Available online at www.sciencedirect.com

SCIENCE DIRECT

Computer Physics Communications 163 (2004) 53–64

Computer Physics Communications

www.elsevier.com/locate/cpc

# Scalable and portable visualization of large atomistic datasets [☆]

Ashish Sharma [*], Rajiv K. Kalia, Aiichiro Nakano, Priya Vashishta

*Collaboratory for Advanced Computing and Simulations, Department of Computer Science, Department of Physics & Astronomy, Department of Material Science & Engineering, University of Southern California, Los Angeles, CA 90089-0242, USA*

Received 15 June 2004; accepted 8 July 2004

Available online 16 September 2004

## Abstract

A scalable and portable code named Atomsviewer has been developed to interactively visualize a large atomistic dataset consisting of up to a billion atoms. The code uses a hierarchical view frustum-culling algorithm based on the octree data structure to efficiently remove atoms outside of the user's field-of-view. Probabilistic and depth-based occlusion-culling algorithms then select atoms, which have a high probability of being visible. Finally a multiresolution algorithm is used to render the selected subset of visible atoms at varying levels of detail. Atomsviewer is written in C++ and OpenGL, and it has been tested on a number of architectures including Windows, Macintosh, and SGI. Atomsviewer has been used to visualize tens of millions of atoms on a standard desktop computer and, in its parallel version, up to a billion atoms.

## Program summary

*Title of program:* Atomsviewer
*Catalogue identifier:* ADUM
*Program summary URL:* http://cpc.cs.qub.ac.uk/summaries/ADUM
*Program obtainable from:* CPC Program Library, Queen's University of Belfast, N. Ireland
*Computer for which the program is designed and others on which it has been tested:* 2.4 GHz Pentium 4/Xeon processor, professional graphics card; Apple G4 (867 MHz)/G5, professional graphics card
*Operating systems under which the program has been tested:* Windows 2000/XP, Mac OS 10.2/10.3, SGI IRIX 6.5
*Programming languages used:* C++, C and OpenGL
*Memory required to execute with typical data:* 1 gigabyte of RAM
*High speed storage required:* 60 gigabytes
*No. of lines in the distributed program including test data, etc.:* 550 241
*No. of bytes in the distributed program including test data, etc.:* 6 258 245
*Number of bits in a word:* Arbitrary

[☆] This paper and its associated computer program are available via the Computer Physics Communications homepage on ScienceDirect (http://www.sciencedirect.com/science/journal/00104655).

[*] Corresponding author.
*E-mail address:* anakano@usc.edu (A. Sharma).

0010-4655/$ – see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.cpc.2004.07.008

*Number of processors used:* 1
*Has the code been vectorized or parallelized:* No
*Distribution format:* tar gzip file
*Nature of physical problem:* Scientific visualization of atomic systems
*Method of solution:* Rendering of atoms using computer graphic techniques, culling algorithms for data minimization, and levels-of-detail for minimal rendering
*Restrictions on the complexity of the problem:* None
*Typical running time:* The program is interactive in its execution
*Unusual features of the program:* None
*References:* The conceptual foundation and subsequent implementation of the algorithms are found in [A. Sharma, A. Nakano, R.K. Kalia, P. Vashishta, S. Kodiyalam, P. Miller, W. Zhao, X.L. Liu, T.J. Campbell, A. Haas, Presence—Teleoperators and Virtual Environments 12 (1) (2003)].

*PACS:* 07.05.Rm; 02.70.Ns

*Keywords:* Visualization; Hierarchical view frustum culling; Probabilistic occlusion culling; Multiresolution algorithm; Molecular dynamics

## 1. Introduction

Recent developments in simulation algorithms and parallel computing technologies have enabled large atomistic simulations involving million-to-billion atoms. For example, some of the largest molecular dynamics (MD) simulations of materials have involved multibillion atoms [1–3]. In these large-scale MD simulations, explorative visualization of simulation datasets is essential for understanding atomistic mechanisms underlying macroscopic material processes [4,5]. Output data from an MD simulation consists of a time series of atomistic data frames. Each data frame is a table consisting of $N$ rows for $N$ atoms, and the $i$th row $R_i$ is a tuple of atomistic attributes of the $i$th atom. As a specific example, the tuple could be

$$R_i = (\lambda_i, \mathbf{r}_i, \mathbf{v}_i, \sigma_i), \tag{1}$$

where $\lambda_i \in \{\mathrm{C}, \mathrm{Si}, \ldots\}$ is the chemical species of the $i$th atom, $\mathbf{r}_i \in \Re^3$, $\mathbf{v}_i \in \Re^3$, and $\sigma_i = (\sigma_i^{xx}, \sigma_i^{yy}, \sigma_i^{zz}, \sigma_i^{yz}, \sigma_i^{zx}, \sigma_i^{xy}) \in \Re^6$ are its 3D coordinate, 3D velocity, and 6-component stress tensor at the simulation time step this data frame is taken from, see Fig. 1. Macroscopic material properties result from complex spatio-temporal interactions among these atomistic attributes, and navigating inside the simulated material, while animating the data frames with atoms color-coded according to one of these attributes, often helps scientists to uncover causal relationships among them.

A major challenge in navigating through a high-end MD simulation dataset is to achieve interactive speed. Although there are quite a few excellent programs to visualize atomistic datasets (e.g., VMD—Visual Molecular Dynamics [6]), we are not aware of any capable of rendering multimillion atoms within a fraction of second. In the past years, we have been developing a scalable and portable software named Atomsviewer for visualizing large atomistic datasets from materials simulations involving billions of atoms. Atomsviewer can visualize tens of millions of atoms on a standard desktop computer, with extensive use of multilevel and probabilistic algorithms, and its parallel version has been used to visualize up to a billion atoms [7].

This paper provides a technical overview of Atomsviewer, the source code of which is disseminated through the Computer Physics Communications Program Library, and documentation describing its usage is contained in the appendices. Section 2 presents an overview of Atomsviewer, and its different algorithms are described in Section 3. Numerical test results are presented in Section 4, and Section 5 contains the summary. Instructions on compiling and building Atomsviewer, is contained in Appendix A.

## 2. System overview

Visualization of large-scale MD simulations is a nontrivial problem because of the large size of the

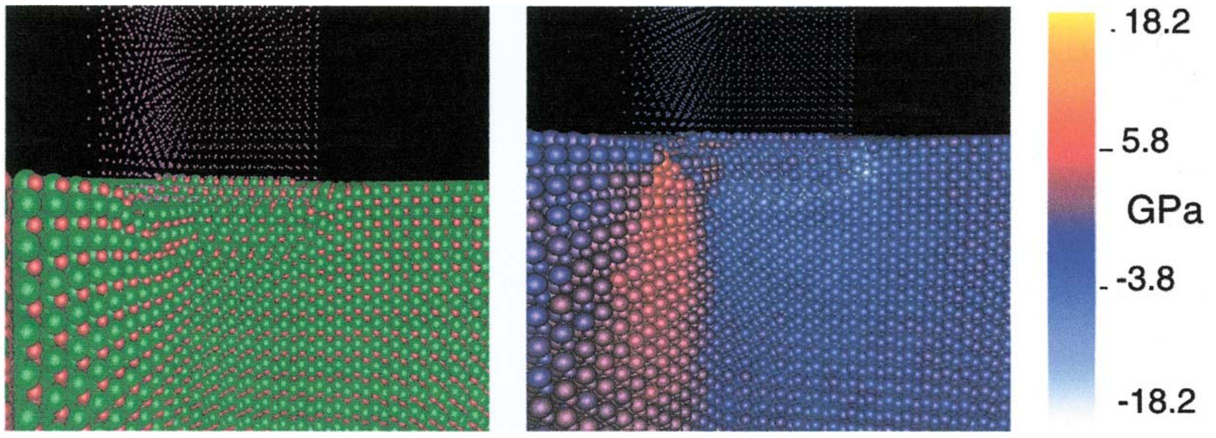

Fig. 1. Visualization of a data frame from an MD simulation of nanoindentation in crystalline silicon carbide (3CSiC), in which a quarter of the system is cut out to make atoms inside the SiC substrate visible [8]. Dots, small spheres, and large spheres represent indenter, Si, and C atoms, respectively. Atoms are color-coded with: (left) species (magenta—indenter, red—Si, green—C); and (right) direction-averaged shear stress, $\sigma_s = (\sigma_{yz} + \sigma_{zx} + \sigma_{xy})/3$.

dataset. For example, an example data frame in Eq. (1) for a 10 million-atom MD simulation contains, for a given time step, 130 million numbers. The large size makes it hard for a single system to store, let alone process, data at speeds that would permit interactive rendering. Rendering millions of atoms becomes harder, since each atom, rendered as a sphere, uses approximately 50 triangular primitives. This is a problem shared by the entire computer graphics community, where the dataset to be rendered almost always exceeds the capabilities of the rendering system. This problem is however overcome by following a simple rule, i.e., "avoid processing data that cannot be seen from the viewer's current perspective". Various algorithms have been developed following this rule. In Atomsviewer, these include frustum-culling and probabilistic/depth based occlusion-culling algorithms, each using the octree data abstraction. Additional user defined filters allow the user to select atoms that satisfy a certain criterion. Finally we have an architecture that provides the user with multiple viewpoints, enhancing the user experience. Fig. 2 shows the breakdown of Atomsviewer into its respective components and the interaction amongst them. In this figure we show the two main applications—FileConverter and Atomsviewer. The FileConverter program reads a plain text file and converts it into a format, in which the atoms are spatially and chronologically clustered. The Atomsviewer program accepts the user position and orientation, and feeds this information to the Data Extraction Module (DEM). The DEM traverses the data file and extracts spatial clusters that fit in the user's field-of-view. These clusters are then passed through user defined coarse and fine grain filters that refine the data. This data is then filtered in the Occlusion Culling Module to remove those atoms that are hidden from the viewer on account of other atoms occluding them. Finally the Graphic Modules render the atoms as spheres of varying size and resolution. The user can also capture the rendered scene as a snapshot or a series of time-stamped snapshots to make a movie.

## 3. Data extraction and rendering algorithms

Visualizing a large atomistic dataset requires a fast and scalable data management system. While the dataset is large, what the user sees at any given instant is a relatively small subset. This observation can be exploited in designing a data management scheme that efficiently extracts atoms that are in the user's field-of-view. In computer graphics, the field-of-view is defined with a truncated pyramidal shape called frustum, and view frustum culling refers to the process of removing atoms that lie outside of it. After frustum culling, atoms that are hidden from the viewer by other atoms are removed by a process called occlusion

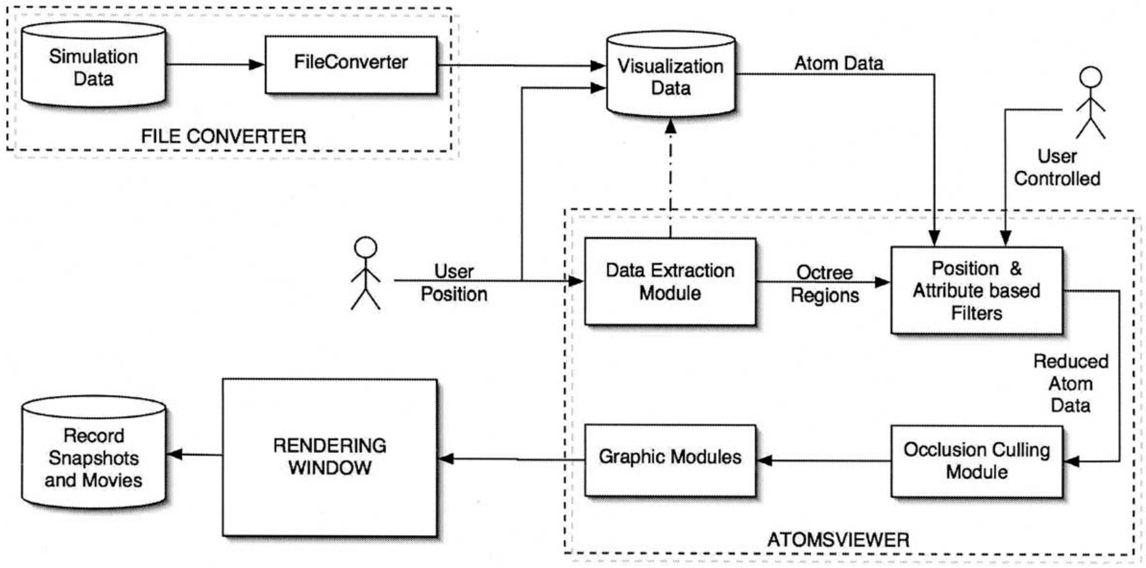

Fig. 2. A schematic of the Atomsviewer program showing its main components and the flow of information among them.

culling. At this stage the atoms are ready to be rendered as spheres. However not all atoms are rendered at the same resolution. Atoms that are far from the viewer will be rendered as coarse spheres, pyramids, and even points. The decision about the resolution of the rendered atoms is known as selecting the Level of Detail (LOD). The following subsections describe the sequence of rendering algorithms used in Atomsviewer.

### 3.1. Hierarchical view frustum culling

To achieve interactive speed and scalability for view frustum culling, spatially close atoms are clustered to provide a convenient abstraction. The clustering of atoms is performed hierarchically, using an octree data structure [9,10]. An octree is a three-dimensional extension of a binary search tree, generated by recursively subdividing the three-dimensional space into smaller subregions, see Fig. 3. Each octree node thus becomes an abstraction of the atoms contained in its subspace, and we can transform the process of extracting atoms to a process of extracting regions that lie in the frustum. This transformation improves the system performance as follows. The total computation time is a sum of the time taken in culling and the time taken to render the resulting reduced set of atoms. For a typical visualization the number of rendered atoms is nearly independent of the user's viewpoint. The culling time however scales linearly with the total number of atoms, and with the octree abstraction, the computational complexity is reduced from $O(N)$ to $O(\log(N/m))$, where $m$ is the number of atoms mapped to an octree node. The number $m$ determines the number of subdivisions (or the depth of the octree), and it is empirically set to $\sim 500$. This number comes from a trade-off between the quality of frustum approximation, and the computation time. A larger depth implies smaller regions and therefore a more accurate representation of the frustum, but with the penalty of greater computation. Our tests involving million-to-billion atom datasets show no significant visual gains with a granularity that is finer than 500 atoms/region.

View frustum culling is implemented using a series of bounding volumes. These are shapes that can be used to select all octree nodes that intersect or are contained in them. Their size, position and orientation in 3D space are determined from the user's position and orientation. The first and foremost of these is a sphere, $S$, that fully encloses the frustum [7]. A coarse extraction is done by approximating all octree nodes as spheres and selecting those spheres that intersect

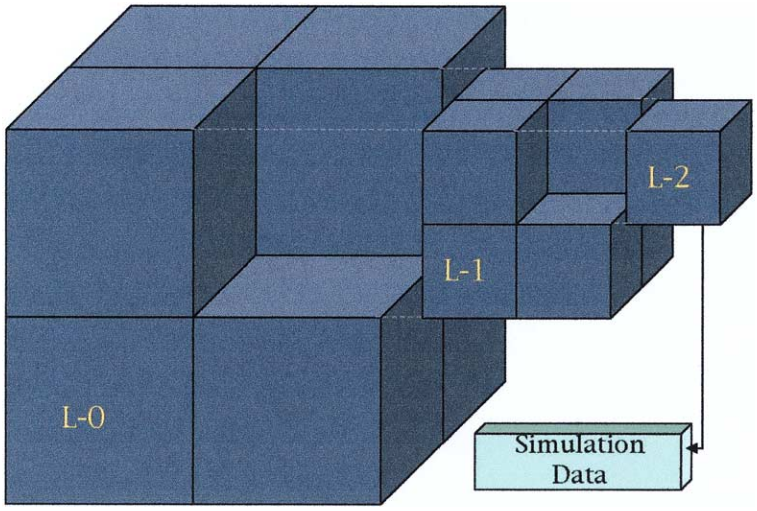

Fig. 3. A three-level octree. Each octree node contains the coordinate bounds of the corresponding subspace. A terminal node at level 2 contains a pointer to a structure that stores data associated with atoms in the region defined by the terminal node.

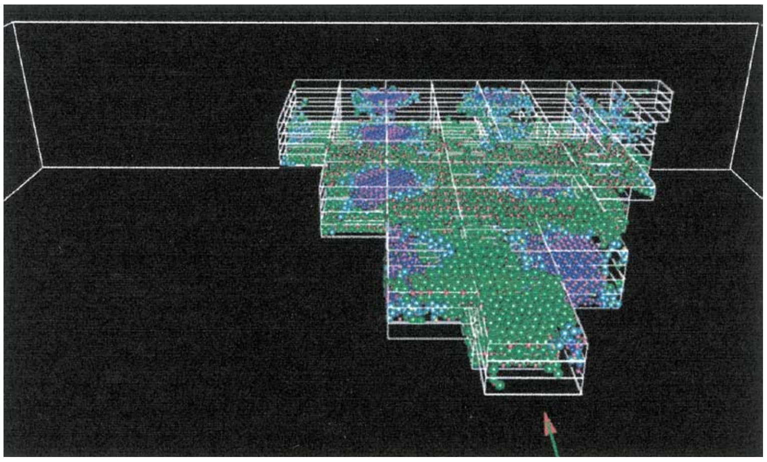

Fig. 4. The octree data structure overlain on the atomistic data. The figure shows only the atoms that are selected for subsequent rendering. The arrow denotes the position and orientation of the viewer.

with $S$. This process is implemented through a traversal of the octree, where a node is tested only if the sphere of the parent node intersects $S$. The end result of the process is a set of terminal nodes (regions), all of which intersect $S$. These regions are then pruned by testing them against a cylinder and subsequently a cone, each of which improves the approximation of the frustum. Fig. 4 shows one such extracted view frustum from a third person perspective.

### 3.2. *Probabilistic occlusion culling*

After view frustum culling, we have a list of regions that completely occupy the view frustum. We can refine this selection by repeating the frustum culling on atoms of those regions that lie on the boundary of the view frustum. However this test is redundant if a sufficient octree depth is taken. There are however many atoms that are in the view frustum that will be completely hidden from the viewer because of other atoms that are closer to the viewer. To remove these atoms we begin occlusion culling. In this process we sort all visible regions in an increasing order of distance from the viewer.[1] If atoms are uniformly distributed across regions,[2] a region that is closer to the viewer's position will contain more visible atoms than one that is further away. This is because the atoms in the nearer region will probably occlude many of the atoms in the farther regions. Thus we calculate the visibility (the fraction of atoms that are probably seen by the user) of an octree region from the recurrence relation,

$$v_c = (1 - D_c)v_{c-1}, \tag{2}$$

where $v_c$ is the visibility of the $c$th octree region and $D_c$ is the normalized density of atoms in the $c$th octree region. The 0th octree region is the one closest to the viewer. The leaf octree regions are traversed using a line-drawing algorithm, and the visibility of each cell is calculated using Eq. (2). In addition to being used in billion-atom datasets, this technique is useful when the viewer is moving. The probabilistic occlusion culling decimates atoms with probability $1 - v_c$, with typically a few percent pixel loss. This loss is acceptable since the user is navigating and not observing the scene. Computationally, this technique is inexpensive since visibility is assigned to octree regions and not the actual atoms.

## 3.3. Depth based occlusion culling

The final step in data extraction involves testing individual atoms to see if any other atoms occlude them. In this test, atoms in the nearest regions are approximated as cubes that fully enclose the spherical projection of the atom. This cube is marked on an array that mimics the viewer's screen. The array entries record the distance of a particular atom from the viewer. Before an array entry can be updated, it is tested against existing entries. For this test, a few chosen test points are compared with the pre-existing entries in the array (the distance of the currently recorded atoms from the viewer). If the distance is less, then the new atoms is closer to the viewer and it will occlude the atom that is recorded in the array. The array entry is thus updated. As the array starts to fill up, we can stop conducting this test on a per atom basis and instead conduct it on a per-region basis. If the area occupied by the $c$th region is occluded by atoms that are closer to the viewer, then we can avoid the per-atom testing for region $c$.

[1] This sorted order of visible regions is subsequently reused in the final rendering stages of the atoms when decisions pertaining to the quality/resolution of the rendering of atoms have to be made.

[2] This assumption holds for very large datasets, and thus we use the probabilistic technique in parallel implementation.

## 3.4. Multiresolution rendering

The atoms selected by the above culling algorithms are rendered as spheres at various LODs using the OpenGL API [11]. The resolution of the sphere is calculated from an exponential function of distance of the atom from the viewer. We first use the sorted list of visible regions that was created in occlusion culling. This list provides an approximate measure of the nearest and farthest atoms, and accordingly we normalize the distance in the range $[0, 1]$. The normalized resolution value $r$ is calculated as

$$r = 2 - 2^{x^2}, \tag{3}$$

where $x$ is the normalized distance of the atom from the viewer. This resolution value $r$ is defined as a fraction of the maximum resolution of the visualization. These resolution values are mappings to the maximum number of polygons that will be used in rendering a sphere. As a trivial optimization, we calculate the resolution value on a per-atom basis, only if the resolution value of the region is greater than 0.7 and the number of resolutions is greater than 10. These numbers were empirically chosen from optimal speed and quality of the rendered image. Additionally the atoms that have a resolution of 1 are usually rendered as points (which are faster to render).

The color of atoms is calculated as discrete values when visualizing atomic species and as continuous values, mapped from a color table, when visualizing atomistic attributes such as pressure. The color table uses the gradient function of the imaging program GIMP [12], and it can be manipulated at runtime to allow the user to get the best visualization effects. Different attributes are mapped to different color tables, and the program comes with several standard color tables.

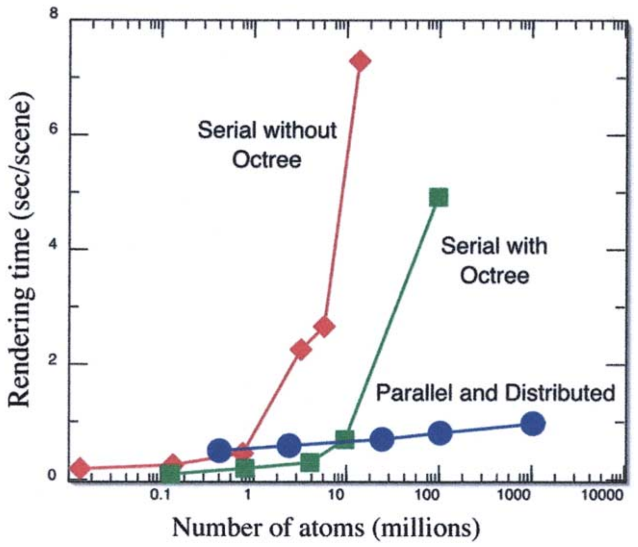

Fig. 5. Rendering time per scene as a function of the number of atoms for the parallel and distributed Atomsviewer is compared with those for the serial Atomsviewer with and without the octree enhancement.

### 3.5. *User controlled data filters*

The final data selection process is user controlled and gives the user an ability to view the insides of a dataset. A typical usage is illustrated in Fig. 1, where one quarter of the system was cut out to observe the inside and outside of the system simultaneously. To make such a cut at runtime, we use cut-planes that are aligned along the 6 surfaces. By controlling their positions, we can make different cuts or slices in a system. The quarter cut is implemented by excluding atoms whose positions lie outside a certain value. We can also exclude atoms on the basis of their attributes. Such an attribute-based plane is used, e.g., in visualizing atoms whose temperature lies within a certain range. Different attribute and position-based planes can be combined using Boolean operators, and when coupled with color tables, provide the user with highly customized visualization tools.

This paper includes documentation needed to build and use Atomsviewer. The appendices at the end of this paper detail the various stages of deployment and subsequent usage of the program. Appendix A describes the system requirements and the steps for building Atomsviewer on a Windows, Apple Macintosh and SGI computers. Appendix B outlines the various components of the program and their purpose. Appendix C describes how a user can convert their dataset to the Atomsviewer file format. Finally, Appendix D contains a complete list of commands and options of the Atomsviewer program. These commands are used to set and manipulate the color, camera and display attributes of the visualization.

## 4. Numerical results

The techniques mentioned in the previous sections have made Atomsviewer scalable, while maintaining good frame rates. However, when visualizing datasets with a billion atoms or more, we run into system limitations such as slow memory bandwidth and disk storage space. To address these issues, we have developed a parallel and distributed version of Atomsviewer [4,5].[3] We have tested the scalability of the parallel and distributed architecture and associated techniques. Fig. 5 compares the timing results of the serial Atomsviewer with and without the octree-based view frustum culling with that of the parallel and distributed Atomsviewer. We see that the time to extract and render the atoms within the field-of-view is

[3] We plan to release the parallel and distributed Atomsviewer shortly.

nearly a constant function of the number of atoms. The communication overhead is successfully overcome by overlapping communication and computation [4].

## 5. Summary

We have developed a scalable and portable code named Atomsviewer to interactively visualize large atomistic datasets, based on hierarchical view frustum-culling, probabilistic occlusion-culling, and multiresolution rendering algorithms. The Atomsviewer has been used to visualize up to tens of millions of atoms on a serial computer and, with its parallel version, up to a billion atoms.

**Acknowledgement**

This work was partially supported by ARL: CHSSI, ARO: MURI, DARPA: PROM, DOE, NSF, and AFOSR: DURINT—USC-Berkeley-Princeton. Visualization data were taken from simulations performed at Department of Defense (DoD)'s Major Shared Resource Centers under DoD CHSSI and Challenge projects. Parallel simulations were also performed on the 1896-processor HPC cluster at the Research Computing Facility and 422-processor Linux clusters at the Collaboratory for Advanced Computing and Simulations at the University of Southern California.

## Appendix A. Compiling Atomsviewer

The library program includes files to ease the building of the executable from the source code. The system requirement for compilation on various platforms is as follows.

*Windows*:

(1) Visual Studio .NET 2003.
(2) OpenGL: Microsoft ships the OpenGL API with Windows 2000/XP. For an older system or for systems without OpenGL, the API and its associated libraries can be downloaded from http://www.opengl.org.
(3) GLUT: This is the windowing toolkit that is used by Atomsviewer, and is freely available from http://www.xmission.com/~nate/glut.html.
(4) Tiff Libraries: These libraries are used by Atomsviewer to make image files from the visualization. They can be downloaded from http://www.libtiff.org/.
(5) GIMP: This is an imaging program, which makes it easy to create color gradients that are used to visualize continuous data attributes such as stress and temperature. It can be downloaded from http://www.gimp.org.

*Mac OS X*:

(1) Apple Xcode or Project Builder v2.1 (Dec 2002 Tools) or higher: These include the OpenGL and GLUT Frameworks that are used by the application.
(2) gcc compiler, v3.1 or higher: Xcode and Project builder ship with this version.
(3) Tiff libraries: These libraries are used by Atomsviewer to make image files from the visualization. They can be downloaded from http://www.libtiff.org or by using Fink (http://fink.sourceforge.net).
(4) GIMP: This program can be downloaded using Fink. Doing so will ensure that all the required dependencies are installed.

*SGI Irix*:

(1) gcc version 3.1 or higher.
(2) OpenGL & GLUT.
(3) Tiff libraries.
(4) GIMP.

Once Atomsviewer is built, a setup file and the following executables are created:

- *Atomsviewer*: This is the visualization program.
- *FileConverter*: This is the program that will convert a text file to the av file format.
- *AV_Setup.txt*: This file is used by the FileConverter and Atomsviewer to identify the directories that will be used by the program to load and save files. A user must edit this file according to their system setup.

## Appendix B. Using Atomsviewer

The typical use of Atomsviewer is a three-step process. In the first step, the data is converted from a text format to an Atomsviewer format (*.av* format) using the *FileConverter* program. This conversion is required the first time the user visualizes a dataset. For subsequent use the user will use the file in the *.av* format. Once the data has been converted, a configuration file needs to be created, and color tables that will be used to color the various data attributes, need to be selected/created. Finally the *Atomsviewer* program can be used to visualize the data file. These three steps and the programs associated with them are described in detail in the following appendices.

## Appendix C. File conversion using FileConverter

*Usage*
FileConverter [**-L** | **-E** ] [ **-C** ] <u>*text file*</u>

*Description*
The *FileConverter* program will read a text file and convert it to an *.av* format. This text file will contain all the data that a user wants to visualize. The output file will have the same name as the input file and a *.av* extension. It will be created in the same directory as the input file. The input text file should be of the following layout:

(1) The first line of the file indicates the number of frames (or time steps) in the file. *This is required even if there is only one frame in the system*.
(2) The second line indicate the number of atoms, $N$, in the first frame.
(3) The third line contains the data for an atom from the first frame. This line is structured as follows:
   - $x, y, z$ coordinates delimited by a white-space or a tab.
   - White-space/tab.
   - The species of the atom as a positive integer. If all the atoms are identical or this field is not recorded, then set this to 1.
   - White-space/tab.
   - Multiple data attributes associated with this atom as integers/floating-point numbers. Each data attribute will be delimited by a white-space or a tab. Data attributes are optional.

Once all the atoms of frame 1 have been listed ($N$ atoms from line 2), list the number of atoms in frame 2 and so on.

*Options*

**-L** Specify the octree level that will be used in the creation of the output file. This option is ignored if the **-E** option is specified.
**-E** Causes the program to make a decision about the octree level that will be used in the creation of the output file.
**-C** Causes a configuration file with typical settings to be created. This file has the same name as the input file but with a *.cfg* extension. The input file that needs to be converted must be specified with its full path.

*Output*

*The following files are created.*

(1) Atomsviewer file: An Atomsviewer file is a binary file with an .av extension that contains the positions and data of all the atoms that will be visualized. The atoms are spatially organized in the file with file headers at frequent intervals to allow the program to skip over portions of the file that are not being rendered. Since atomic data might contain data attributes such as temperature, energy, etc., the file is actually composed of two parts. A main file with only the positions and the species of all the atoms and a set of data files (with extensions .av.1, .av.2, etc.) that contain at most 3 data attributes for every atom in the system. The FileConverter program automatically creates these data files, when the Atomsviewer file is being created.
(2) Configuration file: A configuration file is a text file, with a .cfg extension, that is used to define various graphic properties such as the size of the atomic representations and their colors, the camera viewpoints etc. A basic configuration file is created by the FileConverter program with the -C option. The file consists of a set of commands that can describe any one of the following properties:
   - Color and radius of the atomic representations.
   - Color gradient files that specify the colors, used to visualize various data attributes.

- Color and format specifier for the main display label.
- Camera position and direction.

The various commands that constitute a configuration file are listed in Appendix D. These commands are the same as the commands that can be executed at runtime. It is important that the name of a configuration file must be identical to that of the Atomsviewer file with only a different extension. For example, an Atomsviewer file with the name of foo.run1.txt.**av** would have a configuration file called foo.run1.txt.**cfg**.

(3) Color gradient file: This file describes the colors that are used to visualize various data attributes. The color gradient files have the same name as the Atomsviewer file but with an extension of **.col.1** where the number 1 at the end associates a gradient file with the data attribute in the data file. Thus an extension of **.3** would stand for the third data attribute in the system excluding the position and species of the atoms.

To create a color gradient file, first scale the data range to $[0, 1]$ and then use the gradient facility of the program GIMP (GNU Image Manipulation Program). GIMP is an open source program that is freely available and runs on Windows, Linux and Mac platforms. The color gradient file is a plain text file that gives the color at various cutoff points between 0 and 1. Atomsviewer then uses this information to interpolate the color for a given value. The color values are in the RGBA format, and the interpolation is linear.

## Appendix D. Visualizing

*Usage*

Atomsviewer *av file*

*Description*

The *Atomsviewer* program will read an Atomsviewer file and visualize it. When the program is first launched, it will create a blank window, which will display the visualization, and a console window, that will display various messages associated with the graphical rendering. To start the program, go to the rendering window and press the letter ‘o’. This will allow the user to enter the name of the Atomsviewer file that they wish to visualize. Enter the full path of the file and press enter. If the file is successfully read by the program, the atomic data will be drawn. Different keys on the keyboard are used to control different functions of the program such as camera movement, animating the entire dataset, taking snapshots etc. These commands are shown in Fig. 6.

The following set of commands can be used to specify various parameters of a given visualization. These include the size and color of the atomic representations, label specifiers, etc. These commands can be grouped in a configuration file or are entered at runtime.

BACKGROUND_COLOR $< r, g, b, a >$ Specify the background color of the window in the RGBA format. The values of $r$, $g$, $b$ and $a$, specify the red, green, blue and alpha components of the color, respectively, and lie between 0.0 and 1.0.

BOX Toggle the display of the two bounding boxes.

CAM_DIR $< w', x', y', z' >$ This command allows the user to specify the orientation of the camera. This is useful to recreate a particular orientation when viewing multiple files or record a series of camera orientations and create an animation of an exploration of a visualized system. $w', x', y', z'$ specify the orientation as a quaternion [13].

CAM_POS $< x, y, z >$ This command allows the user to specify the position of the camera. This is useful to recreate a particular position when viewing multiple files or record a series of camera positions and create an animation of a walkthrough in a visualized system.

CAM_P_D $< x, y, z, w', x', y', z' >$ This command allows the user to simultaneously specify the camera position and orientation. A series of these commands are used to record and recreate a walkthrough of a visualized system.

CLIPPING_PLANE $< f >$ Set the clipping plane to the value specified by $f$.

COMMANDS Display the list of possible commands on the console with a brief description.

EXIT Quit the program.

FONT_COLOR $< r, g, b, a >$ Specify the color of the font, which will be used in the display window, in an RGBA format. The values of

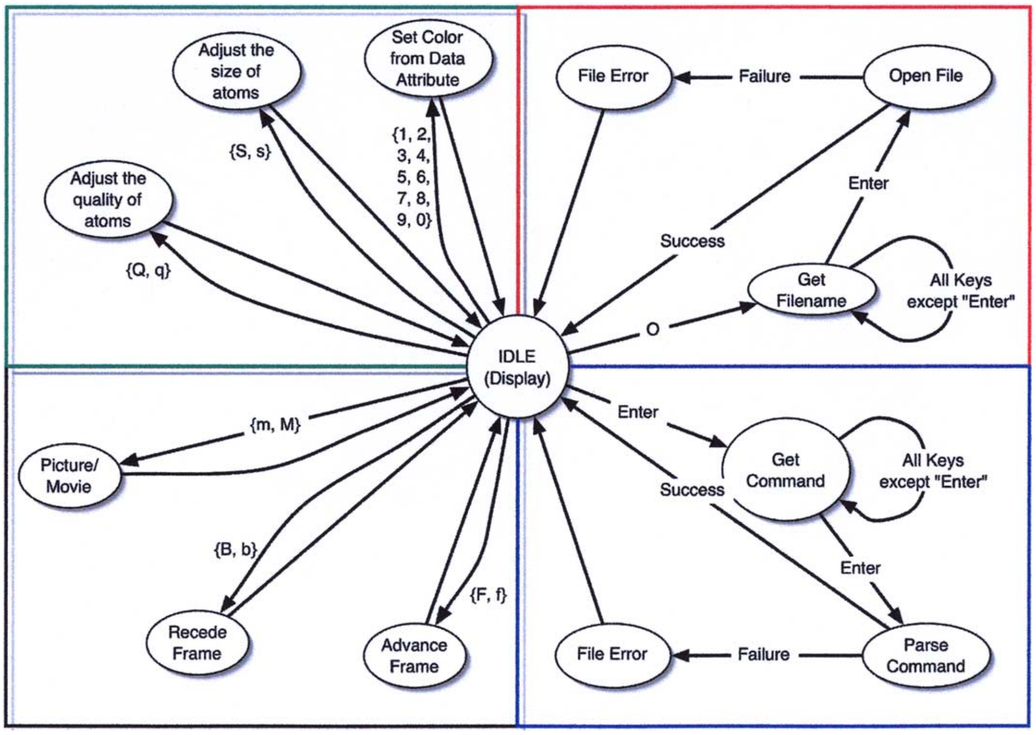

Fig. 6. Frequently used commands of Atomsviewer.

$r$, $g$, $b$ and $a$, specify the red, green, blue and alpha component of the color, respectively, and lie between 0.0 and 1.0.

FRAME $< f >$ Jump to frame $f$.

FRAME_LABEL $< f, s >$ The frame label can be used to specify a tag that can be used to describe the image being visualized. In a multi-frame visualization, this label could for example describe the time associated with each frame.

FRAME_LABEL_START $< f >$ This is useful only in a multi-frame visualization, where it will indicate the value that will be displayed when visualizing the first frame.

FRAME_LABEL_STEP $< f >$ This is useful only in a multi-frame visualization. This value indicates the increment that will be used in the label display of subsequent frames.

HIDE_SPECIES $< i, j >$ Hide species from $i$ to $j$, where $i \leqslant j$.

LOAD_AV_FILE $< s >$ Load the Atomsviewer file (with the extension .av) specified by the string $s$.

LOAD_CFG_FILE $< s >$ Load the configuration file (with the extension .cfg) specified by the string $s$.

LOAD_GRADIENT $< s >$ Load the color gradient file (with the extension .col) specified by the string $s$.

ORIGIN Return the camera to its default position and orientation.

SPECIES_COLOR $< i, r, g, b, a >$ This lets a user set the color of a specie $i$ (an integer) as an $rgba$ quadruple, where each color component is a float value between 0 and 1. This is a command that should be included in all configuration files. The configuration file created by the file converter program will have this command in it, but with certain default color values.

SPECIES_RADIUS $< i, r >$ This sets the radius of a specie $i$ (an integer) to a float value $r$. All atoms default to a radius of 1, and a user can use this command to draw the atoms at proportional radii.

SHOW_SPECIES $< i, j >$ Show species from $i$ to $j$, where $i \leqslant j$.

## References

[1] C.L. Rountree, R.K. Kalia, E. Lidorikis, A. Nakano, L. Van Brutzel, P. Vashishta, Atomistic aspects of crack propagation in brittle materials: multimillion atom molecular dynamics simulations, Annual Review of Materials Research (2002).

[2] F.F. Abraham, R. Walkup, H.J. Gao, M. Duchaineau, T.D. De la Rubia, M. Seager, Simulating materials failure by using up to one billion atoms and the world's fastest computer: Brittle fracture, in: Proceedings of the National Academy of Sciences of the United States of America, 2002.

[3] A. Nakano, R.K. Kalia, P. Vashishta, T.J. Campbell, S. Ogata, F. Shimojo, S. Saini, Scalable atomistic simulation algorithms for materials research, Scientific Programming 10 (2002).

[4] A. Nakano, J.X. Chen, High-dimensional data acquisition, computing, and visualization, Comput. Sci. Engrg. 5 (5) (2003).

[5] A. Sharma, R.K. Kalia, A. Nakano, P. Vashishta, Large multidimensional data visualization for materials science, Comput. Sci. Engrg. 5 (2) (2003).

[6] W. Humphrey, A. Dalke, K. Schulten, VMD: visual molecular dynamics, J. Molecular Graphics 14 (1) (1996).

[7] A. Sharma, A. Nakano, R.K. Kalia, P. Vashishta, S. Kodiyalam, P. Miller, W. Zhao, X.L. Liu, T.J. Campbell, A. Haas, Immersive and interactive exploration of billion-atom systems, Presence—Teleoperators and Virtual Environments 12 (1) (2003).

[8] I. Szlufarska, R.K. Kalia, A. Nakano, P. Vashishta, Nanoindentation-induced amorphization in silicon carbide, Appl. Phys. Lett., submitted for publication.

[9] J.H. Clark, Hierarchical geometric models for visible surface algorithms, Commun. ACM 19 (10) (1976).

[10] D.V. Pinsky, J. Meyer, B. Hamann, K.I. Joy, E.S. Brugger, M.A. Duchaineau, An error-controlled octree data structure for large-scale visualization, Crossroads—The ACM Student Magazine (spring 2000).

[11] M. Woo, J. Neider, T. Davis, D. Shreiner, The OpenGL Programming Guide, third ed., Addison–Wesley, Reading, MA, 1999.

[12] GNU Image Manipulation Program (GIMP), available under GPL from http://www.gimp.org.

[13] K. Shoemake, Animating rotation with quaternion curves, ACM SIGGRAPH Computer Graphics 19 (3) (1985).