VMD Programmer's Guide

J. Gullingsrud, D. Norris, J. Stone

Version 1.6.1

April 20, 2001

Theoretical Biophysics Group[1]
University of Illinois and Beckman Institute
405 N. Mathews
Urbana, IL 61801
VMD WWW home page: `http://www.ks.uiuc.edu/Research/vmd/`

Description

The Programmer's Guide describes the structure and organization of the program VMD, and provides complete descriptions for all important object classes. This guide is designed to aid researchers interested in learning how VMD is constructed and who are interested in making modifications or improvements to the existing code. The program is logically divided into six main categories: (1) Utility objects; (2) Rendering (display) objects; (3) Molecule objects; (4) Remote connection objects; (5) User interface objects; (6) Spatial tracking and 3D pointer object. The Programmer's Guide describes first the organization and conventions used in all source files for VMD, then discusses the program design and relationship between the five components, and concludes with individual descriptions of each object class.

[1] http://www.ks.uiuc.edu/

Contents

List of Figures

List of Tables

Introduction

VMD is a molecular graphics and visualization program designed to be used for interactive display of molecular systems, particularly biopolymers such as proteins, nucleic acids, and biological assemblies such as membrane lipid bilayers. VMD has several goals:

- **General molecular visualization**
  This program is at heart a general application for graphical display of molecules, similar in basic capabilities to commercial program such as Quanta and non-commercial programs such as XMol, Ribbons, and others. The goal of VMD is not to become a complete replacement for these programs; instead, VMD focuses on accomplishing the specific goals outlined here.

- **Visualization of dynamic molecular data**
  VMD can display in a variety of ways dynamically varying quantities for a molecule, such as position, velocity, and energy. Each molecule displayed by VMD consists of an *animation list*, which is comprised of individual frames of the molecule's trajectory as computed by some means (molecular dynamics, energy minimization, etc.). This animation list can be edited, played back, or saved to a file in a variety of formats. This dynamical data may be obtained from previous computation, or may be obtained directly from a concurrently running simulation.

- **Display and control of molecular dynamics simulations**
  VMD contains the ability to act as a graphical front end for a molecular dynamics (MD) program running on a remote supercomputer or high-performance workstation. This allows the user of VMD to set up, initiate, and interactively display and control the MD simulation as it is running. The user can disconnect from the simulation and let it continue, or interactively kill the simulation program and start another. Any number of simulations may be simultaneously displayed and controlled by a single VMD session.

- **Support for several input and display (output) devices**
  Beyond the standard workstation monitor and keyboard + mouse display and input devices available for graphical workstation users, a number of different visual display and control systems are available. For example, spatial tracking devices are available which may be used as a three-dimensional pointer, and stereo image projection devices are possible which may be used to display a three-dimensional image of a molecular system to a number of viewers. VMD supports the use of the two example devices described here, and aims to make support for other such devices relatively simple.

- **Provide an easily modifiable and extendible program**
  VMD is written in C++ and employs an object-oriented design to make the program easy to modify and extend. This Programmer's Guide is specifically designed to aid those people who are interested in changing the current implementation of VMD, or who are interested in adding new features to the program.

This manual, along with the Users Guide and Installation Guide, document the use of VMD.

A major intention of this manual is that it be kept up-to-date with all changes to VMD. As features are changed, these changes should be reflected in the description here, and as new capabilities are added these new features should be added to the relevant section in this document. In particular, when new object classes are added or when existing ones are changed or augmented, the description of the relevant class should be updated in the 'Program Structure' and 'Class Descriptions' chapters. In this way, other uses can also benefit from new additions. This is a major goal for VMD, to become a useful, common resource for all users in the field of computational structural biology.

1 Guide organization

As mentioned, this guide is designed to aid users who are interested in modifying or extending the current implementation of VMD. This manual is organized into the following chapters following this introduction:

- **Source code file structure (section 8)**
  What subdirectories exist within the VMD working directory tree, and how to use the RCS (Revision Control System) to access and modify source files.

- **Using the configure script and makefile (section 10)**
  The design and use of the *configure* script, which creates the proper Makefile needed to compile VMD for a particular architecture and set of options, and how to use the Makefile once it has been created.

- **Source code style guide (section 16.3)**
  A set of guidelines used (mostly) in the naming of C++ classes, functions and variables, and a set of steps to follow when adding new objects and source code files to VMD.
- **Program structure (section 18.1)**
  A comprehensive overview of the design of VMD, concentrating on the organization and relationship of the various C++ objects which make up the program.
- **Function and class descriptions (section 21)**
  With section 18.1, the major part of the Programmer's Guide. This section provides a standard description for each utility function and object class in C++, outlining how to use each class and, for the main base classes in VMD, how to define and create new derived classes.

2 Document style conventions

1. **Object Classes**
   All object class names are indicated with a **typewriter** font, and begin with a capital letter, i.e.,

   CommandQueue

2. **Function Names**
   All function names begin with a lower case letter and are indicated with a `typewriter` font, with the return type in **bold** font; arguments are specified with the type in **bold** font, and the name in *italics*, i.e.,

   char * `stringdup`(**char *** *s*)

   When described specifically as a member function of a particular class, the class name is prepended, i.e.,

   void Displayable::`pick`(void *)

   Unless otherwise specified, functions with no arguments given are assumed to have no arguments, and specify **void** in their function prototype.

3 For more information on BioCoRE, MDTools, NAMD, and VMD

VMD is part of a suite of tools developed by the Theoretical Biophysics group at the University of Illinois.

- BioCoRE
  BioCoRE is a web-based collaborative environment for structural biology which provides tools to allow collaboration between researchers down the hall or around the world. Anyone with access to the internet and a standard web browser can join BioCoRE and create or be added

to research projects, and information about a particular project is shared among all members of that project. BioCoRE is accessed through a Java applet which includes instant messaging and notification of project updates.

BioCoRE currently provides tools in three toolboxes. The Workbench toolbox provides access to structural biology applications such as simulation and visualization programs. The Notebook toolbox supplies a message board where users store project notes. The Conferences toolbox allows interactive text-based chats, which are also archived to the message board for future reference. Soon, the Documents toolbox will provide tools to assist in joint preparation of publications.

BioCoRE is currently accessible to users through the BioCoRE server at UIUC, and a code release is planned which will allow sites to run their own servers. Because BioCoRE is web-based, updates and new features are automatically available every time the user logs in. More information is available at the BioCoRE home page[2]

- MDTools
  JMV is an easy-to-use Java-based molecular viewer which uses Java3D to provide hardware accelerated 3D rendering of molecules. JMV is designed as component software, and can be used within other programs, or as a standalone application. JMV is implemented as a Java Bean, and can be used in the standard Bean-capable development tools. JMV can be used as an applet to provide in-browser molecular visualization capabilities, in situations where VMD is overkill for the task. More information is available at the JMV home page[3]

- MDTools
  MDTools is a collection of programs, scripts, and utilities provided for researchers to make various modeling and simulation tasks easier, and to provide basic code and utilities which can be built up into larger toolsets. Each tool, script, or library has its own separate distribution, documentation, etc. The programs in this collection are provided for the benefit of the research community, but are not officially supported unless otherwise stated. Each program may have a software license and/or citation suggestions which should be listed on its web page. More information is available at the MDTools home page[4]

- NAMD
  A parallel, object-oriented molecular dynamics code designed for high-performance simulation of large biomolecular systems. NAMD uses the CHARMM force field and file formats compatible with both CHARMM and X-PLOR. NAMD supports both periodic and non-periodic boundaries with efficient full electrostatics, multiple timestepping, constant pressure and temperature ensemble simulation methods. NAMD provides several methods of steering a simulation through the application of additional forces, including the ability to connect directly to VMD for interactive steering of a live simulation. NAMD is distributed free of charge and includes source code. More information is available at the NAMD home page[5]

- Structural Biology Software Database
  The Structural Biology Software Database contains programs which are thought to be of

[2] http://www.ks.uiuc.edu/Research/biocore
[3] http://www.ks.uiuc.edu/Development/jmv
[4] http://www.ks.uiuc.edu/Development/MDTools
[5] http://www.ks.uiuc.edu/Researach/namd

interest to researchers in the fields of structural biology, quantum chemistry, and bioinformatics. The database entries are contributed by users, and are moderated to provide a high quality resource. More information is available at the Structural Biology Software Database home page[6]

- VMD
  A general molecular visualization program capable of interactive display and concurrent control of a molecular dynamics simulation running on a remote computer. VMD uses an object-oriented design, and is written in C++. This document describes VMD. More information is available at the VMD home page[7]

For more on any of the individual software efforts, BioCoRE, MDTools, NAMD, or VMD, see the Theoretical Biophysics Group WWW home page[8].

4 Contacting the authors

The current authors of VMD are Justin Gullingsrud, David Norris, and John Stone. We are very interested in and grateful for any user comments and reports of program bugs or inaccuracies. If you have any suggestions, bug reports, or general comments about VMD, please send them to us at `vmd@ks.uiuc.edu`.

5 Credits and Program Reference

The authors request that any published work or images created using VMD include the following reference:

Humphrey, W., Dalke, A. and Schulten, K., "VMD - Visual Molecular Dynamics" *J. Molec. Graphics* **1996**, *14.1*, 33-38.

VMD has been developed by the Theoretical Biophysics group at the University of Illinois and the Beckman Institute. The main authors of VMD are A. Dalke, J. Gullingsrud, W. Humphrey, S. Izrailev, D. Norris, J. Stone, J. Ulrich. This work is supported by grants from the National Institutes of Health (grant number PHS 5 P41 RR05969-04), the National Science Foundation (grant number BIR-9423827 EQ), and the Roy J. Carver Charitable Trust.

6 Copyright and Disclaimer

[6] http://www.ks.uiuc.edu/Development/biosoftdb/
[7] http://www.ks.uiuc.edu/Researach/vmd
[8] http://www.ks.uiuc.edu/

you do not have this file in your distribution, it can be downloaded from:
`http://www.ks.uiuc.edu/Research/vmd/current/LICENSE.html`
Some of the code and executables used by VMD have their own usage restrictions:

- STRIDE
  STRIDE, the program used for secondary structure calculation, is free to both academic and commercial sites provided that STRIDE will not be a part of a package sold for money. The use of STRIDE in commercial packages is not allowed without a prior written commercial license agreement. See `http://www.embl-heidelberg.de/argos/stride/stride_info.html`

- SURF
  The source code for SURF is copyrighted by the original author, Amitabh Varshney, and the University of North Carolina at Chapel Hill. Permission to use, copy, modify, and distribute this software and its documentation for educational, research, and non-profit purposes is hereby granted, provided this notice, all the source files, and the name(s) of the original author(s) appear in all such copies.
  BECAUSE THE CODE IS PROVIDED FREE OF CHARGE, IT IS PROVIDED "AS IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED.
  This software was developed and is made available for public use with the support of the National Institutes of Health, National Center for Research Resources under grant RR02170.
  See `ftp://ftp.cs.unc.edu/pub/projects/GRIP/SURF/surf.tar.Z`

- url_get
  The Perl script url_get, was written by Jack Lund at the University of Texas as Austin. There appear to be no restrictions on its use.

- Python
  Python is made available subject to the terms and conditions in CNRI's License Agreement. This Agreement together with Python may be located on the Internet using the following unique, persistent identifier (known as a handle): 1895.22/1012. This Agreement may also be obtained from a proxy server on the Internet using the following URL:
  `http://hdl.handle.net/1895.22/1012`

- PCRE
  The Perl Compatible Regular Expressions (PCRE) library used in VMD was written by Philip Hazel and is Copyright (c) 1997-1999 University of Cambridge.
  Permission is granted to anyone to use this software for any purpose on any computer system, and to redistribute it freely, subject to the following restrictions:
  1. This software is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
  2. The origin of this software must not be misrepresented, either by explicit claim or by omission.
  3. Altered versions must be plainly marked as such, and must not be misrepresented as being the original software.
  4. If PCRE is embedded in any software that is released under the GNU General Purpose Licence (GPL), then the terms of that licence shall supersede any condition above with which it is incompatible.

- Tachyon
  The Tachyon ray tracing system distributed with VMD was written by John E. Stone and is Copyright (c) 1994-2001 by John E. Stone. Please see the current distribution of Tachyon for redistribution and or licensing information outside of VMD use. Permission is granted to use Tachyon freely along with VMD.

7 Registering VMD

VMD is made available free of charge for all interested end-users of the software (but please see the Copyright and Disclaimer notices). Redistribution of the software to third parties may require a special license, please check the current VMD license agreement for details. We would like to request that you register with us that you are using VMD. This is so that we can maintain some idea of the number of users of the program and so that we know who to contact about program updates, bug fixes, etc. Registration is now part of our software download procedure, so once you've filled out the forms on the VMD download area you are finished.

8 Acknowledgments

The authors would particularly like to thank those individuals who have contributed improvements to VMD in the form of new features or entire replacement codes for old features. Special thanks go to Andrew Dalke, Paul Grayson, and Charles Schwieters for their VMD contributions. The entire VMD user community now benefits from your contributions.

The authors would like to thank the members of the Theoretical Biophysics group, past and present, who have helped tremendously in making suggestions, pushing for new features, and trying out bug-ridden code. Thanks go to Alexander Balaeff, Daniel Barsky, Tom Bishop, Ivo Hofacker, Xiche Hu, Barry Israeliwitz, Dorina Kosztin, Ilya Logunov, Jim Phillips, Ari Shinozaki, Svilen Tzonev, Willy Wriggers, Dong Xu, Feng Zhou. Thanks also to all of you who have tried out the program.

Many external libraries and packages are used in VMD, and the program would not be possible without them. The authors wish to thank Jon Leech for the code to compute the uniform point distributions; John Ousterhout and the other authors of the Tcl and Tk packages; the authors of the VRPN library from the University of North Carolina; Amitabh Varshney, author of SURF, also from UNC; Dmitrij Frishman at EMBL for developing STRIDE; Jack Lund for the url_get perl script; and Ethan Merrit from the University of Washington for developing the algorithm for drawing ribbons.

We also received invaluable assistance from people who got the source code and sent in patches and explicit bug reports. The VMD developers would like to thank Axel Berg, Andrew Dalke, Rick Kufrin, Joe Landman, Clare Macrae, Lukasz Salwinski, Stephen Searle, Charles Schwieters, Michael Tiemann, Raymond de Vries, and Simon Warfield for their bug fixes and correspondence.

Source Code File Structure

Within the main VMD directory there are several subdirectories, as well as a configuration script (section 10) used to configure VMD to compile for a particular architecture and set of options.

9 Directory tree structure

When a directory with the VMD source, library, documentation, and other data files is established, the following subdirectories are present:

`bin` ... Utility scripts and programs, such as the script used to start the proper architecture-specific VMD executable, and the programs used on SGI workstations to change the current stereo mode.

`data` ... Data files used when VMD starts up, i.e., `.vmdrc` and other 'dot' files.

`doc` ... Latex documentation files.

`lib` ... Special libraries and associated header files used by VMD.

`proteins` ... Example protein data files used for testing.

`src` ... Source code files (.c, .C, and .h files).

10 Using the RCS system

NOTE: This section is mainly for use by the main VMD developers, which use RCS to maintain the source code, documentation, and other files. The standard VMD ftp distribution does not include the RCS directory, and the discussions in the manual about RCS do not apply to that case. If you are NOT using RCS, you may simply skip this section and other pertaining to the use RCS when working on VMD.

The Revision Control System (RCS) is used here to allow several people to simultaneously work on modifications to the files used in compiling and running VMD. To do this, each file (i.e., source code files, utility scripts, data or documentation files) is stored in a special RCS format within the directory *RCS* in the main VMD working directory. Within each subdirectory, a link to this main RCS directory is made, so that there is one single official RCS dir, and several links to it. To obtain a copy of a file, a user *checks out* the file from RCS. If the file is to be changed, the user must also *lock* the file when checking it out. When the modifications are complete and verified to be successful, the same user checks the file back in, releasing the lock and depositing the changed version back in the main RCS directory. No other user can lock a file locked by another user until the lock has been released, thus guaranteeing that no two users are attempting to make conflicting changes to the same file simultaneously. Each time a file is modified and checked back in, the current *version* number of that file is increased; through RCS commands, older versions of a file may be retrieved if new changes end up not working out or are not needed. Several utility programs comprise the RCS system; this section describes the basic use of these programs to perform the tasks mentioned above.

- **To add new files to the RCS:**
  The `rcs` program is used to initialize new files, and the `ci` program checks in a new version. The commands to use are:

  ```
  rcs -i -a<username>[,<username>[...]]  -L <new files>
  ci -u <new files>
  ```

  Each file in the RCS should have a special *header* which includes special commands to allow RCS to indicate the name, version, and other data in each file when it is checked out; see section 16.3 for notes on what RCS headers to use for each type of file (Latex, shell script, C/C++ source) and where to find standard RCS header examples.

- **To check files out of the RCS:**
  The `co` program retrieves files from the RCS directory, and puts the retrieved copy in the current directory. There are two ways to use this, the first to simply retrieve a copy of the file for read-only access, and the second to check out and *lock* a file, assuring you are the only one allowed to modify the file. The two commands are, respectively,

```
co <list of files>
co -l <list of files>
```

For locked files, when modifications are complete the file should be checked back in, as described next. Files checked out, but not locked, do not need to be checked in.

- **To check a file back into the RCS:**
  The `ci` program updates the official RCS directory with a new version of a file previously checked out and locked by a user. When the file is checked in, its current version number is incremented, and the user may enter a quick note about what changes were made. It is possible to have these revision notes placed within the file itself as comments, and this is done by VMD through proper use of RCS headers. The command to check in a file is:

  ```
  ci -u <list of checked out files>
  ```

  The `-u` flag then does an automatic `co <files>` command, which assures there is a (non-locked) version of the file still in your directory after checking in a file.

- **To unlock a file that has been locked, without checking it in:**
  Occasionally changes are made to a file that has been locked out, but the changes are not needed or just plain don't work. To simply release a lock, which abandons the changes and in fact will completely erase them unless a copy is made by hand beforehand, use the commands:

  ```
  rcs -u <file>
  chmod -w <file>
  co <file>
  ```

  The first command releases the lock; the others reset the file and check out a copy of the old version (as it existed when initially checked out and locked).

Information on these RCS programs should be available via man pages as well.

Using the Configure Script and Makefile

11 The configure script

VMD is designed to be relatively easy to compile for different operating system versions (i.e., IRIX or HP-UX), and to allow a person to individually include or exclude different optional capabilities such as support for using the Tcl script interpreter library or support for remote simulation control and display. To make this possible, a *configure* shell script has been written which will create a Makefile properly configured for the selected architecture and options.

The `configure` Bourne shell script is found at the top level of the VMD working directory. This script has three purposes:

1. To adapt the Makefile to use the proper libraries and compiler options for the type of operating system being used.

2. To specify which ones of the many optional components of VMD should be included in the final executable, and to include in the Makefile the necessary files for the selected options.

3. To configure the VMD directories and source files to use the values for configurable parameters chosen by the user, i.e., the name of the final executable, the location for where the program is to be installed, etc.

The following steps must be followed to use the `configure` script to set up the VMD directories in order to compile VMD:

1. The file `configure.parameters` must be edited to set the values for certain *configuration parameters* which describe such things as where to install VMD, what names to use for data files, etc. This is described in section 12.
2. After the parameters have been entered, the `configure` script must be run to create the Makefile necessary to compile VMD. This is described in section 13.
3. Once the Makefile has been created, the command `make depend; make` must be run after changing to the `src` subdirectory of the VMD working directory. This will regenerate the source code file dependencies, and then compile the program.
4. After successful compilation, while still in the `src` directory, the command `make install` will install the program and necessary data files. This will copy the necessary executables, scripts, and data files to the proper directories as specified by the values of the configuration parameters.

12 Setting the configuration parameters

The first step in using the `configure` script is to set the values of configurable parameters. The `configure` script first assigns default values to these parameters, and then checks to see if the file `configure.parameters` exists. If it does, the script will read this file and use the values for parameters listed there. Edit the file `configure.parameters` and change the settings of the variables to the required values. The file is interpreted as a Bourne shell script, so blank lines and comments beginning with the `#` character are permissable. All parameters are of the form

```
<keyword>=<value>
```

with **no** space characters on either side of the equal sign. This section lists the keywords, purpose, and possible values for each parameter. If a parameter is not listed at all in the file `configure.parameters`, its default value will be used.

12.1 Installation parameters

The following parameters control where and how VMD is installed **after** compilation. They do not have any effect on how VMD is compiled. It is easiest to change these at the same time other parameters are being changed, so the `configure` script need only be run once. Edit the file `configure.parameters` in the main working directory to change these settings.

- `INSTALLBINDIR (default=/usr/local/bin)`
  The directory where scripts and utility programs are installed. This should be a directory which is in the path of all users interested in running VMD.

- `INSTALLLIBDIR (default=/usr/local/lib/vmd)`
  A directory in which VMD data files and architecture-specific executables are placed. This directory should *not* be in the path of users running VMD.

- `INSTALLNAME (default=vmd)`
  VMD uses a shell script to run the proper VMD executable for the given version of Unix (the script also sets some environment variables and starts an `xterm` running to act as the VMD console). `INSTALLNAME` is the name given to this shell script, and will be the command users type to run VMD. When the program is installed, the file `bin/vmd` is then copied to the file `INSTALLBINDIR/INSTALLNAME`.

12.2 Compilation parameters

These parameters determine various aspects of how VMD is compiled, for example the names of data files and maximum array sizes. Edit the file `configure.parameters` in the main working directory to change these settings.

- `TCL_INCLUDE_DIR (default=/usr/local/include)`
  The Tcl library is used by VMD to interpret and execute text scripts; there is also the option to use the Tk library for the graphical user interface. This parameter determines where to look for Tcl and Tk header files. If Tcl is not installed on your system, this option is ignored. ALso, if Tcl is not installed, do not request the TCL or TK options when configuring VMD.

- `TCL_LIBRARY_DIR (default=/usr/local/lib)`
  This parameter determines where to look for Tcl and Tk library files, just as `TCL_INCLUDE_DIR` determines where to look for header files. If Tcl is not installed on your system, this option is ignored.

- `DEFBABELBIN (default=INSTALLBINDIR/babel)`
  VMD uses the program `babel` to convert different molecular data files to PDB files, in order to allow it to understand a large number of file formats. `DEFBABELBIN` determines the location of the `babel` executable which VMD should use. It should include the complete path and name of the `babel` program. If VMD cannot find a `babel` executable, only PSF, PDB, binary DCD files and Gromacs files will be understood by the program. If `babel` is not installed on your system, this option is ignored. The environment variable `VMDBABELBIN` can also be used to override this value when VMD is run.

- `DEFDISPLAY (default=WIN)`
  The default display device to use when VMD starts up, if not set by the initialization file or a command-line option. This can be `WIN`, `CAVE`, or `TEXT`.

- `DEFDIST (default=-2.0)`
  The default value for the distance, in 'world' coordinates, from the origin to the display screen. If this is zero, the origin of the coordinate system in which molecules are drawn coincides with the center of the display. If it is < 0, the origin is located between the viewer and the screen, while if it is > 0, the screen is located closer to the viewer than the origin. A value < 0 puts any stereo image in front of the screen, aiding the three-dimensional effect; a value > 0 results in a stereo image that is behind the screen, a less dramatic (but easier to see, for some people) stereo effect when stereo display is in effect.

- `DEFHEIGHT (default=6.0)`
  This parameter, with `DEFDIST`, defines the size and distance of the display screen. `DEFHEIGHT` is the default value for the screen height, which is the vertical size of the display screen in 'world' coordinates. Each molecule is initially scaled and translated to fit within a 2 x 2 x 2 box centered at the origin; so the height of the screen helps determine how large the molecule appears initially. The default value is 6, and with the default value of `DEFDIST` this allows the molecule to fill up most of the screen at the start. If VMD is being displayed on a workstation monitor only, it is best not to change this value much. This parameter is used mainly to configure the VMD display to the dimensions and position of a large-screen display, such as a projector, that may be being used as a stereo display. See the "Customizing VMD" section of the Installation Guide for more discussion about the `DEFDIST` and `DEFHEIGHT` parameters.

- `DEFHTMLVIEWER (default=Mosaic)`
  Online help information is provided by VMD by displaying a help file in HTML format, using an external HTML viewer. This parameter sets the default name of the program to use to view HTML files. The environment variable `VMDHTMLVIEWER` can also be used to override this value when VMD is run.

- `DEFTITLE (default=ON)`
  The default setting for the flag which indicates whether to display a title screen when VMD starts up. This can be `ON` or `OFF`.

- `DEFTMPDIR (default=/tmp)`
  The directory which VMD should use to store temporary files.

- `INITFILENAME (default=.vmd_init)`
  The name of the VMD initialization file.

- `MAXSTRINGLEN (default=6)`
  The maximum number of characters which are considered when words in a text command are processed. It determines how many characters maximum distinguish different words. It is suggested to not change this value, which can be any positive integer.

- `PROGVERSION`
  The current version number of the program.

- `PROMPTSTRING (default="vmd >")`
  The string displayed as a prompt for text command input. This can be any string, and should be enclosed in double quotes.

- `STARTUPFILENAME (default=.vmdrc)`
  The name of the VMD startup command script.

13 Configuring the Makefile

Once all parameters have been properly specified, the `configure` script must be run to create a Makefile for compiling and installing VMD. This Makefile is copied into the `src` and `doc` directories. The syntax for running the configure script is:

```
configure [<OSTYPE> [OPT1 [OPT2 ...]]]
```

where `<OSTYPE>` is a keyword specifying one of the operating systems mentioned below, and `OPT1, OPT2, ...` are keywords specifying which optional VMD components should be included when the program is compiled and linked. The total possible optional keywords are explained below. When multiple options are specified and the list contains contradictory options, the last value specified in the command is used. Each time `configure` is run, a copy of the list of parameters used is written to the file `configure.options`; if simply the command

```
configure
```

is run, the values contained in `configure.options` will be used as the arguments. In this way, when a "default" configuration has been established previously, and a new Makefile must be generated for some reason, the user need only go to the main VMD working directory and type `configure`. The file `configure.options` may be edited by hand to set the list of options to the required values, after which the user may simply type `configure`.

13.1 Optional program components

Beyond the core capabilities of VMD, several additional features may be selectively included or excluded from the program by specifying the proper keywords to the configure script and then recompiling. As well, for some core features of VMD, a single value from a list of several possible options must be selected. This helps reduce the size of the resulting executable for users who have no need for certain options, and allows VMD to be compiled on architectures for which certain options are not available (due, for example, to a missing library or a hardware limitation). The following arguments to the configure script select these optional components; where multiple names are given in a list separated by 'or' symbols (`|`), only ONE of the items in that list should be given.

- `OPENGL | MESA | NOGRAPHICS (default=OPENGL)`
  This option determines which 3D graphics rendering library to use. `OPENGL` indicates to use the OpenGL library on all workstations with support for hardware accelerated OpenGL rendering. Finally, if for some reason no graphics capabilities are desired in the program at all (perhaps if VMD is to be compiled to act only as a filter between PDB files and image files), `NOGRAPHICS` indicates to not include any graphics display capabilities at all.

- `FLTK | XFORMS | NOGUI (default=FORMS)`
  This option determines which GUI library should be used by VMD, if any. VMD uses several separate user interface components, including the text console interface and the graphical user interface (GUI). `FORMS` indicates to use the Forms library of Mark Overmars; If no GUI is possible or desired, specifying `NOGUI` will not include any GUI library at all in VMD, and all VMD actions will have to be performed via text commands.

- `TCL | NOTCL (default=NOTCL)`
  Tcl is a script parser and interpreter language which provides the capability to have interpreted scripts including control loops, variable substitution, function and procedure calls, and many other features. If the Tcl library is installed already on your system, VMD can use this library to parse and interpret text command scripts. Specifying the `TCL` option configures VMD to compile with this feature. If the Tcl library is not available on your system, `NOTCL` should be specified, which indicates for VMD to parse all text commands itself.

- `REMOTE | NOREMOTE (default=NOREMOTE)`
  This option specifies to include (exclude) the capability to display and control molecular dynamics simulations running on remote supercomputers or workstations. This is a key feature for VMD, and the inclusion of this option is strongly encouraged. However, the option to exclude it from the final executable is available for users who have no need or interest in directly visualizing molecular dynamics simulations, or who wish to use VMD for some other preferred visualization feature.

- `VRPN | NOVRPN (default=NOVRPN)`
  Indicates to have VMD use (or not use) the VRPN Tracker library developed at the University of North Carolina, Chapel Hill to access external 3D pointer devices. If `VRPN` is specified, the VRPN Tracker library will be linked with VMD, and the program will be able to use external spatial tracking devices such as a Polhemus Fastrak as a 3D pointer. Combined with some form of 3D display equipment such as a stereo projector, this provides a complete 3D environment for one or possibly several researchers. However, such equipment is not available for all users, and the Tracker library is not available on all OS types, so support for using the VRPN library may be omitted.

- `CAVE | NOCAVE (default=NOCAVE)`
  The CAVE is a 3D display environment developed by the Electronic Visualization Laboratory at the University of Illinois at Chicago. VMD supports the use of the CAVE as a display device; VMD also supports the use of a CAVE-controlled spatial tracker as an input/control device when the CAVE is used for graphical display (similar, but distinct from, the use of the VRPN Tracker library for working with other spatial tracking equipment). If the `CAVE` option is specified, VMD will include the capability to display molecules in a CAVE, and will link with the CAVE library which must be available in the `lib` directory for the target operating system. Some notes about the CAVE option:

  – The CAVE option is currently only available for SGI workstations, and uses the OpenGL library. This option implies the `OpenGL` option.

  – Even if the CAVE option is included, VMD can still function normally using the workstation monitor as the display device. A command-line switch is then used to specify which display device to use.

  – Of course, this option is only useful for users with access to a CAVE system, of which there are admittedly very few. Please contact the VMD authors for information on who to contact about questions on CAVE access and design.

- `EXTERNAL | NOEXTERNAL (default=NOEXTERNAL)`
  An experimental capability of VMD is to use an external program for generation of text commands, which are then sent via PVM routines to VMD for execution. The `EXTERNAL` option enables this capability, and requires that PVM 3.3 or more recent be installed on your system. It is suggested not to use this option unless you know what you're doing.

- `DEBUG | NODEBUG (default=NODEBUG)`
  VMD optionally may contain specific code to print debugging messages to the console during execution, for testing and development. If this code is included (by specifying the `DEBUG` option), user commands are available to turn on or off the printing of these messages, and to set the level of detail for the debugging output. However, VMD runs excruciatingly slow when

this debugging code is included, so it is recommended to exclude it unless a specific problem is encountered that requires debugging output. By default, the `DEBUG` option is turned off, and the debugging code in VMD is commented out when the program is compiled. This option is mutually exclusive with the `OPTIMIZE` option described next. Including the `DEBUG` option also indicates to use the `-g` flag to the compiler to include necessary symbolic debugger information.

- `OPTIMIZE | NOOPTIMIZE (default=NOOPTIMIZE)`
  If `OPTIMIZE` is specified, all files are compiled with the `-O` optimization flag. If it is not specified, all files will be compiled with the `-g` flag, which includes symbolic debugger data in the executable. `OPTIMIZE` is mutually exclusive with the `DEBUG` option.

- `MAKE | NOMAKE (default=NOMAKE)`
  After configuration is complete, and a new Makefile has been generated and copied to the proper directories, the configure script will change to the `src` directory and execute a "make" command if directed. If this is done, no files are checked out, only a recompilation is done. If `MAKE` is specified, this automatic execution of a make command is done; if `NOMAKE` is specified, the make command is not executed, instead a warning message is issued and the configuration script completes.

- `SILENT | NOSILENT (default=SILENT)`
  Determines whether make commands are echoed to the screen as they are executed (`NOSILENT`), or whether they are hidden (`SILENT`).

- `ALL`
  Same as `OPENGL FLTK TCL REMOTE CAVE VRPN OPTIMIZE`.

13.2 Supported operating systems

Currently, VMD may be compiled on the following operating systems (the corresponding value for `<OSTYPE>` for the configure script is given in parentheses):

- HP-UX 10.X (HPUX10), for version 10.20 or later.
- HP-UX 11.X (HPUX11), for version 11.0 or later.
- IRIX 5.X (IRIX5) or IRIX 6.X (use IRIX5 for both cases).
- IRIX 6.X (IRIX6)
- Linux (LINUX), RedHat 6.2 or later
- Solaris 2.x (SOLARIS2), for version 2.7 or later.
- Solaris 2.x (SOLARISX86), for version 2.7 or later.

Not all opiontal components are available for all architectures. Table 1 describes which optional components may be used for each supported architecture. A dash indicates that the option is not available, while the word 'yes' indicates that the option is allowed.

14 Compiling VMD

Finally, after the proper Makefile has been generated, VMD may be compiled. In the main working directory, after running `configure`, enter the following commands:

```
cd src
make clean
make depend
make
```

The first make command removes all old object files from the `src` directory, and the second make command regenerates the source code dependencies (which are put in the file `Makedata.depend`). It is necessary to do these two commands if the configuration has been changed in any way (e.g., some parameters have been changed, or a different set of optional features are being included). Otherwise, if the most recent change since compiling is simply that a source code file has been edited, the first two make commands are not necessary. Finally, `make` will compile and link the VMD source code files.

Note for RCS users: If the RCS system is being employed, use the command `make co` instead of `make depend` in the list above. This will check out up-to-date versions of all the files necessary for the optional features requested. See the Programmers Guide for more information on using RCS.

Option	IRIX4	IRIX5	HPUX9
OPENGL	yes	yes	yes[1]
FORMS	-	yes	yes[1]
FLTK	-	yes	yes[1]
TK	yes[2]	yes[2]	yes[2]
TCL	yes[3]	yes[3]	yes[3]
REMOTE	-	yes	yes
VRPN	yes	yes	-
CAVE	-	yes	-
EXTERNAL	yes[4]	yes[4]	yes[4]

[3]This option requires use of the Tcl library, which must be installed on the system. The Tcl/Tk libraries are not included with the VMD distribution.
[4]This option requires used of the PVM 3.3 (or more recent) library, which must be installed on the system. The PVM libraries are not included with the VMD distribution.

Table 1: Available options for architectures supported by VMD.

15 Makefile commands

Once the Makefile has been created by running the configure script, several useful Makefile commands are available. A copy of the same Makefile is placed in the `doc` and `src` directories, however, the following Makefile commands are useful only in the proper directories. The commands are categorized by where they may be issued, `src` directory, `doc` directory, or either directory.

15.1 `src` directory Makefile commands

The following Makefile commands may be issed while the `src` directory is the current directory:

- `make [`*blank* `| all | default]`
  Completely compiles and links VMD. Copies of the proper files must be checked out from the RCS directory, and the source code dependencies must be up to date (see `make co` and `make depend`).

- `make` *module*`.o`
  Compile a single source code file (*module*, with proper filetype extension).

- `make install`
  This will install the VMD executables, documentation files, and data files into a particular VMD library directory determined by setting configuration variables in the configure script (see section 12). As well, utility programs and scripts necessary to run VMD will be copied into a directory also specified in the configure script. This command should be run after the Makefile has been generated, and VMD has been successfully compiled with the requested options.

- `make distrib`
  This creates a tarred and compressed version of the VMD working directory structure, which is suitable for distribution to other users via FTP. The resulting `.tar.Z` file is placed in the directory `distrib` in the VMD working directory. The distribution version contains all files needed to compile and run VMD, as well as the documentation files. This command is not needed by general users, it is used by VMD developers to generate the files which are made available via anonymous FTP.

15.2 `doc` directory Makefile commands

The following Makefile commands may be issed while the `doc` directory is the current directory:

- `make doc | make doc.all`
  Will run Latex on the VMD documentation files, and produce all the guides for VMD. The following documents will be produced:

  1. `ig.dvi` ... the VMD Installation Guide.
  2. `pg.dvi` ... the VMD Programmers Guide.
  3. `ug.dvi` ... the VMD Users Guide.

- `make doc.[ig | pg | ug]`
  Will run Latex and create the specified guide document, i.e., the VMD Users Guide if `make doc.ug` is entered.

- `make doc.book`
  This will format all the VMD documentation guides into a single document in *book* format, with each individual Guide in a separate Part. The resulting file will be called `vmd.dvi`.

15.3 Makefile commands for either directory

The following Makefile commands may be issued from either the `doc` or `src` directory:

- `make co[.src | .h | .doc | .bin | .data | .extra]`
  Checks out sets of files from the RCS (but does not lock them). The particular files checked out depend on which optional components were specified when the configure script was run and the Makefile created. The first form, `make co`, checks out copies of *all* files, and then does a `make depend` command (see below). The other options check out the following subsets of files:

  `co.src` ... All `.C` and `.c` files, which are placed in the `src` directory.

  `co.h` ... All header (`.h`) files, which are placed in the `src` directory.

  `co.doc` ... All documentation files, which are placed in the `doc` directory. If just a single documentation guide is to be created, the files required just for the installation, programmers, or users guides may be checked out with the command `co.doc.ig`, `co.doc.pg`, or `co.doc.ug`, respectively.

  `co.bin` ... All utility programs and scripts, which are placed in the `bin` directory.

  `co.data` ... All data files, which are placed in the `data` directory.

  `co.extra` ... All other special files necessary to compile VMD, which are placed in the `src` directory.

- `make clean`
  Removes all object files, backup files, and other unnecessary files still remaining after compilation has finished.

- `make depend`
  The source code file dependencies are generated, when possible, using special compiler flags (on SGI workstations, for example, this is done with the `-M` flag to the CC compiler). The resulting dependencies are placed in the file `Makedata.depend` in the `src` directory, and are included by the Makefile. Dependencies are determined based on the current optional component configuration; if the configuration changes, or new files are introduced, or the list of included files for a source code file changes, the dependencies should be recomputed. This is done automatically when a `make co` command is issued. Every time the dependencies are updated, a `make version` command is executed as well.

- `make install | make install.exe`
  Installs the VMD executables and data files into the VMD library directory, and copies the startup scripts and utility programs used by VMD into a user-specified directory. These directories are specified by the configurable parameters `INSTALLLIBDIR` and `INSTALLBINDIR` in the file `configure.parameters`.

- `make install.doc`
  Copies the documentation guides into a directory `doc` within the VMD library directory, where program executables and data files were previously installed. This is not necessary for VMD to run, but may be useful to users.

- `make install.all`
  Same as typing `make install.exe install.doc`.

- `make locks`
  Prints out a list of all RCS-locked files; the username of the person who locked each file is indicated by the ownership of the file.

- `make version`
  Creates two files, which contain data used by the compiler and by Latex to determine the values of configurable parameters. These files are:

  1. `src/config.h` ... A C header file which contains the program name, and current date, as a single string, and the values for several configurable parameters as set in the file `configure.parameters`.
  2. `doc/vmd_version.tex` ... A Latex file which contains the program name, version, and current date as Latex macros.

16 Changing the configure script

The configure script is organized into four sections, which must be updated whenever new files or options are introduced into VMD or a change in the Makefile must be made. Also, the settings of VMD configuration variables must be changed to the proper settings for the user. For most users, the only section that will need to be changed is the very first section, where parameters are set. Developers of VMD will need to change all the sections of the configure script when code changes are made.

The first section of the configure script is present simply to set the values for specific configuration parameters, i.e., the directories to install VMD data files and utilities. See section 12 for a description of these parameters. New configuration parameters should be introduced and initialized in this section.

Following the parameters section, the configure script contains commands to parse the command-line options to the script, and initialize internal variables which store the requested OS type and list of required options. When new optional components for VMD are introduced, this section must include commands to look for the command-line option requesting the new component, and must initialize new variables to indicate the requested option.

The third section of the configure script, which is the largest part of the file, is the set of instructions used to write out the Makefile. This is accomplished by copying text embedded within the configure script to the newly forming Makefile, substituting the values of configuration parameters when necessary. Any changes to how the Makefile operates, or to specific make targets, should be made by altering this section of the configure script. At the start of the Makefile are placed general variable settings, based on the values of the configuration parameters set at the start of the configure script. Following this, the names of all files which comprise the different components of VMD are placed within the Makefile. For the different optional components, the corresponding sets of filenames are included or excluded from the Makefile based on whether the option was included or excluded. This part also includes the names of all documentation, data, etc. files. To conclude the Makefile, the definitions of all make targets are copied over.

At the end of the configure script is a section which finishes up after a new Makefile has been successfully created. This section copies the new Makefile to the proper subdirectories, saves the settings used to create this Makefile to the file `configure.options`, and then if requested executes a "make" command in the `src` directory. There is little need to change this part.

16.1 Adding new configurable parameters

Initialize new configurable parameters at the beginning of the configure script, in the relevant section (installation, or compilation). Later the settings for these parameters are written out to the Makefile; new parameters should also be written out to this Makefile. If the parameter is to be used as a macro within the source code, it should be written out to the file `config.h`, which is done by the `make version` command after configure has been updated and a new Makefile has been created.

16.2 Adding new configuration options

When a new option is added to the configure script, several things must be changed. It is best to search through the file for occurrances of a similar option and add references to the new option as well. Several things must be checked:

- The option should be mentioned in the usage message, and in the configure script header.
- The option should be parsed near the beginning of the script, and remembered if it has been selected.
- The option should be written out to the file `configure.options` at the end of the script if the run was successful.

16.3 Adding support for a new architecture

If a new OS type or machine architecture is to be supported, follow these steps:

1. Select a string name for it, upper-case if possible and specifying the OS version number, i.e., "IRIX5".
2. Add this string as a configuration option in the OSTYPE category in the configure script.
3. When the Makefile is created, a section of settings are written out for the particular OS type selected which specify the compiler name, compiler options, and names of utility programs needed to do things such as file copy, etc. A new section must be entered into the configure script for the new architecture, and this section should be written to the Makefile when the respective OS type is selected for configuration.
4. The startup script used to run VMD must check for this OS type in order to run the proper executable when VMD is launched. Edit the file `vmd` in the `bin` directory (this script is entered into the RCS system) to check for the OS type.

Source Code Style Guide

The names of variables, functions, and object classes in VMD follow certain conventions in their capitalization and syntax. Also, the files used for VMD code, documentation, and data are formatted according to specific guidelines. This section describes these style guidelines.

17 Naming conventions

17.1 Class names

Names of C++ classes should begin with a capital letter, and not contain any hyphens. Class names comprised of multiple words should have each word capitalized. Example:

DisplayDevice

Classes derived from a base class should generally prepend a descriptive word to the base class name, unless the new word begins with a number in which case the additional word should be added to the end of the base class name. Examples:

CaveDisplayDevice

and

Displayable3D

17.2 Class member variable names

Names of variables which are members of C++ classes follow the same formatting rules as names of classes, except variable names should begin with a *lower* case letter. Quite often, if the variable is an instance of a particular class, the name of the variable is identical to the class name but with a lower case initial letter.

17.3 Class member function names

Names of functions which are members of C++ classes should be in all lower case letters, and use hyphens to separate distinct words in their names. Example:

void Scene::`prepare_draw(`**DisplayDevice *** *d*`)`

17.4 Global variable names

The names of global variables in VMD follow the same rules as for variables which are members of a particular class. It is recommended to use one of the following suggestions when naming a global variable:

1. Choose a single-word name, and use all lower case.
2. Choose a name similar to the class name of the variable, and prepend "vmd" to the name.
3. If it is assured there should be only one instance of a particular class, and the instance should be global, choose a name identical to the *base* class of the variable but with the first letter in lower case.

Examples:

scene
moleculeList

17.5 Global function names

Global functions, which are found mainly in the files `utilities.C` and `Global.C`, follow one of the following two rules:

1. Functions in `Global.C` have the word `VMD` prepended, and contain no hyphens. Distinct words in the name begin with a capital letter, except the first word after the `VMD` prefix.
2. Functions in `utilities.C` contain all lower case letters, and either contain no hyphens at all, or contain hypens to separate words.

Examples:

> **void** `VMDinitUI()` (in `Global.C`)
> **double** `time_of_day()` (in `utilities.C`)

17.6 Source code file names

Each C++ class, except for very small classes used by only a small number of other classes, should be placed in a separate `.h` file, and if necessary a `.C` file as well. The base name of the files for the class should be identical to the class name itself. Examples:

> **DisplayDevice.h** and **DisplayDevice.C**
> **NameList.h** (no **NameList.C** required)

C source code files should have a `.c` extension (that is, use a lower case c), while C++ files should have a `.C` extension. ALL header files should have a `.h` extension, and any Fortran files should have a `.f` extension. Latex files should end in `.tex`.

17.7 Documentation file names

Documentation files, which are in Latex format, all contain a `.tex` extension and begin with a prefix of one of the following: `vmd`, `ig`, `pg`, or `ug`. Files with a `vmd` prefix are used by all the Guides; files with the remaining prefixes are used only by that particular Guide. For all files except the main driver Latex files for the Guides and the Book, the filenames contain a hyphen after the prefix and a concluding descriptive word. No capital letters are used. Examples:

```
vmd_macros.tex
pg_chapters.tex
ug.tex
```

18 Creating new files

When new files are to be added to VMD, for any of the different subdirectories, the following steps should be followed:

1. Once an initial version of the file is ready, it must be formatted properly to conform to the standards used by files similar to it, i.e., to be similar to other `.h` files if the file is a C/C++ header. The rules for formatting each type of file are given in the following section.

2. If RCS is being used, then after formatting for almost all cases the file must be put in the RCS system. The only type of files which should NOT be placed in the RCS are the following:
   - Binary executable files, i.e., binary executable files which are in the `bin` directory.
   - Image files used for the documentation, i.e., which are in the `doc/pictures` directory. There is a single copy of these files, which cannot work with RCS due to the way RCS stores files internally.
   - Libraries, including the library archive file and the library header files. These should all be in a single place, and all users should simply have a link from their working `lib` directory to the single official VMD `lib` directory.

   To put a file in RCS, follow the steps in section 10.
3. Once placed in the RCS, the configure script must be updated, if necessary, to include the name of the new file in the appropriate list. For example, if a new C++ object is being added and there are two new files *newobj.C* and *newobj.h*, the names of these files would go into the lists of VMD C++ source code and header files in the configure script.

18.1 File formats

There are several types of files which may be added to the whole set of VMD working files. This section describes how to format them, and where to put them.

Many types of files (particularly, C/C++/Fortran source code or header files, Latex documentation files, and shell script files) require an RCS *header* at the beginning of the file. This header should be placed at the very beginning, before any other text in the file. It consists of a set of comment lines which describe the name, purpose, and history of revisions to the file. This is done by using RCS *keywords* embedded in the comments, which are replaced by the proper values when the file is checked out, and by having a section in the comments for a basic description of the purpose of the file. Templates of RCS headers for each of the different file types which require them are provided in the directory `RCS`. When a new file is created, a copy of the relevant header template should be placed at the top of the file, and the file description inserted as comments in the section of the template provided for this purpose. The descriptions below of how to format each file also describe the name of the RCS template to use.

- **Documentation text files**
  The documentation for VMD is in Latex, and files should have a `.tex` extension. The files should all be placed in the `doc` directory, be put in the RCS, and have the RCS header `RCS/RCSheader.tex` prepended.
- **Documentation image files**
  Image files for the documentation should be placed in the directory `doc/pictures`, which should be a link to a directory writable by all people working on VMD. These files are NOT to be placed in the RCS, due to the problems with how RCS stores files.
- **Source code files**
  All source code files, either C, C++, or Fortran, should be placed in the `src` directory, along with all header files. These files should be entered into the RCS; a copy of the RCS template `RCS/RCSheader.h` should go at the start of header files, and a copy of `RCS/RCSheader.c`

should be placed at the start of C and C++ source files. Fortran files should begin with a copy of the file `RCS/RCSheader.f`.

All header files should bracket their text between a `#ifndef ... #endif` pair, and define a macro to indicate the header file has been processed. For example, right after the RCS header should come the text

```
#ifndef DISPLAYDEVICE_H
#define DISPLAYDEVICE_H
⋮
#endif
```

- **Shell script files**
  Files which are `sh` or `csh` scripts should be placed, most likely, in either the main working directory or the `bin` directory. These should be entered into the RCS, and have the RCS header `RCS/RCSheader.make` placed at the beginning. When adding such files to the RCS, care must be taken to have the *comment leader* for the RCS file set properly. This is done by using the command-line switch

  ```
  -c"# "
  ```

  added to the options to the `rcs` program (described in section 10).

- **Data files**
  Files which contain data or configuration parameters needed by VMD should be placed in the `data` directory, and be put in the RCS. In most cases, the `"#"` symbol is the comment character, and so they can have the same type of RCS header as used for shell scripts. This is true, for example, for the file `data/.vmdrc`, which is in the RCS.

Program Structure

VMD is written in C++ and uses an object-oriented methodology for all program components. This greatly aids in making VMD modular and extensible to such things as new types of display devices, new user interface libraries, and new molecular data file formats. As this chapter will explain, VMD uses the *inheritance* and *polymorphism* features of an object-oriented design (in C++) extensively. Users interested in extending or modifying VMD should have a good knowledge of programming in C++; this Guide uses terminology and explainations which assume such knowledge. Two excellent references for users interested in learning more about C++ are books by Lippman and Strousop.

The VMD source code consists of sets of utility functions, independent and interrelated object classes, and global variables. The classes are organized into a few logical program components, e.g., the component responsible for displaying images, the user interface component, and others. This chapter describes the structure of each of these components and mentions the objects which the components contain. These components utilize several external libraries for such things as the graphical user interface, control of external spatial tracking devices, and support for displaying images in the CAVE display environment; the particular libraries which VMD uses are described at the chapters conclusion.

19 Program component descriptions

For each program component, a figure illustrates the classes which comprise the component, and their relationship among each other. Objects are generally represented as rectangles labeled with the class name. Classes which are derived from one or more parent classes have a solid arrow pointing from the derived class to the parent class (an *is a* relationship). Classes which use some capability of another class, but are not either derived from nor contain the other class, have a dashed line pointing to the class which is being used (a *uses a* relationship). Classes which contain one or more instances of another class indicate this by having the rectangle for the contained class located within the containing class (a *has a* relationship).

VMD has a few important *base classes*, for which a single global instance is created either during program initialization or as a result of a user request. For several of these base classes an instance of a specialized class derived from the base class is created, and the address of the instance assigned to a pointer of type *base* *, with the instance accessed through *virtual functions*. In the figures describing each program component, important base classes are indicated in **bold** font. If a global instance of a base class exists, the name of the global variable is shown in `typewriter` font in parentheses below the name of the base class in the figure.

19.1 Utility objects

Figure 1 illustrates the utility objects used in VMD program development. Several of these objects are C++ *templates*; instances of these classes are creates for different types, for example a `Stack` of integers or a `Stack` of `char *` items. These template and the other global objects are each detailed in section 23.

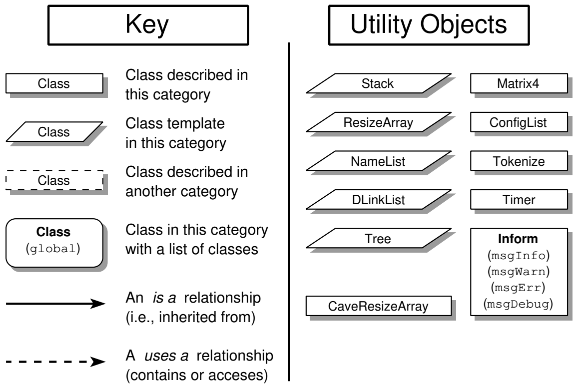

Figure 1: Utility objects used in VMD, and a key describing the function of diagrammatic items in these object description figures.

The **Inform** class is deserving of particular notice, since it is one of the most widely used objects in VMD. This class provides a streams-like object which is used to print messages to the VMD console. There are four global instances of this class, which are used for the following purposes:

- `msgInfo`
  Used to display informative messages, which provide some data or message to the user which is either requested or "typically" provided. For example, the messages describing the current

status of the process of reading a molecule from a file or a network connection are printed by sending text to the `msgInfo` instance of an **Inform** object.

- `msgWarn`
  Used to display warning messages. These are messages which do not indicate a fatal program condition, but which caution the user that things may not be going as they expect.

- `msgErr`
  Error messages are displayed by sending text to the `msgErr` instantiation.

- `msgDebug`
  Where debugging messages are printed. It is possible to exclude the source code lines which print debugging messages when VMD is compiled, to produce a smaller and faster executable, by eliminating the `DEBUG` compilation option.

To use an **Inform** object, text and data are given to object using insertion (`<<`) operators, i.e.,

```
msgInfo << "This is message number " << intVariable << "." << sendmsg;
```

The `sendmsg` manipulator ends the message; this will cause the message to be printed to the console, prepended by a string indicating the type of messages, and appended by a newline.

There are also several utility functions which are not part of any class; these are described in section 22.

19.2 Display objects

Figure 2 illustrates the structure and relationship of the many objects which are used by VMD for storing and rendering graphical objects. These objects are described in detail in section 24. Images are displayed through the use of four main objects (and subclasses thereof):

1. **DisplayDevice** objects, which are responsible for rendering a list of drawing commands to a given device or file;

2. **DispCmd** objects, of which there are several subclasses; each **DispCmd** subclass represents a single drawing command, such as a command to draw a line or to set the current color.

3. **Displayable** objects, which each represent a single graphical item with a list of drawing commands that may be rendered to a **DisplayDevice**;

4. **Scene** objects, which store lists of **Displayable** objects. When requested, a **Scene** takes a given **DisplayDevice** and requests the device to render each of the **Displayable** objects it stores.

Device-Specific Rendering. **DisplayDevice** is the base class for objects which do the actual rendering of a **Scene**. Each derived class of this parent must provide versions of virtual functions which prepare the device for drawing (i.e., clear a screen or open a file), render a list of drawing commands, and update the display after drawing, among many other functions. This class is where library-specific drawing commands are encapsulated, for example there exists a `OpenGLDisplayDevice` subclass which is used to draw images using OpenGL graphics hardware. A **DisplayDevice** can also be defined which renders a **Scene** to a file instead of a monitor, i.e., to a postscript file, a raytracing program input script, or a bitmap image file.

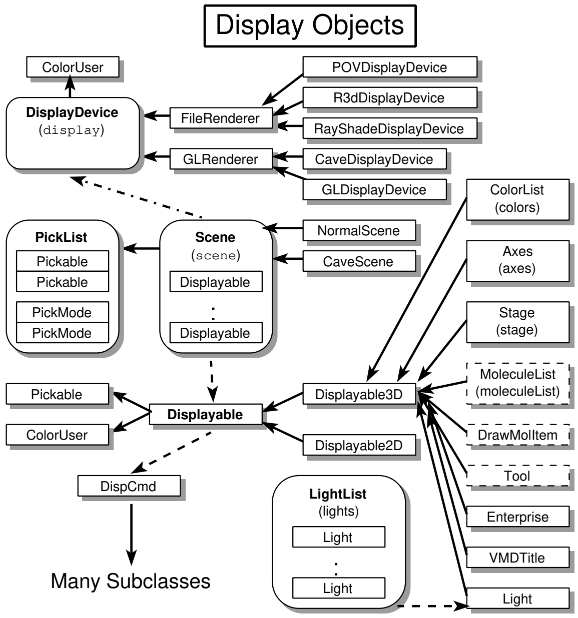

Figure 2: Display objects used in VMD.

Drawing Tokens. **DispCmd** objects are used by **Displayable** objects to construct a list of drawing commands (tokens) in a format which can be processed by a **DisplayDevice**. When rendering an image, a **DisplayDevice** does not work directory with a **Displayable**; instead, the **DisplayDevice** is given a simple byte array into which drawing commands have been assembled. Each **Displayable** contains one or more of these *drawing lists*; they are stored by the **Scene** and processed by the `render` routine of a **DisplayDevice**. The **DispCmd** objects act to put the data into these drawing lists in the proper format for the operation specified. Each **DispCmd** object appends its data to the current end of a provided drawing list in this format and order:

1. Integer code specifying the drawing operation (codes are defined in the file `DispCmds.h`).
2. Integer stating the size of the command, in bytes (not including the code or this size count).
3. Data necessary for the drawing command (i.e., XYZ positions of the endpoints of a line or a cylinder).

Why are things done this way? There are a few reasons:

- Speed – Once a compact display list has been created in a single block of memory, that block may be read quickly by the **DisplayDevice** when drawing the object. Putting it all as a contiguous block helps take advantage of cache memory.
- CAVE access – The CAVE programming paradigm now requires the drawing program to fork into several different processes, including one process for each wall and an "update" process. All these processes run on the same computer (a multi-headed SGI, i.e. a tower Onyx) and use shared memory to communicate what to draw between the update process and the drawing processes. Using a list of simple integer codes and copies of the coordinates

for drawing lines, triangles, etc. instead of, say, a list of pointers to **DispCmd** objects helps reduce the amount of shared memory required and simplifies the drawing processes. Drawing in the CAVE then requires only a specialized **CaveDisplayDevice** and `CaveScene` to deal with this shared memory requirement (along with a few other CAVE support routines).

Displayable objects. Each **Displayble** object contains a display list, as well as information about the item to be drawn such as whether to display or hide the object, it's current *transformation* (how much to rotate, scale, and translate the object), whether it is a 2D or a 3D object, whether it is *fixed* (set to ignore any requests to scale, translate, or rotate the object), and a list of *children.* A **Displayable** *registers* with one or more **Scene** objects, so that the **Scene** then will contain a list of all the **Displayable** objects which should be drawn when requrested. The **Scene** is responsible for providing the physical memory storage for a display list, through a request by a **Displayable**. Each molecule and other graphical item in VMD is a **Displayable** subclass. These derived objects supply the methods to fill the object's display list with drawing commands for the item to be drawn, i.e., the commands to draw the points for each atom and the lines for each bond in a molecele.

A key feature of each **Displayable** is that it may contain any number of *child* **Displayable** objects, and may also be a child of some other *parent* **Displayable**. Each child may be individually translated, rotated, etc., and may be turned on or off individually; but operations to a parent such as a transformation typically affect *all* the children of that parent as well. For example, if a parent is currently being hidden, so will be all the children of that parent. Also, only **Displayable** objects which have **no** parent register with a **Scene**; all children of a parent are drawn properly when the parent is drawn, so it is only necessary for the very top-level parent to be stored in the **Scene**.

The Scene. Since it has been discussed quite a bit already, it is somewhat obvious at this point that **Scene** objects are used to maintain a database on what should be drawn to a user-specified **DisplayDevice**. A **Scene** contains routines for applying transformation such as rotate, scale, etc. to all the (non-fixed) **Displayable** objects which have registered with it, and contains routines to manage the memory used for display lists. Each **Scene** contains actually two categories of lists of items, and in each category there are two lists, one for 2D objects and one for 3D objects. These two types of lists are:

1. Pointers to all the **Displayable** objects which have registered with the **Scene**.
2. Pointers to the beginning of each display list (essentially treated as arrays of `char`s). There may be more of these lists than there are registered **Displayable** objects, since each **Displayable** (regardless of whether it is a parent, child, or both) provides the **Scene** object with a copy of the pointer to its display list.

A **Scene** is the object primarily responsible for collecting all the objects that are to be drawn and for giving these objects to a **DisplayDevice** for rendering. This is done as follows:

1. The routine `scene->prepare(DisplayDevice *)` is called by the user. The **Scene** will call a routine for each registered **Displayable** that allows the object to prepare to be drawn. This preparation may include, for example, changing which frame in an animation should be shown, or updating the current position of a 3D pointer, or most anything else that needs to be done each time the **Scene** is drawn.
2. The routine `scene->draw(DisplayDevice *)` is called by the user. This routine proceeds in the following steps:

(a) Prepare the **DisplayDevice** for drawing 3D objects.
(b) For each 3D display list stored by the **Scene**, call the routine `display->render(void *)` with the pointer to the display list.
(c) If stereo is being used, repeat the following step for the other eye.
(d) Prepare the **DisplayDevice** for drawing 2D objects.
(e) For each 2D display list stored by the **Scene**, call the routine `display->render(void *)` with the pointer to the display list.
(f) Update the **DisplayDevice** after all drawing is complete.

As shown in figure 2, there are many classes derived from **Displayable**, and actually in fact from **Displayable3D**. Global **Displayable** objects of note are **LightList**, which contains a list of **Light** objects; **Axes**, which displays a set of XYZ axes in a corner of the display; **Stage**, which displays a checkerboard panel to one side of the objects in the **Scene**; **VMDTitle**, which displays the VMD title credits and rotating letteres; and **ColorList**, which is described next. The **Light** objects each represent one light used to illuminate the objects in a **Scene** which have defined *material characteristics* for their surfaces.

Colors. Colors in VMD are handled mainly by the **ColorList** object. This maintains a list of 16 unique colors in the VMD "colormap", as well as a color scale of 1024 colors arranged in a selectable pattern of blue → green → red or blue → white → red. For each color there are two versions, a solid color and a semi-transparent color. Each color may be changed through user commands, and for each color there are corresponding material characteristics which are used when solid objects are drawn and the **Displayable** drawing the solid objects requests that materials be used. The **ColorList** also maintains several lists of names which are used to specify a color: using the **NameList** template, a set of color *categories* are stored in the **ColorList**, and for each color category there are any number of color *objects*, which consist of a *name* and *index*. For example, in the color category "Stage" there are two objects, an "Even" object (for the even-numbered stage checkerboard squares) and an "Odd" object (for the odd-numbered squares). The index for these objects indicates which of the 16 VMD colors to use to color that square. **Displayble** objects may request to have new categories created, and to add new color objects to each category. This capability is provided by the **ColorUser** class, from which each **Displayable** is derived, thus all objects can automatically access the colors through the functionality of the **ColorUser** class.

Picking Items. The action of *picking* graphical objects with a pointer, like clicking on an atom with the mouse, is handled by the Display objects as well. Picking objects requires that VMD know where all the points are in space that may be selected, where the pointer is located when a button is pressed, and what to do once an item is picked. This is managed by objects derived from three 'picking' base classes: **Pickable**, **PickList**, and **PickMode**, which are discussed below.

Each **Displayable** object is derived from the **Pickable** base class; a **Pickable** is an object which contains a drawing list with special **DispCmd** tokens which indicate "this is a point in space which may be selected". These *picking tokens* do not result in anything being drawn on the screen; they are used by the **void Displayable::**`pick(void *)` routine to determine if a pointer is over a selectable item. Each picking token contains also an integer *tag*, which is returned by the picking routine when the pointer is found over an item. When a **Displayable** wants to put picking tokens in its display list, and then wants to be have its display list be checked for pickable points, it must register with a **PickList** object (described below). This is done by calling the routine

void Pickable::`register_with_picklist`(PickList *)

in the constructor of the **Displayable**, using the pointer to the **Scene** in which the **Displayable** is registered (this is because **Scene** is derived from **PickList**). There are several virtual functions which must be provided by an object derived from **Pickable**, which will be discussed below.

There are two types of pointers which may be used to pick items, 2D or 3D. 2D pointers (i.e. the mouse) report their position in 'relative scaled coordinates', that is, the X and Y position of the mouse is provided to the rest of the program as values in the range 0 ... 1, so that the lower-left corner of the graphics window is (0,0), and the upper-right corner is (1,1). The coordinates of 3D pointers, on the other hand, are given to the rest of the program as just the location of the pointer in 'eye' coordinates, i.e. the 3D position after it has been transformed by any internal transformation matrix of the pointer. Each pointing device (the mouse, 3D spatial trackers, etc) must be in a certain *picking mode*, which determines what action is done when an item is picked; new modes can be easily added to a central source by any object that wants to extend the usability of the picking mechanism. A picking operation consists of three phases:

1. Start: when a button is pressed by a pointer, a command is issued which checks to see if something is under the current pointer position. If so, a new picking operation is started, and continues until the button (or whatever the pointer is using) is released.

2. Moving: As the button is held down, if the start of the picking operation did indeed find something under the pointer, commands are executed as the pointer moves to allow objects to be continually manipulated by the pointer.

3. End: When the button is released, some final action may be required, and this is signaled by a command to end the current picking session.

The effect on objects during these phases may be dependent on which button was pressed (two buttons, *left* and *middle*, are assumed), and what the current mode was when the button was pressed. **PickMode** objects are used to handle the different actions required for different picking modes.

Each picking mode is embodied by a special derivation of the **PickMode** abstract base class. An example, **PickModeQuery**, has been added which does the very simple job of just printing out the name of the **Pickable** object when the picking is ended (and only if the pointer position does not move much between the start and end). Each **PickMode** simply contains three virtual functions:

1. **int PickMode::**`pick_start`(DisplayDevice *, Pickable *, int button, int tag, int dim, float *pos)
   When a pick is successfully started (which means the *tag* of the point which was picked has been determined), this routine is called for the **PickMode** object corresponding to the current pick mode, to let that object perform some special action based on the selection. The item selected is provided, as is the button pressed, the dimension (2 or 3) and position of the pointer, and the **DisplayDevice** used to find the selected item (this is necessary to allow access to routines which convert 2D screen positions to 3D world coordinates). Finally, the tag of the point in the **Pickable** that was actually clicked on is provided (more on this later).

2. **int PickMode::**`pick_move`(same args)
   Again, called only for the **PickMode** object of the current mode for the pointer used; this is called every time the pointer moves to a new position.

3. **int PickMode::**`pick_end`(same args)
   Same as the others, just called when the pointer button is released.

A **PickList** object contains a list of all the current picking modes which the mouse may be in; these all have unique id's from 0 ... num_modes - 1. The Mouse object (described in a later section) gets this list and adds a submenu to the graphics window pop-up menu. When the mouse is in a picking mode, the cursor changes to a crosshair. The **Scene** is derived from **PickList**; it is the **PickList** which maintains all the coordinating data to manage all the objects which can be picked, and all the different picking modes. The **PickList** maintains two lists:

1. All the **Pickable** objects which have registered themselves as items which contain points which can be selected with a pointer.

2. All the **PickMode** objects which are used to provide different action capabilities to the same pointer, in an extendible fashion.

The pointer objects call routines in **PickList** to get the current number of names of picking modes, to check if the current pointer position is over a pickable point (and to find out which one), to tell the program that the pointer is moving while an object is being picked, and to tell the program that the button has been released following a picking operation. Thus, **PickList** is the "coordinator" for all picking operations. **PickList** contains a routine similar to the draw routine, but which instead checks for picked item and executes the proper action if one is found. The algorithm used is:

1. For each **Pickable** registered, check if it is interested in the current mode. If so, call **int Pickable::**`pick_start(see later)`. If not, skip the **Pickable** and go to the next. 2) Then, call **int PickMode::**`pick_start(f)`or the **PickMode** object corresponding to the current pick mode.

To set up a **Displayable** to operate properly as a useful **Pickable**, these things must be added to the **Displayable** (see Axes.C and Axes.h for a good example of how to do these things).

- A version of the virtual routine **int Pickable::**`want_pick_mode(int)` must be supplied, which returns TRUE if the given pick mode is one which the item is interested in. If no version of this routine is supplied, the default is to return FALSE, which means the **Pickable** will never be told when a pick starts, moves, or ends (this may actually be desirable, however, if the **PickMode** object is to do all the work).

- If the **Pickable** will need to add new **PickMode** objects to the **PickList** (which is the same as the **Scene** object which the **Displayable** is added to), or find the index for an existing mode:

  1. The routine **int Pickable::**`add_pick_mode(char *, int)` should be called to add a new mode. The name given is used by this routine to check the **PickList** if the mode exists already. If so, this returns the current index of that mode without creating a new instance of the proper **PickMode** object. If the mode has not yet been added, the 2nd arg is a code value given to the virtual function **int Pickable::**`create_pick_mode(int)`, which will create a new **PickMode** instance (the particular subclass based on the value of the integer arg). This avoids unnecessary duplication of **PickMode** objects.
  2. The index must be saved by the **Pickable**, and used by `want_pick_mode` to report if the pointers current pick mode is one the **Pickable** is interested in.

- In the constructor for the specialized **Pickable**, the routine **int Pickable::**`register_with_picklist`(PickList *) must be called, to tell the **PickList** (**Scene** in this case, since **Scene** is derived from **PickList**) that the **Displayable** can be picked.

- Also in the constructor go calls to `add_pick_mode`, if necessary.

- Pick drawing tokens must be placed in the draw list of the object.

- Finally, if the **Displayable** will be doing some action based on when it is picked and moved (independent of action performed by **PickMode**'s during this same time), versions of `pick_start`, `pick_move`, and `pick_end` must be supplied. By default, these virtual functions do nothing. Again, see Axes for an example (Axes actually only supplies a version of the `pick_move` routine, since no action is required by Axes at the start or end).

A key thing that may be needed during the `pick_move` phase for both **Pickable** and **PickMode** objects is the ability to convert a 2D screen coord (the relative scaled coords x and y, valued 0 ... 1) to a 3D world coordinate. The difficult of course is the abiguity in what the Z-axis coordinate should be. The routine

DisplayDevice::`find_3D_from_2D`(float *A3D, float *B2D, float *B3D)

takes a 3D world coordinate (at point A, A3D), and a 2D relative scaled coordinate (the screen position of point B, B2D), and returns the 3D world coordinate for point B (B3D). This works assuming the eye is looking along the Z axis. The coordinate returned is the point where the line formed by the 2D's projection back into 3D space intersects the plane parallel to the XY plane which contains point A (i.e. the point will have the same Z-coordinate as the given point A).

19.3 Molecule objects

The objects used for displaying and rendering graphical objects in VMD are quite general, and can be used to draw essentially anything. The objects used by VMD to create, store, and manipulate molecules, which are illustrated in figure 3, are much more specific to the purpose of VMD, which is to visualize to dynamic properties of biopolymers (in particular proteins and nucleic acids). The heart of this category of object classes is the **Molecule** class, which is actually inherited from a number of base classes and for which several subclasses exist. **MoleculeList** is an object which maintains a list of all current molecules. There are also several helping objects which store data about particular components of each molecule.

At the very top level of the **Molecule** hierarchy is the **Animate** class, which stores a list of **Timestep** objects. Nothing is known about the molecule at this level other than the number of atoms; a **Timestep** stores simply arrays of floating-point values for each of these atoms for each discrete timestep in the trajectory of the molecule. The **Animate** class also maintains the *current* frame in the trajectory, and the direction (i.e., fast-forward, reverse, pause) and speed of animation. The **Timestep** objects, one for each frame of animation in each molecule, are stored simply as pointers in a **ResizeArray** instance within the **Animate** object. A **Timestep** is currently quite simple, and stores data as publicly-available floating point arrays which are allocated when a new **Timestep** is created (or for some data which may be optionally stored for each step, by the users request). This is done primarily for speed since this data is accessed quite often. It may be helpful to improve this class in the future, by making a much more general method to store different types of atomic data for timesteps which would not require a change to the **Timestep** class each time.

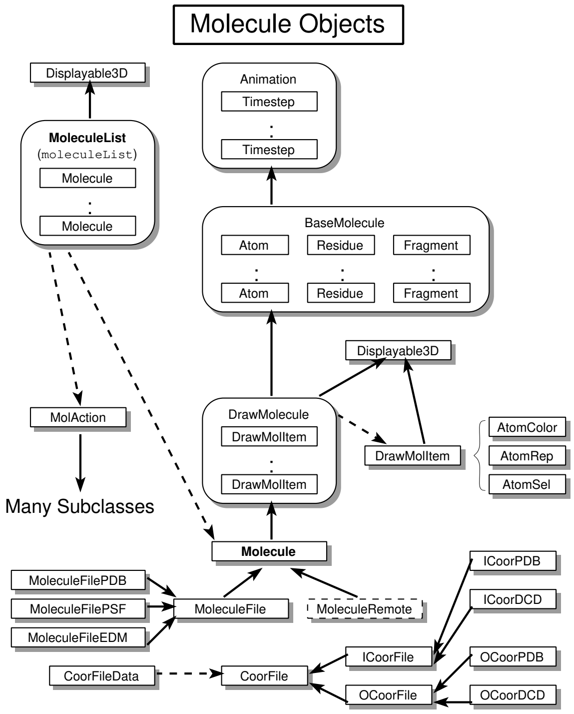

Figure 3: Molecule objects used in VMD.

For example, the **Timestep** may just store a list of pointers to something like a **TimeStepData** class instance, where each **TimeStepData** would store some number of floating-point values.

At the next level, the **BaseMolecule** object is inherited from the **Animate** object. This object stores all the basic information which comprises the structure of the molecule. Data about the coordinates are stored by **Animate**, while **BaseMolecule** stores how the atoms are connected, what residues and segments exist, etc. When it is created, a **BaseMolecule** is empty, indicating no atoms or anything present. A virtual function **int BaseMolecule::**`create()` is used by **BaseMolecule** and all other class derived from **BaseMolecule**; this function is called when a new molecule is to be created, and derived classes do their creation tasks after which they call `create` for the parent class. **BaseMolecule** does NOT contain any data or functions for the drawing of the molecule, just for storing the structure. After a new molecule has been read in from some files or from a network connection, the structure of the molecule is analyzed and stored in a retrievable format. Several small classes help in this storage; they include the following:

- `Atom`
  This object stores the data for one atom in a molecule, including the name, segment, type, charge, mass, etc. A list of these **Atom** objects are stored in a **BaseMolecule**. Atoms are numbered 0 … N-1, where N is the number of atoms in the molecule; a pointer to the nth **Atom** object may be obtained through the routine **Atom * BaseMolecule::**`atom(`**int** n`)`. Bonds are not stored as separate objects; instead, in each **Atom** object there is noted the number of bonds in which the **Atom** participates, and a list of the indices of the other atoms to which it is connected. Note that a bond is effectively stored twice (to speed rendering of the molecule), once for each atom which defines the bond.

- `Residue`
  Each residue such as an amino acid or nucleic acid is referenced by a single **Residue** object, which stores the indices of the corresponding atoms and information about how the residue is connected to other residues. A list of all the residues in a molecule is kept in the **BaseMolecule** class.

- `Fragment`
  A *fragment* is defined as a contiguous string of residues. This may be as short as one (i.e., a water molecule), or as long as an entire protein from N-terminus to C-terminus. A **Fragment** object stores a list of residues which form a fragment, noting their order and connectivity. A **BaseMolecule** stores a list of all the fragments which are found within the molecule.

- `NameList`
  A **BaseMolecule** also contains several lists of names which are basically **NameList** templates. They store the lists of unique names which occur in the molecule, i.e., the list of all atom names. These lists are maintained as public data member for speedy access by other parts of the program.

From **BaseMolecule** and from **Displayable3D** the **DrawMolecule** object is derived. This level of the **Molecule** hierarchy stores all the information about how to *draw* the molecule. A molecule in VMD is drawn as a composition of one or more *representations* of the molecule structure, which are contained within a **DrawMolItem** object (described later). A **DrawMolecule** stores a list of all the different representations (**DrawMolItem**) of the molecule that the user has selected, and contains routines to add, change, or delete these representations. Since it is also a **Displayable**, a **DrawMolecule** can specify its own display list, but currently all molecule drawing commands are contained within **DrawMolItem** objects.

A **DrawMolItem** is also derived from **Displayable3D**; its function is to maintain the display list with the proper drawing commands to render one specified representation of a molecule. When a new **DrawMolItem** is created, it is given the molecule for which it is to render an image, and instances of the following three objects which describe exactly *what* the representation is to be:

- `AtomColor`
  Stores the color index number which is to be used for each atom in the molecule when it is drawn; essentially, *how to color* the molecule. The coloring can be done in any number of ways, for example each residue a different color, or shaded through the color scale from the midpoint of the molecule outwards.

- `AtomRep`
  Stores what shape to draw the molecule as; essentially, *how to draw* the molecule. For example, bonds may be represented as thin lines, or solid cylinders, or not drawn at all.

- `AtomSel`
  Which atoms to draw. This objects takes as input a text string with an atom selection command, and determines from the string which atoms of the molecule the user wishes to have drawn.

An instance of all three of these objects, taken together as a group, completely define how a represenation of a molecule should be drawn, and so each **DrawMolItem** requires one of these objects to allow it to construct the list of drawing commands for the representation. Each **DrawMolItem** is a child **Displayable** of the parent **DrawMolecule**, and may be independently turned on or off. By doing it this way, a complex image of the molecule may be constructed by separate more basic components, which may be separately manipulated, while the set of components are kept with the same transformation (rotation, scaling, and centering) applied to the molecule as a whole.

The main base class **Molecule** is then derived from **DrawMolecule**. This is the level at which most other objects in VMD work with molecules, as pointers to instances of a **Molecule** class. In fact relatively little functionality is includes at this level. What this class does do, in fact, is provide the routines for reading in or writing out of animation frames from or to different coordinate file formats (i.e., PDB or DCD files). This is done through the use of a **CoorFileData** object, which encapsulates the information on *how* to read/write such a coordinate file (this includes storing which frames are to be read or written, the coordinate file format, and the current status of such an operation). Since **Molecule** is a subclass of **Displayable**, it has a `prepare` virtual routine which is called each time the **Scene** is to be drawn to the current **DisplayDevice**. **Molecule** uses this call to `prepare` to read/write a single coordinate set from/to the current coordinate file, if one is being processed. Thus, a coordinate file is not processed in one single operation, instead one frame is processed each time the **Scene** is drawn. This allows VMD to continue to animate and check for user commands while a coordinate file is being read or written. For the actual coordinate file reading or writing, the **CoorFile** base class and derived **ICoorFile** and **OCoorFile** classes abstract the action of taking a set of XYZ positions for a molecule and reading or writing a trajectory file. Specific versions of these classes for PDB and DCD files are used, and any other number of trajectory formats may be supported by developing new subclasses of **ICoorFile** and **OCoorFile**, with also an update to **CoorFileData**.

Up to the point just described are all the classes necessary to store and manipulate a molecule. However, there are several different ways for a molecule to be imported into VMD, and each method has a specific subclass of **Molecule** to provide the functions to read in the proper data and store it into the standard internal format of the **Molecule** class hierarchy. Currently, the following subclasses of **Molecule** exist:

- `MoleculeFile`
  Used to create a new molecule by reading structure information from a file, such as a PDB or PSF file. This object provides routines which understand the format of these data files, and which convert the data within the files to the internal molecular structure format used by **BaseMolecule**. This action is done when the `create` routine is called; after reading in the file and successfully creating the molecular structure, the `create` routine for **Molecule** is called, which then calls `create` for **DrawMolecule**, . . .

- `MoleculeRemote`
  This reads a molecular structure directly from a molecular dynamics simulation program running as a separate process on the same or another computer. The data is transferred directly over the network, without using any intermediate files, using sockets (see section 3). Otherwise, it functions identically to **MoleculeFile**.

Thus, the steps in creating a new molecule in VMD are as follows:

1. Create a new empty **Molecule** object, by creating an instance of one of the subclasses in the list above and assigning it to a **Molecule *** variable. This new molecule is empty, with 0 atoms, 0 bonds, etc.

2. Give information to this new object on how the molecule is to be created. Much of this data is actually specified as arguments to the constructor of the special subclass used.

3. Call the function `create` for the new **Molecule**, which will do most of the work in creating the new molecule. During this process, the following actions are done:
   (a) Data in the original format is read in from the proper source.
   (b) Each atom is added one by one to the molecular structure via routines in **BaseMolecule**. The lists of unique names for molecule components are constructed during this process.
   (c) After all atoms are read, the bonds between atoms are stored. With the atoms and namelists, this completely defines the basic structure of the molecule. This step may involve finding the bonds through a nearest-neighbor search (necessary when the structure is read from a PDB file).
   (d) "Extra" data helpful in displaying the molecule is read, such as which atoms are hydrogen-bond donors or acceptors.
   (e) After all data is read and the basic structure defined, the `create` routine in **BaseMolecule** analyzes the molecule and creates the lists of **Residues**, **Fragments**, and anything else which helps define the structure.
   (f) An initial representation of the molecule is created by **DrawMolecule**, and the display list necessary to draw the molecule is constructed.
4. After it has been created, a request to read in any trajectory files is given to the new molecule if necessary.

The object which keeps track of all the currently-loaded **Molecule** objects is **MoleculeList**, of which there is exactly one in VMD assigned to the global variable `moleculeList` (although there is no reason why there could not be more than one). **MoleculeList** is an important object: it manages all the molecules, contains routines to allow an operation to be performed on a number of molecules at the same time, and supplies information on how the molecules are related to each other. **MoleculeList** is derived from **Displayable** as well: it is the top-level parent **Displayable** with all **Molecule** objects as child **Displayable**s. Thus, turning off the **MoleculeList** turns off *all* the molecules, and similarly rotating or scaling the **MoleculeList** does so to all the molecules. There are no drawing commands currently for the **MoleculeList** itself (although they could be added for something which indicates relationships between the molecules). This is a useful trick in VMD: have a container class which is derived from **Displayable**, but which has no drawing commands of its own; instead, have it contain several child **Displayable** objects which form components of the complex object which is to be drawn. By applying rotations, translations, etc. to the container class, all the child components are similarly transformed, and they may be separately altered or turned on or off.

19.4 Remote connection objects

To create, maintain, and access a connection to a molecular dynamics simulation running on a remote computer (which may just happen to be the same computer running VMD), the objects illustrated in figure 4 are used.

The **Remote** object is used to encapsulate the functionality for initializing, accessing, and controlling a remote simulation. This object contains all the data necessary to create a new simulation or to attach to an already-running simulation. It also contains member functions for querying whether a simulation may be run on another computer, for retrieving the list of available MD

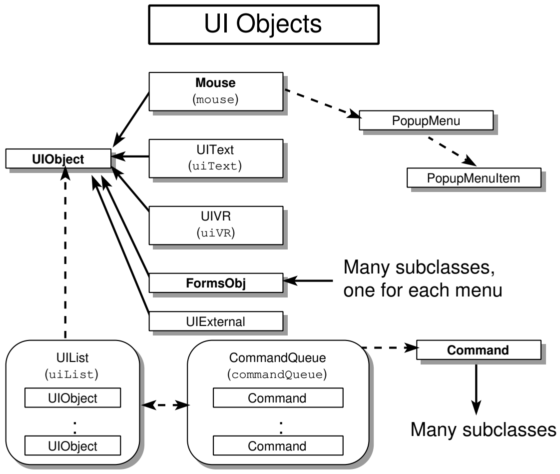

Figure 4: Remote simulation access and control objects used in VMD.

programs or jobs on that computer, for obtaining and modifying parameters necessary to start a new job, and for actually starting, attaching to, or stopping an MD simulation.

There are three phases in the task of displaying a molecule in VMD which is being simulated by a separate MD process; they must be accomplished in this order:

1. Select a computer for the MD simulation, and connect to it to retrieve the list of running jobs there and the list of available MD simulation packages. When this is done by a **Remote** object, the lists are stored and may be retrieved by other objects in VMD. If the proper daemon process is not running on the selected computer, this step will fail.

2. If the user is to connect to a job which is already running, this phase is skipped; if the user wishes to start a new job, this phase is where the parameters for the job are retrieved from the remote computer and modified to the users requirements. This requires first selecting which one of the possible MD simulation programs the user wishes to run, and then having the **Remote** object retrieve the list of optional and required parameters for that program. This list is stored **Remote**, which also supplies routines to query and change these settings. Once this is accomplished, a routine in **Remote** is called to start a new simulation (a similar routine exist to attach to an already-running job). After this is called, the simulation starts, loads its data, and this connection procedure proceeds to the final phase.

3. Once a connection is made to a *running* application (as opposed to the state for the earlier phases where a simulation was being *initialized*), the molecular structure is transferred from the remote process to VMD, and coordinate sets are then sent over as they are calculated. The structure sent via the network is stored in an internal format in the **Remote** object (basically, as a `struct`; this data is then given to a **MoleculeRemote** object (described later) and used to create a **Molecule** in VMD. The **Remote** object also has routines for changing the state of the simulation after it is running, and for *disconnecting* from the simulation (which leaves it running, and in a state where VMD can attach to it again later if preferred) as well as for *killing* the simulation process.

As mentioned, once the connection is established and the structure sent over the network to

VMD, a new **Molecule** can be created. At the very beginning of the procedure just outlined for creating a connection, an instance of a **Remote** is created to proceed through the phases. After this is complete and the connection is successful, the **Remote** instance is given to a new **MoleculeRemote** object, which uses the data in **Remote** to construct a new molecule in VMD just as if the data were being read from a file. If another simulation is to be started or attached to, another **Remote** instance must be created. Any number of simulations may be attached to by VMD during a single session, even all the same time. However, there can only be one **Remote** object being used to initialize a new connection at any one time. Thus, there is one global instance of a **Remote** object, referred to by the global variable `remote`; this is used for setting up a simulation, and if the setup is successful the instance is used to create a new **MoleculeRemote**, after which a brand new instance of a **Remote** is created and assigned to `remote`.

MoleculeRemote acts very much like a **MoleculeFile**, except structure data is retrieved from the provided **Remote** instance. It also provides a version of the `prepare` virtual routine (originally defined in the **Displayable** class). When `prepare` is called, the **MoleculeRemote** checks for and processes any new data is available from the remote connection (i.e., new coordinate sets recently computed by the simulation). It also maintains any special items used for graphical display of the simulation, for example the **DrawPatch** object. **DrawPatch** is a **Displayable**, and is used to graphically depict the shape and status (via different coloring schemes) of the volumes of space in which the molecule moves in the simulation. This acts very much like **DrawMolItem**, by being a child **Displayable** to the **MoleculeRemote** which creates it.

The final object used specifically for remote simulation control is the **RemoteList**, which functions very much like a **MoleculeList** but which instead keeps a list of **MoleculeRemote** pointers. If a molecule is read from just some data files, it is stored by the **MoleculeList** but not the **RemoteList** (since it is not from a remote connection). If the molecule is from a remote simulation, the pointer to the molecule is stored by *both* **MoleculeList** (as a `Molecule *` pointer) and **RemoteList** (as a `MoleculeRemote *` pointer). This allows VMD to distinguish which molecule is part of a presently or previously active simulation. Even if the simulation is terminated, the reference in **RemoteList** is maintained since the molecule was at some point part of a simulation.

19.5 User interface objects

A major design point for VMD is to make it relatively easy to add completely different user interface (UI) methods, and allow for each interface to provide a means for accomplishing tasks that may also be accomplished by using the other interfaces as well. Figure 5 illustrates the objects which are used to realize this design. There are four main or base-class level objects used in this category:

- **Command** objects, which each represent a single task which may be accomplished in VMD;
- **CommandQueue**, which maintains a queue of **Command** instances and which requests these **Command**s to be executed;
- **UIObject** objects, each of which represent one UI component.

Since there are to be several different UI components, there needs to be a way to avoid duplication of the code required to carry out the tasks requested by the user manipulating the user interface. This is the purpose of the **Command** object: each subclass of **Command** represents a single operation or tasks which the user may request to be done via a user interface of some form. These **Command** objects may take parameters to tell them exactly how to perform the task, but are designed to be rather specific about exactly what they should do. For example, **CmdMolNew**

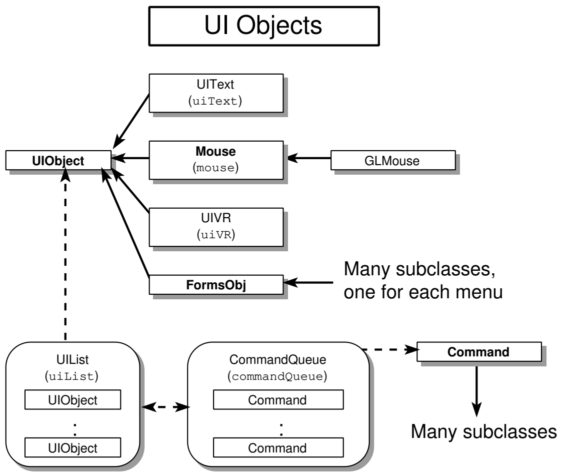

Figure 5: User interface objects used in VMD.

is the **Command** subclass which contains all the code necessary to create a new molecule via the algorithm described earlier, while the **CmdRotate** object knows how to apply a specified rotation to the current **Scene**. Each **Command** has a unique code, defined in the file `Command.h`, and requires derived classes to do the following things:

1. In the constuctor, the data necessary to perform the command must be given to the class and stored for the time when the command will be executed.

2. In a virtual function **void Command::**`create_text()`, use a streams output technique to place within the *protected* variable "*ostrstream *cmdText*" a string which is the text command equivalent of the requested operation. For example, for **CmdRotate**, if `deg` is the amount specified to rotate the scene, the function contains lines such as these:

```
*cmdText << "rot " << axis;
*cmdText << ( byOrTo == CmdRotate::BY ? " by " :  " to ");
*cmdText << deg;
*cmdText << ends;
```

3. Provide a version of the protected virtual function **int Command::**`do_execute()`, which is called when the **Command** is requested to perform the actions it must do. More completely, to execute a **Command** the routine **int Command::**`execute()` is called, which then calls `do_execute`.

Since the **Command** will contain a text version of the requested action, it is relatively simple to create a text log of a VMD session: each time a **Command** is executed, the string for that command is simply written to a file.

There are many many actions which need to be done each time through the main execution loop of VMD (section 20). The **CommandQueue** object is used to queue and execute all the actions that need to be done. This is essentially a FIFO queue, and there is just one instance of this in VMD (stored in the global variable `commandQueue`). This object also contains routines for logging a VMD session. New **Command** objects are added to the queue via the routine

void CommandQueue::`append(`**Command * **`)`, and are appended to the end of the queue; the routine **void CommandQueue::**`execute( )` then executes the top **Command** in the queue, and then `deletes` the **Command** instance. After the **Command** is executed, but before it is deleted, **CommandQueue** informs the **UIObject**'s (described later) that the action was done (why this is so is also described later). Since the **Command** is deleted after it is executed, an instance of the **Command** must be created via `new`, and then left to **CommandQueue** to be deleted. This is done because due to the asynchronous nature of this method of executing commands, it is not known exactly when the data in the **Command** will be needed, and thus it is unknown when the storage space may be freed up for other use. The only object which knows this is **CommandQueue**, and so it must be given new copies of these **Command** objects which it must delete when done.

The objects which create these **Command** objects are derived from the **UIObject** base class. This base class forms the heart of all the different types of UI components which VMD provides. For example, the text console UI (**UIText**), the mouse interface (**Mouse**), and all the GUI forms (**FormsObj** and derivations thereof) are derived from **UIObject**. All the **UIObjects**, when they are initialized, register with a **CommandQueue** object, which maintains the list of all **UIObject**s and can work with them as a group. The **UIObject** is given a unique ID code when it registers, which it uses to identify later if any actions being done were a result of a request from itself.

Each **UIObject** basically works with a subset of all the possible **Command** objects which VMD contains. Typically a UI component displays some graphical feedback or status of the current state of the program, such as displaying via a slider or lighted button what the value of some variable is. When an action is performed the UI components must be informed because this graphical status must be updated to reflect any changes. Any number of different UI components may require such an update, but since the number of **Command**s which can result in a change to the particular graphical display of each **UIObject** is much smaller than the total number of available actions, it would be very inefficient to have every UI component notified when each action is performed. Instead, the **UIObjects** each maintain a list of the integer codes for the **Command**s in which they are interested. When a **Command** is executed, the **Commandqueue** notifies only those **UIObject**s which have indicated they are interested in the **Command**. However, a **UIObject** can create any available **Command** instance, and give it to the **CommandQueue** to be executed. When a new **Command** is created, the ID of the UI which is creating it is also given to the **Command**, so that later when the UI components are notified of the action they can tell who requested the activity.

The purpose of each **UIObject** is to provide a means for the user to input commands, and to display to the user the current status of the program. The virtual routine **int UIObject::**`check_event()` is called once for each UI during the main execution loop to allow the UI component to check for user events (such as keyboard entries, mouse button presses, or manipulations of GUI components such as buttons or sliders). If such an event is found, a new **Command** is created for the event (events are simply derived from **Command**, and contain the data specifying the type of event) and put on the **CommandQueue**. After all **UIObjects** are checked for events, the **CommandQueue** is told to start executing its queued actions, continuing until the queue is empty. When an event action is processed, typically it results in some other form of **Command** to be requested, which is done by creating the proper special derivationof **Command** for the action and giving it to the **CommandQueue**. Eventually all events are processed, and the actions requested by them are then processed, and finally the queue is empty. As each **Command** is processed the requested action is done and all the **UIObjects** which expressed an interest in the action are notified, which allows them to update their display. When the queue is empty, VMD proceeds to then redraw the

Scene. This execution loop is summarized in section 20.

It is relatively simple to create a new **UIObject**; each on-screen menu is a separate **UIObject** as is the mouse, the text console (which almost never needs to be updated due to a command being executed), and the 3D UI. Each **UIObject** can contain the ability to execute as many or as few actions as is desired. New **UIObject**s should be `new`'ed in the `VMDinitUI` routine, after the and **CommandQueue** global instance are created.

19.6 Tracker objects

The objects responsible for controlling the external spatial tracking devices, and for displaying and using the 3D pointers, are currently in the experimental stage, and will be described later when their design is closer to being final.

20 Main VMD execution loop

After the program starts up and initializes all global objects as well as processing any startup data files, an execution loop is entered which continues until the user requests to quit VMD. This loop is mainly contained in the routine `VMDupdate` (in the file `Global.C`), which is called continuously by the `main` routine of VMD. The algorithm for this loop is:

1. The VMDupdate routine requests each **UIObject** to check for any events such as mouse button presses or GUI button/slider/browser/whatever presses. These events are entered into the **CommandQueue** objects queue as **Command** object instances.

2. All commands in the **CommandQueue** are processed until the queue is empty. Typically, the processing of an event will result in one or more new **Command** object being created and placed at the end of the **CommandQueue**. The **CommandQueue** will just continue processing each command until the queue is complete empty, which will deal with all the events found in the previous step, as well as any other commands generated while processing the events.

3. Each **UIObject** is told to do any update that needs to be done each cycle through the execution loop. This accomplishes such things as updating the current frame in the form used for molecule trajectory animation.

4. The current **Scene** is rendered to the current **DisplayDevice**.

21 External library descriptions

There are several libraries used by VMD for the different optional components. These libraries are either located in the `lib` subdirectory of the VMD working directory, or are located in standard system directories. This section describes the purpose, usage, and location of these libraries. Note that in the following, the list item headers contain the library name and associated header files in parentheses; also, the word `ARCH` refers to the name of the Unix version for which VMD is to be compiled (i.e., IRIX5 or HPUX9). For almost all cases, the listed libraries are used when a specific configuration option is used to add an optional component into VMD (e.g. the `FORMS` or the `TCL` options).

- `libgl_s (gl/gl.h, gl/device.h, gl/sphere.h)`
  Configuration option: `OpenGL`.
  Expected library location: system directory, typically `/usr/lib`.
  Expected header file location: system directory, typically `/usr/include`.
  Purpose: this is the graphics library developed by Silicon Graphics. It is used when VMD is being compiled for a workstation with hardware accelerated graphics. This library is needed by all the **FormsObj** base- and sub-classes, as well as all classes which begin with the word `OpenGL` or `Cave` (i.e., OpenGLRenderer, CaveDisplayDevice).

- `libforms (forms.h)`
  Configuration option: `FORMS`.
  Expected library location: `lib/forms/lib_ARCH`.
  Expected header file location: `lib/forms/include`.
  Purpose: this is the FORMS library from Mark Overmars; it provides an easy to use set of buttons, sliders, dials, etc. for use as components of a graphical user interface.

- `libtcl (tcl.h)`
  Configuration option: `TCL`.
  Expected library location: must be already installed on your system.
  Expected header file locaton: must be already installed as well.
  Purpose: This library is used to parse and interpret text command scripts, including the capability to have variable substitution, control loops, procedure and function definitions, etc. This library is used by the **UIText** object.

- `libcave (cave.h)`
  Configuration option: `CAVE`.
  Expected library location: `lib/cave/lib_ARCH`.
  Expected header file location: `lib/cave/include`.
  Purpose: this is the CAVE library, used to display images in the CAVE virtual environment. It is used by all objects and files with the word `Cave` in their names, i.e., **CaveDisplayDevice**.

- `libtracker, libquat (tracker.h)`
  Configuration option: `UNC`.
  Expected library location: `lib/unc/lib_ARCH`.
  Expected header file location: `lib/unc/include`.
  Purpose: used by the **UNCTracker** object to access an external spatial tracking device connected to the graphics workstation running VMD. It uses a configuration file `.tracker` to simplify configuation of these spatial trackers, and allows VMD to use a variety of different devices without recompilation. It has been developed at the University of North Carolina, Chapel Hill.

Function and Class Descriptions

This chapter provides descriptions of most of the major objects and utility functions in VMD. Since VMD uses an object-oriented design, the descriptions here offer a good detailed look at how the program components are interfaced together. Not all objects used in VMD are documented here, however. Only widely-used classes, as well as important base classes, are listed. For each category of objects, a table lists the names of all the objects in that category, and indicates which ones are documented in this chapter.

The first part of this chapter (section 22) describes important utility functions used throughout the program. The remaining sections describe the important classes used in each category of objects in VMD.

22 Utility functions

Generally useful utility functions are located in the file `utilities.C`, with an associated header file `utilities.h`. These functions include:

- `char *stringdup(char *)`
  Creates a duplicate of the given string.

- `char *stringtoupper(char *)`
  Converts the given string to upper case; returns pointer to the same string.

- `int strupcmp(char *a, char *b)`
  Compares string `a` to string `b` in a case-insensitive manner. Otherwise acts like a call to `strcmp`.

- `int strupncmp(char *a, char *b, int n)`
  Like `strupcmp`, but only considers the first `n` characters.

- `void breakup_filename(char *s, char **p, char **n`
  Takes the filename in string `s` and breaks it into the file name `n` and directory `d`. Space is allocated for both the name and path strings.

- `char *command_tokenize(char *s, int *argc, char **argv)`
  Breaks the *command* string `s` into tokens, provides as pointers to the different token strings in `argv` (`argv` should be an array of pointers to `char` with enough items to hold a pointer for each token in the given string). `argc` is set to the total number of tokens found. Tokens are only searched for in the string until a `#` character is reached; the rest of the string `s` is then skipped if this comment character is found. If the first non-whitespace character is a `#` symbol, the routine returns `NULL`. A *command* string is a string of the form

  ```
  <keyword> = <string>
  ```

  If the equal sign is not present as the second token in the string, this routine returns `NULL`.

- `char *str_tokenize(char *s, int *argc, char **argv)`
  Similar to `command_tokenize`, without the restriction of having the string in the format of a *command* string, instead `s` may be any string.

- `double time_of_day(void)`
  Returns the elapsed time, in seconds, from some reference time (which *may* vary between different versions of Unix). Best used for calculating time differences, not for finding the "current" time.

- `char *username(void)`
  Returns a pointer to a string with the username of the person running the program.

- `int str2bool(char *s)`
  Returns 1 if the string `s` is any type of string indicating "yes" in some form, or 0 if the string indicates "no" in some form. If the routine cannot determine if yes or no was indicates, this returns -1.

- `float *cross_prod(float *x1, float *x2, float *x3)`
  Calculates $\vec{x1} = \vec{x2} \times \vec{x3}$, and returns a pointer to `x1`. The array `x1` must be different than the storage used for `x2` and `x3`.

- `float dot_prod(float *x1, float *x2)`
  Returns the value of $\vec{x1} \cdot \vec{x2}$.

- `float *normalize(float *x)`
  Normalizes the vector $\vec{x}$; returns a pointer to `x`.

- `float norm(float *x)`
  Returns the norm (length) n of a vector $\vec{x}$: $n = \sqrt{\vec{x} \cdot \vec{x}}$.

- `void add(float *x1, float *x2, float *x3)`
  Computes quickly $\vec{x1} = \vec{x2} + \vec{x3}$.

- `void subtract(float *x1, float *x2, float *x3)`
  Computes quickly $\vec{x1} = \vec{x2} - \vec{x3}$.

23 Utility objects

The following objects are useful utility objects used throughout the program. Items which have a specific section listed for them are explained in detail in that section.

Class Name	Section	Files
CaveResizeArray	-	`CaveResizeArray.h and .c`
ConfigList	-	`ConfigList.h and .C`
Grid	-	`Grid.h and .c`
Inform	23.1	`Inform.h and .C`
NameList	23.2	`NameList.h and .c`
ParseTree	-	`ParseTree.h and .C`
ResizeArray	23.3	`ResizeArray.h and .c`
Stack	23.4	`Stack.h and .c`
SymbolTable	-	`SymbolTable.h and .C`
Tokenize	-	`Tokenize.h and .C`

Table 2: VMD utility objects.

23.1 Inform

Files:	`Inform.h, Inform.c`
Derived from:	none
Global instance (if any):	msgInfo, msgWarn, msgErr, msgDebug
Used in optional component:	Part of main VMD code

Description

A streams-like utility object used to format messages for display at the console or other selectable `ostream` object. Messages are given to an **Inform** object via insertion operators (`<<`), and the messages are printed out to a provided `ostream` device, prefixed by a certain string. A special manipulator 'sendmsg' is used to signal when a mesage should be printed.

Each **Inform** object has a current 'message level', which is also indicated when the messages are printed out. Routines are provided to set the current level of the **Inform** object, so that only messages with a level smaller than or equal to the current output level are printed. Levels range from 1 (most likely to be printed) to 10 (very unlikely to be printed). The main use of this feature in VMD is in conjunction with debugging messages, where the level of debugging information can be changed via text commands.

There are four global instances of **Inform** which are available to the entire program (as long as they include the file `Inform.h`):

- `msgInfo` – Generally informative messages.
- `msgWarn` – Special warning messages, which do not signal an error but warn of some possible problem with the current state.
- `msgErr` – Error messages.
- `msgDebug` – Debugging messages. The macro `MSGDEBUG()` exists which should be used when printing messages to msgDebug; if the macro `VMDDEBUG` is set during compilation, the debugging messages will be included in the executable. An example of use:

```
MSGDEBUG("This is a debug message" << sendmsg)
```

Constructors

- **Inform::**`Inform`(char *name, int ison=1)
  The name argument is the name of the instance of this object; when messages are printed out they are profixed with the name, the message level, and a right paren, i.e.

  ```
  Info 1) Info message.
  ```

  The 'ison' argument determines if messages should be printed or not at the start. Besides the message levels, each Inform object can be on or off. When off, message are not printed, otherwise they are if the message level is set properly.

Enumerations, lists or character name arrays

Inform contains several *manipulators* used to determine how the message is printed or to do actions. They are:

- `sendmsg` – used to signal that the message is complete, and should be printed. So, this should be that last item inserted into the **Inform** stream.
- `level1 ... level10` – sets the output level of the current message to X, where X is the digit at the end of this name (1 ... 10).

Internal data structures

- `char *name` – name of the instance. Used to print message prefix.
- `ostrstream *msg` – character stream used to hold the message string to be printed.
- `ostream *msgDest` – destination stream, where the message will be printed. This could be the console (cout), or also a file.
- `int On` – whether the object should print out its messages.
- `int needNewline` – before the message is actually printed, sometimes a newline must be printed, and sometimes not. This flag tells whether to print a newline. Once a message is printed, this flag is cleared. The member function `need_newline` may be used to set this flag.
- `int outputLevel` – the current output message level of the instance. This level is compared with the level of the message currently being requested to be printed; if the level of the message is less than or equal to the current message level of the **Inform** object, then the message is printed, otherwise it is just ignored.
- `int msgLevel` – the current output level of the message that is to be printed. This may be changed anytime during the construction of the message; the level at the time when the request to print the message is made is the one used.

Nonvirtual member functions

- `void on(int)` – set the object to be on or off.
- `int on(void)` – return the current on/off status.
- `output_level(int)` – set the current output level.
- `int output_level(void)` – return current output level of the **Inform** object.
- `msg_level(int)` – set the current level of the next message to be printed.
- `int msg_level(void)` – return the current message output level.
- `void need_newline(int)` – set the current flag for whether a newline is needed or not.
- `int need_newline(void)` – return current newline print status.

- `void destination(ostream *)` – give the provided destination stream to the **Inform** object as a place where to print messages.
- `ostream *destination(void)` – return destination to where messages will be printed.
- `send(void)` – prints the contents of the current message buffer to the destination stream, if necessary.
- `operator<<(various types)` – overloaded insertion operators which are used to put data into the current message buffer. These are used just like regular stream insertion operators, and when the message is complete, either the routine `send` should be called, or else the `sendmsg` manipulator should be inserted into the stream. Newline characters may be put in the stream, **Inform** will print out each newline-separated line with the proper prefix.

Method of use

As stated, there are four global instances of **Inform** which should be used for printing all messages in VMD. They are used just like a stream, except the manipulator `sendmsg` is used instead of `endl` to signal that the string should be printed. An example:

```
msgWarn << level3 << "This is a level-3 warning." << sendmsg;
```

In this case, if `msgWarn` is on, and it's output level is greater than or equal to 3, then this message will be printed, otherwise it will be ignored.

For debugging messages (printed to `msgDebug`), make sure to use the `MSGDEBUG` macro, so that these debugging messages can be conditionally excluded from the executable.

Suggestions for future changes/additions

Right now, only the debug **Inform** object takes advantage of the message level facility, all other messages (information, warning, or error) are all level-1 messages. More messages should be printed out by VMD, but at different levels of verbosity and content, and text commands should be put in to control the current output level of `msgInfo`, `msgWarn`, and `msgErr`.

Text commands should be added to allow the message text to be sent to a file, or to the console (right now, they are always sent to the console). This would not be very difficult, a new text command would need to be added which would open a file, and call the `destination` routine with the respective `ostream` pointer.

23.2 NameList

Files:	`NameList.h, NameList.c`
Derived from:	none
Global instance (if any):	none
Used in optional component:	Part of main VMD code

Description

This is a template class which implements a simple associative array. A **NameList** consists of N items, where each item has a name, an index, and an associated data value (the type of which is determined by the type given to the template).

Constructors

- **NameList::**`NameList`(void)

Internal data structures

- `ResizeArray<char *> names` – the array of names, each of which has a data value associated with it.
- `ResizeArray<T> Data` – the data associated with each name. The size of this array is the same as the size of `names`.
- `ResizeArray<int> sortedOrder` – an array of integers which indicates the sorted order for the `names` array. It is the same length as `names` and `Data`.
- `int Num` – the number of items in these lists.

Nonvirtual member functions

- `int num(void)` – return number of items in the **NameList**.
- `int add_name(char *, const T&)` – adds a new name to the list, with the given data of type T associated with it. This returns the index (also called *typecode*) of the name in the list. If the name already exists in the list, this just returns to index of the existing name, and does not change the data stored.
- `char *name(int)` – return the name of the Nth item.
- `char *sort_name(int)` – return the name of the Nth sorted item, i.e. the Nth highest name when they are put in sorted order.
- `int typecode(char *, int = -1)` – return the index for the given name. If the second argument is greater than 0, it is used as the max length of the names to check for a match. If it is negative, an exact match must be found. If no match is found, this returns -1.
- `int sort_typecode(int)` – return the actual index of the Nth sorted item.
- `T data(char *, int = -1)` – return a copy of the data for the given name, in the same way as for `typecode(char *, int)`.

- `T data(int)` – return the data for the Nth item.
- `void set_data(int, const T&)` – changes the data value associated with the Nth name to the given value.

Method of use

This is a template used to associate sets of data of any type with character strings (names). Names and data are added with `add_name`, and the other routines are used to query info about these names. Many objects in VMD use **NameList**s, for example the colors are all stored this way.

23.3 ResizeArray

Files:	`ResizeArray.h, ResizeArray.c`
Derived from:	none
Global instance (if any):	none
Used in optional component:	Part of main VMD code

Description

A template class which implements a dynamically-growing, automatically resizing array of data of a given type. Elements in the array may be accessed via the [] operator. When new data is added to the end of an array, the size of the array is automatically increased if necessary.

A **ResizeArray** has two sizes that are important:

- The *internal* size, which is currently allocated size of the array. This is almost always larger than the total number of items stored in the array. A larger-than-necessary buffer is kept so that data may be easily put at the end of the array without having to always reallocate and copy the existing array storage.
- The *external* size, which is the largest index in the array which has been accessed by the user (either by reading from that index, or writing a value at that index. This is what the user would consider the 'size' of the array.

Constructors

- **ResizeArray::**`ResizeArray(`int = 10, float = 2.0`)`
  The first argument is the initial internal size of the array, i.e. the initial number of elements for which to allocate memory (although the initial external size of the array will be zero). The second argument is the *resizing factor*: when an element of the array beyond the size the internal array is accessed, the array will be made larger by multiplying the current internal size by the rescaling factor. This must be larger than 1.

Internal data structures

- `T *data` – the current array of data, which will be made larger as the array grows.
- `int sz` – the internal size of the array.
- `float resizeFactor` – factor by which the number of elements in `data` is increased when the array grows too large. sz will be made larger by `sz *= resizeFactor`.
- `int currSize` – the external size of the array, that is, the largest index into `data` which has been accessed. This is always less than or equal to `sz`.

Nonvirtual member functions

- `int num(void)` – return the external size of the array.
- `T& item(int N)` – return (as an lvalue) the Nth item in the array. If N is larger than the internal size of the array, the size of the array will be automatically increased by the current resize factor.

- `T& operator[](int N)` – same as `item`, just a nicer form (this really makes it look like a regular array. In fact, you can have 2D and 3D resizing arrays by using this operator, and declaring a template of a template).
- `int append(const T&)` – append the given data at the end of the array, resizing it if necessary. Returns the index of the item.
- `int insert_after(int, const T&)` – inserts the given data in the location following the given index. If the given index is less than 0, this puts the item at the beginning of the array. If the given index is larger than the external size, this puts the item at the end of the array. Returns the index of the item.
- `int insert_before(int, const T&)` – inserts the given data in the location just before the given index. If the given index is less than 0, this puts the item at the beginning of the array. If the given index is larger than the external size, this puts the item at the end of the array. Returns the index of the item.
- `void remove(int M= -1, int N= -1)` – removes the items with index M ... N, with all items in array positions below this region moved up to fill the resulting hole. If both arguments are negative, this removes ALL the array items. If the second argument is negative, only item M is removed.
- `int find(const T&)` – scans the array for an element which matches the given argument, and returns the index of the first matching item found. If no match is found, this returns -1.

Method of use

A **ResizeArray** is created by specifying the type of data to store, i.e.

```
ResizeArray<char *> stringArray;
```

Once created, it looks just like an array, and you can use the [] operator to get and set data. For example:

```
stringArray[3] = "Some text.";
```

or

```
cout << stringArray[3] << endl;
```

23.4 Stack

Files:	`Stack.h, Stack.c`
Derived from:	none
Global instance (if any):	none
Used in optional component:	Part of main VMD code

Description

A template class which implements a simple stack of arbitrary data. Items are pushed onto and popped off of the stack, or may just be copied from the stack.

Constructors

- **Stack::**`Stack`(int s)
  The argument `s` is the maximum size of the stack.

Internal data structures

- `T *data` – array storing the actual data in the stack.
- `T *curr` – pointer to the current 'top' item in the stack.
- `int sz` – maximum number of items which the stack can hold.
- `int items` – current number of items on the stack.

Nonvirtual member functions

- `int stack_size(void)` – return number of items on the stack.
- `int num(void)` – same as `stack_size()`.
- `push(const T&)` – copies the given item onto the top of the stack, moving all other items down one.
- `push(void)` – moves all items down one, keeping a copy of the previous top item on the top of the stack.
- `T& pop(void)` – returns the top item on the stack, while also removing it and moving all the other items up one.
- `t& top(void)` – returns a reference to the top of the stack, but does not change anything.

Method of use

You instantiate a stack by telling it how large it can grow, and then just push and pop data on/off the stack. If the stack gets too large or small, warning messagea are printed.

24 Display objects

The following objects are used mainly for the purpose of displaying images to the screen. Items which have a specific section listed for them are explained in detail in that section.

Class Name	Section	Files
Axes	-	`Axes.h and .C`
CaveDisplayDevice	-	`CaveDisplayDevice.h`
CaveScene	-	`CaveScene.h and .C`
ColorList	24.1	`ColorList.h and .C`
ColorUser	24.2	`ColorUser.h and .C`
DispCmd (many)	24.3	`DispCmds.h and .C`
DisplayDevice	24.4	`DisplayDevice.h and .C`
Displayable	24.5	`Displayable.h and .C`
FileRenderList	-	`FileRenderList.h and .C`
FileRenderer	-	`FileRenderer.h and .C`
GLDisplayDevice	-	`GLDisplayDevice.h and .C`
GLRenderer	-	`GLRenderer.h and .C`
Light	-	`Light.h and .C`
LightList	-	`LightList.h and .C`
Matrix4	24.6	`Matrix4.h and .C`
NormalScene	-	`NormalScene.h and .C`
POVDisplayDevice	-	`POVDisplayDevice.h and .C`
PickList	-	`PickList.h and .C`
PickMode	-	`PickMode.h and .C`
PickModeDrag	-	`PickModeDrag.h and .C`
PickModeMolLabel	-	`PickModeMolLabel.h and .C`
PickModeQuery	-	`PickModeQuery.h and .C`
Pickable	-	`Pickable.h and .C`
R3dDisplayDevice	-	`R3dDisplayDevice.h and .C`
RayShadeDisplayDevice	-	`RayShadeDisplayDevice.h and .C`
Scene	24.7	`Scene.h and .C`
Stage	-	`Stage.h and .C`
VMDTitle	-	`VMDTitle.h and .C`

Table 3: VMD display objects.

24.1 ColorList

Files:	`ColorList.h, ColorList.C`
Derived from:	**Displayable3D**
Global instance (if any):	`colors`
Used in optional component:	Part of main VMD code

Description

Contains a list of all the items which have an assigned color, and the list of all the different color *categories* which exist. VMD uses a *colormap* made of 16 base colors (plus black), and a *colorscale* made of 1024 colors in user-definable gradient (e.g., red-green-blue, or red-white-blue). Color categories are used to contain the names of items which are related and which may have their colors changed; an example of a color category is 'Axes', which contains the colors for the 'X', 'Y', and 'Z' axes. The categories contain just a list of names and the associated colors assigned to these names (an integer, from 0 ... 16). Other objects in VMD which are derived from **ColorUser** can add new color categories (or can query if one of a specific name already exists), and can add new names to an existing color category. The colors assigned to the named items in each category can be changed via the Color menu, or through the `color` text command. Once a category is created by some **ColorUser** object (or, if it already exists, has been obtained by the object from **ColorList**), the **ColorUser** can use the category to retrieve the proper color to use for whatever purpose it needs it (i.e., creating the object's display list if it is a **Displayable**).

Constructors

- **ColorList::**`ColorList(`Scene *`)`

Enumerations, lists or character name arrays

There are three different ways to set the colorscale:

- `RGB`: red – green – blue.
- `RWB`: red – white – blue.
- `BLK_W`: dark grey – light grey.

There are sixteen colors in the colormap, along with black, for seventeen total colors. They are: blue, red, grey, orange, yellow, tan, silver, green, white pink, cyan, purple, lime, mauve, ochre, iceblue, and black.

Internal data structures

- `NameList<int> colorNames` – a list of all the different colors, and their corresponding colormap index.
- `float colorData[MAXCOLORS][COLOR_ITEMS]` – the data which define each color, including all the colormap colors and the colorscale colors.

- `NameList<NameListIntPtr> categories` – a list of all the color categories. Each category has a name, and an associated list of named items (so, first the right category is found by looking up a name in `categories`, and then the right color is found by looking up a name in the list associated with the given category name).

Method of use

VMD contains one global instance of this class, `colors`. It is used by all objects derived from **ColorUser**, which includes all **Displayable** objects as well as **DisplayDevice**. An object which wishes to allow the user to be able to change its colors goes through these steps:

1. call **int ColorList::**`add_color_category`(char *) ; this adds a new category with the given name and returns a unique index for that category, or else returns the index of the category if it already exists. This index should be saved for later use.

2. once a category exists, names should be added to the category that correspond to all the components that can be configured, with default colors. This is done by doing the following:

   ```
   (colors->color_category(catIndex))->add_name("item name", defaultColor)
   ```

3. When the colors have been added, they may be retrieved later when creating a display list by doing the following:

   ```
   (colors->color_category(catIndex))->data("item name")
   ```

   This will return the index of the color assigned to the given item name.

24.2 ColorUser

Files:	`ColorUser.h, ColorUser.C`
Derived from:	none
Global instance (if any):	none
Used in optional component:	Part of main VMD code

Description

Simple base class for objects which will be needing to use a ColorList for colors. This class provides routines for being given a ColorList object, and for being informed of when colors are changed. The user can create new *categories* of colors in the ColorList, and add specific *color objects* to the category with an identifying name. The user can edit these colors via commands.

Constructors

- **ColorUser::**`ColorUser`(void)

Internal data structures

- `ColorList *colorList` – a pointer to a ColorList object for this object to use. The **ColorList** is the object which keeps the global lists of color categories.

Virtual member functions

- `virtual void use_colors(ColorList *)` – this function is used to set up this object to use the specified **ColorList** to store and retrieve data. By default, it will save the pointer and then call `do_use_colors`.

- `virtual void color_changed(ColorList *collist, int c)` – after a color has been changed via the GUI or a text command, the **ColorUser** must be notified of the change so that it can update any internal state. This routine is called when color `c` in the specified **ColorList** is changed. By default, this just calls the virtual function `do_color_changed`.

- `virtual void do_use_colors(void)` – this is called after a new **ColorList** object is provided. It is a protected function, and by default does nothing. It should only be defined if the derived class needs to do something special when a new **ColorList** is given, such as create a new color category.

- `virtual void do_color_changed(ColorList *collist, int c)` – this is called after a color in the given category is changed. It is a protected function, and by default does nothing. It should only be defined if the derived class needs to do something special when a color is changed, such as rebuild a display list or recalculate some data.

Method of use

ColorUser is a base class designed to be specified as a parent for some class which desires to use a **ColorList** to maintain categories of colors associated with names, and to make these names be customizeable to the user via the VMD user interfaces. All **Displayable** objects are derived from **ColorUser**, as are all **DisplayDevice** objects.

ColorList describes how to create new categories and how to access them. If a new class wishes to use these features, it should derive from **ColorUser**, and define versions of the protected virtual functions `do_use_colors` and `do_color_changed`. This first is designed to be only called once, right after the object is created – it should create new categories, and add the necessary names to these categories. The second is designed to be called every time a particular color changes. The `do_color_changed` should check to see which color was modified, and then update itself if necessary.

24.3 DispCmd

Files:	`DispCmds.h, DispCmd.C`
Derived from:	none
Global instance (if any):	none
Used in optional component:	Part of main VMD code

Description

DispCmd is a base class for a large number of objects which are used to create the *display lists* in **Displayable** objects. Each subclass represents a particular drawing primitive, such as plotting a line, changing the current color, or applying a rotation. These objects take the data required for the particular primitive they represent (for example, the endpoints of a line, or the index of a color) and append this data to the end of the display list for a given **Displayable**. Routines are available to permit a **DispCmd** to be used more than once, and also to have the **DispCmd** remember the location where the data was most recently appended so that new data may be copied over existing data in a display list.

Constructors

- **DispCmd::**`DispCmd`(int code, int size)
  `code` is a special index used to distinguish what type of **DispCmd** this is; it is supplied by the derived class, and should be one of the enumeration values listed at the top of the file `DispCmds.h`. `size` is the number of bytes that the data will occupy when placed into the display list; it is also supplied by the derived class. It can be zero.

Enumerations, lists or character name arrays

At the beginning of `DispCmds.h` is an enumeration with entries for all the different **DispCmd**-derived classes. These names are used by the `render` routine in each **DisplayDevice** to determine what the commands are in a display list. When new drawing primitives are added, a new name should be put in this list, and that name should be specified in the **DispCmd** base class constructor.

Following this are enumerations describing the different types of spheres that may be drawn, and the different types of lines.

Internal data structures

- `int cmdCode` – the unique integer ID of the command; this is one of the names in the main enumeration described above.
- `int cmdSize` – the number of bytes which the data will occupy when copied into a given display list.
- `void *Data` – a block of `cmdSize` bytes which stores the data to be copied into a given display list.
- `void *dataLoc` – a pointer to the last place into which the bytes in `Data` were copied. If it is NULL, the data has not yet been copied. This is stored so that the `reput` command can copy new data over previously-stored data, instead of just appending the data to the end of a display list.

Nonvirtual member functions

- `void put(Displayable *dobj)` – This is the main function in each **DispCmd** object. It copies the values stored in `Data` to the end of the specified **Displayable** object's display list. As well, the integer code indicating which primitive this is is copied into the list, and also the size of the data (in bytes). The order of copying data is as follows:

  1. `cmdCode` (one integer)
  2. `cmdSize` (one integer)
  3. `Data` (`cmdSize` bytes)

- `reput(Displayable *dobj)` – Same as `put`, except that if the same **DispCmd** has had the `put` routine called, this command will copy the data into the same location as was used previously. This can be used to copy data over previously-copied data. A particular example is in **Displayable**, which uses `reput` to replace the matrix data in the multiply-transformation-matrix command used at the beginning of each **Displayable**'s display list. If `put` has NOT been called prior to calling `reput`, then `reput` does exactly the same thing as `put`.

Method of use

Each class derived from **DispCmd** should have two constructors:

- A *default* constructor, with no arguments - this will just initialize all the internal storage to default values.
- A data constructor, which takes arguments with the data necessary for the command. This will copy the arguments into the internal data storage of the object.

If a derived class does not require any arguments (for example, the PUSH command, which just signals to push the top transformation matrix, but does not require any data itself), only the default constructor is used. Otherwise, each derived class should also provide two new functions;

- `void putdata(`*args*`, Displayable *dobj)` – This should copy the data specifed in *args*, and then call `put(dobj)`. This is used when an instance of this class is created with a default constructor, and the instance is to be used more than once. Each time the instance it used, `putdata` is called to append new data to the end of $\widehat{d}$obj's display list.
- `void reputdata(`*args*`, Displayable *dobj)` – Same as `putdata`, but it should call `reput` after copying the given data instead of `put`.

Within the `put` and `reput` functions, **DispCmd** uses several routines in **Displayable** to start the action of writing a new command to the end of the display list, to copy over the data itself, and to signal the data is copied.

Suggestions for future changes/additions

At present, most **Displayable** objects contain instances of these commands as private members, which are used to create their internal display lists. This is not really necessary, it would probably be better if there were one global instance of each different **DispCmd**, available for use by the **Displayable** objects. This would decrease the memory used, and the time for construction of the **Displayable** using the global instances instead of a local copy.

24.4 DisplayDevice

Files:	`DisplayDevice.h, DisplayDevice.C`
Derived from:	**ColorUser**
Global instance (if any):	`display`
Used in optional component:	Part of main VMD code

Description

DisplayDevice is one of the main classes in VMD. It is a base class for all the objects which are responsible for drawing the current scene, either to an on-screen graphics window, or to a file. It is also responsible for encapsulating the window-manager-specific routines which are different for each type of windowing environment (e.g., X-Windows-based window management and OpenGL rendering, etc.). Finally, it is used to determine which item a pointer is currently selecting when a pointer button is pressed, either by a 2D mouse or by a 3D pointer. There are a large number of virtual functions which each subclass may define (if relevant). The **DisplayDevice** base class provides default versions for all the virtual functions, however, so that by default a bare **DisplayDevice** acts as an "empty" device, which essentially does nothing but which still functions to the rest of the program as a working **DisplayDevice**.

The main **DisplayDevice** instance used by VMD is also responsible for checking for and reporting window events, such as when the window is resized, or when a mouse button is pressed, etc. The main graphics window has one main *popup menu* (which could be implemented as a pulldown menu if required), which the **DisplayDevice** must be able to create and acquire events from.

There are two types of subclasses of **DisplayDevice** used in VMD:

- On-screen rendering devices, which manage a graphics display window on the workstation monitor, and which define new versions for basically all virtual functions in **DisplayDevice**.

- Image-file rendering devices, which are used to render the current scene to an image file, or to create an input script for some post-rendering processor (such as a ray-tracing program or the *Raster3D* application). These subclasses are designed to be used to create a new file each time the *render* routine is called. Since file-rendering **DisplayDevice**'s do not have to manage a window, deal with picking objects with a pointer, etc., they are much simpler and do not define new version of many of the virtual functions. For file rendering devices, many of the internal variables, enumerations, and functions are irrelevant and may be ignored when new file rendering subclasses are written.

Constructors

- **DisplayDevice::**`DisplayDevice(char *)`
  The argument given is the name of the **DisplayDevice**.

Enumerations, lists or character name arrays

- `DisplayEye` lists the different locations for the viewers eye: NOSTEREO if the display is not in stereo, LEFTEYE or RIGHTEYE if the view from either eye is being used.

- `Buttons` lists the different input buttons which are assumed to be available. B_LEFT, B_MIDDLE, and B_RIGHT are for the three buttons on a mouse, while B_F1 through B_F12 are for 12 function keys, and B_ESC is for the 'esc' key of the keyboard.
- `EventCodes` lists the different types of window events which are assume to be possible. These are all prefixed with 'WIN_', and include redraw, mouse and keyboard button events, as well as events for when mouse focus changes or the window is exposed.

Internal data structures

- `int colorCat` – the index of the color category which holds the 'Background' color item. **DisplayDevice** is derived from **ColorUser**, and creates a new category 'Display' which holds the names used for coloring items in **DisplayDevice**. The background color is accessed through this category, when the display is cleared.
- `long xOrig, yOrig, xSize, ySize` – the position of the lower-left corner of the graphics window, and the size of the window, in pixels.
- `float backColor[3]` – the color of the background, as should be used the next time the display is cleared. When the background color is changed via user commands, the virtual function `do_color_changed` gets called and the value of `backColor` gets updated.
- `int Dim` – the current dimension of the display, 2 or 3. Note that presently 2D drawing still needs work.
- `Stack<Matrix4> transMat` – the transformation matrix stack for this display. The top matrix is the matrix used to transform 'world' coordinates to pre-projection coordinates. It can be pushed and popped with the `push()` and `pop()` virtual routines in **DisplayDevice**. Some devices may wish to use some other matrix stack, for example the OpenGL matrix stack.
- `int lineStyle, lineWidth, sphereRes, sphereMode` – the current settings for these drawing characteristics. They may be modified through **DisplayDevice** calls, but are most often changed via drawing command in a display list created by a **Displayable**.
- `float eyePos[3], nearClip, farClip, vSize, zDist, Aspect, cpUp, cpDown, cpLeft, cpRight` – these describe the current viewing geometry, by specifying where the viewer's eye is located, how far from the eye to the near and far clipping planes, and how large the screen is vertically and how far it is from the origin. When these values have been given, the routine `calc_frustrum()` is used to calculate the current aspect ratio, and the location of the corners of the viewing frustrum (the pyramid-shaped view formed from the eye (located at the tip of the pyramid) to the screen (located at the base of the pyramid).
- `int inStereo, stereoModes` – the current stereo mode; 0 always means the display is not in stereo, while values greater than 0 indicate one of the display's available mode. `stereoModes` indicates how many different modes are available. Each **DisplayDevice** must set this to indicate if it has stereo capabilities.
- `char **stereoNames` – a pointer to an array of names used to indicate the different stereo modes available. There should be `stereoModes` number of names in this list. Note that 'stereo off' counts as one mode, and is the default. So, if a new **DisplayDevice** cannot display stereo, it should not change this variable, or `stereoModes`.

- `float eyeSep, eyeDist, eyeDir[3], upDir[3], eyeSepDir[3]` – these describe the current stereo viewing geometry. Along with the eye position, these define vectors describing where the eye is looking, where the 'up' direction is, and a vector along the line formed between the two eyes. These also describe how far apart the eyes are, and how far from the current eye position to the viewing focal point.
- `int lightDefined[], lightOn[]` – flags indicating whether data for the light sources has been provided (via a drawing command), and whether the light sources are currently on or off.
- `float lightColor[][3], lightPos[][3]` – the RGB color, and XYZ position, of each light source.
- `int matDefined[], materialsActive, materialOn` – flags indicating whether data for material characteristics has been provided (via a drawing command), whether material characteristics should be used or not when drawing polygons, and the current material index that should be used to draw the next polygon (if `materialsActive` is true).
- `float matData[][]` – the color, emissitivity, specularity, transparency, etc. of each material that has been defined.
- `pickRegion` – how large an area to use as the search region when a pointer is attempting to select an item. When picking items, the graphics window is assumed to be in 'relative scaled coordinates', that is, it is assumed to have coordinates 0 ... 1 in both the X and Y directions, with the origin in the lower left corner. Having a `pickRegion == 1.0` means the entire screen. A typical value for this is 0.01.

Virtual member functions

- `virtual void queue_events(void)` – Indicate to the event manager that windowing and mouse movement events need to be reported. The actual commands executed through this function will be different based on the window manager used, and the GUI library being used (basically, it depends on who is handling events, which is different when just using straight GL, when using the FORMS library, when using just Xlib, when using ... ack).
- `virtual int test_events(void)` – Check and see if there is an event pending, but do not get the event yet.
- `virtual int read_event(long &, long &)` – Check for an event; return true if one was found, and return the code and value for the event in the arguments.
- `virtual int x()` – The current X position of the mouse, measured in pixels from the lower-left corner of the screen.
- `virtual int y()` – The current Y position of the mouse, measured in pixels from the lower-left corner of the screen.
- `virtual int button_down(int)` – Check if the specified button is currently being pressed.
- `virtual void set_stereo_mode(int = 0)` – Changes to a new stereo mode.

- `virtual void abs_screen_loc_3D(float *, long *)` – Given a point in 3D 'world' coordinates (the first arg), this routine converts the point to absolute 2D 'screen' coordinates, i.e. the location measured in pixels from the lower-left corner of the screen.

- `virtual void rel_screen_loc_3D(float *, float *)` – Given a point in 3D 'world' coordinates (the first arg), this routine converts the point to 'relative, scaled' 2D coordinates, which are 0 ... 1 inside the graphics display window in each X, Y dimension, and undefined outside this region.

- `virtual void find_3D_from_2D(float *A, float *B, float *C)` – Given a point A, and a 2D relative screen position point B, this computes the 3D point corresponding to the position of the 2D point. Since the 2D point is not sufficient to determine the 3D position in space for that point, the point A is given to serve as a 'reference' point. Currently, the algorithm only supports the simple case where the eye is looking directly along the Z axis.

- `virtual void push()` – pushes the top matrix on the transformation matrix stack; the top matrix is unchanged, but is saved one level down on the stack.

- `virtual void pop()` – pops the matrix stack, restoring a previously pushed matrix state.

- `virtual void loadmatrix(Matrix4 &)` – copies the given 4 x 4 matrix into the top matrix on the stack, destroying the previous matrix value there.

- `virtual void multmatrix(Matrix4 &)` – premultiplies the top matrix on the stack by the given 4 x 4 matrix.

- `virtual void prepare2D(int = true)` – get the display ready to draw 2D objects, by setting the proper projection matrices and viewport. The argument indicates whether to clear the screen.

- `virtual void prepare3D(int = true)` – get the display ready to draw 3D objects, by setting the proper projection matrices and viewport. The argument indicates whether to clear the screen.

- `virtual void clear(void)` – erase the current display window, setting the background to the proper color.

- `virtual void left(void)` – prepare to draw the left eye image when in stereo. Note that when drawing in stereo, the left eye should always be drawn first.

- `virtual void right(voivd)` – prepare to draw the right eye image when in stereo. Note that when drawing in stereo, the right eye should always be drawn last.

- `virtual void normal(void)` – prepare to draw a non-stereo image.

- `virtual void update(int = true)` – after drawing is complete, this routine 'cleans up', by doing any actions which only need to be done once at the end of drawing (for example, swapping buffers which drawing using double buffers). The argument indicates whether to actually perform a buffer swap.

- `virtual void reshape(void)` – refresh the display after it has been reshaped or exposed.

- `virtual void render(void *)` – the most important routine in **DisplayDevice**: this routine takes the given display list (created by some **Displayable**, and goes through the list executing the drawing commands as they are listed. Each **DisplayDevice** subclass must provide a version of `render`, to do the actions required to draw a scene.
- `virtual int pick(int, float *, void *, float &)` – pick objects based on given list of draw commands, and determine which (if any) item in the list of drawing command was under the given pointer position. The arguments are the dimension of picking (2 or 3), the position of the pointer, draw command list, and returned distance from object to eye position (this last argument is set to the distance from the object to the current eye position, which can be used to determine which item is closest when multiple objects are under the pointer). This function returns the ID code ('tag') for the item closest to the pointer, or (-1) if nothing was picked. If an object is picked, the eye distance argument is set to the distance from the display's eye position to the object (after its position has been found from the transformation matrix). If the value of the argument when `pick` is called is negative or zero, a pick will be generated if any item is near the pointer. If the value of the argument is positive, a pick will be generated only if an item is closer to the eye position than the value of the argument. For 2D picking, the pointer coordinates are the relative position in the window from the lower-left corner (both in the range 0 ... 1). For 3D picking, the pointer coordinates are the world coordinates of the pointer. They are the coordinates of the pointer after its transformation matrix has been applied, and these coordinates are compared to the coordinates of the objects when their transformation matrices are applied.

Method of use

Once a **DisplayDevice** has been defined, and an instance created, a simple sequence of routines are used to have the device render a scene. This sequence is implemented by the **void Scene::**`draw`(DisplayDevice *display) routine. A **Scene** object maintains a list of display lists and **Displayable** objects which create those lists (see section 24.7). The `draw` routine uses the **DisplayDevice** as follows:

1. `display->prepare3D()`
2. Set the stereo mode to the left eye, or just normal, using the command `display->left()` or `display->normal()`
3. For each **Displayable** in the **Scene**, the `prepare` routine is called to make that object ready to be drawn.
4. For each display list maintained by the **Scene**, the command `display->render(displist)` is called.
5. If the display is in stereo, `display->right()` is called and the previous step repeated.
6. `display->update()`

Drawing to a file instead of the screen is almost identical, with the following exceptions:

- At present, only non-stereo file rendering is supported.
- The routine **int Scene::**`filedraw`(char *method, char *filename, DisplayDevice *display) is used instead of `draw`, since a specific filetype and filename must also be given.

Suggestions for future changes/additions

Most of the intended future improvements previously listed here have already been completed.

24.5 Displayable

Files:	`Displayable.h, Displayable.C`
Derived from:	**ColorUser**, **Pickable**
Global instance (if any):	none
Used in optional component:	Part of main VMD code

Description

The **Displayable** base class is the parent class for all items which need to be drawn to a **DisplayDevice**. For example, the set of axes which appear in the corner of the VMD display screen are maintained as a **Displayable**, and each **Molecule** is one as well. The **Scene** (section 24.7) object maintains a list of all the **Displayable**'s that are to be drawn to the screen. Each **Displayable** consists of the following components:

- A *transformation matrix* which describes how the graphical object represented by the **Displayable** is transformed (via translations, rotations, and scaling).
- A *display list* which contains a set of *drawing tokens*, in a device-independent format. The display list is used by a **DisplayDevice** to actually draw the object.
- A *parent* **Displayable**, which acts as an 'owner'; transformations applied to the parent are also applied to its children. If an object does not have a parent, it is a *top-level* **Displayable**.
- A set of *child* **Displayable** objects.

The **Displayable** class is the fundamental building block for all items which want to draw something to the screen. Consequently, it is a rather complicated beast.

Constructors

- **Displayable::**`Displayable(`TransMethod, char *, int, Scene *, int`)`
  `TransMethod` is an enumeration value indicating how the objects transformation matrix should affect the matrix of the **DisplayDevice** (either it should multiply the matrix (MULT), replace it (LOAD), or it should be ignored altogether (NONE)). The following arguments are the name of the object, the dimension (2 or 3), the **Scene** with which this object should register, and the initial size of the display list storage (in kilobytes).
- **Displayable::**`Displayable(`TransMethod, char *, int, Displayable *, int`)`
  This is the same as the first constructor, but instead of specifying a **Scene** to register with, a parent **Displayable** is given. The new **Displayable** will become a child of the given parent, instead of registering with a **Scene**.

Enumerations, lists or character name arrays

`TransMethod` is a public enumeration which is used to indicate the method of use of the objects transformation matrix. It is one of the following:

- `MULT`: multiply the **DisplayDevice**'s transformation matrix by this object's matrix when the object is drawn.

- LOAD: replace the **DisplayDevice**'s transformation matrix by this object's matrix when the object is drawn.
- NONE: do not use the object's transformation matrix when rendering.

Internal data structures

- Displayable *parent – each **Displayable** has a parent, which may be null. The parent is in charge of collecting *children* and displaying them as a single entity with separate display lists. Operations which are applied to the parent are also applied ot the children, such as rotations, commands to turn the item on or off, etc. Note that the reverse is NOT true: operations applied to children are not applied to their parent. Thus, if you rotate a parent, all the children will rotate in the same way, but it is possible to rotate each child separately.
- ResizeArray<Displayable *> children – the list of children for this object. This is quite often empty. But other **Displayable**'s have many children, for example **MoleculeList** has as its children all the **Molecule** objects.
- void *cmdListBeg – the start of the display list for this object
- void *cmdListPos – the location in the display list where the next drawing command should be placed
- Scene *origScene – each **Displayable** must register itself with a **Scene** object, and this item stores a pointer to that **Scene**
- ResizeArray<Scene *> sceneList – if the **Displayable** is registered with more than one **Scene**, they are stored here
- Matrix4 tm – the current transformation matrix. This is a public variable so it may be accessed quickly. The transformation matrix is actually just four different matrices multiplied together; these matrices are centm, rotm, globm, and scalem. The order of multiplication is as follows:

  $$\mathrm{tm} = \mathrm{globm} * \mathrm{rotm} * \mathrm{scalem} * \mathrm{centm}$$

  When a vector is multiplied by tm, the operations these matrices represent are:

  - centm – a centering translation. When tm multiplies a vector, this centering translation is applied first.
  - scalem – scales the vector. This is done after the centering translation, so the 'units' for the centering are in the same units as used for the 'world' coordinates.
  - rotm – rotates the vector. This follows the scaling operation (although the two may be done in either order).
  - globm – a 'global' translation, which is the last operation done to the vector. Since this is done *after* the scaling, the units are in post-scaled world coordinates.

  Displayable contains many routines which add to or set the values of these matrices.
- char *name – the name of the object

- `TransMethod transMethod` – the method for use of the transmforamtion matrix. At the beginning of each display list is placed commands to either multiply the current **DisplayDevice**'s trans matrix, or to replace it, or to not modify it at all. This variable indicates how this should be done. In almost all cases it should be `MULT`
- `doCent, doRot, doGlob, doScale` – flags indicating whether this object should obey commands to modify the respective transformation components. Routines exist to toggle them on or off
- `int *displayObj, *isFixed, *Dim` – flags indicating whether to draw the object, whether to keep it *fixed* (so that it does not respond to any transformation commands), and of what dimension it is (2 or 3). These values are actually stored at the beginning of the display list, and these pointers reference those locations

Virtual member functions

- `virtual void prepare(DisplayDevice *)` – this should be supplied by each derived class that needs to do some preparation before drawing. It is called by the **Scene** right before the scene is to be rendered. By default, it does nothing.

Method of use

A **Displayable** must either be registered with a parent (i.e. be a child of some parent), or it must be registered with a **Scene** (in which case it cannot be the child of any other **Displayable**). To create a new class derived from **Displayable**, you need to provide two constructors, one taking a **Scene** pointer, one a **Displayable** pointer.

Once created, then in the derived-class-provided prepare routine, the display list of for the **Displayable** must be set up. If this display list will never change during the program, it can be done just once in the constructor and no prepare is necessary. The display list is created by using **DispCmd** objects.

The display list for each **Displayble** must be given to a **Scene**, regardless of whether it is child or parent or both. It is these display lists which are given to the **DisplayDevice** to be rendered. The **Scene** keeps a list of registered **Displayable**'s as well so that it can prepare them for drawing. When a parent is prepared for drawing, it also prepares all its children, so only top-level parents need to be registered with the **Scene**.

Since this class is derived from **Pickable** and **ColorUser**, there are several virtual functions which may need to be supplied in order for the derived **Displayable** to be picked by the pointers, or to have access to the color database. See the descriptions of these objects for more info.

One note about the display lists: the reason they are done in the current scheme is that the CAVE requires the use of shared memory to hold the information for the rendering processes to draw, while a separate update process keeps track of updating the information in shared memory. Since the shared memory cannot have pointers to non-shared memory locations, the display lists are designed to ONLY hold integer and floating-point data. When drawing in the CAVE, a special **CaveScene** supplies shared memory blocks for the **Displayable**'s to use for their display lists.

Suggestions for future changes/additions

Right now every item is redrawn each time through the VMD event loop. However, there are times when nothing changes, and no redraw is necessary. The `prepare` routine might be modified to have it return TRUE if a redraw is needed, so that if none of the current **Displayable**'s need a redraw, that extra work is avoided.

Also, this class is kind of a mess, but most of it is necessary ...

24.6 Matrix4

Files:	`Matrix4.h, Matrix4.C`
Derived from:	none
Global instance (if any):	none
Used in optional component:	Part of main VMD code

Description

A 4 by 4 matrix of floats, used primarily for the transformation matrix of **Displayable** and **DisplayDevice** objects. Along with general routines to rotate, translate, and scale the matrix, this also contains overloaded operators for = (copy, either a matrix or a scalar), *= (multiply, by either a matrix or a scalar), and += (add a matrix).

There are many functions in this class to simply add or multiply the matrix, to take the inverse of the matrix, or to apply rotations, translations, etc. Most importantly, functions exist to take a 3- or 4-vector, and multiply (transform) it by this matrix.

Constructors

- **Matrix4::**`Matrix4(`void`)` – creates a 4 by 4 identity matrix.
- **Matrix4::**`Matrix4(`float`)` – creates a 4 by 4 matrix with all elements set to the given constant.
- **Matrix4::**`Matrix4(`float *`)` – creates a 4 by 4 matrix, copying the given array of 16 floats into the matrix (in row-major format).
- **Matrix4::**`Matrix4(`const Matrix&`)` – creates a 4 by 4 matrix identical to the matrix given.

Internal data structures

- `float mat[4][4]` – the matrix data itself.

Method of use

The many functions in this class are not listed here. The most common use of this is to maintain a transformation matrix for transforming 3D vectors from one coordinate space to another, i.e. to rotate, translate, and scale them. Once a 4 by 4 matrix 'M' has been set up, then a vector 'V' is tranformed and placed in the view vector 'VT' as:

```
M.multpoint3d(V, VT)
```

24.7 Scene

Files:	`Scene.h, Scene.C`
Derived from:	**PickList**
Global instance (if any):	`scene`
Used in optional component:	Part of main VMD code

Description

The **Scene** maintains and organizes a database with all the **Displayable** objects which are to be drawn by a **DisplayDevice**. It contains routines to add new **Displayable**'s to its database, and to draw them to the screen or to a file. It is derived from **PickList**, and so maintains the information about what diffent picking modes are available, as well routines to pick an item with a pointer.

There are two different sets of lists stored by the **Scene**, with each list kept for both the 2D case and the 3D case. The lists are:

- Registered **Displayable**'s; this list is used to prepare items to be drawn. Only top-level **Displayable** objects are kept in this list. When a **Displayable** is created, it registers itself with a **Scene**, and so gets added to this list.
- Registered *display lists*, the lists of drawing tokens created by **Displayables**. This list is used to actually draw the graphics, each item in this list represents one display list and is given to a **DisplayDevice** to be rendered.

This object also contains routines to apply a constant rotation to all **Displayable**'s each time they are drawn, as well as to apply general transformations to all the items in the scene. The most important function of **Scene** is, however, to draw the scene to a display or to a file.

Constructors

- **Scene::**`Scene`(void)

Internal data structures

- `int numDisplayable2D, numDisplayable3D` – the number of 2D and 3D **Displayable**'s which have registered. Derived classes maintain the actual lists.
- `int numCmdLists2D, numCmdLists3D` – the number of 2D and 3D display lists have been registered. Derived classes maintain the actual lists.

Virtual member functions

- `virtual void prepare(DisplayDevice *)` – goes through all the registered objects and prepares them for drawing. This should be called before `draw`.
- `virtual void draw(DisplayDevice *)` – actually draws the scene, using the given **DisplayDevice**. If this display is in stereo, this will first draw the left eye view, then the right.
- `virtual int filedraw(char *, char *, DisplayDevice *)` – very similar to `draw`, except this draws the scene to a file-based **DisplayDevice**. The first two arguments are the format, and the filename. The different formats and file-rendering objects are maintained by a global instance of the object **FileRenderList**.

Method of use

A **Scene** is created at the start of the program, and all **Displayable** objects should be added to that scene (or to another **Displayable** as a child). It is used in the primary event loop to prepare and draw all the **Displayable**'s. It is also used to pick items with a pointer. The global routine `VMDupdate` contains an example of using a **Scene** object.

There are two derived classes from **Scene**:

- **NormalScene** – this is the basic version, which is used for every case except when the CAVE display device is being used. It uses standard memory allocation routines to create storage for display lists, and to store the lists of **Displayable**s and display lists.

- **CaveScene** – this uses shared memory to store the display lists, so that the CAVE processes can all see the same data for rendering the scene.

Suggestions for future changes/additions

As mentioned for **Displayable**, the `prepare` routine could be modified to return a flag indicating whether anything has changed in the **Displayable** objects which would require a redraw. If nothing changed, then a redraw could be avoided.

25 Molecule objects

The following objects are used to store and represent the molecules in VMD, including their static data as well as their dynamic data. Items which have a specific section listed for them are explained in detail in that section.

Class Name	Section	Files
Animation	25.1	`Animation.h and .C`
Atom	25.2	`Atom.h`
AtomColor	-	`AtomColor.h and .C`
AtomRep	-	`AtomRep.h and .C`
AtomSel	-	`AtomSel.h and .C, AtomParser.h`
BaseMolecule	25.3	`BaseMolecule.h and .C`
CoorDCD	-	`CoorDCD.h and .C, ReadDCD.h and .C`
CoorFile	-	`CoorFile.h and .C`
CoorFileData	-	`CoorFileData.h and .C`
CoorPDB	-	`CoorPDB.h and .C, ReadPDB.h and .C`
DrawMolItem	-	`DrawMolItem.h and .C`
DrawMolecule	25.4	`DrawMolecule.h and .C`
DrawPatch	-	`DrawPatch.h and .C`
Fragment	-	`Fragment.h`
Geometry	-	`Geometry.h and .C`
GeometryAngle	-	`GeometryAngle.h and .C`
GeometryAtom	-	`GeometryAtom.h and .C`
GeometryBond	-	`GeometryBond.h and .C`
GeometryDihedral	-	`GeometryDihedral.h and .C`
GeometryList	-	`GeometryList.h and .C`
GeometryMol	-	`GeometryMol.h and .C`
GeometryTug	-	`GeometryTug.h and .C`

Table 4: VMD molecule objects, part 1.

Class Name	Section	Files
MolAction (many)	-	`MolAction.h and .C`
Molecule	25.5	`Molecule.h and .C`
MoleculeFile	-	`MoleculeFile.h and .C`
MoleculeFilePDB	-	`MoleculeFilePDB.h and .C`
MoleculeFilePSF	-	`MoleculePSF.h and .C`
MoleculeFileRaster3D	-	`MoleculeFileRaster3D.h and .C`
MoleculeList	-	`MoleculeList.h and .C`
MoleculeRemote	-	`MoleculeRemote.h and .C`
MoleculeSigma	-	`MoleculeSigma.h and .C`
Residue	-	`Residue.h`
Timestep	25.6	`Timestep.h and .C`

Table 5: VMD molecule objects, part 2.

25.1 Animation

Files:	`Animation.h, Animation.C`
Derived from:	none
Global instance (if any):	none
Used in optional component:	Part of main VMD code

Description

Animation is a base class for all **Molecule** objects in VMD. It is responsible for storing the *dynamic* data for molecule – those items which vary with time, instead of remaining constant. **Animation** stores an *animation list*, and has controls to add new data to the end of the list, as well as to position a *current frame* pointer in the list. Each time a **Molecule** is prepared (by calling it's virtual `prepare` routine since **Molecule** is derived from **Displayable**), the current position in the animation list is updated.

The **Animation** consists of basically an array of **Timestep** objects. Each **Timestep** contains the data for the molecule for a single step in the trajectory of the system.

Constructors

- **Animation::**`Animation`(void)

Enumerations, lists or character name arrays

The `AnimDir` enumeration lists the different directions in which the animation can be moving:

- `FORWARD`: Animation proceeds continually forward each update.
- `FORWARD1`: Animation goes to the next step, then is `PAUSE`d.
- `REVERSE`: Animation proceeds continually backward each update.
- `REVERSE1`: Animation goes to the prev step, then is `PAUSE`d.
- `PAUSE`: Updates to not change the current position in the list.

The `AnimPos` enumeration lists different positions in the list to which the animation position can jump, or can have new elements inserted or deleted:

- `END`: The end of the animation list.
- `START`: The start of the animation list.
- `BEFORE`: The frame before a specified frame.
- `AFTER`: The frame after a specified frame.
- `DELETE`: Unused.
- `WRITE`: Unused.

The `AnimFrames` enumeration lists the different ways in which frames can be added or deleted. It is either `ALL`, or `SELECTION`.

The `AnimStyle` enumeration lists the different ways in which animation can be done:

- `ONCE`: When the animation gets to the end or beginning, it will then pause.
- `LOOP`: When the animation gets to the end or beginning, loop around to the other end and continue through again in the same direction.
- `ROCK`: When the animation gets to the end or beginning, reverse direction and continue animating.

Internal data structures

- `double lastTime` – the clock time when the last position was determined. This is used to determine if the animation should proceed to a new frame, to control the speed.
- `int frameChanged` – whether the animation has changed to a new frame recently.
- `int needSetFrameChanged` – when this flag is set, then the `frameChanged` flag should be changed the next time it is possible to do so.
- `ResizeArray<Timestep *> tsList` – the frames themselves.
- `int currFrame` – the current frame, an index into `tsList`.
- `int frameSkip` – if we are jumping to a new frame, this indicates the frame to jump to.
- `float Speed` – the animation speed, from 0 (slowest) to 1 (fastest).
- `AnimPos appendPos` – the method to use the next time a frame is to be appended.
- `int appendFrame` – if we are inserting frames, the frame which we are to append before or after.
- `AnimDir animDir` – the current animation direction.
- `AnimStyle animStyle` – the method for doing animation.
- `int moveTo` – if this is not negative, this indicates a frame to which we are to jump the next time the animation position is updated.
- `float currTime` – the time which has elapsed since the beginning of the animation.

Nonvirtual member functions

- `int num(void)` – returns current number of frames in the list.
- `int frame(void)` – returns the current frame number.
- `int is_current(void)` – returns whether there *is* a current frame.
- `Timestep *current(void)` – returns the current frame itself, or NULL if there is none.
- `Timestep *item(int)` – returns the Nth frame.
- `void goto_frame(int)` – jump to the specified frame.
- `int has_frame_changed(void)` – returns whether the current frame has been changed recently.
- `void delete_animation(void)` – deletes all the currently stored frames.
- `void delete_frame(int)` – delete the Nth frame.
- `int append_frame(Timestep *)` – adds the given frame to the end of the animation list (or wherever has been previously determined for it to be placed).
- `int anim_update(void)` – updates the current position of the animation in the list, based on the current animation mode, and returns the index of the new current frame. Also sets the flags indicating whether the frame has changed. This is called during each call to `prepare` in the **Molecule** object.
- `void append_end(void)` – indicate to append the next frame at the end of the list.
- `void append_start(void)` – indicate to append the next frame at the start of the list.
- `void append_after(int)` – indicate to append the next frame after the specified frame.
- `void append_before(int)` – indicate to append the next frame before the specified frame.
- `void skip(int)` – set the current frame skip rate (the increment by which the current frame pointer is changed).
- `int skip(void)` – return the current frame skip rate.
- `void anim_dir(AnimDir)` – set the current animation direction.
- `AnimDir anim_dir(void)` – return the animation direction.
- `void anim_style(AnimStyle)` – set the animation method.
- `AnimSTyle anim_style(void)` – return the animation method.
- `float speed(float)` – set the speed of the animation.
- `float speed(void)` – return the current animation speed.

Method of use

Adding new frames: To add new frames, do the following:

1. Set the method for appending, either at the end, or beginning, or before or after some current frame.
2. Call the append routine.

Updating the current position: The `anim_update` routine changes the current position based on the current animation direction and style. It also sets flags indicating whether the current frame was changed any. It will return the index of the new current frame, which can be used to get the actual **Timestep** object for that frame.

Suggestions for future changes/additions

Right now, this is very top-level base class for **Molecule**, even higher than the **BaseMolecule** level. The functionality here could probably be rolled into **BaseMolecule**, so that a single class would have info about the structure and size of the molecule is was storing dynamic data for. Basically, it's backwards to have **Animate** above **BaseMolecule**.

25.2 Atom

Files:	`Atom.h`
Derived from:	none
Global instance (if any):	none
Used in optional component:	Part of main VMD code

Description

An **Atom** object stores the data for a single atom in a molecule. Each atom has the following information:

- Mass
- Charge
- Atom name
- Atom type
- Residue name
- Residue ID number
- Chain identifier
- Segment name
- List of bonds

Each **BaseMolecule** contains a list of these **Atom** objects.

Constructors

- **Atom::**`Atom`(int, float *, float *, char *, char *, char *, char *, char *, char *)
  The first argument is the index of the atom; the rest are the names used to distinguish this atom (as mentioned in the list above).

Enumerations, lists or character name arrays

Each atom has a set of 'extra' data values, other than it's position. The codes used as indices for these extra data values are:

- `ATOMCHARGE`
- `ATOMMASS`
- `ATOMBETA`
- `ATOMOCCUP`
- `ATOMRAD`
- `ATOMEXTRA` – the number of 'extra' data items.

The `BackboneType` enumeration lists the different types of backbone bonds which may exist:

- `NORMAL` – used if nothing else special about the backbone bond can be determined.
- `PROTEINBACK` – used for protein backbone bonds.
- `NUCLEICBACK` – used for nucleic acid backbone bonds.

The `ResidueType` enumeration lists the different classes of residues which this atom may be a part of:

- `NOTHING` – not in a residue of any noted special group.
- `PROTEIN` – a standard protein residue, i.e. an amino acid.
- `NUCLEIC` – a standard nucleic acid resiude, i.e. A,T,G,C, or U.
- `WATERS` – a solvent residue, most likely water.

Internal data structures

- **`int index`** – the index of this atom, as provided in the constructor.
- **`int bonds`** – the total number of bonds which this atoms participates in. Note that a bond will be stored twice, once for each of the atoms it connects.
- **`int uniq_resid`** – a unique integer indicating which residue this atom is in. This is not the same as te residue ID, since that is not necessarily unique in a given structure (particularly if the structure contains more than one segment or chain).
- **`int fragment`** – which fragment this atom belongs in.
- **`int bondTo[MAXATOMBONDS]`** – the indices of the other atoms to which this atom is bonded. An atom can bond to a maximum number of other atoms before an error occurs, this maximum is currently 8.
- `BackboneType bondType[MAXATOMBONDS]` – lists what type of bond each to the bonds to this atom are.
- `BackboneType atomType` – indicates if this atom is part of the backbone, and if so what kind of backbone.
- `ResidueType residueType` – what type of residue this atom is a part of.
- **`float pos[], extra[]`** – position and 'extra' data values for this atom. These are initial, or 'default', values for the atom when no other coordinates or extra values are known.
- **`char namestr[], int nameindex`** – the atom name, and the index of that name in the unique atom name array in the **BaseMolecule** which created the **Atom** object.
- **`char typestr[], int typeindex`** – the atom type name, and the index of that name in the unique atomtype name array in the **BaseMolecule** which created the **Atom** object.

- `char resnamestr[], int resnameindex` – the residue name, and the index of that name in the unique residue name array in the **BaseMolecule** which created the **Atom** object.
- `char residstr[], int residindex` – the residue ID, and the index of that name in the unique residue ID array in the **BaseMolecule** which created the **Atom** object.
- `char segnamestr[], int segnameindex` – the segment name, and the index of that name in the unique segment name array in the **BaseMolecule** which created the **Atom** object.
- `char chainstr[], int chainindex` – the chain ID, and the index of that ID in the unique chain ID array in the **BaseMolecule** which created the **Atom** object.

Nonvirtual member functions

- `float radius()` – return the atom radius.
- `float mass()` – return the atom mass.
- `float charge()` – return the atom charge.
- `float beta()` – return the atom beta value.
- `float occup()` – return the atom occupancy value.
- `void add_bond(int, BackboneType)` – add a new bond to this atom, which connects this atom and the specified atom, of the given bond type.
- `int bonded(int a)` – return TRUE if this atom is bonded to the specified atom.

Method of use

To create a new **Atom**, you specify all the names in the constructor, and then add the bonds to the atom one at a time using the `add_bond` routine. Later, when the molecular structure has been analyzed, the flags indicating the residue type, fragment index, etc. must be set (they are currently public members, so they can be accessed directly).

25.3 BaseMolecule

Files:	`BaseMolecule.h, BaseMolecule.C`
Derived from:	**Animation**
Global instance (if any):	none
Used in optional component:	Part of main VMD code

Description

BaseMolecule is the second-highest level in the class hierarchy for **Molecule** objects; it is derived from **Animation**, and is used to store the *static* information for the molecule, which is the basic information about the structure and contents of the system which do not change with time. **BaseMolecule** has no ability to read in this information itself, instead it provides routines which derived classes call in order to add new molecules, bonds, etc. to the molecule. **BaseMolecule** has no responsibility for graphically displaying the molecule, either; that is left to the **DrawMolecule** class, which is derived from **BaseMolecule** and **Displayable**.

You do not ever create a **BaseMolecule** instance directly, instead you create an instance of a class derived from **Molecule**, for which **BaseMolecule** is a base class (among others). When initially created, a new **BaseMolecule** is empty, with zero atoms and zero bonds. The derived classes contain the actual code to read in the molecular structure from a file or from a network connection, and they add the components to the internal storage via routines in **BaseMolecule**. When all the structure is completely read in, then routine are called in **BaseMolecule** to analyze the structure, and calculate such things as what atoms are in what residues, how many residues there are, what are the backbone bonds, etc. In fact, a molecule contains these structural features which are either directly added to **BaseMolecule**, or calculated by **BaseMolecule** after the basic structure is read in:

- N atoms, which are added to the system as they are read in from some source. Each atom has associated with it several names, which help to distinguish the atoms and make it possible for the VMD atom selection mechanism to choose subsets of atoms. These names are:
  - The *atom name*, which is usually a standard chemical nomenclature name. For example, alpha carbons in proteins have the name `CA`.
  - The *atom type*, which for some molecular data files is a name from a much smaller set of total names, used to classify atoms into small sets which are more manageable. For example, PSF files use atom types to simplify the parameterization of the atoms for molecular dynamics simulations. If the atom type is not known from the input files, it is just set to be the same as the atom name.
  - The *residue name*, a three-letter code which is usually quite standard. All glycine amino acids, for example, are in residues with the name `GLY`.
  - The *residue ID*, a numeric value assigned to the residues in a molecule, quite often in increasing order from one end of a linear chain to another. Particular useful in proteins, which are unbranched polymers.
  - The *chain ID*, a single-letter code used to distinguish atoms among different subcomponents of a molecule. If it is unknown for an atom, it is given a default value of 'X'.
  - The *segment name*, similar to the chain ID but allowed to be up to four characters. This is not as standard as the chain ID, and if it is unknown it will be given a default value of `MAIN`.

- N bonds, which are added to the system as they are read in or calculated by a derived class. The bonds are not stored directly in a list, however; instead, each atom stores a list of the bonds it participates in, which makes it much faster to display the molecule.
- N protein backbone bonds, and N nucleic-acid backbone bonds, which are determined by **BaseMolecule** after the atoms and bonds are added.
- N residues, where each residue is a collection of atoms and bonds which form some subunit.
- N segments, which each consist of a collection of atoms in a functional substructure within the molecule. For example, quite often a protein is a segment, and surrounding water molecules are another segment.
- N fragments, where each fragment is a collection of connected residues. If a system consists of three disconnected alpha helices, for example, then each helix would be a separate fragment. There are lists of protein fragments as well as nucleic acid fragments.

After the atoms and bonds are added to the **BaseMolecule**, then the connectivity is analyzed and the names of the atoms are used to find the backbone bonds, the residues in the system, and the fragments. Atoms, bonds, residues, and other components are numbered 0 ... N-1 in their respective lists.

One other item which **BaseMolecule** stores is the unique molecule ID number, which is assigned when the molecule is created. Each new molecule in VMD gets assigned an integer ID. The assigned ID values increase by one as each new system is loaded. The commands used to affect the molecules use these ID numbers to determine which molecule the command should affect. The name of the molecule displayed in the Molecule on-screen menu form has this ID number appended to the end of it.

Constructors

- **BaseMolecule::**`BaseMolecule`(void)

Enumerations, lists or character name arrays

The `MoleculeType` enumeration lists the different type of molecules which VMD understands. When the structure is analyzed, the type of molecule is determined. The types are:

- `UAPROTEIN`
- `EHPROTEIN`
- `UAPROTDNA`
- `EHPROTDNA`
- `NUCLEIC`
- `ORGANIC`
- `INORGANIC`

Internal data structures

- `MoleculeType type` – type of this molecule (from above list).
- `int nAtoms` – number of atoms in this molecule. Can be zero.
- `int nBonds` – number of bonds in this molecule.
- `int nBackProtein` – number of protein backbone bonds.
- `int nBackDNA` – number of nucleic-acid backbone bonds.
- `int nResidues` – number of residues.
- `int nSegments` – number of segments.
- `int ID` – molecule integer ID number.
- `int maxAtoms` – maximum storage currently allotted to store the atoms (i.e. size of `atomList` array, which may be larger than the actual number of atoms stored there).
- `Atom **atomList` – array of **Atom** objects.
- `NameList<int> atomNames` – list of unique atom names in this molecule.
- `NameList<int> atomTypes` – list of unique atom types in this molecule.
- `NameList<int> resNames` – list of unique residue names in this molecule.
- `NameList<int> resIds` – list of unique residue ID's in this molecule.
- `NameList<int> chainNames` – list of unique chain ID's in this molecule.
- `NameList<int> segNames` – list of unique segment names in this molecule.
- `ResizeArray<Residue *> residueList` – list of which residues are connected to which.
- `ResizeArray<Fragment *> fragList` – list of connected residues, which form fragments.
- `ResizeArray<Fragment *> pfragList` – list of connected protein residues which form protein fragments. A protein fragment is a single chain from N to C.
- `ResizeArray<Fragment *> nfragList` – list of connected nucleic acid residues, which form nucleic acid fragments. A nucleic acid fragment is a single chain from 5' to 3'.

Nonvirtual member functions

- `void init_atoms(int)` – initializes storage to store data for N atoms. This only allocates memory, it does not store anything in that memory. This should be called when constructing a new molecule, when the number of atoms has been determined but before the atom data itself is stored into the **BaseMolecule** structures.
- `int add_atom(char *, char *, char *, char *, char *, char *, float *pos, float *extra)` – add a new atom to the molecule, with the specified names, and given starting x, y, z position (`pos`) and given starting extra data (such as beta value and occupancy).

- `int add_bond(int, int, Atom::BackboneType = Atom::NORMAL)` – add a new bond between the atoms specified as the first two arguments, where the bond is of the specified type. See the description of **Atom** for a list of the different bond types.

- `int find_backbone(void)` – determines which bonds are backbone bonds, and stores this data in the **Atom** objects stored in the `atomList` member. Returns the number of backbone bonds found.

- `int find_residues(void)` – find which atoms are in which residues, and store this data. Returns the number of residues found.

- `int find_waters(void)` – Find the waters, based on their residue name, and return the number found.

- `int find_segments(void)` – Find the segments in the molecule, and store this data. Return the number round.

- `int find_fragments(void)` – Find the fragments in the molecule, and store this data. Return the number found.

- `int find_atom_in_residue(char *nm, int r)` – find the index of the first atom in the specified residue with the given name, or return -1 if none is found with that name.

- `int id(void)` – return the ID of the molecule.

- `Atom *atom(int)` – return the Nth **Atom** for the molecule.

- `char *atom_full_name(int, char * = NULL)` – return a string containing the full name specification for the Nth atom. If the second argument is not NULL, the name will be placed in the given character array. Otherwise, an internal static buffer will be used to hold the name. The name is of the form:

  `<mol ID>:<atom index>`

  This name is guaranteed to be unique for each atom.

- `char *atom_short_name(int, char * = NULL)` – the same as for the full name, except the name returned is of the form:

  `<residue name><residue ID>:<atom name>`

  This form is nicer to read, but is not generally unique for a given atom.

- `float default_charge(char *)` – returns a default partial charge to use for the specified atom name. Used when this information is not supplied by the source of molecular structure. The following routines also supply default data based on a given atom name.

- `float default_mass(char *)`

- `float default_radius(char *)`

- `float default_occup(char *)`

- `float default_beta(char *)`

Virtual member functions

- `virtual int create(void)` – the main virtual routine provided by this class. This is used after a new **Molecule** subclass has been created (with the required information for reading the molecule given in the constructor). Initially the **Molecule** is empty; to initialize it, the `create()` routine is called which will then start the actual process of reading in the data. Each version of `create()` supplied by the derived classes should, after doing it's own creation, call the `create()` routine in the parent class. This routine returns the success of the creation operation.

- `virtual float scale_factor(void)` – returns (possibly calculating first) the scaling factor required to scale the coordinates for the current timestep to fit in a box from -1 ... 1 in all dimensions.

- `virtual void cov(float &, float &, float &)` – return the position of the center of volume of the current coordinate set.

Method of use

A new molecule is first created by using 'new' with the proper subclass of **Molecule** (**Molecule** is the 'standard' class to use for all molecule objects in VMD; classes derived from **Molecule** are specialized to read in data from different sources, while classes above the **Molecule** level only deal with some of the information required to store and display and animate a structure.). Then, after the new instance is assigned to a **Molecule** pointer, then the `create()` virtual function should be called. This will actually result in all the action being done, for example data files will be read or network connections will be established. The version of `create()` in **BaseMolecule** should be called *after* the molecule has been read in by the derived classes. It analyzes the structure and finds the backbone bonds, fragments, etc. When `create()` is finished, the molecule is ready to go. If `create()` does not return TRUE, however, the creation failed (i.e. the files could not be opened), and the new molecule will still be empty.

25.4 DrawMolecule

Files:	`DrawMolecule.h, DrawMolecule.C`
Derived from:	**BaseMolecule**, **Displayable**
Global instance (if any):	none
Used in optional component:	Part of main VMD code

Description

DrawMolecule is derived from **BaseMolecule**, and is the next level in the **Molecule** class hierarchy. It embodies the level of control and functionality responsible for creating a graphical image of the molecule. It is derived from **Displayable** (actually, from **Displayable3D**), and so will be part of the **Scene** that will be drawn by a **DisplayDevice**.

Each **DrawMolecule** keeps a list of **DrawMolItem** objects; each **DrawMolItem** is also a **Displayable**, and is responsible for a single *representation* of the molecule. Each representation contains a display list used to draw a single view of the molecule. To define a single representation, each **DrawMolItem** contains one instance each of the following objects:

- **AtomColor** – defines how to color each atom in the representation.
- **AtomRep** – defines what shape to draw the components of the representation.
- **AtomSel** – determines which atoms out of all the ones in the molecule are to be drawn in that representation.

When a new **DrawMolecule** is created (i.e. when a new **Molecule** is created), it is added as a child **Displayable** to a global **MoleculeList** object. Thus, you can rotate just the **MoleculeList**, and this will rotate all the molecules at the same time. When a molecule is first displayed, it is scaled and translated to fit within a -1 ... 1 size box centered around the origin. In order to have all the molecules be the proper size in relation to each other, and to preserve the spatial position of the molecules, only one molecule is used to determine the scaling and translation positions, and the others then use those same scaling and translation values.

Each **DrawMolItem** added to this object are added as child **Displayable** objects of **DrawMolecule**, and so scaling and rotating the molecule will do so to all the **DrawMolItem** objects at the same time.

Constructors

- **DrawMolecule::**`DrawMolecule`(Scene `*)`
  Specifies the **Scene** to which to add this object. Should only be used if no **MoleculeList** is being used.
- **DrawMolecule::**`DrawMolecule`(Displayable `*)`
  Specified a parent **Displayable** to which to add this object as a child. This is the main form used in VMD, as all **Molecule** objects are added to a global **MoleculeList**.

Internal data structures

- `int active` – is this molecule an active one? This is used by **MoleculeList**.

- `int needRedraw` – a flag indicating whether this item needs to reconstruct its display list during the prepare routine. After the display list is remade, this flag is cleared.
- `ResizeArray<DrawMolItem *> repList` – the list of representations.

Nonvirtual member functions

- `int atom_displayed(int)` – returns TRUE if any of the representations are currently displaying the given atom.
- `int components(void)` – returns the number of representations in the molecule.
- `DrawMolItem *component(int)` – return a pointer to the Nth representation.
- `int del_rep(int)` – delete the Nth representation, and return the success.

Virtual member functions

- `virtual void set_name(char *)` – used to change or set the name of the molecule. Called during the create phase of molecule construction.
- `virtual void create_cmdlist(void)` – regenerates the display list, which contains the list of primitive drawing commands necessary to draw the object.
- `virtual int create(void)` – the version of create for **DrawMolecule** ... this is called by derived classes after they have read in and initialized the molecule. The **DrawMolecule** version of create will then call the **BaseMolecule**'s create routine, and finally then construct the initial version of the display list.
- `virtual void prepare(DisplayDevice *)` – provided since this is a **Displayable** object. This determines if the display list needs to be reconstructed, and if so it does the reconstruction.
- `virtual int add_rep(AtomColor *, AtomRep *, AtomSel *)` – requests for a new representation to be made, using the given objects to describe what the representation should be. Returns success.
- `virtual int change_rep(int, AtomColor *, AtomRep *, AtomSel *)` – changes the Nth representation to use the new settings specified in the given objects.

Method of use

The user should never create a **DrawMolecule** directly; it should instead be used as a base class for **Molecule**.

25.5 Molecule

Files:	`Molecule.h, Molecule.C`
Derived from:	**DrawMolecule**
Global instance (if any):	none
Used in optional component:	Part of main VMD code

Description

This is the class used generally throughout VMD to represent and access a molecule. It is derived from several base classes which provide the storage and control over the molecule's structure (**BaseMolecule**), animation list (**Animation**), and graphical display (**DrawMolecule**). It still does not contain the abilities to read in new molecular data from different sources, instead these are in classes derived from **Molecule**. This class is used as the basic class that all other parts of VMD are aware of, since once a molecule is read in from some source, it acts just like any other molecule.

The one level of extra functionality which is provided directly by the **Molecule** class is the ability to read in new coordinate frames from an external coordinate file, either a PDB or a DCD file.

Constructors

- **Molecule::**`Molecule`(char *, Scene *)
  The first argument is the *source* of the data, it is a string indicating either a file, or some remote computer, or perhaps some other source. The second argument is the **Scene** to which this molecule should be added. This version of the constructor should only be used if no **MoleculeList** is being used.

- **Molecule::**`Molecule`(char *, Displayable *)
  Same as above, except a parent **Displayable** is specified instead of a **Scene**. This is the version that should be used mainly in VMD, with the global **MoleculeList** object specified as the parent **Displayable** for the molecule.

Internal data structures

- `char *source` – the source of the molecule structure data.

- `ResizeArray<CoorFileData *> coorIOFiles` – data on which files containing atomic coordinates are to be read in by this object. Each **CoorFileData** instance contains data on which file to read, what type of file it is, and which frames to read from the file. The same data is stored in the case where a file is to be written instead of read.

Nonvirtual member functions

- `int read_coor_file(char *, int, int=(-1), int=(-1), int=(-1))` – request to read in the specified file. The second argument is the type, and the remaining are the beginning frame, ending frame, and frame skip values. If any are not specified or are ¡ 0, default values are used.

- `write_coor_file(char *, int, int=(-1), int=(-1), int=(-1))` – same as above, but requests to write out a file.

Virtual member functions

- `virtual void prepare(DisplayDevice *)` – does any preparation necessary before redrawing. If data files are being read, this will read the next frame from them.

Method of use

New molecules are created in VMD in the **CmdMolNew** object, which is a particular **Command**-derived object. When a new molecule is to be created, the type of molecule source and necessary data are specified (such as filenames, remote computer names, etc) and given to the constructor of a specialized subclass of **Molecule**. The resulting (empty) molecule is assigned to a `Molecule *` pointer variable, and then the virtual `create` routine is called. After this, the molecule is added to the **MoleculeList**, and any extra actions are taken such as requesting for more coordinate files to be read, etc.

25.6 Timestep

Files:	`Timestep.h, Timestep.C`
Derived from:	none
Global instance (if any):	none
Used in optional component:	Part of main VMD code

Description

The **Timestep** class is used to hold the dynamic data for a **Molecule** for a single step in a trajectory. It hold the coordinates, velocities, energies, etc. that vary with time. As each new **Timestep** is read in from some source (a file, or from a network connection) it is added to the *animation list* for the associated molecule; this list is maintained by the **Animation** object, from which each **Molecule** is derived.

Almost all the items in this class are public, so they may be retrieved quickly by other objects in VMD. This is not such a great design, however; see the notes below about future changes.

Constructors

- **Timestep::**`Timestep(`int n, float DT, float *newdata = NULL`)`
  The first argument is the number of objects (atoms) for which dynamic data will be stored. The second is the time which has elapsed between generation of this step a previous step (for example, for a molecular dynamics simulation, this would be the integration timestep). The final argument is the data to store in this step, if it has already been allocated. If not, new space will be allocated by **Timestep**.

Enumerations, lists or character name arrays

There are a number of globally defined macros in this class which are used as indices into the data arrays stored by this object. They are:

1. **Energy Data Indices:**
   - `TSE_BOND` – index to bond energy value.
   - `TSE_ANGLE` – index to angle energy value.
   - `TSE_DIHE` – index to dihedral energy value.
   - `TSE_IMPR` – index to improper dihedral energy value.
   - `TSE_VDW` – index to van der Waal energy value.
   - `TSE_COUL` – index to Coulomb energy value.
   - `TSE_HBOND` – index to hydrogen-bond energy value.
   - `TSE_KE` – index to total kinetic energy value.
   - `TSE_PE` – index to total potential energy value.
   - `TSE_TOTAL` – index to total energy value.
   - `TSE_TEMP` – index to temperature value.
2. **Per-Atom Data Indices:**

- TSP_X – x coordinate.
- TSP_Y – y coordinate.
- TSP_Z – z coordinate.

3. **Per-Patch Data Indices:**

    - TSP_XLEN – length of patch in x direction.
    - TSP_YLEN – length of patch in y direction.
    - TSP_ZLEN – length of patch in z direction.
    - TSP_LOAD – computational load on the patch.
    - TSP_ATOMS – number of atoms in the patch.
    - TSP_NODE – node which contains the patch.

Internal data structures

- `int needDataDelete` – flag for whether we need to delete the storage space for the `data` array when this object is deleted.
- `int num` – number of atoms in this **Timestep**.
- `float *pos` – coordinates of all the atoms. This array has 3 * `num` elements, arranged as ([x1 y1 z1] [x2 y2 z2] ...).
- `float *data` – for each atom, there can be 'extra' data, which **Timestep** does not not know the contents of. This is the data provided as the last argument in the constructor.
- `float energy[]` – energy values for this timestep. By default, they are zero unless they are explicitly changed by the creator of this timestep.
- `int numPatches` – the number of patches stored in this step. It may be zero (and definitely is zero for data which did not come from a network connection).
- `float *patchData` – data stored for each patch (the size, location, atoms per patch, etc.).
- `float minpos[], maxpos[], mindata[], maxdata[], etc` – the minimum and maximum values found in this **Timestep** for the various data quantities. Several parts of VMD need to know the range of data values some quantity takes in a **Timestep**.
- `float COV[]` – the x,y,z position of the center of volume.
- `float scale_factor` – the amount by which you would need to scale the coordinates in this step in order to fit them in a box in the range (-1 ... 1) in all dimensions.
- `float dt` – the time required to compute this timestep. If this is unknown, it is set to zero.
- `int Initialized` – has this been initialized? If so, then all the data items have been placed into the proper arrays, and the max and min values have been calculated.

Nonvirtual member functions

- `float *create_patch_storage(int)` – allocates internal storage to store data for the given number of patches. The number of patches is stored internally as well. Returns a pointer to the newly-allocated array.
- `void init(void)` – after all the data has been placed into a new **Timestep** instance, this routine calculate the max/min values for all the quantities, and anything else required based on atom coordinates, etc. Used for scaling and translating purposes.

Method of use

A new **Timestep** is created each time new coordinate data needs to be added to the end of the animation list. Use the following steps:

1. Create a new instance:

   ```
   Timestep *ts = new Timestep(atoms, dt);
   ```

2. Copy coordinate and energy data into the proper arrays, by accessing the public data items described above.
3. If there is patch data, first call `create_patch_storage` with the number of patches (to allocate storage), and then copy the patch data into the array which is returned by the function.
4. After all data is entered, the last step is to call the `init` routine. This goes through all the stored quantities (x, y, z position, atoms/patch, etc) and calculates the maximum and minimum values. It also finds the position of the center of volume, and the proper scaling factor.

Suggestions for future changes/additions

This could perhaps benefit from having a **NameList** store the different quantities, with a specific name associated with each quantity. Then, instead of having to hard-code into **Timestep** what data is being stored, you could just have a routine to add new items with a given name to the object, and other routines to access data by providing the proper name and index (or maybe just provide the name, and have returned the proper array storing the data).

26 Remote connection objects

The following objects are used to allow VMD to connect to a remote MD simulation program, and display the results of that simulation as they are calculated. The objects responsible for remote simulation connection and control are currently in the experimental stage, however, and will be described later when their design is closer to being final.

Class Name	Section	Files
DrawPatch	-	`DrawPatch.h and .C`
MoleculeRemote	-	`MoleculeRemote.h and .C`
Remote	-	`Remote.h and .C`
RemoteList	-	`RemoteList.h and .C`

Table 6: VMD remote simulation control objects.

27 User interface objects

The following objects are used to implement the user interface modules in VMD, including the text interface, the 2D mouse interface, the GUI interface, and (eventually) the 3D interface. Items which have a specific section listed for them are explained in detail in that section.

Class Name	Section	Files
Buttons	-	`Buttons.h and .C`
Command	27.1	`Command.h and .C, Cmd*.C and .h`
CommandQueue	27.2	`CommandQueue.h and .C`
FormsObj	-	`FormsObj.h and .C, *FormsObj.h and .C`
Mouse	27.3	`Mouse.h and .C`
UIObject	27.4	`UIObject.h and .C`
UIText	27.5	`UIText.h and .C`
UIVR	-	`UIVR.h and .C`

Table 7: VMD user interface objects.

27.1 Command

Files:	`Command.h, Command.C, Cmd*.h, Cmd*.C`
Derived from:	none
Global instance (if any):	none
Used in optional component:	Part of main VMD code

Description

This is a base class for all the objects used in VMD to peform actions (commands) on the molecules or program state. Each time a request is to be made to do some command (i.e. rotate the current scene, load a new molecule, change a color, etc), an instance of a class derived from **Command** is created, and given to a **CommandQueue** object. The queue will then call virtual routines within **Command** to perform the action it knows how to do. There are a large number of relatively small classes derived from **Command**, each one used to perform a particular type of action. These objects are all in the files with names starting with 'Cmd', for example CmdDisplay.h and .C.

Each class derived from **Command** has these characteristics and abilities:

- A unique integer code, which must be one of the items in the enumeration `Cmdtype` located at the top of the Command.h file. The code for the particular subclass of **Command** is passed to the **Command** constuctor, and available via the `gettype()` function.
- The data necessary to perform the required action. This data must be given to the object via the constructor when it is created.
- A textual equivalent of the command. This is created after the object is instantiated, based on the particular data given to that instance. This string is used to echo commands to the console or to a file.
- The ability to perform the command itself. The virtual function `execute()` is used to request the **Command** object to do its stuff.

The basic philosophy behind the use of **Command** is that each action that the user may possibly request to do should be encapsulated within a class derived from **Command**. Each subclass should know also how to create a text equivalent of the command. User interface objects in VMD (those derived from **UIObject** must use these **Command**'s to do the actions requested by the user, by creating new **Command** instances and giving them to a **CommandQueue** object to execute.

Constructors

- **Command::**`Command`(Cmdtype, int)
  The first argument specifies the type code for the command, while the second argument is the ID of the user interface object which created this **Command** instance (see section 27.4).

Enumerations, lists or character name arrays

There is one large enumeration Ĉmdtype in **Command**, which contains a set of unique ID's for each derived class of **Command**. This is done as an enumeration here so that other objects in the program (notably **UIObject**s) .. can be written with explicit codes to allow them to check what type of command they may be working with. When a new Command object is being written, a new value must be added to this list.

Internal data structures

- `Cmdtype mytype` – unique code for this derived class.
- `int textExists` – whether or not the text equivalent of this command has been created yet. This only needs to be done at most once, and sometimes not at all (if commands are not being logged to a file or the screen, for example).
- `int hasTextCmd` – whether or not this command even HAS a text equivalent ... some commands do not, for example Event's.
- `char cmdtextbuf[256]` – character buffer to hold the text equivalent string. This can be at most 256 characters.
- `ostrstream *cmdText` – character stream used to format the text equivalent.

Nonvirtual member functions

- `void check_and_create_text(void)` – when called, this routine will create the text equivalent of the command, if necessary (that is, when the command HAS a text equiv and the string has not yet been created). This will be done by calling the virtual function `create_text()`.
- `int execute(void)` – execute the command; this will call the virtual function `do_execute()`, and return whether the action was successful.
- `int has_text(void)` – return whether or not the command has a text equivalent (if not, the text equivalent is an empty string).
- `char *text(void)` – return the text equivalent. This will call the routines to create the string if necessary.
- `Cmdtype gettype(void)` – return the type code for this object.
- `int getUIid(void)` – return the ID number of the **UIObject** which created this **Command** instance.
- `friend ostream& operator<<` – an overload of the ¡¡ operator to allow a **Command** to be easily printed to an ostream.
- `friend Inform& operator<<` – the same, to send the text equivalent of the **Command** to an inform object. For example:

```
Command *cmd = new CmdAck(); msgInfo << *cmd;
```

Virtual member functions

- `virtual int do_execute(void)` – the 'heart' of each class derived from **Command**. This function performs the required task (if possible), and returns TRUE if successful, FALSE if there is an error. When errors occur, messages should be printed to `msgErr` in this function, and preferrably NOT within the lower-level objects (i.e. the **Molecule** or **DisplayDevice** objects).
- `virtual void create_text(void)` – formats the text equivalent of the command, writing the text to the `cmdText` stream.

Method of use

Using a Command object: Whenever an action is to be performed, you create a new instance of the particular **Command** derivative with a 'new' operation:

```
Command* cmd = new CmdTranslate(x, y, z, 0);
```

Then, the command is queued, by adding it to a **CommandQueue** object, by appending it to the queue:

```
commandQueue->append(cmd);
```

When doing this in a **UIObject**, there is a member function `addcommand(Command *)` to do this more easily.

Once queued, the command will be executed during the main event loop in VMD, and then the instance will be deleted after it is done executing. Do NOT queue a static **Command** object, it must have been created using new. (But see the description of the `runcommand` function in **CommandQueue**, section 27.2.)

Developing a new class derived from Command: The following 'checklist' should be followed when creating a new type of command:

1. Identify the 'type' of command ... to affect the display, the molecules, the on-screen forms, or whatever. There are several Cmd* files, find one where it belongs, or create a new one of the form CmdXXXX.h and .C

2. Create a text equivalent of the commands. Text commands in VMD consist of one or more words separated by whitespace, and terminated by a newline. The first word of the command should be an 'identifying' word to indicate the general type of action to be performed, and the other words should be arguments to describe exactly what action to do in the general category.

3. Put a new code in the `Cmdtype` enumeration in Command.h

4. Create the class definition in CmdXXXX.h. The only functions that are needed are the constructor/destructor, and the virtual functions `do_execute` and `create_text`. If the command does not have a text equivalent, you do not need to provide a version of the latter.

5. Put the member functions in the file CmdXXXX.C. The easiest thing to do is to follow the patters in the Cmd*.C files.

6. All commands must be understood by the text parser (**UIText**). The parser reads in new commands, looks at the first word, and calls a text *callback function* to process that command. If the new command is just a variant of another existing command (i.e. the first word of this new command is the same as some existing command), then it should be put in an existing Cmd* file, and the function 'text_cmd_WWWW' at the top of the respective .C file (where WWWW is the first word of the command) should be updated to understand the new command. Add code to look for the proper strings, to create a new instance of the new command, and to add it to the given **CommandQueue**. If this command contains instead a new word for the VMD vocabulary, go on the next step, otherwise the next step can be skipped.

7. For commands which contain a new starting word, so that they are placed in new files CmdXXXX.h and .C, two things must be done to allow VMD to be aware of the text versions of these commands and to be able to execute them:

   (a) At the beginning of CmdXXX.h and .C, a new function 'text_cmd_WWWW(int argc, char **argv, CommandQueue *cq, int id)' must be defined and written. This routine must know how to examine the text command (as provided in token form in argc and argv) and create instances of the proper **Command** object to be added to the command queue. Other Cmd* files contain examples.

   (b) In the file UIText.C, there is a section where all the words that are at the beginning of VMD commands are 'registered', with the callback function to call when that word is seen. Add a line similar to the others in this section, to register the new word and to specify the new function 'text_cmd_WWWW'.

8. If other user interfaces (other than the text interface) are to have the ability to execute this same command, add this ability to them as well.

9. Finally, update the on-line help file `vmd_help.html`, and the User's Guide.

Suggestions for future changes/additions

The printing of error message is right now very uneven. Some commands do so in the `do_execute` routine, others leave it to the objects being operated upon to do the error message printing. It would be best if all error messages were printed (if possible) in `do_execute` routines, with the underlying objects just returning error codes to allow `do_execute` to tell what is going wrong.

27.2 CommandQueue

Files:	`CommandQueue.h, CommandQueue.C`
Derived from:	none
Global instance (if any):	`commandQueue`
Used in optional component:	Part of main VMD code

Description

CommandQueue maintains a queue of **Command**'s, and will go through this queue and execute them when requested. The queue is first-in first-out. This object also has the ability to log commands to a file.

There is one global instance of this class in VMD, called `commandQueue`. It is used by all the user interface objects (**UIObject** classes). Each time a new action is requested by the user or some other part of VMD, a new **Command** instance is created and added to the **CommandQueue**. Within the main event loop of VMD, after each **UIObject** is checked for new events, the commands in the queue are all executed until the queue is empty (since the execution of one command may result in the queuing of a new command, this process continues until the queue is empty). Each time the queue is checked, the following occurs:

- The next **Command** instance at the top of the queue is fetched to be executed.
- If commands are being logged to a file, the text representation of the command is written to that file.
- The command itself is executed, and the return code (success of the command) is stored.
- After it is executed, each **UIObject** active at that time is informed about the command, and are given both the pointer to the command, and it's success flag. This is done so that user interfaces can update their visual display or internal state to reflect the fact that something has changed. In this way, even though a single user interface component may do some action, ALL user interface's can know when to update their state to reflect the changes in the program.
- After this, the **Command** instance is deleted. There is one routine which does this differently, 'runcommand' (see below).

Constructors

- **CommandQueue::**`CommandQueue`(void)

Internal data structures

- `ResizeArray<Command *> cmdlist` – the queue of commands.
- `int loggingCmds` – flag indicating whether commands are being logged to a file.
- `FILE *logfile` – pointer to the FILE strcture for the output log file, if one is being used.

Nonvirtual member functions

- `int do_execute(Command *)` – executes the given command, echoing it to the log file if necessary and informing all the **UIObject**s. This will NOT delete the **Command**; this routine is called by other routines in **CommandQueue**.
- `void delete_current(void)` – deletes the command which is at the top of the queue, and moves all the other commands up. This is called by 'execute' after the command has been run.
- `int logging(void)` – return TRUE if logging of commands is turned on.
- `void log_on(char *)` – turns on logging of commands, to the given file.
- `void log_off(void)` – turns off logging, closes the open log file if necessary.
- `int num(void)` – return number of commands in the queue.
- `int append(Command *)` – puts the given command instance at the end of the queue. This will not execute it; commands are not executed until one of the following three routines are called. This returns TRUE if the command could be added.
- `int execute(void)` – executes just the first command in the queue, by calling `do_execute` and then `delete_command`. Returns the success code of the command, or FALSE if not command is available.
- `void execute_all(void)` – just calls `execute` until the queue is empty.
- `int runcommand(Command *)` – this routine executes the given command, but *without* adding it to the queue. It should be used when you definitely know that a command should be done immediately, and can be done out-of-order with respect to the other commands which might be in the queue. When the command is done, it is deleted. This returns the success code of the command.

Method of use

When objects other than **UIObject**s wish to execute a command, they should use the global object `commandQueue`, and append the command:

```
commandQueue->append(new CmdReshape());
```

This command will not actually be run until the main event loop in VMD is run. If a command needs to be executed immediately, use runcommand:

```
int success = commandQueue->runcommand(new CmdReshape());
```

When **UIObject**s are adding commands, there are special funcions in **UIObject** to make this faster, 'addcommand' and 'runcommand'. Each **UIObject** is given a pointer to a **CommandQueue** object, and so the 'addcommand' routine will take a command and add it to that queue. For example, in **UIText** there might be:

```
addcommand(new TextEvent("text command", id()));
```

Most commands have as their last argument an id code for the **UIObject** which created them. This is that so **UIObject**s can tell, when they are told that the command has executed, who started it all.

Suggestions for future changes/additions

There should be a version of runcommand that allows for executing a command without 'new'ing it. At least, there should be an argument to runcommand to allow this possibility.

27.3 Mouse

Files:	`Mouse.h, Mouse.C`
Derived from:	**UIObject**
Global instance (if any):	`mouse`
Used in optional component:	Part of main VMD code

Description

The **Mouse** class provides all the capabilities to provide a 2D pointer for a particular **DisplayDevice**. **Mouse** will use virtual functions with **DisplayDevice** to determine when a mouse button has been pressed, and where it is located. Based on this, the **Mouse** will rotate, translate, or scale the objects in the current **Scene**. It can also be used to 'pick' items, by clicking one of the mouse buttons while the cursor is positioned over the item. Finally, it is used to activate and choose from a *pop up menu*. **Mouse** is derived from **UIObject**, and is one of the main user interface methods in VMD (along with the text console, the GUI, the 3D pointers, and any external command input programs).

It is assumed the **Mouse** has three buttons, left, middle, and right. The right button is always used to active the pop-up menu, while the other two are used to apply transformations to the current **Scene**. The left and middle buttons are also used to select items. For some things, you can only click on them (press button + release) for something useful to happen. For other things, you can select them (press button), *pull* them (move the mouse while the button is held down), and release. The **Mouse** will issue commands to do the button presses, mouse motion, and button releases associated with picking objects.

The **Mouse** is always in one of several *modes*, which are changed by various means (text commands, pop-up menu options, or keyboard shortcuts). These modes are:

- *Rotation* mode: When held down, the left button will rotate things about the X or Y axis as the mouse is moved, in a 'virtual trackball' method. The middle button is used to rotate about the Z axis (perpendicular to the screen).

- *Translation* mode: The left button will move the objects in the X-Y plane; the middle button will move things in the Z direction (toward or away from the viewer).

- *Scaling* mode: The left button, when held down and the mouse moved left-right, will enlarge or shrink the scene. The middle button will do the same thing, but with a larger scaling factor (so the left is for fine tuning, the middle is for coarse-grain scaling).

- *Light* mode: This acts just like rotation mode, but instead of rotating all the displayed objects, the mouse will only rotate the position of one of the light sources. When setting this mode, you also select which light to rotate (0 ... 3).

- *Pick* mode: While the mouse can be used to select certain items while it is in the other modes, there are certain special pick-and-drag operations in VMD which can only be done if the pointer is put in a special *picking mode*. By default there are five picking modes, which are numbered 0 ... N-1, in this order: query, select atom, select bond, select angle, or select dihedral. Picking on an object when in a non-picking mode does not allow you move the mouse around any while the item is selected; in that case, when the button is pressed, if an item is selected it acts just as if the button were immediately released. But if the mouse is in a special picking mode, then when the button is pressed and an item is selected, you can move the mouse around while the button is held down and have extra actions be taken during the motion. For example, there are special modes which are used to add perturbative forces to atoms; to to do this, the mouse must be placed in the atom-tug mode. When setting the mouse into a pick mode, you also must select which mode it is to be.

Note that clicking on an item with the left button while the mouse is NOT in a pick mode is equivalent to clicking-and-releasing the button with the mouse in the atom select mode. Clicking on an item with the middle button in these same situations is the same as using the left button, but instead the picking operation is done as if the mouse were in the bond selection mode.

Finally, the **Mouse** contains the ability to check the keyboard for special events as well as the 2D pointing device. The **Mouse** maintains a list of user-customizable *keyboard shortcuts*, which associate a specific keypress (i.e. 'A' or 'r') with a text command. When that key is pressed while the **Mouse** is in the graphics display window, the associated text command is executed as if the entire command had been entered at the console prompt.

There is one global instance of the **Mouse** class, `mouse`. It is created after the global **DisplayDevice**, and **CommandQueue** objects are created. Note that if a 'default' **DisplayDevice** is created, the **Mouse** will not be able to check for events, determine a pointer position, etc, and so will do nothing.

Constructors

- **Mouse::**`Mouse(`CommandQueue *, DisplayDevice *`)`
  The first argument is the **CommandQueue** object which this class should use when creating new **Command**s to be executed, and the second argument is the **DisplayDevice** which this class should be a 2D pointer for. The **DisplayDevice** must provide all the device-specific information for the **Mouse**, such as the X,Y location of the pointer, the state of the mouse buttons, and the ability to post and activate a pop-up menu.

Enumerations, lists or character name arrays

The `MoveMode` enumeration lists the possible mouse modes. These are:

- `ROTATION`.
- `TRANSLATION`.
- `SCALING`.
- `LIGHT`.
- `PICKING`.

Internal data structures

- `DisplayDevice *dispDev` – display to use for device-specific functionality, such as checking for events, determining the pointer position, etc.
- `MoveMode moveMode` – current mode the mouse is in.
- `int activePickMode` – if the current mode is `PICKING`, this variable indicates which pick mode is active.

- int pickInProgress – if this is zero, no picking operation is currently in progress (which means that it is NOT that case that a mouse button has been pressed and selected something, with the mouse button still left down). If something is currently being selected with the mouse and the button is still down, this flag indicates which button is being used. In fact, pickInProgress = pick-button + 1.
- int moveObj – if the current mode is LIGHT, this variable indicates which light to rotate.
- int currX, currY, oldX, oldY – current and previous positions of the pointer, measured in pixels from the lower-left corner of the global display screen.
- float transInc, rotInc, scaleInc – the amount of change to apply each time a translation, rotation, or scaling operation is requested.
- float xRotVel, yRotVel, zRotVel – current angular velocity of the scene; when the system has angular velocity, even if no event is generated by the **Mouse**, the current scene will be rotated by the angular velocity amount.
- NameList<NameList<char *> *> userMenuList – a list of **NameList** objects which hold the definitions of user menus. Each name in userMenuList is the title of a submenu to be added to the main pop-up menu, with an associated list of names which define the submenu. This is used to store the user-defined pop-up menu commands.
- <NameList<char *> *mainUserMenu – this is the one user-controlled menu which is always added to the main menu. The user can add two types of items to the main menu:
  - Single commands, which are put into the mainUsermenu list.
  - Submenu commands, which are put into a new submenu with a specified name that is added to the main menu. This allows for grouping of related commands under a separate submenu in the main menu.
- NameList<char *> userKeys – in a manner very similar to adding new menu items, the user can assocate keyboard shortcuts (or hotkeys) with a specified text command. This list stores the keystrokes, and the text commands associated with those keystrokes.

Nonvirtual member functions

- int mouse_moved(int b1Down, int b2Down) – performs the necessary action when the mouse is moved while a button is pressed. The arguments indicate which buttons are currently down.
- void create_key_commands(void) – creates the initial list of keyboard shortcuts; this is only done when the **Mouse** is initialized.
- void stop_rotation(void) – sets the current angular velocity to zero.
- int x(void) – returns the current x position of the **Mouse**, as measured from the lower-left corner. This just calls a similar (but virtual) function in **DisplayDevice**.
- int y(void) – as as x(), but for the vertical position.
- int button_down(int) – returns the current state of the given button.

- `int move_mode(MoveMode, int = 0)` – sets the current mouse mode. If the new mode is `PICKING` or `LIGHT`, the second argument must be given to indicate which picking mode or light. Otherwise, the second argument is ignored.
- `MoveMode curr_move_mode(void)` – returns current mode.
- `char *move_mode_name(MoveMode)` – returns a string describing the specified mode.
- `int curr_pick_mode(void)` – returns current picking mode, or -1 if the current mouse mode is not `PICKING`.
- `int picking(void)` – returns TRUE if somthing is being pickied, or FALSE otherwise.
- `int create_user_menu(char *label)` – indicates that a new user-customizable submenu should be added to the main menu, with the given label. The new menu will be initially empty.
- `int add_user_menu_item(char *text, char *txtcmd)` – adds a new user menu item to the main user submenu ... the name in the menu will be 'text', and the command executed when that menu option is chosen is 'txtcmd'.
- `int add_user_menu_separator(void)` – puts in a separator at the current end of the main user submenu.
- `int add_user_submenu_item(char *submenu, char *text, char *txtcmd)` – adds the command 'txtcmd' to the submenu 'submenu', under the name of 'text' in that submenu.
- `int add_user_submenu_separator(char *submenu)` – adds a separator to the end of the given submenu.
- `int add_user_key_command(char, const char *)` – adds a new keyboard shortcut (or replaces a previous one), for the given key.

Virtual member functions

- `virtual void reset(void)` – resets the user interface to the initial state it was in when it was created.
- `virtual int act_on_command(int, Commmand *, int)` – this is called each time a **Command** is executed that the **Mouse** has expressed interest in hearing about. This will be when a mouse event occurs, such as a mouse button press.
- `virtual int check_event(void)` – calls the necessary routines in **DisplayDevice** to determine if a button has been pressed. If not, then any angular velocity is used to rotate the current scene.

Method of use

Once it is created and given the proper **CommandQueue** and **DisplayDevice**, **Mouse** only needs to have its virtual function `check_event()` called regularly. This is done in the main event loop of VMD.

The **Command** objects which have the word 'user' as their first word control the customization of the pop-up menu and keyboard shortcuts.

Suggestions for future changes/additions

When a new **DisplayDevice** is eventually developed to use X-Windows and OpenGL, there will also need to be included routines there to post and activate the pop-up menu and for all the virtual functions in **DisplayDevice** which the **Mouse** uses. If this is done successfully then **Mouse** should be completely device-independent. It may be necessary (or preferred) to convert to the use of a pull-down menu in the X-Windows case.

27.4 UIObject

Files:	`UIObject.h, UIObject.C`
Derived from:	none
Global instance (if any):	none
Used in optional component:	Part of main VMD code

Description

UIObject is the base class for each object in VMD which provides some form of user interface. There is a **UIObject** subclass for dealing with the text console, for the 2D pointer interface (the **Mouse**), and for the 3D user interface (currently in development). Also, each of the different GUI forms which appear on the screen are a separate **UIObject**.

VMD contains many different **UIObject**, each for the most part independent of the others. A **UIObject** has the ability to get input from the user on what that would like to do, and convert these into particular commands to do that action. These commands (embodied by different subclasses of the **Command** object) are put in a queue for execution. Each **UIObject** must also present some form of visual representation of the state of VMD to the user, which changes as different actions are performed. As commands are executed, each **UIObject** must be informed of what the action was, in order to update its display. No user interface component is capable of executing ALL the commands available in VMD, however, and conversely no user interface component is interested in hearing about all the different possible actions in VMD.

Each **UIObject** has the following characteristics:

- A list of the particular **Command** objects which are interesting to the **UIObject**. When it is initialized, each **UIObject** must call particular functions which store the list of commands which will require some change to the user interface when they are executed. This is usually done in the specialized constructor for the class derived from **UIObject**.
- A unique ID code, which is provided by the **CommandQueue** object with which the **UIObject** registers when it is created.
- A **CommandQueue** variable which is used by the **UIObject** to have **Command**s queued and executed.

The main **UIObject**s in VMD are all created in the routine `VMDinitUI`, in the file `Global.C`.

Constructors

- **UIObject::**`UIObject(`char *, CommandQueue *`)`
  The arguments are the name of the object, and the **CommandQueue** which this should use to register and queue **Command** objects for execution.

Internal data structures

- `char *name` – the name of the object; this is public so that it can be accessed by other objects in VMD.
- `int myID` – the ID code for this object, as returned by the **CommandQueue**.
- `int uiOn` – whether this object is active; if not, it will not check for events from the user.

- `int maxCmds` – the total number of commands which this object can possibly be interested in.
- `char *doCmd` – an array of flags which are used to indicate if a command is wanted or not.
- `CommandQueue *cmdQueue` – the **CommandQueue** that this object should use to queue and execute commands.

Nonvirtual member functions

- `int addcommand(Command *)` – adds the given **Command** instance to the end of the command queue. Returns success.
- `int runcommand(Command *)` – have the command queue execute the given command immediately. Returns the success of the command, or -1 if the command was queued for later execution.
- `void command_wanted(int)` – sets the proper flag in the `doCmd` array to indicate that the given command code is a command the **UIObject** is interested in.
- `void command_not_wanted(int)` – opposite of `command_wanted`.
- `int id(void)` – return the ID of this **UIObject**.
- `int want_command(int)` – return whether the given command is one this object is interested in.

Virtual member functions

- `virtual int is_menu(void)` – return whether this object is an on-screen menu form, or another type.
- `virtual int is_on(void)` – return whether the object is currently active.
- `virtual void On(void)` – turn on the **UIObject**.
- `virtual void Off(void)` – turn off the **UIObject**.
- `virtual void move(int, int)` – moves the **UIObject** the the given X,Y position on the screen, where X and Y are in pixels measured from the lower-left corner of the display. This may not be applicable to all **UIObject**'s, if not is is ignorned.
- `virtual void where(int &, int &)` – return the position of the **UIObject**, if applicable.
- `virtual void init(void)` – initialize the user interface. This is called once at the beginning of VMD, after all the **UIObject**s have been created.
- `virtual void reset(void)` – resets the user interface object, forcing an update of all the informative displays, etc.
- `virtual void update(void)` – updates the display, for example for some visual items which much change each time the scene is redrawn. This is called in the main loop of VMD after all queued **Command**s have been executed, before the scene is actually rendered.

- `virtual int act_on_command(int, Command *, int)` – this is called after a command is executed, and it is seen that this **UIObject** is interested in that command. The first argument is the command code, the second the **Command** itself, and the third is the success of the command. This routine should check what the code is, and based on that update any visual or other such representation of the program to reflect the change caused by execution of the given command. Returns TRUE if this **UIObject** actually did something due to this command being executed.

- `virtual int check_event(void)` – checks whatever external interface is necessary to see if a new command from the user has been entered. If so, this routine creates a new **Command** object, and adds it to the queue.

Method of use

To create a new **UIObject**, you must do the following:

- In the constructor for the new class, call the routine `command_wanted` to indicate the commands you are interested in.

- Provide versions of at least the routines init, reset, update, act_on_command, and check_event.

- Add code to the VMDinitUI routine to have the **UIObject** created at startup.

27.5 UIText

Files:	`UIText.h, UIText.C`
Derived from:	**UIObject**
Global instance (if any):	`uiText`
Used in optional component:	Part of main VMD code

Description

The particular **UIObject** which is responsible for getting text commands from the user, parsing them, and executing them. Each time it is requested to do so, this object will check to see if the user has entered a line of text at the VMD *prompt*, and if so, this will read in the command, break it up into the individual words, and determine what action is being requested. Every action in VMD which the user can request to have performed has a corresponding **Command** object associated with it, and a text command representation.

Each text command consists of N separate words separated by whitespace, with the first word in the command used to distinguish between different types of commands. With each word there is an associated *callback function* which is called when a text command is read in and the first word is found in the list.

This object can also be directed to read commands from a file, and to echo text commands to the console.

UIText is able to use the Tcl library, if available, to parse the text commands and to provide sophisticated interpreted script capabilities such as control loops, variable substitution, if-then-else constructs, and user-defined subroutines and functions. If Tcl is not available, this object will still function, but only the basic VMD commands will be understood.

Constructors

- **UIText::**`UIText`(CommandQueue *)

Internal data structures

- `NameList<TextCallback *> textProcessors` – a list of all the words which this object understands as the first word in a specific command, and the callback function to call when a command with the starting word is entered. `TextCallback` is a typedef for the callback function, as defined in the file `Command.h`.

- `int needPrompt` – whether or not the prompt needs to be printed the next time it is possible to do so.

- `Stack<FILE *> input_files` – a stack containing the FILE structures for the files to read commands from. When a new file is opened for reading, the currently open file is pushed down on the stack and the new file is put at the top of the stack. When the end of the file is reached, the stack is popped and reading from the new top file is resumed. In this way files may be read in a nested fashion. When the stack is empty, input is read from standard input.

- `int doEcho` – flag to indicate if text commands should be echoed back to the console after they have been entered (even if from a file or from the console).

- `Inform txtMsgEcho` – the **Inform** object to use for echoing commands.
- `float delay` – the amount of time to wait (in seconds) before attempting to read the next command. Once the waiting period is over, this is set to zero.

Nonvirtual member functions

- `int num_commands(void)` – the number of words which this object understands as the first word in a command.
- `char *word(int)` – the Nth word that this object understands as the first word in a command.
- `void add_command(char *, TextCallback *)` – adds a new word to the **UIText** vocabulary, and registers the given function as the callback for that word. If the word is already known, this does nothing.
- `int process_command(int argc, char **argv)` – given a tokenized text command (one that has been broken up into 'argc' individual words, with the words in the 'argv' array), this routine checks the first word to see if it is understood, and calls the proper callback function or prints an error message.
- `void read_from_file(char *)` – instructs this object to read text commands from the given file until the end-of-file is reached.
- `void wait(float)` – instructs this object to wait for the given number of seconds before attempting to read in the next text command.
- `int echo(void)` – returns the current echoing status.
- `void echo(int yn)` – turns on/off echoing of text commands.

Virtual member functions

- `virtual int check_event(void)` – checks to see if a text command has been entered at the console or is available from a file. If so, this reads in the string, and queues a 'TextEvent' command (which is a subclass of **Command**) containing the string.
- `virtual int act_on_command(int, Command *, int)` – called after a text event command is executed. This will take the text command string in the text event, break it into tokens, and if the first word is understood this will call the proper callback function. If this first word is not understood, or the callback function returns an error code, this will print an error message.

Method of use

In the constructor for this object, first the code for the 'TextEvent' command is registered as a command this object is interested in, and then all the words this object can understand (with their callback functions) are added to the object's internal list. Each time a new word is added to the VMD vocabulary, a new line must be added in this section to add the callback function (see the discussion of **Command**, section 27.1). The callback functions themselves should be declared in the Cmd*.h files, and defined in the Cmd*.C files.

When a text command is actually to be executed, is it broken into an argc, argv tokenized format, and given to the `process_command` routine. This searches for the first word in the internal list of known words, and calls the callback function with the argc, argv pair, and also the **CommandQueue** object to use for queuing new commands as well as the ID of this **UIObject**.

In the case where Tcl is being used, there is a slightly different order to these events. After a string is read, it is given to the Tcl evaluation routine, which checks it for special Tcl commands or for VMD commands. When VMD commands are seen, Tcl will tokenize the command itself and call a particular Tcl-tailored callback function. This callback function is registered with Tcl for every word in the VMD vocabulary. In this Tcl callback function, the first word is checked, and the regular text callback function is called.

Suggestions for future changes/additions

The reading of commands from files should be converted from the `stdio` style I/O to the use of the `streams` library.

28 Tracker objects

The objects responsible for controlling the external spatial tracking devices, and for displaying and using the 3D pointers, are currently in the experimental stage, and will be described later when their design is closer to being final.