ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, IL 60439

ANL/MCS-TM-216

MolView Users Guide

by

Brian P. Walenz

Mathematics and Computer Science Division

Technical Memorandum No. 216

June 1996

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38, and by the Argonne Director's Individual Investigator Program.

Contents

MolView Users Guide

by

Brian P. Walenz

Abstract

A system for viewing molecular data in a CAVE virtual reality environment is presented. The system, called MolView, consists of a frontend driver program that prepares the data and a backend CAVE program that displays the data. Both are written so that modifications and extensions are relatively easy to accomplish.

1 Introduction

MolView is an extensible system for viewing molecular data in the CAVE virtual reality environment. The user can view the molecule as if it were a physical model, rotating and viewing the object from different angles.

The MolView system consists of a backend CAVE program, and several routines that allow a frontend driver program to easily send data to the CAVE program. All communication is done via MPI, allowing the driver to be executed on a wide variety of machines.

MolView is capable of storing and displaying an arbitrary number of datasets, each of which may have an arbitrary number of time steps, or frames. Thus, there are four general types of data that can be viewed:

- Single dataset, single time step
- Single dataset, multiple time steps
- Multiple datasets, single time step per dataset
- Multiple datasets, multiple time steps per dataset

MolView can currently only display data. There are no mechanisms to allow the CAVE user to modify a dataset and send it back to the driver program for later computation.

The remainder of this document examines the execution of MolView applications (Section 2), two example driver programs, *ion* and *pdb* (Section 3), and the CAVE interface (Section 4).

Section 5 introduces the programming level details of MolView. Sections 6 and 7 explain how to create custom drivers and modify the CAVE program to accept new data from custom drivers

2 Execution

It is the responsibility of the driver program to supply the CAVE with data to show. How a driver generates the data is arbitrary; some drivers simply read a data file, whereas others compute the solution to a problem in real time.

For drivers that execute on a machine that the display trusts * execution is straightforward:

```
MolView driver options [...]  displayname
```

Here, `driver` is a path to the driver executable that is to be used, `options` are the options to the driver, and `displayname` is the name of the display that the CAVE program is executed on, chosen from one of four forms, as shown below:

Display Type	*Action*
local	Displays on the local monitor, equivalent to using ":0.0"
machine:display.screen	Displays on a remote monitor
cave	Starts the CAVE
idesk	Starts the ImmersaDesk

For example, to run the `pdb` driver program in `bin` to show all the pdb files in ~/proteins and display them on the display attached to `vogon`, cd to the MolView home directory and type

```
MolView bin/pdb ~/proteins/* vogon:0.0
```

The display type `cave` will perform all the magic that is necessary to start MolView in the CAVE; check first that the CAVE is free for use. The display type `idesk` will start MolView and show the display on the ImmersaDesk; you must ensure that the ImmersaDesk is free and take the projector out of standby mode. † Both `cave` and `idesk` are Argonne specific. See the `MolView` script itself for low-level execution details.

2.1 Advanced Execution

When the driver is executed on a machine that is not trusted, or is a parallel program, the `MolView` script will not work. The first problem is solved by making the machine trusted or by using a secure server. Consult your support staff or a local MPI guru for help.

*An easy way to tell if machine A trusts machine B is to attempt a remote shell from machine B to machine A, for example, `rsh A date`.

†Point the projector remote control at the ImmersaDesk screen and press and hold the "standby" button. Repeat to turn off—just be sure to hold the button down!

```
local               0
spnode2             1  /sphome/walenz/ION/ion.x
spnode3             1  /sphome/walenz/ION/ion.x
spnode4             1  /sphome/walenz/ION/ion.x
spnode5             1  /sphome/walenz/ION/ion.x
alaska.mcs.anl.gov  1  /afs/fl/home/walenz/ION/bin/StartMolViewCave
```

Figure 2.1: Sample `procgroup` file using 5 SP nodes and showing the results in the CAVE

When the driver is executed in parallel on a trusted machine, startup will need to be handled explicitly. This process depends heavily on which communication device is being used, and that depends on which machine the driver is being run on. The next two sections explain two common methods at Argonne.

2.1.1 MPICH, p4 Device

The MPICH communication method exists for most machines and is relatively easy to use. First, a `procgroup` file must be created where the driver is executed. This file contains a list of machines and the program they should run. A sample `procgroup` file is shown in Figure 2.1. The first column contains the name of the machine, the second is the number of processes to run on this machine, and the last is the executable to run. The first entry, `local 0` says to run 0 *additional* copies on the current machine.

The p4 device uses Unix sockets to perform the actual communication. This strategy allows it to run on a wide variety of machines, but also means that performance is not optimal.

Once the `procgroup` file is created, the application is launched with `driver -p4pg procgroup`. Note that this will fail if the machines listed do not trust the local machine, usually returning `permission denied`.

2.1.2 Nexus

For better performance when using a multiprocessor driver on the IBM SP, Nexus MPI may be used. Instead of using p4 for all communication, Nexus uses MPL for communication between SP nodes, and p4 between the SP and the CAVE.

Like MPICH p4, Nexus needs a list of what to run. Since two different methods of communication are being used (MPL and p4), two different lists of processors are needed. Jobs started using the MPL startup routines are specified in much the same way that normal MPL jobs are specified; the user defines a set of envrionment variables.

For machines that are not started via the MPL startup routines, Nexus consults a database file (Figure 2.2). Like the p4 `procgroup`, this file tells Nexus what executable

```
alaska.mcs.anl.gov \
    startup_dir=/afs/fl/home/walenz/WORK/ION/bin \
    startup_exe=StartMolViewCave
flying.mcs.anl.gov \
    startup_dir=/afs/fl/home/walenz/WORK/ION/bin \
    startup_exe=StartMolViewIdesk
```

Figure 2.2: Sample Nexus database file

to run on various machines. The example in Figure 2.2 has two entries: the CAVE on `alaska.mcs.anl.gov`, and the ImmersaDesk on `flying.mcs.anl.gov`. Which one is used depends on the command line used to start the jobs.

```
#!/bin/sh

MP_HOSTFILE=/sphome/$LOGNAME/SPnodes.`getjid`
MP_PROCS=`cat $MP_HOSTFILE | wc -l`
MP_PULSE=0
MP_EUILIB=us

export MP_HOSTFILE MP_PROCS MP_PULSE MP_EUILIB

ion.x -mpi -dbfile ~/SP/demo.rdb -nodes alaska.mcs.anl.gov -nonameexpand
```

Figure 2.3: Sample Nexus startup script

The script in Figure 2.3 will start the multiprocessor driver on the SP and the CAVE on `alaska.mcs.anl.gov`.

The magic behind Nexus startup is in the command line ("`ion.x -mpi ...`"):

- `-mpi` tells Nexus that the remaining arguments are for it.
- The Nexus database file is `/sphome/walenz/SP/demo.rdb`.
- In addition to the MPL job startup, start a job on `alaska.mcs.anl.gov`. The parameters for the job are defined in the database file.
- Do not use the name `ion.x` when starting the alaska.mcs.anl.gov job.

3 Driver Programs

3.1 PDB Driver

The PDB driver `pdb` will accept several input `.pdb` files, which can be compressed (names ending with ".gz" or ".Z") or uncompressed (names ending with anything else). The program will parse the file and send a list of atoms to the CAVE that are sized and colored as follows:

Atom Type	*Color*	*Relative Size*
Nitrogen	Blue	0.58
Hydrogen	White	0.58
Carbon	Green	0.73
Sulfur	Yellow	0.90
Oxygen	Red	0.50
Unknown	Dark Gray	0.25

To change these values modify the `pdb` driver code and recompile.

Note: No bonds are shown between atoms.

Because of speed considerations, only the first 2000 atoms are used. If more than 2000 atoms are present in a given file, a warning message is printed, and the rest are ignored. This approach is taken because showing large numbers of atoms is exceptionally slow.

The `pdb` driver accepts any number of command line arguments. All of them are treated as names of `.pdb` files to show.

3.2 Ion Driver

The ion driver `ion` will read in a data file of ion positions, process the data to determine a shell structure, color each shell differently, and send this processed molecule to the CAVE.

The ion driver allows all four types of execution described in the introduction.

The general format of the data file is shown in Figure 3.1, and a sample input file is in Figure 3.2.

The `ion` driver has one mandatory command line argument, the name of the data file to show. An optional argument, -d, forces `ion` to use a distance-based coloring scheme rather than the default shell-based scheme, and is useful for viewing data sets with more than one frame that do not have a shell structure—the visualization of a minimization procedure, for example. It prevents, (or tries to prevent) the color scheme from changing rapidly while viewing the minimization process. In some cases, it will come close to coloring each shell differently; in others, a single shell might have two different colors. The -d argument *must* be the first argument on the `ion` command line.

```
{ NAME OF MOLECULE
  NUMBER OF ATOMS IN MOLECULE, n
  { "FRAME"
    ENERGY OF THIS CONFIGURATION
    LIST OF x, y, z POSITIONS OF n ATOMS
  }
  "END_OF_MOLECULE"
}
```

Figure 3.1: General ion input file format. The brackets represent blocks in the input file. Each block must be included at least once. There is no limit on the number of blocks

```
OneFrame
5
FRAME
.33057547E+02
  -.65301218E+00     .57223214E+00    -.12000438E+01
   .10593340E+01     .56625598E+00     .86668641E+00
   .48532373E+00     .13598468E+01    -.39817273E+00
  -.14570831E+00    -.96484823E+00    -.11143683E+01
  -.26061340E+00    -.17363993E+00     .14477256E+01
END_OF_MOLECULE
TwoFrames
4
FRAME
.39404080E+02
  -.11071736E+01     .51080004E+00    -.94137694E+00
   .47903853E+00     .55072851E+00     .13601102E+01
  -.10358628E+00    -.84134423E+00     .12899959E+01
   .12626000E+01     .92686556E+00     .81957154E-01
FRAME
.46088283E+02
  -.10480169E+01    -.12001274E+01     .14769891E+00
  -.22749490E-01     .80373052E+00    -.13834594E+01
   .59944587E+00    -.14829747E+01    -.43713973E-01
   .22749490E-01    -.80373052E+00     .13834594E+01
END_OF_MOLECULE
```

Figure 3.2: Sample data file for the ion driver

4 The CAVE Interface

MolView uses the CaveMenu system [4]. Briefly, interaction with the menus is done by pointing the wand at a menu gadget and pressing the third (right) button.

The menus in MolView are grouped by function. Each menu is callable from the main menu by selecting the appropriate button.

4.1 Main Menu

The main menu (Figure 4.1) consists of a few buttons to call other menus (options, frame), buttons to toggle modes (rotation, translation), a slider to change what data set is shown, and the "Quit" button.

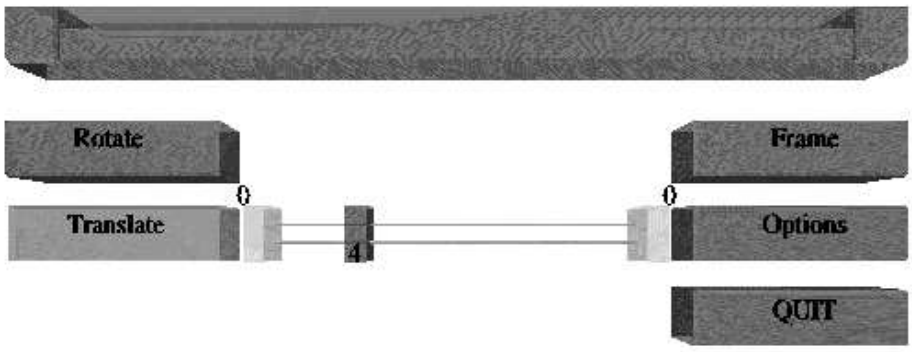

Figure 4.1: The main menu

Selecting *Options* or *Frame* will bring up the options menu (Section 4.2) or frame menu (Section 4.3), respectively.

Rotate and *Translate* toggle the ability to rotate and translate the molecule in the CAVE. When *Rotate* is selected, pressing the left wand button and rotating the wand will rotate the molecule. If the left button is released while the wand is still rotating, the molecule will continue to rotate at a constant rate. To stop the molecule from rotating, either hold the wand steady and release the left button, or turn off the *Rotate* menu button. When *Translate* is selected, holding the middle wand button and moving the wand will move the molecule in the CAVE.

The main menu also contains a slider gadget that allows you to select the dataset to view. Clicking on the slider knob allows you to drag the knob, while clicking on the slider beams will snap the slider knob to that position. Clicking on the ends of the slider bars will move the knob in that direction one step.

Selecting *Quit* will quit the application.

4.2 Options Menu

The options menu (Figure 4.2) has three sliders. From the top, there are sliders to change the detail level of atoms, size of atoms, and size of the molecule.

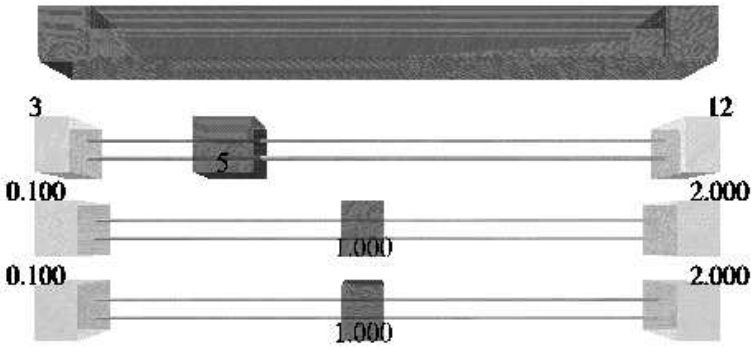

Figure 4.2: The options menu

The top slider controls how smooth the atoms appear. By increasing the detail, more polygons are used to draw each sphere. Be forewarned; beauty comes at a large price. The CAVE can draw only a finite number of polygons per second, so increasing the detail level will directly affect the response time. The result might not be so terrible when using the CAVE simulator, but when in the CAVE every move will (usually) change the viewing angle. If the response time is very high, the CAVE will appear to be jumpy; if the response time is low, the CAVE can smoothly keep up with your movements, giving a more realistic three-dimensional illusion.

The middle slider changes the size of all atoms in the molecule. This can be used to transform the molecule to a space-filling model.

The bottom slider scales the molecule, either bringing the atoms closer together or spreading them farther apart. This is useful for making large molecules manageable and small molecules large enough to see. Note that changing the size of the molecule not change the size of the atoms.

4.3 Frame Menu

For molecules with more than one frame, the frame menu (Figure 4.3) allows you to cycle through all the frames in the dataset.

Two methods exist for viewing other frames: using the slider, or clicking on *Reverse* or *Forward*. By using the slider, you can quickly view the entire sequence of frames, but using *Reverse* or *Forward* will iterate through the sequence of frames like a movie. Toggling *Cycle* will let *Forward* and *Reverse* loop from end to end.

Faster and *Slower* change the speed that *Reverse* and *Forward* iterate through the frames.

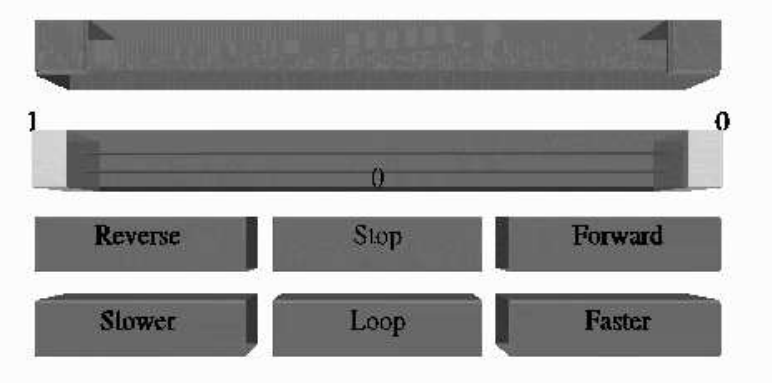

Figure 4.3: The frame menu

To stop playback, either turn off *Reverse* or *Forward* (whichever is on), or select *Stop*.

5 For the Programmer

The remainder of this document describes MolView from a programmer's perspective. First, the directory structure and compilation of MolView are explained. Next, driver responsibilities and driver creation are discussed. Finally, the internals of the CAVE component of MolView and techniques for extending MolView to show different styles of data are described.

5.1 Directory Structure

The directory structure is simple:

bin	Binaries and the MolView execution script
drivers	Driver source code
include	Header files
lib	Driver library files—libMolView.a
obj	Object code from building MolView
src	MolView CAVE and library source code

In `include` are the following:

MolView.h	Datatype definitions
MolViewCave.h	CAVE internal datatypes

In `src` are the following:

CleverMath.C	Some mathematics
Datatypes.c	Handles the creation of MPI datatypes for sending and receiving data
Display.C	The CAVE draw routine
DisplayList.C	User display list creation function
Initialize.C	Initialization and memory handing routines
Menus.C	Menu creation and handling
Receive.C	Routines for receiving data from driver programs
Remote.c	Routines for sending data from drivers to the CAVE
Cave.C	Main CAVE routine

5.2 Building MolView

To build MolView, one simply types `make` from the MolView root directory. This will build `MolViewCave` in `bin/`, `libMolView.a` in `lib/`, and any drivers in `drivers/`. If the build is being done on a multiprocessor machine, a parallel build will be done if the `PARALLEL` environment variable is set to the number of steps to perform concurrently.

The command `make clean` will remove the object files, and `make distclean` will remove everything that is generated by `make`, including libraries and executables.

6 Driver Responsibility and Creation

Driver programs are responsible for one important task—creating data for the CAVE to show. This task comprises three pieces, although the line between each piece is fuzzy.

1. Create the data. This task can be done by the driver program, for example, by embedding the MolView system into a prewritten computational program. Alternatively, the driver program can just read a data file.

2. Massage the data into the MolView data structures (Section 6.2).

3. Transfer the data to the CAVE (Section 6.3).

These do not need to be distinct steps and, in some cases, must not be distinct. The simplest example here is a driver program that computes the data using MolView data structures: computation and packaging step are merged. An example merging all three steps is a driver that minimizes the energy of a molecule and sends the molecule to the CAVE after each minimization step.

6.1 Driver Initialization

To ease the pain of creating a multiprocessor driver, *every* driver (even uniprocessor!) must call `MV_SplitCommunicator()` (or `MVF_SplitCommunicator()` if in Fortran) immediately after calling `MPI_Init()`. This routine will create a new MPI communicator consisting of all the nodes that are driver nodes, allowing the driver to do collective communication between driver processors only.

Failure to call this routine will result in the CAVE locking up immediately after startup.

6.2 Data Formats

Three data structures exist for driver to CAVE communication: `struct MoleculeData`, `struct FrameData`, and `struct AtomData`, all of which are defined in `include/MolView.h`.

The `struct MoleculeData` data structure contains book-keeping information necessary to tell the CAVE about each molecule. `MoleculeID` is how a driver refers to a CAVE molecule and thus should be unique. The CAVE stores molecules in increasing `MoleculeID` order.

A description string is provided; however, its use is not mandatory. If a description is not needed, the string should the set to zero length.

```
struct MoleculeData {
  int        MoleculeID;
  char       Desc[81];
};
```

The `struct FrameData` data structure is an atom-level description of the molecule. Since frames are added independently of molecules, each frame needs both a `FrameID` and a `MoleculeID`. The `FrameID` allows the CAVE to order the frames, while the `MoleculeID` tells the CAVE which molecule this frame is a part of.

`NumAtoms` is the number of atoms that this frame contains. No atoms are stored in `FrameData`; instead, they are passed in separately to `MV_AddFrame()` (Section 6.3.2).

`Center` is the coordinate that the CAVE uses to rotate about. To have the molecule rotate about the center of mass, `Center` should be set the the center of mass. Likewise, to rotate about a specific atom, `Center` should be the location of that atom.

```
struct FrameData {
  int        FrameID;
  int        MoleculeID;
  int        NumAtoms;
  float      Center[3];
};
```

```
void        MV_AddMolecule(MoleculeData *M)
void        MV_AddFrame(FrameData *F, AtomData *A)
void        MV_Exit(void)
```

Figure 6.1: C driver to CAVE interface

```
MVF_AddMolecule(MoleculeID)
            integer MoleculeID
MVF_AddFrame(MoleculeID, FrameID, NumberOfAtoms, Center
                    AtomPositions, AtomColors, AtomSizes)
            integer MoleculeID, FrameID, NumberOfAtoms
            double precision(3) Center
            double precision(*) AtomPositions, AtomColors, AtomSizes
MVF_Exit()
```

Figure 6.2: Fortran driver to CAVE interface

The `AtomData` data structure contains a complete description of a single atom in the molecule. At the very least, this description must contain the position, color and size of the atom.

```
struct AtomData {
  float      Position[3];
  float      Color[4];
  float      Size;
};
```

6.3 Data Transmission

A driver program uses three routines to communicate with the CAVE: one to create a molecule, one to add a frame to a previously created molecule, and one to tell the CAVE that the driver is done. Since Fortran is not able to use C structures, separate routines exist for C (Figure 6.1) and Fortran (Figure 6.2).

For low-level details, see the code in `Remote.c` and `Datatypes.c`, as well as Section 7. This section explains only how to use the routines in a driver program and assumes that the required data structures are completely and correctly created.

Several sample drivers are provided in `drivers/`; `simpledriver.c`, `framedriver.c`, and `simplefort.f`.

6.3.1 MV_AddMolecule()

`MV_AddMolecule()` requests that the CAVE allocate space for a new molecule with ID `MoleculeID`. No frames are transmitted or allocated.

From Fortran, it is not possible to pass a description string to the CAVE.

6.3.2 MV_AddFrame()

`MV_AddFrame()` adds a frame to the molecule with ID `MoleculeID`. If the molecule has not been created, the CAVE will print an error message and fail.

The atom list does not need to be sorted, but several optimizations can be made if it is. See the code in `DisplayList.C` for ideas.

The choice of `double precision` in the Fortran interface is reasonably arbitrary. Look in `Remote.c` for details on what needs to be changed to use `real`.

6.3.3 MV_Exit()

This routine should be called when the driver is done sending data to the CAVE. Once called, the driver may proceed with any cleanup it needs to do, then call `MPI_Finalize()`, and exit. The CAVE will continue to allow the user to view the data.

7 CAVE Program Modification

The structure of the CAVE portion of MolView roughly consists of two parts: initialization and communication. Everything else is handled through callback routines by the CaveMenu or CAVE library.

Them main loop (in `cave.C`) performs several initialization tasks:

- Starts MPI, and creates a driver-only communicator.
- Allocates a large chunk of shared memory for the menus and MolView data.
- Initializes the basic MolView data structures.
- Creates the menus.
- Configures the CAVE, and sets the callback routines.

Once initialized, MolView enters a loop where it receives data from the driver until either the `ESC` key is pressed or the driver announces it is done.

The behavior of MolView is changed by modifying the MolView data structures and then modifying the various callback routines that act on the data structures. The callback routines that need to be modified are in:

`Datatypes.c`	MPI datatypes.
`Remote.c`	Driver to CAVE transfer functions.
`Receive.C`	CAVE receive functions.
`Display.C`	CAVE display list generation routine.
`Menus.C`	The Menu system.

Such modifications, even for moderate extensions, are not excessively involved. As an example, look through the ION code. All the "shell" operations are essentially an extension to the base MolView system.

Note: Referring to the code while reading this section will greatly enhance comprehension.

7.1 MolView Structures

The MolView structures specific to the CAVE contain an instance of the appropriate driver structure (`CAVEAtom` contains `AtomData`, for example) and any additional storage that the CAVE needs.

`CAVEAtom` currently does not contain any additional information. Later, for example, if one wishes to draw sticks to connect atoms, an array of pointers to other `CAVEAtom`s could be added.

Note that if the `AtomData` data structure is modified, `ReceiveFrame()` must be modified as well. See Section 7.4.2 for an explanation.

```
struct CAVEAtom {
  AtomData          Data;
};
```

`CAVEFrame` contains a list of the atoms in this frame and a pointer to the next frame in the sequence.

```
struct CAVEFrame {
  FrameData         Data;
  CAVEAtom         *Atoms;
  CAVEFrame        *next;
};
```

`NumFrames`, `CurrentFrame`, `cF`, and `Frames` are used to keep track of which frame is currently being shown, and should be included in all derived data structures.

```
struct CAVEMolecule {
```

```
  MoleculeData     Data;
  int              NumFrames;
  int              CurrentFrame;
  CAVEFrame       *cF;
  CAVEFrame       *Frames;
  CAVEMolecule    *next;
};
```

7.2 MPI Datatypes

The routines in `Datatypes.c` are responsible for informing MPI about the data that one intends to send. These routines are critical. If they are incorrect, MPI will (probably) send junk data to the CAVE.

Basically, `MD_Create*Datatype()` determines the size and relative position of every *block* of data. A block of data is any number of structures that all have the same type, so four `int` variables in a row is one block.

When modifying the MolView structures, one simply modifies `MD_Create*Datatype()` so that the blocks of data are specified. For more information, see an MPI manual such as [1, 2].

7.3 Driver to CAVE Transfer Functions

The driver to CAVE transfer functions are in `Remote.c`. They create the MPI datatype(s) needed and call `MPI_Send()`. All the work is done by the MPI datatype. These functions probably will not need to be modified.

7.4 CAVE Receive Functions

The CAVE receive functions in `Receive.C` are responsible for receiving the data from a driver and placing it in the CAVE data structures. While doing this, they could optimize the data, for example, taking array indices and translating them to C pointers.

7.4.1 ReceiveMolecule()

The molecule receive function should need very little attention. The molecule data structure is responsible for holding all the CAVE related book-keeping information. Any user-defined data generally is held in `struct MoleculeData`, which is received directly to the CAVE data structure.

7.4.2 ReceiveFrame()

The frame receive function is not nearly as friendly as the molecule receive function. Atoms are stored in the frames as an array. `CAVEFrame` contains an array of `CAVEAtom` to allow for CAVE local data in an atom (a list of pointers to other atoms, for example). Since `CAVEAtom` and `AtomData` generally will be different, atom data must be copied manually to the array in `CAVEFrame`. Thus, *any* changes to `struct CAVEAtom` must be propagated through to `ReceiveFrame()`.

7.5 Display Routine

Modification of the display routine should consist just of changing the display list creation routine, `ComputeDisplayList()`, in `DisplayList.C`. Anything that is legal in an OpenGL display list is legal here, but everything that is done here will be treated as part of the molecule—they will rotate and translate with the molecule. If the behavior of the molecule needs to be modified, then `Draw()` in `Display.C` will need to be modified.

Note: since each processor must have its own local copy of the display list, each processor must call `ComputeMoleculeDisplayList()` whenever the molecule changes. Failure to do so will result in the correct molecule being displayed on one CAVE wall, and something else on other CAVE walls.

See the OpenGL Programming Guide [3] for details on display list creation.

7.6 Menu System

Menu routines are in `Menus.C` and can be modified according to [4].

Sliders merit special attention: they must be updated whenever the values they are associated with change. Currently, they are updated whenever something happens, not just when their value changes. Ambitious programmers could figure out when the sliders must be updated, and only update there, but most programmers will realize that the overhead for updating constantly is low.

Also meriting attention is `RemakeMolecule()`. This function tells the display processes to create a new display list, in addition to updating sliders.

References

[1] M. P. I. Forum, *MPI: A message-passing interface standard*, International Journal of Supercomputing Applications, 8 (1994).

[2] W. Gropp, E. Lusk, and A. Skjellum, *USING MPI Portable Parallel Programming with the Message-Passing Interface*, The MIT Press, Cambridge, 1994.

[3] M. W. J. Neider, T. Davis, *OpenGL Programming Guide*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1993.

[4] B. Walenz. Unpublished information, Argonne National Laboratory, 1996.