Structure, Vol. 13, 447–462, March, 2005, ©2005 Elsevier Ltd All rights reserved. DOI 10.1016/j.str.2005.01.010

A Component-Based Software Environment for Visualizing Large Macromolecular Assemblies

Michel F. Sanner[1,*]
Department of Molecular Biology, MB5
The Scripps Research Institute
La Jolla, California 92037

Summary

The interactive visualization of large biological assemblies poses a number of challenging problems, including the development of multiresolution representations and new interaction methods for navigating and analyzing these complex systems. An additional challenge is the development of flexible software environments that will facilitate the integration and interoperation of computational models and techniques from a wide variety of scientific disciplines. In this paper, we present a component-based software development strategy centered on the high-level, object-oriented, interpretive programming language: Python. We present several software components, discuss their integration, and describe some of their features that are relevant to the visualization of large molecular assemblies. Several examples are given to illustrate the interoperation of these software components and the integration of structural data from a variety of experimental sources. These examples illustrate how combining visual programming with component-based software development facilitates the rapid prototyping of novel visualization tools.

Introduction

Recent advances in cryoelectron microscopes (cryoEM), automated image acquisition, and image processing offer the potential of near-atomic resolution (3–5 angstroms) of large molecular assemblies by cryoEM methods (Subramaniam and Milne, 2004). As a consequence of these advances in experimental techniques, we have recently witnessed a rapid increase of the number of large molecular complexes that have become available in the Protein Data Bank (Berman et al., 2000). The transition from studying small molecular systems in isolation toward the study of molecular systems in complex environments, potentially as large as a complete cell, creates numerous new computational challenges. For molecular systems comprising millions of atoms (viral capsids, molecular motors, ribosomes, etc.) it is not practical or even possible to store each atom in memory explicitly. However, the study of local interactions requires this level of detail. Hence, novel computational and visual representations of molecular systems are needed.

These problems can be addressed using hierarchical levels of abstractions and navigation tools which help in interactively exploring such assemblies and their overall architecture. The investigation of specific interactions requires the ability to increase the level of detail to the atomic level as we zoom in on a particular region. Similar challenges are found in disciplines such as database visualization, navigation in virtual cities, or the visualization of large CAD/CAM models, and the solutions are likely to rely on computational techniques developed in a variety of fields. Designing computational environments capable of integrating heterogeneous models spanning a wide range of biological scales and allowing the interoperation of a variety of multidisciplinary computational techniques further accentuates the software and data integration problems that we are already facing today.

Fortunately, progress has been made to address these challenges. Our laboratory has explored the use of symmetry operators for building molecular assemblies (Macke et al., 1998) and multiresolution representations of protein shapes and properties (Duncan and Olson, 1995a, 1995c). More recently, techniques taking advantage of the new programmable graphics cards for displaying multiresolution models of large assemblies have been developed (Bajaj et al., 2004). However, these exploratory techniques are typically not available in widely used molecular visualization software.

To further our understanding of large and complex macromolecular assemblies, we need a flexible software infrastructure that will facilitate the exploration of a variety of representations and interaction methods in order to identify the most effective ones. In addition, this infrastructure should facilitate the incorporation of new computational methods as they become available and support their interoperation. This means that computational methods should be developed as software components, rather than stand-alone programs, so that they can ultimately be used as part of a larger workflow. Unfortunately, most of the currently available biological simulation and analysis software has been developed with the goal of answering specific biological questions (e.g., how does this molecule's conformation change over time in a given force field? How do these two molecules interact?). This has lead to "monolithic" programs in which the parts of the code dealing with each computational subtask are not independent of one another. Even programs implemented in object-oriented languages such as C++ or Java can suffer from this problem, because compiled languages do not promote compartmentalization. Software components written in compiled languages require the writing of a "main" function whose sole purpose is to ascertain if this component compiles independently. This is rarely done, and the potential dependencies between components in a compiled program only appear when components need to be reused or replaced. This difficulty in extracting and reusing components leads to the frequent reimplementation of software that performs the same tasks, and to programs which are difficult to maintain and to extend.

A growing number of molecular visualization software packages provide an application program inter-

*Correspondence: sanner@scripps.edu
[1]Lab address: http://www.scripps.edu/~sanner/software

face (API), allowing the embedding of third party computational methods and are therefore extendable. However, this approach imposes one particular software framework upon users, and often computational methods added to such frameworks are not easily reused in other software environments. We refer to these programs as "application centric," as the framework is an application.

In addition to extendability, the software infrastructure should allow users to combine available computational methods in new and unanticipated ways. Interrogative visualization, in which the data are viewed to gain insight into a problem, is an inherently iterative process. A particular view might raise new questions that prompt different visual representations or simulations to be performed. It is impossible for a programmer to foresee all the possible ways in which a user might want to combine computational methods. Hence, it is desirable that the software infrastructure empowers the domain scientists with the ability to combine software components and interactively reconfigure the computational workflow, without having to learn a programming language.

Such a goal can be achieved using visual programming environments. Here, users build computational networks by graphically connecting the input and output ports of computational nodes. Visual programming environments such as AVS from AVS Inc. (Upson et al., 1989), Iris Explorer (2003), SCIRun (Parker, 1999), and OpenDX (IBM, 2002) have been used for their ability to encapsulate computational methods into nodes that can be shared with other users, and because they allow nonprogrammers to create new computational networks. However, these environments also fall into the application centric category and often place serious restrictions on the user. In particular, nodes developed for one of these environments can only be used within that particular program.

Over the past five years, we have developed a "language-centric" approach to software development (Sanner, 1999, 2005) in which the framework is the high-level, interpretive language Python (Lutz and Asher, 1999). The software we developed is written as extensions to the Python language. This approach promotes compartmentalization and enables the reuse of these components (or extensions) in any program written in Python, or that has an embedded Python interpreter. We have found that Python provides a powerful glue for assembling computational components and, at the same time, a flexible language for interactive scripting of applications. While Python has excellent support for seamlessly integrating code written in compiled languages such as C, C++, or FORTRAN (SWIG), we found that many components are not time critical and therefore can be written in Python with no noticeable impact on execution performance. The advantage of these components is that they are platform independent.

So far, we have implemented more than 20 software components, half of which are implemented in the Python programming language, while the others wrap C or C++ libraries. The 10 software components written in Python amount to over 220,000 lines of code and define close to 1400 classes. We have created several stand-alone applications from these software components, including PMV (Sanner, 1999), a fully-fledged molecular visualization and analysis environment; ADT, a front end to the automated docking code AutoDock (Morris et al., 1998); Vision, (Sanner et al., 2002), a visual programming environment; and PyARTK, an augmented reality application for structural biology (Gillet et al., 2004a, 2004b). These applications have been distributed to over 9000 users so far and the underlying software components have been reused beyond the boundary of our laboratory and outside the field of biology.

In this paper, we describe a component-based software environment that supports the interactive and visual combination of computational nodes into networks that correspond to algorithms coded at a high level. We describe several software components developed for the visualization of biological molecules and discuss particular features that are relevant to the visualization of very large complexes. We also present several applications of this software that demonstrate the flexibility of the proposed approach and its ability to support the rapid prototyping and implementation of novel representations and interaction techniques for studying macromolecular assemblies.

Results and Discussion

In this section, we present the visual programming environment Vision and demonstrate, in the context of the visualization of large molecular assemblies, how it enables the rapid and intuitive integration of functionality implemented in several independent software components. We also describe several other software components that are relevant to the visualization of large molecular assemblies.

Vision: A Software Component for Visual Programming

The AVS data-flow visualization environment has been used for scientific visualization at TSRI for over 15 years (Sheehan et al., 1996; Duncan and Olson, 1995b). While AVS is extendable and allows a nonprogrammer to create simple programs in the form of computational networks, it is application centric. The lack of a scripting language, the lack of control over the execution flow, and the constraining data types proved to be serious handicaps for advanced use. When we shifted our software development from AVS to the Python-centered approach, we gained a tremendous amount of programmability, flexibility, and reusability, but lost the visual programming capability. This prompted the development of Vision, a Python software component supporting the visual-programming paradigm.

Vision was designed to provide nonprogrammers with an intuitive interface for building networks describing new computational pipelines and novel visualizations of data (Figure 1). Its graphical user interface was intentionally designed to look and feel like AVS to facilitate its adoption by TSRI scientists. However, the underlying architecture has a number of fundamental differences, including no restrictive data types or data models, interactive programmability at runtime, and a general purpose, high-level, interpreted language at the

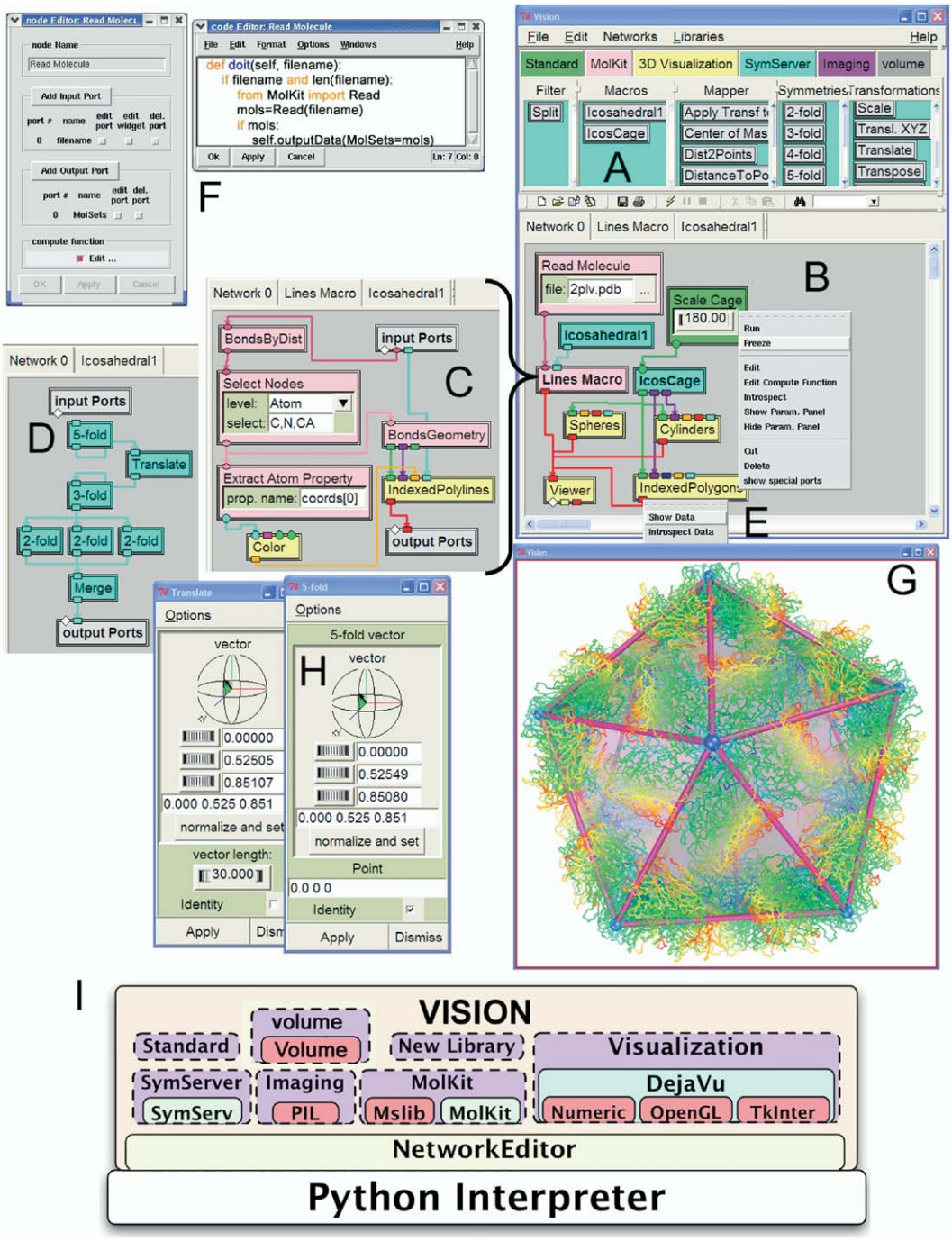

Figure 1. Vision: A Visual Programming Environment

(A) Vision nodes are organized in libraries which can be loaded at run time. (B) Nodes can be placed onto a canvas and connected to create computational networks. This particular network reads the polio virus coat protein (2plv.pdb), and passes it into a macro node (C) that generates a line representation for covalent bonds between C, N, and C-α atoms. The vertices of the line representation are colored using a blue to red ramp according to the X coordinates of the atom centers ("Extract Atom Properties" nodes extracting coords [0] for each atom). The line geometry is output by the macro and displayed using a "Viewer" node. The "Icosahedral1" node is a macro node (D), which combines point symmetry operators to create 60 matrices describing the icosahedral arrangement of the protomer in a vial capsid. These matrices are used to duplicate the line geometry 60 times to form a complete capsid. A "Translate" node inserted between the "5-fold" and the "3-fold" nodes allows translating each pentamer along its 5-fold axis. The "IcosCage" macro node provides vertices, edges, and faces describing an icosahedron. These are used to display spheres, cylinders, and triangles outlining the icosahedral symmetry. (E) Clicking with the right mouse button on items in the network displays object-sensitive menus. Here, the menus are displayed for an output port and for a node. (F) The node editor allows the inspection and modification of the function executed by a node. Here, the node editor was started on the "Read Molecule" node. (G) The interactive 3D camera corresponding to the "Viewer" node. (I) Architectural view of Vision. Nested boxes denote dependencies between packages. Note that most libraries of nodes depend on other Python packages which are independent of Vision. Boxes with red backgrounds are packages containing platform-dependent code written in C or C++.

heart of the system providing powerful scripting capabilities. The key difference with other visual programming environments is architectural. While Vision can be used as a stand-alone application as shown in Figure 1, it was designed as a software component. Hence, it is reusable in any application running a Python interpreter.

We have demonstrated this ability by integrating Vision with our molecular visualization program PMV (Stoffler et al., 2003). In Vision, computational methods are exposed as "nodes" which are grouped into libraries (Figure 1A) such as "Standard," "MolKit," and "3D Visualization." Nodes can be placed onto a canvas and their input and output ports can be connected (Figure 1B). This can be done using in a drag-and-drop fashion using the mouse, or programming. Nodes in Vision networks can be colored according to the library they originated from. The connections between the nodes define a directed graph that is used to propagate data between nodes. Nodes typically execute when they receive new data. Subnetworks can be encapsulated into macro nodes (Figures 1C and 1D), allowing the recursive nesting of networks. Tool tips and balloon help provide runtime information about the function, input, and output ports of nodes. Data flowing through nodes can be monitored interactively (Figure 1E). Input and output ports can optionally specify a data type which is reflected in the port's graphical representation, providing helpful visual hints for connecting the proper outputs with the proper inputs. Data types are also used to validate data flowing through the network. If no type is provided, the port accepts any Python object. Networks and their configuration parameters can be saved in a file as Python code that can be executed later to restore the networks from the file. This approach provides a flexible and general-purpose programming language for describing networks, while avoiding the choice or definition of a network file format. The network restoration mechanism is fault tolerant, allowing loading of networks in which some parts are broken. This is an important feature given that software inevitably evolves, leading to the potential obsolescence of parts of networks in older projects.

Another unique feature of Vision is that new nodes are easy to create. The simplicity of the Python language makes writing a new node for Vision much easier for casual programmers than it is in C or C++ based systems. New nodes can be created interactively during a session using the node editor (Figure 1F). In this editor, new ports are added with the click of a mouse button and the only Python code that needs to be written is the node's computational function. This editor also enables users to inspect the implementation of a node and modify it interactively without having to recompile or quit the application. The node editor shown in Figure 1F was started for the "Read Molecule" node. The code of this node demonstrates how Vision promotes the separation between the visual programming environment and the computational methods exposed as nodes in Vision. The Python function "doit" associated with the Read Molecule node receives a filename as a string from the input port of the node (in this node the value comes from a widget bound to the port) and uses the Read function from the MolKit Python package to read the file and build an object representing the molecules read from the file. This object is then output on the node's output port. The node merely serves as an adaptor to expose a functionality provided by MolKit, thus allowing MolKit to remain completely independent of the Vision environment. In fact, most libraries of Vision nodes expose functionality from packages that are independent of Vision, as shown in the architectural diagram of Vision (Figure 1I). The only library of nodes which does not expose functionality from another Python software component is the Standard library. This library contains nodes exposing Python syntactical elements such as "range" which creates a list of values, or the "iterate" node which allows looping over a list of objects in a network, etc. This library also provides nodes for the input of simple data types such as floats, integers, strings and filenames, and simple widgets (dials, entries, file browsers, etc.).

The combination of the visual programming paradigm and the ability to interactively inspect and edit nodes written in a high-level language creates an unprecedented number of levels at which users can interact with the program. It also creates an easy and incremental way for nonexpert users to increase their computational skills. Scientists can begin by using networks created by others. As they grow familiar with Vision, they can start modifying existing networks and later, create their own. If they learn the basics of Python scripting they will be able to customize existing nodes to their needs and ultimately learn to write nodes from scratch.

To date, several libraries of nodes exposing functionality of several Python packages are available including "3D Visualization," "volume," and "SymServer" which we will describe in more detail below, and Molkit, exposing functionality from the MolKit Python package for reading, writing, and manipulating molecules. In the section called "Applications," we provide examples in which we use Vision to rapidly design, prototype, and evaluate novel representations of complex molecular assemblies.

SymServ: The Symmetry Server

We have implemented a set of Python objects for generating point symmetry transformations. These objects are available as a library in the vision environment (Figure 1A) and allow the creation of streams of transformation matrices known as instance matrices, which can be assigned to any geometry in the 3D scene to duplicate this geometry in various locations of the scene (see the section on DejaVu below). This library provides a macro node defining the 60 matrices needed to create an icosahedral viral capsid in a standard orientation (Figure 1D). The matrices are obtained by nesting a 5-fold, a 3-fold, and three 2-fold symmetry operators. In the network in Figure 1B, these matrices are output by the macro node "Icosahedral1" and passed into the "Lines Macro" node (Figures 1B and 1C) node where they are set as instance matrices for the lines geometry created by the "IndexedPolylines" node, leading to the display of 60 copies of the line representation of the polio virus coat protein (2plv.pdb) (Figure 1G). Building the 60 matrices using a network of symmetry operators

instead of using a hard-coded list of transformations enables inserting additional transformations in the pipeline. Figure 1D shows a "Translate" node inserted between the "5-fold" and the "3-fold" nodes. After setting the translation vector in the parameter panel of this node (Figure 1H) to match the 5-fold axis, it is possible to translate each pentamer along its 5-fold axis using the thumbwheel labeled "vector length." A value of 30 was used to create the exploded view of the capsid (Figure 1G). If the amplitude of the translation is resulting from a calculation rather than specified by a user, one can simply unbind the thumbwheel widget currently bound to this parameter which then becomes visible as an input port on the node. The "IcosCage" macro node (Figure 1B) provides the coordinates of the vertices and the indices describing edges and faces of an icosahedron. This description is converted into geometric objects (a sphere for each vertex, a cylinder for each edge, and triangles for the faces) which are added to the viewer to visualize the icosahedron. The "Scale Cage" node's thumbwheel allows scaling of the icosahedron. The viewer's graphical user interface (GUI) (Figure 6B) can then be used to define the color and transparency of these geometric primitives.

Using this software component, we are able to display complete viral capsids while only storing in memory the molecules comprised in one asymmetric unit, thus reducing the memory requirements by a factor of 60. This enables the computational representation on commodity hardware of the largest viral capsids for which atomic structures are available today, such as the bluetongue virus. This particular virus contains a total of 960 proteins comprising almost 3 millions atoms in the entire capsid, while a single asymmetric unit only contains 15 proteins comprising 49063 atoms.

Volume: A Software Component for 3D Volumetric Data

The Volume software component implements low-level support for handling 3D volumetric data such as electrostatic fields or density maps obtained from experiments such as X-ray crystallography or cryoEM. This package defines the Grid3D object for representing regular three-dimensional grids of scalar values. This object stores information about the grid such as its origin, the extent of its dimensions, and the number of grid points in each dimension, along with the actual 3D array of values. The Grid3D object is implemented in Python and uses Python's numeric extension for efficiently storing and manipulating the large 3D arrays of values. Subclasses are available for the most often encountered data types (e.g., unsigned characters, short integers, integers, floating point values, double precision, etc.). Parsers are provided for a variety of commonly used 3D volumetric data file formats (see Table 1). All currently available parsers are implemented in Python. For file formats supporting multiple data organizations, such as the CCP4 and the MRC file formats, the parsers support all possible data organizations, although the C-style (i.e., the index of the third dimension moves fastest and the index of the first dimension moves slowest) and FORTRAN-style (the opposite of C-style) data organizations are read much more efficiently by these parsers, as no explicit loop over the data has to be implemented in the Python language.

In addition to parsers, this software component provides objects to perform basic manipulations on 3D arrays of values stored in the Grid3D objects, such as mapping the data to a new range of values, performing trilinear interpolations, subsampling, clipping and selecting data values, etc. This functionality is exposed to users in the visual programming environment Vision through a library of computational nodes (Figure 2A). Nodes in a Vision library are organized in categories. The "Input" category of the volume library provides nodes for parsing the volumetric data formats mentioned earlier. The nodes in the "Operators" category perform basic operations on Grid3D objects. The nodes in the "Mapper" category either receive a Grid3D object as an input and transform it into another data type, or receive some other data type and output a Grid3D object. Finally, the "Filter" category provides nodes for selecting data in specific regions of a grid. The region used to select data can be specified using a "Region Box" node (Figures 2B–2D) allowing choosing an octant or one half of the volume, an axis aligned box (Figure 2E), one or more spheres (Figure 2F), or an arbitrary mask (Figures 2G and 2H). A mask is a Grid3D object containing 0 and 1 values. When selecting data in a grid using a mask, only data in the regions where the mask is not null will be kept. Mask arrays can currently be created by using the "Threshold Mask" node on a grid object to create a mask where the data is greater, less, equal, or different from a user-specified value, or by setting the mask to 1 inside an arbitrary set of spheres (Figures 2F and 2H). The "logicOP" node performs logical operations on mask grids, enabling the creation of complex masks such as the spherical slab shown in Figure 2F.

For volumes describing crystallographic cells, the Grid3D object also contains an instance of a Crystal object. This object provides the edge lengths and angles of the crystallographic cell, along with methods for converting between cartesian and fractional coordinates. These methods are used for the proper rendition of iso-surfaces and grid bounding boxes for nonorthogonal volumes (Figure 2I). The "Orthogonalize" node allows sampling a nonorthogonal grid onto an orthogonal one.

This Vision Volume library provides a node for the rapid calculation of iso-surfaces. All iso-surfaces shown in this paper were computed using this node. Direct volume rendering (Gelder and Kim, 1996) enables the visualization of volumetric data without the need for creating a polygonal geometry. A transfer function associates an opacity and color to each value found in the 3D dataset and the complete volume is rendered by blending these opacity and colors along rays going from the viewpoint through the volume. This technique provides an interesting alternative to the iso-surfaces, which can easily contain millions of polygons. On computers supporting either paletted 3D textures or equipped with a programmable GPU, the Vision's volume library provides nodes for performing direct volume rendering and for editing the transfer function (Figures 3F and 3D). The iso-contouring and volume rendering

Table 1. Volumetric Data File Formats for which Parsers Are Currently Available

Binary	C-style binary 3D array of data. Dimensions, data type, and byte order can be specified.
Delphi	Electrostatic field file format produced by the Delphi program (Honig and Nicholls, 1995).
UHBD	Electrostatic field file format produced by the UHBD program (Ilin et al., 1995).
AutoGrid	AutoDock affinity grids (Morris et al., 1998).
BinaryRawiv	Raw binary format with header (http://ccvweb.csres.utexas.edu/docs/data-formats/rawiv.html).
CCP4	CCP4 density map file format. All data organizations are supported.
MRC	MRC density map file format. All data organizations are supported.
CNS/Xplor	Xplor density map file format.
OpenDX	Electrostatic potential produced by APBS (Holst et al., 2000).
SPIDER	Electron microscopy 3D image stack (http://www.wadsworth.org/spider_doc/spider/docs/index.html).
BRIX	Volumetric file format used by the macromolecular crystallographic modeling program O.

nodes are based on libraries written in the C++ programming language and developed in the Bajaj laboratory at The University of Texas at Austin (Bajaj et al., 1996, 2002).

The functionality currently available in the volume software package provides the basic elements for building visualization tools useful for the presentation and analysis of experimental data such as density maps resulting from cryoEM experiments. The availability of this functionality in the form of computational nodes in the Vision environment facilitates their interactive combination and their integration with other computational methods. Additional features are planed, including support for clipping data using boxes with an arbitrary orientation relative to the data grid, and arbitrary slicing planes. Python packages such as Numarray (Greenfield, 2003) and SciPy (Oliphant, 2003) provide advanced numerical methods similar to those found in IDL, Matlab, or Octave. These software packages are able to operate directly on the numeric arrays we use to store the volumetric data, thus making it trivial to extend the library with new nodes exposing filtering operators, etc. This example illustrates how our approach can leverage other software components developed by the Python community. We are also working on creating higher-level user interfaces to networks performing common tasks. In these user interfaces, the underlying network will be hidden from the user, and only a subset of controls will be provided for specifying the input data and some of the parameters influencing the calculation or visualization. However, advanced users will still be able to display, inspect, and customize the network.

DejaVu: An OpenGL-Based 3D Graphics Visualization Component

DejaVu is our OpenGL-based, platform-independent, general-purpose 3D geometry visualization component. It is written entirely in Python and defines classes describing objects such as Viewer, Camera, Light, ClippingPlane, ColorEditor, Geometry, etc. The Viewer class implements a fully fledged visualization application. It provides control over a large number of rendering parameters including user controllable depth-cueing; global anti-aliasing; perspective and orthographic projection modes; multiple light sources; as well as per geometry: rendering modes (points, lines, polygons, and outlined), shading modes (flat and Gouraud), culling modes (back, front, and none), arbitrary clipping planes, magic lenses that reveal the geometry only inside the lens, blending functions for transparency, etc. Each DejaVu Viewer object maintains a hierarchy of geometric objects displayed in the Viewer's GUI (Figure 6B). Rendering attributes and 3D transformations can be defined for any particular geometry in this hierarchy, or can be inherited from a parent. A Viewer object contains a virtual trackball object for rotating, translating, and scaling. This trackball can be bound to any geometry, camera, light source, clipping plane, or texture.

DejaVu's set of geometry objects is extendable and currently includes: Polylines, IndexedPolylines, IndexedPolygons, QuadStrips, TriangleStrips, Spheres, Cylinders, Ellipsoids, Arcs3D, Arrows, Box, Points, CrossSets, and TextLabels. Such objects can be instantiated and added dynamically to a viewer. A list of 4 × 4 transformation matrices called "instance matrices" can be assigned to any objects in a DejaVu hierarchy causing it to be drawn in multiple locations and orientations.

DejaVu's functionality is exposed in the Vision environment by the 3D Visualization library (Figure 3A). When a Viewer node is added to a network, it creates a DejaVu Viewer object to which geometry objects can be added.

Upon a redraw event, the hierarchy of objects is traversed and every object is asked to draw itself. Most objects compile OpenGL display lists to accelerate the drawing process. However, it is possible for any geometry to be rendered in immediate mode (i.e., without the display list). We have used this mechanism to create special geometry objects for direct volume rendering. For these objects, each redraw calls the object's display function that either talks to a hardware-accelerated volume rendering board, or calls a C++ library for 3D texture-based volume rendering. The visualization of the electrostatic potential of superoxide dismutase (Figure 3) illustrates this rendering technique.

DejaVu greatly facilitates the addition of advanced 3D visualization capabilities to any application running a Python interpreter. It is used as the 3D graphics engine in all of our visualization applications and is under active development. The latest additions include tiled rendering for rendering images at resolution larger than the display's resolution while using the graphics hardware acceleration and nonphotorealistic rendering (i.e., nonshaded rendering augmented with silhouette outlines).

Other Software Components and Applications

We have developed several additional software components as Python packages (see Table 2). Some of these

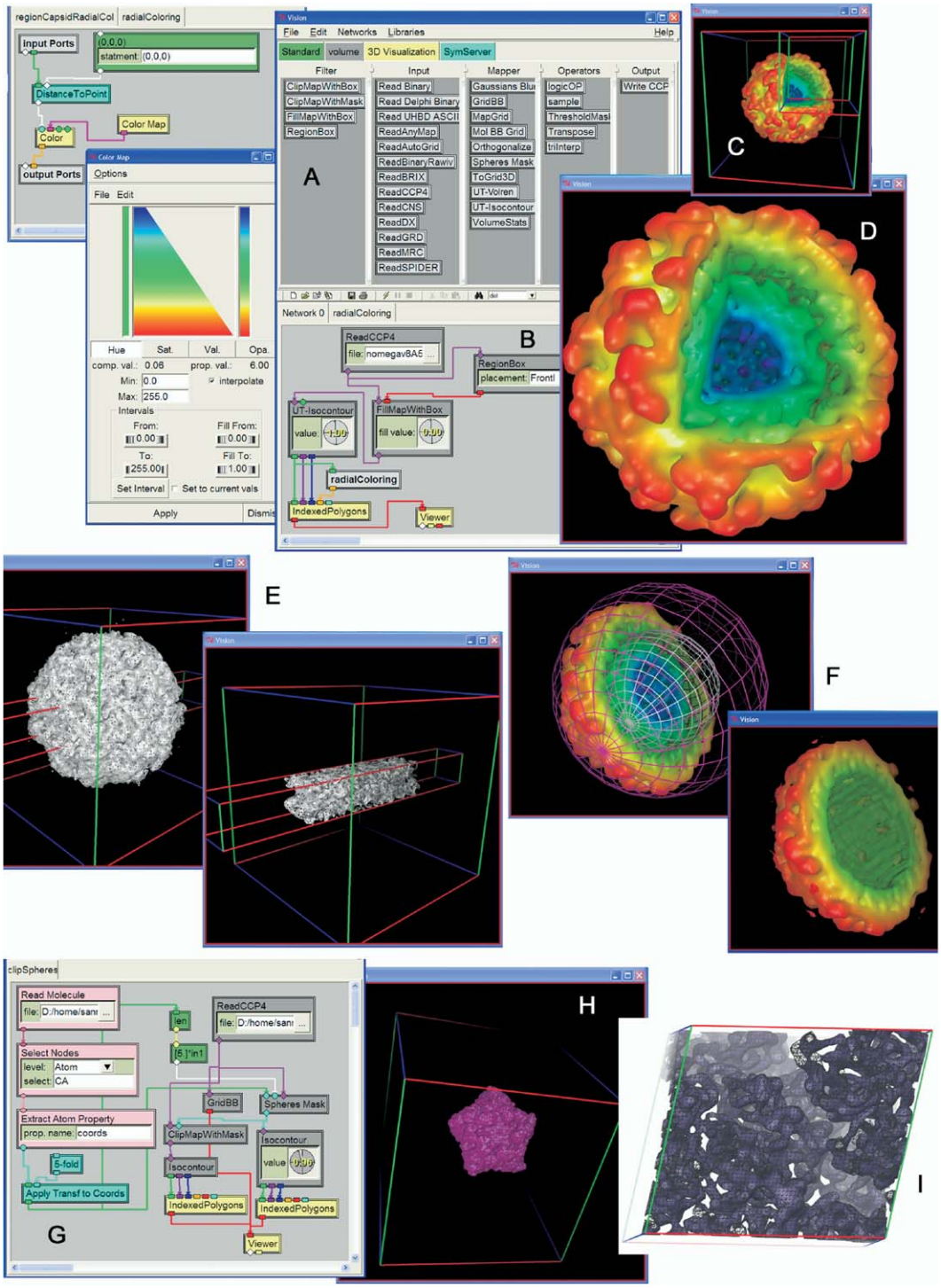

Figure 2. The Volume Software Component Used in Vision

(A) The functionality of the Volume software component is exposed as a library of Vision nodes. (B) An example of a network using nodes from the volume library. A density map for the N-omega viral capsid is read from a file in the CCP4 format. The upper-right-front octant of the grid is selected (C) using a "RegionBox" node and the density values in this octant are overwritten with the value 0.0 by the "FillMapWithBox." The resulting grid is iso-contoured and colored using a blue to red ramp by the "radialColoring" macro node (D). (E) Volume data can be clipped using axis-aligned boxes. (F) Complex clipping masks can be created using logical operations on mask grids. Here, a mask grid is computed for two spheres with the same center. The masks have value 1 inside the spheres and 0 outside. An XOR operation creates a spherical slab mask with values 1 only in-between the 2 spheres. (H) A mask grid is computed for the union of a set of spheres, in this case all atoms in the five copies of the asymmetric unit of the N-omega virus. Masks can be used to zero-out density on grids everywhere outside the mask. (I) The edge lengths and angles of crystallographic cells are stored along with the 3D grid of data. This information is used to properly render volumetric data crystallographic cells that are not orthogonal.

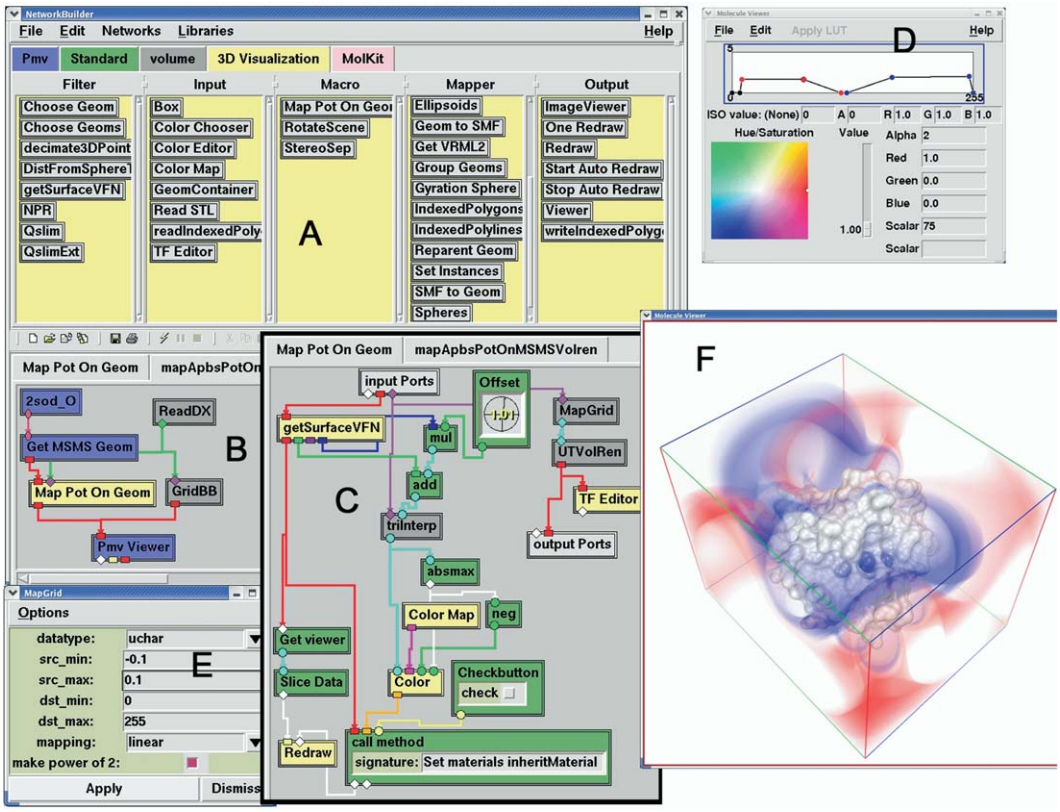

Figure 3. Immediate Rendering Mode to Create

(A) The 3D Visualization library exposes the functionality of the DejaVu component. The "Viewer" node provides a fully fledged OpenGL-based 3D geometry renderer (F). (B) This network obtains a handle to the molecular surface of a molecule loaded in the PMV application, reads an electrostatic potential calculated using APBS (Holst et al., 2000). The macro node "Map Pot On Geom" calculates points at a given offset from the surface along the normal vector of each vertex of the surface. The potential is looked up at these points by the "trilnterp" node and mapped onto the surface using a red, white, and blue color map. The electrostatic grid is also volume rendered using the transfer function shown in (D).

components are written entirely in the Python programming language, while others use wrapper functions to expose functionality implemented in C and C++ libraries to the Python interpreter. Small amounts of Python code are needed to glue these components together and allow them to interoperate seamlessly with all other components.

Most of these components have been developed independently of each other, which is a key design feature and makes them reusable in a broader context than structural biology. For instance we have used DejaVu to visualize evolutionary trees of protein mutations (Stoffler et al., 2002). Similarly, the ViewerFramework component has been reused to develop a heart model viewer for visualizing cardiac simulation data (Front-end to the Continuity simulation code, A. McCul-

Table 2. Python Packages Developed in Our Laboratory

Package	Description
MolKit	Read, write, and build hierarchical representation of molecular data structures.
ViewerFramework	Visualization application template. Uses DejaVu for rendering 3D geometry.
PyBabel	Reimplementation of Babel 1.6 (molecular file formats conversion). Supports assigning of atom type and bond order, ring detection, Gasteiger charges, and protonation.
Mglutil	Various packages containing mathematical functions, GUI widgets, etc.
FlexTree	Provides hierarchical, high-level representations of molecular flexibility.
PyQslim	Python wrapper of the polygonal mesh decimation library QSlim (Garland, 1999).
Mslib	Python wrapper of the MSMS library (Sanner et al., 1996) for computing molecular surfaces.
Isocontour-UT	Python wrapper of a C++ library for fast-iso-contouring (Bajaj et al., 1996).
SFF	Python wrapper of a C-implementation of the AMBER Force Field (Simple Force Field).
SpatialLogic	Python wrapper of a C library performing BSP-Tree-based CSG operations.
Gle	Python wrapper of the GL Extrusion library (Vepstas, 1991)

The first five packages listed are platform independent (i.e., written entirely in Python).

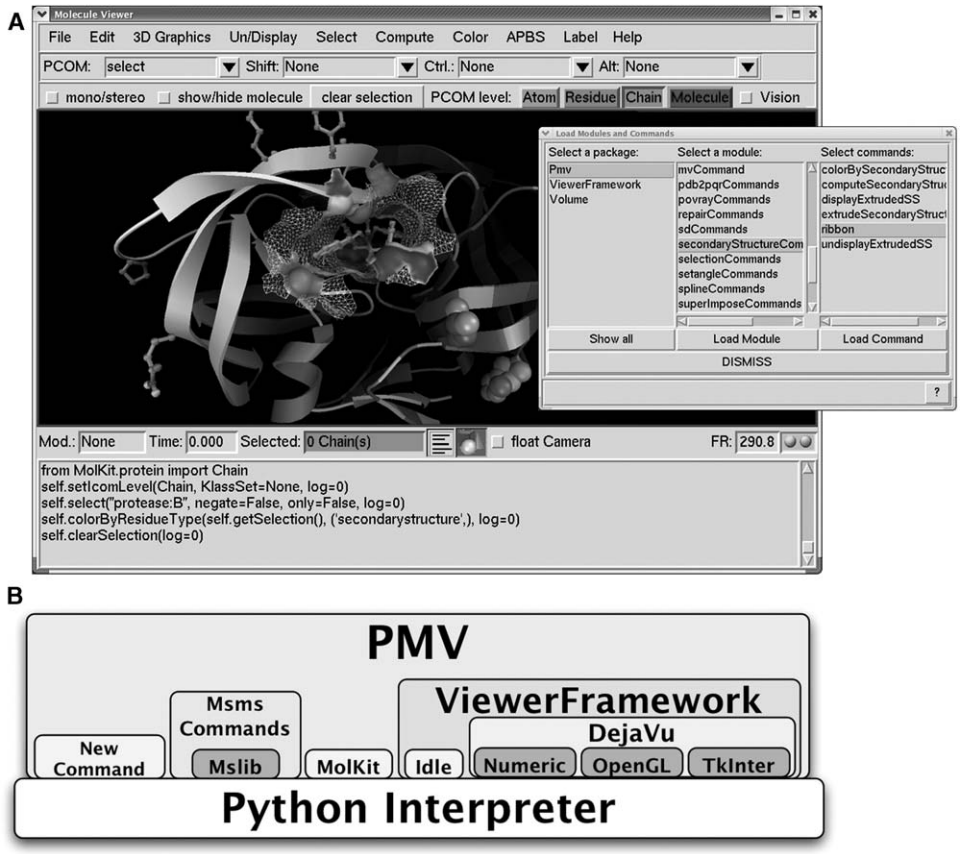

Figure 4. PMV: A Component-Based Molecular Viewer

(A) The PMV application is a fully-fledged molecular visualization environment.

(B) The architectural view of PMV. Note the number of software components that are also used in Vision (Figure 1I).

loch, personal communication). These components are the basic building blocks from which several domain-specific applications have been developed. PMV (Figure 4A) is a molecule viewer built from the components described above. It relies on the MolKit component for reading, writing, querying, and representing molecules in memory. PMV is extendable and provides support for developing molecular visualization and manipulation commands at a high level. PMV offers most features available in standard molecular visualization applications as well as several unique features, including unlimited undo, automatic logging of commands, Python-based scripting, and the ability to bind any command to mouse picking events. PMV currently has parsers for PDB, MOL2, PQR, PDBQ, and PQDBQS file formats. An mmCIF parser is under development and an XML parser is planned. In PMV, every graphical representation of a molecule (i.e., DejaVu geometry objects) is connected to the molecule it represents. This feature enables PMV commands to operate on partial geometries corresponding to molecular fragments such as chains, or lists of amino acids or atoms, for instance. The most innovative aspect of PMV is that it is built from reusable software components (Figure 4B). PMV forms a generic platform for developing graphical front-ends to computational methods dealing with molecules. AutoDockTools (ADT) is an example of specializing PMV to create a graphical front-end to the automated docking code AutoDock. ADT leads the user through the preparation of input files, helps launch AutoDock calculations, and provides tools for visualizing and analyzing the results of these calculations. PMV, ADT, and Vision have been distributed to over 9000 scientists around the world.

Applications

In this section, several examples are used to demonstrate how the software components described above are combined to visualize large macromolecular assemblies.

A Simple Visualization of a Large Viral Capsid

The goal of this first example is to build and display a low-resolution representation of the complete capsid of the bluetongue virus. We start by loading the 15 coat proteins composing this asymmetric unit of the viral capsid (2btv.pdb) into PMV (Figures 5A and 5B). A single molecule containing 15 chains (i.e., one for each protein subunit) is created in PMV. Using the PMV command "colorByChains" we assign different colors to the line representation of each protein. Since PMV currently has no command for automatically building low-resolution representations of molecular shapes, we start the

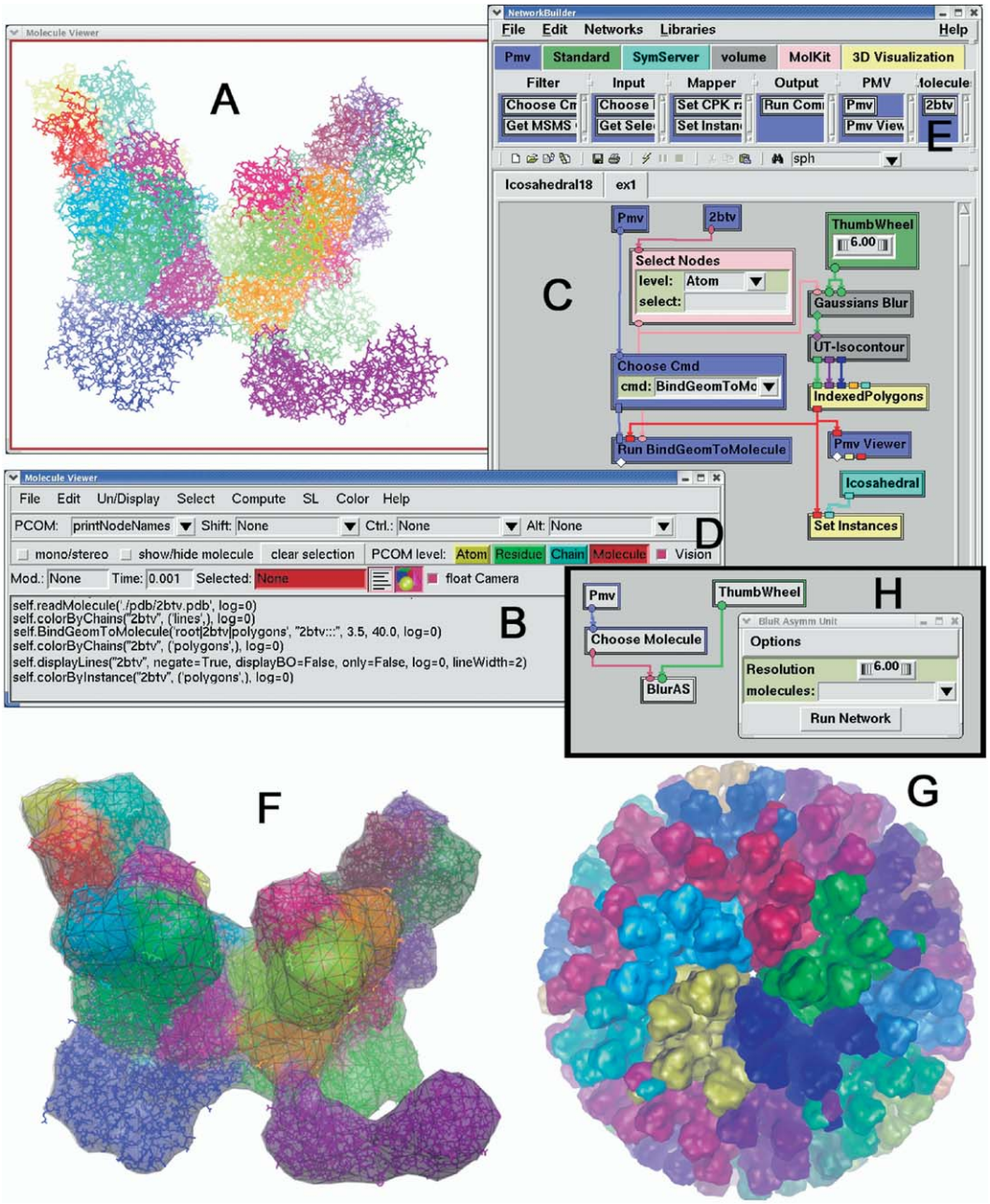

Figure 5. Visualization of the Viral Capsid of the Bluetongue Virus

(A and B) The 15 proteins of the asymmetric unit are loaded into PMV, displayed as lines, and colored by chain (each protein subunit is stored as a chain in the pdb file). (C) The Vision network to compute the low-resolution surface of the asymmetric unit shown in (F). This surface is obtained by blurring atoms onto a grid and computing an iso-surface. The surface is translucent, showing the line representation of the proteins behind the mesh. The surface has been connected to the underlying atoms by the network shown in (C). The PMV command "colorByChains" was used to colorize the surface by protein subunit. (G) The full capsid is generated, duplicating the geometry shown in (F) 60 times. The transformation matrices are provided by the "Icosahedral1" node from the "SymServ" library. The surfaces have been colored by instance in (G) (i.e., one color per copy) using PMV's "colorByInstance" command. (H) An example of encapsulating a network into a macro node and creating a high-level graphical user interface. The network shown in (C) was placed inside the macro node named "Blur AS" and a panel was created to expose the blurring resolution parameter. A molecule chooser node was added to allow selecting the molecule for which to compute a low-level representation. The Vision environment and its networks can be hidden, leaving only this simplified user interface visible to the user.

Vision environment (Figure 5C) to build a computational network for creating such a representation. Vision can be started in PMV using the check-button labeled "vision" in the PMV button bar (Figure 5D). When Vision is started from PMV, a special library of nodes called "Pmv" is created to expose Python objects from the

PMV application in the Vision environment (Figure 5E). Molecules loaded into PMV will automatically appear as nodes under the "Molecules" category in this library. Dragging and dropping the "2btv" node from this library onto the canvas provides a handle to this particular molecule in the Vision environment. The "Gaussian Blur" node from the Volume library allows accumulating Gaussian distributions placed on atoms. We use the "Select Node" from the MolKit library to transform the molecule output by the 2btv node into a list of atoms, which is then fed into blurring node. The Thumbwheel node controls the blurring of the atoms by setting the grid spacing and the half width of the Gaussian distribution. The result of the blurring operation is a Grid3D object, which is handed to an iso-contouring node. This node computes an iso-surface for a user-specified contour value and outputs the surface as lists of 3D vertices, triangular faces indices, and normal vectors. These lists are converted into a DejaVu surface geometry object by the "IndexedPolygons" node. This geometry object can be displayed in any DejaVu Viewer. In this example, we add the surface geometry to the "PmvViewer" from the Pmv library in order to display the surface in the viewer used by PMV to display its molecules. Because of the coarse resolution of the grid used in this example (6 angstroms spacing), the surface is made of less than 10,000 triangles, which are easily displayed using today's graphics hardware.

A second branch of the network is used to bind the iso-surface to the molecule. PMV has a command for efficiently finding the closest atom for each vertex of any given geometry. To invoke this command in our network, we use the Pmv node to obtain a handle to the PMV application. The "Choose Cmd" node discovers dynamically all commands currently loaded in PMV through introspection of the PMV application. We use this node to select the "bindGeomToMolecule" command and pass it to a "Run Command" node from the Pmv library. This node renames itself after the PMV command it represents. Using introspection on the command, it detects the signature of the command (i.e., the arguments that can be given to the command and their default values), and configures itself to expose these arguments as input ports. For arguments that have a default value, the node automatically generates appropriate widgets (e.g., dials for floating point and integer values, check-buttons for Booleans, entry fields for strings, etc.) and binds them to the corresponding input port. In the case of the bindGeomToMolecule command, only two arguments have no default value. These arguments are the geometry to be bound and the list of atoms to be used for binding the geometry. Values for these two arguments can be provided to the node through input ports on the node. Connecting the iso-surface geometry to the second input port and the list of all atoms in the asymmetric unit to the third input port of the node triggers the execution of the node and of the PMV command.

Any PMV command can be incorporated into a Vision network using this mechanism. After the bindGeomToMolecule command has completed, it is possible to operate on the iso-surface geometry using PMV commands. For example in Figure 5F we have colored the surface using the same colorByChains command used initially to color the line representation of the molecule, but this time we applied it to the polygonal surface. Similarly, we could use PMV's display command to selectively show or hide parts of the surface corresponding to selections of atoms. Obviously, this binding is somewhat arbitrary at the atomic level; however it is meaningful at coarser levels, such as the amino acid-, secondary structure-, or chain-level. Although any subset of atoms could be used to bind a geometry object to a molecule, we chose to bind the geometry to all atoms for the sake of simplicity of the network.

Finally, to obtain a complete capsid (Figure 5G) we need to display 60 copies of the computed iso-surface, organized according to an icosahedral symmetry. The 60 transformation matrices defining such an arrangement are provided by the "Icoshadral1" macro node from the SymServer library. These matrices are set as instance matrices for the iso-surface geometry using the "Set Instances" node. The surfaces in Figure 5G are colored by instance (i.e., one color for each copy) using PMV's "colorByInstance" command which was invoked through PMV's menu of commands.

If this particular graphical representation of the capsid is deemed useful, the network for creating it for any icosahedral viral capsid can be encapsulated into a macro node (Figure 5H) and added to a library for later use. A new macro node can be created through the Edit pull down menu in Vision, and nodes can be placed inside the macro by cut and paste operation on selected nodes. For this network, all nodes except for the molecule (i.e., the 2btv node) would be placed inside the macro and the input port of the Select Nodes node exposed as an input port of the macro node. A user-interface panel for exposing parameters controllable by the user can also be created, and widgets from nodes in the network can be moved to this panel. In the example shown in Figure 5H, the thumbwheel controlling the resolution of the blurring operation has been moved to the user panel. We also added a Pmv node and a "*choose Molecule*" node to allow the user to pick a molecule for which to compute this representation. We can move the molecule selection widget from the Choose Molecule node to the user interface panel. Once such a simplified user interface is created, the Vision environment and the network can be hidden completely from users.

Refining the Representation to Gain Flexibility in the Visualization

The capsid of the bluetongue virus is made of an inner shell containing the protein subunits "A" and "B," and an outer shell containing the 13 other protein subunits. Blurring the individual proteins and computing a separate surface for each of them provides more flexibility for the visualization of the assembly.

The network shown in Figure 6A creates such a representation. The two "*GeomContainer*" nodes are used to create the "*inner Shell*" and "*outer Shell*" objects in the DejaVu geometry hierarchy (Figure 6B). These nodes will serve as parents in the DejaVu hierarchy of objects for the low-resolution surfaces generated for the individual proteins. The two Select Nodes extract the proteins of the inner shell (i.e., chains A and B), and the proteins of the outer shell (i.e., the 13 other chains). These lists of chains are each fed into "blur

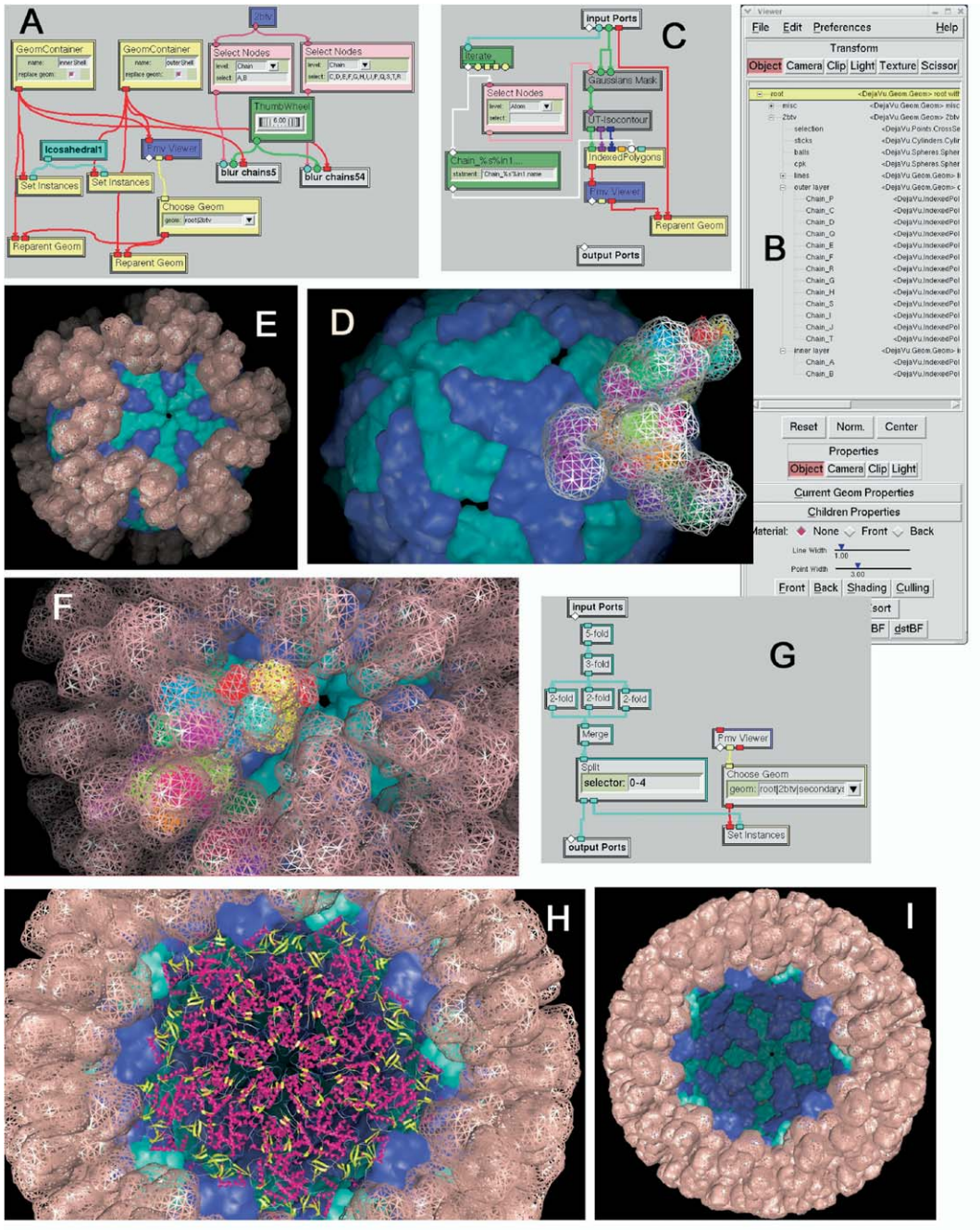

Figure 6. Refinement of the Representation of the Bluetongue Viral Capsid

These networks and representations are discussed in the application section of the paper.

chains" macro nodes (Figure 6C). This macro iterates over a list of chains and for each one creates a pseudo density map and computes an iso-surface. The "Chain_%s" node is an "Eval" node from the Standard library used to create a unique name for the surface of each subunit. The inner and outer shell geometry objects are each passed into one of the blurring macros and are used to make each iso-surface a child of the right container object. After executing the network, the inner and outer shell containers will have been added under the 2btv node in the DejaVu hierarchy (Figure 6B) and will respectively contain 2 and 13 children geometry.

Note that in this network, we have chosen to apply the 60 transformation matrices to the container objects rather than the individual surfaces (Figure 6A, Icosahedral1 node). These transformation matrices are automatically inherited by the children geometry of the containers. The hierarchical nesting of geometry objects allows a user to instantly show or hide a particular shell by selecting a container object in the DejaVu Hierarchy

and switching its visibility flag on or off. This flag is found in the pull down menu named "Current Geom Properties" (Figure 6B). Having individual geometry objects for each protein also enables altering of the visual representation of all symmetry-related copies of a particular protein (visible or not, solid rendering or lines, color, transparency, etc.). All these controls are available in the viewer's GUI (Figure 6B). Examples of possible representations are given in Figures 6D–6F.

A "Split" node can be used to extract a subset of the transformation matrices (Figure 6G). This node can extract any number of matrices and output them on separate ports. In the example shown in Figure 6G, the five first matrices from the stream, corresponding to a pentamer in the capsid, are extracted and output on the second port while the remaining 55 matrices are output on the first port. Figure 6I shows the result of using only the 55 matrices, while in Figure 6H, the five extracted matrices are used as instance matrices on the secondary structure geometry.

The models shown in Figures 6 draw interactively at frame rates ranging from 2 to 20 frames per second, depending on the rendering modes (i.e., lines versus shaded, monochrome versus colored, etc.) on a Dell Linux desktop workstation with an nVidia Quadro FX 1000/AGP graphics card and 2 MB of memory. We also built and displayed these models on a Sony laptop running windows XP with 1 MB of memory and a Radeon 7000 graphics card. On this machine, update rates were somewhat lower because the graphics card is less powerful. A key element of our approach is the flexibility provided to adapt the complexity of the model to the available computational resources. Frame rates can easily be increased by simplifying the geometry.

Alternative Low-Resolution Surfaces

It is trivial to replace the part of the network performing the blurring of the atoms and computing the iso-surface by an "MSMS" node from the MolKit library, which computes a molecular surface using the MSMS library (Sanner et al., 1996). While these surfaces provide a much more accurate description of the shape and interfaces of the proteins in the capsid, even with very low density of surface points, the number of triangles in these surfaces rapidly overwhelms the capabilities of graphics hardware. Molecular surfaces calculated for each protein in an asymmetric unit at densities 0.5, 1.0, and 3.0 contain a total of 434,000; 636,000; and 1.55 million triangles, respectively. When duplicated 60 times, these numbers range from 16 to 93 million triangles. Such a large number of triangles cannot be rendered interactively with today's graphics hardware. However, great advances have been made recently in the field of mesh simplification and several efficient algorithms for decimating meshes while preserving certain characteristics such as genus, topology, and shape have become available. We have wrapped the QSlim library (Garland, 1999) which allows the efficient decimation of meshes and properties. Figure 7 shows a comparison of original MSMS surfaces, QSlim-decimated surfaces, and blurred densities iso-surfaces for protein subunits A and B. The MSMS surfaces in the top row are calculated using a probe radius of 1.5 Å and a density of 3.0 and 0.5 vertices per Å^2, respectively (Figures 7.1 and 7.2). The MSMS surfaces computed for a density of 3.0 were decimated using QSlim to 10% (Figure 7.3) and 1% (Figure 7.4) of their original number of triangles. The Gaussian blur method was applied to the two subunits with a grid spacing of 2.48 and iso-contoured at a level of 1.0 (Figure 7.5) and 5.0 (Figure 7.6). Finally, pseudo density maps were calculated for a grid spacing of 6 Å and iso contoured at a level of 1.0 (Figure 7.7) and 5.0 (Figure 7.8).

The number of triangles in the MSMS surfaces cannot be reduced to a manageable count using only the vertex density parameter in the MSMS calculation. The decimation algorithm allows drastic reduction of the number of triangles while maintaining a good accuracy of the shape. The blurred surfaces smooth out details which seem to be better retained in the decimated molecular surfaces, suggesting that decimated molecular surfaces could provide more accurate representations of molecular interfaces. The blurred surfaces, however, provide visually appealing surfaces at very low resolutions. Finding the best representation for different purposes is an open research question which we plan to investigate. We feel that the ability to easily switch from one representation to another and explore new ones is a great advantage. The goal here was to illustrate the ease with which such comparisons are made possible in the described modular and component-based software environment.

Software Availability

All the software described in this paper is freely available for download at http://www.scripps.edu/~sanner/software. Several components (DejaVu, MolKit, Pmv, symserv, AutoDockTools, etc.) have already been distributed to over 9000 users. More recent components such as the Volume package have only become available in our latest release. Since this code is under constant development, some features described in this paper are more recent than the last release available online. However, our latest distribution contains CVS entries allowing any package to be updated directly from our CVS source tree. Instructions for updating the components are available at our software distribution web site. After installing our latest release and updating all components, all the functionality described in this paper is available.

All of our software components are free for academic use. A small subset of the components have restrictions for commercial applications. We offer two software bundles on our download site: the *academic* bundle, which contains all the components; and the *commercial* bundle which is stripped of the components with restrictions for commercial applications. Only functionality relying on the missing components is disabled in the free commercial bundle.

Conclusion

We have presented several software components and discussed specific features in these packages that are relevant to the visualization of large molecular assemblies. In particular, we have demonstrated (1) exploitation of symmetry information for efficiently representing molecules in memory, while retaining the ability to change the visual representation of any symmetry-related copy

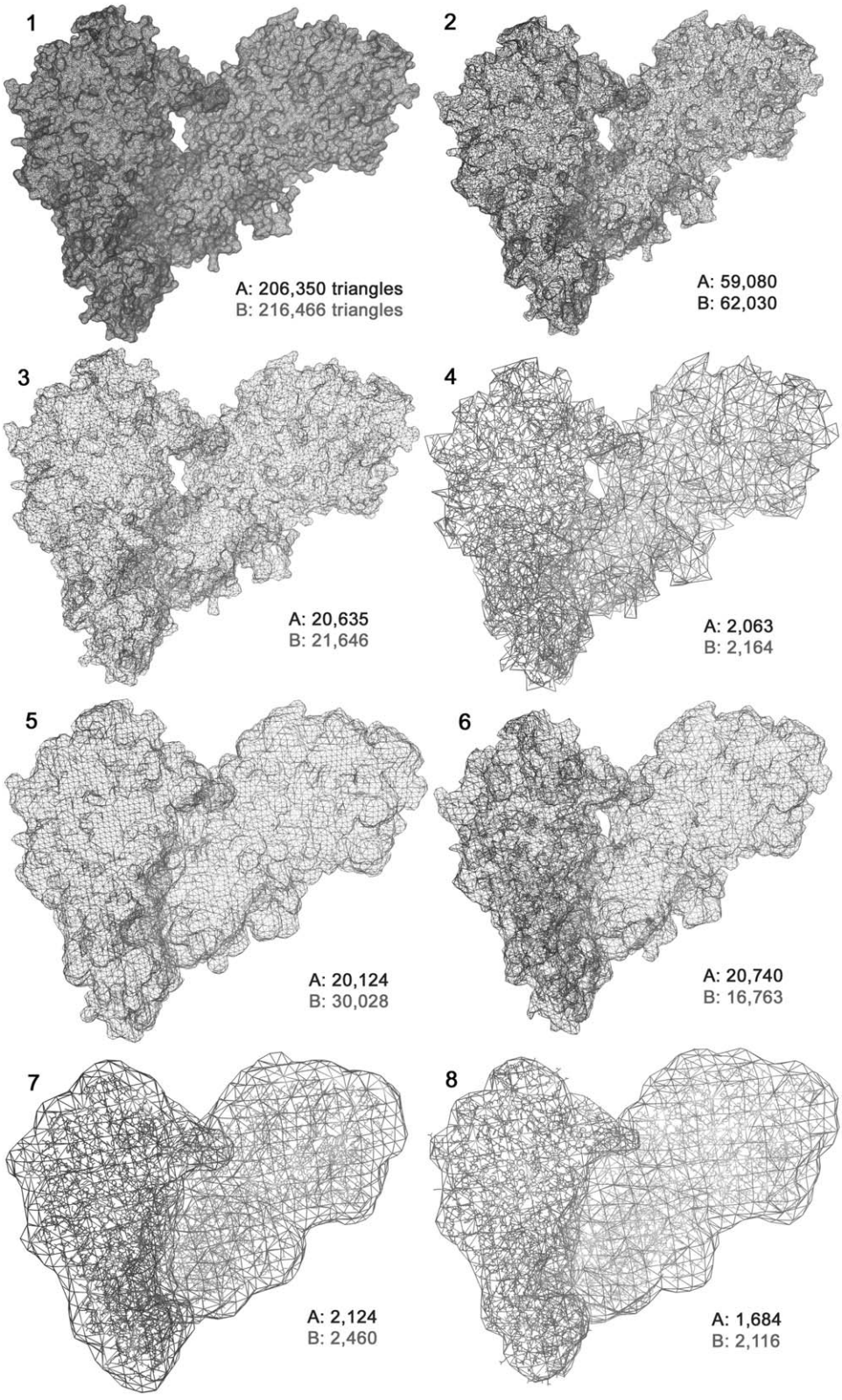

Figure 7. Variations on Molecular Shapes

Surfaces computed using different methods are shown for subunits A (molecule on the left) and B (molecule in the right) of the bluetongue viral capsid. The upper row shows molecular surfaces computed using MSMS with densities of 3.0 (7.1) and 0.5 (7.2). The surfaces calculated with density 3.0 have been decimated to have 10% (7.3) and 1% (7.4) of their original triangle count. Blurred pseudo-densities were calculated for a grid spacing of 2.48 (7.5 and 7.6) and iso-contoured at level 1.0 in Figure 7.5 and 5.0 in Figure 7.6. In figures 7.7 and 7.8, iso-surfaces of pseudo densities calculated for a grid spacing of 6.0 were iso-contoured at levels 1.0 in (7.7) and 5.0 in (7.8).

of a protein in the capsid; (2) variations of simplified graphical representations of the assembly based on pseudo-density maps, and decimated molecular surfaces; (3) mechanisms for organizing the geometric objects in the capsid in a logical manner which facilitates user interaction with the graphical representation; and (4) the integration of low-resolution representations into the PMV molecular visualization environment enabling the inspection and visualization of molecular interactions at an atomic level of detail. In addition, we have described a highly flexible software environment built from software components. This environment enables the exploration of novel representations for large assemblies by interactively combining software components and the functionality they provide into computational networks. We have demonstrated the use of this environment for exploring possible representations of large biological assemblies and for capturing successful approaches and exposing them with a user-friendly interface to biologists. We plan to use the described software to explore and compare multiresolution representations of molecular shapes and properties, to identify effective mechanisms for navigating through these complex assemblies, to determine classes of generally useful representations, and to encapsulate them into easy-to-use commands.

Acknowledgments

The results presented in this paper leverage technology developed under several projects by a number of people including Dr. Daniel Stoffler who developed the Symmetry server component and is a co-author of the Vision environment; Sophie Coon, the main developer of PMV; Anna Omelchenko, who wrapped several C and C++ libraries including the iso-contouring, volume rendering, and surface decimation software; and Dr. Gabriel Lander who contributed greatly to the Volume software component. We are grateful to Dr. Arthur Olson for his continued support, and numerous discussions and suggestions, and to Dr. Chandrajit Bajaj for providing us with the iso-contouring and volume rendering libraries. This work has been funded by NIH through the National Biomedical Computation Resource (NBCR) grant RR08605 and the National Partnership for Computing Infrastructure (NPACI) grant NSF, CA ACI9619020. Finally, we would like to thank Dr. Garrett M. Morris for his careful reading of the manuscript and numerous suggestions.

Received: November 23, 2004
Revised: January 20, 2005
Accepted: January 24, 2005
Published: March 8, 2005

References

Bajaj, C., Djeu, P., Siddavanahalli, V., and Thane, A. (2004). TexMol: interactive visual exploration of large flexible multi-component molecular complexes. Proceedings of IEEE Visualization 2004 (VIS'04), Austin, Texas.

Bajaj, C., Park, S., and Thane, A.G. (2002). Parallel multi-PC volume rendering system. CS & TICAM Technical Report, University of Texas at Austin.

Bajaj, C., Pascucci, V., and Schikore, D. (1996). Fast isocontouring for improved interactivity. Proceedings of ACM Siggraph/IEEE Symposium on Volume Visualization, San Francisco, CA.

Berman, H.M., Westbrook, J., Feng, Z., Gilliland, G., Bhat, T.N., Weissig, H., Shindyalov, I.N., and Bourne, P.E. (2000). The Protein Data Bank. Nucleic Acids Res. *28*, 235–242.

Duncan, B.S., and Olson, A.J. (1995a). Approximation and visualization of large-scale motion of protein surfaces. J. Mol. Graph. *13*, 250–257.

Duncan, B.S., and Olson, A.J. (1995b). Biomolecular visualization using AVS. J. Mol. Graph. *13*, 271–282.

Duncan, B.S., and Olson, A.J. (1995c). Texture mapping parametric molecular surfaces. J. Mol. Graph. *13*, 258–264.

Garland, M. (1999). QSlim simplification software http://www-2.cs.cmu.edu/afs/cs/user/garland/www/quadrics/qslim.html

Gelder, A.V., and Kim, K. (1996). Direct volume rendering with shading via three-dimensional textures. Proceedings of Symposium on Volume Visualization *96*, 23–30.

Gillet, A., Goodsell, D., Sanner, M.F., Stoffler, D. (2004a). A tangible model augmented reality application for molecular biology. IEEE Visualization Vis04, Austin Texas.

Gillet, A., Goodsell, D., Sanner, M.F., Stoffler, D. (2004b). Computer-linked autofabricated 3D models for teaching structural biology. SigGraph 2004, Los Angeles, CA.

Greenfield, P. (2003). numarray: a new scientific array package for python. http://www.stsci.edu/resources/software_hardware/numarray

Holst, M., Baker, N., and Wang, F. (2000). Adaptive multilevel finite element solution of the Poisson-Boltzmann equation I. Algorithms and examples. J Comput Chem (USA) *21*, 1319–1342.

Honig, B., and Nicholls, A. (1995). Classical electrostatics in biology and chemistry. Science *268*, 1144–1149.

IBM (2002). Open visualization data explorer. OpenDX. http://www.research.ibm.com/dx/

Ilin, A., Bagheri, B., Scott, L.R., Briggs, J.M. (1995). Parallelization of poisson boltzmann and brownian dynamics calculations. ACS Symposium Series: Parallel Computing in Computational Chemistry.

IrisExplorer (2003). IrisExplorer. http://www.nag.com/welcome%5Fiec.html

Lutz, M., and Asher, D. (1999). Learning Python (Sebastopol, CA: O'Reilly & Associates).

Macke, T.J., Duncan, B.S., Goodsell, D.S., and Olson, A.J. (1998). Ineractive modeling of supramolecular assemblies. J. Mol. Graph. *16*, 115–120.

Morris, G., Goodsell, D., Halliday, R., Huey, R., et al. (1998). Automated docking using a Lamarckian genetic algorithm and an empirical binding free energy function. J. Comp. Chem. *19*, 1639–1662.

Oliphant, T. (2003)) SciPy. http://www.scipy.org/

Parker, S.G. (1999). The SCIRun problems solving environment and computational steering system. Dept. of Computer Science, University of Utah.

Sanner, M.F. (1999). Python: a programming language for software integration and development. J. Mol. Graph. Model. *17*, 57–61.

Sanner M.F. (2005). Using the python programming language for bioinformatics. In Encyclopedia of Genomics, Proteomics and Bioinformatics (New York, Addision-Wesley, John Wiley & Sons, Ltd.), in press.

Sanner, M.F., Olson, A.J., and Spehner, J.-C. (1996). Reduced surface: an efficient way to compute molecular surfaces. Biopolymers *38*, 305–320.

Sanner, M.F., Stoffler, D., and Olson, A.J. (2002). ViPEr, a visual programming environment for Python. Proceedings 10th International Python Conference, Alexandria, VA.

Sheehan, B., Fuller, S.D., Pique, M.E., and Yeager, M. (1996). AVS software for visualization in molecular microscopy. J. Struct. Biol. *116*, 99–106.

Stoffler, D., Coon, S.I., Huey, R., Olson, A.J. (2003). Integrating biomolecular analysis and visual programming: flexibility and interactivity in the design of bioinformatics tools. Proceedings of the Thirty-Sixth Annual Hawaii International Conference on Systems Sciences, Waikoloa, Hawaii. Computer Society Press.

Stoffler, D., Sanner, M.F., Morris, G.M., Olson, A.J., and Goodsell,

D.S. (2002). Evolutionary analysis of HIV-1 protease inhibitors: methods for design of inhibitors that evade resistance. Proteins *48*, 63–74.

Subramaniam, S., and Milne, J.L.S. (2004). Three-dimensional electron microscopy at molecular rresolution. Annu. Rev. Biophys. Biomol. Struct. *33*, 141–155.

Upson, C., Faulhaber, T., Kamins, D., and Laidlaw, D. (1989). The application visualization system: a computer environment for scientific visualization. IEEE Comput. Graph. Appl. *9*, 30–42.

Vepstas, L. (1991). GL extrusion library http://linas.org/gle/