl` is derived from Qt's `QMainWindow` and thus realizes an application's main window. It contains only the most essential data structures: The `CompositeManager` stores all molecular entities (`Composite` objects) and the `PrimitiveManager` is responsible for the representations (*i.e.* geometric models) and the thread for their (re)calculation. All additional functionality (*e.g.* reading and writing of structures, OpenGL visualization, etc.) are added to the main window by instantiating one of the classes derived from `ModularWidget`.

Modular Widgets

`ModularWidget` is a common base class for all the modules, that can be combined to form an entire application. While the modular widgets are widely independent, they can still notify each other about the current work flow. This is achieved by a messaging system, that allows a `ModularWidget` to send a message which is then received by all other modular widgets (see Section 6.2).

In addition to the messaging system, the `ModularWidget` class provides many other commonly needed features to ease and accelerate the development of new modules:

- Showing status and error messages
- Management of menu and toolbar entries

- Registering widgets and menu entries for the help system
- Management of preferences dialogs
- Reading/writing of settings from/to a configuration file
- Registering of supported file formats *e.g.* for parsing command line arguments or drag-and-drop support
- Access to individual instances with the method `getInstance()`
- Access to the `MainControl` and thus to the loaded molecules and representations
- Locking of molecular entities while multithreaded code is running

For many different tasks (see Table 6.1), the VIEW library already contains a wide variety of modular widgets. Since they are widely independent from each other, they can easily be combined to an application. All that is needed, is the instantiating of modular widgets with the `MainControl` as their parent, as can be seen in the following code snippet:

```
Mainframe::Mainframe(...)
: MainControl(...)
{
    new LogView(this);           // widget (1)
    new DatasetControl(this);    // widget (2)
    new PyWidget(this);          // widget (3)
    new MolecularControl(this);  // widget (4)
    new GeometricControl(this);  // widget (5)
    new Scene(this);             // widget (6)
}
```

These few lines of code (header includes were omitted for brevity) create a fully-fledged molecular structure viewer. In addition to using already existing widgets, users can easily implement new ones, especially since `ModularWidget` already offers many common features (see above). As a result, users can freely combine new and existing widgets both to extend `BALLView` with new functionality or to create entirely new custom-tailored applications.

Example for the implementation of a modular widget

To visualize how new modular widgets can be created, the following pages will outline the implementation of the class `LabelDialog`, which is part of the VIEW library. Its purpose is to create labels in the 3D view for a list of highlighted molecular entities and it has the following capabilities: The menu entry "Add Label" toggles the dialog's visibility and is disabled if no molecular entities are highlighted in the

6.1. MODULARITY

Name	Functionality
DatasetControl	Management of data sets
DisplayProperties	Creation and modification of models
DockingController	Molecular docking
DockWidget	Move and dock windows within the main window
DownloadPDBFile	Downloads from the protein database
EditableScene	Molecular editing
FDPBDialog	Calculation of electrostatic potentials
FileObserver	Observing changes in a molecular file
GeometricControl	Management of graphical representations
LabelDialog	Creation of labels in the 3D view
LogView	Logging window
ModifyRepresentationDialog	Custom colorings for models
MolecularControl	Hierarchical overview of loaded molecules
MolecularFileDialog	Reading and writing of molecular files
MolecularStructure	Force field and molecular mechanics features
PubChemDialog	Download a structure from PubChem
PyWidget	Python scripting
Scene	Three-dimensional graphics
ShortcutDialog	Editor for keyboard shortcuts
SnapshotVisualisation	Visualization of trajectories
TestFramework	Recording and playback of user input

Table 6.1: Overview of the classes derived from *ModularWidget*. Each individual class was created for one specific domain of features and is widely independent from the other widgets.

MolecularControl. When the dialog is shown, the user can select a font and its color, the desired text for the label(s), and choose if only one label is to be created for the entire selection or one label for every atom/residue. When the "Apply" button is pressed, a new *Representation* is created with the newly created label(s). For convenience the chosen color and font are stored in the application's configuration file for future usage.

The dialog's layout was done with the program "Qt Designer" (see Section 6.4). This results in a source file, containing the base class *Ui_LabelDialogData*, which defines the dialog's layout. The actual dialog class is derived from this layout class. This procedure accelerates the development process and makes the dialog's layout independent from its function. The following code is the content of the header file

6. VIEW PROGRAMMING

for the actual dialog. The includes, namespaces, and some documentation lines were omitted for brevity and the overloaded methods from the `ModularWidget` base class were marked.

```
class BALL_VIEW_EXPORT LabelDialog
: public QDialog,
  public Ui_LabelDialogData,
  public ModularWidget
{
    // macro needed for Qt's slot mechanism:
    Q_OBJECT

    // Macro from the Embeddable class:
    BALL_EMBEDDABLE(LabelDialog,ModularWidget)

public:
    LabelDialog(QWidget *parent = NULL, const char *name = NULL );
    virtual ~LabelDialog();

    // method for message handling, overloaded from ModularWidget
    virtual void onNotify(Message* message);

    // method for reading settings, overloaded from ModularWidget
    virtual void fetchPreferences(INIFile &inifile);
    // method for written settings, overloaded from ModularWidget
    virtual void writePreferences(INIFile &inifile);

    // method for e.g. initializing menu entries, overloaded
    // from ModularWidget
    virtual void initializeWidget(MainControl& main_control);

    // Overloaded from ModularWidget
    virtual void checkMenu(MainControl& main_control);

protected slots:
    virtual void accept();
    virtual void editColor();
    virtual void addTag();
    virtual void fontSelected();
    virtual void modeChanged();
    void textChanged();

protected:
    Representation* createOneLabel_();
    Representation* createMultipleLabels_();
    QAction* menu_entry_;
    ColorRGBA custom_color_;
    QFont font_;
};
```


The following paragraphs will discuss the actual implementation, starting with the constructor: It must be called with the `MainControl` as parent to enable the registration of the `ModularWidget`. Since this example class is derived from three other classes, these also have to be initialized. The function `setupUi` stems from the `”*.ui”` file and defines the layout of the dialog. Next the individual buttons and check boxes are connected to their slots. The call of `registerWidget` is of special importance since it enables the internal mechanism that connects all modular widgets with each other (*i.e.* the messaging system).

```
LabelDialog::LabelDialog(QWidget* parent, const char* name)
: QDialog(parent),
  Ui_LabelDialogData(),
  ModularWidget(name)
{
    // apply the dialogs layout:
    Ui_LabelDialogData::setupUi(this);

    // signals and slots connections
    connect(apply_button_, SIGNAL(clicked()),
            this, SLOT(accept()));
    connect(buttonCancel, SIGNAL(clicked()),
            this, SLOT(reject()));
    connect(edit_button, SIGNAL(clicked()),
            this, SLOT(editColor()));
    connect(add_tag_button, SIGNAL(clicked()),
            this, SLOT(addTag()));
    connect(font_button, SIGNAL(clicked()),
            this, SLOT(fontSelected()));
    connect(all_items, SIGNAL(toggled(bool)),
            this, SLOT(modeChanged()));
    connect(text_box, SIGNAL(editTextChanged(const QString&)),
            this, SLOT(textChanged()));

    setWindowTitle("Add_Label");
    setObjectName(name);
    hide();

    // register the widget with the MainControl
    ModularWidget::registerWidget(this);
}
```

All modular widgets can have their own menu entries in the applications menu bar. These entries should be initialized in the virtual method `initializeWidget`, which will be automatically called by the `MainControl` for all registered `ModularWidget` subclasses. For this concrete class, the method creates the menu entry `”Add Label”` and connects it to a slot, which will open the dialog. The creation of the menu entry is done through a call of `insertMenuEntry`, which will also create the given menu (if it does not yet exist) and return a pointer to a `QAction`.

6. VIEW PROGRAMMING

This pointer is then assigned to the variable `id_`, which will later be used for enabling and disabling the menu entry. Corresponding to the method `initializeWidget` exists the `initializePreferencesTab` method to add sub pages to the applications preferences dialog. Since this dialog does not have a configuration dialog, this function is not needed in the `LabelDialog` class.

```
void LabelDialog::initializeWidget(MainControl& main_control)
{
    menu_entry_ = ModularWidget::insertMenuEntry(
        MainControl::DISPLAY, "Add_Label", this,
        SLOT(show()));
    setMenuHint("Add_a_label_for_selected_molecular_objects");
}
```

The next method `onNotify` does the message processing: It is overloaded from the `ModularWidget` class and gets called when a `Message` arrives for the widget. In the `LabelDialog` class, it does the following: If no molecular entities are highlighted in the `MolecularControl`, the dialog's "Apply" button is disabled and the menu entries are checked if they have to be enabled/disabled. This is done to ensure, that the dialog can not be used while *e.g.* a simulation is running, since the molecular entities could changed at any time. This is checked in the `checkMenu` method with a call to `MainControl::isBusy()`, which returns true if a modular widget locked the molecular entities because they could be changed or if an model update is underway.

```
void LabelDialog::onNotify(Message* message)
{
    ControlSelectionMessage* sm =
        RTTI::castTo<ControlSelectionMessage>(*message);

    if (sm != 0)
    {
        // disable apply button, if selection is empty
        apply_button->setEnabled(!sm->getSelection().empty());
        checkMenu(*getMainControl());
    }
}

void LabelDialog::checkMenu(MainControl& mc)
{
    Size selection_size = mc.getMolecularControlSelection().size();
    menu_entry->setEnabled(selection_size && !mc.isBusy());
}
```

The following methods are only of minor interest and just shown for the sake of completeness. They are called when a user clicks on the corresponding buttons and perform auxiliary functions, like choosing the label's font or color.

```
void LabelDialog::editColor()
```

```
{
    custom_color_.set(chooseColor(color_sample_));
}

void LabelDialog::addTag()
{
    QString tag;
    if      (tag_box->currentText() == "Name")           tag = "%N";
    else if (tag_box->currentText() == "Residue_ID")      tag = "%I";
    else if (tag_box->currentText() == "Atom_Type")      tag = "%T";
    else if (tag_box->currentText() == "Atom_Charge")    tag = "%C";
    else if (tag_box->currentText() == "Atom_Type_Name") tag = "%Y";
    else if (tag_box->currentText() == "Element")        tag = "%E";

    text_box->lineEdit()->setText(text_box->currentText() + tag);
}

void LabelDialog::fontSelected()
{
    bool ok = true;
    QFont font = QFontDialog::getFont(&ok, font_, 0);
    if (!ok) return;

    font_label->setFont(font);
    font_ = font;
}

void LabelDialog::modeChanged()
{
    tag_box->setEnabled(!all_items->isChecked());
    add_tag_button->setEnabled(!all_items->isChecked());
}

void LabelDialog::textChanged()
{
    apply_button->setEnabled(text_box->currentText() != "");
}
```

When a user presses the dialog's "Apply" button, the method `accept()` is called. It does the actual label creation and adds the label(s) to the 3D view: First the current list of highlighted molecular entities is obtained from the `MainControl`. Then a `LabelModel` processor is created and added to a `Representation`. Next the values from the dialog are applied to the `LabelModel` and the list of molecular entities is stored in the `Representation`. This approach may seem a bit complex but it has a distinct advantage: The position of the label will be automatically updated if the 3D positions of the molecular entities should change. Finally the `Representation` is stored in the `MainControl` and it is rendered in the 3D view with a call of `MainControl::update(Representation&)`. The methods in the `MainControl`

6. VIEW PROGRAMMING

for inserting and updating representations were added just for convenience reasons and contain only the code for sending the corresponding messages. As an example, `MainControl::insert (Representation&)` only consists of the following code:

```
notify_(new RepresentationMessage(representation,
                                   RepresentationMessage::ADD));
```

After the last modular widget was notified, the message will be automatically deleted. Therefore messages that are about to be send, have to be created on the heap.

```
void LabelDialog::accept()
{
    List<Composite*> selection =
        getMainControl()->getMolecularControlSelection();

    // no selection present => return
    if (selection.empty()) return;

    Representation* rep = new Representation;
    rep->setProperty(Representation::PROPERTY__ALWAYS_FRONT);
    rep->setModelType(MODEL_LABEL);

    LabelModel* model = new LabelModel;
    model->setText(ascii(text_box->currentText()));
    model->setColor(custom_color_);
    model->setFont(font_);

    if (all_items->isChecked())
    { model->setMode(LabelModel::ONE_LABEL);
    }
    else if (every_atom->isChecked())
    { model->setMode(LabelModel::ALL_ATOMS);
    }
    else if (every_residue->isChecked())
    { model->setMode(LabelModel::ALL_RESIDUES);
    }
    else if (every_item->isChecked())
    { model->setMode(LabelModel::ALL_ITEMS);
    }

    rep->setModelProcessor(model);

    // process all objects in the selection list
    List<Composite*>::ConstIterator list_it = selection.begin();
    List<const Composite*> composites;
    for (; list_it != selection.end(); ++list_it)
    {
        composites.push_back(*list_it);
    }
}
```

```
rep->setComposites(composites);
getMainControl()->insert(*rep);
getMainControl()->update(*rep);
text_box->addItem(text_box->currentText());
setStatusbarText("Label_added.");
}
```

The next two methods are responsible for reading and writing the of this widget's settings. The first method, `fetchPreferences`, restores the settings from the configuration file. If the this file has a section with the name `WINDOWS` and a key `"Label::customcolor"` within, then the content of this key is read and converted to a color, which is then assigned to the dialog's label and stored in the variable `custom_color_`.

```
void LabelDialog::fetchPreferences(INIFile& inifile)
{
    // restore the color
    if (inifile.hasEntry("WINDOWS", "Label::customcolor"))
    {
        custom_color_.set(inifile.getValue("WINDOWS",
                                           "Label::customcolor"));
        setColor(color_sample_, custom_color_);
    }

    // restore the font
    if (inifile.hasEntry("WINDOWS", "Label::font"))
    {
        font_.fromString(inifile.getValue("WINDOWS",
                                           "Label::font").c_str());
    }
    font_label->setFont(font_);
}
```

To write all user defined options in a configuration file, the method `ModularWidget::writePreferences` was overridden:

```
void LabelDialog::writePreferences(INIFile& inifile)
{
    ModularWidget::writePreferences(inifile);

    // store the color
    inifile.insertValue("WINDOWS", "Label::customcolor",
                       custom_color_);

    // store the font
    inifile.insertValue("WINDOWS", "Label::font",
                       ascii(font_.toString()));
}
```

This concludes the implementation of the `LabelDialog` class. It can be used in a derived `MainControl` class, simply by adding the following line:

```
|| new LabelDialog(this, "LabelDialog");
```

This one line of code is all that is needed to make the dialog work, all needed function calls inside the `LabelDialog` class will be automatically done by the VIEW framework.

This example demonstrates, how effective the encapsulation of features into distinct modules works: Since the modules are mostly independent from each other and the interface base classes take care of all basic functions, it is very easy to extend the application with new features.

6.2 Messaging System

A special mechanism was needed for allowing the individual modular widgets to be mostly independent from each other but still be able work together *i.e.* by notifying each other about the current work flow. This was achieved by a messaging system, that allows a `ModularWidget` to send a message which is then received by all other modular widgets. As an example: The `MolecularControl` provides a hierarchical overview of the loaded molecules and notifies the other widgets when a user highlights some of its entries. This is needed, since other modular widgets offer tool bar entries for features (like saving molecules) that operate on the currently highlighted objects. These entries get disabled if no molecular item is highlighted. To notify the other widgets, the `MolecularControl` sends a `ControlSelectionMessage` which is then received by the `MolecularFileDialog`. This class provides the functionality for loading and writing molecular files and will then disable the corresponding menu and toolbar entries.

Since the modular widgets have to notify each other about many different events, the class `Message` has many subclasses (see Table 6.2), which store different data types, like selections or object pointers. If a new event is introduced, it can easily be added by creating an other `Message` subclass.

The actual sending of messages is done in the method `ModularWidget::notify_`, while the message is received by `ModularWidget::onNotify()` (for an example of a concrete implementation see Page 46). In this method the modular widget has to decide if it needs to react to the message. This is in general done by using runtime type identification. In addition, many different messages also provide enumeration values for defining further specialized message subtypes. A `CompositeMessage` can *e.g.* cope with the events of an added or deleted `Composite` by using the types `CompositeMessage::NEW_COMPOSITE` or `CompositeMessage::REMOVED_COMPOSITE`. The usage of these enumeration subtypes has the advantage, that much less subclasses are needed to distinguish different classes of events, resulting in a slimmer implementation. Thus we only added new `Message` subclasses, when a message had to transmit a new kind of data, or if

CompositeMessage	SceneMessage
GenericSelectionMessage	ControlSelectionMessage
NewSelectionMessage	GeometricObjectSelectionMessage
RepresentationMessage	MolecularTaskMessage
ShowDisplayPropertiesMessage	CreateRepresentationMessage
RegularDataMessage	RegularData1DMessage
RegularData2DMessage	RegularData3DMessage
NewDockResultMessage	NewTrajectoryMessage
ShowDockResultMessage	DockingFinishedMessage
DeselectControlsMessage	

Table 6.2: *Derived message classes. All the individual classes can contain specific datas and some have an addition enumeration type to differ between subtypes of messages.*

no appropriate class yet existed.

6.3 Design of the Visualization Classes

Geometric objects

We created a base class `GeometricObject` to provide a general interface for geometric shapes, that can be calculated and rendered in the VIEW framework. From this base class, we then derived the classes for the individual tangible geometric objects (see Table 6.3). The instances of these derived classes are created by the different model processors and later get colored by the color processors, and stored in a `Representation`. The renderer classes can then translate their information such that they can *e.g.* be drawn on the screen or processed by an external program.

Box	Disc
Label	GridVisualisation
Mesh	QuadMesh
Point	SimpleBox
Sphere	Tube
Line	TwoColoredLine
Tube	TwoColoredTube
IlluminatedLine	

Table 6.3: *Overview of the different geometric objects that are supported in the VIEW framework.*

We created numerous derived `GeometricObject` classes, to guarantee that virtually all thinkable shapes and objects can be visualized. If nevertheless in the fu-

ture the need for a new kind of geometric object would arise, this could be realized with minimal effort: All that would have to be done, is to create a new derived `GeometricObject` class and add the corresponding rendering methods to the `Renderer` classes (see Page 53).

Representations

To offer the user an intuitive way of handling models and their coloring, we created the class `Representation`. For each visualized object, this class stores the selection of molecular entities, the used model and coloring method, the drawing style, and the geometric objects representing the model (see Fig. 6.1). This approach has many advantages:

- The different models and coloring methods can be freely combined.
- Users can create customized representations *e.g.* by using the Python scripting interface.
- Users can combine as many different representations as they like to compose complex molecular visualizations.
- Individual representations can be enable/disable any time.
- It is possible to write project files, which store all representations for later usage.
- A `Representation` can easily be redrawn, when the corresponding atoms have changed.
- A user can disable all updates to a representation *e.g.* to visualize the differences between two steps in a trajectory.

We wanted to make the creation and modification of representations as easily as possible. Thus we designed a user friendly dialog, which can assign all the different settings of a `Representation`.

The classes `ModelProcessor` and `ColorProcessor` which are responsible for creating the different models and coloring schemes are described in the section below.

Models and Coloring

We created a wide variety of different models. All these model classes are derived from the `ModelProcessor`. This class is again derived from the `BALL class UnaryProcessor<Composite>`, which provides a general interface for recursively iterating and processing a `Composite` tree. Therefore a `ModelProcessor` can be applied to entire proteins as well as to individual atoms, which makes it is easily

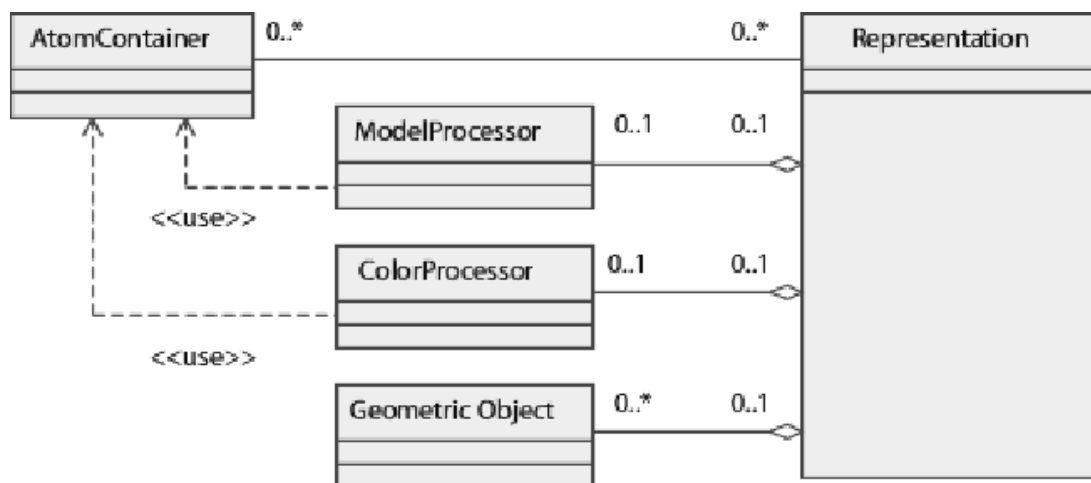


Figure 6.1: Each visualized object corresponds to an instance of the class *Representation*. The *ModelProcessor* creates *GeometricObjects*, e.g. tubes or meshes for all atoms stored in the *AtomContainers*. Next, the *ColorProcessor* colorizes the *GeometricObjects*, e.g. by element, charge or temperature factor. The individual model types and coloring methods are realized by derived classes. This approach simplifies the creation of new models and coloring methods and allows their free combination.

possible to create models for user defined subselections of molecules. When a model processor is applied to such a selection, it first iterates over the *Composite* tree and collects the information necessary for the model's creation. With this information, the method `createGeometricObjects()` can then create the individual geometric object, which form the model. The geometric objects are created on the heap and stored in a list inside the *Representation*, which is then responsible for deleting them.

The counterpart to *ModelProcessor* is *ColorProcessor*, which iterates over a list of *GeometricObjects* and finds the corresponding color for each one. Currently 16 different coloring methods are implemented and every single one is derived from *ColorProcessor*.

Since users directly perceive on how long it takes to create a representation, we made great efforts to ensure maximum performance for the model and coloring calculations.

Renderer

Currently *BALLView* supports two different renderers: Real time graphics are provided by the OpenGL renderer (*GLRenderer*) while high quality graphics are available through the *POVRay* exporter (*POVRenderer*). Both classes are derived from a common base class *Renderer*, which provides a general interface. This ensures

that in future versions, arbitrary renderer can easily be added by deriving a further class from `Renderer`. The actual rendering of a `Representation` is done in the method `Renderer::render(const Representation&)`. It iterates over all geometric objects in the `Representation` and, by using runtime type identification, finds the corresponding rendering method:

```
if (RTTI::isKindOf<Point>(*go))
{ renderPoint_(*const Point*) go);
}
else if (RTTI::isKindOf<Disc>(*go))
{ renderDisc_(*const Disc*) go);
}
else if (RTTI::isKindOf<Line>(*go))
{ renderLine_(*const Line*) go);
}
```

These methods are overridden in the derived classes with the real rendering code, *e.g.* OpenGL calls in the `GLRenderer`. This approach ensures that the different renderer can easily be extended with support for new kinds of geometric objects, simply by adding a new rendering method and an appropriate runtime check.

Unlike the OpenGL renderer, the `POVRenderer` does not provide real time graphics, but an interface to the external POVRay renderer. To do so, it translates the data in the geometric objects to a form that can be parsed by the POVRay application. The resulting text is then written to an output stream, which is either a file or the standard console output.

6.4 Creating Dialogs

To layout the dialogs in the VIEW library, we used the program "Qt Designer". It is part of every Qt-package and provides a comfortable "What you see is what you get" (WYSIWYG) interface for designing widgets. The result of the "Qt Designer" program is a ".ui" file, which is then transformed into a set of C++ source files by the Qt program "uic". These source files contain a base class, which defines the dialog's layout. The actual dialog class will be derived from the layout class and contains the dialog's actual functionality.

While this procedure may seem a bit complicated, it is actually straightforward and very useful: Not only does the "Qt Designer" WYSIWYG interface accelerate the development process, the resulting "*.ui" files also uncouple the dialogs' layout from its function. Thus a software engineer can extend the functionality without having to care about the dialog's layout, while a GUI designer can change the layout without the need to adapt the source code.

6.5 User Defined Settings

We wanted to give BALLView's users the opportunity to adapt it to their liking in any thinkable way, including the different models, coloring methods, and display options. Therefore an extensible graphical user interface was needed for applying these settings. For this purpose, we developed the `Preferences` dialog, which can contain an arbitrary number of child dialogs. These child dialogs are stored in a `QWidgetStack` and shown as entries in a hierarchical list. If a user clicks on such an entry, the corresponding dialog is then shown in the widget stack. This approach allows to cluster the different settings in a hierarchical way and users can freely browse and apply the individual settings. Furthermore the `Preferences` dialog can have any number of child dialogs and still have a concise layout. In an earlier implementation, we used a tab widget, which is the standard approach for such dialogs. This solution proved to be less suited, since the child dialogs can not be clustered and the layout becomes unhandy, if many children are added.

Automation of the (re)storing process

All the settings that can be adapted in the `Preferences` dialog, shall be stored when BALLView is closed. To this end all configuration dialogs have to store the content of their GUI elements, like line edits or check boxes. In the early versions of our implementation, every dialog provided its own routines for this purpose. Since this created a lot of overhead in means of redundant source code, we wanted to automate the (de)serialization: We designed a base class `PreferencesEntry`, which can act as a base class for any dialog. It automatically registers a dialog's GUI elements, whose content is then later saved or restored. All what is now needed, is to add one line of code in the the dialog's constructor:

```
|| registerWidgets_();
```

This sole line ensures, that the dialog's data get stored or read. Compared to the earlier implementation, which often had dozens of lines for this task, this is an essential improvement.

The storing process

To store the content of the registered GUI elements, their content is transformed into a string (see below) which is later written to the configuration file along with the name of the GUI element.

```
|| if (RTTI::isKindOf<QLabel>(*widget))
|| { value = getColor(dynamic_cast<const QLabel*>(widget));
|| }
|| else if (RTTI::isKindOf<QLineEdit>(*widget))
|| { value = ascii(dynamic_cast<const QLineEdit*>(widget)->text());
|| }
```

6. VIEW PROGRAMMING

```
else if (RTTI::isKindOf<QCheckBox>(*widget))
{ value = String((dynamic_cast<const QCheckBox*>(widget))->isChecked());
}
```

The resulting configuration file is line based and divided into sections, which can correspond to individual dialogs. The following lines illustrate the section for the dialog that configures and starts energy minimization runs. From these lines the whole content of the dialog can be reconstructed and thus the minimization settings restored.

```
[MINIMIZATION]
energy_difference_lineedit=0.0001
max_iterations_lineedit=100
refresh_iterations_lineedit=25
minimization_group=conjugate_button
max_grad_lineedit=1.000000
```

Further extensions

Since the described approach for storing the content of dialogs turned out to be very effective, we extended its usage. Now, dialogs no longer have to be child widgets in the `Preferences` dialog, to use this feature. In addition, the `PreferencesEntry` class now also supports the storing of default values, that are applied when a dialog's "Defaults" button is pressed. In just the same way a dialog gets restored to its originally values, when the "Cancel" button is pressed.

Another extension was made to support more sophisticated GUI elements: We created a base class `ExtendedPreferencesObject` that defines an interface for (re)storing the content of composite widgets, like *e.g.* the tables for the setup of the different coloring methods. This approach further improves the extensibility, since new, derived `ExtendedPreferencesObject` classes can be designed and added. What is even more important: Compared to a basic implementation, the described approach is also much less error-prone, since a developer can no longer accidentally forget to add the (re)storing code for one GUI element.

Summary

We designed a very user-friendly way to apply any arbitrary number of options. In addition, the implemented approach is also very handy for developers, since it is very extensible and minimizes the efforts for (re)storing the content of further dialogs.

The VIEW library in its current state has more than 20 dialogs, whose content is (re)stored. These dialogs have in total more than 200 widgets that contain user defined data. A conservative estimation of 8 lines of code per widget for the storing/restoring of its data, results in the saving of more than 1500 lines of code.

6.6 Multithreading

Molview, the precursor of BALLView was designed as a single-threaded application. As a result, the graphical user interface would freeze, while a calculation like a molecular dynamics simulation was running. Therefore users could no longer interfere with the application. This was especially tiresome since the calculations could not be aborted, except by shutting down the entire program. To circumvent these limitations, we had to redesign the VIEW library to use multithreading techniques. Now all long running calculations like MD simulations and energy minimizations are started in their own thread. This has several advantages:

- The user interface stays responsive at any time and may thus *e.g.* print estimated run times.
- Multithreaded calculations can be stopped with the ease of one mouse click.
- The 3D graphics widget can be used to show intermediate results *e.g.* while a minimization is running.
- Users can reposition the viewpoint *e.g.* to focus on one functional group while a MD simulation is running.
- Multiple threads can make efficient use of multi-processor or multi-core computers.

Since multithreading has so many advantages, we also used it for further purposes: The (re)calculation of models and colorings are now also started in separate threads. A further functionality were the multithreading technique became very handy is the dialog for downloading structures from the protein database. Here it allows to monitor the progress of the download and to abort it at any time.

Locking data structures and synchronization of threads

Unfortunately multithreading is one of the most complex fields in programming since the different threads have to be synchronized. The early multithreaded versions of our software had serious stability issues: It could *e.g.* happen that users modified or deleted molecular structures, which were used in multithreaded calculations like an MD simulation. This then resulted in immediate crashes. Other frequent problems were deadlocks, when two threads competed for access on the same data and race conditions, when two threads depended on each other.

To solve these and other problems, we redesigned the VIEW library such that it now uses strict mutex locking: Only one modular widget can get exclusive access to the molecular structures or representations. To do so, it has to lock the molecular entities (*i.e.* `Composite` objects) by calling the following function:

6. VIEW PROGRAMMING

```
|| bool ModularWidget::lockComposites()
```

If this call is successful, the modular widget can safely access and modify the `Composite` objects or start a thread for doing so. While the molecular entities are locked, further calls of `lockComposites` will fail and thus prevent any harmful changes. When the locking widget no longer needs access to the `Composite` objects, it must give up the lock with the following method:

```
|| bool ModularWidget::unlockComposites()
```

While the molecular entities are locked, the application has to notify the user that any changes to the structures are now forbidden: Beside showing a "busy" mouse cursor, all corresponding menu entries and widgets get disabled. This also provides direct feedback on which actions can still be performed. The disabling of potentially harmful GUI elements and keyboard shortcuts also acts as an additional protective barrier that prevents any adverse effects in the program's flow.

With this two interlocking mechanism for the prevention of any harmful changes, the multithreading approach runs stable and it became one of the central features in the VIEW library.

Information flow between threads

Unfortunately we had to consider several constraints in the design of the Qt library: For example only the main thread is qualified to modify GUI elements. Therefore we had to find a proper mean to transition data between any additional threads and the main thread, which is a prerequisite for many basic tasks like showing status messages. To this end, we decided to use the `QEvent` messaging system. It allows one thread to send an event that will then be received in the main thread. Since the adequate class for passing user defined data is `QCustomEvent`, we derived the class `MessageEvent` from it, which can contain any arbitrary VIEW message. As an example the thread that (re)calculates the model and coloring of a `Representation` notifies the main thread that it has finished with the following code:

```
|| sendMessage_(new RepresentationMessage(*rep,  
|| RepresentationMessage::FINISHED_UPDATE));
```

`sendMessage_` looks like the following:

```
|| void BALLThread::sendMessage_(Message* msg)  
|| {  
|| if (main_control_ == 0) return;  
|| // Qt will delete the MessageEvent when done  
|| qApp->postEvent(main_control_, new MessageEvent(msg));  
|| }
```

In the main thread, the `MainControl` then receives this event and acts accordingly:

```
|| bool MainControl::event(QEvent* e)  
|| {
```

```
if (e->type() == (QEvent::Type) MESSAGE_EVENT)
{
    Message* msg = dynamic_cast<MessageEvent*>(e)->getMessage();
    sendMessage(*msg);
    return true;
}

return QMainWindow::event(e);
}
```

As a result, all modular widgets will receive the VIEW message from the other thread.

6.7 How to Create a Geometric Primitive

In VIEW there are a number of predefined geometric primitives already available, *e.g. Sphere, Tube* etc. But sometimes a needed primitive may not be available and therefore must be programmed anew. In this section we want to create a new geometric primitive called 'Cross'. We define a cross to be a shape that consists of three lines that merge in one point. Additionally we require all lines to be axis aligned and meet each other in the middle.

To accomplish this we need three properties for the geometric object: the float member `radius` that describes the half length of each line, the class `Vertex` for the middle point of the geometric primitive and the class `ColorExtension` which contains methods for changing the color of the cross. In addition to these classes we need the main base class for creating a geometric primitive: The `GeometricObject` implements the interface each geometric shape must have.

The definition of `Cross` looks as follows:

```
class Cross:
    public Vertex,
    public GeometricObject
{
    public:

        Cross();

        virtual ~Cross();

        float getRadius() const;

        void setRadius(float new_radius);

    protected:

        float radius_;
};
```

6. VIEW PROGRAMMING

As this object is derived from all the base classes, we only need to implement a standard constructor, the destructor and the get- and set- methods for the radius.¹ All additional functionality is provided by inheritance.

We will now have a closer look at the implementation of the drawing method. To be able to draw the new geometric object class, we have to add the new method `renderCross_` to the classes `Renderer` and `GLRenderer`.

The method `Renderer::render_` defines which drawing methods are called for which geometric objects. We add the four new lines at the bottom, so it recognizes the new class `Cross`.

```
void Renderer::render_(const GeometricObject* object)
{
    if (RTTI::isKindOf<Sphere>(*object))
    { renderSphere_(*const Sphere*) object;
    }
    else if (RTTI::isKindOf<TwoColoredLine>(*object))
    { renderTwoColoredLine_(*const TwoColoredLine*) object;
    }
    else if (RTTI::isKindOf<Cross>(*object))
    { renderCross_(*const Cross*) object;
    }
    ...
}
```

The method `Renderer::renderCross_(const Cross& cross)` will be overloaded by derived `Renderer` classes, so it only contains a warning, which will appear if we forget to implement it in a derived `Renderer`:

```
virtual void renderCross_(const Cross& /* cross */)
{
    Log.error() << "renderCross_ not implemented in derived_"
                << "Renderer_class" << std::endl;
}
```

The method `GLRenderer::renderCross_(const Cross& cross)` does the actual rendering, so we use OpenGL code here:

```
void GLRenderer::renderCross_(const Cross& cross)
{
    glPushMatrix();

    // if cross is selected, use the selection color,
    // otherwise use its own color. (method from GLRenderer)
    setColor4ub_(cross);

    // move to the position of the cross (method from GLRenderer)
    translateVector3_(sphere.getVertex());
}
```

¹The copy constructor and the copy assignment methods have been omitted because they are not crucial to the implementation of a primitive.

6.7. HOW TO CREATE A GEOMETRIC PRIMITIVE

```
// OpenGL code for rendering the cross.
glBegin(GL_LINES);
glVertex3f((GLfloat)(getVertex().x - cross.getRadius()),
           (GLfloat)(getVertex().y),
           (GLfloat)(getVertex().z));
glVertex3f((GLfloat)(getVertex().x + cross.getRadius()),
           (GLfloat)(getVertex().y),
           (GLfloat)(getVertex().z));

glVertex3f((GLfloat)(getVertex().x),
           (GLfloat)(getVertex().y - cross.getRadius()),
           (GLfloat)(getVertex().z));
glVertex3f((GLfloat)(getVertex().x),
           (GLfloat)(getVertex().y + cross.getRadius()),
           (GLfloat)(getVertex().z));

glVertex3f((GLfloat)(getVertex().x),
           (GLfloat)(getVertex().y),
           (GLfloat)(getVertex().z - cross.getRadius()));
glVertex3f((GLfloat)(getVertex().x),
           (GLfloat)(getVertex().y),
           (GLfloat)(getVertex().z + cross.getRadius()));
glEnd();

glPopMatrix();
}
```

7

Python Extensions

7.1 Overview

Python is an object-oriented scripting language [25] that is well suited both as a language for embedding scripts into BALL applications and as a rapid prototyping language using the underlying BALL objects. BALL provides Python bindings for most of its classes in order to allow Rapid Methodology Development (RMD). Some of the functionality BALL provides (*e.g.* template classes, iterators) is unavailable due to the fundamental differences between the two languages. However, the majority of the classes is available and workarounds exist for some of the template- and iterator-related problems.

Since release 1.2 of BALL, the Python support is enabled by default. The remainder of this section assumes that you are somewhat familiar with the most important language concepts of Python.

BALL relies on SIP [24] version 4.6.0 to translate its class headers semi-automatically into Python wrapper classes. For each C++ class SIP creates a subclass defining the Python interface and a Python class using that C++ interface class. The Python class has the same name as the C++ class, so porting code from C++ to Python (and vice versa) gets trivial. The C++ code

```
System S;  
HINFile f("test.hin");  
f >> S;
```

translates to the Python code

```
S = System()  
f = HINFile("test.hin")  
f >> S;
```

In this example, the main difference is how C++ and Python handle constructors. Another important difference concerns iterators. The STL-like kernel iterators of BALL map to a set of functions (called extractors). An extractor traverses the whole container and creates a Python sequence object from it. Instead of having an `AtomIterator` iterating over all atoms of a residue

```
AtomIterator ai = residue.beginAtom();  
for (; +ai; ++ai)  
{  
    std::cout << ai->getName() << std::endl;  
}
```

an `atoms` extractor is used to create a sequence object containing all objects of the residue in Python:

```
for atom in atoms(residue):  
    print atom.getName()
```

7. PYTHON EXTENSIONS

For the template problem, we pre-instantiated some of the commonly used instances, *e.g.* `UnaryProcessor<Atom>` maps to the `AtomProcessor` class and classes derived from it in Python.

The `BALLView` application contains an interactive interpreter window if `BALL` was compiled with Python support. You can even access the data structures of the viewer from there. Assuming that you are currently displaying a structure in the viewer, you can retrieve a reference to the first system displayed through the somewhat cryptic command

```
|| system = MainControl::getInstance(0)\\  
|| .getCompositeManager().getComposites()[0].
```

Since this is not very convenient, we added a Python startup script that is always executed when `BALLView` starts up. It can be found under `BALL/data/startup.py`. By using one of the methods defined in this file, it is possible to obtain the first `System` by simply calling

```
|| system = getSystem(0).
```

This is very useful for extracting properties of loaded molecules and other datasets you are currently displaying, but not recommended if you start modifying internals of the viewer. You should also not try to destroy those objects in the viewer, or you will be rewarded with a core dump. Currently there is no further documentation of the Python support available.

7.2 Installation

Download, configure, and install the SIP version 4.6.0 or later from <http://www.ballview.org/Downloads/Contrib/> or from [24]. Run the configure script with the command `"python configure.py -x"` to disable the Qt support. It is important to compile `BALL` with the same C++ compiler SIP and Qt were compiled with. You can then enable the Python support of `BALL` by specifying the option `"--with-python=<path>"`, where `path` should point to the executable of an installed Python (version at least 2.5). You will also have to specify the path to the SIP executable, its headers, and its library (`libsip.a|so`). Adding these four options might look something like this on your system:

```
|| ./configure \  
|| --with-python=/opt/bin/python2.5\  
|| --with-sip=/opt/bin/sip\  
|| --with-sip-incl=/opt/include/sip\  
|| --with-sip-lib=/opt/lib
```

After configuring and building `BALL`, you should then change to `BALL/source/PYTHON/EXTENSIONS`. Here, you will have to run

7.2. INSTALLATION

```
|| make sip  
|| make depend  
|| make  
|| make install
```

in order to build the Python bindings, which are then installed to BALL/lib/<platform>. You can then execute the `pyballinit.py` script in BALL/source/PYTHON/EXTENSIONS to get an interactive Python shell with the BALL bindings or just start the Python interpreter and import the BALL bindings through

```
|| from BALL import *
```

Note, that you have to add the directory where the bindings are installed to Python's module search path (`sys.path`).

8

FAQ

An up-to-date and searchable version of this FAQ is available at our website:
<http://ball-trac.bioinf.uni-sb.de/wiki/FAQ>

Documentation

Question 1: *Are there Research reports available?*

Answer: Yes. Publications and research reports on BALL are listed on our website:
<http://www.ball-project.org>

Question 2: *Where do I get the latest version?*

Answer: The latest version of BALL, bug fixes, and updates are available from our website <http://www.ball-project.org>

Question 3: *Is there further documentation besides this document?*

Answer: Further documentation, a Wiki and a bug tracker, can be found on <http://ball-trac.bioinf.uni-sb.de/> providing a Code Library with Code snippets, further tutorials, Release Notes, and FAQs.

Question 4: *How do I use BALLView?*

Answer: Take a look at the built-in BALLView Tutorial or on the documentation at <http://www.ball-project.org/Documentation> or under BALL/doc/BALLView

Installation

Question 5: *Will BALL run on my hardware/with my compiler?*

Answer: BALL should run on all major current platforms, in particular Linux, MacOS X, and Windows. Older BALL versions have been regularly tested also on more exotic platforms, and these should also work; however, the BALL developers no longer have access to these platforms and cannot test new versions on them.

License

Question 6: *Under what kind of license is BALL available?*

Answer: BALL is mostly being distributed under the Lesser GNU Public License (LGPL). Parts of BALL (BALLView) are under the GNU Public License (GPL).

Acknowledgments

For proof-reading and contributing to this tutorial we want to thank Oliver Gärtner, Enrico Glaab, Christine Hedderich, Sophie Weggler, and Daniel Stöckel.

Index

`accept ()` (BALL class), 47

accessors

definition, 14

`add_hydrogens (processor)`, 23

`AddHydrogensProcessor`

(BALL class), 24

adding hydrogens, 23

AMBER force field, 24

`amber94.ini` (file), 24

`AmberFF` (BALL class), 24

`apply` (method), 16

`Atom`

accessors

`getCharge`, 14

`getElement`, 14

`getForce`, 14

`getName`, 14

`getPosition`, 14

`getRadius`, 14

`getType`, 14

`getTypeName`, 14

`getVelocity`, 14

`setCharge`, 14

`setElement`, 14

`setForce`, 14

`setName`, 14

`setPosition`, 14

`setRadius`, 14

`setType`, 14

`setName`, 14

`setVelocity`, 14

`Atom` (BALL class), 14, 18, 36, 37

`atom`, 14

`Atom::BondIterator` (BALL class), 38

`AtomConstIterator` (BALL class), 39

`AtomConstReverseIterator`

(BALL class), 39

`AtomContainer` (BALL class), 21, 36

`AtomContainers` (BALL class), 53

`AtomIterator` (BALL class), 38, 39, 63

`AtomProcessor` (BALL class), 64

`AtomReverseIterator` (BALL class),
39

`atoms` (BALL function), 63

`autobuild` (file), 7

BALL

classes

`accept ()`, 47

`AddHydrogensProcessor`, 24

`AmberFF`, 24

`Atom`, 14, 18, 36, 37

`Atom::BondIterator`, 38

`AtomConstIterator`, 39

`AtomConstReverseIterator`,
39

`AtomContainer`, 21, 36

`AtomContainers`, 53

`AtomIterator`, 38, 39, 63

`AtomProcessor`, 64

`AtomReverseIterator`, 39

`Bond`, 18, 37

`CanonicalMD`, 25

`checkMenu`, 46

`ColorExtension`, 59

`ColorProcessor`, 53

`ColorProcessor`, 52, 53

`Composite`, 36, 41, 50, 52, 53, 57, 58

`CompositeManager`, 41

`CompositeMessage::-`

`NEW_COMPOSITE`, 50

INDEX

CompositeMessage::-
 REMOVED_COMPOSITE, 50
CompositeMessage, 50
ConjugateGradient-
 Minimizer, 24
ControlSelectionMessage, 50
createGeometricObjects(),
 53
Cross, 59, 60
custom_color_, 49
DatasetControl, 43
DisplayProperties, 43
DockingController, 43
DockWidget, 43
DownloadPDBFile, 43
EditableScene, 43
Element, 14
Expression, 24
ExtendedPreferencesObject,
 56
FDPBDDialog, 43
fetchPreferences, 49
File, 29
FileObserver, 43
Fragment, 37
FragmentDB, 23, 24
GeometricObject, 52, 59
GeometricObjects, 53
GeometricControl, 43
GeometricObject, 51
GeometricObjects, 53
getInstance(), 42
GLRenderer, 53, 54, 60
HINFile, 17, 21
id_, 46
initializeWidget, 45
initializePreferencesTab,
 46
initializeWidget, 46
insertMenuEntry, 45
Label::customcolor, 49
LabelDialog, 42, 43, 46, 49, 50
LabelModel, 47
lockComposites, 58
LogView, 43
MainControl, 41, 42, 45, 47, 49, 58
MainControl::-
 insert(Representation&),
 48
MainControl::isBusy(), 46
MainControl::-
 update(Representation&),
 47
Message, 46, 50
MessageEvent, 58
MicroCanonicalMD, 25
ModelProcessor, 52, 53
ModelProcessor, 52, 53
ModifyRepresentation-
 Dialog, 43
ModularWidget, 41–46, 50
ModularWidget::notify_, 50
ModularWidget::onNotify(),
 50
ModularWidget::write-
 Preferences, 49
MolecularControl, 50
MolecularFileDialog, 50
MolecularControl, 43, 46
MolecularDynamics, 25
MolecularFileDialog, 43
MolecularStructure, 43
Molecule, 15, 18, 36, 37
NucleicAcid, 38
Nucleotide, 38
onNotify, 46
Options, 25
options, 26
PDBFile, 21, 29
PeriodicBoundary, 26
POVRenderer, 53, 54
Preferences, 55, 56
PreferencesEntry, 55
PreferencesEntry, 56
PrimitiveManager, 41
Protein, 37
PTE, 18
PubChemDialog, 43

INDEX

- PyWidget, 43
- QAction, 45
- QCustomEvent, 58
- QEvent, 58
- QMainWindow, 41
- QWidgetStack, 55
- registerWidget, 45
- Renderer, 52–54, 60
- Renderer::render(const Representation&), 54
- Representation, 43, 47, 51–54, 58
- Residue, 37, 39
- ResidueChecker, 23, 24
- ResidueIterator, 22
- Scene, 43
- SecondaryStructure, 38
- Selector, 24
- setupUi, 45
- ShortcutDialog, 43
- SnapshotVisualisation, 43
- String, 31, 32
- System, 16, 18, 29, 37, 64
- TCPTransfer, 29
- TestFramework, 43
- TransformationManager, 29, 30
- TranslationProcessor, 16, 18
- Ui_LabelDialogData, 43
- UnaryProcessor<Composite>, 52
- UnaryProcessor<Atom>, 64
- Vector3, 15, 16
- Vertex, 59
- VIEW, 41
- WINDOWS, 49
- Composite**
 - apply, 23
- EnergyMinimizer**
 - options, 25
- functions**
 - apply, 16
 - atoms, 63
 - countAtoms, 16, 21
 - countBonds, 16
 - createBond, 15
 - getLength, 16
 - getName(), 22
 - getPosition, 16
 - GLRenderer::render-Cross_(const Cross& cross), 60
 - insert, 15
 - minimize, 25
 - operator +, 38
 - renderCross_, 60
 - Renderer::render_, 60
 - Renderer::render-Cross_(const Cross& cross), 60
 - setElement, 14
 - setMaxGradient, 25
 - setPosition, 15, 16
 - setup, 24, 26
 - simulate, 26
 - substitute, 31
- namespaces**
 - BALL, 18
 - std, 18
- overview, 2**
- processors**
 - add_hydrogens, 23
 - build_bonds, 23
- Residue**
 - isTerminal, 38
- System**
 - beginAtom(), 38
 - beginChain(), 38
 - endAtom(), 38
 - endChain(), 38
- BALL (BALL namespace), 18
- BALL/data/Fragments (directory), 23
- BALL/data/startup.py (file), 64
- BALL/FORMAT (directory), 18
- BALL/include/BALL/config.h (file), 12
- BALL/include/BALL/MOLMEC (directory), 26
- BALL/source (directory), 7

INDEX

- BALL/source/BENCHMARKS (directory), 12
- BALL/source/TUTORIAL (directory), 18, 22, 26
- bash, 11
- basic components, 3
- Bond (BALL class), 18, 37
- bonds, 15
- build_bonds (processor), 23
- building bonds, 23
- CanonicalMD (BALL class), 25
- checking residues, 24
- checkMenu (BALL class), 46
- ColorExtension (BALL class), 59
- ColorProcessor (BALL class), 53
- ColorProcessor (BALL class), 52, 53
- common.mak (file), 7
- Composite (BALL class), 36, 41, 50, 52, 53, 57, 58
- apply (member of Composite), 23
- CompositeManager (BALL class), 41
- CompositeMessage::NEW_COMPOSITE (BALL class), 50
- CompositeMessage::-REMOVED_COMPOSITE (BALL class), 50
- CompositeMessage (BALL class), 50
- config.h (file), 7
- config.log (file), 12
- config.mak (file), 7
- configure
 - usage, 7
- ConjugateGradientMinimizer (BALL class), 24
- ControlSelectionMessage (BALL class), 50
- countAtoms (method), 16, 21
- countBonds (method), 16
- createGeometricObjects() (BALL class), 53
- createBond (method), 15
- Cross (BALL class), 59, 60
- csh, 11
- custom_color_ (BALL class), 49
- DatasetControl (BALL class), 43
- deep copying, 16
- design patterns, 2
- DisplayProperties (BALL class), 43
- DockingController (BALL class), 43
- DockWidget (BALL class), 43
- DownloadPDBFile (BALL class), 43
- EditableScene (BALL class), 43
- Element (BALL class), 14
- element, 14
- Energy minimization, 23
 - options (member of EnergyMinimizer), 25
- exec., 29
- Expression (BALL class), 24
- ExtendedPreferencesObject (BALL class), 56
- FDPBDdialog (BALL class), 43
- fetchPreferences (BALL class), 49
- FFTW, 7
- File (BALL class), 29
- FileObserver (BALL class), 43
- force field, 3
- foundation classes, 2
- Fragment (BALL class), 37
- fragment database, 24
- FragmentDB (BALL class), 23, 24
- ftp., 29
- GeometricObject (BALL class), 52, 59
- GeometricObjects (BALL class), 53
- GeometricControl (BALL class), 43
- GeometricObject (BALL class), 51
- GeometricObjects (BALL class), 53
- getCharge (accessor), 14
- getElement (accessor), 14
- getForce (accessor), 14
- getInstance() (BALL class), 42
- getLength (method), 16
- getName (accessor), 14
- getName() (method), 22

INDEX

- getPosition (accessor), 14
- getPosition (method), 16
- getRadius (accessor), 14
- getType (accessor), 14
- getTypeName (accessor), 14
- getVelocity (accessor), 14
- GLRenderer (BALL class), 53, 54, 60
- GLEW, 6
- GLRenderer::renderCross_(const Cross& cross) (method), 60
- HINFile (BALL class), 17, 21
- http:, 29
- id_ (BALL class), 46
- initializeWidget (BALL class), 45
- initializePreferencesTab (BALL class), 46
- initializeWidget (BALL class), 46
- insert (method), 15
- insertMenuEntry (BALL class), 45
- iostream (file), 18
- iterators, 21
- kernel, 3
- Label::customcolor (BALL class), 49
- LabelDialog (BALL class), 42, 43, 46, 49, 50
- LabelModel (BALL class), 47
- ld, 11
- LD_LIBRARY_PATH, 11
- libBALL.a (file), 10
- libBALL.so (file), 10
- libnsl, 9
- libsocket, 9
- libVIEW.a (file), 10
- libVIEW.so (file), 10
- libxnet, 9
- lockComposites (BALL class), 58
- LogView (BALL class), 43
- MainControl (BALL class), 41, 42, 45, 47, 49, 58
- MainControl::-
 - insert (Representation&) (BALL class), 48
- MainControl::isBusy() (BALL class), 46
- MainControl::-
 - update (Representation&) (BALL class), 47
- Makefile (file), 7
- Mesa, 6, 10
- Message (BALL class), 46, 50
- MessageEvent (BALL class), 58
- MicroCanonicalMD (BALL class), 25
- minimize (method), 25
- ModelProcessor (BALL class), 52, 53
- ModelProcessor (BALL class), 52, 53
- ModifyRepresentationDialog (BALL class), 43
- ModularWidget (BALL class), 41–46, 50
- ModularWidget::notify_ (BALL class), 50
- ModularWidget::onNotify() (BALL class), 50
- ModularWidget::write-Preferences (BALL class), 49
- Molecular Mechanics, 3
- MolecularControl (BALL class), 50
- MolecularFileDialog (BALL class), 50
- MolecularControl (BALL class), 43, 46
- MolecularDynamics (BALL class), 25
- MolecularFileDialog (BALL class), 43
- MolecularStructure (BALL class), 43
- Molecule (BALL class), 15, 18, 36, 37
- mutex locking, 57
- NucleicAcid (BALL class), 38
- Nucleotide (BALL class), 38
- onNotify (BALL class), 46
- operator + (method), 38
- operator *, 22, 39

INDEX

- operator <<, 17, 39
- operator >>, 21
- optimizing hydrogens, 23
- Options (BALL class), 25
- options (BALL class), 26

- PDB file, 21
- PDBFile (BALL class), 21, 29
- PeriodicBoundary (BALL class), 26
- persistence, 2
- Poisson-Boltzmann, 3
- POVRenderer (BALL class), 53, 54
- Preferences (BALL class), 55, 56
- PreferencesEntry (BALL class), 55
- PreferencesEntry (BALL class), 56
- PrimitiveManager (BALL class), 41
- processors, 16
- Protein (BALL class), 37
- PTE (BALL class), 18
- PubChemDialog (BALL class), 43
- Python, 6, 52, 63
- PyWidget (BALL class), 43

- QAction (BALL class), 45
- QCustomEvent (BALL class), 58
- QEvent (BALL class), 58
- QMainWindow (BALL class), 41
- Qt, 6
- QWidgetStack (BALL class), 55

- Rapid Methodology Development, 63
- Reference Manual, 2
- registerWidget (BALL class), 45
- renderCross_ (method), 60
- Renderer (BALL class), 52–54, 60
- Renderer::render (const Representation&) (BALL class), 54
- Renderer::render_ (method), 60
- Renderer::renderCross_ (const Cross& cross) (method), 60
- Representation (BALL class), 43, 47, 51–54, 58
- Residue (BALL class), 37, 39
- isTerminal (member of Residue), 38
- ResidueChecker (BALL class), 23, 24
- ResidueIterator (BALL class), 22

- Scene (BALL class), 43
- SecondaryStructure (BALL class), 38
- selecting atoms, 24
- Selector (BALL class), 24
- setCharge (accessor), 14
- setElement (accessor), 14
- setElement (method), 14
- setForce (accessor), 14
- setMaxGradient (method), 25
- setName (accessor), 14
- setPosition (accessor), 14
- setPosition (method), 15, 16
- setRadius (accessor), 14
- setType (accessor), 14
- setName (accessor), 14
- setup (method), 24, 26
- setupUi (BALL class), 45
- setVelocity (accessor), 14
- sh, 11
- ShortcutDialog (BALL class), 43
- simulate (method), 26
- SIP, 7, 63, 64
- SnapshotVisualisation (BALL class), 43
- solvation methods, 3
- Standard Template Library, 21
- std (BALL namespace), 18
- STL, 21
- String (BALL class), 31, 32
- substitute (method), 31
- System (BALL class), 16, 18, 29, 37, 64
- beginAtom() (member of System), 38
- beginChain() (member of System), 38
- endAtom() (member of System), 38
- endChain() (member of System), 38

- TCPTransfer (BALL class), 29
- tcsh, 11
- test programs, 12
- TestFramework (BALL class), 43
- testing, 12

INDEX

TransformationManager
 (BALL class), 29, 30
TranslationProcessor (BALL class),
 16, 18

Ui_LabelDialogData (BALL class), 43
UnaryProcessor<Composite>
 (BALL class), 52
UnaryProcessor<Atom> (BALL class),
 64

Vector3 (BALL class), 15, 16
Vertex (BALL class), 59
VIEW, 3
VIEW (BALL class), 41
visualization, 3

WINDOWS (BALL class), 49

References

- [1] BALL Reference Manual. <http://www.ball-project.org/Documentation>, 2009.
- [2] H. M. Berman, J. Westbrook, Z. Feng, G. Gilliland, T. N. Bhat, H. Weissig, I. N. Shindyalov, and P. E. Bourne. The Protein Data Bank. *Nucleic Acids Res.*, 28(1):235–242, 2000.
- [3] N. P. Boghossian, O. Kohlbacher, and H.-P. Lenhof. BALL: Biochemical Algorithms Library. Research report, Max-Planck-Institut für Informatik, Saarbrücken, 1999.
- [4] N. P. Boghossian, O. Kohlbacher, and H.-P. Lenhof. BALL: Biochemical Algorithms Library. In J. S. Vitter and C. D. Zaroliagis, editors, *Algorithm Engineering, 3rd International Workshop, WAE'99, Proceedings*, volume 1668 of *Lecture Notes in Computer Science (LNCS)*, pages 330–344. Springer, 1999.
- [5] B. R. Brooks, R. E. Bruccoleri, B. D. Olafson, D. J. States, S. Swaminathan, and M. Karplus. CHARMM: A program for macromolecular energy, minimization, and dynamics calculations. *J. Comput. Chem.*, 4(2):187–217, 1983.
- [6] W. D. Cornell, P. Cieplak, C. I. Bayly, I. R. Gould, K. M. Merz Jr., D. M. Ferguson, D. C. Spellmeyer, T. Fox, J. W. Caldwell, and P. A. Kollman. A second generation force field for the simulation of proteins, nucleic acids and organic molecules. *J. Am. Chem. Soc.*, 117(19):5179–5197, 1995.
- [7] M. Frigo and S. G. Johnson. FFTW: The Fastest Fourier Transform in the West, version 3.1.2. <http://www.fftw.org>, 2006.
- [8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley, Reading, MA, 1995.
- [9] T. A. Halgren. Merck molecular force field. I. Basis, form, scope, parameterization, and performance of MMFF94. *J. Comput. Chem.*, 17(5–6):490–519, 1996.

REFERENCES

- [10] T. A. Halgren. Merck molecular force field. II. MMFF94 van der Waals and electrostatic parameters for intermolecular interactions. *J. Comput. Chem.*, 17(5–6):520–552, 1996.
- [11] T. A. Halgren. Merck molecular force field. III. Molecular geometries and vibrational frequencies for MMFF94. *J. Comput. Chem.*, 17(5–6):553–586, 1996.
- [12] T. A. Halgren. Merck molecular force field. V. Extension of MMFF94 using experimental data, additional computational data, and empirical rules. *J. Comput. Chem.*, 17(5–6):616–641, 1996.
- [13] T. A. Halgren and R. B. Nachbar. Merck molecular force field. IV. Conformational energies and geometries for MMFF94. *J. Comput. Chem.*, 17(5–6):587–615, 1996.
- [14] HyperChem release 4.5. Hypercube Inc., 1995.
- [15] O. Kohlbacher. BALL – A Framework for Rapid Application Development in Molecular Modeling. In *Beiträge zum Heinz-Billing-Preis 2000*, number 56 in Forschung und wissenschaftliches Rechnen - GWDG Berichte, pages 13–30. Gesellschaft für wissenschaftliche Datenverarbeitung mbH, Göttingen, 2001.
- [16] O. Kohlbacher and H.-P. Lenhof. Rapid software prototyping in computational molecular biology. In *Proceedings of the German Conference on Bioinformatics (GCB'99)*, 1999.
- [17] O. Kohlbacher and H.-P. Lenhof. BALL – rapid software prototyping in computational molecular biology. *Bioinformatics*, 16(9):815–824, 2000.
- [18] The Mesa 3D graphics library. <http://www.mesa3d.org/>.
- [19] A. Moll, A. Hildebrandt, A. Lenhof, and O. Kohlbacher. BALLView: An object-oriented molecular visualization and modeling framework. *J. Comput.-Aided Mol. Des.*, 19(11):791–800, 2005.
- [20] A. Moll, A. Hildebrandt, A. Lenhof, and O. Kohlbacher. BALLView: A tool for research and education in molecular modelling. *Bioinformatics*, 22(3):365–366, 2006.
- [21] OpenGL API documentation overview. <http://www.opengl.org/documentation>.
- [22] Qt release 4.5. Troll Tech ASA. <http://qt.nokia.com/products/>.
- [23] The Boost developers. Boost c++ libraries, version 1.35. <http://www.boost.org/>, 2008.

REFERENCES

- [24] P. Thompson. SIP: A tool for generating python bindings for C and C++ libraries, version 4.7.9. <http://www.riverbankcomputing.co.uk/software/sip/>, 2009.
- [25] G. van Rossum. Python, version 2.6. <http://www.python.org>, 2009.
- [26] C. Zhang, G. Vasmatzis, J. L. Cornette, and C. DeLisi. Determination of atomic desolvation energies from the structures of crystallized proteins. *J. Mol. Biol.*, 267(3):707–726, 1997.