

BALLView

A molecular viewer and modeling tool

Dissertation zur Erlangung des Grades des Doktors der
Ingenieurwissenschaften der Naturwissenschaftlich-
Technischen Fakultäten der Universität des Saarlandes

vorgelegt von

Diplom-Biologe Andreas Moll

Saarbrücken im Mai 2007

Tag des Kolloquiums: 18. Juli 2007

Dekan:

Prof. Dr. Thorsten Herfet

Mitglieder des Prüfungsausschusses:

Prof. Dr. Philipp Slusallek

Prof. Dr. Hans-Peter Lenhof

Prof. Dr. Oliver Kohlbacher

Dr. Dirk Neumann

Acknowledgments

The work on this thesis was carried out during the years 2002-2007 at the Center for Bioinformatics in the group of Prof. Dr. Hans-Peter Lenhof who also was the supervisor of the thesis. With his deeply interesting lecture on bioinformatics, Prof. Dr. Hans-Peter Lenhof kindled my interest in this field and gave me the freedom to do research in those areas that fascinated me most.

The implementation of BALLView would have been unthinkable without the help of all the people who contributed code and ideas. In particular, I want to thank Prof. Dr. Oliver Kohlbacher for his splendid work on the BALL library, on which this thesis is based on. Furthermore, Prof. Kohlbacher had at any time an open ear for my questions. Next I want to thank Dr. Andreas Hildebrandt, who had good advices for the majority of problems that I was confronted with. In addition he contributed code for database access, field line calculations, spline points calculations, 2D depiction of molecules, and for the docking interface. Heiko Klein wrote the precursor of the VIEW library. Anne Dehof implemented the peptide builder, the secondary structure assignment, and a first version of the editing mode. Bettina Leonhardt and Carla Haid implemented the docking interface and its graphical frontend. Andreas Bertsch developed the SMARTS matcher and the ring perception algorithms that are the basis for the MMFF94 atom typing process. Stefan Strobel wrote the molecular surface calculation code. Jan Küntzer contributed code for the PDB and PubChem access. Andreas Krauser developed the first version of the BALLView installer for Microsoft Windows. Dr. Dirk Neumann and Dr. Andreas Kämper had the patience to explain some of the finer details on organic chemistry to me. Alexander Rurainski gave me advice on some mathematical and numerical problems while Sophie Weggler discussed the details of the MMFF94 forces implementation with me. Gernot Ziegler kindly provided hints on OpenGL graphics and access to the MPI multimedia room. Furthermore, my thanks go to all other member of the chair for bioinformatics who made this time in my life so enjoyable.

Last, but certainly not least, I want to thank my wife Petja Moll and my parents for their understanding and support.

Abstract

Over the last ten years, many molecular modeling software were developed, but most of them offer only limited capabilities or are rather difficult to use. This motivated us to create our own molecular viewer and modeling tool BALLView, based on our biochemical algorithms library BALL. Through its flexible and intuitive interface, BALLView provides a wide range of features in fields of electrostatic potentials, molecular mechanics, and molecular editing. In addition, BALLView is also a powerful molecular viewer with state-of-the-art graphics: it provides a variety of different models for biomolecular visualization, e.g. ball-and-stick models, molecular surfaces, or ribbon models. Since BALLView features a very intuitive graphical user interface, even inexperienced users have direct access to the full functionality. This makes BALLView particularly useful for teaching. For more advanced users, BALLView is extensible in different ways. First, extension on the level of C++ code is very convenient, since the the underlying code was designed as a modular development framework. Second, an interface to the scripting language Python allows the interactive rapid prototyping of new methods. BALLView is portable and runs on all major platforms (Windows, MacOS X, Linux, most Unix flavors). It is available free of charge under the GNU Public License (GPL) from our website (www.ballview.org).

German abstract

Im Laufe der letzten zehn Jahre wurden viele verschiedene Molecular Modeling Programme geschrieben, aber die meisten bieten nur eingeschränkte Funktionalität, oder sind sehr unintuiv zu bedienen. Dies impliziert, dass viele Forscher Probleme mit diesen Programmen haben und benutzerfreundlichere Software vorziehen würden. Dies inspirierte uns dazu, mit BALLView ein neuartiges Modellierungsprogramm zu entwickeln, basierend auf unserer biochemischen Algorithmenbibliothek BALL. Durch seine flexible Oberfläche bietet BALLView eine reiche Palette an Funktionen in den Bereichen Elektrostatik, Molekularmechanik und dem Editieren von Molekülen an. Darüberhinaus ist BALLView auch ein leistungsfähiges Programm zur Visualisierung von Molekülen, das über Grafikfähigkeiten verfügt, die dem neuesten Stand der Technik entsprechen. BALLView unterstützt neben allen Standard-Molekülmodellen wie bspw. Stick, Cartoon, Ribbon und Oberflächen auch die Visualisierung von elektrostatischen Feldern. Alle aufgeführten Funktionen können auch von unerfahrenen Benutzern verwendet werden, da BALLView eine sehr intuitive Benutzeroberfläche besitzt. Dadurch ist es hervorragend geeignet zum Einsatz in der Lehre. Für fortgeschrittene Benutzer ist BALLView erweiterbar auf zwei unterschiedlichen Wegen: Durch das Design der zugrundeliegenden Klassenhierarchie sind Erweiterungen auf der Ebene des C++ Programmcodes sehr einfach zu realisieren. Desweiteren bietet BALLView ein Interface zur Skriptsprache Python, die interaktives Rapid-Prototyping von neuen Funktionen erlaubt. BALLView ist portierbar und kann auf allen verbreiteten Plattformen (Windows, MacOS X, Linux, die meisten Unix-Derivate) verwendet werden. Es ist frei verfügbar unter der LGPL Lizenz und kann von unserer Webseite heruntergeladen werden (www.ballview.org).

German summary

Im Laufe der letzten zehn Jahre wurden die Methoden zur Aufklärung der dreidimensionalen Struktur komplexer organischer Substanzen wie bspw. Proteinen und Nukleinsäuren ständig weiterentwickelt. Als Folge dessen nimmt die Anzahl der neu publizierten Strukturen von Jahr zu Jahr zu. Die vielleicht grösste Herausforderung unserer Zeit ist es nun, diese Daten in nutzbringendes Wissen zu verwandeln: Ein tieferes Verständnis des Verhaltens und Zusammenspiels der einzelnen Biomoleküle wird zu bahnbrechenden Entwicklungen in der biologischen und biochemischen Grundlagenforschung führen. Darüberhinaus wird dieses Verständnis die Entwicklung von neuartigen Medikamenten ermöglichen, die gleichzeitig wirksamer und spezifischer (geringere Nebenwirkungen) sein werden. Die Schlüsseltechnologie für die Aufklärung der zugrundeliegenden Mechanismen ist das sogenannte "Molecular Modeling". Es kann definiert werden, als die theoretischen Methoden und Berechnungstechniken, die dazu dienen, das Verhalten von Moleküle zu modellieren, simulieren und visualisieren. In den letzten Jahren wurden viele verschiedene Modellierungsprogramme entwickelt, aber die meisten bieten nur eingeschränkte Funktionalität, oder sind schwer zu verstehen und zu benutzen. Dies impliziert, dass viele Forscher und Studenten Probleme mit diesen Programmen haben und leichter zu bedienende Software vorziehen würden. Das inspirierte uns dazu, mit BALLView ein neuartiges Modellierungsprogramm zu entwickeln, basierend auf unserer biochemischen Algorithmenbibliothek BALL. Durch seine flexible Oberfläche bietet BALLView eine reiche Palette an Funktionen in den Bereichen Elektrostatik, Molekularmechanik sowie dem Editieren von Molekülen an. Darüberhinaus ist BALLView auch ein leistungsfähiges Programm zur Visualisierung von Molekülen, das über Grafikfähigkeiten verfügt, die dem neuesten Stand der Technik entsprechen. So unterstützt BALLView neben allen Standard-Molekülmodellen wie bspw. Stick, Cartoon, Ribbon und Oberflächen auch die Visualisierung von elektrostatischen Feldern. Desweiteren erlaubt es die Erstellung von hochqualitativen Abbildungen in beliebigen Auflösungen, durch den Export zum Raytracer POV-Ray. Benutzer können zusätzlich auf sehr einfache Weise Videodateien erstellen oder 3D-Stereo-Projektionen verwenden.

Alle aufgeführten Funktionen können auch von unerfahrenen Benutzern verwendet werden, da BALLView eine sehr intuitive Benutzeroberfläche besitzt. Um den Benutzer weitere Hilfestellung zu reichen, bietet sie eine integrierte Dokumentation und kontextsensitive Hilfsfunktionen an. Dadurch eignet sich BALLView auch hervorragend zum Einsatz in der Lehre, insbesondere zur Vermittlung biochemischer Grundlagen und bioinformatischer Methoden. Für fortgeschrittene Benutzer ist BALLView auf zwei unterschiedlichen Wegen erweiterbar:

- Durch das objektorientierte Design der zugrundeliegenden Klassenhierarchie lassen sich Erweiterungen auf der Ebene des des C++ Programmcodes sehr leicht

realisieren. Desweiteren wurde die zugrundeliegende Funktionalität in verschiedene, unabhängige Module zerlegt, die beliebig kombiniert und ergänzt werden können. Dadurch lassen sich sehr einfach benutzerdefinierte Programme schreiben, die genau die gewünschte Funktionalität besitzen.

- Zusätzlich bietet BALLView ein Interface zur Skriptsprache Python. Dazu wurde die C++ Klassenbibliothek in ein Python-Modul umgewandelt. Die darin enthaltenen Python-Klassen sind dabei nahezu identisch zu den C++ Klassen, was die notwendige Einarbeitungszeit minimiert. Als Resultat erlaubt die Pythonschnittstelle den schnellen Zugriff auf alle geladenen Daten in Realzeit. Ausserdem ermöglicht das Python-Interface interaktives Rapid-Prototyping von neuen Funktionen. Da Python als Skriptsprache nicht kompiliert werden muss, kann sein Einsatz die Entwicklungsphase deutlich beschleunigen. Um das Arbeiten mit dem Python-Interpreter zu vereinfachen, ist BALLView auch eine Integrierte Entwicklungsumgebung (IDE): Der Editor für Pythonskripte bietet unter anderem Syntaxhervorhebung, Autovervollständigung sowie kontextsensitive Hilfe zum Klasseninterface. Daher können auch Einsteiger sehr einfach eigene Skripte schreiben.

Da BALLView so einfach zu erweitern ist, eignet es sich hervorragend als Grundlage für Studentenprojekte im Bereich der strukturellen Bioinformatik. Hier erlaubt es den Studenten sich auf die Kernproblematik ihrer Projekte zu konzentrieren anstatt bspw. Dateiformate zu implementieren. Dadurch werden auch Projekte ermöglicht, die bisher auf Grund der benötigten Basisfunktionalität nicht realisiert werden konnten. Dies hat bereits zu einer interessanten Lehre geführt. Da BALLView frei verfügbar ist (unter der LGPL Lizenz) kann es auch von anderen Forschungsgruppen verwendet werden um ihre eigenen Softwareprojekte darauf aufzubauen. Um dies weiter zu vereinfachen, wurde BALLView sehr portierbar implementiert, so dass es auf allen verbreiteten Plattformen (Windows, MacOS X, Linux, die meisten Unix-Derivate) verwendet werden kann. Um die Installation zu erleichtern, bieten wir automatische Installer bzw. Binärpakete für die verschiedenen Plattformen an. Die entsprechenden Dateien können von der Webseite des Projektes heruntergeladen werden (www.ballview.org).

Contents

1. Introduction	3
1.1. Current state-of-the-art	5
1.2. Aims of this work	7
2. Design and implementation	11
2.1. Overview	11
2.2. Design goals	12
2.3. Modularity	16
2.3.1. MainControl	17
2.3.2. Modeling of the modular widgets	17
2.3.3. Messaging system	21
2.4. Extensibility	21
2.4.1. Support for arbitrary data sets	23
2.4.2. Creation of dialogs	25
2.4.3. Configurability	25
2.5. Class design for the visualization features	29
2.5.1. Geometric objects	29
2.5.2. Representations	30
2.5.3. Molecular models and their colorings	30
2.5.4. Renderer	31
2.6. Performance tuning	32
2.6.1. Visualization	33
2.6.2. The force field calculations	34
2.6.3. Multithreading	35
2.6.4. Tuning the OpenGL rendering	40
2.7. Quality assurance	42
2.7.1. Verification	42
2.7.2. GUI testing	43
2.7.3. Usability testing	44
2.8. Comparison with other visualization and modeling frameworks	45

3. Features and applications	47
3.1. Graphical user interface	47
3.1.1. Architecture	47
3.1.2. Usability	49
3.1.3. Documentation	52
3.2. Visualization functionality	55
3.2.1. Representations	56
3.2.2. Molecular models and colorings	57
3.2.3. Visualization of electrostatic potentials	59
3.2.4. OpenGL graphics	64
3.2.5. Creation of images and movies	69
3.2.6. Comparison with related software	73
3.3. Molecular modeling functionality	74
3.3.1. Basic modeling features	74
3.3.2. Molecular Mechanics	77
3.3.3. Molecular editing	89
3.3.4. Docking	92
3.3.5. Electrostatics calculation	93
3.3.6. Comparison with related software	94
3.4. Python interface	97
4. Conclusion and discussion	105
A. MMFF94 forces	111
Bibliography	113
Figures	119

1. Introduction

"If the 20th century was the century of physics, the 21st century will be the century of biology." [100]

Over the last ten years the experimental methods for resolving three-dimensional structures of complex organic compounds like proteins and nucleic acids have been steadily improved. This resulted in ever increasing numbers of newly published molecular structures per year (see Fig. 1.1). Along with the number of structures the complexity of the newly resolved structures increased significantly: The protein database [49] now contains large macromolecular machines with up to 100,000 atoms, like entire proteasomes or ribosomal subunits.

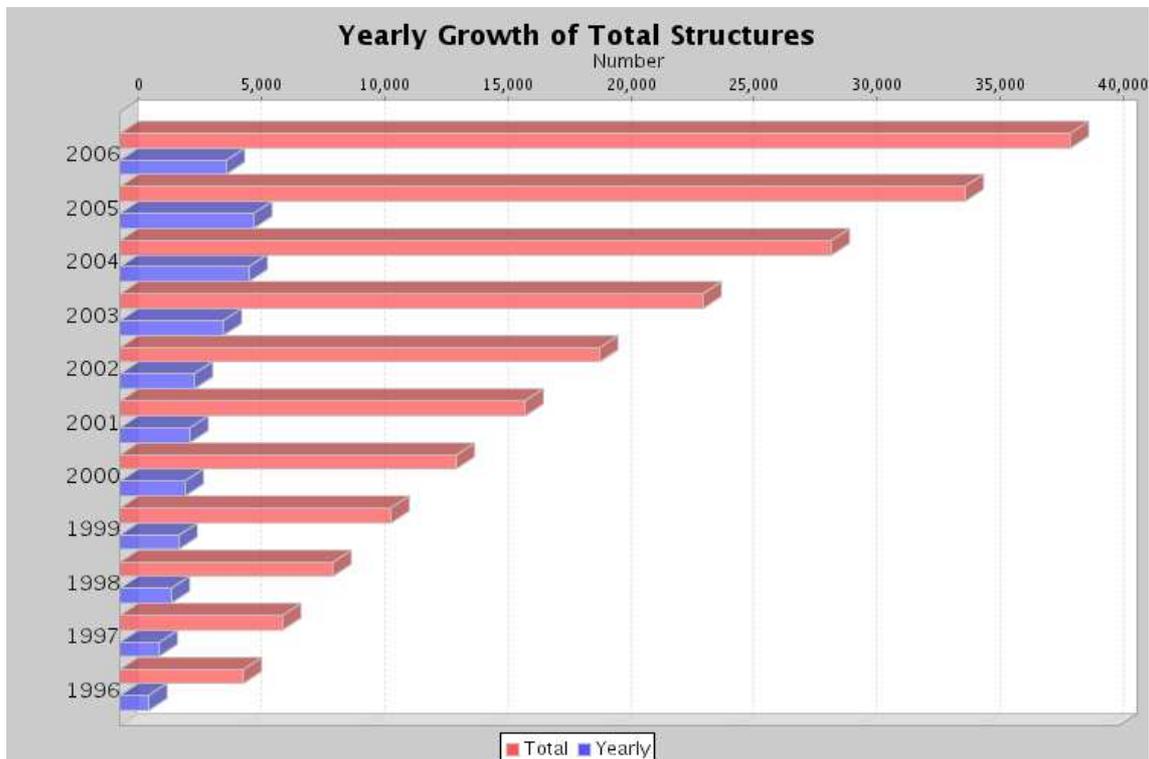


Figure 1.1: Growth of the protein database in total numbers and new entries per year. The red bars represent the total numbers of entries, the blue bars the new entries per year. (Values for the year 2006 as at early September, taken from www.pdb.org)

The challenge of our time is to transform this huge pile of data into knowledge. A deeper understanding of the function, behavior, and interplay of the individual molecules will not only lead to advances in the fields of biochemics and biology, but will also facilitate the development of new, more specific drugs: While in the past drugs were mostly naturally occurring substances, nowadays, active substances can be interactively designed in the computer. As a result, future drugs may become increasingly effective and at the same time have less side effects. The key technology for this process is called molecular modeling which is the field that develops theoretical methods and computational techniques to model, simulate, and visualize the behavior of molecules [78].

While many molecular modeling software tools were released over the years, most of them offer only limited capabilities or are difficult to use. This motivated us to develop our own molecular viewer and modeling tool BALLView. It should offer an intuitive graphical user interface to access the wide molecular modeling functionality of our biochemical algorithms library BALL [76]. In addition to the molecular modeling capabilities, we wanted to provide BALLView with state-of-the-art visualization capabilities. With this combination of visualization and modeling capabilities, users would no longer require multiple applications to calculate and visualize their molecular data. Thus, they could be more productive, since they no longer have to master several programs or need to exchange data between their modeling application and their visualization tool via file operations.

In contrast to most other available software tools, BALLView should be extensible by its users. This should be possible on two different levels: First, we wanted to realize a new extensible C++ software library VIEW that provides the most common features for molecular modeling and visualization. This new library was designated to become a part of our BALL software framework and the basis of our molecular viewer and modeling application BALLView. By using the VIEW library, users could either extend the existing program or create new software tools.

Second, we wanted to offer full access to the BALL and VIEW software libraries through a standard scripting language. This would enable real-time inspection and modification of any underlying data and the automation of repetitive tasks. Furthermore, such a scripting interface can serve as basis for student research projects, making BALLView an ideal tool for teaching molecular modeling and structural bioinformatics.

To get an idea about the prerequisites necessary to realize our vision, we first had a look at comparable software products. The next section gives an overview of the existing tools.

1.1. Current state-of-the-art

The software in the field of structural bioinformatics falls into three groups (see Fig. 1.2): One can differentiate between pure molecular viewers and molecular modeling software which concentrate on functionality like molecular mechanics. A third group is composed of molecular software libraries like Gchemical [13], which also aim at developers and can be used for own development projects. The Table 1.1 gives an overview of the most prominent tools in the different groups.

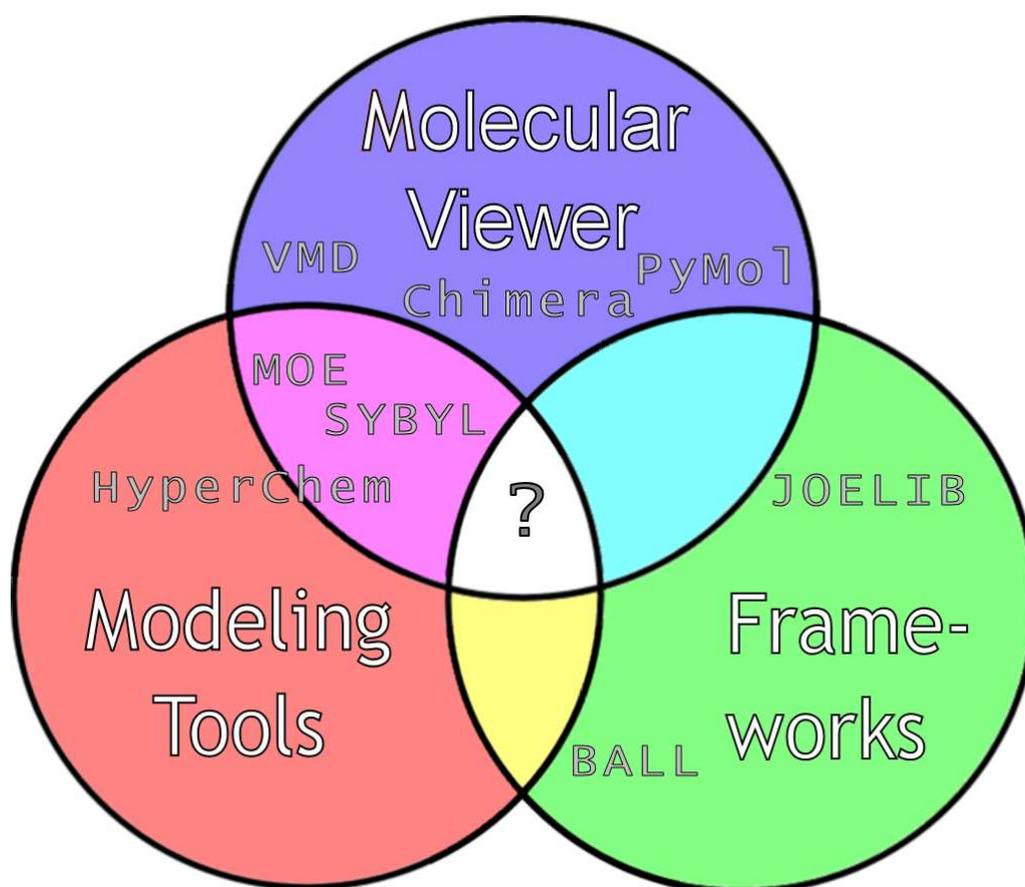


Figure 1.2: Molecular software tools can be divided into viewers, modeling tools and software frameworks. No available software provides an elaborate combination of these three groups.

While testing the available molecular visualization / modeling tools and libraries, we observed that most of them shared common drawbacks (see below). These design flaws were addressed in our own molecular viewer and modeling tool.

- Most software tools only support a limited number of operating systems or hardware platforms.

Molecular viewers	
Chimera	http://www.cgl.ucsf.edu/chimera
DS Visualizer	http://www.accelrys.com
PyMol	http://pymol.sourceforge.net
Raster3D	http://skuld.bmsc.washington.edu/raster3d
VMD	http://www.ks.uiuc.edu/Research/vmd
YASARA	http://www.yasara.org

Molecular modeling software and editors	
Benchware 3D explorer	http://www.tripos.com
Cerius2	http://www.accelrys.com/products/cerius2
HyperChem	http://www.hyper.com/products/Professional
MOE	http://www.chemcomp.com
Mopac	http://comp.chem.umn.edu/mopac
MMTK	http://www.python.net/crew/hinsen/MMTK
RasMol	http://www.umass.edu/microbio/rasmol
Spartan 04	http://www.chemistry-software.com/molecmod.htm
SYBYL	http://www.ch.cam.ac.uk/cil/SGTL/Tripes

Molecular software libraries	
Ghemical	http://www.bioinformatics.org/gchemical
JOELIB	http://joelib.sourceforge.net

Table 1.1: Examples for software products in molecular modeling and structural bioinformatics

- Visualization packages (e.g. AVS [1]) are extremely powerful for general visualization tasks, but quite difficult to adapt to specific tasks in molecular modeling.
- Most freely available molecular visualization tools (like VMD [67], PyMOL [57], WebLabViewer [11], or RasMol [94]) are more or less monolithic applications, well-suited for molecular visualization, but lacking further functionality. Only a few programs like the Swiss-PdbViewer[61] offer additional modeling capabilities. In contrast, most available molecular modeling tools have very limited visualization capabilities. Thus, users often have to master two different applications to produce and visualize their data. And even more, it is necessary to transfer data between the different applications via file operations, which can be both, time consuming and annoying.
- Commercial molecular modeling packages like SYBYL [43], Discovery studio [11], or MOE [23] provide a broad functionality in molecular modeling and computer-aided drug design. Unfortunately, they are highly expensive and rarely allow for extension on the source code level.

- Only few software tools have scripting features like PyMOL [57] to allow for automation of repetitive tasks or the interactive inspection of the underlying data. Even if an application provides scripting capabilities, it often does not support a standard language and the scripting interface is thus difficult to learn and offers only limited functionality. In addition, it is often hard to determine which features are available through the scripting interface and how this can be done.
- Many molecular modeling tools only offer complex textual or file interfaces that are hard to learn let alone to master. While molecular viewers at least supply a graphical user interface (GUI), many of them are still unintuitive to handle (VMD [67] or PyMOL [57]), because they are not based on standard user interface concepts and modern GUI frameworks.
- Very few molecular viewer and modeling tools can be extended. In many cases either the tool's source code is cryptic or it is not available at all. Thus, few tools can be used for developing own projects.
- Many molecular modeling tools and software libraries only offer inadequate documentation, which is often outdated or difficult to search and browse.
- Since most programs have some of the above design flaws, only few tools are suitable for teaching biochemistry, molecular modeling, or structural bioinformatics: They are often either too expensive, not extensible, too difficult to learn and use, or do not run on the desired platform.

1.2. Aims of this work

This section gives a short overview of BALLView's main design goals that distinguish it from any other available software tool.

State of the art graphics

Since the size of resolved molecular structures has drastically increased over the last years, molecular viewer need fast and effective rendering engines that enable the visualization of complex molecules like entire ribosomal subunits. Therefore, BALLView should harness the growing power of modern 3D graphic accelerator cards in order to offer powerful visualization capabilities like molecular surfaces or cartoon models. Furthermore, it should make use of advanced computer graphics features, for instance to visualize electrostatic potential data.

Combination of powerful molecular visualization and modeling features

Most molecular software tools presented earlier either focus on modeling capabilities or

on their visualization capabilities. In contrast, we wanted to create a new application that was specifically designed to combine state-of-the-art techniques from both fields. As a result, our software would allow its users to perform all standard modeling tasks under one intuitive user interface and visualize the results of their calculations in real time. As an example, it should be possible to load a molecule, perform and observe an energy minimization run, stop it at any time, modify the molecule, and restart the minimization. Furthermore, our program should offer the capabilities to visualize the resulting structure in a variety of ways and support the creation of images, either as screenshots or through an external renderer.

Scripting and automation

While many tools either do not provide scripting support at all or only at a very restricted level, we wanted to offer full scripting capabilities through a standard language and decided to use the Python language [35]. Python is widely used, powerful, and easy to learn. It would allow for extending BALLView with functionality that is not offered through the graphical user interface and for automating repetitive tasks. Furthermore, the Python interface should enable BALLView to serve as a powerful tool for rapid prototyping.

Ease of use

All features should be offered through one flexible and convenient graphical user interface which would make BALLView easily amenable even for inexperienced users (e.g. for students). In addition, the user interface should provide all possible means to increase the productiveness of experienced users.

A tool for research and teaching

We planned to use BALLView as a tool for teaching, and thus we wanted to fulfill the following points:

- BALLView should become easy to learn and operate, such that it can be operated by less experienced users, like students.
- BALLView should run on all major platforms and low-end hardware.
- BALLView and the underlying libraries should be available under a free license like the LGPL and extensible on the source code level. Thus, they would be the ideal basis for software projects of research groups.
- A scripting interface with access to the full functionality of the BALL library should make BALLView ideally suited for student projects. The students thus would not need to implement any basic functionality and could thus concentrate on the more challenging parts of their projects.

Functionality

With the above design goals in mind, we created the new molecular modeling and visualization tool BALLView. It provides numerous visualization capabilities, including all standard molecular models and sophisticated methods for displaying electrostatics potentials. To render these kinds of data, BALLView uses an integrated OpenGL engine that allows for realtime visualization. But the rendering possibilities are not limited to the internal renderer. As an alternative, users can export their scenes to the external renderer POVRay and thus achieve very detailed images.

BALLView is not limited to visualization alone. From the very start of its development, we also designed it as a powerful tool for molecular modeling that provides a common graphical interface for the wide range of functionality implemented in the BALL library [76, 51]. Therefore, it supports molecular mechanics features, like the AMBER [55], CHARMM [52], and MMFF94 [62] force fields, molecular dynamics simulations and energy minimizations. Furthermore, BALLView provides molecular editing, the calculation of molecular electrostatic potentials and molecular docking.

All this functionality is not only available in the form of a standalone application. The underlying VIEW framework allows for full access to all feature domains, like the molecular mechanics functionality or the 3D visualization. These domains were encapsulated in individual widgets that are independent from each other (see Page 16). Therefore, programmers can easily combine the existing functionality to create new, custom-tailored programs.

In order to allow access to the BALL and VIEW libraries from within BALLView, we extended its graphical user interface with an embedded Python interpreter. It supports Integrated Development Environment features like syntax sensitive help and auto completion and thus greatly simplifies the development of Python scripts.

The description of the modeling, implementation, and functionality of BALLView and the underlying VIEW framework are the content of this work. The next chapter will give an overview on the applied design concepts and some implementation details, followed by the presentation of the software's features and capabilities in Chapter 3. Finally, a concluding summary and an outlook on future work will be given.

2. Design and implementation

2.1. Overview

At the start of this work, the Biochemical Algorithms Library already contained visualization functionality [76, 51]. This code was still immature and had some serious problems:

- Memory leaks, e.g. in the OpenGL rendering code
- Crashes, e.g. if a MD simulation produced strange energies
- Inefficiencies, e.g. in the calculation and rendering of the models

Furthermore, the existing implementation was quite difficult to use, since its control flow was difficult to understand. As an example, it was often unclear which classes and functions were responsible for performing basic tasks like adding new molecules or atoms, updating the 3D view or (re)building models. Thus, the existing code had to be simplified. Moreover, the original visualization code was divided in two distinct software libraries, "libVIEW" for general visualization tasks and "libMOLVIEW" for the visualization of molecules. This distinction was artificial, because some classes could not be clearly assigned to one of the two domains and the division in two distinct libraries caused unnecessary overhead. Therefore, a redesign of the entire software project was necessary. We created a new software library "VIEW" that contains all visualization features of the BALL framework. Next, we performed a detailed analysis of the existing implementation and came to the conclusion that our new visualization library should still use the majority of the existing classes (for a description of the original approach see [73]). To adapt the classes to the new software design, they had to be heavily re-engineered and most of them were basically rewritten from scratch. In addition, further efforts were needed to ensure that the existing code worked correctly on Microsoft Windows and Mac OS.

Dependencies on external libraries

Since the visualization library should be accessible through a graphical user interface, it had to rely on a GUI toolkit. This toolkit should be portable, object-oriented, written in C++, and easy to use. Therefore, the Qt framework [36] was the only adequate choice: It implements all the functionality required to write GUI based applications and offers support for the integration of the OpenGL library [26]. This was very useful, since OpenGL is the

only feasible basis for our 3D graphics engine, as it is platform independent and provides high-performance rendering.

Unfortunately, OpenGL's platform independence is also the reason for an unpleasant drawback. The versions of the OpenGL header files can differ in-between the individual platforms and every graphics driver delivers its own header files. Therefore, it can not be guaranteed that all OpenGL methods are defined in one set of header files. To circumvent this problem, OpenGL 1.1 introduced the usage of function pointers to access the newly added methods. Unfortunately, the syntax for obtaining these pointers is different under Linux, Mac OS, and Windows. Thus, wrapping code would be needed around every piece of code that uses advanced OpenGL features. This would not only be inconvenient for the programmers, but also very error-prone. To circumvent these problems, our implementation uses the open-source library GLEW [14], which offers a convenient interface to the OpenGL functions. GLEW is released under the GLX Public License, available for all platforms, and thus fits well in our software project.

Structure of the VIEW framework

The functionality in the VIEW framework can be categorized into different domains (see Fig. 2.1)). First, we created a set of basic data structures and functions. Based on these, we developed the application functionality, like the visualization classes. They consist of simple geometric objects, the different models, coloring schemes and several renderers. Next, we designed a set of dialogs and classes that form the graphical user interface and guarantee the modularity of our approach. To this end, we implemented a set of modular widgets that are widely independent from each other and encapsulate one set of features each, like e.g. the 3D graphics, or the molecular modeling functionality. By combining these and other widgets to an application, we were able to build BALLView with roughly 500 lines of additional code.

The following sections will describe the more interesting domains in the VIEW library and their interplay. Since the library currently consists of roughly 180 classes, 50 dialogs and more than 30,000 lines of code, the following text can only give a very limited overview. Therefore, it concentrates on the main design principles and gives some examples of the library's usage.

2.2. Design goals

Many molecular viewers are available without costs, but most of them are not available under a public free license like the GPL [15], i.e. users can not freely copy or modify the software. Even worse, most molecular modeling tools are closed source software and come with a hefty price tag, which may overextend the budgets of many research

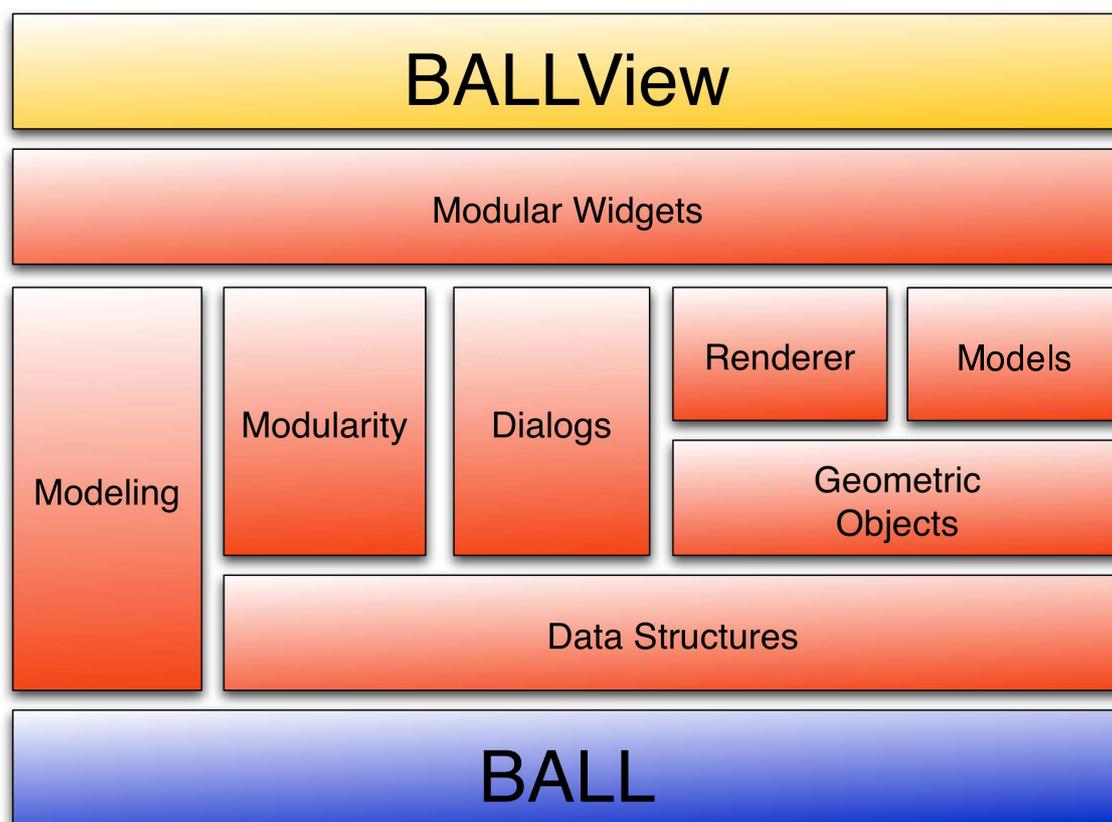


Figure 2.1: The design of the VIEW framework: Based on the BALL library, we developed a set of classes for the GUI, visualization, and molecular modeling. These different functionality domains were encapsulated into individual modular widgets which were combined to build BALLView.

institutes. BALL, VIEW, and BALLView should become free software such that they can be used worldwide at every institute, by individual researches, students, and even pupils. In addition, every user should be allowed to modify the source code to his liking, such that he can adapt it to his needs. Therefore, BALLView should be ideally suited as a tool in teaching or research.

With this vision in mind and after analyzing the available software tools, we found that our molecular viewer and modeling tool should address the following issues:

Generality:

"the degree to which a system or component performs a broad range of functions" [89]

Our software should provide a wide range of features, challenging all other comparable tools. Since we based BALLView on our Biochemical Algorithms Library (BALL), it benefits from BALL's powerful capabilities in the field of structure based bio- and chemoinformatics. Thus, we were able to create a tool which allows its users to perform all standard

modeling task via an intuitive user interface and visualize the results. To this end, BALLView offers advanced visualization features like molecular surfaces or the rendering of electrostatic potentials (see Section 3.2.3).

Efficiency:

"the degree to which a system or component performs its designated functions with minimum consumption of resources (CPU, Memory, I/O, Peripherals, Networks)" [89]

While the runtime of complex calculations is often of key concern, a high efficiency is even more important for all fields that require real time interaction since users strongly dislike programs that force them to pause their work. This particularly applies for rendering three-dimensional graphics. Therefore, we optimized the time-critical parts of our software such that it can fluently process and render large molecular complexes. Furthermore, BALLView is thoroughly memory efficient and supports rendering at different detail levels such that it can be run on low-end hardware. For an overview on the performance optimizations see Section 2.6.1.

Usability:

"the ease with which a user can learn to operate, prepare inputs for, and interpret outputs of a system or component" [89]

Many of the software tools described in Section 1.1 are difficult to use: While molecular viewers offer at least some kind of graphical user interface (GUI), these are often unintuitive. Furthermore, many modeling tools or molecular mechanics packages only offer interfaces based on cryptic configuration files. This results in book sized manuals, which have to be studied even for basic tasks like reading or writing a molecular file format. In contrast, we wanted to offer a wide functionality through an intuitive graphical user interface. To meet the need for such a state-of-the-art, user-friendly, and portable graphical user interface, we decided to base BALLView on Qt [36], a GUI toolkit available for all relevant platforms. Qt not only provides all essential functionality to build graphical applications in C++, its object-oriented design fits well with the design of BALL, allowing for a seamless integration of the two libraries. To further ease the program's usage, we developed new context sensitive help systems. A full description of our software's usability features is given in Section 3.1.2.

Portability:

"the ease with which a system or component can be transferred from one hardware or software environment to another" [89]

Most molecular modeling tools only support a limited number of operating systems and often depend on one type of hardware, like the x86 architecture. This is annoying for users that depend on another operating system for their day to day work. Therefore, BALL and BALLView support all common operating systems and most modern hardware. This is achieved through the combination of C++ , Qt, and OpenGL, which ensures a maximum of portability (see Page 11).

Extendability:

"the ease with which a system or component can be modified to increase its storage or functional capacity" [89]

Almost all available molecular viewers and modeling tools have a fixed set of functionality and can not be easily extended by their users. In contrast, BALLView was designed as open source software that is extensible and adaptable by any user with knowledge in C++ or Python. The means to achieve a high extendability are described in Section 2.4.

Maintainability:

"the ease with which a software system or component can be modified to correct faults, improve performance, or other attributes, or adapt to a changed environment" [89]

Ensuring the maintainability of large software projects is often a nontrivial task, because it is difficult to estimate if changing one part of a software results in potential problems in any other part. Therefore, both BALL and the VIEW library are strictly object oriented. In addition, we applied a modular software design that results in individual functionality modules, like file input/output or the 3D graphics, which are largely independent from each other. To monitor if a change in an existing implementation would introduce any faults in other parts of our software, we created an elaborated testing framework for the individual classes. Next, we produced comprehensive documentation for all the classes and integrated it into the source code to ease any subsequent changes.

Correctness:

"the degree to which a system or component is free from faults in its specification, design, and implementation" [89]

Large software projects impose huge engineering challenges: Through their mere size it can become almost impossible to ensure the correctness of the source code, let alone the program as a whole. The situation even gets worse if additional dependencies on external libraries exist. Here, especially GUI frameworks are troublesome if they have

their own stability issues or the behavior of individual methods changes between individual versions. Further problems arise if different platforms or compilers are involved.

Unfortunately, all the above points apply to the VIEW framework. Therefore, we had to invest much time and effort to ensure our software's correctness and stability. Details are given in Section 2.7.1.

Robustness:

"the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environment conditions" [89]

Biochemical data is often erroneous or incomplete. Hence, we needed to ensure that our software still works correctly if the data input contains any flaws. This is achieved by adding try/catch blocks or consistency queries to the critical pieces of the source code. Faulty input data thus results in descriptive error messages instead of crashes or unexpected behavior. Another source of trouble arises from the combination of multithreading techniques with a graphical user interface: such a program has to enforce at any time that users can not accidentally interfere with any ongoing tasks in a way that causes instabilities. Our approach to this problem is twofold. First, we disable all user interface entries that must not be used at a given time. Second, we implemented a locking mechanism that aims at preventing any harmful function calls while additional threads are running (see Section 2.6.3).

2.3. Modularity

Most available tools in the fields of structural bioinformatics are monolithic software packages that do not easily allow for any further extensions. In contrast to such tools (like VMD or PyMOL) the VIEW library was designed with the goal to achieve a maximal extensibility and maintainability. To realize these demands, we encapsulated the individual feature domains into different modules (like OpenGL rendering or force field methods) that were realized as a set of independent modular widgets (see Fig. 2.2). These modules automatically connect to each other and thus can be freely combined to form an application. In addition, this approach allows users to create their own widgets and thus incorporate new kinds of functionality.

To this end, we have designed a set of base classes describing the interactions of the individual widgets. 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. To improve the readability, class names and methods will be shown in typesetting style.

2.3.1. MainControl

The class `MainControl` is derived from Qt's `QMainWindow` and thus realizes an application's main window. To minimize the potential overhead, it contains only the most essential data structures that are always present in a molecular viewer and modeling tool: the molecular structures and the visualization objects.

To ease the management of these data in the `MainControl`, we integrated two additional classes: the `CompositeManager` stores all molecular entities (`Composite` objects) and the `RepresentationManager` is responsible for the visualization models and the thread for their (re)calculation. All further functionality (like reading and writing of structures, the scripting interface, etc.) are added to the main window by instantiating one of the classes derived from `ModularWidget`.

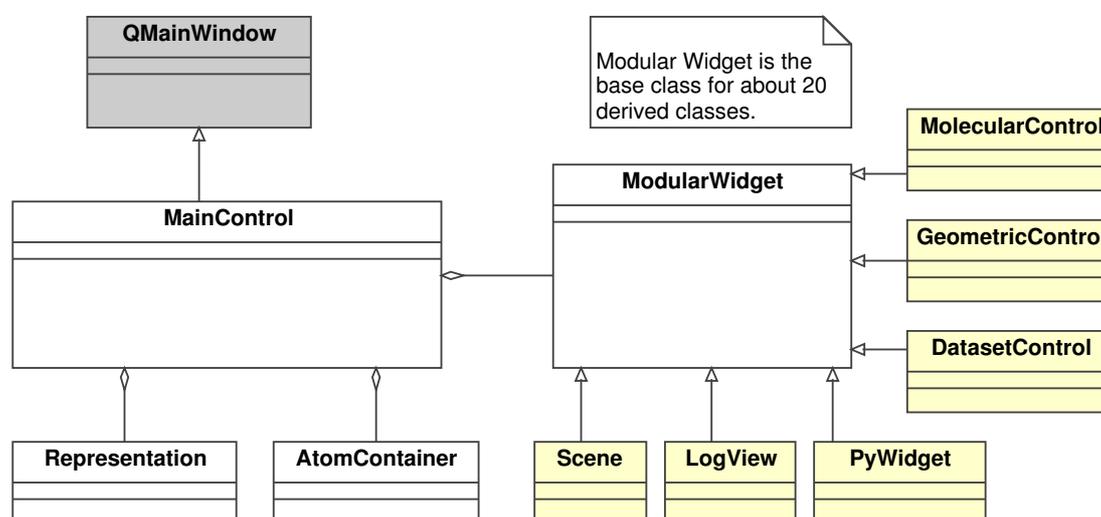


Figure 2.2: UML diagram of BALLView's core architecture: The `MainControl` is the main window of every VIEW application and serves as a container for all loaded molecules and representations. It also connects the modular widgets with the messaging system. The classes derived from `ModularWidget`, which are shown in yellow, can be found in Fig. 3.1. `QMainWindow` stems from the Qt-library.

2.3.2. Modeling of the modular widgets

`ModularWidget` is a common base class for all the modules that can be combined to form an application. As stated above, the `MainControl` only contains the most basic set of data structures, the loaded molecules and the visualization objects. The design of the VIEW framework allows these data sets to be represented and modified by several modular widgets at the same time. This is realized through the so called "**Observer Synchronization**" pattern [30]. Every individual widget can act as an "Observer" on commonly shared data and provide user interface elements to modify this data. If a widget modifies the

data, it notifies all other widgets of these changes, which will respond by updating their visual content.

To reduce the complexity of this approach, we have designed the individual widgets such that they do not need to know about each other: instead of passing the messages directly to the receiving widgets, the sending widget hands the message over to the `MainControl` which then notifies all registered `ModularWidget` one by one. A receiving widget freely decides if it has to react to the incoming message, by checking the message's type and contained data. The details of the messaging system will be described in detail in the next section.

As a complementary technique, we used "**Flow Synchronization**", i.e. screens get synchronized with the underlying model based on the flow of the user interaction between the screens [30]. This approach was used for dialogs that are not constantly needed and thus were not implemented as modular widgets. Instead they are contained in one `ModularWidget`, which is responsible for keeping their content synchronized. In the majority of cases, this was achieved by setting these dialogs to be modal, i.e. no other user interface elements can be accessed, while such a dialog is open. Thus, the dialog's content only needs to be synchronized before and after showing it.

The VIEW library already contains numerous prefabricated modular widgets, each of which is designed for one specific task (see Table 2.1). Therefore, in order to build an application, the user only has to instantiate the modular widgets with the `MainControl` as their parent, as can be seen in the following code snippet:

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

These few lines of code (header includes were omitted for brevity) create a fully-fledged molecular structure viewer with the set of widgets shown in Fig. 3.1.

Additional features of the modular widgets

The modularity of our approach greatly contributes to the extendability of our software since users can add new functionality, by implementing additional modular widget. To further ease and accelerate this kind of extensions, we designed the `ModularWidget` class to provide many commonly needed features:

Name	Functionality
DatasetControl	Management of data sets
DemoTutorialDialog	Demo and stepwise tutorial
DisplayProperties	Creation and modification of models
DockingController	Molecular docking
DownloadPDBDialog	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
HelpViewer	Integrated documentation viewer
LabelDialog	Creation of labels in the 3D view
LogView	Logging window
ModifyRepresentationDialog	Customization of models
MolecularControl	Hierarchical overview of loaded molecules
MolecularFileDialog	Reading and writing of molecular files
MolecularStructure	Molecular mechanics and modeling features
PubchemDialog	Download and creation of small ligands
PyWidget	Python scripting
Scene	Three-dimensional graphics
SnapshotVisualisationDialog	Visualization of trajectories
TestFramework	Recording and playback of user input

Table 2.1: Overview of the classes derived from `ModularWidget`. Each individual class was created for one specific functional domain and is widely independent from the other widgets.

- **Messaging system**

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 2.3.3).

- **Printing status and error messages**

To keep the user informed about the application's current state and potential problems, all modular widgets support the sending of status and error messages. These can either be piped to the standard output device (like a shell window), printed in a special log windows, or shown in the application's status bar.

- **Management of menu and toolbar entries**

The `MainControl` as the application's main window provides a menu bar that is accessible by the modular widgets, to add their own entries. The widgets are responsible for disabling their menu entries if the corresponding action is disallowed at a given time.

- **Registering widgets and menu entries for the help system**

The VIEW framework features a context sensitive help system (realized in yet another modular widget). It allows for binding individual GUI elements to a corresponding section in the documentation (see Page 54). If a modular widget contains such GUI elements, it is responsible for creating the binding. As an example, to register the 3D view to its documentation, all that is needed is the following line in the classes constructor:

```
registerForHelpSystem(this, "scene.html");
```

- **Configurability**

Every modular widget can have an arbitrary number of child pages in the application's preferences dialog (see Section 2.4.3). In addition, the `ModularWidget` class defines an automatic interface for reading, storing and applying a widget's settings to/from a configuration file.

- **Registering of supported file formats**

Since the libraries functionality is separated into independent modules, the modules have to register their supported file formats. This is needed for parsing command line arguments or for the drag-and-drop support. The registration process is rather simple. A derived `ModularWidget` class just has to overload the two following methods:

```
virtual bool canHandle(const String& fileformat) const;  
virtual bool openFile(const String& filename);
```

- **Access to individual instances**

While the molecular widgets are widely independent, they sometimes still might need directly access to each other. Therefore, we derived the modular widgets from the class `Embeddable`. It provides access to a classes instances through the method `getInstance()`.

- **Access to the MainControl, the loaded structures, and the representations**

All modular widgets can easily access the `MainControl` and through it the loaded molecules and representations.

- **Locking of molecular entities**

To achieve better performance and responsiveness of the graphical user interface, the VIEW library supports multithreading techniques (see Section 2.6.3). These techniques require the locking of shared data, i.e. the molecular structures. Therefore, all modular widgets have the means to lock and unlock this data.

- **Freely placeable and dockable widgets**

With the class `DockWidget`, we designed a derived `ModularWidget` class that allows for the creation of freely placeable and dockable widgets (see Fig. 3.1).

By providing these features, the class `ModularWidget` greatly simplifies the development of additional widgets. As a result, users can easily combine new and existing widgets both to extend `BALLView` with new functionality or build new custom-tailored applications.

2.3.3. Messaging system

A special mechanism was needed for enabling the individual modular widgets to be independent from each other, but still be able to work together, by notifying each other about the current work flow. This was achieved through 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` is a modular widget that provides a hierarchical overview of the loaded molecules. Other widgets offer functions that operate on the currently highlighted molecular entities in the `MolecularControl`. The menu entries for these features must be disabled if no molecular item is highlighted. Therefore, when a user highlights some of its items, the `MolecularControl` sends a `ControlSelectionMessage` to notify the other widgets. Since the modular widgets have to notify each other about many varying types of events, the class `Message` has many subclasses (see Table 2.2), which store different data, like selections or object pointers. If a new event is introduced, it can easily be added by creating another `Message` subclass.

The actual sending of messages is done in the method `ModularWidget::notify_`, while the message is received by `ModularWidget::onNotify()`. In this method the modular widget decides if it needs to react to the message, which is done through runtime type identification. Furthermore, many types of message provide enumeration values for defining further specialized message subtypes. As an example, a `CompositeMessage` can cope with the events of an added or deleted molecule by using the types `CompositeMessage::NEW_COMPOSITE` or `CompositeMessage::REMOVED_COMPOSITE`. By using such enumeration subtypes, less subclasses are needed to distinguish different types of events, resulting in a more compact implementation. Thus, new `Message` subclasses were only added, when a message had to transmit a new kind of data or if no appropriate class existed.

2.4. Extensibility

`BALLView`'s extensibility was of central importance, since it was designed to become the basis for our future developments in molecular modeling and molecular visualizations. Here, it allows students to integrate their research projects into one common graphical

CompositeMessage	SceneMessage
GenericSelectionMessage	ControlSelectionMessage
NewSelectionMessage	GeometricObjectSelectionMessage
RepresentationMessage	MolecularTaskMessage
ShowDisplayPropertiesMessage	CreateRepresentationMessage
DatasetMessage	DockingFinishedMessage
DeselectControlsMessage	

Table 2.2: Derived message classes. Almost all can contain specific data or have additional enumeration types.

user interface. This will both, ease these developments and extend BALLView's functionality. In addition, the VIEW framework is released as free software and can thus also serve as basis for development projects in other research groups or companies.

With these ideas in our mind, we spend a substantial amount of time and effort to guarantee that every aspect of BALLView and the underlying VIEW framework can be adapted and extended:

- The modularity of our approach, described in Section 2.3.2 allows for incorporating new functionality through the implementation of additional modular widgets. These new widgets can either be independent from the already implemented ones or use the existing functionality by calling the corresponding modular widget methods. Since the graphical user interface is modular and can be freely rearranged, it allows for the seamless integration of any additional widgets.
- Both, the application and the underlying libraries, have a fully object-oriented design to simplify any future extensions. As an example, all existing modular widgets contain virtual methods that are intended to be overloaded in derived classes. Thus, users can easily change the behavior of the existing widgets by adding new methods.
- The application's preferences dialog can easily be expanded (see Section 2.4.3). To add a new child dialog, only one line of code is needed:

```
preferences.insertEntry(new ModelSettingsDialog(this, "Models");
```

- Arbitrary types of data sets can be stored and manipulated in a specially designed widget. This `DatasetControl` can easily be extended with support for additional data types (see Section 2.4.1).
- The class design for the rendering engine (see Section 2.5.4) allows to develop further types of renderers by deriving a new class from `Renderer`. Such a new renderer could in the future, for instance export a scene to VRML format or provide an integrated ray tracing engine. In addition, support for rendering further geometric

objects (see Section 2.5.1) can be added by extending the existing renderers with the new methods.

- New models and coloring processors can be added by implementing a derived model information class and overloading the corresponding dialogs.
- The Python interface (see Section 3.4) allows to add new functionality without the need to write, compile, and link any C++ source files.

All these extensions can be realized without any changes to the existing code. To further improve the extendability, we wrote an elaborate documentation, describing the usage of the BALL and VIEW library as well as the graphical interface (see Section 3.1.3). Furthermore, we created tutorials that covers the usage of the basic BALL classes and the extension of the viewer functionality.

2.4.1. Support for arbitrary data sets

As described on Page 17, the `MainControl` only stores the two most important and commonly needed types of data: the molecules and their visualizations. Support for storing any additional data types is available through a specialized modular widget, the `DatasetControl` (see Fig. 3.1). When we implemented this class, our goal was to make the class design as modular, extensible, maintainable, and easy to use as possible. Therefore, we decided to employ the "**ModelViewController (MVC)**" pattern [30] and thus split the interface into three parts: the model (or data), the view (or widget), and the controller. Since the data management is thus decoupled from the presentation in the widget, the implementations of the different data types can be realized in separate, independent classes. As a result, future extensions to provide additional data types can be implemented without changing any existing code. To realize the above described approach, a common generic interface was needed. This interface is defined in the `Dataset` and `DatasetController` base classes (see Fig. 2.3).

A `Dataset` can contain any type of data and supplemental information, like the objects name and a pointer to the molecular entity from which it originates. The actual data storage is performed in derived classes to achieve a type safe encapsulation. As an example, the class `DockResultDataset` stores a docking run result. To simplify the creation of derived `Dataset` classes, we developed a preprocessor macro that automatically defines and implements such a class. As a result, only one line of code is needed to define a derived data set class, as can be seen in the following example for the class `DockResultDataset`:

```
BALL_CREATE_DATASET(DockResult)
```

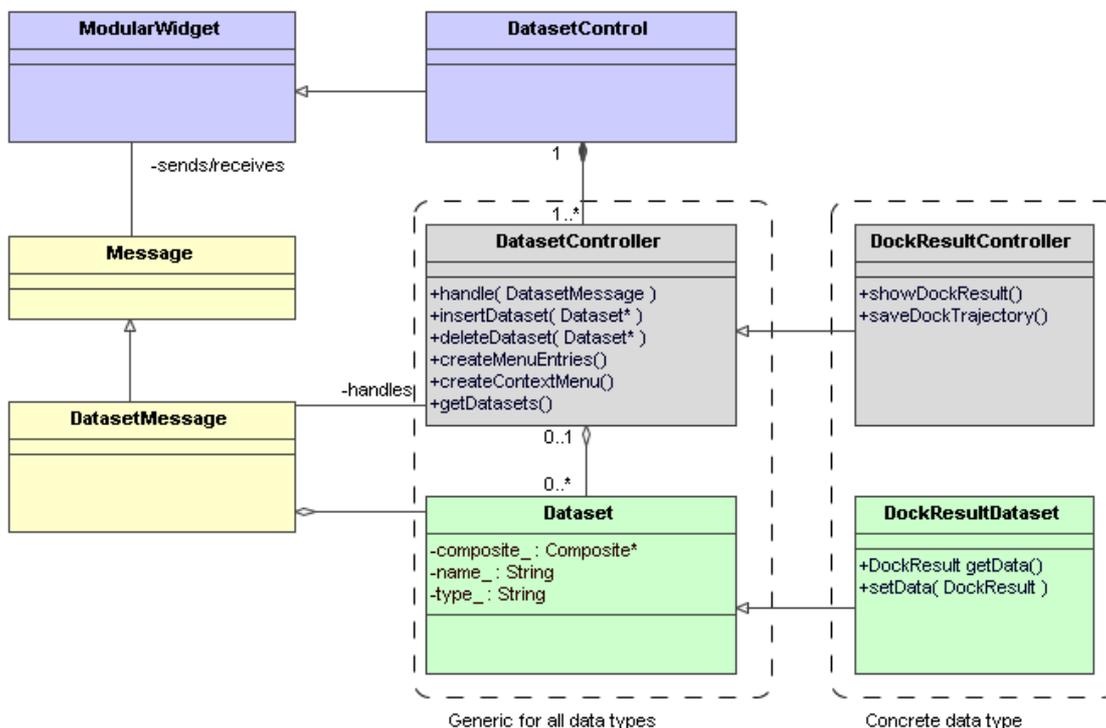


Figure 2.3: UML diagram for the DatasetControl. Widgets are shown in blue, messages in yellow, data set controllers in gray, data set classes in green. For the description see Page 23.

The DatasetController base class defines the interface for storing data sets and accessing the respective functionality like file operations, visualization and deletion. These features are implemented in derived classes, where each class provides the support for one data type. For instance, we created the DockResultController class for DockResultDataset, .

To make their functionality available, the controllers provide their own menu entries which are automatically added to the application when a controller is registered in the DatasetControl. This registration is very simple, as can be seen in the following example:

```

DatasetControl* dc = DatasetControl::getInstance(0);
dc->registerController(new RegularData3DController);
dc->registerController(new VectorGridController);
dc->registerController(new TrajectoryController);
dc->registerController(new DockResultController);
    
```

These few lines install the controllers for the four data types that are currently supported in BALLView: scalar data grids, vector grids, trajectories, and docking results.

The controllers must also support the management of data sets that are created elsewhere. This happens for example in the case of docking runs, where another modular widget is responsible for starting the docking runs and thus creating the corresponding data object. Therefore, the messaging system (see Section 2.3.3) had to be extended to support such data set related events. This was achieved by implementing a new `DatasetMessage` class that stores a `Dataset` object (see Fig. 2.3). When such a message is received by the `DatasetControl`, it compares the contained data type with the registered controllers. If a matching controller is found, it is used to process the message, for instance to add a new item to the list view.

While the described approach may seem somewhat complicated, it improves the maintainability and simplifies future modifications and extensions. The `DatasetControl` can, e.g. easily be customized to provide only support for the needed types of data. This can lead to leaner applications. Also, support for additional data types can be added by deriving a new `DatasetController` class and registering it. Finally, the chosen approach allows for a much simpler messaging system since only one message class is needed for any kind of data.

2.4.2. Creation of dialogs

To layout the dialogs in the VIEW library, we used the program "Qt Designer" (see Fig. 2.4). 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 is derived from the layout class and contains the dialog's functionality.

While this procedure may seem a bit complicated, it is straightforward and very useful. Not only does the WYSIWYG interface accelerate the development process, the resulting "*.ui" files uncouple the dialog's 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. Also, after creating one dialog layout, this design can be applied to several dialogs which can differ in functionality and behavior.

2.4.3. Configurability

A common problem with software products is the lack of configurability, i.e. users can not adapt a program's features to their needs. In contrast, we intended to give BALLView's users the opportunity to adjust it to their liking in any conceivable way, including the different models, coloring methods, and display options. Therefore, an extensible graphical

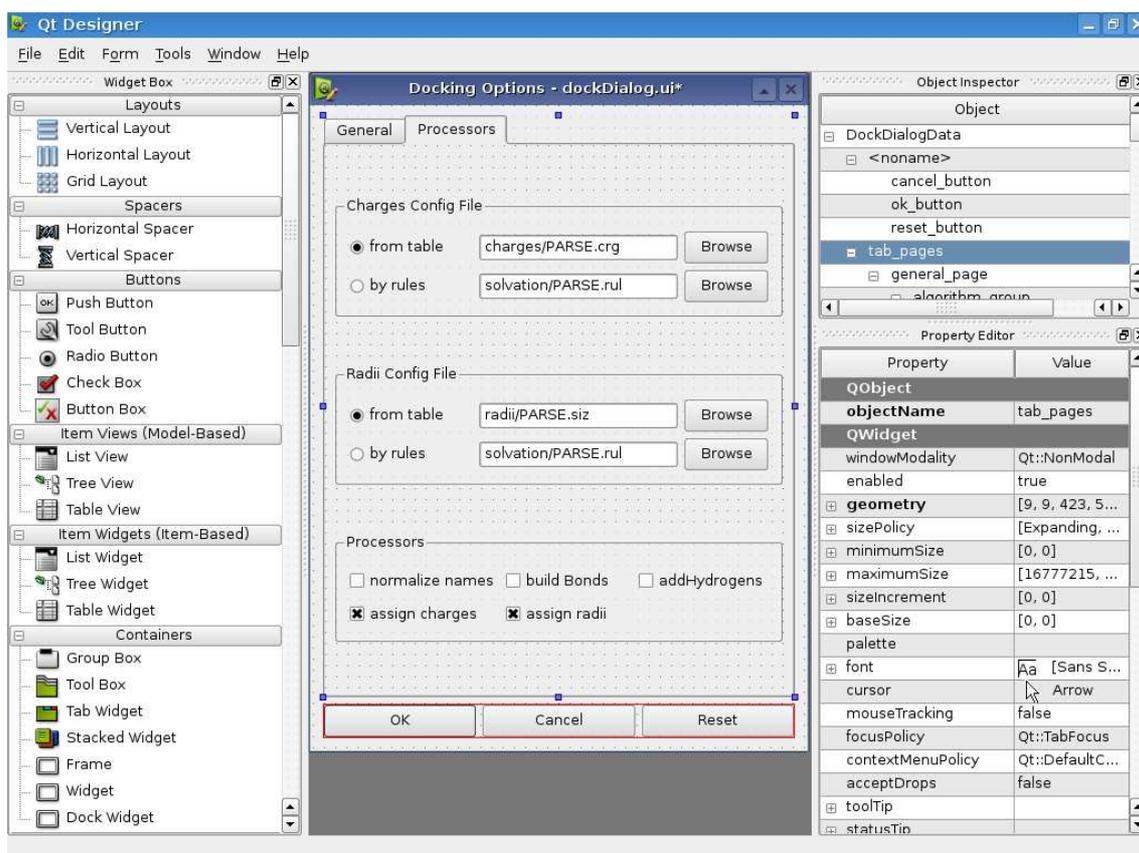


Figure 2.4: The Qt Designer for developing graphical user interfaces.

user interface for applying these settings was needed. 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 are shown as entries in a hierarchical list (see Fig. 2.5). If a user clicks on such an entry, the corresponding dialog is shown in the widget stack. This approach allows to cluster the settings in a hierarchical way and enables users to freely browse and apply the individual settings. Furthermore, the Preferences dialog can have any number of child dialogs and still have a concise layout.

Automating the (re)storing of the settings

The configurability of our software would not make much sense if the settings would not get stored for later usage since users would find themselves forced to enter the same settings over and over again. Therefore, all of BALLView's settings are stored when the application is closed. This process relies on two mechanisms: First, every individual modular widget has methods to read, store, and apply any arbitrary number of settings in a line based configuration file. As a second alternative mechanism, we provided the Preferences dialog with the ability to append its settings to the same configuration file. In the early versions of our implementation, every modular widget and dialog contained

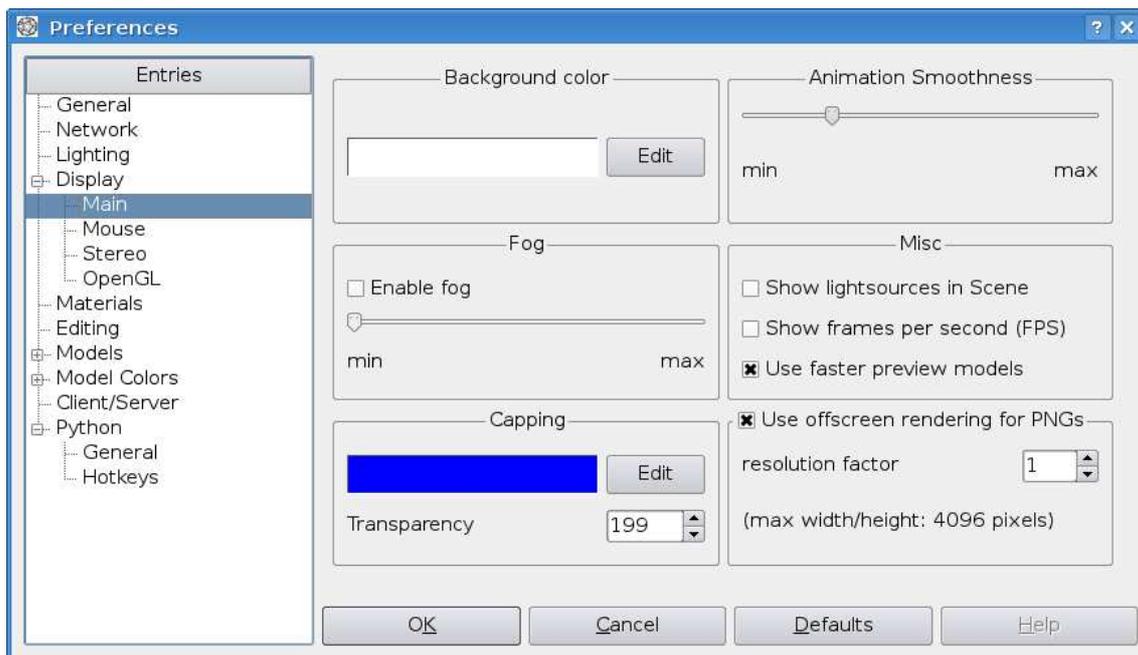


Figure 2.5: The PreferencesDialog allows to adjust almost all kind of features in BAL-LView

its own routines for the reading and storing process. Since this resulted in large overhead in terms of redundant source code, we automated 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 later saved or restored. Thus, extending a dialog with this feature is very simple, since exactly one line of code has to be added to it's constructor:

```
registerWidgets_();
```

This sole line ensures that the dialog's data get stored or read. Compared to the earlier implementation, which required dozens lines of code per dialog, 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());
}
...

```

The resulting configuration file is divided into sections, which can correspond to individual dialogs. The following lines illustrate the file format, by presenting the section for the energy minimization setup dialog. From these lines the dialog's content can be reconstructed and thus the minimization settings restored.

```
...
[MINIMIZATION]
energy_difference_lineedit=0.0001
minimization_group=conjugate_button
max_grad_lineedit=1.000000
...

```

Further applications

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 supports the storing of default values that are applied when a dialog's "Defaults" button is pressed (see e.g. Fig. 2.5). In the same way a dialog is 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 the tables for the setup of the different coloring methods. This approach further improves the extensibility, because new derived ExtendedPreferencesObject classes can be designed and thus support for sophisticated composite GUI elements be added.

Summary

We designed a user-friendly way to apply any arbitrary number of options. Furthermore, the implemented approach is very powerful, since it is extensible and minimizes the efforts for (re)storing the content of further dialogs. Even more importantly, the described approach is less error-prone than the naive implementation, since developers can no longer forget to add the (re)storing code for one GUI element.

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 storing user defined values. A conservative estimation of 8 lines of code per widget for the storing/restoring of its data yields a saving of more than 1500 lines of code.

2.5. Class design for the visualization features

This section describes the class hierarchy for computing and rendering visualizations. First, the different geometric objects which may belong to a visualization are presented. Next, the concept of a representation is described and the process of creating a visualization is illustrated. Finally, the section concludes with an overview of the supported rendering techniques.

2.5.1. Geometric objects

We designed a base class `GeometricObject` which provides a general interface for geometric shapes that can be computed and rendered in the VIEW framework. From this base class, we then derived the classes for the individual geometric objects (see Table 2.3). The instances of these derived classes are constructed by a model processor, are later colored by the color processor, and finally stored in a `Representation`. The renderer classes then translate their information such that they can be drawn on the screen or processed by external programs.

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

Table 2.3: Overview of the different geometric objects that are supported in the VIEW framework.

To guarantee that a wide range of shapes and objects can be visualized, we implemented numerous derived `GeometricObject` classes. If nevertheless the need for new kinds of geometric objects will arise, such extensions can be realized with minimal effort. All that is needed is the creation of a derived `GeometricObject` class and the adding of the corresponding rendering methods to the `Renderer` classes (see Page 31).

2.5.2. Representations

To offer the user an intuitive way of handling models and their coloring, we designed 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. 2.6). This approach has many advantages:

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

To simplify the creation and modification of representations, we designed a user-friendly dialog which allows to assign all settings of a `Representation`.

2.5.3. Molecular models and their colorings

We created a wide variety of different models (see Section 3.1), each of which is implemented in one corresponding class. All these model classes are derived from the class `ModelProcessor`. This class is derived from the BALL class `UnaryProcessor<Composite>`, which provides a general interface for recursively iterating and processing a molecular entity (`Composite`) tree. Therefore, a `ModelProcessor` can be applied to entire proteins as well as to individual atoms, which allows for the building of 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 construction. With this information, the method `createGeometricObjects()` computes the individual geometric objects, 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.

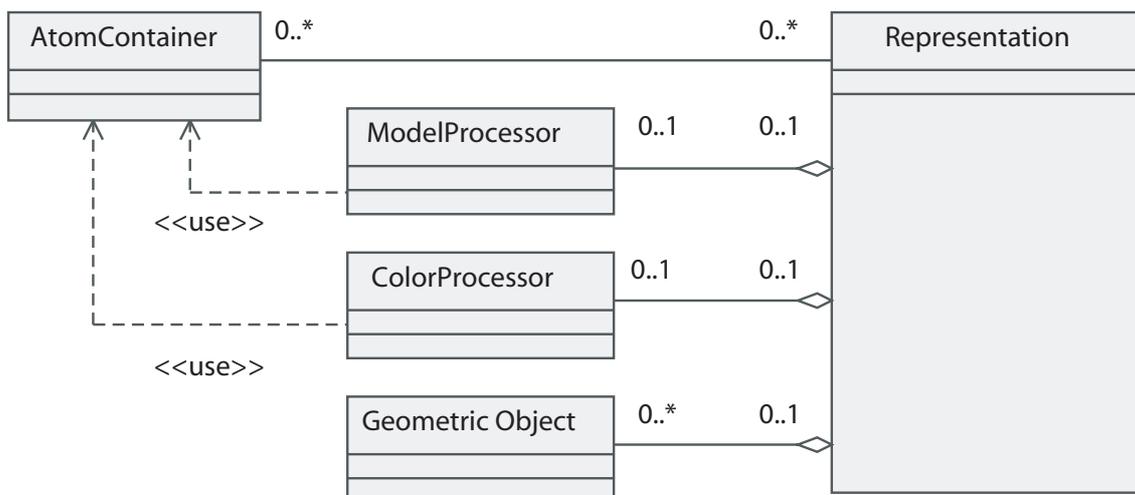


Figure 2.6: Each visualized object corresponds to an instance of the class `Representation`. The `ModelProcessor` creates `GeometricObjects`, like tubes or meshes for all atoms stored in the `AtomContainers`. Next, the `ColorProcessor` colorizes the `GeometricObjects`, e.g. by element, charge or temperature factor. The individual model types and coloring methods are realized by derived classes. This approach simplifies the implementation of new models and coloring methods and allows their free combination.

The counterpart of `ModelProcessor` is the `ColorProcessor` class, which iterates over a list of `GeometricObjects` and computes their colors. Currently, 16 coloring methods are available and each is implemented as a derived `ColorProcessor` class.

Since users directly perceive how long it takes to compute a representation, we made great efforts to ensure maximum performance for the model and coloring calculations. For details on these optimizations see Section 2.6.1.

2.5.4. Renderer

Currently `BALLView` support two different renderers. Real time graphics are offered 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 defines a general interface. This common interface ensures that, in future versions, arbitrary renderers can be added by deriving a further `Renderer` class. 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 actual rendering code, like OpenGL function calls in the `GLRenderer`. This approach ensures that the renderers can be extended with support for new kinds of geometric objects, simply by adding a new rendering method and the corresponding runtime check.

Unlike the OpenGL renderer, the `POVRenderer` does not provide real time graphics, but an interface to the external POV-Ray renderer. This is achieved by translating the data in the geometric objects to a form that can be parsed by the POV-Ray application. The resulting text is then written to an output stream, which is either a file or the standard console output. For further details about the POV-Ray exporter see Page 70.

2.6. Performance tuning

This section illustrates some of our efforts to optimize the performance of both, the visualization and the modeling features.

Hardware specific tuning

Several pieces of code in the VIEW library had further tuning potential if we could obtain detailed information about the hardware, on which it is run. Therefore, we added corresponding code that can for instance query the computer's amount of free memory. As an example, this information can be used to increase the resolution of regular three-dimensional data grids, which can improve the performance of geometric hashing algorithms for finding non-bonded-interactions (see Section 2.6.2).

A current trend in modern hardware design are multi-core and multi-processor computers. Thus, the information on the number of available CPUs is getting more and more important. Therefore, we added support for querying this information so that future versions of the VIEW library can determine the optimal number of parallel threads for updating models or subdividing calculations.

Benchmarking through Python scripts

For BALLView's development, it was important to have precise benchmark results because they allowed the comparison of varying implementations and hardware setups. Since BALL provides an interface to the Python scripting language and Python was ideally suited for these benchmarking tasks we wrote several such scripts. One script for example measures the OpenGL performance for all standard models and calculates a total score in so called "BALLView-GL-stones". This script is not only useful for the developers, but also for users, who want to compare the performance of their hardware setup with reference values. When we analyzed the results of this script for different computers, we came to the conclusion that BALLView's OpenGL rendering performance directly

corresponds to the power of the graphics accelerator cards, especially its clock speed. A corresponding script was written for measuring the performance of the model building algorithms. By using this script, we were able to identify and remove several bottle necks, and we could thus accelerate the model and coloring calculations.

Profiling

In addition to the benchmark scripts described above, we used several profiling programs (KCachegrind, Quantify, gprof) and could thus find and remove further performance problems. As an example, we were able to half BALLView's startup time. Without the usage of the profiling tools, this would probably never have been noticed.

2.6.1. Visualization

Models and coloring methods

Initially `ModelProcessors` and `ColorProcessors` were reapplied every time the model's atom positions changed. This was computationally expensive, especially in the course of MD simulations or when a user visualized a trajectory. To accelerate the update process, we developed the following approach. Some `GeometricObjects` like `Sphere` and `Tube` can have their positions specified by `Vector3` pointers, which can be set to the pointer of an atom's three-dimensional position. Thus, when the atom is moved, the corresponding `Sphere` in a *Stick* model still shares the atom's current position. This eliminates the need to recalculate the *Stick* model. Analogically, we were able to accelerate most `ColorProcessors` like the `ElementColorProcessor` which colors an `GeometricObject` with the color of the element of the corresponding atom. If only simple geometric objects like spheres or tubes have to be colored in such a way, and the hierarchy of the `Composite` tree did not change, there is no need to reapply the coloring. This results from the fact that under these conditions, a model's geometric objects are not destroyed and recreated in recurring updates. Table 2.4 gives an overview which models and coloring methods support the described techniques.

Mesh coloring through geometric hashing

For the most time consuming coloring tasks, the coloring of triangulated surfaces, the above described approach is not feasible. Even if only one single atom in a molecule moves, the whole set of vertices in a mesh must be recalculated. As a result, the mesh's coloring needs to be recalculated and the vertices have to be mapped to their corresponding (nearest) atoms with respect to their van-der-Waals radius. Since this has to be done at every step of a trajectory, an efficient solution was needed. To this end, we developed an algorithm that is based on geometric hashing. It applies the `BALL` class `HashGrid3`, which can store any type of data and supports arbitrary grid spacings. By using this grid

Models	Coloring Methods
Line	by element
Stick	by atom charge / distance
Ball and stick	by residue index / name / type
Van-der-Waals (VDW)	by secondary structure
Solvent-excluded/accessible surface	by chain / molecule
Ribbons	by forces
Backbone	by occupancy
Cartoon	by temperature factor
Hydrogen-bonds	by a custom color

Table 2.4: Some models and coloring methods could be implemented in a time saving way, such that they do not have to be reapplied when only the positions of the corresponding atoms have changed. These models and coloring methods are shown in a bold font.

class, we can divide the 3D space, containing the atom and vertices position into equally sized cubes. We then apply the constraint that a vertex and its corresponding atom can at most be a given distance away from each other. The exact value for this distance depends on the model type; for *SE surfaces*, we use 2.62 Angstrom, which corresponds to the estimated largest possible van-der-Waals radius. This maximum distance is used for the edge length of the individual cubes. Now, for a given vertex, the nearest atom can only be contained in the same cube as the vertex, or in the 26 direct neighboring cubes. Therefore, we can reduce the search space for finding the corresponding atoms and thus significantly accelerate the mesh coloring.

2.6.2. The force field calculations

One of the computationally most intense calculations in a force field (see Section 4) is the calculation of the pair list for the non-bonded interactions like the VDW and electrostatics components. This list contains all pairs of atoms that are close enough to interact with each other via non-bonded interactions. Since this list has to be updated whenever atoms are moved (e.g. in the course of an MD simulation or minimization), this calculation can be responsible for a good portion of the runtime. Therefore, any approach to accelerate the computing of the pair list is highly desirable. To this end, we implemented a new geometric hashing algorithm that is similar to the above described approach for coloring meshes. It also uses the `HashGrid3` class to compare the distances between the atoms in the individual grid cubes. But since the algorithm aims at finding all pairs of atoms with at most a given distance to each other and each pair has only to be added once, the algorithm can speed up the calculation. It was designed such that for each atom only half of the neighboring cubes must be taken into account, which almost doubles the algorithm's speed. In addition, the implementation was carefully tuned, for instance to

accelerate the testing for the grid bounds. As a result, the described approach scales much better than any of the earlier algorithms in BALL. Therefore, it is ideally suited for larger molecules with more than 1000 atoms. For smaller molecules, the brute force algorithm that simply compares all atoms with each other, is still faster, since it lacks the overhead for creating the grid.

2.6.3. Multithreading

Molview, the precursor of BALLView was designed as a single-threaded application. As a result, the graphical user interface froze during time intensive calculations like molecular dynamics simulations, and users could thus 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 our software to use multithreading techniques. Now all time intensive 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.
- Multithreaded calculations can be stopped with one single mouse click.
- The 3D graphics widget can show intermediate results, e.g. of a minimization run.
- Users can reposition the viewpoint to focus on a specific substructure.

Since multithreading has so many advantages, we used it for further purposes, like the (re)calculation of representations and for downloading structures from databases.

Locking data structures and synchronization of threads

Unfortunately, multithreading is one of the most complex fields in programming since the individual threads have to be synchronized:

"Granted, multithreading can lead to very elegant programs, especially when network access or long-running computations are involved, but you will not care much about this elegance when you bang your head on the wall because you cannot find the cause of a synchronization problem." [56]

Unfortunately, this citation particularly applies to GUI applications, as we experienced ourselves. The early multithreaded versions of our software had serious stability issues. As an example, users could modify or delete molecular structures, which were used in multithreaded calculations like an MD simulation, resulting in immediate crashes. Other frequent problems were deadlocks, when two threads competed for access on the same data or race conditions, when two threads depended on each other.

To solve these and other problems, we redesigned the library to use strict mutex locking. Only one modular widget at a time can obtain the exclusive read and write access to the molecular structures or representations. This is achieved by locking the molecular entities (Composite objects) by calling the following function:

```
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 free 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 are disabled. This 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 flow.

With these two interlocking mechanisms, the multithreading approach runs stable and it became one of the central techniques in the VIEW library.

Inter-Thread Communication

We had to consider a problematic aspect in the design of the Qt library: Only the main thread is qualified to modify GUI elements. This issue was addressed in the design of our library, by transmitting data between the different threads. We decided to use Qt's internal messaging system that allows one thread to send events that are then received in the main thread. The adequate class for passing user defined data is `QCustomEvent`, so we derived several classes from it, which can contain arbitrary data. This approach has the important advantage that it allows for delivering messages from the VIEW library (see Section 2.3.3) over thread boundaries. To this end, we derived the class `MessageEvent`, which can contain a `Message` instance. Thus, any thread can post such an event to the `MainControl`, where its `Message` is then sent through the conventional VIEW messaging system. As an example, the thread that calculates MD simulations notifies the main thread when it has computed a given number of steps with the following lines of code:

```
CompositeMessage* msg = new CompositeMessage(  
    composite, CompositeMessage::CHANGED_COMPOSITE);  
qApp->postEvent(main_control_, new MessageEvent(msg));
```

In the main thread, the `MainControl` then receives this event and sends the VIEW message to itself and all modular widgets:

```
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);
}
```

Example for the usage of multithreading

This paragraph illustrates the multithreaded calculation of a molecular dynamics simulation (see Fig. 2.7) together with the resulting update of the loaded representations. In this process, three different threads are used:

1. The main thread handles the VIEW messaging system, the GUI, and the Qt event loop and is thus responsible for the responsiveness of the graphical user interface.
2. An additional `UpdateRepresentationThread` which is derived from Qt's thread class `QThread`, calculates the visualization updates. This thread is created at the application's startup and performs a continuous loop until the program is closed. In every cycle of this loop, the thread queries the `RepresentationManager` if a `Representation` is to be updated. If so, it calls `Representation::update_()` and thus (re)computes the representation's model and coloring. When no update is needed, the thread sleeps for some microseconds to prevent unnecessary CPU usage until processing another cycle.
3. The computation of the MD simulation is performed in an instance of `MDSimulationThread`. This thread is created in the modular widget `MolecularStructure` when a user clicks on the corresponding menu entry. The MD simulation is performed in a loop (see below), such that intermediate results can be rendered. To this end, a `MessageEvent` containing a `CompositeMessage` is posted to the main thread, where it is received by the `MainControl`. As a result, the `RepresentationManager` marks the representation to be updated and the `UpdateRepresentationThread` will perform the model and coloring calculation. The `MDSimulationThread`'s main loop tests once per cycle if the user wants to abort the computation, by checking the return value of `MainControl::stoppedSimulation()`:

```
// iterate until done and refresh the screen
while (!main_control_ ->stoppedSimulation() &&
        mdsim_->getNumberOfIterations() < steps_ &&
        mdsim_->simulateIterations(steps_between_updates_, true);
{
    updateStructure_();
    waitForUpdateOfRepresentations_();
    ...
}
```

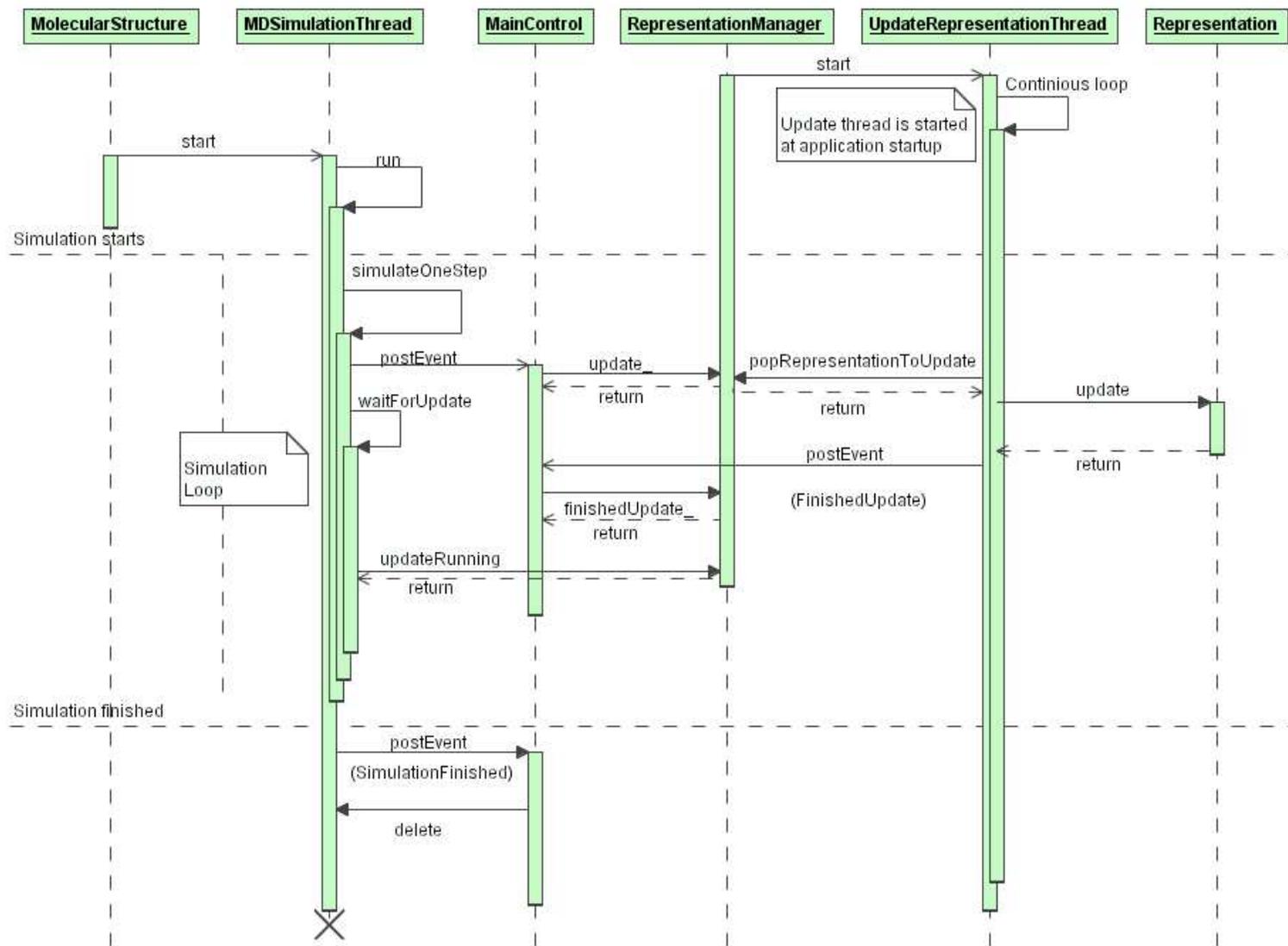


Figure 2.7: UML diagram for a multithreaded MD simulation. Three different threads are used. The main thread handles the GUI and the VIEW messaging system, the MDSimulationThread performs the molecular dynamics simulation, and the UpdateRepresentationThread recomputes all models for changed compounds.

2.6.4. Tuning the OpenGL rendering

Both, CPU's and GPU's performance, increased exponentially over the last years (Moore's Law) and will probably continue to do so for the next years. Thus, one might be tempted to think that molecular viewers no longer require highly optimized graphics performance. But, since the newly-published molecular complexes are getting bigger and bigger, a high rendering performance is still as crucial as in the beginnings of molecular visualization. Therefore, we optimized BALLView's rendering engine to achieve top performance.

Basic tuning

The original Molview program constructed backbone models with interleaving spheres and tubes. This created considerable overhead and was thus much slower than the triangulated backbone in the current BALLView version. Wherever feasible, BALLView now uses triangulated meshes, which have several advantages: Beside the better appearance and the faster rendering, these meshes can be drawn as wireframes or in toon mode (see Page 65), they can be colored according to scalar regular data grids (see Fig. 3.8), and they support additional tuning techniques (see below).

Quite small modifications in the OpenGL rendering code can often result in drastic changes in performance. Therefore, the source code for drawing the different geometric objects like spheres or meshes (see Section 2.5.1) was optimized. Furthermore, model specific changes were introduced. As an example, in the *Stick* model, the atoms with three or more bound partners are almost not visible and can therefore be drawn with less detailed spheres than the other atoms.

Culling

OpenGL provides many different techniques to accelerate the rendering and one of the most basic, but still most important ones is the so called "Culling". This means that the normally invisible backside of geometric shapes will not be drawn: OpenGL decides which parts of a shape is the backside by comparing all triangle normals with the direction of the view vector. Unfortunately, in BALLView it is often desired to view the inside of, e.g., a solvent-excluded surface, either because the view point lies in the molecule itself or because the user is using clipping planes (see Page 66). In contrast, for transparent surfaces and *Backbone/Cartoon* models, clipping is highly desirable. Therefore, the GLRenderer class decides whether clipping should be applied based on the representation's model and transparency. This increases the performance but still shows the backsides when needed.

Display Lists

Another technique for improved rendering performance are "Display Lists". Such a list contains a group of OpenGL commands which are processed and stored on the graphics accelerator card for subsequent execution. Thus, if an object has to be drawn more than once, the usage of Display Lists can significantly accelerate the rendering. In the VIEW library, Display Lists are used in two different ways to achieve best results: First, often occurring geometric objects, like spheres are rendered into a Display List. Second, every Representation is rendered into its own Display List. This approach showed to be very effective: When the scene has to be redrawn (e.g. because a user changed the viewpoint), the rendering is accelerated up to a factor of two.

Vertex buffer objects

Vertex Buffer Objects (VBOs) are similar to Display Lists, since they can accelerate the rendering process by storing data on the graphics accelerator card, but they are only suited for numerical data. Unfortunately, only recent version of OpenGL and graphics accelerator cards support VBOs. Therefore, we test if VBOs are available and if so, we use them for storing the vertices, normals, and coloring of meshes. This accelerates the drawing of meshes, about a further 30 percent compared to the Display List approach (depending on the graphics accelerator card).

Preview mode

The advanced techniques described above yield a significant improvement of the rendering speed. But, unfortunately even the most powerful hardware available today will not allow to fluently render the largest currently available structures with optimal quality. This is most often the case for *Ball-and-Stick* and *VDW* models which can contain large numbers of spheres and tubes corresponding to huge numbers of vertices and triangles. On slower computers, this will prevent the fluent rotation of a molecule. To circumvent this problem, we added an optional preview mode. It renders the stick and sphere based models with less detail and without anti-aliasing (see below) while a user is modifying the scene or moving a molecule. When the user releases the mouse button and thus ends such an action, the models are redrawn with their full level of detail.

Improving the rendering quality

To achieve a more appealing visualization, BALLView can use two different anti-aliasing methods, when available. First, OpenGL can be told to try anti-aliasing for lines and polygons by calling:

```
glEnable(GL_LINE_SMOOTH);  
glEnable(GL_POLYGON_SMOOTH);
```

Since the polygon smoothing was not thoroughly convincing and resulted in problems on some machines, we only enabled line smoothing.

A newer and more powerful technique is called supersampling. It works by rendering the image at a much higher resolution than the one being displayed, and then shrinking it to the desired size, using the extra pixels for a smooth interpolation. In BALLView we enable this OpenGL feature via the Qt library. The results are high quality images, which can almost compete with the results of the POVRay renderer. Unfortunately, many graphics accelerator cards do not support supersampling for off-screen rendering (which will be described in Section 3.2.5).

2.7. Quality assurance

2.7.1. Verification

Since we have to ensure the stability of our software on multiple platforms and with different compilers, quality assurance and verification play a significant role in BALLView's development. Therefore, we use several complementary methods to verify our source code:

- To verify BALLView's basic functionality we performed unit tests with BALL's integrated testing framework (see [73]).
- The implementation of the MMFF94 force field was validated by a specially written test program. It compares our results with the values from the MMFF94 validation suite [22], which roughly consists of 750 molecules and 17,000 atoms. For all individual atoms, the assigned types, constants, and calculated energies were compared to the reference values. This ensures maximum consistency with the original implementation.
- To further improve the stability, we used several programs (Rational Rose [39], Purify [34], valgrind [44]) to verify our source code. This led to the detection and solving of many flaws in the implementation. We could remove several memory leaks, uninitialized members, and invalid memory access through invalid pointers and iterators. In addition, the above programs were used to analyze the code coverage.
- Where appropriate, we used Python scripts for automatic testing on large data sets. One such script tested our implementation of the DSSP algorithm [70] for assigning secondary structures and the *Cartoon* model processor. For all entries in the protein database [49], the script calculated a *Cartoon* model before and after reassigning the secondary structures. Occurring crashes were logged for finding stability

problems. In addition, the script created a screenshot for every individual molecule. From the resulting images, several hundred were chosen randomly and manually checked for potential problems.

- We performed extensive manual tests with the application on all supported platforms. This enabled us to find crashes and other problems that could not be found with any of the other described methods.

2.7.2. GUI testing

The methods described in the last section have a common drawback: they do not allow for automatic testing the graphical user interface. Thus they can only incompletely test for problems that arise from the interplay of the individual components. To circumvent this limitation, we had to perform automatic GUI testing. Unfortunately, all available applications turned out to be inappropriate for the task since none of these programs had support for Qt 4.0 and higher. We thus decided to implement our own GUI testing framework. It is divided into three distinct phases. First, BALLView allows to record any user input, both from the mouse and the keyboard. This input is then serialized in a line based format that stores one input event (e.g. mouse click) and the corresponding time stamp per line. Second, the resulting files can be modified manually to incorporate consistency checks. These checks can be composed of any valid Python code and thus access the full functionality of the BALL and VIEW libraries (see Section 3.4). Therefore, very detailed query checks can be performed, for instance for the state of the loaded molecules or visualization objects. To illustrate the usage of these tests, we wrote the following example line:

```
T|0.1|len(getMainControl().getSelection())|1|
```

The first field "T" defines that this line corresponds to a test and is not, for instance a user input event. Next follows the time in seconds to wait until the test is performed and the Python code. In this example it tests for the size of the current molecular selection. Finally, the last field contains the expected result for the test. It is string-wise compared with the result of the Python command. If the two strings differ, this is written to the standard output and into a logging file. Third, BALLView can be started with the resulting file from step 2. Now, all previously performed user inputs are simulated by using Qt's internal QTestLib framework. Unfortunately, this framework does not support the (de-)serialization of the events, so we had to write the corresponding code ourselves. Here, the main problem was the identification of the correct widget which shall receive, for instance a mouse event. To uniquely identify a widget, we decided to use the QObject tree. QObject is a base class for every Qt GUI element and each QObject has exactly one parent, with the QMainWindow as the top parent, resulting in a tree-like structure. In

addition, every QObject can have a string as its name. Thus, if all QObjects only have children with differing names, we can uniquely identify each widget, by storing its name and the name of all of its ancestors in the tree. Next, we had the problem that the size of the individual GUI elements can differ as a result of the GUI styles and user defined font sizes. This is problematic, since we need the exact screen positions for the simulation of mouse events, for instance for pressing entries in the QTreeWidgetItem in the hierarchical overview of molecules. To enforce that the GUI at runtime is consistent with the state at the recording time of a test file, the test file supports the loading of BALLView's project file. Since this file include all settings that influence the layout of the GUI elements we can thus fully restore the GUI settings. As a result, test files can also be correctly be replayed if they were recorded on a differing platform or with an other Qt version. Thus, we were able to create a platform-independent GUI testing framework. In addition to the usage for automatic testing it can furthermore ease the bug-tracking process: user that experience strange behavior in BALLView can record a corresponding test file and attach it to a bug report. The developers can then replicate the error, simply by starting the test file. Furthermore, the described functionality also allows to record and playback user input for other purposes: users can for instance prepare a presentation or tutorial for later usage.

2.7.3. Usability testing

To ensure that, despite its rich functionality, BALLView remains intuitively usable, we repeatedly performed usability studies. To obtain results from users with differing levels of experience, we made studies with individual researchers that had large experience in molecular modeling, small groups of undergraduate students, and even pupils that had never seen a molecular viewer before. The users were only given a short introduction and were then asked to perform standard tasks like opening molecular files, creating representations, and editing molecules. In addition to the comments of the users in and after the tests, we carefully analyzed their course of action. As a result, we were able to find potential problems and optimize the usability. For example, we redesigned multiple dialogs in order to simplify their interface. Several times, we also tested different alternative implementations. For example, we implemented three algorithms for the placement of clipping planes and tested them with several users until the most intuitive handling was found.

2.8. Comparison with other visualization and modeling frameworks

The BALL and VIEW libraries were designed as extensible software frameworks, such that they can serve as basis for applications in both molecular visualization and modeling. Only few software libraries with a comparable scope exist and all of them have serious drawbacks:

- **Chimera** [10] is supposed to be extensible, but it is closed source and does not support the insertion of new methods or members into its molecular classes. For these reasons, it is not suited for large scale extensions.
- **Ghemical** [13] is a computational chemistry package with an OpenGL interface that is released under a free GPL license. While it has interesting molecular mechanics and visualization features, it lacks any serious documentation of its classes.
- **JOELIB** [18] is a bioinformatics library, written in Java that has a wide range of features. Unfortunately, its documentation is very sparse and does not give any details on how to use the individual classes and their methods. Furthermore, it provides only very limited visualization capabilities.
- **MOE** [23] is based on its own novel programming language, the Scientific Vector Language (SVL). While this language may be powerful and efficient in terms of runtime, its usage is limited to this application. MOE therefore requires users to learn and master its own proprietary language. In addition, it is very expensive and the source code is not available, so developers cannot freely modify the application. Thus, it is less suited as a basis for own development projects.

3. Features and applications

This chapter gives an overview of BALLView's functionality, but due to its wide range of features, this overview can only be very perfunctory. Therefore, we concentrate on the presentation of the major features in the different domains. First, the graphical user interface is described, including its design, layout, and features. Next, the visualization and molecular modeling capabilities are discussed, along with some examples for the different techniques. Finally, the overview ends with the description of the Python scripting interface, its graphical frontend, and the Integrated Developer Environment functionality.

3.1. Graphical user interface

In contrast to most other molecular visualization and modeling tools, BALLView provides a state-of-the-art graphical user interface that complies with modern user-interface design principles. The appliance of these principles makes our software more user-friendly. As a result, new users can get easily acquainted with the application, while experienced users have access to BALLView's rich functionality which may increase their productivity. This section will describe the architecture of the graphical user interface and the design principles that were applied.

3.1.1. Architecture

BALLView's graphical user interface mainly consists of a set of independent modular widgets (see also Section 2.3.2). The most important of these widgets are shown in Figure 3.1. Each widget encapsulates a different domain of features:

1. The `MolecularControl` is a hierarchical list of all loaded compounds. It provides detailed information about the structures and supports many standard tasks like copying, deletion, and selecting substructures.
2. The `GeometricControl` lists all representations and allows for their manipulation. Users can for example customize representations or switch them on and off.
3. The `DatasetControl` lists data sets like electrostatic potential grids, docking results or trajectories.

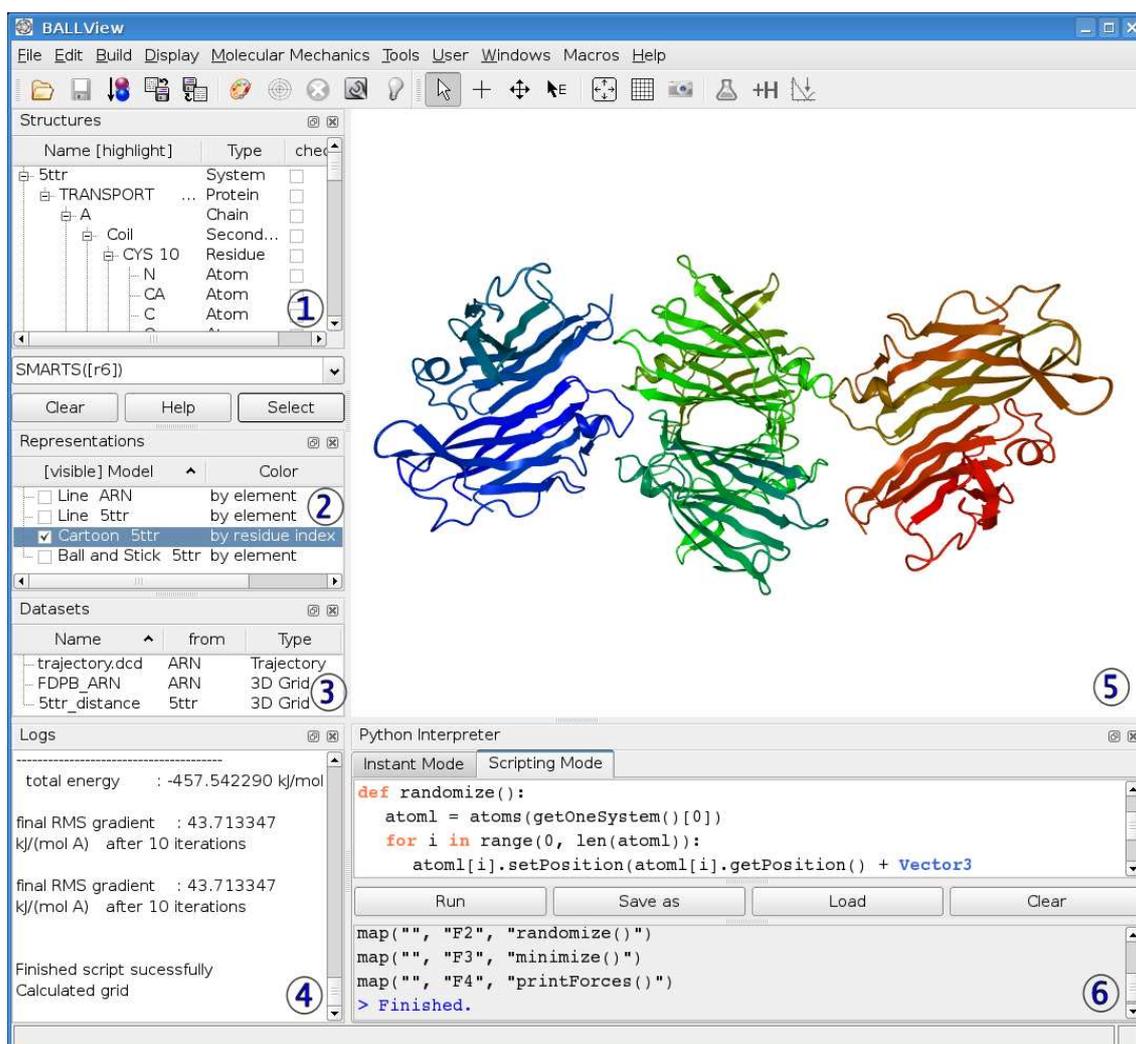


Figure 3.1: BALLView's graphical user interface consists of independent widgets that can be combined at the users discretion to form an application. They can be freely placed, docked in and out of the main window and switched on and off.

4. The LogView provides access to the logs.
5. The Scene is a 3D view of the currently active models.
6. The PyWidget contains the graphical user interface to the Python interpreter.

In addition to the visible widgets, several hidden ones provide menu and toolbar entries, e.g. for molecular modeling features or database access. Users can freely place the visible widgets, undock them from the main window and even switch them on and off. As a result, the graphical user interface is highly configurable and can therefore provide the optimal layout for very diverse tasks:

- For molecular modeling, a user can employ the `MolecularControl` and the logging widget and thus run energy minimizations.

- For mere visualization tasks, the GUI can be reduced to the fullscreen 3D graphics widget.
- Users that develop new Python scripts in the Integrated Developer Environment can switch off all widgets except for the BALL class documentation and the Python widget.

3.1.2. Usability

Usability can be defined as "the ease with which a user can learn to operate, prepare inputs for, and interpret outputs of a system or component" [89]. We worked hard to ensure that our software can easily be operated, even by inexperienced users. Therefore, BALLView not only supports all standard features of graphical interfaces, but we also developed new ways to assist the user. The following pages will describe our efforts to achieve a high usability.

Installation

Apparently, an application's ease-of-use already begins with the installation procedure. For a long time, installing BALLView was quite complex, since the entire BALL library had to be configured and compiled first. Since this is very time consuming and complicated for inexperienced users, we have created prebuilt binary packages for most of the common platforms, like Debian, Ubuntu, Mac OS and Windows. For users that need to build the library and application from the sources, we developed automatic installers that ease this process significantly.

Standard interface elements

By using the Qt library, BALLView has support for all standard widgets, like text editor fields or comboboxes. In addition, different display themes can be used, including the native styles of the different operating systems and window managers like Windows, Mac OS, and KDE.

Effective layout

The layout of dialogs and widgets was done according to standard norms and heuristics [86, 90]:

- **Keep it simple:** We tried to design all user interfaces with as few elements as possible.
- **Grouping:** We used group boxes or sub menus to arrange elements with similar functionality where appropriate.

- **Consistency:** We used the same vocabulary, syntax and styling in all dialogs and widgets. Furthermore, similar dialogs have similar layouts.
- **Predictability:** The labeling of BALLView's user interface elements provides clear information on what will happen if they are used.
- **Provide clear exits:** We added clearly marked buttons to cancel dialogs and abort actions. This prevents users from being "trapped" in a task or dialog.
- **Access aid:** Through direct observation, we learned how important access aid can be for handicapped people that for instance have problems using a mouse. Therefore, we ensured that all dialogs can be navigated with the "Tab" key.
- **Default actions:** Most dialogs have a default button that is especially marked and can be quickly activated by pressing the "Return" key.
- **Informative error messages:** We made all error and warning messages as understandable as possible and suggest possible solutions.
- **Allow only legal inputs:** Where appropriate, we used Qt's feature to allow only valid inputs, for example in the form of integer or floating point numbers. As an alternative we used slider widgets that can limit the possible input values between a sensible minimum and maximum value.

Fast access to all features

We made sure that BALLView's functionality is quickly available through corresponding menu entries. In addition, we added toolbar entries for the most important and commonly used features like opening molecular files, downloading PDB files, or building representations. Furthermore, most of BALLView's features are also available through keyboard hotkeys. This can be faster and more convenient than using the mouse (e.g. when changing the modes in the 3D view). To allow for the creation of user-defined keyboard shortcuts and menu entries, we developed a unique feature that no other molecular software tools provides. Users can write their own Python scripts and map them to hotkeys (see Section 3.4). From these hotkeys, the application then automatically creates corresponding menu entries. This user-defined shortcuts allows a noticeable acceleration of the workflow when performing repetitive tasks.

Navigational jumps

Where appropriate, we used navigational jumps from one dialog to another. As an example, the dialogs for starting molecular dynamics simulations or energy minimizations contain a button to setup the chosen force field through another dialog. While this dialog can also be opened from BALLView's main menu, the described approach allows faster

access to this functionality.

Since we use HTML as basis for BALLView's help system, the documentation allows for jumping to individual elements (see below).

Responsiveness

Since we implemented time intensive computations in separate threads, BALLView's graphical user interface stays responsive while performing MD simulations or calculating complex visualization. Moreover, the multithreading approach allows to cancel simulations and progress information can be shown with the estimated remaining runtime of the current task.

Continuous feedback

We wanted to provide users with a continuous feedback on any ongoing task. Therefore, we added a status bar at the bottom of the main window that shows hints, warnings, and error messages. In addition, important messages get logged in a special widget to allow for later examination.

As another way to provide feedback, we use several distinct mouse cursors that indicate the current mode or inform the user when the application is busy. Furthermore, BALLView's menu and toolbar entries are enabled/disabled when their preconditions do / do not hold. Thus, users are always informed about the actions that can be performed at a given time. If the mouse cursor is placed on a disabled menu entry, the user is informed why this feature is currently unavailable.

Integration into the window manager

BALLView can be integrated into most common window managers for the registration of file formats or drag-and-drop operations, for opening structures or Python scripts.

Adjustability

We designed BALLView to be highly configurable, i.e. almost all features, including the different models, coloring methods, and display settings, can be adjusted to the user's liking. To ease the setup process and to provide a common interface for the individual preferences, we designed a comfortable dialog (see Fig. 2.5). It clusters the options in hierarchical groups and allows to freely browse and apply the individual settings. (For the implementation details see Section 2.4.3.) Furthermore, the 3D graphics view supports adjustable mouse sensitivities. In particular, disabled people can thus benefit of low mouse sensitivities.

Since all configuration options are stored when BALLView is closed, it has the same look and feel when it is restarted. This includes the size, placement, and coloring of the wid-

gets as well as the options for the different models. Therefore, users can adapt BALLView to their liking once and keep their customized working environment.

Project files

After elaborately composing a complex scene, users often want to save their work for later usage. Therefore, we implemented a special file format for storing BALLView's settings along with the loaded molecules and models. This enables users to store a specific setup, e.g. for a presentation. In addition, users can perform a "quicksave" or "quickload", which writes/reads a project file named "quick.bvp" in the user's home directory. This can serve as a kind of undo/redo functionality. For faster access, the "quicksave" and "quickload" functionality is accessible through toolbar entries.

Automatic contextual help

We worked very hard to make BALLView as easily usable as possible, however, users may sometimes still have difficulties apprehending the function and usage of individual interface elements. As an example, it may be difficult to understand why a given button is disabled or what will happen if it is pressed. A common way to support users in such situations is automatic contextual help through tooltips. These labels pop up when the mouse cursor rests for a certain time over an interface element. A conceptual problem of tooltips is the length of this time. If too short, users get annoyed by undesired popups, while longer times make users wait. BALLView in general provides tooltips, but we also implemented an other approach. The status bar instantly shows a short description when the mouse cursor is placed on a menu entry. If the entry is disabled, the description tells why. In addition, more extensive documentation is available through a special help mode (see below).

3.1.3. Documentation

The effect of a detailed documentation on a software's usability can not be overestimated. Therefore, we invested considerable time and effort into a well-elaborated documentation for the BALL and VIEW class libraries as well as BALLView's graphical user interface.

Documentation for the framework

Similar to the BALL framework, the visualization library contains the documentation for its class interface in the header files. This approach allows for keeping the documentation consistent with the current implementation. To extract the documentation out of the source code, we use the tool "doxygen" [12], which parses the class declarations as well and can thus automatically include information on class hierarchies, member protection, parameters, and return types. Since doxygen supports the creation of HTML output

(see Fig. 3.2) and since BALLView provides an integrated HTML viewer, the class documentation is available in the application itself. This significantly eases the development of Python scripts in BALLView (see Section 3.4), since the class documentation can be shown next to the Python editor. In addition to the integrated documentation viewer, we provide the HTML files on the project's website for online access.

[Main Page](#) | [Modules](#) | [Namespace List](#) | [Class Hierarchy](#) | [Class List](#) | [Directories](#) | [Namespace Members](#) | [Class Members](#)

BALL::VIEW::MainControl

VIEW::MainControl Class Reference

[Widget connectivity and message queuing]

MainControl is the main administration unit for a program and must be used by all applications. [More...](#)

```
#include <mainControl.h>
```

Inheritance diagram for VIEW::MainControl:

```

classDiagram
    class BALL_VIEW_ConnectionObject["BALL::VIEW::ConnectionObject"]
    class BALL_Embeddable["BALL::Embeddable"]
    class BALL_VIEW_MainControl["BALL::VIEW::MainControl"]
    BALL_VIEW_MainControl --|> BALL_VIEW_ConnectionObject
    BALL_VIEW_MainControl --|> BALL_Embeddable
    
```

[List of all members.](#)

Automatic module registration, menu construction and preferences handling.

QAction * insertMenuEntry (Position parent_id, const String &name, const QObject *receiver=0, const char *slot=0, QKeySequence accel=QKeySequence()) throw () <i>Insert a new menu entry into menu ID (creates a new menu if ID not existent).</i>
void removeMenuEntry (Index parent_id, Index entry_ID) throw ()
virtual QMenu * initPopupMenu (int ID) throw () <i>Initialize a new popup menu ID.</i>
void insertPopupMenuSeparator (int ID) throw () <i>Insert a separator into the popup menu ID.</i>
virtual void applyPreferences () throw () <i>Apply all preferences.</i>

Figure 3.2: Example for the class interface documentation in HTML format. It supports easy navigation between the individual classes.

For developers, the BALL library provides a tutorial for the installation of the library, the class interfaces and modules. We added further chapters describing the extensions of the viewer with new widgets and instructions on how to add new geometrical primitives to the rendering engines.

Documentation for BALLView

We provide additional documentation describing the usage of BALLView's graphical user interface. This documentation written in HTML format is available via the project's website and integrated in the application (see Fig. 3.3). Furthermore, the usage of HTML has the advantage that different parts of the documentation can easily be linked together.

The documentation also contains a quick reference of the possible actions and the corresponding keyboard shortcuts.

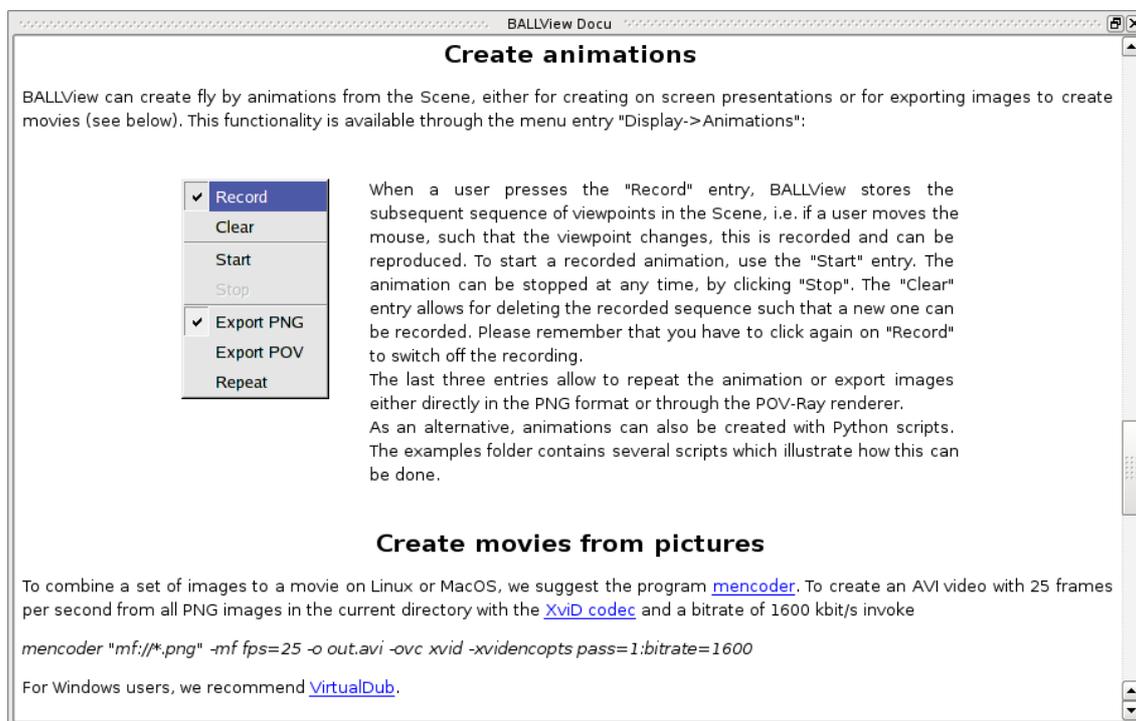


Figure 3.3: Example for the BALLView documentation shown inside the program itself.

Context sensitive help

While developing BALLView, special care was taken to prevent long familiarization phases for inexperienced users, for instance by providing context sensitive help (see Page 52). Further detailed information for individual elements in the graphical user interface is provided via a special help mode. It is available by the "What's this?" entry in the "Help" menu. To inform the user that the application has entered this mode, the mouse cursor is transformed into a question mark. While the mode is active, a left mouse click on any widget will open the corresponding entry in the documentation (see Fig. 3.3). To leave the "What's this?" mode, the right mouse button or the "Escape" key can be used. As an alternative, the "F1" key will pop up the documentation of the widget at the mouse cursor's current position.

Demo and tutorial

To familiarize new users with BALLView, it provides an integrated demo and a step-by-step tutorial that describes the essential interface elements, their usage, and some basic concepts with simple examples. Hopefully, this will motivate users to continue learning BALLView's user interface on their own.

3.2. Visualization functionality

BALLView was designed as a molecular viewer and thus provides numerous visualization capabilities. It supports all standard molecular models and features sophisticated methods for displaying electrostatics potentials. To render this data in realtime, BALLView contains an integrated OpenGL engine. However, BALLView not only offers this internal renderer. As an alternative, users can export their scenes to an external renderer like POVRay and thus create very detailed images. As already discussed, this functionality is also available through the underlying VIEW framework which allows for the realization of own visualization applications.

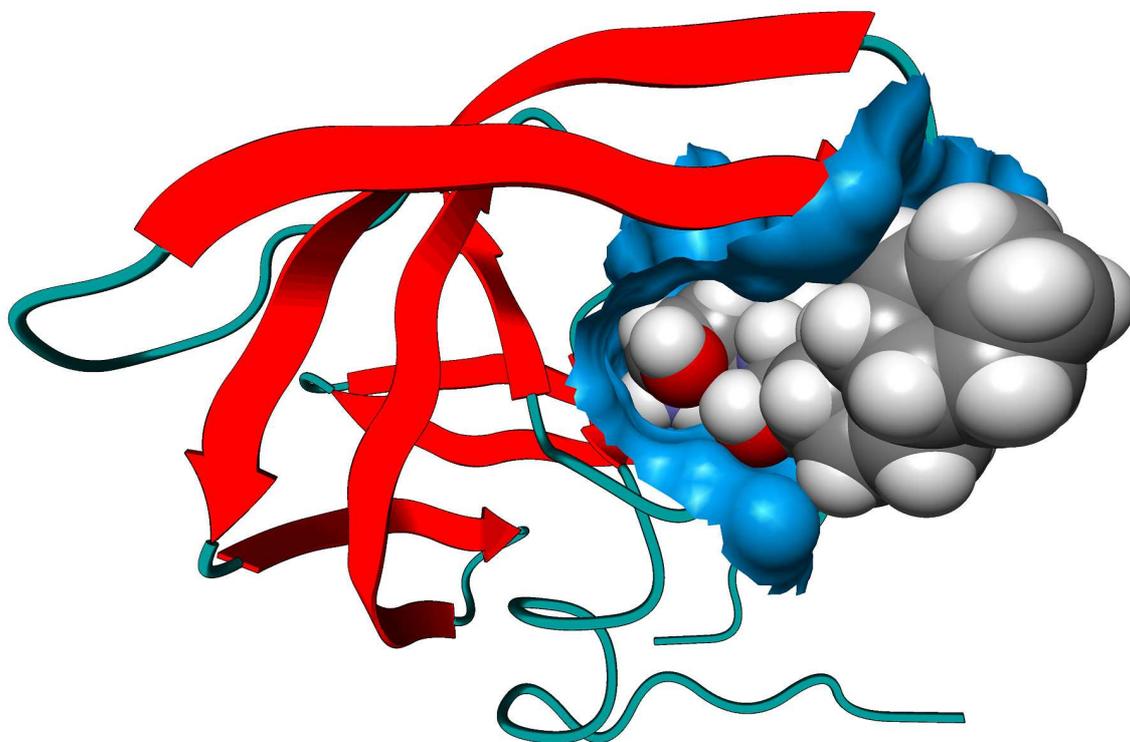


Figure 3.4: Visualization of HIV protease with an inhibitor (PDB Id 1FB7). It shows a *Cartoon* model of the protease in combination with a *van-der-Waals* model of the inhibitor. Additionally, a solvent excluded surface was split in a given range around the ligand to visualize the binding pocket. These three different models correspond to three "Representations" (see Section 3.2.1).

This section will present the underlying ideas and features of the visualization functionality, beginning with the concept of a "Representation", the supported molecular models and coloring methods. Then the different options to render electrostatic potentials (or any other kind of scalar data grids) are explained. Next follows a description of BALLView's

integrated OpenGL rendering engine, including its features and supported techniques. Finally, the section presents several ways to create image files and movies.

3.2.1. Representations

To offer an intuitive way of handling individual models and their coloring, we designed the concept of a Representation, which corresponds to one entity of visualized objects. The advantages of this concept have been discussed in Section 2.5.1. A Representation stores the user defined selection of the considered molecular system, the model, coloring method, and drawing style that are applied. After the model and coloring have been created, a Representation also contains the geometric objects representing the model (see Fig. 2.6).

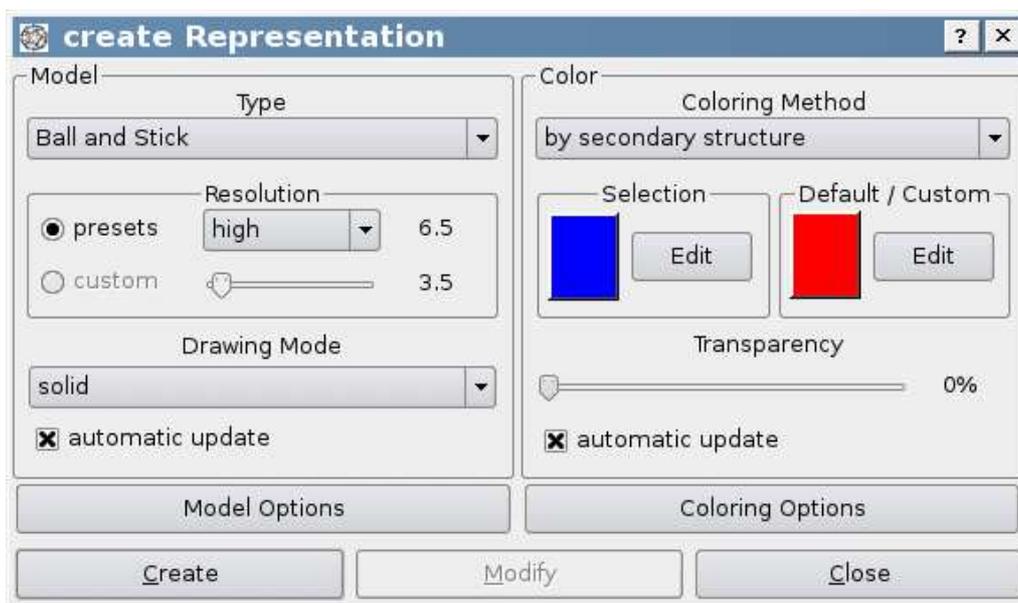


Figure 3.5: The DisplayProperties dialog was designed to simplify the creation and modification of a Representation by providing direct access to all available options. Users can freely choose and combine the different models, coloring methods, levels of detail, drawing modes, and transparency.

Since the individual representations are independent from each other, users can easily construct very complex scenes with varying models for different functional groups. As an example, Fig. 3.4 shows the visualization of a ligand in its binding pocket. To simplify the creation and modification of representations, we designed a user-friendly dialog that allows for the assignment of all the different settings (see Fig. 3.5). As a result, the application of a model to a given selection just takes a few mouse clicks.

3.2.2. Molecular models and colorings

BALLView allows for visualizing complex molecular scenes along with additional information. It provides all standard molecular models (see Table 3.1) that are available in modern molecular viewers and many different coloring methods. The details of the different models will be discussed on the following pages.

Models	Coloring Methods
Line	by element
Stick	by atom charge
Ball-and-stick	by atom distance
Van-der-Waals (VDW)	by residue index
Solvent-excluded surface	by residue name
Solvent-accessible surface	by residue type
Backbone	by secondary structure
Ribbon	by chain
Cartoon	by molecule
Hydrogen-bonds	by forces
Forces	by occupancy
	by temperature factor
	by a custom color

Table 3.1: Molecular models and coloring methods in BALLView. All models and coloring schemes can be freely combined and applied to arbitrary subsets of atoms.

- **Line models** allow the visualization of large molecules even on low end machines.
- **Ball-and-stick models** offer detailed information on bond orders and aromaticity. As an alternative, this model can be reduced to a `Stick` model.
- **Hydrogen-Bond models** allow deeper insights into the mechanism that form a molecule's secondary structure.
- **Forces models** are very useful in MD simulations and minimizations to provide information about suboptimal placed atoms.
- **Surfaces** are among the most important models implemented in BALLView and come in three different definitions: Solvent accessible and solvent excluded surfaces (SES/SAS) can be computed with adaptable probe radius and degree of triangulation. Regularly spaced data grids can be used to calculate isocontour surfaces, e.g. for visualizing electrostatics (see Fig. 3.15). In addition, these data grids can also be used for coloring all kinds of surfaces (see Fig. 3.8). Furthermore, BALLView allows for reducing surfaces to patches, for instance in vicinity of a ligand. For an example see Fig. 3.4.

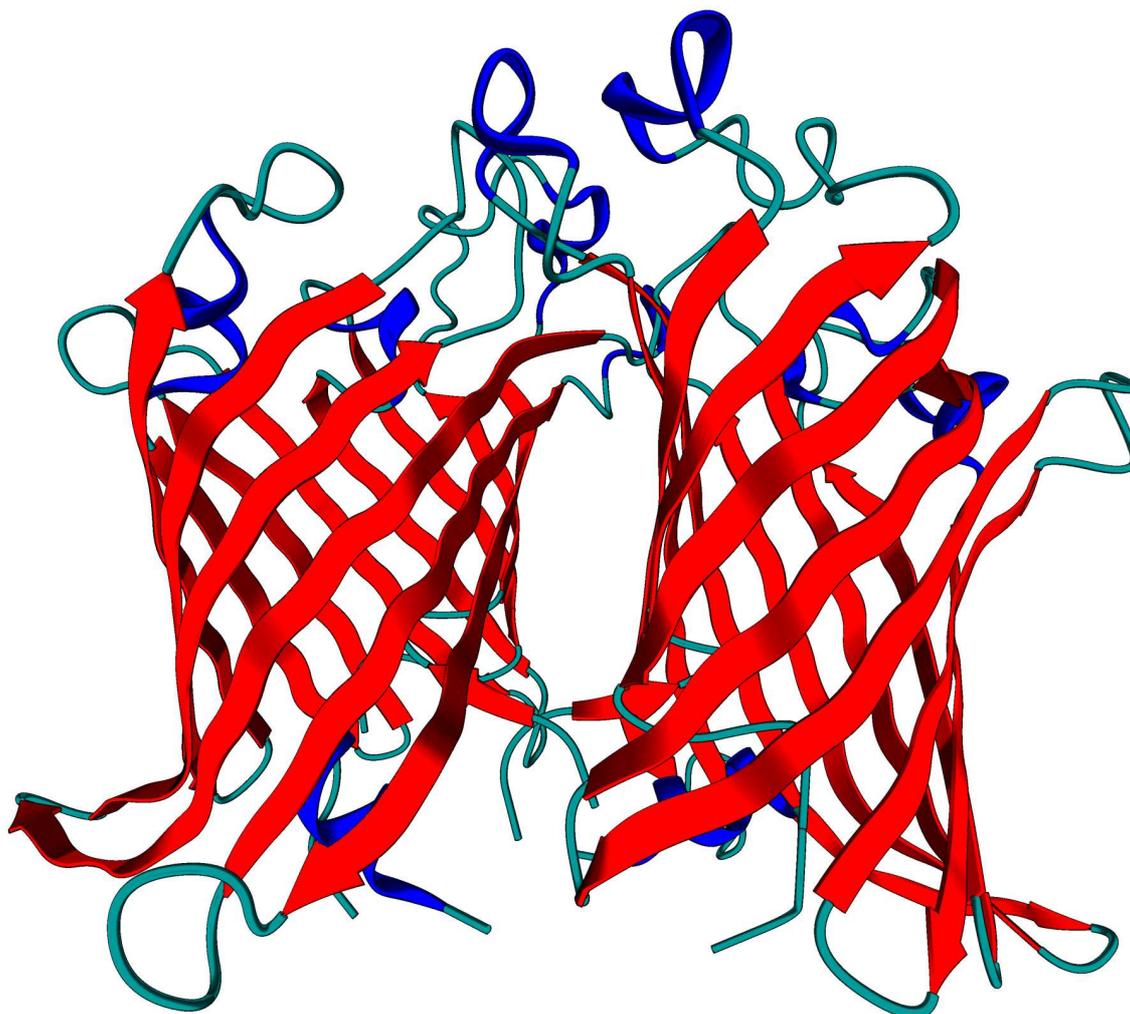


Figure 3.6: Example for a *Cartoon* model. Helices are colored blue, sheets red, and turns green. This image was rendered with the Toon Shader mode and offscreen rendering (PDB Id 1QD6).

- **Visualization of secondary structures**

For the visualization of secondary structures, we implemented a *Cartoon* model that supports proteins as well as nucleic acids. For proteins, it shows the three most important secondary structure elements. Helices are shown as ribbons, Sheets as arrows, and Turns as tubes (see Fig. 3.6). For nucleic acids, we provide two different models. The first, the so-called *Ladder* model, is rather simple. It renders a nucleic acid's phosphate backbone as a continuous tube and creates one cylinder for every individual base. The second model was inspired by the original hand-made wireframe model of Watson and Crick [103]. The phosphate backbone is shown as a ribbon, while the sugar residues and individual bases are abstracted by small plates (see Fig. 3.8).

Since the *Cartoon* model provides an abstracted view, it is often better suited for il-

lustrating a molecule's characteristics than any other model. This especially applies for large molecules with thousands of atoms, where most other models are simply too packed.

- **Labels**

When creating molecular visualizations, users often wish to add labels to individual atoms or functional groups. To meet this need, we added support for automatic and persistent labels, which can contain the name, charge, or type information of individual atoms (see Fig. 3.7). If the respective values of the atoms change, the labels are automatically updated. Of course, it is also possible to create labels with user defined text.

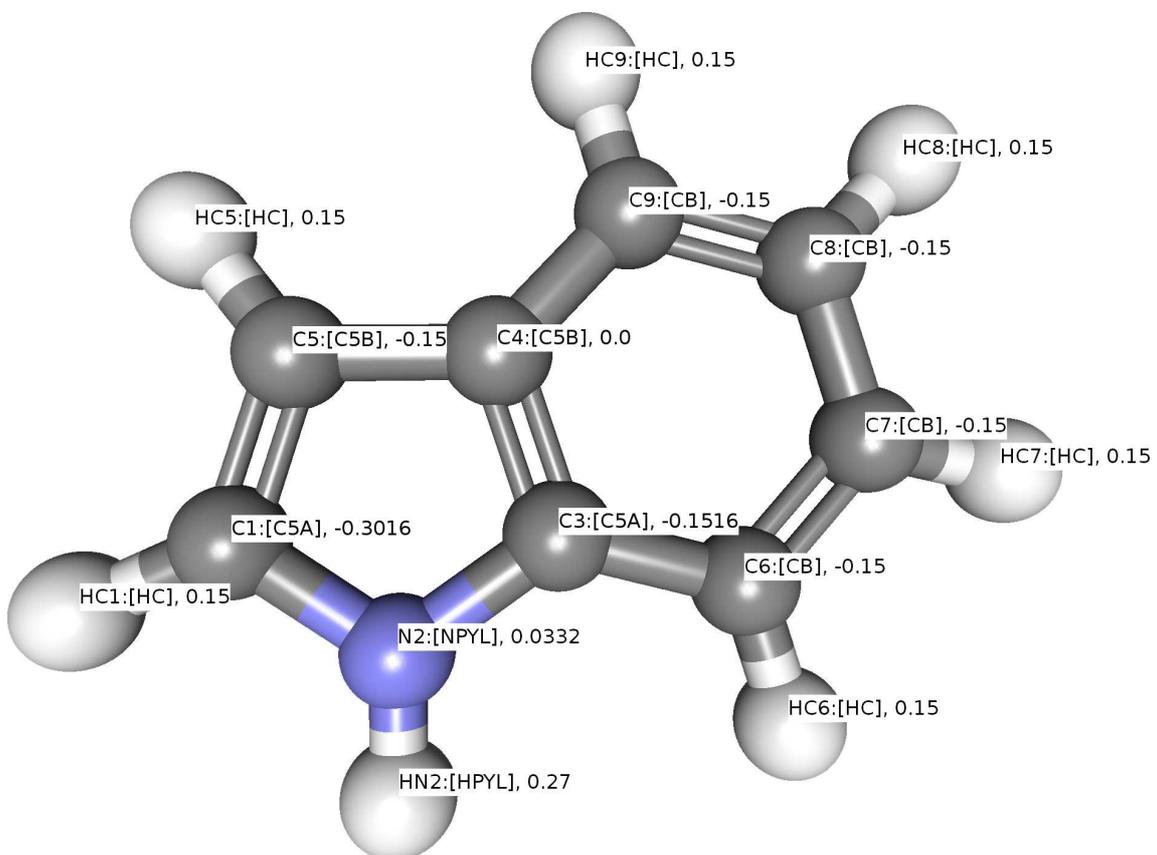


Figure 3.7: Labels can be created for individual atoms as well as entire residues and chains. The figure shows the automatic labeling of atoms with their names, types and partial charges.

3.2.3. Visualization of electrostatic potentials

Electrostatic interactions play a crucial role in protein folding and stability, enzyme catalysis or protein-protein recognition. In contrast to most other molecular modeling tools,

BALLView provides a rich functionality for the calculation and visualization of electrostatic potentials, similar to tools like GRASP [85] and DelPhi [84]. Potential grids can either be read from external sources or calculated through the integrated Finite-Difference Poisson Boltzmann (FDPB) solver (see Section 3.3.5). The resulting potential grids can be visualized in a variety of ways, which will be illustrated in the following paragraphs. All these methods can also be applied to any other kind of scalar data grids.

Coloring of surfaces

One of the most common methods to visualize a molecule's electrostatic potential is the coloring of the molecular surface (see Fig. 3.8). For this purpose, BALLView features a comfortable dialog (see Fig. 3.9) which allows for the assignment of all colors and interpolation values. Aside from molecular surfaces, all other kinds of triangulated meshes (like *Cartoon* models) can be colored in the same way.

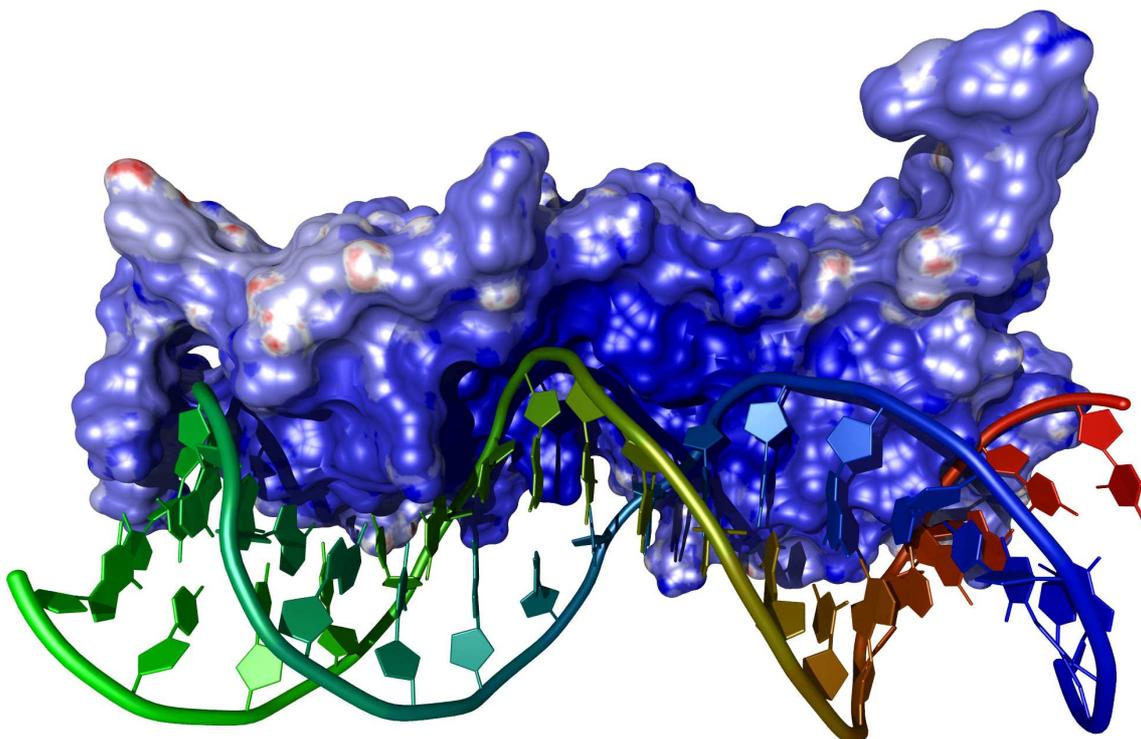


Figure 3.8: Electrostatic potential of the DNA-binding domain of the glucocorticoid receptor. The receptor is shown via its solvent-excluded surface colored by its electrostatic potential (blue: positive potential, red: negative potential). The potential was computed using BALLView's integrated FDPB solver. The DNA is drawn as a *Cartoon* model showing the individual bases, sugar residues, and the phosphate backbone in a schematic representation (PDB Id 1GLU).

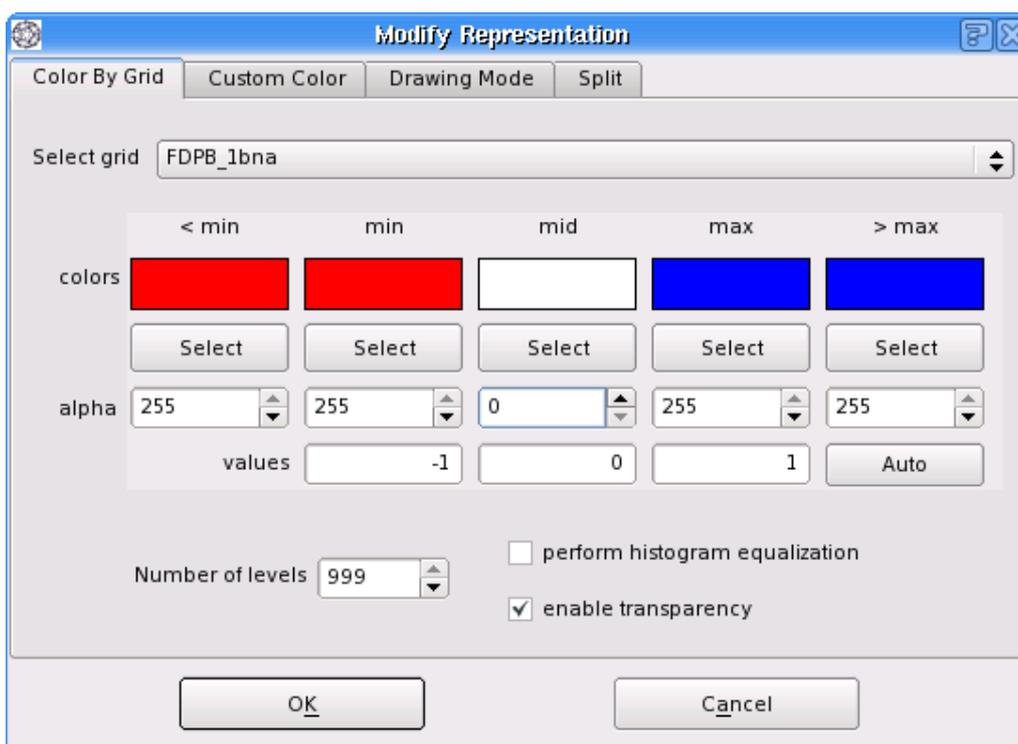


Figure 3.9: Dialog for coloring meshes by scalar grid data.

Isocontour surfaces

Another approach to visualize scalar grid data are isocontour surfaces (also called isosurfaces), which show all the points in a data set with the same given value. The computation of these isosurfaces is available through BALL's internal "Marching Cubes Algorithm". To start such a calculation, a user only has to open the corresponding dialog and enter the desired value and color. While isosurfaces are supported by a variety of modeling tools, BALLView provides advanced features for isocontour surfaces.

- Users can freely re-colorize isosurfaces or change their rendering mode. For instance, surfaces can be rendered transparent or as wireframe to allow a view inside a molecule.
- To perceive the three-dimensional gradient of a molecule's electrostatic potential, it is possible to combine multiple isocontour surfaces and slice them with a clipping plane (see Fig. 3.15).
- A Python script that is part of the actual BALLView installation allows for the creation of movies to visualize the electrostatic potential. Between a lower and upper value, contour surfaces are calculated and shown one by one. This provides a good overview of the potential's gradient. Such a movie is available for download at the project's website [4].

Volume Rendering

The capabilities of modern graphics accelerator cards to render three-dimensional textures allow for new innovative methods for the visualization of scalar grids. BALLView now supports Volume Rendering that can be considered as a fog with color values and density according to the values of the scalar grid. To achieve this effect, we render several successive planes that are colored by a texture representing the grid values. Due to the missing depth perception in a static view, a figure as 3.10 can only give a glimpse on this feature's capabilities.

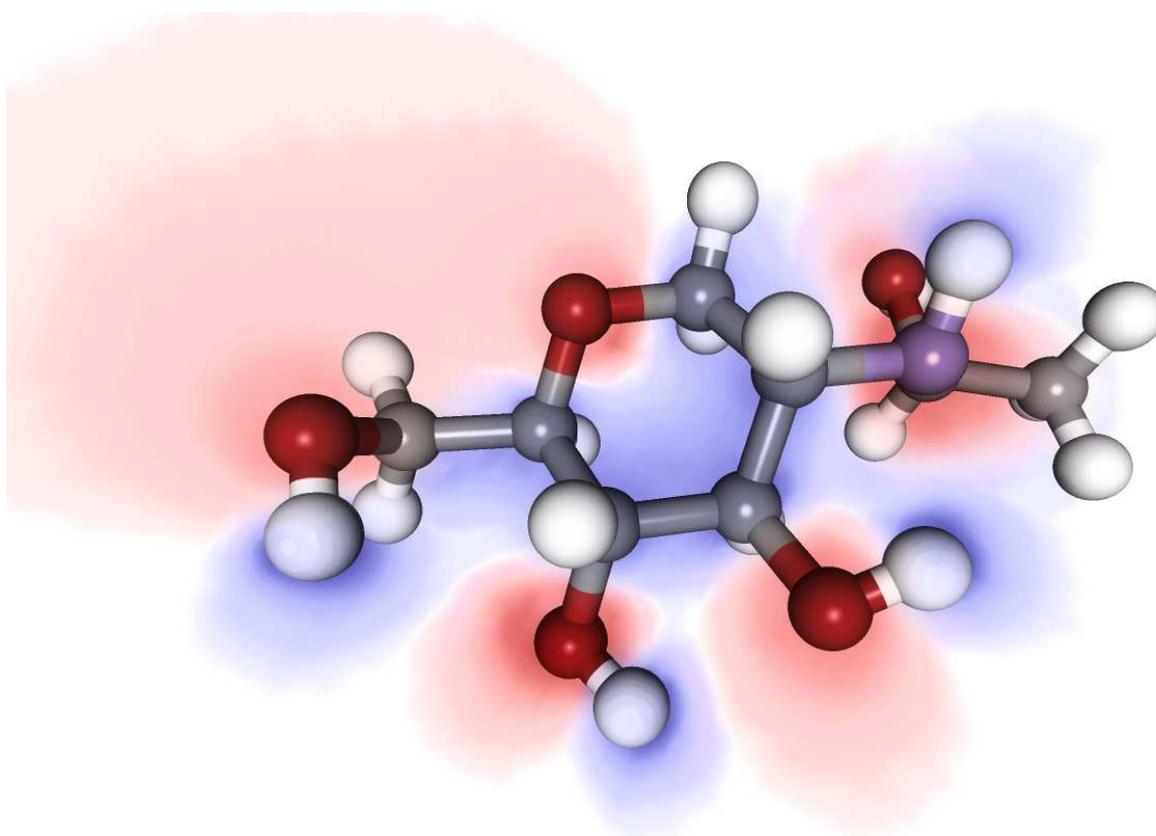


Figure 3.10: Example for BALLView's Volume Rendering. This image visualizes the electrostatic potential of N-acetyl-D-glucosamine. The potential was calculated with the integrated FDPB solver based on the charges from the MMFF94 force field.

Projection on a plane

To visualize the electrostatic potential at a given position, we added the possibility to create and freely place planes, which are colored using a potential grid. This can be useful for visualizing the potential in a binding pocket. The planes are colored using three-dimensional OpenGL textures, similar to the above described approach for volume rendering. With a provided Python script, it is also possible to create movies by stepwise

translation of a projection plane. Such a movie is available for download at the project's website [5].

Field Lines

In addition to the methods for visualizing electrostatic potentials discussed above, BAL-LView supports field lines for the direct visualization of the electrostatic field. The field

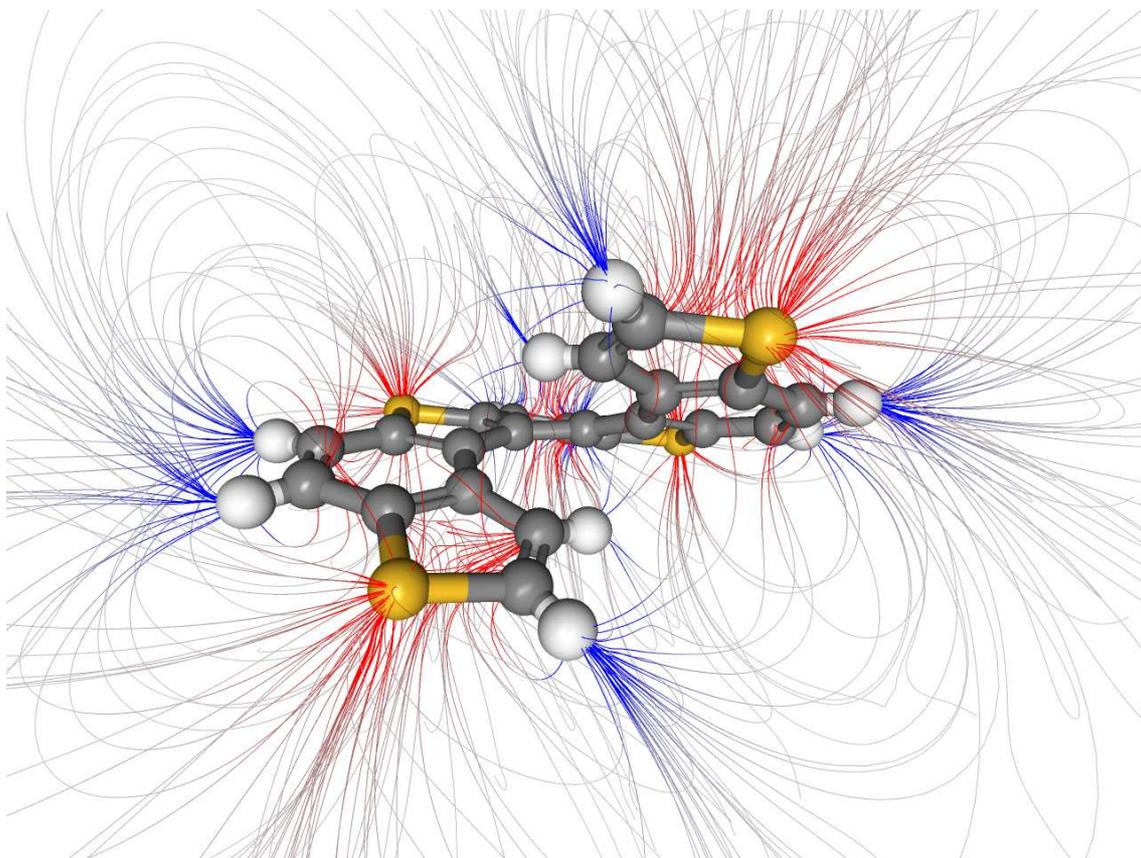


Figure 3.11: Example for the visualization of field lines. Their starting positions were calculated by using the approach of Stalling *et al.* [97]. To provide further information about the field, the field lines were colored by the potential's values.

lines or integral curves for a given vector field are those lines that are tangent to the field at any point in space. While the path of a field line is determined by the vector field, there are many different ways for controlling the distribution of field lines in three-dimensional space. Therefore, we implemented two distinct ways for calculating the starting positions of the individual field lines. The first approach generates a number of equidistant points for every individual atom. Each point becomes the start position for one field line.

Next, we implemented an approach that was described by Stalling *et al.* [97], in which the line placement is based on the electrostatic field strength. The data grid is divided into equally sized cubes and the probability for one field line to start in any given cube is

proportional to the strength of the field in that cube. The exact three-dimensional position in the cube is then randomly chosen. This approach provides a good overview of a field's strength, since it directly corresponds to the spatial field line density.

To obtain further information on the potential field's characteristics, BALLView supports the coloring of individual field lines with respect to the electrostatic potential (see Fig. 3.11). To achieve best results, users can freely choose the colors and interpolation values or use transparency.

3.2.4. OpenGL graphics

BALLView's integrated three-dimensional visualization is based on OpenGL [26], the current industry standard for platform-independent 3D graphics. In addition to its high performance, OpenGL has the advantage to be available on all major operating systems and for all graphics accelerator cards. As a result, BALLView is very portable. The next paragraphs will give an overview of the OpenGL engine's features.

Adaptable levels of detail

Graphics accelerator cards come in very different flavors and price classes. Therefore, a molecular viewer must be able to cope with several magnitudes of graphic performance. To solve this problem and to enable the visualization of really huge molecules, like entire ribosomal subunits, all models support four levels of detail with the exception of the *Line* model, where this is no issue. The lowest level shows only a very coarse approximation of the models, while the highest level provides very detailed representations, which are also suited for closeups. This enables users to create visualizations for molecules with thousands of atoms even on laptops with low-end graphics accelerator cards and software rendering. BALLView uses three methods for achieving the different levels of details.

For all OpenGL Utility Library (GLU) [26] based geometric shapes, like spheres or tubes, it is straightforward to scale the level of detail, since GLU can build these objects in the specified level of detail. For solvent-accessible or solvent-excluded surfaces, the level of detail is set via the parameter "degree of triangulation". The adaptation of the *Backbone* and *Cartoon* models is more complex, since these models do not consist of prebuilt geometric shapes. Here, the model processor is responsible for creating the desired level of detail by setting the interpolation degree for the splines and the number of points per circle/ellipse to appropriate values.

Drawing modes

BALLView features four different drawing modes to meet the needs for sophisticated visualizations. The **dot mode** renders all models as a set of simple dots and is thus the fastest mode. It was very helpful in the project's first years, since the graphics accelerator cards

were often too slow to render the other drawing modes. Today it is still part of BALLView, mainly for historical reasons, but for some users it may still be of use. By far more common is the **solid mode**, which renders all objects as opaque polygons and produces the most "realistic" graphics. Despite of its name, the solid drawing mode also supports the rendering of transparent models (see Fig. 3.13). The **wireframe mode** is often a good alternative for rendering objects transparent since it allows to visualize surfaces along with a model of the individual atoms (see Fig. 3.12). Furthermore, drawing wireframes is quite fast and thus allows for efficient visualizations on computers with limited graphics performance. We added a fourth drawing mode, the so called **toon mode** (see Fig. 3.14), which uses cel shading [8] for drawn-like results that are especially suited for schematic drawings. To achieve the desired effect, first, a black wireframe model with an increased line thickness is rendered. Second, the model is redrawn as solid polygons, but without lighting. Third, the lighting is simulated by using a 1D texture and by comparing all surface normals with the view vector. The toon mode is slower than the other modes, since all models have to be drawn twice and it does not support rendering through "Display Lists" (see Page 41). This results from the fact that the drawing depends on the actual viewing vector and thus can not be preprocessed. The idea for the toon mode stems from an on-line OpenGL cel-shading tutorial [27]. The toon rendering is the only drawing mode that is currently not supported through by the POV-Ray export, since it uses OpenGL specific features that are not available in the POV-Ray renderer.

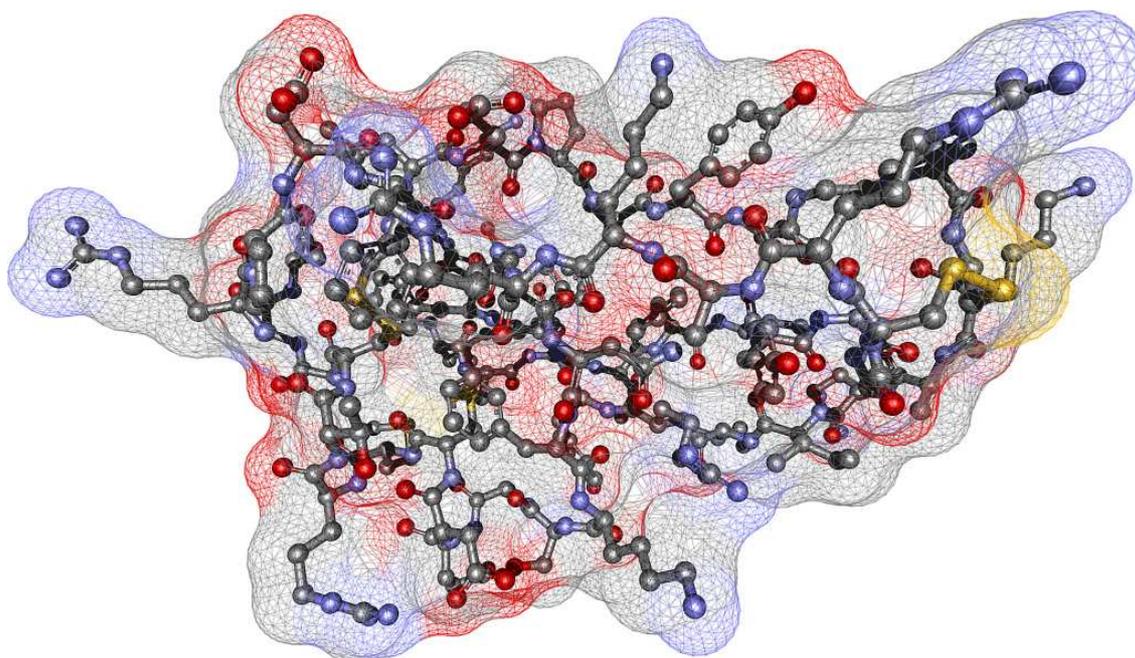


Figure 3.12: A solvent excluded surface as wireframe in combination with a solid *Ball-and-stick* model (PDB Id 2PTC).

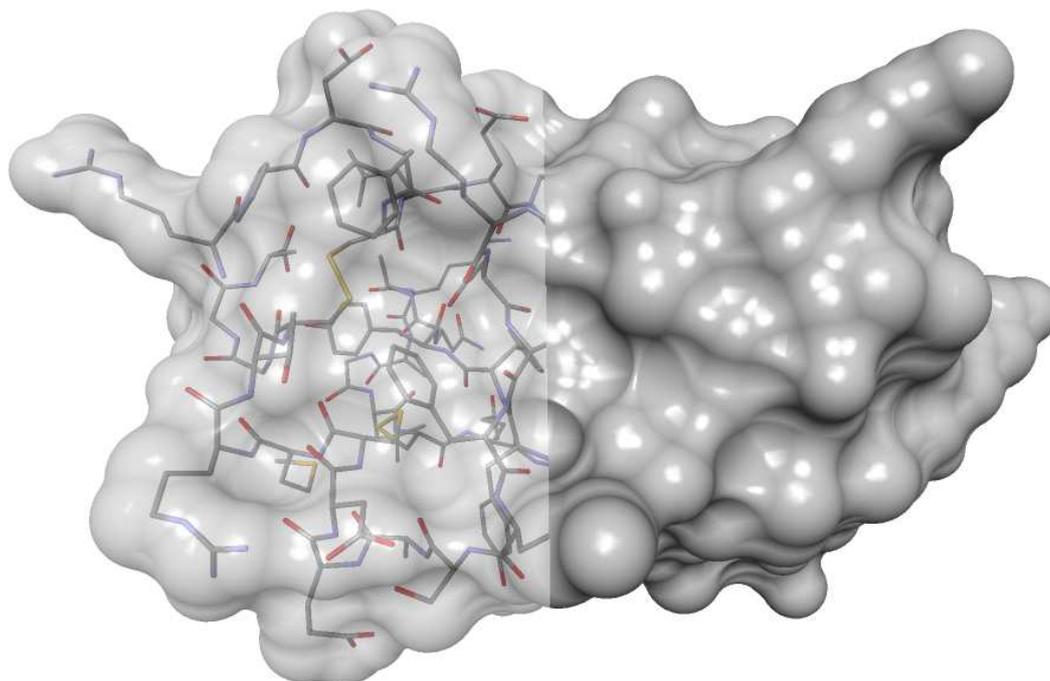


Figure 3.13: Comparison of solid and transparent surfaces. This image was created by using two clipping planes, one for each surface (PDB Id 2PTC).

User defined settings for the 3D graphics

We offer users a wide variety of means to adapt the 3D graphics to their liking, both to enable them to create sophisticated visualizations and to achieve the best looking results. Therefore, in addition to the possibility to modify the model properties like thickness and coloring, the rendering options are adaptable. This includes the material parameters that specify the behavior of the light reflections, depth cueing (i.e. fog), and the positions and intensities of the light sources. Thus, it is possible to highlight specific parts of a molecule or achieve better depth perception.

Clipping/Capping planes

The most interesting parts of a molecule are often hidden in a binding pocket. In such cases, it is difficult to find a perspective where no disturbing parts hide the point of interest. This problem can sometimes be solved by visualizing only a selected part of the molecule. Therefore, BALLView offers OpenGL clipping planes (see Fig. 3.15) which can cut individual representations at any given position in space. The positions can either be changed with the mouse or by specifying a point on the plane and a normal vector. In addition, the planes can be switched on/off with one mouse click. Another way of using

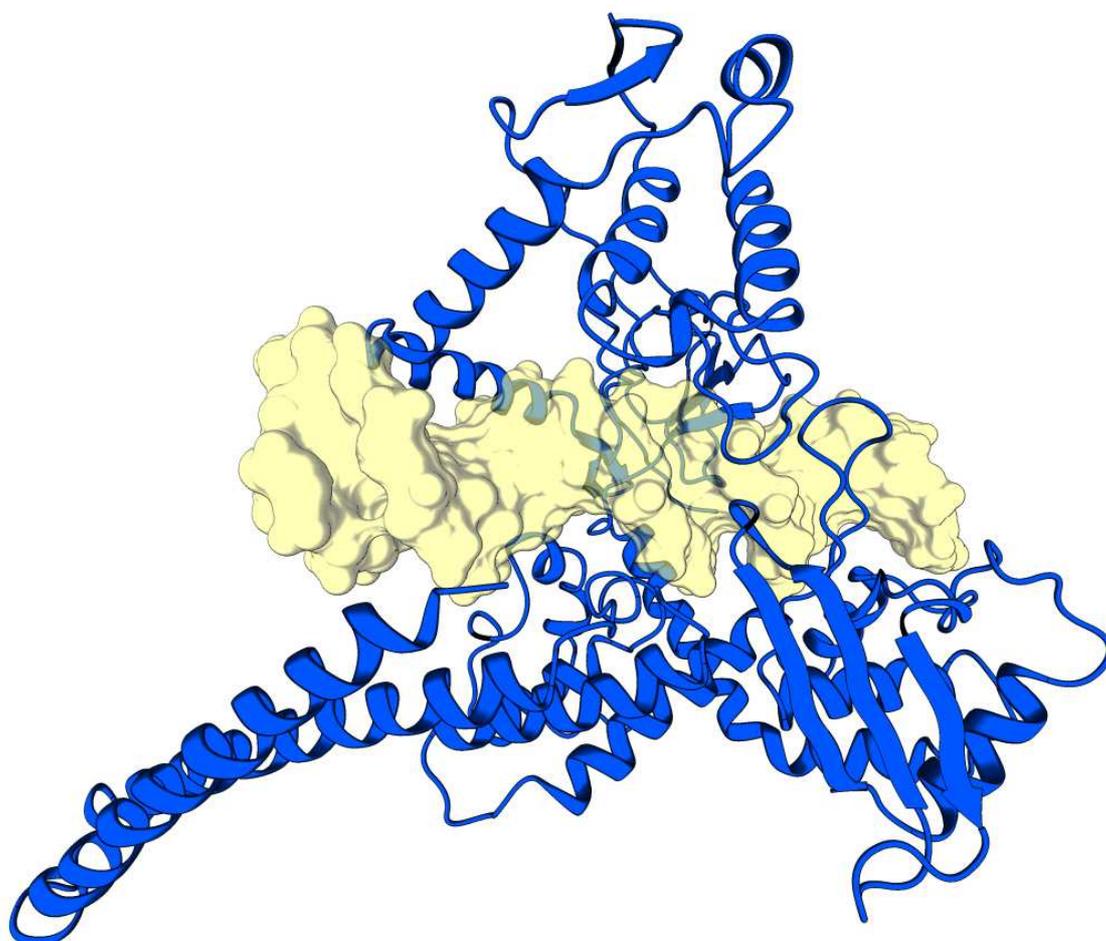


Figure 3.14: Toon shader model for a topoisomerase (PDB Id 1A36).

the clipping plane functionality are capping planes, which allow not only for slicing models, but also for enclosing them (see Fig. 3.16).

3D stereo visualization

In computer graphics three-dimensional objects are usually projected into a two-dimensional plane that represents the resulting images. Since this reduces the available information for the human eye, users often have problems with their spatial perception. This especially applies for the visualization of molecules, where the interesting parts of a structure are often hidden in a binding pocket. To circumvent this problem, molecular viewers often provide stereoscopic vision by rendering two half images with distinct view points and thus creating the illusion of depth. BALLView provides two different ways to achieve 3D stereo graphics with affordable hardware.

The first mode is called Side-by-Side projection. Here a left and right half image are shown, for instance on two projectors by using polarization filters. The disadvantage of

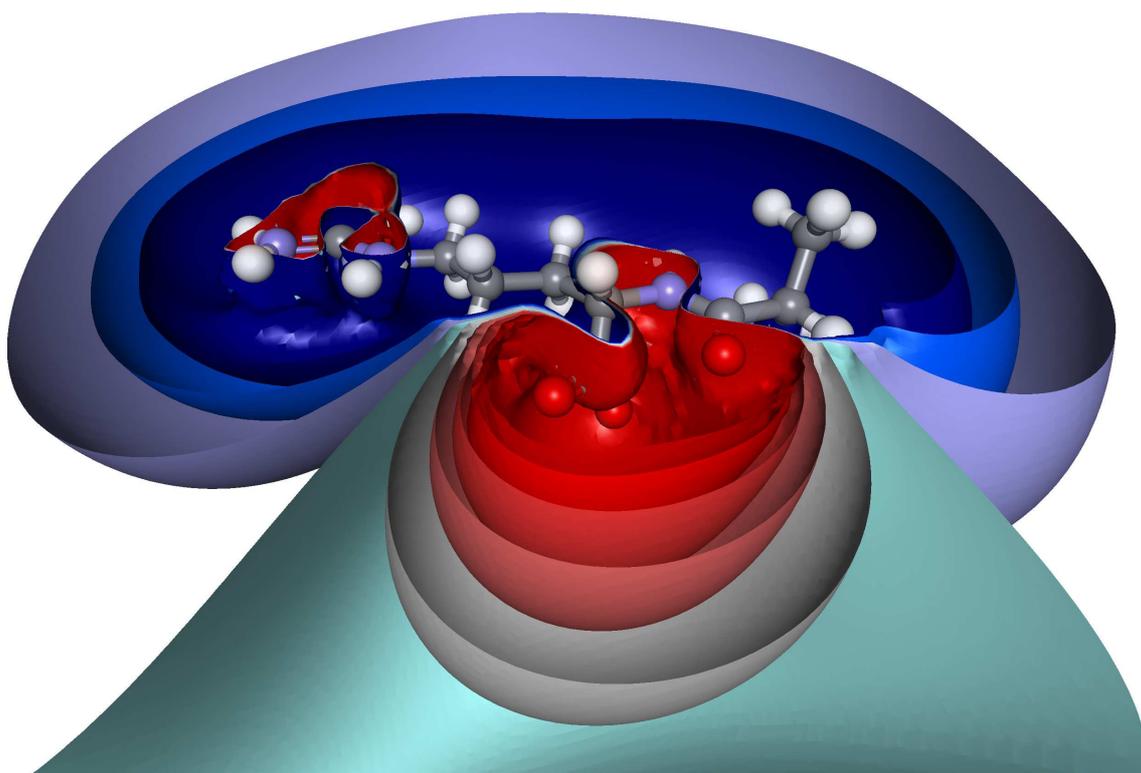


Figure 3.15: Example for the usage of clipping planes. Several isocontour surfaces were created for a dipeptide and sliced by one clipping plane to allow the view on the potential's gradient.

this method is that the horizontal resolution gets halved.

The second mode uses shutter glasses along with a CRT monitor, which shows the two half images in turn. To reach the desired effect, the half images have to be synchronized with the shutter glasses, which is often done via an infrared sensor. Unfortunately, on some platforms, this approach requires quite expensive graphics cards with the so called "Quad Buffer" feature. Furthermore, a CRT monitor with a high refresh rate is needed, because the rate for every half image is half the monitor's original frequency. If the refresh rate for a half image falls below 60 Hz, users may experience dizziness and headaches. We performed many stereoscopic presentations and achieved good results with both methods. For a pleasant experience, it is important that users can adapt the strength of the 3D effect, since otherwise the stereoscopic view may be too weak or lead to dizziness. The effect's strength is defined by the distance between the two distinct viewpoints (the so called eye-distance). BALLView offers a keyboard shortcut to modify the eye-distance such that the view can be adapted without leaving the stereoscopic view. Alternatively, the graphical user interface can be used. Since the eye distance gets stored when BALLView is closed, the setting is available for the next presentation.

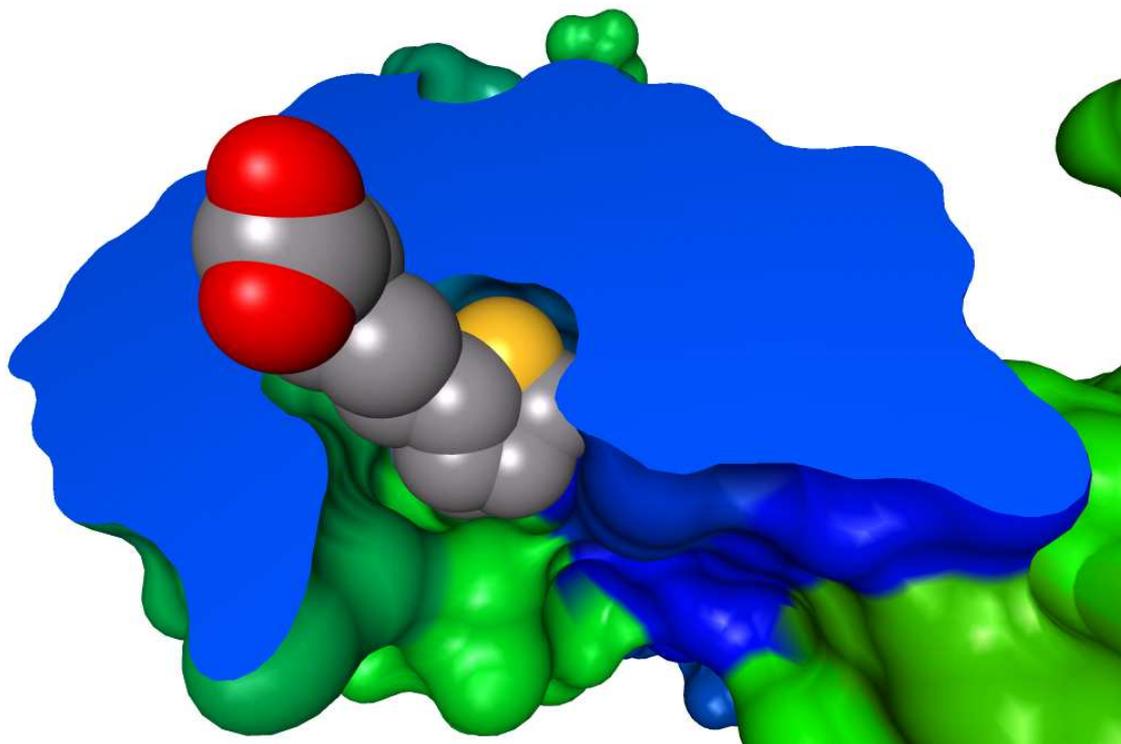


Figure 3.16: Example for the usage of capping planes. For a better view on the binding pocket, the receptor's surface has been clipped.

3.2.5. Creation of images and movies

Researchers often have to create images from molecular data for teaching, presentations, and publications. For some of these applications, users want to quickly produce images via snapshots, while for other purposes high quality images are required. To this end, we offer three distinct mechanisms to create images. The easiest and fastest way is to dump a screenshot in the "Portable Network Graphics" (PNG) format [31]. This file format provides lossless compression and is supported on virtually all operating systems. Unfortunately, screen dumps have some common drawbacks. They can only be used if no other window overlaps BALLView's 3D graphics window and the resolution of the resulting images is limited to the screen's physical resolution.

Offscreen rendering

To circumvent the above limitations, we implemented an additional way to create images that makes use of OpenGL's "Offscreen Rendering" feature (for a result see Fig. 3.6). Instead of producing a visible image, OpenGL can render into a hidden "Frame Buffer Object" (FBO), whose content can then be written to a PNG file. This approach supports resolutions of up to 4096 * 4096 pixels, which is sufficient for all common use-cases. If the full resolution of 16 megapixels is not needed, the image files can be downsampled

with any graphics software to achieve high quality anti-aliasing. The offscreen rendering has a further advantage. Since no visible window is needed to create an image, it can be used in non-GUI-applications, e.g. for the automated visualization of large image sets. Unfortunately, offscreen rendering is not supported by older graphic cards. For users with such a card, we provide an alternative procedure to obtain high quality images by exporting to the POVRay renderer (see below).

Export to the POVRay renderer

BALLView's internal OpenGL renderer is just one way of visualizing molecular structures. In a similar fashion, almost all models can be exported to POVRay (Persistence of Vision,

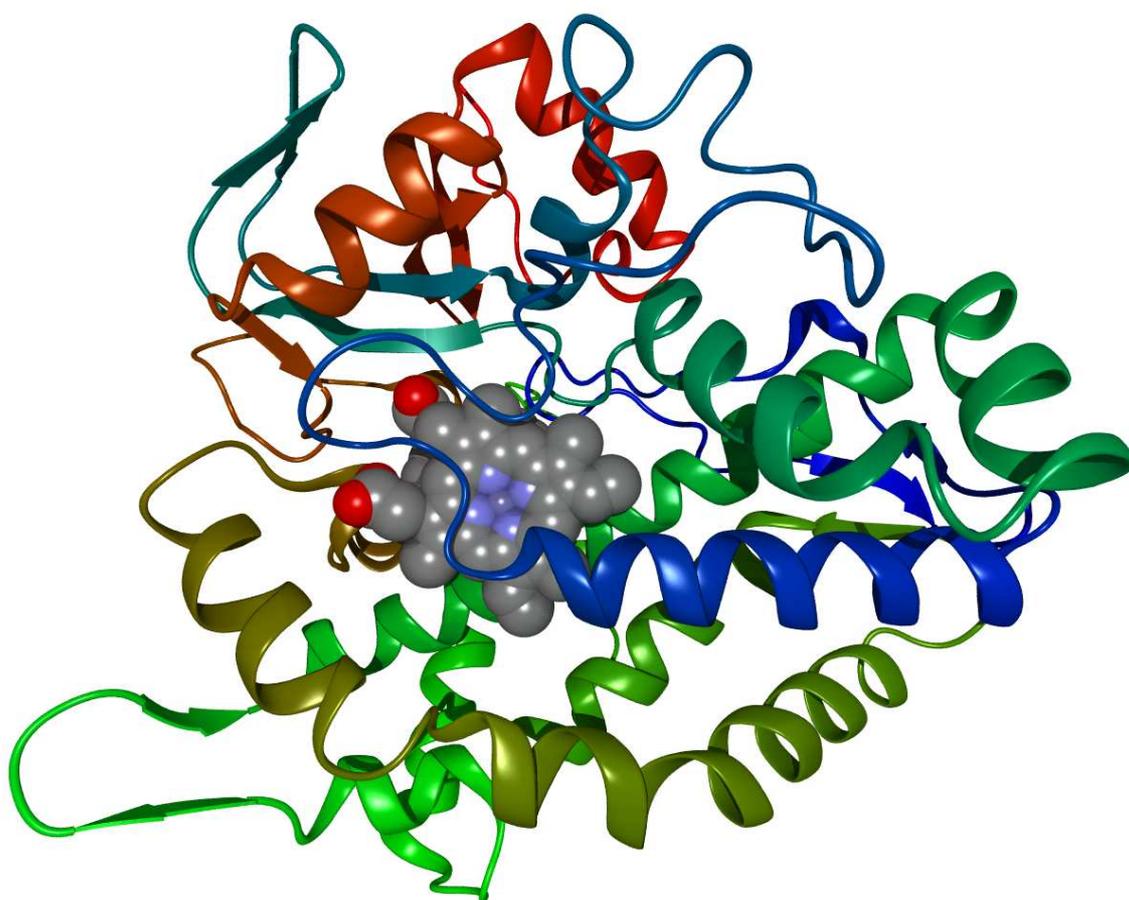


Figure 3.17: Example for BALLView's POVRay export. It shows a *Cartoon* model for the protein in combination with a *VDW* model for the ligand (PDB Id 1EGY).

a raytracer [32]) to produce publication-quality images with correct shadows and arbitrary resolutions (for an example see Fig. 3.17). This is done by translating all the different geometric objects into POVRay's notation and writing the resulting text stream into a file. Unfortunately, for complex scenes, these files can have a size of several hundreds megabytes. Therefore, it was important to optimize the `POVRenderer` class, such that it

uses time and space-efficient algorithms, while creating the POVRay files. The following techniques accelerate the rendering times and reduce the resulting file sizes up to a factor of ten compared to the naive implementation:

1. Indexing all occurring colors, instead of repeating this data.
2. Usage of limited precisions for the 3D positions data.
3. Usage of POVRay's Mesh2 objects.
4. Usage of POVRay macros.

POVRay macros provide not only the means to reduce the resulting file sizes, but also a convenient way for users to customize the resulting images to their liking. Thus, the header of our POVRay files allows for the adjustment of the transparency, lighting, and material parameters. While these settings can also be modified in the graphical user interface, it is sometimes faster to manually change the values in the file, since the export can be time intensive for complex scenes.

To ease the usage of the POVRay renderer, the first few lines of our POVRay files contain a command line with all required arguments:

```
povray +I1glu.pov +FN +O1glu.png +Q9 +W929 +H870 +A0.3
```

This line starts POVRay with reasonable values for the most important options, like the resulting image size, image format, quality settings, and antialiasing.

Unfortunately, some of the features offered in the OpenGL renderer are not available through the POVRay exporter. Cel-shading, capping planes, and volume rendering are currently not supported by POVRay and adding the corresponding functionality to POVRay is out of the scope of this work.

Creating movies

Images are clearly the most common and most important results of molecular viewers. But a single image does not suffice to visualize complex information like the course of a MD simulation. For such a task, movies and animation are often better suited. Therefore, BALLView, unlike other molecular viewers, offers a variety of ways to easily produce such movies and animations:

- through the POVRay renderer
- with the help of prebuilt Python scripts
- through a recording mode in the 3D view

- visualization of trajectories (see below)

All these methods have in common that they first store the resulting image files and then merge them with an external program in a second step. For this task, many free software tools exist, like mencoder [20] for Linux and Virtualdub [45] for Windows. The project's website contains several example movies that were created with the different approaches.

Visualizing trajectories

Trajectories computed by molecular dynamics simulations in BALLView or by external programs can be read and stored in DCD-format. By using a comfortable dialog (see Fig. 3.18), they can be visualized with any combination of models. Additionally, movies can be created from trajectories, either by using hard-copy images or the POV-Ray export. An example for such a visualization is available at the project's website [6].

We also implemented a new way for visualizing the mobility of the individual atoms in the course of an MD simulation. Here, a Python script iterates over all snapshots in a trajectory and builds a sphere at every atom position. This leads to images like Fig. 3.30.

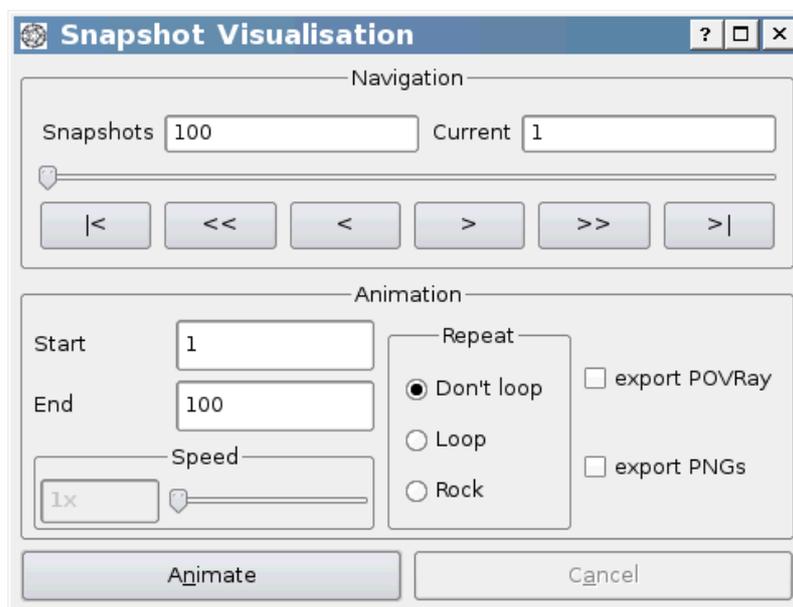


Figure 3.18: Trajectories can either be visualized snapshot by snapshot or as animation. The individual snapshots can be exported as PNG or POV-Ray files.

3.2.6. Comparison with related software

While a plethora of molecular viewers exists, only few are still under active development, and even fewer have a range of features that can compete with BALLView. Of these tools, the most advanced are: Chimera [10], PyMOL [57], and VMD [67]. Compared with these applications, BALLView has several advantages:

BALLView is much more user-friendly. This already starts with the graphical user interface framework: Since BALLView uses the Qt framework, which is one of the most advanced standard graphical user interface toolkits, users are accustomed to the look and feel of the GUI elements. This makes BALLView much more intuitive than the above tools that use proprietary or outdated GUI frameworks. Additional means like a sensible layout, context sensitive help, and user-defined hotkeys further improve the usability.

BALLView provides a higher adjustability than most comparable tools: A comfortable dialog allows to setup almost every feature to the user's liking. This includes the model and coloring options, the rendering's detail level as well as the design and layout of the graphical user interface: users can e.g. freely place all the main widgets and switch them on and off. Therefore, BALLView can be specifically adapted to the task at hand.

We designed BALLView such that users can easily obtain detailed information on the loaded structures. This includes hierarchical overviews of the compounds, their bond orders, charges, as well as atom names and types. These pieces of information are available both in custom-build dialogs and through their rendering in the 3D graphics widget.

BALLView provides state-of-the-art visualization features that are not supported by most molecular viewers. As an example, it provides an intuitive way of handling the combination of models and colorings: users can quickly create as many varying representations (see Page 29) as they like and customize them to their liking. Even more, since the different models and coloring methods can be freely combined, BALLView allows to visualize very complex relations very easily. BALLView, can thus e.g. visualize a molecule's electrostatic potential along with its structure.

3.3. Molecular modeling functionality

From the very beginning, BALLView was also designed as a powerful tool for molecular modeling that provides a common graphical interface to the BALL library [51, 76]. This combination of molecular visualization and modeling capabilities offers some advantages:

- Users can work more efficiently, since they no longer need to exchange data between their modeling application and their visualization tool via file operations. Since no data has to be exchanged between the different programs, many potential error causes like file format incompatibilities are avoided.
- BALLView's comfortable graphical user interface and its powerful visualization functionality can accelerate the setup of many molecular modeling tasks. As an example, users can quickly find and solve potential problems when working with erroneous molecular structures.
- The real time visualization for molecular modeling processes, like e.g. MD simulations or energy minimizations allows for monitoring such a calculation and for aborting it, e.g., if undesired results are produced.
- The combination of visualization and modeling features is ideal for teaching. As an illustration, students can experiment with the setup options of force fields, MD simulations, and minimizations and visualize the results in realtime. This may lead to a faster and deeper understanding of the underlying mechanisms.

3.3.1. Basic modeling features

File operations and database access

Importing and exporting data sets are prerequisites for virtually all computational tasks. Thus, BALLView contains native support for a wide range of file formats. It can read and write PDB, HIN, MOL, MOL2, and SD files.

In addition, the graphical user interface provides the means to download PDB files directly from the protein data bank [49] by using the PDB identifiers. Since many users have Internet connections that only allow access to HTTP and FTP through a proxy server, we added proxy support. In addition, it is possible to enter own URLs for downloading PDB files, either to use local database servers or any existing PDB mirror site. Furthermore, this feature will be useful if the PDB database should change the URLs of its files.

Working on subselections

When simulating a highly complex system such as large bio-molecules, it is often desirable to work on certain interesting subsets, like for example a special set of atoms in a molecular structure. BALLView offers three different ways to define such a subset. The

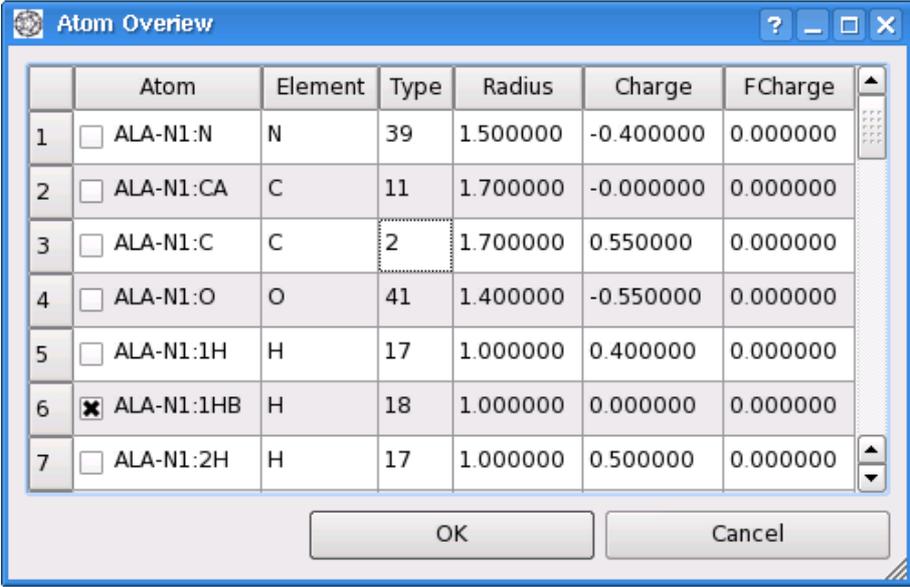
first possibility is to select atoms or molecules from the 3D graphics widget by clicking into the scene. The second possibility is provided by the structure widget (see Fig. 3.1, No. 4) which contains a hierarchical overview of the amino acid sequences and the corresponding atoms. It also allows for selecting certain objects like individual molecules, chains or residues. As a third possibility, advanced users can enter complex Boolean expressions into a special text field. As an example, the expression "element(H) AND residue(LYS)" selects all hydrogen atoms in lysine side chains. Since BALLView supports SMARTS expressions (see Page 80), users can even search for very complex molecular patterns, like all carbon atoms in carboxy groups.

Independent from its origin, a sub-selection can be used to copy, cut, paste, and delete the corresponding molecular entities. In addition, models can be created from subselections, e.g. to highlight a ligand.

Obtaining information on atoms, residues and chains

Another common task in molecular modeling is the identification of atoms based on their 3D representation, i.e. to identify the atoms at a given 3D position. In the "Picking" mode, BALLView enables the user to identify an atom with one mouse click. As a result, the atom is highlighted in the hierarchical list of loaded structures. Similarly, it is possible to mark atoms in the 3D graphics widget, by selecting them in the hierarchical list. Aside from the simple identification, BALLView provides many other means to obtain detailed information about a molecular entity:

- When a user double-clicks on an atom's representation in the 3D graphics, the atom's residue, name, type, and charge (formal or partial) are displayed.
- In addition, automatic and persistent labels are available with the name, charge, and type information of individual atoms (see Section 3.2.2).
- The distances, bond and torsion angles between atoms can easily be printed by selecting them.
- The number of atoms, bonds, and residues in a chain or molecule can be counted by using the corresponding context menu entry.
- Custom-built dialogs provide all available data for a given atom, bond or residue.
- Detailed information on all the atoms in a structure, including charge, formal charge, radius and atom type are available through a custom-build dialog (see Fig. 3.19). It allows to modify any of these values.



The screenshot shows a dialog box titled "Atom Overview" with a table of atom properties. The table has columns for Atom, Element, Type, Radius, Charge, and FCharge. Row 6 is selected, indicated by a checkmark in the first column.

	Atom	Element	Type	Radius	Charge	FCharge
1	<input type="checkbox"/> ALA-N1:N	N	39	1.500000	-0.400000	0.000000
2	<input type="checkbox"/> ALA-N1:CA	C	11	1.700000	-0.000000	0.000000
3	<input type="checkbox"/> ALA-N1:C	C	2	1.700000	0.550000	0.000000
4	<input type="checkbox"/> ALA-N1:O	O	41	1.400000	-0.550000	0.000000
5	<input type="checkbox"/> ALA-N1:1H	H	17	1.000000	0.400000	0.000000
6	<input checked="" type="checkbox"/> ALA-N1:1HB	H	18	1.000000	0.000000	0.000000
7	<input type="checkbox"/> ALA-N1:2H	H	17	1.000000	0.500000	0.000000

Buttons: OK, Cancel

Figure 3.19: A custom-built dialog provides access to all atoms and their properties. It allows for adjustments of the elements, types, charges (partial and formal) and radii.

Working with incomplete and erroneous molecular structures

A frequent problem while applying molecular mechanics methods are incomplete structures due to missing atoms in the x-ray structures. In particular, in low- to medium-resolution protein structures the hydrogens are missing. To solve these issues, BALL contains efficient heuristics for placing missing atoms which are available via the graphical user interface. This function even allows for adding missing side chains.

In addition, BALLView can search for common structural problems, e.g. overlapping atoms, "strange" charges (i.e. charges that are uncommon for a given element and connectivity), or extreme bond lengths. The atoms in question are highlighted to ease their inspection and manipulation. Users can then move or delete these substructures, or manually assign missing atom types and charges, if necessary.

Assignment of secondary structures

The knowledge about a protein's secondary structure can provide deep insights into its organization, similarity to other structures, and functions. Although most entries in the protein database already contain a secondary structure assignment, many of these assignments are simply wrong. Therefore, we implemented a variant of the DSSP algorithm [70], which is one of the most renowned algorithm for assigning secondary structures. Based on hydrogen bond patterns, that are identified in a first step, it generates an automatic assignment of a proteins secondary structures.

3.3.2. Molecular Mechanics

Molecular modeling mainly relies on two different techniques for describing the energies and forces on the molecular scale: quantum mechanics and molecular mechanics. Quantum mechanics uses wave functions to fully describe the atomic particles in a molecular system. It has the advantage to yield very accurate results without the need for any prior knowledge on the system's behavior (ab initio calculations). Unfortunately, the number of required operations grows with at least $O(n^4)$, where n is the number of symmetry-independent basis orbitals [77]. Therefore, quantum mechanics approaches are less suited for computing the behavior of large molecules with hundreds or more atoms. To circumvent this limitation, molecular mechanics methods have been developed. They can cope with molecular systems consisting of thousands of atoms, because here the computation time grows with at most $O(n^2)$, where n is the number of atoms. To achieve this reduction in complexity several assumptions are made:

1. Each atom is simulated as a single particle such that relatively simple physical models can be applied. These models can be thought of as mutually independent springs that restore "natural" bond lengths, angles, etc. Unfortunately, this approach leads to less accurate results compared with quantum mechanics methods since explicit interactions between individual electrons are neglected.
2. To describe the interactions between individual atoms, often equilibrium values are used, e.g., for bond lengths, bond angles or inter-nucleic distances between non-bonded atoms. These parameters sometimes stem from ab initio quantum mechanics calculations, but are mostly obtained through fitting experimental data. Therefore, to calculate the behavior of one type of molecules, previous experimental values for similar structures are required.
3. Since atoms of the same element can show different chemical behavior, molecular mechanics differentiates between individual types of atoms. For this purpose, often the hybridization of the atoms is used which leads, for example to sp , sp^2 , or sp^3 carbon atom types. The atom types define which values are used for the parameters in the individual components, e.g., for equilibrium bond lengths.
4. Molecular mechanics calculates a system's potential energy as a sum of energy terms that are presumably independent from each other, e.g.:

$$E = \underbrace{E_{bond} + E_{angle} + E_{dihedral}}_{E_{covalent}} + \underbrace{E_{electrostatic} + E_{vdW}}_{E_{non-bonded}}$$

For instance, E_{bond} is the sum of all bond stretching energies. Here, the energy of one bond between the atoms i and j is calculated through the following equation,

based on the difference Δr_{ij} between their current inter-nucleic distance and the equilibrium distance (for a more detailed description of this equation see Page 88):

$$EB_{ij} = 143.9325 \frac{kb_{ij}}{2} \Delta r_{ij}^2 \left(1 + cs \Delta r_{ij} + \frac{7}{12} cs^2 \Delta r_{ij}^2 \right)$$

One such equation and the corresponding parameters is called a "force field component" while an entire set of components is a "force field". Force field components can be divided into the bonded interactions that are applied on atoms connected through covalent bonds and the so called "non-bonded" interactions.

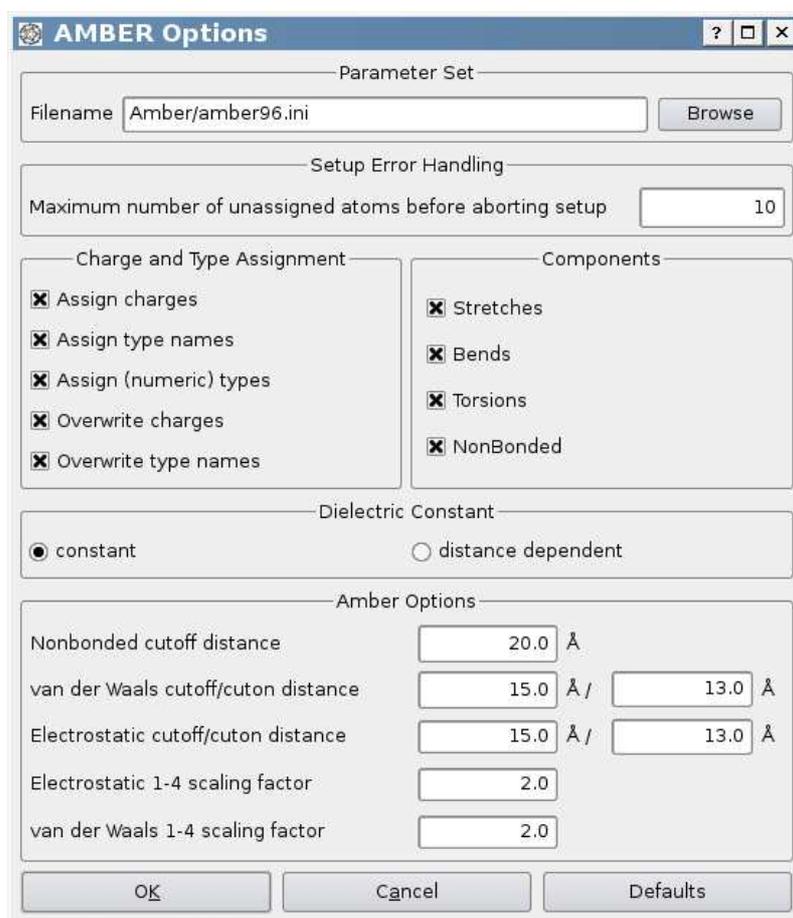


Figure 3.20: Configuration dialog for the AMBER force field [55]. Like all other force fields in BALLView, it can be easily configured, including enabling/disabling the individual components of the force field as well as their cutoff values.

BALLView provides a common graphical interface to the wide range of molecular mechanics capabilities implemented in the BALL library [51, 76]. Users can choose between the AMBER [55], CHARMM [52], and MMFF94 [62] force fields and use them for a wide variety applications like the calculation of single point energies, molecular dynamics simulations, or the relaxation of strained conformations through energy minimizations. In

addition, the force fields provide routines for the assignment of formal and partial charges that are suitable as basis for electrostatic potential calculations or visualization tasks. The graphical user interface provides many different ways to ease these tasks. For instance, the force fields can be easily be configured to user-friendly dialogs (see Fig. 3.20). This also includes the enabling/disabling of individual force field components, like electrostatics or the van-der-Waals component. The configuration of the force fields is automatically stored, such that it will be identical the next time BALLView is started.

When a force field is applied to a molecular system, any problems with the parametrization are logged to allow their inspection. Furthermore, if a user defined threshold of unassigned atoms is exceeded, the calculation is aborted and the problematic atoms are highlighted to allow for manual assignment of the atom types.

Since we implemented the molecular dynamics simulations and energy minimizations as separate threads (see Section 2.6.3), a user can observe their realtime visualization, stop the calculations at any time and continue them later on.

The MMFF94 force field

At the beginning of this work, the BALL library contained the implementations of two force fields: AMBER [55] and CHARMM [52]. While these force fields are very powerful, they have one common drawback. In their original form, both are parametrized for amino acids and nucleotides but they lack suitable data for small compounds. Therefore, we wanted to add a further force field with a parametrization for these kinds of molecules. This force field could not only be used for Protein-Ligand Docking but also for minimizations of structures that contain small ligands. In addition, it would be especially useful for BALLView's molecular editing functionality (see Section 3.3.3). We analyzed the specifications of the existing force fields and found the MM3 [48], CFF [87], GAFF [58], and MMFF94 [62, 63, 64, 65, 66] force fields the most promising ones. Out of these candidates, we chose the Merck Molecular Force Field (MMFF94) since it is well documented and has a wide parametrization. The following pages will describe our implementation of the MMFF94 force field.

Atom type assignment

As mentioned above, the MMFF94 force field can be applied to a wide variety of compounds. To this end, MMFF94 requires a complex atom typing mechanism that differs from the type assignment in our implementations of AMBER and CHARMM. In these two force fields, the atom types were simply assigned based on the atom names. In contrast, the MMFF94 atom types are based on molecular patterns describing the chemical neighborhood of the considered atoms, where e.g. one atom type corresponds to the carbon

atoms in carboxy groups. The type assignment procedure can be divided into four distinct phases:

1. Assignment of basic types

The atom typing process starts with the assignment of the atom types for all heavy atoms. MMFF94 differentiates between roughly 100 different atom types and each type can require several rules, resulting in more than 150 assignment rules. The atom types are represented by numerical values (1-99) and textual symbols (e.g. "C = O").

2. Assignment of aromatic types

Based on the previously assigned general atom types, new types are assigned to the atoms in aromatic rings. The main criteria for this process is the number of bonds between an atom and its aromatic ring's hetero atom.

3. Assignment of hydrogen atoms types

Hydrogen atoms are assigned last in correspondence to their partner atoms. Every hetero atom that can share a bond with a hydrogen atom thus also defines an atom type for the connected hydrogens.

The SMARTS matcher

Since the Merck force field requires more than 150 assignment rules, implementing this assignment procedure in C++ would have been quite complex and time consuming. Therefore, we decided to base the basic atom type assignment on a SMARTS [41] matcher which was implemented by Andreas Bertsch. SMARTS is a language for describing molecular patterns similar to SMILES [104] (Simplified Molecular Input Line Entry System) which can be used for searching patterns in compound databases. SMARTS provides symbols for describing atomic properties (like the atomic symbol and charge, see Table 3.2) along with bond properties (e.g. bond order). Both, atomic symbols and bond properties, can be combined with Boolean operators to describe complex substructures. For example, phenol has the SMARTS expression "[OH]c1ccccc1", where the "[OH]" prefix defines a hydroxy group that is bound to one of six aromatic carbon atoms that form an aromatic ring. The usage of SMARTS for realizing the rule set had several advantages compared to a C++ implementation. The development process was much faster since the rule set could be modified without the need to recompile and link the software library. In addition, users can customize or extend the rule sets for new atom types, since we developed an easily modifiable file format for the assignment rules. It stores one assignment rule per line, starting with the type's element, next the textual type, the numerical type, and the SMARTS expression:

Symbol	Atomic properties
*	any atom
a	aromatic
A	aliphatic
D< n >	< n > explicit connections
H< n >	< n > attached hydrogens
h< n >	< n > implicit hydrogens
R< n >	in < n > SSSR rings
r< n >	in smallest SSSR ring of size < n >
v< n >	total bond order < n >
X< n >	< n > total connections
x< n >	< n > total ring connections
-< n >	-< n > formal charge
+< n >	+< n > formal charge
#n	atomic number < n >

Symbol	Bond properties
-	single bond (aliphatic)
=	double bond
#	triple bond
:	aromatic bond
~	any bond (wildcard)
@	any ring bond

Symbol	Meaning
!e1	not e1
e1&e2	a1 and e2 (high precedence)
e1,e2	e1 or e2
e1;e2	a1 and e2 (low precedence)

Table 3.2: Overview on the SMARTS syntax. It provides symbols for atom and bond properties as well as Boolean operators.

*ELE	SYMBOL	TYPE	RULE
C	CR	1	[#6X4]
C	CSP2	2	[#6X3]
C	C=C	2	[\$([#6]=[#6])]
C	C=O	3	[\$([#6]=O)]

Unfortunately, the SMARTS matching process is computationally expensive for expressions like

[\$([#7](-!#7)=[#6X3]-[#7X3][!#7])]

which is only a part of the expression for a type of nitrogen atoms that appears in some resonance structures. Here, a pattern search has to be performed over a depth of four

atoms, to ensure that the connectivity of all atoms matches the sought pattern: a nitrogen that is not bound to an other nitrogen but that shares a double bond with a carbon atom (that has three connections) which is again bound to an other nitrogen atom (with three connections). This nitrogen must not be bound to an other nitrogen. In a naive implementation, all 150 rules would have to be successively applied to the entire atom set, resulting in long runtimes. Therefore, we had to tune the assignment process. To this end, the implementation sorts the rules into groups for the individual elements such that a rule only has to be applied for the atoms with the corresponding element (line 8). Here, the SMARTS matcher returns the atoms that match a given rule. These atoms are then assigned with the corresponding type for the current rule (line 10). To further accelerate the matching code, it was modified such that the more specific atom types are assigned first. Thus, successfully assigned atoms can be removed from later assignment trials (line 11).

```
1 | clearPriorAssignment()
2 | SMARTSMatcher smarts
3 | all_atoms = collectAllAtoms()
4 | for rule_group in all_rules:
5 |     element = rule_group.element
6 |     atoms_to_assign = collectAtoms(all_atoms, element)
7 |     for rule in rule_groups:
8 |         matched_atoms = smarts.match(atoms_to_assign, rule.expression)
9 |         for atom in matched_atoms:
10 |             atom.setType(rule.type)
11 |             atoms_to_assign.erase(atom)
```

For a further acceleration, the SMARTS expressions were carefully tuned to deliver optimal performance. Now, the more stringent and faster testable constraints (like the number of valence electrons) are checked first, before e.g. comparing the connectivity of the neighboring atoms. By using the above techniques, we managed a significant acceleration of the atom typing process. As an example, the atoms in the MMFF94 validation suite [22] are now typed more than ten times faster, compared to our earlier implementation.

Since for many self drawn structures no explicit charge information would be available, we decided to base these assignment rules as far as possible on the bond connectivity instead of the formal charge information. The only cases where the charge information is still needed are monoatomic ions as Fe^{2+} or Fe^{3+} , since they have no covalent bonds.

Since the described approach showed to be very effective, we implemented the SMARTS atom typer as a generic class that can easily be adapted for future assignment tasks, for instance in the implementation of additional force fields.

Verification of the type assignment rules

Unfortunately, the description of the individual atom types in the original paper is in parts very vague. As a result, for many types different alternative expressions were thinkable that could match the short textual descriptions. Therefore, a lot of testing and adaption was needed to ensure the consistency with the original implementation for all assignment rules. The testing was performed against the MMFF94 validation suite [22], which consists of roughly 17,000 atoms in more than 750 different molecules. The final SMARTS rule set now matches all these atoms to the correct MMFF94 atom types. Since the validation suite does not contain proteins or the full set of amino acids, we had to perform additional tests for these kind of structures. Therefore, we created peptides containing the 20 common amino acids and compared the resulting atom types with the assignment of the MMFF94 implementation in the CHARMM program.

Kekulization

Organic compounds often consist of aromatic groups, meaning that some of the participating electrons are delocalized and shared between the individual atoms. The bonds in such groups can either be annotated as conjugated single/double bonds or simply as "aromatic". While many molecular file formats may contain the latter kind of bond order assignment, the MMFF94 force field unfortunately does not support it. Thus, if one wants to use such molecules with the Merck force field, the annotation of these bonds has to be transformed into conjugated single/double bonds, a process often referred to as "Kekulization" after the pioneer work of Kekulé [72] (see Fig. 3.21).

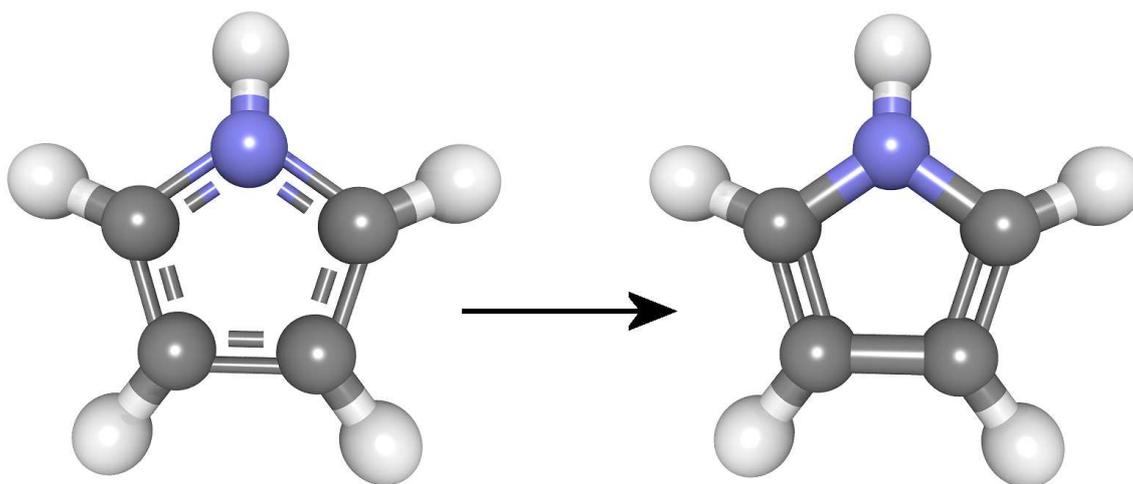


Figure 3.21: Kekulization of Pyrrole. The aromatic bond assignment on the left is transformed into conjugated single and double bonds on the right. Pyrrole is one of the smallest aromatic compounds and its Kekulization is therefore almost trivial. Fig. 3.22 shows the Kekulization of a more complex structure.

Therefore, we extended BALL with a Kekulizer implementation. This also became very useful, when we developed the molecular editing functionality (see Page 90), since it allows a user to sketch aromatic rings without explicit bond order assignment. While the Kekulization process is trivial for all atoms that are not part of an aromatic ring system, the situation drastically changes for polycyclic aromatic compounds. Here, no simple rules can be applied for the assignment. Unlike other Kekulizer implementations, for instance in the JOELIB [18] software, our version should not require explicit information on the formal charges of the participating atoms and thus accelerate the sketching of molecules (see Section 3.3.3). Furthermore, if for a given structure different assignments are thinkable, the best solution shall be found. Best here means (in the order of importance):

1. the resulting charges match any previously set formal charges
2. the least amount of newly introduced formal charges
3. the electronegativities of the individual atoms are taken into account
4. for assignments with several charged atoms, the optimal charge distribution shall be found (equally charged atoms shall be as far apart as possible, oppositely charged atoms as near as possible).

The first three points are realized in a simple function that calculates a penalty for an atom's current bond assignment. The penalty is based on the atom's element and the resulting formal charge and accounts for chemical preferences:

```
calculatePenaltyForCharge (atom):  
    if use_formal_charges and atom.formal_charge != atom.charge  
        return 100  
    if atom is positively charged nitrogen return 10  
    if atom is negatively charged nitrogen return 11  
    if atom is negatively charged carbon return 25  
    if atom is positively charged carbon return 26
```

While this heuristic is rather simple, it provides accurate results (see Page 86). If nevertheless the need should arrive for a more sophisticated approach, the Kekulizer class can easily be modified to exchange the scoring function.

The core of the Kekulizer is designed as a branch-and-bound algorithm, which recursively iterates over all atoms in an aromatic system. First, some initializations are performed where the lowest found penalty for any found solution is set to the highest possible value (line 1) and the penalty for the current assignment is set to zero. Next, the assignment is started for the first atom in the current aromatic system (line 3). The method `fixAromaticSystem` has only one parameter which is the number of the atom that is to be processed.

Kekulizer algorithm

```
1 lowest_penalty = INT_MAX
2 current_penalty = 0
3 fixAromaticSystem(0) // 0 = first atom in current aromatic system
4 if lowest_penalty < INT_MAX
5     applySolution(calculateBestDistributedSolution())
6
7 fixAromaticSystem(atomno):
8     if current_penalty > lowest_penalty return
9
10    current_atom = getAtom(atomno)
11    if no more atoms
12        if current_penalty < lowest_penalty
13            clearSolutions()
14            lowest_penalty = current_penalty
15            storeSolution()
16        return
17
18    if not further bonds are to be assigned for current_atom
19        X = calculatePenaltyForCharge(current_atom)
20        current_penalty += X
21        fixAromaticSystem(atomno + 1)
22        current_penalty -= X
23        return
24
25    if current_atom is uncharged fixAromaticSystem(atomno + 1)
26
27    if further double bonds can be assigned to current_atom
28        if not atom is uncharged with one double bond
29            Y = calculatePenaltyForCharge(current_atom)
30
31        for all aromatic bonds of current_atom
32            if partner_atom allows further double bond
33                Z = calculatePenaltyForCharge(partner_atom)
34                assignDoubleBond(current_atom, partner_atom)
35                current_penalty += Y + Z
36                fixAromaticSystem(atomno + 1)
37                eraseDoubleBond(current_atom, partner_atom)
38                current_penalty -= Y + Z
39
40    if not current_atom has minimum numbers of double bonds
41        A = calculatePenaltyForCharge(current_atom)
42        current_penalty += A
43        fixAromaticSystem(atomno + 1)
44        current_penalty -= A
45    return
```

In the line 8, the algorithm tests if the penalties for the currently assigned bonds and charges is higher than the penalty for a previously found solution. If this is the case, the algorithm steps back one atom and tries an other assignment. Otherwise the algorithm tests if there are still atoms to be processed (line 11). If there are no remaining atoms, the algorithm has found a solution that is at least as good as any previous solution. If the newly found solution is better, the earlier found solutions are cleared (line 13). In any case, the solution and its penalty are stored (line 14-15) for later access. If the algorithm reaches line 17, there are still atoms to be processed. The algorithm then tests if no more double bonds can be assigned for this atom. If this is the case and thus a further charge must be assigned to the aromatic system, the corresponding penalty is calculated (line 19). Anyhow, the assignment process is continued for the next atom in line 21.

Next, a solution is sought where the current atom is uncharged (line 25). This can lead to faster results for aromatic systems where no charges must be assigned. In the lines 31-38, the algorithm iterates over all aromatic bonds of the current atom and tries a double bond assignment for one bond after the other. Since these newly assigned double bonds are only created to atoms with a higher atomno, the previously calculated penalties still apply. As a last test, the lines 40-45 try an assignment where the current atom does not obtain a double bond at all.

The described process in `fixAromaticSystem` thus first searches for an assignment without introducing any additional charges and then stepwise adds more and more charges until a solution is found. Further solutions are sought until it becomes obvious that no better assignment can be found. At this point, the algorithm stops and processes the found solutions. If a solution was found where no additional charge were assigned, it is applied. Otherwise, the solution with the best distributed charges (see point 4 above) is assigned (line 5). The described approach theoretical has an exponentially growing runtime with the numbers of atoms in a aromatic system. But for the numerous test cases that we have computed so far (see below), the branch-and-bound algorithm quickly identifies new solutions and thus rapidly reduces the search space. Thus, runtimes in the region of milliseconds are achieved, even for complex systems.

To test the Kekulizer implementation, we mainly relied on the MMFF94 validation suite [22] since it contains a wide variety of aromatic compounds. To perform the testing, we implemented the following procedure:

1. Aromatic bond types are assigned to all bonds in aromatic ring systems.
2. To achieve a more stringent test, we clear any formal charge assignment.
3. Next, our Kekulizer is applied.
4. The newly assigned bond orders and formal charges are compared to the original values in the validation suite.

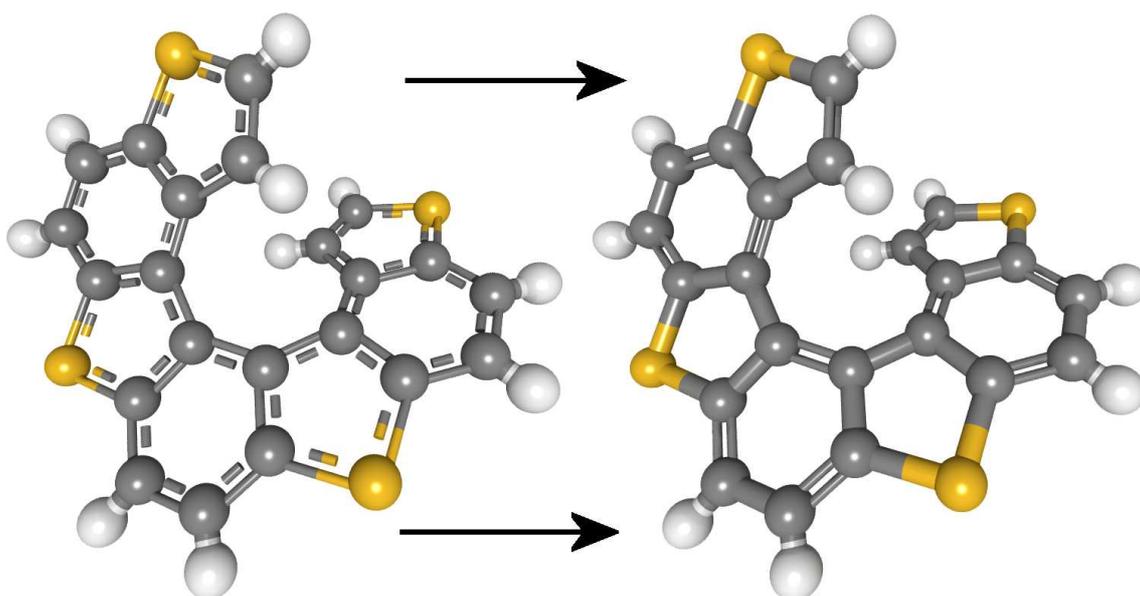


Figure 3.22: The process of Kekulization for a complex heteroaromatic polycyclic compound. Since the order of any ring bond depends on the order of any other ring bond, Kekulizing such a molecule is quite sophisticated.

Our results showed to be identical to the initial values, except for two molecules (Ids GEY-WOW and FITTIL), where our Kekulizer produced another, but still correct assignment. These two compounds are charged in a way that the formal charge information is needed to achieve the original assignment. If our Kekulizer is allowed to use these charges, it computes the original values. As a result, we were able to show that our Kekulizer implementation exhibits a reasonable assignment.

To ensure that our Kekulizer can also cope with very complex and uncommon structures, we performed additional tests. We searched in compound databases for different kinds of heteroaromatic polycyclic structures (see e.g. Fig. 3.22) that were not included in the MMFF94 validation suite. Our Kekulizer also correctly assigned these molecules.

Components

The Merck force field consists of seven different components, which contribute to the energy expression:

$$E_{MMFF} = \sum EB_{ij} + \sum EA_{ijk} + \sum EBA_{ijk} + \sum EOO_{P_{ijk,l}} + \sum ET_{ijkl} + \sum E_{vdW}_{ij} + \sum EQ_{ij}$$

The subscripts i, j, k, l correspond to the individual atoms. The components will be described on the following pages. Here, large letters I, J, K, L will mark constants which only depend on the atom types rather than on the atom positions.

The **bond stretching** component employs a quartic function that calculates a potential

energy based on the difference Δr_{ij} between the current bond length and the reference bond length of atoms i and j . kb_{IJ} is a force constant in millidyne/angstrom (md/Å) that is specific for the atom types I and J . $cs = -2 \text{ \AA}^{-1}$ is the "cubic-stretch" constant.

$$EB_{ij} = 143.9325 \frac{kb_{IJ}}{2} \Delta r_{ij}^2 \left(1 + cs \Delta r_{ij} + \frac{7}{12} cs^2 \Delta r_{ij}^2 \right) \quad (3.1)$$

For the **angle bending**, MMFF94 uses a cubic expansion to calculate the resulting energies with respect to the deviation ϑ_{ijk} (in degree) of the optimal angle between the atoms i , j , and k . $cb = -0.007 \text{ deg}^{-1}$ is the "cubic-bend" constant.

$$EA_{ijk} = 0.043844 \frac{ka_{IJK}}{2} \Delta \vartheta_{ijk}^2 (1 + cb \Delta \vartheta_{ijk}) \quad (3.2)$$

The **stretch-bend** component applies the following equation, where kba_{IJK} and kba_{KJI} are force constants that couple the $i - j$ and $k - j$ stretches to the $i - j - k$ bend.

$$EBA_{ijk} = 2.51210(kba_{IJK} \Delta r_{ij} + kba_{KJI} \Delta r_{kj}) \Delta \vartheta_{ijk} \quad (3.3)$$

For trigonal centers, the **out-of-plane** component calculates the potential through the following equation. Here, $\chi_{ijk;l}$ is the Wilson angle between the bond $j - l$ and the plane $i - j - k$. The three angles that arise at a given central atom j are all assigned the same $koop_{IJK;L}$ force constant (md Å/rad²).

$$EOOP_{ijk;l} = 0.043844 \frac{koop_{IJK;L}}{2} \chi_{ijk;l}^2 \quad (3.4)$$

The **torsion** component calculates the potential for the atoms i, j, k and l , where $i - j$, $j - k$, and $k - l$ are bonded pairs and ϕ is the $i - j - k - l$ torsion angle.

$$ET_{ijkl} = 0.5(V_1(1 + \cos(\phi)) + V_2(1 - \cos(2\phi)) + V_3(1 + \cos(3\phi))) \quad (3.5)$$

While the above components all arise between bonded atoms, the **van-der-Waals** component only applies to atoms, that are separated by at least three bonds. ϵ_{IJ} is the well depth (calculated based on the atomic polarizability), R_{ij} the current distance, and R_{IJ}^* the optimal distance between the two atoms.

$$E_{vdW_{ij}} = \epsilon_{IJ} \left(\frac{1.07 R_{IJ}^*}{R_{ij} + 0.07 R_{IJ}^*} \right)^7 \left(\frac{1.12 R_{IJ}^{*7}}{R_{ij}^7 + 0.12 R_{IJ}^{*7}} - 2 \right) \quad (3.6)$$

The **electrostatics** component uses a buffered coulombic form, where q_i and q_j are the partial charges, R_{ij} is the distance of the two atom nuclei, $\omega = 0.05 \text{ \AA}$ is the "electrostatic buffering" constant, and D is the "dielectric constant" (default = 1). To achieve a distance dependent dielectric constant, n can be set to 2, but the default is 1. In contrast to the VDW interactions, 1,4-electrostatic interactions are scaled by a factor of 0.75.

$$EQ_{ij} = 332.0716 q_i q_j / (D(R_{ij} + \omega)^n) \quad (3.7)$$

Unfortunately, the runtime for the non-bonded interactions grows exponentially since almost all atoms can interact with each other. To accelerate these components by several magnitudes, we added support for cutoff distances, beyond which non-bonded interactions are ignored.

To obtain the force equations (see Appendix A), we differentiated the above energy equations. For some components, like the torsion and out-of-plane terms, several ways can be thought of for distributing the arising forces on the individual atoms. Therefore, we compared our equations with the MMFF94 implementation in the CHARMM package, where our results showed to be consistent with the original values.

3.3.3. Molecular editing

The rational design and modification of molecular structures requires the availability of powerful molecular editing functionality. It can not only lead to deeper insights into molecular interactions but can also be used for developing new lead structures. To the best of our knowledge only a few commercial software tools like HyperChem [16] offer full editing functionality. Therefore, we decided to supplement BALLView with molecular editing functionality that is comparable to the best of the currently available tools. Thus, users can now freely add, move, delete, and modify atoms and bonds. As an example, it is possible to add atoms and bonds with one click and a double click can change an atom's element or transform bond orders. In addition, the editing mode supports keyboard hotkeys, which can sometimes be faster than the corresponding mouse actions. Table 3.3 gives an overview of the supported keys and their triggered actions.

More sophisticated features aim at automating and accelerating the editing process:

Templates

BALLView provides template compounds, like simple aromatic rings, nucleotides or amino acids, that ease the construction of larger structures. The templates can be selected from a list, freely placed in the 3D view and then connected with bonds. BALLView also provides an interface to the large ligand data set in the PubChem database [33].

The interface allows for searching by keywords or SMARTS expressions and the search results can be previewed as two-dimensional thumbnails, then downloaded, and placed. Furthermore, we added support for creating ligands by using SMILES expressions.

Assigning aromatic rings

As described on page 83, we implemented a Kekulizer algorithm for transforming "aromatic" bond type assignments into conjugated single/double bonds. This Kekulizer algorithm is also useful for the editing mode. First, users can mark a ring system as aromatic, by double-clicking on it. Second, the Kekulizer places single and double bonds. This makes the editing process faster and more convenient.

Adding of hydrogen atoms

To simplify the creation of new ligands, we developed an algorithm for saturating compounds with hydrogen atoms. This new approach does not rely on predefined templates, like amino acids. Instead, for every atom, the number of preferred valence electrons is calculated, based on the connectivity, formal charge, and group in the periodic table of elements. To place the new hydrogen atoms, we first calculate the bond lengths through a modified Schomaker-Stevenson rule [66]. It is based on the electronegativities of the two binding partners and their atom radii. The exact placement of the new hydrogen atoms is based on their heavy-atom partners and the existing bond connectivity.

This version of the modified Schomaker-Stevenson rule is used in the MMFF94 force field for the calculation of reference bond lengths. The applied bond angles are mainly based on the standard reference bond angles for sp , sp^2 and sp^3 hybridized atoms, which can be found in any standard chemistry textbook (see e.g. [53]).

Quick optimization

Hand sketched molecules are always coarse since the bond lengths and angles can differ strongly from their values in the real world. Therefore, a tool is needed to quickly optimize the placement of the individual atoms. While most other programs like HyperChem [16] rely on simple heuristics, with limited accuracy, we chose another approach that applies the Merck force field. First, we calculate small random perturbations of the positions of all atoms. Then, we perform an energy minimization, followed by a short MD simulation, and another energy minimization. This usually results in more realistic structures than a minimization alone.

Peptide builder

While BALLView's general molecular editing capabilities are powerful, they are less suitable for building peptides. It would be a lengthy and tiresome operation to correctly

Mouse button	Functionality
Left click on empty space	Create a new atom
Left click on atom and drag	Move atom in X and Y direction.
Left click on atom and drag + Shift	Move atom in Z direction
Double click on atom	Set the atom's element
Double click on ring bond	Make whole ring aromatic
Double click on bond	Cycle through bond orders
Middle (or Left + Control)	Create bond
Right	Context menu for item under cursor
Mouse wheel	Zoom in or out

Key pressed	Effect
Escape	Switch to last mode (e.g. rotate mode)
H,N,C,O,P,S	Select the element for the next atom
D	Delete atom under cursor
Backspace	Delete bond under cursor

Table 3.3: Mouse and keyboard shortcuts in the edit mode. We ensured that the most important actions in the edit mode can be done with one key or mouse click.

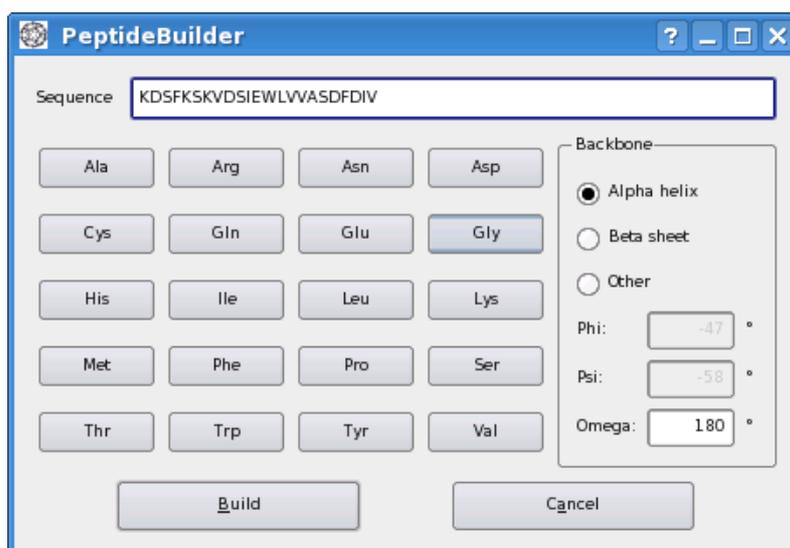


Figure 3.23: Dialog for creating peptides from a given amino acid sequence and angles.

place the individual amino acids and connect them with bonds. Instead, BALLView supports the more comfortable creation of peptides via a given amino acid sequence and the corresponding backbone dihedral angles. To realize this feature, we designed the PeptideDialog (see Fig. 3.23), which allows to quickly construct any peptide. (This functionality and also other parts of the editing mode stem from a student research project of Anne Dehof.)

3.3.4. Docking

The computational techniques in molecular modeling that predict how strong two molecules bind to another are called "molecular docking". While multiple methods exist for this kind of predictions, the most common ones are geometric approaches that compare the shapes of the two binding partners, molecular mechanics methods that model the interactions between the molecules, and genetic algorithms. Since molecular docking can find potential interaction partners, for instance by scanning large compound databases, it plays a significant role in the field of drug design. Therefore, we decided that BALLView should provide a graphical user interface for molecular docking. The corresponding functionality was then developed in the student research projects of Bettina Leonhardt and Carla Haid. As a result of this work, BALLView can be used to prepare a structure, run a docking algorithm and visualize its results, which can accelerate the workflow. All this functionality is available through a comfortable and intuitive graphical user interface. To start and setup a docking run, we provide a generic dialog. Here, a user can choose the two docking partners, the docking algorithm, and the scoring function. An additional tab provides access to preprocessing steps like adding hydrogens, building bonds, and assigning charges or radii. Of course, the options of a docking run can be stored to rerun it again, or to uncouple the docking's setup and its application. Users can thus apply the settings on one machine, then transfer the resulting configuration file to another dedicated computing machine and start the docking run there. For docking runs that are performed within BALLView, it offers a dialog that continuously provides detailed information about the current progress, the used setup options or any other data from the algorithm (see Fig. 3.25). Of course, this dialog also allows to abort the docking run and access the preliminary results. When a docking run is finished, another dialog provides detailed information about the results (see Fig. 3.26). The scoring of all computed potential complex conformations is shown. The same dialog provides the ability to rescore the conformations or to carry out a finer sampling around one result.

By developing the above dialog, we achieved a graphical docking interface that is much easier to use than the complex configuration files in other applications. BALLView thus also allows absolute beginners to perform a docking run. Therefore, the docking interface is ideally suited for teaching purposes, especially for introducing students into molecular docking.

Unfortunately, the number of supported docking algorithms in BALLView is still very limited, since it currently only supports a variant of the Katchalski-Katzir et al. algorithm [71] for protein-protein docking based on the geometric conformation of the two docking partners. But work is under way to add further docking algorithms. In the near future a genetic protein-ligand docking approach similar to AutoDock [60] and GOLD [101] will be added to BALLView. Furthermore, we are currently extending the docking interface with sup-

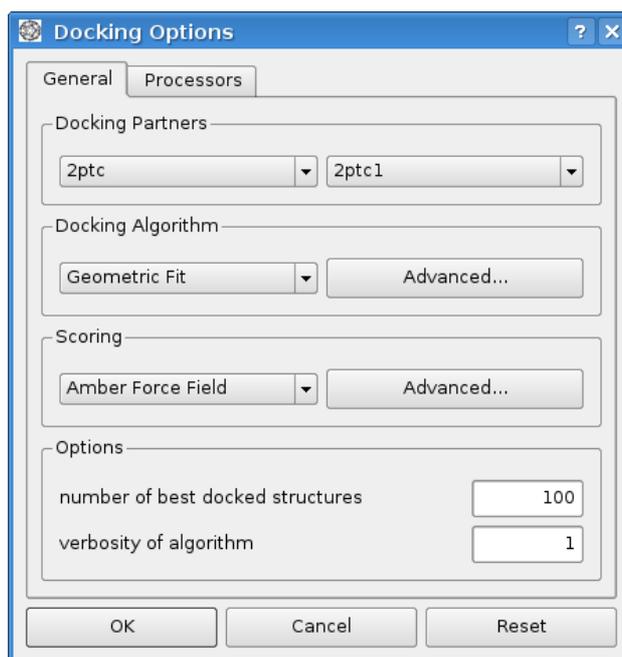


Figure 3.24: Dialog for starting a docking run. Users can choose the docking partners, the applied algorithm, and the scoring function. An additional tab provides access to preprocessing steps like adding hydrogens, building bonds, and assigning charges or radii.

port for parallel computing: future versions will allow the setup and monitoring of docking algorithms that run distributed on multiple processors or different machines.

3.3.5. Electrostatics calculation

As was already mentioned in Section 3.2.3, BALLView allows for calculating electrostatic potentials of arbitrary compounds through an integrated Finite-Difference Poisson Boltzmann (FDPB) solver [84]. To illustrate the ease of this feature, we will now describe the steps required to create a potential grid:

First, a molecular file is downloaded from the protein database [49] using BALLView's graphical user interface. Second, hydrogen atoms are added and optimized with one of the implemented force fields. This automatically assigns the initial charges to the individual atoms, which will be used to compute the potential. Finally, the dialog for the FDPB solver (see Fig. 3.27) can be used to compute a user-defined potential grid.

Since the preparation of the structure and the calculation of the potential can be done within one application, user do not have to switch between multiple tools or get acquainted with different file formats and interfaces. Therefore, it takes a user less than one minute to create a grid (the time for the minimization of the hydrogen atoms not accounted for). The

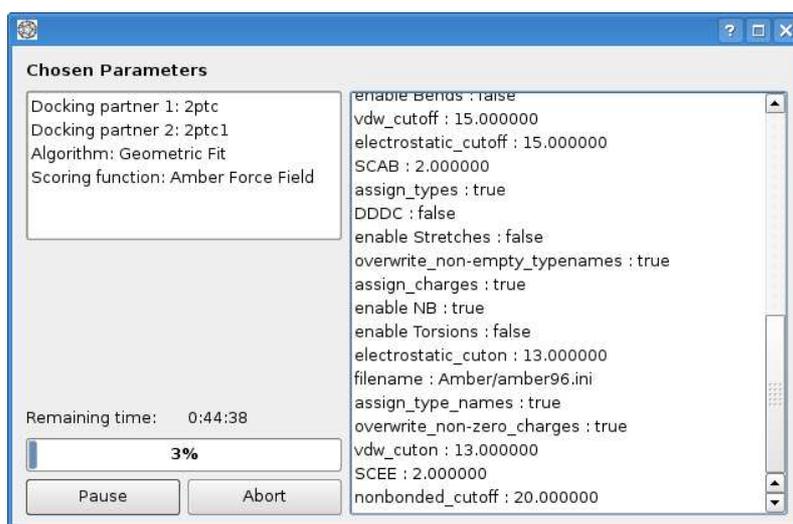


Figure 3.25: Dialog that informs about a docking run's current progress. It gives an overview of the used options as well as the estimated remaining run time and allows the user to abort the docking run.

resulting potential grids can then be exported into files for external usage, or visualized directly in BALLView with any of the visualization capabilities discussed in Section 3.2.3.

3.3.6. Comparison with related software

In the following pages, we will compare BALLView with some of the most advanced and sophisticated modeling tools:

Chimera [10] is not only a molecular viewer, but also a modeling tool that shares many features with BALLView: it supports assignment of hydrogen atoms and charges, energy minimizations and molecular editing. But many of these features still look unfinished and have serious design flaws and other drawbacks. For instance, users can not abort minimization runs. In addition, Chimera's editing mode does not support templates and it is not possible to change bond orders. Moreover, the handling is very cumbersome: atoms and bonds can not easily be placed or deleted with the mouse.

HyperChem [16] is a sophisticated molecular modeling environment that combines 3D visualization with quantum chemical calculations, semi-empiric models, molecular mechanics, and dynamics. Further features include a ligand database and a wide set of QSAR and other prediction functionality.

At its current state BALLView can not fully compete with this wide range of computational methods, since it is specialized in molecular mechanics and does not include quantum mechanics calculations. But current work is underway to add corresponding functionality

	Snapshot	Amber Force Field
1	0	-48946.1
2	1	-48770.3
3	2	-48763.1
4	3	-48622.9
5	4	-48586.5
6	5	-48509.4
7	6	-48477.6
8	7	-48474.8
9	8	-48443.4
10	9	-48442
11	10	-48422.3
12	11	-48422.2
13	12	-48417.2
14	13	-48404.5
15	14	-48402.7
16	15	-48386.1

The dialog box also features a 'Reranking' panel on the right with the following controls:

- A dropdown menu currently set to 'Amber Force Field'.
- Buttons: 'Advanced...', 'New Scoring', 'Scoring Options', 'Delete Scoring', 'Docking Options', 'Redock', 'Show snapshot', 'Up', 'Down', and 'Close'.

Figure 3.26: The results of a docking run are shown in a dialog that allows to rescore the calculated conformations.

to BALLView, like ligand database access and QSAR methods.

While HyperChem is without doubt a very powerful modeling tool, it still has some disadvantages. Compared with BALLView, HyperChem has an outdated and unintuitive GUI that slows users down. HyperChem's biggest drawback is probably its limited rendering functionality. Since it only provides a very limited spatial perception of the shown compounds, HyperChem is not well suited for modeling large molecules like proteins. In contrast, BALLView supports state-of-the-art visualization for any kind of structures and thus for instance allows to modify and model a ligand directly in its binding pocket.

MOE [23] and **SYBYL** [43] are very powerful commercial modeling suites which cost several thousands dollars per year. Both programs were developed and steadily improved over more than ten years and represent the current state-of-the-art in molecular modeling. While BALLView can not fully compete with their rich functionality, this section will nevertheless try to give a comparison. First of all, both commercial modeling suites also provide molecular visualization. Here BALLView can stand the comparison with MOE and SYBYL, both in terms of supported visualization methods, as well as the rendering quality. Next, if one compares the modeling functionality, one finds that all three applications have many features in common, like molecular mechanics, molecular docking, and the

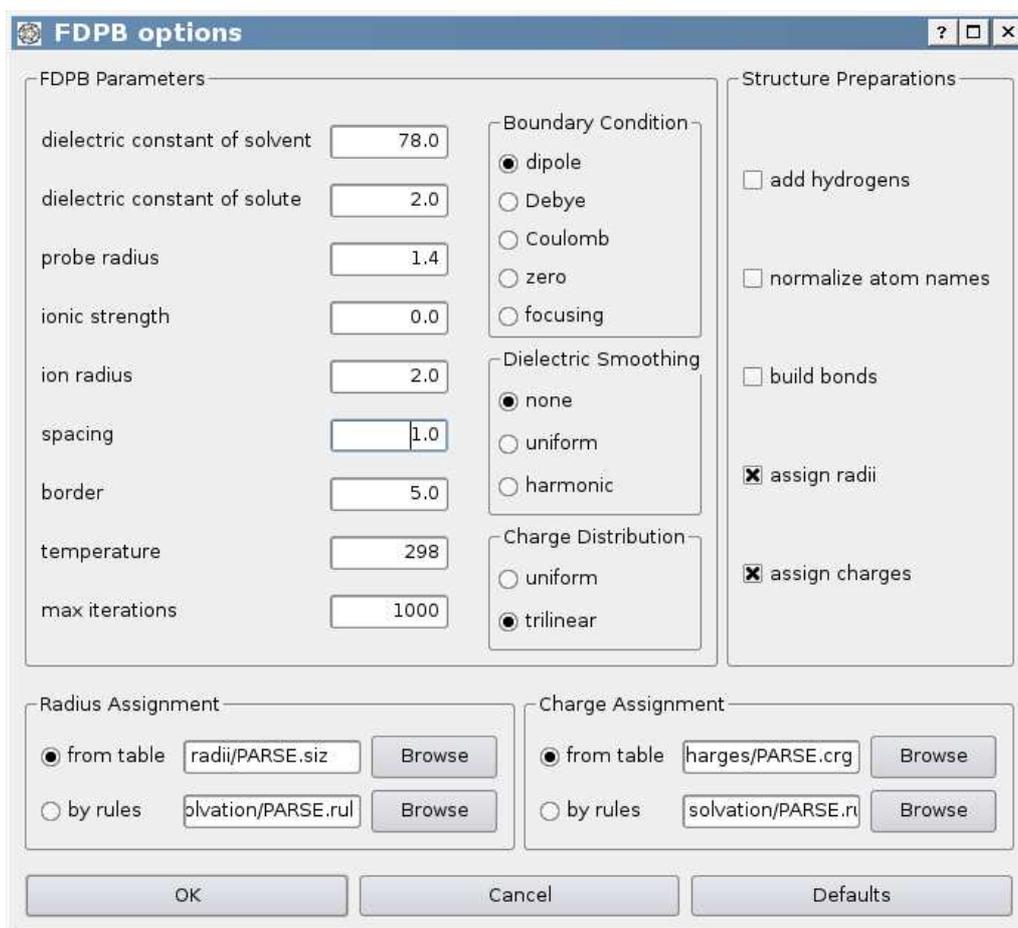


Figure 3.27: Configuration dialog for the FDPB solver.

calculation of electrostatics. While BALLView also provides this functionality, the commercial applications often provide better configurability and performance, a clear result of the companies man-power and the long development time.

3.4. Python interface

Scripting capabilities are of great importance for visualization and modeling programs since they allow users to extend these software tools with own functionality and to automate repetitive tasks. Fortunately, the BALL library was designed to provide an interface to Python [35]. Python is a powerful, object oriented scripting language for which many extensions exist, especially in the fields of scientific computing and engineering. Since Python it is an interpreted language it does not require recompilation of the code between changes. Thus, the overhead times for compiling, linking, and starting the application drop out which can significantly reduce the development time for new methods.

To realize the Python interface, the BALL and VIEW libraries provide a corresponding Python class for most of their C++ classes. These Python classes are semi-automatically generated by the wrapper generator SIP [40] and share a virtually identical interface with their C++ counterpart. Since Python is easy to learn and we used an object oriented design of the underlying C++ class interface, new users can get easily acquainted with the scripting interface.

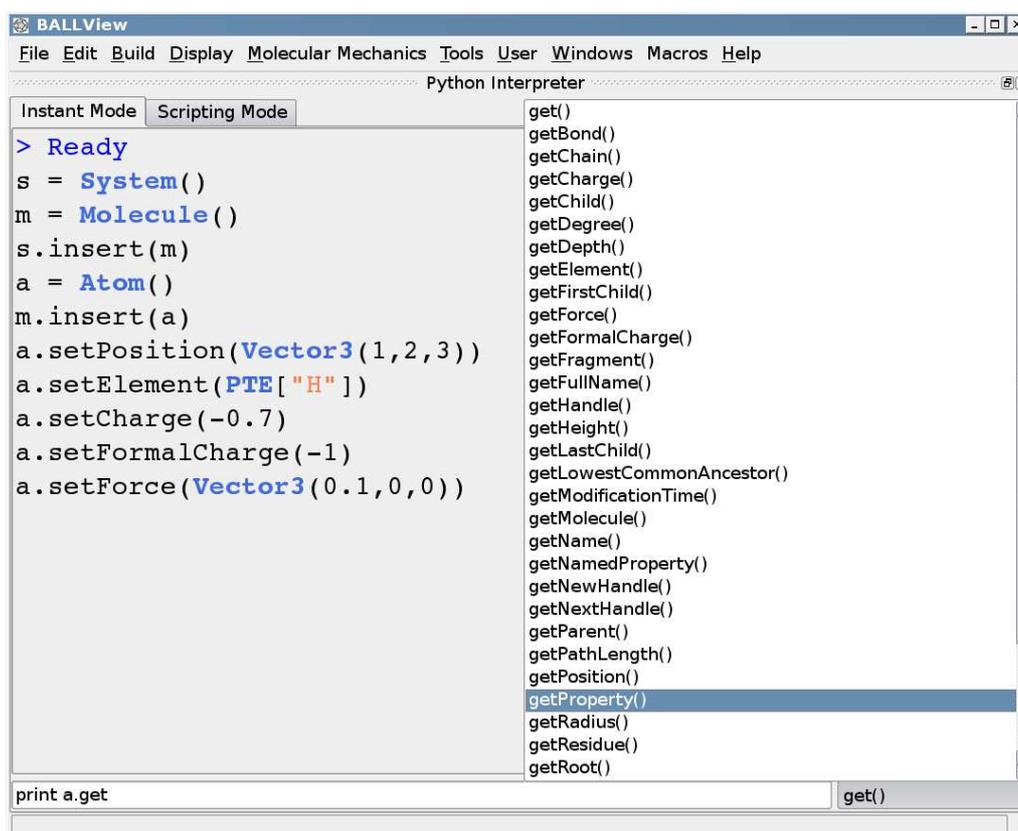


Figure 3.28: BALLView's graphical user interface reduced to the Python widget. It is divided into a window for instant interpretation and an editor widget for writing scripts (see Fig. 3.29). This figure illustrates some of the Integrated Developer Environment (IDE) features like syntax highlighting and completion.

In order to make Python available from within BALLView, we extended its graphical user interface with an embedded Python interpreter. The corresponding window is divided into two parts, represented by two widgets. The first one (see Fig. 3.28) allows for instant access to the Python interpreter, i.e. any entered line of code is processed when the return key is pressed. The second widget (see Fig. 3.29) enables the development and execution of sophisticated scripts. Users thus have full access to the rich functionality in the BALL and VIEW libraries as well as to Python's own modules. As a result, BALLView can be easily extended at runtime.

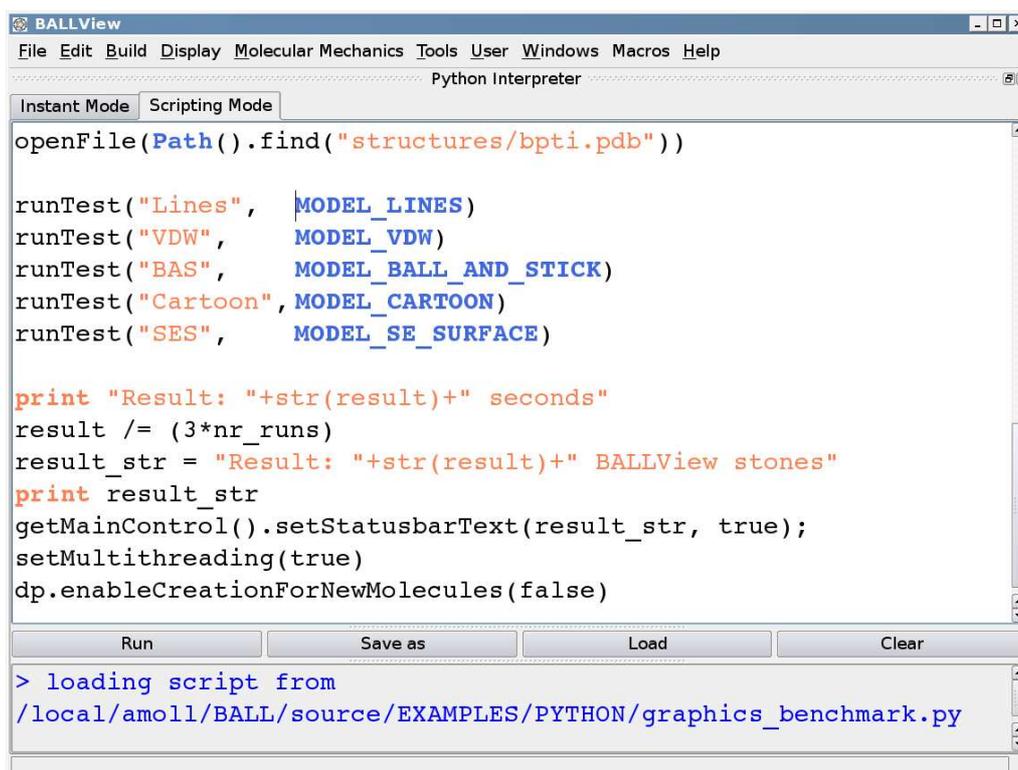


Figure 3.29: BALLView's editor for developing Python scripts. It supports user defined fonts, syntax highlighting and context sensitive help.

To further simplify the usage of the Python interface and accelerate the creation of scripts, BALLView provides Integrated Developer Environment (IDE) features, which will be described in the following text.

Since the BALL and VIEW libraries altogether consist of several hundred classes with thousands of members, even experienced users sometimes do not remember which functions a class provides. To help users in such situations we implemented a source code completion. It allows for the completion of classes and object names as well as the search for all methods in a given object, which match a given prefix (see Fig. 3.28). If for such a search multiple matches are found, they are shown in a combo box allowing users to choose the desired completion. If only one feasible completion is found, the correspond-

ing text is automatically inserted at the cursor's current position. The implementation of the completion feature works, by first parsing the text line that is to be completed. Here, the last object in front of the cursor position is searched for, based on Python's syntax. Next, the Python commands `__class__` and `dir()` are used to obtain the object's type and its members.

To make full use of the rich functionality in the BALL and VIEW libraries, users need an easy accessible documentation for the class interface. To this end we have implemented a syntax sensitive help. For the object or function at the current cursor position, it can show the corresponding documentation entry inside the application. This allows for a quick overview of a class's methods or the expected arguments for a given function. An example for such a documentation page is given in Fig. 3.2. The implementation of the syntax sensitive help works similar to the above described completion by parsing the text line and applying the same Python commands. Next, to find the corresponding help page, the BALL documentation's file names are compared with the class name of the object.

To better illustrate the Python code, the editor supports syntax highlighting. This can accelerate the development process and is especially useful for finding potential problems and errors in a script. If nevertheless the execution of a Python script fails, the Python widget parses the interpreter's output and attempts to identify and highlight the erroneous code fragment.

As a result of the above described approach for creating the Python bindings and the IDE features, which are not supported by any other molecular modeling or visualization tool, BALLView's scripting features are much easier to use and more powerful. Thus very sophisticated scripts can be written with very few code, as can be seen in the following examples.

Structure preparation and molecular mechanics

BALLView's Python interface is ideally suited for automating repetitive tasks. As an example, we present a typical script that reads a molecule from a PDB file, adds the missing hydrogen atoms, and selects them. Finally, a steepest-descent minimization is performed for a hundred steps on the hydrogen atoms.

```
pdb = PDBFile("bpti.pdb")
S = System()
pdb.read(S)
getMainControl().insert(S)
getMolecularStructure().addHydrogens()
getMolecularControl().applySelector("element(H)")
amber = getMolecularStructure().getAmberFF()
amber.setup(S)
m = SteepestDescentMinimizer(amber)
m.minimize(100)
```

```
getMainControl().update(S)
```

BALLView can be assigned a Python start-up script, which is applied every time the program starts. This can, e.g., be used to load given structures or perform any kind of setup actions via a user defined script similar to the above.

Access to external programs

While BALLView is already a powerful modeling tools, it still lacks some functionality like for instance quantum mechanics methods. These limitations can often be circumvented by starting external applications with data from within BALLView. To illustrate how this can be done, we wrote the following script. It creates a MOPAC [24] input file containing the atoms of the first system currently loaded in BALLView. It then calls MOPAC to execute a single-point calculation using the PM3 semi-empirical Hamiltonian. Finally, the script prints the resulting total energy. With just a few lines more, this example could be adapted to other quantum chemistry packages or to minimize the structure's energy and return the optimized structure.

```
S = getSystem(0)
text = "PM3 1SCF MMOK GEO-OK\n\n\n"
for i in atoms(S):
    e = i.getElement().getSymbol()
    p = i.getPosition()
    text += "%s %f 1 %f 1 %f 1\n" % (e, p.x, p.y, p.z)

open("tmp.dat", 'w').write(text)
os.system("run_mopac tmp")
print os.popen('grep "TOTAL ENER" tmp.out').read()
```

User built representations

Another application of the Python interface are user defined visualizations: If none of the predefined models should suit a user's needs or he wants to modify an existing model, Python allows him to produce customized visualizations. As an example, the Python interface can be used to quickly create single geometric objects, e.g. to visualize a bounding box. This requires only a few lines of code including the setup of the geometric object's position, radius, color, and transparency. Of course, more sophisticated applications can also be realized. As an illustration, Figure 3.30 shows the visualization of a trajectory through spheres for the positions of all non-hydrogen-atoms. This functionality was realized by the following script with 20 lines of code, including opening the molecule and the trajectory, computing the *Stick* model, iterating over all atoms, and placing the spheres:

```
dp = getDisplayProperties()
dp.selectModel(MODEL_STICK)
dp.selectColoringMethod(COLORING_ELEMENT)
openFile("AlaAla.hin")
dcd = DCDFile("alaala.dcd")
system = getSystem(0)
ssm = SnapShotManager(system, AmberFF(), dcd)
rep = Representation()
current_ss = 0

while ssm.applyNextSnapShot():
    current_ss += 1
    ratio = float(current_ss) / float(dcd.getNumberOfSnapShots())
    for atom in atoms(system, "!element(H)":
        sphere = Sphere()
        sphere.setPosition(atom.getPosition())
        sphere.setColor(ColorRGBA(1.0 - ratio, 1.0 - ratio, ratio))
        sphere.setRadius(0.03)
        rep.insert(sphere)

getMainControl().insert(rep)
getMainControl().update(rep)
```

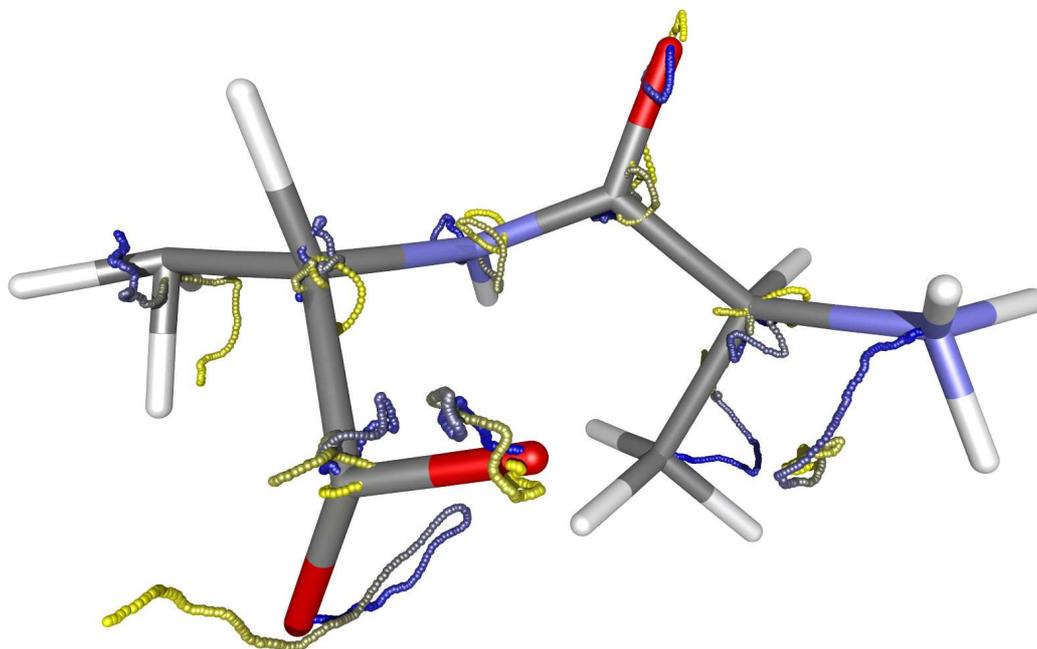


Figure 3.30: Example for the usage of the Python interface for visualizing the atom mobility in an MD simulation. Every individual sphere corresponds to the position of one non-hydrogen-atom at one trajectory step. The spheres were colored according to the step number, blue for the first step and yellow for the last step.

Movies

The Python scripting interface allows for the creation of movies to visualize complex dynamic data. We have written a variety of predefined scripts for different purposes:

- "Fly by" animations to show a molecule from varying angles [3]
- Moving of molecular subunits [2], e.g. for the simulation of docking processes
- Fading models in/out, for instance for smooth transitions
- Visualization of electrostatic potentials [4, 5]
- Time courses for instance from MD simulations [6]

Some results of these scripts are available for download from the project's website. All prebuilt scripts can easily be integrated to create movies that include the different animations. As a further advantage, they allow the adaptation of all important variables, like step width, transparency or colors.

Hotkeys with user assigned functionality

To further ease the usage of Python scripts, BALLView enables the user to map them to user-defined hotkeys (see Fig. 3.31) that allow for starting a specific script by pressing a single key.

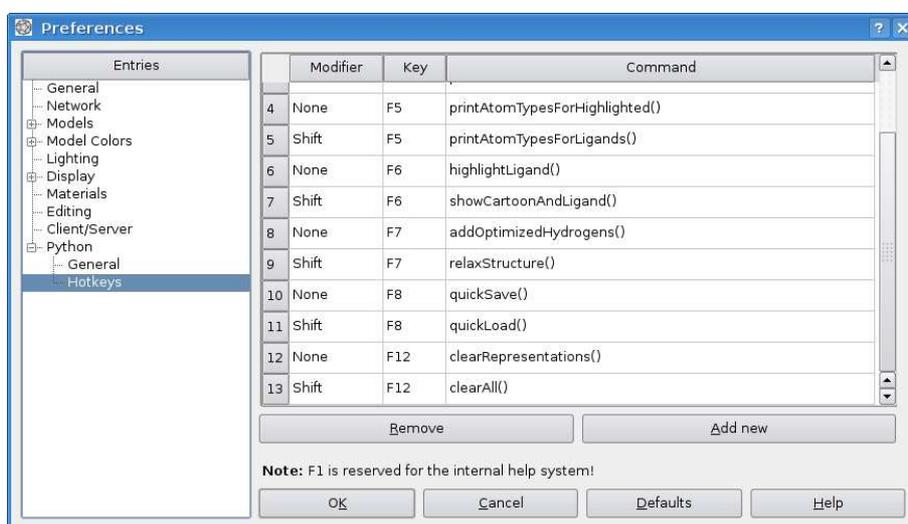


Figure 3.31: Dialog for mapping Python scripts to hotkeys. Any F-key can be assigned to any line of Python code.

In total, more than 30 user-defined scripts can be mapped to individual hotkeys. For those users that have difficulties to remember the mapping of the hotkeys, BALLView provides menu entries for starting the scripts. For around a dozen of standard task, we offer short

scripts, mapped on predefined hotkeys. Thus, it only takes one key press to remove all water molecules or to add optimized hydrogen atoms.

Conclusion

As we have shown in the above examples, the development of new features with the Python scripting interface is straightforward. When the development of a new feature is finished, the Python code can easily be transformed into C++, since all classes share the same interface (i.e. methods and members) in the both languages. Python scripts that have been transformed into C++ code, can then benefit of the better runtime performance and eventually may become part of the standard BALL library.

4. Conclusion and discussion

Based on our C++ software library BALL, we created the powerful molecular viewer and modeling tool BALLView. BALL was specifically developed for rapid prototyping of applications in molecular modeling and structural bioinformatics. Therefore, BALLView benefits from BALL's functionality in the field of molecular modeling and in particular molecular mechanics: three different force fields (AMBER, CHARMM, and MMFF94) can be used for molecular mechanics simulations, energy minimizations, and molecular docking. Furthermore, an integrated Poisson-Boltzmann solver allows for calculating electrostatic potentials which can be directly visualized in a variety of ways. In addition, we incorporated molecular editing features for creating and modifying polypeptides or other organic compounds. Since all these features are offered through one common interface, BALLView can accelerate the development of lead structures or the identification of potential drug targets.

Molecular modeling is a multi-disciplinary field in which researchers with very different backgrounds are involved. Therefore, great efforts have to be made to make molecular modeling applications easy to understand and use. Unfortunately, most commonly available modeling tools have severe usability problems. In contrast, we ensured that our software can be easily operated: beside offering all standard graphical user interface features, BALLView also provides more sophisticated support for the user, like a context sensitive help.

In addition to its modeling functionality, BALLView is also a molecular viewer with state-of-the-art graphics. While this combination of molecular visualization and modeling capabilities is still rather uncommon, it has the major advantage that users no longer need to rely on multiple applications to calculate and visualize their molecular data. Thus, they do not have to exchange data between the different tools via file operations. Consequently, over the last years some molecular viewers like Chimera were extended with molecular modeling functionality, but the resulting features are still somewhat immature (see Page 94). In contrast, BALLView from its very start was designed to combine both domains. As a result, BALLView offers its users a much wider range of modeling functionality. Moreover, we developed new visualization methods to increase the information density of molecular visualizations. BALLView allows for the rendering of many different structural features simultaneously. This includes molecular models, automatic and persistent labels, resulting forces, potential fields, and many other molecular properties. Moreover, users can render

as many representations at the same time as they like. In doing so, they can benefit from advanced features like transparency and wireframe models such that the individual models can be shown without hidden each other. Therefore, BALLView may allow for a deeper understanding of the underlying mechanisms and processes in molecular structures.

All above-mentioned features are not only accessible through an intuitive graphical user interface, but also through the object-oriented scripting language Python. To make this interface more powerful, we developed a new concept for a tight interlocking of scripting and GUI functionality. First, we extended BALLView with a graphical frontend to the Python scripting language. This frontend was designed as an Integrated Development Environment (IDE) with syntax highlighting, completion, and context sensitive help such that even inexperienced users can develop own scripts. Second, we developed the necessary means such that the Python interface enables the modification of the graphical user interface, for instance the creation of user-defined shortcuts and menu entries. This eases the automation of repetitive tasks since user defined scripts can be started with one key stroke. But the usage of the scripting interface is not limited to the access of predefined functionality. Since the rich set of features in the BALL and VIEW libraries are available through Python, users can develop complex new applications with a few lines of Python code. Hence, BALLView is a Problem Solving Environment (PSE) in the field of molecular modeling: for many problems/theories in this field, researchers can quickly implement a model/solution by using BALL's built-in features and the rapid prototyping capabilities through Python. The resulting data can be visualized directly in the application which significantly eases the analysis. Moreover, the scripting interface can be used to inspect any of the underlying data in realtime. Any findings can then be incorporated to refine the model and the development cycle can be restarted without ever leaving the graphical user interface. This makes BALLView unique in the field of molecular modeling applications.

In addition to the stand-alone program BALLView, we designed the visualization functionality as an extensible software framework that can easily be adapted to the needs of users and is available under an Open Source license (GPL). Thus, for the first time, a free software package is available that combines state-of-the art visualization and modeling functionality, high extensibility, as well as a unique scripting interface (see Fig. 1.2). This enables other molecular modeling research groups to base their own developments on our software and thus save much time and effort. As a result, we may see an accelerated progress in biochemistry and structural bioinformatics, since researchers can concentrate on their main fields of interests, instead of having to implement the same functionality over and over again.

Over the last years, we have successfully used BALL and BALLView in graduate and undergraduate courses on computer-aided drug design, molecular modeling, structural bioinformatics, and protein structure. In our experience, BALLView has proven to be an ideal tool for teaching in these fields. This is mainly due to the intuitive interface which lowers the barriers usually imposed by the difficult handling of many standard tools. The teaching could thus focus on the methods and the theory behind them rather than spending too much time on interface and file format issues. Moreover, BALL and BALLView served as a basis for many student exercises on implementations and algorithms. Here, they allowed a faster introduction of the key issues in structural bioinformatics. Tasks which could not be implemented before due to the amount of program code necessary were often realized with a few lines of Python or C++ code. Due to the modularity of our approach, the results of the student projects could then be refined and became integral parts of BALL and BALLView.

In addition to the application in teaching, BALLView was also intended as basis for our research in the fields of molecular modeling and visualization. Here, it accelerated many different projects, like the development of new docking algorithms and energy minimizers as well as the implementation of non-local electrostatic potentials. Therefore, we can conclude that the underlying modular framework is well suited for most research projects in structural bioinformatics. Moreover, BALLView showed to be an excellent tool for creating sophisticated images for publications.

While we can only guess the number of BALLView's users, we had a substantial amount of downloads of the binary installers and source packages: in total around 6500 over the last three years. Moreover, we presented BALLView at the major bioinformatics conferences and got positive comments. Most people were impressed by the detailed and sophisticated visualizations. We also received a warm welcome in the Debian mailing list where several people intend to port BALL and BALLView to Debian. As a result, BALLView will become a standard Debian package in the near future. In addition, it will be incorporated in Debian-Med and DebiChem, two specialized Debian distributions for biochemistry-related software.

Outlook

While BALLView is already a powerful visualization and modeling tool, it still has huge potential to grow. Currently, we are working on adding further functionality, in particular in the following fields:

- 2D depiction
- automatic structure generation from SMILES expressions
- alignment of structures

- detection of binding pockets
- QSAR and CoMFA functionality
- advanced non-local electrostatics calculations
- database interfaces
- new, more sophisticated minimizers
- adding of further docking algorithms for protein-ligand docking
- new features for homology modeling

Many of these features will already be included in the next major BALLView release. In addition to this already running work other tasks still remain to be done:

Over the last years, we had rather long development cycles with up to 18 months between stable releases. We plan to shorten these periods to significantly less than a year. To accelerate the development cycle we will reintroduce automatic testing of daily builds. Here, the new automatic build process under windows and the GUI testing framework will make the testing even more powerful. Thus, we will be able to find errors earlier in the development phase.

A current trend in computer graphics (CG) are the programmable shaders of modern graphics accelerator cards, available for instance through the "OpenGL Shading Language". Up to now we have not yet worked on adding corresponding functionality to BALLView, since we focused on visualization techniques that are supported by almost all graphics cards. Since now all currently sold cards provide this functionality, we could add features like "Real Time Ambient Occlusion" or "Depth Aware Silhouette Enhancement" which are, for instance, offered by QuteMol [37] to provide better 3D perception. Moreover, the programmable shaders could improve the rendering of volumetric data sets. Here, they would allow a more appealing look and a better rendering performance. To make best use of the volume rendering functionality, we will add support for standard volumetric data file formats (e.g. the Gaussian cube file format). Another trend in CG goes to realtime ray-tracing [102]. Adding support for such a realtime raytracer engine could result in more fluent and more realistic graphics with improved lighting, arbitrary shadows as well as a higher level of detail. Since the underlying VIEW framework is fully modular, any of these extensions could be realized without significant changes to the existing code.

In the field of molecular modeling, we have seen the emergence of new hardware systems that revolutionize complex calculations like MD simulations. By utilizing the capabilities of the new, high performance Graphics Processing Units (GPUs) or the new Cell processor,

developers were able to achieve performances that previously were only possible on supercomputers. The next step in this direction are chips that incorporate vast amounts of cores in one processor like, for instance, Polaris, the new experimental 80-core chip from Intel. But also low-budget multi-core processors are getting more and more common. As a result, for future molecular modeling applications, it will be crucial to make efficient use of multi-threading. While we already incorporated such techniques in BALLView, we will need to add support for subdividing computations like MD simulations such that the applications can run in parallel on multiple cores. Otherwise, we will risk that our software becomes obsolete. In addition, BALLView should obtain support for distributed computing. Here, it could enable the efficient management of these calculations by providing the graphical frontend for starting the computations on the individual machines, the presentation of intermediate results, and the analysis of the final results.

Another field is the support for sequence data. In particular, we should add multiple sequence alignment functionality. BALLView could then be used to organize, display, and analyze both sequence and structural data. Thus, it would be better suited for processes like homology modeling. Furthermore, we could add collaborative working environment features, additional data base functionality and support for external programs. In particular, we should develop wrapper scripts or graphical interfaces to ab initio quantum mechanics packages, like GAMESS and Gaussian.

While developing BALLView, special care was taken to achieve a high usability, but we still can do better, especially much work remains to be done for the documentation. In the future, it should contain detailed information for every single parameter in the different computations. The goal here is that also users with less knowledge in the individual fields can efficiently work with BALLView. Moreover, we will start developing teaching materials like stepwise tutorials such that pupils and students can learn the elementary mechanism in biochemistry and molecular modeling. BALLView may then become a standard teaching aid in high schools and universities.

As shown above, still many challenges await us. But, with the presented work we laid the proper groundwork for these challenges. Since BALL, VIEW and BALLVIEW will be continued at two bioinformatics chairs (of Prof. Lenhof and Prof. Kohlbacher), their development will proceed for many years such that the above issues can be addressed in the future.

A. MMFF94 forces

The following equations define the forces for the individual MMFF94 components. Vectors are shown in bold, like \mathbf{d}_{ij} for the vector from atom i to atom j . Normalized vectors are shown with a "hat", like $\hat{\mathbf{d}}_{ij}$. The forces were derived from the potential's negative gradient:

$$\vec{F} = -\vec{\Delta}V$$

For a further description on the individual variables see Page 87.

Stretch forces

$$S_{ij} = \frac{143.9325}{2} kb_{IJ} \Delta r_{ij} \left(2 + 3 cs \Delta r_{ij} - \frac{14}{3} cs \Delta r_{ij}^2 \right)$$

$$\mathbf{FS}_i = -\mathbf{FS}_j = S_{ij} \frac{\hat{\mathbf{d}}_{ij}}{|\mathbf{d}_{ij}|}$$

Bend forces

$$B_{ijk} = \frac{0.043844}{2} ka_{IJK} \Delta \vartheta_{ijk} (2 + 3cb \Delta \vartheta_{ijk})$$

$$\mathbf{FB}_i = B_{ijk} \frac{\hat{\mathbf{d}}_{ij} \times \hat{\mathbf{d}}_{ki} \times \hat{\mathbf{d}}_{ij}}{|\mathbf{d}_{ij}|}$$

$$\mathbf{FB}_k = B_{ijk} \frac{\hat{\mathbf{d}}_{ij} \times \hat{\mathbf{d}}_{ki} \times \hat{\mathbf{d}}_{kj}}{|\mathbf{d}_{kj}|}$$

$$\mathbf{FB}_j = -\mathbf{FB}_i - \mathbf{FB}_k$$

Stretch Bend forces

$$A_{ijk} = -(2.51210)^2 (kba_{IJK} \Delta r_{ij} + kba_{KJI} \Delta r_{kj}) \Delta \vartheta_{ijk}$$

$$\mathbf{FSB}_i = A_{ijk} kba_{IJK} \frac{\hat{\mathbf{d}}_{ij} \times \hat{\mathbf{d}}_{ki} \times \hat{\mathbf{d}}_{ij}}{|\mathbf{d}_{ij}|}$$

$$\mathbf{FSB}_k = A_{ijk} kba_{KJI} \frac{\hat{\mathbf{d}}_{ij} \times \hat{\mathbf{d}}_{ki} \times \hat{\mathbf{d}}_{kj}}{|\mathbf{d}_{kj}|}$$

$$\mathbf{FSB}_j = -\mathbf{FSB}_i - \mathbf{FSB}_k$$

Torsion forces

$$T_{ijkl} = V_1 \sin(\phi) - 2V_2 \sin(2\phi) + 3V_3 \sin(3\phi)$$

$$\mathbf{FT}_i = T_{ijkl} \frac{-\hat{\mathbf{d}}_{ij} \times \hat{\mathbf{d}}_{jk}}{\sin(\phi)^2 |\mathbf{d}_{ij}|}$$

$$\mathbf{FT}_l = T_{ijkl} \frac{\hat{\mathbf{d}}_{jk} \times \hat{\mathbf{d}}_{kl}}{\sin(\phi)^2 |\mathbf{d}_{kl}|}$$

$$\mathbf{FT}_j = \mathbf{FT}_i \frac{|\mathbf{d}_{ij}|}{|\mathbf{d}_{jk}| (-\cos(\phi))} - 1 - \mathbf{FT}_l \frac{|\mathbf{d}_{kl}|}{|\mathbf{d}_{jk}| (-\cos(\phi))}$$

$$\mathbf{FT}_k = -(\mathbf{FT}_i + \mathbf{FT}_j + \mathbf{FT}_l)$$

Out-of-plane forces

$$O_{ijkl} = \frac{0.043844 \chi_{ijk;l} * \text{koop}_{IJK;L}}{\cos(\chi_{ijk;l})}$$

$$\mathbf{FO}_i = O_{ijkl} \frac{\hat{\mathbf{d}}_{jk} \times \hat{\mathbf{d}}_{jl} + (-\hat{\mathbf{d}}_{ji} + (\hat{\mathbf{d}}_{jk} \cos(\vartheta_{ijk}) \sin(\chi_{ijk;l}) / \sin(\vartheta_{ijk}))}{|\mathbf{d}_{ji}| \sin(\vartheta_{ijk})}$$

$$\mathbf{FO}_k = O_{ijkl} \frac{\hat{\mathbf{d}}_{jl} \times \hat{\mathbf{d}}_{ji} + (-\hat{\mathbf{d}}_{jk} + (\hat{\mathbf{d}}_{ji} \cos(\vartheta_{ijk}) \sin(\chi_{ijk;l}) / \sin(\vartheta_{ijk}))}{|\mathbf{d}_{jk}| \sin(\vartheta_{ijk})}$$

$$\mathbf{FO}_l = O_{ijkl} \frac{\hat{\mathbf{d}}_{ji} \times \hat{\mathbf{d}}_{jl} / \sin(\vartheta_{ijk}) - \hat{\mathbf{d}}_{jl} \sin(\chi_{ijk;l})}{|\mathbf{d}_{jl}|}$$

$$\mathbf{FO}_j = -(\mathbf{FO}_i + \mathbf{FO}_k + \mathbf{FO}_l)$$

VDW forces

$$q = R_{ij} / R_{IJ}^*$$

$$V_{ij} = \frac{\epsilon_{ij}}{R_{IJ}^*} \left(\frac{1.07}{q + 0.07} \right)^7 \left(\frac{-7.84 q^6}{(q^7 + 0.12)^2} + \frac{-7.84 / (q^7 + 0.12) + 14}{q + 0.07} \right)$$

$$\mathbf{FV}_i = -\mathbf{FV}_j = V_{ij} \hat{\mathbf{d}}_{ij}$$

Electrostatic forces

$$E_{ij} = 332.0716 \frac{q_i q_j^n}{D (R_{ij} + \omega)^{n+1}}$$

$$\mathbf{F}_i = -\mathbf{F}_j = E_{ij} \hat{\mathbf{d}}_{ij}$$

Bibliography

- [1] AVS visualization software: <http://www.avs.com>.
- [2] BALLView movie (animation): <http://www.ballview.org/Gallery/DNA.avi>.
- [3] BALLView movie (fly by): <http://www.ballview.org/Gallery/ribosome.avi>.
- [4] BALLView movie (isosurface): <http://www.ballview.org/Gallery/fdpb.avi>.
- [5] BALLView movie (project plane): <http://www.ballview.org/Gallery/1FB7.avi>.
- [6] BALLView movie (trajectory): <http://www.ballview.org/Gallery/mds.avi>.
- [7] BALLView website: <http://www.ballview.org>.
- [8] Cel shading:
<http://www.gamedev.net/reference/programming/features/celshading>.
- [9] Cerius: <http://www.accelrys.com/cerius2/>.
- [10] Chimera: <http://www.cgl.ucsf.edu/chimera/>.
- [11] Discovery Studio: <http://www.accelrys.com/products/dstudio/index.html>.
- [12] Doxygen: <http://sourceforge.net/projects/doxygen/>.
- [13] Ghemical: <http://www.uku.fi/~thassine/projects/ghemical/>.
- [14] GLEW: <http://glew.sourceforge.net>.
- [15] GNU: <http://www.gnu.org>.
- [16] HyperChem: <http://www.chemistry-software.com/>.
- [17] Interface design principles:
<http://www.asktog.com/basics/firstPrinciples.html>.
- [18] JOELib 2: <http://www-ra.informatik.uni-tuebingen.de/software/joelib>.
- [19] KD Executor: <http://kdab.net/kdexecutor>.
- [20] Mencoder: <http://www.mplayerhq.hu>.

- [21] Meshi: <http://www.cs.bgu.ac.il/~meshi>.
- [22] MMFF94 validation suite: <http://ftp.ccl.net/cca/data/MMFF94/index.shtml>.
- [23] Molecular Operating Environment (MOE): <http://www.chemcomp.com/>.
- [24] MOPAC: <http://www.cachesoftware.com/mopac/index.shtml>.
- [25] Open Inventor: <http://oss.sgi.com/projects/inventor/>.
- [26] OpenGL library: <http://www.opengl.org>.
- [27] OpenGL tutorial:
<http://nehe.gamedev.net/data/lessons/lesson.asp?lesson=37>.
- [28] OpenMOIV: <http://www.tecn.upf.es/openMOIV>.
- [29] Overview of molecular visualization and modeling tools:
<http://molvis.sdsc.edu/visres/molvisfw/titles.jsp>.
- [30] Patterns of Enterprise Application Architecture:
<http://martinfowler.com/eaaDev>.
- [31] PNG file format: <http://www.libpng.org/pub/png>.
- [32] POV-Ray renderer: <http://www.povray.org>.
- [33] PubChem: <http://pubchem.ncbi.nlm.nih.gov>.
- [34] Purify: <http://www.ibm.com/software/awdtools/purifyplus>.
- [35] Python scripting language: <http://www.python.org>.
- [36] Qt library: <http://www.trolltech.com>.
- [37] QuteMol: <http://qutemol.sourceforge.net>.
- [38] Raster3D: <http://skuld.bmsc.washington.edu/raster3d>.
- [39] Rational Rose: <http://www.ibm.com/software/rational>.
- [40] SIP (Python bindings generator):
<http://www.riverbankcomputing.co.uk/sip>.
- [41] SMARTS:
<http://www.daylight.com/dayhtml/doc/theory/theory.smarts.html>.
- [42] Squish: <http://www.froglogic.com>.
- [43] SYBYL 7.0, Tripos Inc., 1699 South Hanley Rd., St. Louis, Missouri, 63144, USA.

- [44] Valgrind: <http://valgrind.org>.
- [45] Virtualdub: <http://www.virtualdub.org>.
- [46] Website on usability: <http://www.useit.com>.
- [47] International Standard ISO/IEC 14882: Programming languages – C++. American National Standards Institute, 11 West 42nd Street, New York 10036, 1998.
- [48] N. L. Allinger, Y. H. Yuh, and J. H. Lii. Molecular Mechanics. The MM3 Force Field for Hydrocarbons. 1. *J. Am. Chem. Soc.*, 111, 8551-8565., 1989.
- [49] 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, 2000.
- [50] F. Bernstein, T. Koetzle, G. Williams, E. Meyer Jr, M. Brice, J. Rodgers, O. Kennard, T. Shimanouchi, and M. Tasumi. The protein data bank: a computer-based archival file for macromolecular structures. *Eur. J. Biochem.*, 80:319–24, 1977.
- [51] N. P. Boghossian, O. Kohlbacher, and H. P. Lenhof. Rapid software prototyping in molecular modeling using the biochemical algorithms library (BALL). *J. Exp. Algorithmics*, page 16, 2000.
- [52] 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.*, 4:187–217, 1983.
- [53] P. Bruice. *Organic Chemistry*. Prentice Hall, Upper Saddle River, NJ, 2003.
- [54] U. Burkert and N. L. Allinger. *Molecular Mechanics*. ACS Monograph, Washington, D.C., 1982.
- [55] W.D. Cornell, P. Cieplak, C.I. Bayly, I.R. Gould, K.M. Merz, 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.*, 117:5179–5197, 1995.
- [56] M. Dahlheimer. *Programming with Qt*. O'Reilly, Sebastopol, 2002.
- [57] W.L. DeLano. The PyMOL molecular graphics system, 2002.
- [58] J. Wang et al. Development and testing of a general AMBER force field. *J. Computat. Chem.*, 25, 1157-1174, 2004.
- [59] M. Fowler and K. Scott. *UML distilled*. Addison-Wesley, Boston, 1997.

- [60] D.S. Goodsell, G.M. Morris, and A.J. Olson. Automated docking of flexible ligands: applications of autodock. *J. Mol. Recognit.*, pages 9, 1–5., 1996.
- [61] N. Guex and M.C. Peitsch. SWISS-MODEL and the Swiss-PdbViewer: An environment for comparative protein modeling. *Electrophoresis*, 18:2714–2723, 1997.
- [62] T. Halgren. Merck Molecular Force Field. I. Basis, Form, Scope, Parameterization and Performance of MMFF94. *J. Comput. Chem.*, 17:490–519, 1996.
- [63] T. Halgren. Merck molecular force field. II. MMFF94 van der Waals and electrostatic parameters for intermolecular interactions. *J. Comput. Chem.*, 17:520–552, 1996.
- [64] T. Halgren. Merck molecular force field. III. Molecular geometries and vibrational frequencies for MMFF94. *J. Comput. Chem.*, 17:553–586, 1996.
- [65] T. Halgren. Merck molecular force field. IV. Conformational energies and geometries for MMFF94. *J. Comput. Chem.*, 17:587–615, 1996.
- [66] T. Halgren. Merck molecular force field. V. Extension of MMFF94 using experimental data, additional computational data, and empirical rules. *J. Comput. Chem.*, 17:616–641, 1996.
- [67] W. Humphrey, A. Dalke, and K. Schulten. VMD – Visual Molecular Dynamics. *J. Mol. Graphics*, 14:33–38, 1996.
- [68] J. J. Irwin and B. K. Shoichet. ZINC - A free database of commercially available compounds for virtual screening. *J. Chem. Inf. Comput. Sci.*, 45:177-82, 2005.
- [69] J. Johnson. *GUI Bloopers*. Morgan Kaufmann, San Fransisco, 2000.
- [70] W. Kabsch and C. Sander. Dictionary of protein secondary structure: Pattern recognition of hydrogen bonded and geometrical features. *Biopolymers*, 22:2577–2637, 1983.
- [71] E. Katchalski-Katzir, I. Shariv, M. Eisenstein, A. Friesem, C. Aflalo, and I. Vakser. Molecular surface recognition: determination of geometric fit between proteins and their ligands by correlation techniques. *Proc. Natl. Acad. Sci. USA*, pages 2195–2199, 1992.
- [72] F. A. Kekulé. Untersuchungen über aromatische Verbindungen. *Liebigs Annalen der Chemie*, 137, 129-36, 1866.
- [73] O. Kohlbacher. *New approaches to protein docking*. Dissertation, Saarland University, Saarbrücken, 2001.

- [74] O. Kohlbacher, A. Burchardt, A. Moll, A. Hildebrandt, P. Bayer, and H.P. Lenhof. A NMR-spectra-based scoring function for protein docking. In *RECOMB 2001 – Proceedings of the Fifth Annual International Conference on Computational Molecular Biology*, pages 169–177. ACM press, 2001.
- [75] O. Kohlbacher, A. Burchardt, A. Moll, A. Hildebrandt, P. Bayer, and H.P. Lenhof. Structure prediction of protein complexes by a NMR-based protein docking algorithm. *J. Biomol. NMR*, 20:15–21, 2001.
- [76] O. Kohlbacher and H.P. Lenhof. BALL - Rapid Software Prototyping in Computational Molecular Biology. *Bioinformatics*, 16:815–824, 2000.
- [77] R. W. Kunz. *Molecular Modelling für Anwender*. Teubner Studienbücher Chemie, Stuttgart, 1997.
- [78] A. Leach. *Molecular Modeling: Principles and Applications*. Addison-Wesley, Boston, 1996.
- [79] S. Meyers. *Effective STL*. Addison-Wesley, Boston, 2001.
- [80] S. Meyers. *Effective C++. 55 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley, Boston, 2005.
- [81] A. Moll, A. Hildebrandt, H.P. Lenhof, and O. Kohlbacher. BALLView: a tool for research and education in molecular modeling. *Bioinformatics*, 22:365–6, 2006.
- [82] A. Moll, A. Hildebrandt, H.P. Lenhof, and O. Kohlbacher. BALLView: an object-oriented molecular visualization and modeling framework. *J. Comput. Aided. Mol. Des.*, 19:791–800, 2006.
- [83] D. Musser, A. Saini, and G. Derge. *STL tutorial and reference guide : C++ programming with the standard template library*. Addison-Wesley, Boston, 2001.
- [84] A. Nicholls and B. Honig. A rapid finite difference algorithm, utilizing successive over-relaxation to solve the poisson-boltzmann equation. *J. Comput. Chem.*, 12:435–445, 1990.
- [85] A. Nicholls, K. Sharp, and B. Honig. Protein folding and association: Insights from the interfacial and thermodynamic properties of hydrocarbons. *Proteins: Struct. Funct. Genet.*, 11:281ff, 1991.
- [86] J. Nielsen. *Usability Engineering*. Morgan Kaufman, San Francisco, 1994.
- [87] S. R. Niketic and K. Rasmussen. The Consistent Force Field: A Documentation. *Lecture Notes in Chemistry*, 37, vii + 212 p., 1977.

- [88] Object Management Group. Unified modeling language specification, 1998. <http://www.omg.org>.
- [89] Institute of Electrical and New York Electronics Engineers. IEEE standard computer dictionary: A compilation of IEEE standard computer glossaries, 1990.
- [90] H. Pradeep. *User Centered Information Design for Improved Software Usability*. Artech House Publishers, 1998.
- [91] J. Raskin. *The Humane Interface: New Directions for Designing Interactive Systems*. Addison-Wesley, Boston, 2000.
- [92] J. Robbins. *Debugging Applications for Microsoft .NET and Microsoft Windows*. Microsoft Press, 2003.
- [93] J. Rumbaugh, I. Jacobson, and G. Booch. *The unified modeling language reference manual*. Addison-Wesley, Boston, 1999.
- [94] R. Sayle and J. Milner-White. RasMol: Biomolecular graphics for all. *Trends Biochem. Sci.*, 20:374, 1995.
- [95] D. Shreiner. *OpenGL Reference Manual. The Official Reference Document to OpenGL*. Addison-Wesley, Boston, 2004.
- [96] D. Shreiner, M. Woo, and J. Neider. *OpenGL Programming Guide. The Official Guide to Learning OpenGL*. Addison-Wesley, Boston, 2005.
- [97] D. Stalling, M. Zockler, and H.-C. Hege. Fast display of illuminated field lines. *IEEE Trans. Vis. Comput. Graph.*, 3:118–128, 1997.
- [98] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Boston, 2000.
- [99] L. Stryer. *Biochemie*. Spektrum Akademischer Verlag, Heidelberg, 1991.
- [100] C. Venter and D. Cohen. The century of biology. *New Perspectives Quarterly*, 21:73–77, 2004.
- [101] M. Verdonk, J. Cole, M. Hartshorn, C. Murray, and R. Taylor. Improved Protein-Ligand Docking Using GOLD. *Proteins*, 52:609–623, 2003.
- [102] I. Wald, C. Benthin, and P. Slusallek. Distributed interactive ray tracing of dynamic scenes. *Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, 2003.
- [103] J. D. Watson. *The Double Helix: A Personal Account of the Discovery of the Structure of DNA*. Atheneum, New York, 1968.

- [104] D. Weininger. SMILES, a chemical language and information system. 1. Introduction to methodology and encoding rules. *J. Chem. Inf. Comput. Sci.*, 28, 31 - 36., 1988.

List of Figures

1.1. Growth of the protein database	3
1.2. Overview of molecular software tools	5
2.1. Design of the VIEW library	13
2.2. UML diagram of BALLView's core architecture	17
2.3. UML diagram for the DatasetControl	24
2.4. Qt Designer for developing GUIs	26
2.5. PreferencesDialog	27
2.6. UML diagram for the Representation class	31
2.7. UML diagram for a multithreaded MD simulation	39
3.1. BALLView's graphical user interface	48
3.2. Example for the class documentation	53
3.3. Example for the BALLView documentation	54
3.4. Visualization of HIV protease	55
3.5. The DisplayProperties dialog	56
3.6. Example for a <i>Cartoon</i> model	58
3.7. Example for the usage of labels	59
3.8. Example for an electrostatic potential visualization	60
3.9. Dialog for coloring meshes by scalar grid data	61
3.10. Example for the Volume Rendering	62
3.11. Visualization of field lines	63
3.12. Example for a SE surface in wireframe mode	65
3.13. Comparison of solid and transparent surfaces	66
3.14. Toon shader model for a topoisomerase	67
3.15. Example for the usage of clipping planes	68
3.16. Example for the usage of capping planes	69
3.17. Example for BALLView's POVRay export	70
3.18. Dialog for visualizing trajectories	72
3.19. Atom overview dialog	76
3.20. Configuration dialog for the AMBER force field	78
3.21. Kekulization of Pyrrole	83

3.22. Kekulization of a complex structure	87
3.23. Dialog to create peptides	91
3.24. Dialog for starting a docking run	93
3.25. Docking progress dialog	94
3.26. Docking result dialog	95
3.27. Configuration dialog for the FDPB solver	96
3.28. GUI interface to Python	97
3.29. Python script editor	98
3.30. A user built representation	101
3.31. Dialog for assigning hotkeys	102