FPV : Fast Protein Visualization Using Java 3D™

Tolga Can Yujun Wang Yuan-Fang Wang Jianwen Su

Department of Computer Science,
University of California, Santa Barbara, CA 93106-5110, U.S.A
{tcan,yjwang,yfwang,su}@cs.ucsb.edu

ABSTRACT

We have developed a protein visualization system based on Java 3D™. Java 3D provides compatibility among different systems and enables applications to be run remotely through web browsers. However, using Java 3D for visualization has some performance issues with it. The primary concerns about molecular visualization tools based on Java 3D are in their being slow in terms of interaction speed and in their inability to load large molecules. This behavior is especially apparent when the number of atoms to be displayed is huge, or when several proteins are to be displayed simultaneously for comparison. In this paper we present techniques for organizing a Java 3D scene graph to tackle these problems. We demonstrate the effectiveness of these techniques by comparing the visualization component of our system with two other Java 3D based molecular visualization tools. In particular, for van der Waals display mode, with the efficient organization of the scene graph, we could achieve up to eight times improvement in rendering speed and could load molecules three times as large as the previous systems could.

Keywords

Protein visualization, Performance analysis, Java 3D, Scene graph optimization

1. INTRODUCTION

Protein visualization has become an important research topic, especially in light of the accomplishment of the Human Genome Project [3]. The ability to visualize the 3D structure of proteins is critical in many areas such as drug design and protein modeling. This is because the 3D structure of a protein determines its interaction with other molecules, hence its function, and the relation of the protein to other known proteins. For example, hemoglobin's cup shape, which accommodates the oxygen-binding heme group, suggests its ability to carry oxygen in the bloodstream. There are many well established ways of visualizing the 3D protein structures. Each way of visualization highlights a different aspect of the protein molecule, as mentioned by Clay Shirky [13].

Growing number of new structure data in Protein Data Bank open new ways for collaboration, thus emphasizes the need for visualization tools that are portable. Moreover, studying the interaction between protein molecules may also require visualizing huge numbers of atoms, thus researchers also need tools that are capable of loading and displaying this huge amount of data.

In this paper, we describe in detail our protein visualization system, which is built using the Java 3D API [1]. There is growing trend in adopting the Java™ technology in the fields of bioinformatics and computational biology [11]. The main advantages of Java are its compatibility across different systems/platforms and having the ability to be run remotely through web browsers. Using Java 3D as a graphics engine has also the additional advantage of rapid application development, because Java 3D API incorporates a high-level scene graph model that allows developers to focus on the objects and the scene composition. Java 3D also promises high performance, because it is capable of taking advantage of the graphics hardware in a system. The speed observed should depend on the quality of the graphics hardware on the machine. However, a common complaint about visualization systems based on Java 3D is their being slow in terms of interaction speed even with a good graphics hardware accelerator. Also memory errors may be seen even with a small number of objects. The reason for these anomalies may be the developer himself (constructing a bad scene graph) or certain limitations of the Java 3D API, which is discussed below.

The Java 3D API implementations are layered on top of the existing lower-level immediate-mode [4] 3D rendering APIs, such as OpenGL and Direct3D. Java 3D is fundamentally a scene-graph-based API. Most of the constructs in the API are biased toward retained mode and compiled-retained mode rendering [10]. Java 3D itself also offers immediate-mode rendering if a developer wants more control and flexibility. The programmer can ignore the scene graph structure and send the graphical constructs directly to the renderer. However, in immediate mode, Java 3D has no high-level information concerning graphical objects or their composition. Because it has minimal global knowledge, Java 3D can only perform localized optimizations on behalf of the programmer. Thus, using immediate-mode directly may cause drastic performance drops. Using a scene-graph-based de-

[1]http://java.sun.com/products/java-media/3D/

velopment scheme a developer should expect better performance, but some molecular scenes (e.g. containing too many atoms) may require too much memory or computation time. Thus, performance drops occur because of an heavyweight scene graph. In this paper we propose techniques to create efficient scene graph structures, which allow loading large molecules (more than 4000 amino acids) and render them in an acceptable interactive speed. We demonstrate that by carefully organizing the scene graphs, our system achieves an interactive rendering speed eight times faster and is able to load molecules three times larger than the other systems, namely JMV[2] and JMVS2[3].

The remainder of this paper is organized as follows: The overview of the system components is described in Section 3. We explain how different 3D visual representations are created from PDB data in Section 4. We present the techniques for speeding up interaction and implementation details in Section 5. Performance comparison tests and their results are in Section 6. Finally we summarize our work and discuss future research direction in Section 7. FPV is freely available with source code at the following URL:

http://www.cs.ucsb.edu/˜tcan/fpv/.

2. RELATED WORK

Many tools have been developed to visualize a protein whose structure has been determined. In this section we will talk about a subset of these tools, which are closely related to our molecular visualization system. One of the earliest of those tools is Roger Sayle's RasMol [12]. RasMol is now being developed under the name of Protein Explorer. SwissPdbViewer [5], which is tightly linked to the automated protein modeling server Swiss-Model, provides a user-friendly interface to analyze several proteins at the same time. MOLMOL [8] is another molecular graphics program for the display, analysis, and manipulation of the 3D structures of biological macromolecules, with special emphasis on nuclear magnetic resonance (NMR) solution structures of proteins and nucleic acids. Most of these programs are implemented using C language and OpenGL API and they have relatively large user communities.

There are relatively few protein visualization tools which were developed using Java and the Java 3D API. MOLVIE [4], a molecular visualization environment, is one of them. WebMol [14] is a protein structure viewing and analysis program, which has more functionality, but limited 3D model types. These two programs do not use the Java 3D API; instead they use their own graphics constructs based on Java.

JIMD Interactive Molecular Dynamics with Java[5], is being developed using Java 3D, but their focus is on molecular dynamics and simulation. Tripos Java3D Molecule Viewer [6], is a new tool currently under development. JMV and JMVS2 (two systems we used for performance comparison) are molecular visualization tools and offer a variety of 3D representations and display options. JMV is developed by the Theoretical Biophysics Group in the Beckman Institute for Advanced Science and Technology at the University of Illinois at Urbana-Champaign with NIH support. These two tools have very similar functionality compared to our molecular visualization system. One advantage of JMV over JMVS2 is that it is being build as a toolkit, so that other developers can use it as part of their systems.

Molecular Biology Toolkit[7] is another general toolkit that includes visualization components based on Java3D. However, it is still an ongoing work and right now no visualization application using this toolkit is available for evaluation and testing purposes.

[2]http://www.ks.uiuc.edu/Development/jmv/
[3]http://www.adcworks.com/projects/jmvs
[4]http://guanine.cs.ucsb.edu/Molvie/
[5]http://www.gwdg.de/˜ovormoo/jimd/
[6]ftp://ftp.tripos.com/pub/java3d

3. SYSTEM OVERVIEW

Figure 1 shows the main components of our visualization system. The **Main Event Handler Module** handles input, output processing and object passing between different modules. The user provides the PDB id of the protein to be visualized. Protein structure information is then loaded by the **PDB Loader Module** by reading either the local structure file or the automatically downloaded structure file from PDB web site[8].

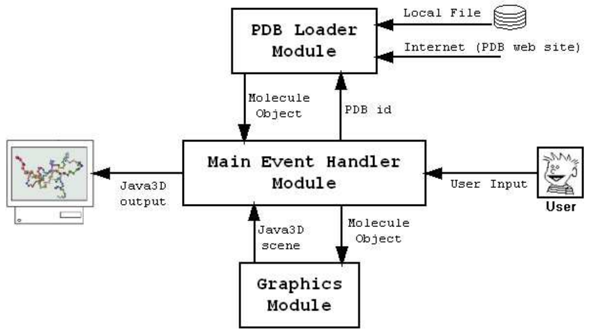


Figure 1: FPV System Overview.

The **PDB Loader Module** creates a *Molecule* object, which is composed of several Java classes, and then passes this object to the main event handler module. This *Molecule* object is then passed to the **Graphics Module**, which is responsible for generating the 3D model of the corresponding protein molecule. The 3D model is represented by a Java3D scene graph, which is returned back the the main module. The scene is rendered by the Java3D engine.

The most critical component of the visualization system is the Graphics module. We explain the techniques we've used to optimize this module for best rendering performance in the implementation section.

4. VISUALIZATION

In this section, we briefly discuss how we create molecular scenes from the protein data. We also present two accompanying textual views, which are helpful in browsing the amino acid sequence and viewing the hierarchical organization of the protein data. The techniques for expediting rendering based on Java 3D will be discussed in Section 5.

4.1 Data

PDB files are obtained from the Protein Data Bank (PDB) [1], which is an archive of experimentally determined 3D structures of biological macromolecules. PDB files contain

[7]http://mbt.sdsc.edu/
[8]http://www.rcsb.org/pdb/

3D coordinates of each atom of the protein molecule. We use these 3D coordinates and atom types to calculate the bonding information and to estimate the secondary structure. This information is needed for some of the 3D molecular representations described below.

4.2 3D Representations

Each representation of a protein molecule highlights a different aspect of the structure. They have advantages and disadvantages compared to each other. For example, the space-fill model can be helpful in understanding the volume a protein molecule occupies, but it lacks information about how amino acids are connected to each other, i.e. how the chain is formed. We describe below different 3D models provided by our visualization system, and explain their use and the way they are built.

Bonds Model:
Bonds model is created as a wire-frame model representing the bonding information in the protein molecule. Figure 2 shows a *bonds* representation of the molecule Oxygen Binding (PDB ID: 2MHR).

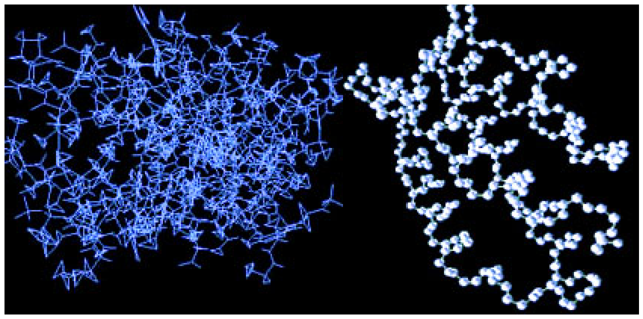


Figure 2: Bonds (left) and Backbone (right) Models

Backbone Model:
The backbone model is created by using the *alpha carbon, carbon,* and *nitrogen* atoms in the molecule. The position of the atoms are used to transform the spheres that represent them. The backbone bonds within each amino acid and the peptide bonds (between amino acids) are also shown in the model. This model is useful for understanding the protein molecule as a chain, and realizing amino acids' positions in this chain.

Figure 2 shows the backbone model of the molecule Oxygen Binding (PDB ID: 2MHR). When we interact with the 3D model of the backbone of a molecule, we can easily recognize how the amino acid sequence is formed in four parallel helices.

Balls and Sticks Model:
The balls-sticks model shows all of the existing bonds in the molecule as sticks and all the atoms as equal sized spheres.

Space-fill (van der Waals) Model:
The space-fill model is useful in visualizing the volume a protein molecule occupies. It gives an overall view of the molecule and thus provides a good view of the tertiary structure. In this model each atom is modeled using its van der Waals radius, so that the viewer gets an idea of the relative sizes of the atoms making up the protein molecule. The atoms are represented by concrete spheres centered at the corresponding atomic coordinates read from the PDB file.

Ribbon Model:
The ribbon model is used to display the secondary structures in the protein molecule. The secondary structure is predicted from the atomic coordinates in the PDB file, by using the algorithm developed by Kabsch and Sander [7]. The ribbon model is created using hermite curves. Our implementation is based on the program called MolScript [9]. Figure 4 shows the ribbon model of the same molecule 2MHR. Here, different colors for different secondary structures are used.

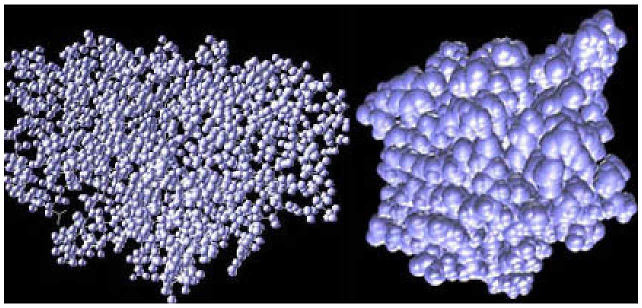


Figure 3: Balls and Sticks (left) and Spacefill (right) Models.

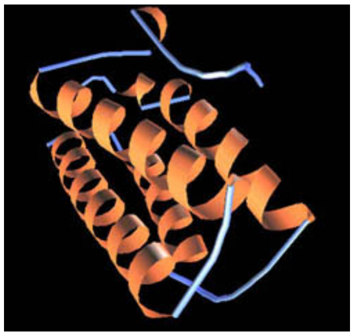


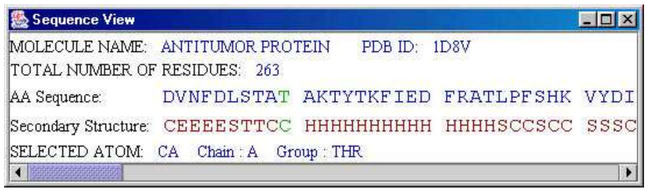
Figure 4: Ribbon Model.

4.3 Textual Information Windows

Having a textual representation of the protein molecule has many benefits. First of all it shows the linearity of the protein structure. The name of amino acids forming the chain is provided in a sequence view. Furthermore, the underlying hierarchy of the molecule can be captured when a tree view is used. We describe below the two accompanying information windows provided by our visualization component.

Molecule Information Window:
The molecule information window contains information about molecule's name, number of amino acids it contains, the amino acid chain, the secondary structure information, and information about currently selected sub-structure. The amino acid chain is displayed using one-letter representations of the amino acids. The molecule name info is read from the PDB file. Although it is possible to gather secondary structure information also from the PDB file, because of the fact that most of the PDB files available do not contain that information, the secondary structure information is calculated by using the prediction algorithm developed by Kabsch and Sander [7]. The information about the secondary structure is also displayed using one letter codes aligned with the amino acid codes (H:helix, B:residue in isolated beta bridge, E:extended beta strand, G:310 helix, I:pi helix, T:hydrogen bonded turn, S:bend).

When the user makes selections on the molecule during the interaction with a 3D model, the corresponding part of the amino acid chain in the information window is highlighted. If the selection is in the level of atoms, the selected atom information is also displayed in the *information window.*



MOLECULE NAME: ANTITUMOR PROTEIN PDB ID: 1D8V
TOTAL NUMBER OF RESIDUES: 263
AA Sequence: DVNFDLSTAT AKTYTKFIED FRATLPFSHK VYDI
Secondary Structure: CEEEESTTCC HHHHHHHHHH HHHHSCCSCC SSSC
SELECTED ATOM: CA Chain : A Group : THR

Figure 5: Molecule Information Window.

Figure 5 shows the molecule information window during interaction with the Antitumor Protein (PDB ID: 1D8V) protein. The currently selected amino acid is *Threonine*, whose one letter code is *T*, and it is the 10th amino acid in the first (and only) chain of the protein molecule. We see in the secondary structure information that this amino acid is part of a *coil*, and currently selected atom is *alpha carbon.*

Tree View Window:
Although a protein is a linear structure of amino acids, there's a hierarchy in the primary structure of protein molecules. A protein *molecule* is composed of one or more *chains* of amino acids. A chain may contain several *amino acids*, probably in the order of hundreds. Each amino acid has an eight atom *body* and a *side chain*, i.e. residue, which may be made up of 1 to 18 atoms. We provide a *tree view* window that visualizes this hierarchical structure of a protein molecule.

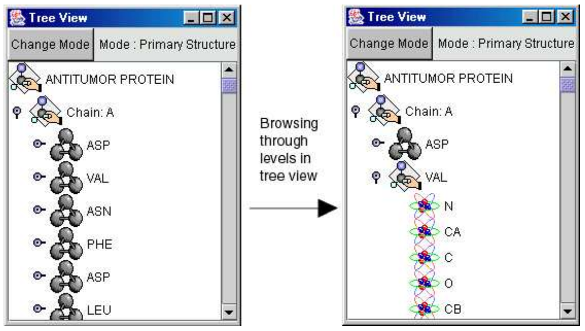


Figure 6: Tree View Window.

Figure 6 shows the tree view window while browsing through the hierarchy. In this snapshot the molecule has a very simple hierarchy, since it contains only one chain. But it is still useful to understand how the protein molecule is built. We provided a two-way interaction between the tree view and the 3D view. The user can interact with the tree by selecting its nodes. The corresponding sub-structure is highlighted in the 3D model. When the interaction is with the 3D model, and if a selection is made on it, the corresponding tree node is highlighted accordingly.

5. IMPLEMENTATION OF VISUALIZATION

In this section we describe the techniques we have used to speed up real time interaction and to be able to load very large molecules. The key issue here is the way the scene graph structure is created from a protein structure file (PDB). A scene graph consists of Java 3D objects, called nodes, arranged in a tree structure. The factors that affect efficiency are the *number* and *types* of nodes in the scene graph structure.

All the node objects in a scene graph are derived from the Node class. Java 3D refines the Node object class into two subclasses: Group and Leaf node objects. Group node objects group together one or more child nodes. A group node can point to zero or more children but can have only one parent. Leaf node objects contain the actual definitions of shapes (geometry), lights, sounds, and so forth. A leaf node has no children and only one parent.

Our method comprises two components:
(i) Converting TransformGroup nodes to Group nodes by applying the transformation in the Geometry node level,
(ii) Combining shapes that have the same appearance into a single Shape3D node.
The first component helps increasing the real time interaction speed while the second component decreases the memory needed by the scene graph structure, thus allowing loading larger molecules.

We explain these two techniques by giving an example of creating a space-fill (van der Waals) model of a protein molecule. The space-fill model consists of spheres of different sizes transformed to the their correct atomic locations according to the 3D atomic coordinates read from the PDB file. The intuitive way to create a space-fill model is to use the Sphere objects provided by the Java 3D API to create spheres of desired size and add them to the TransformGroup objects to translate them to their correct position. Figure 7 shows a scene graph structure created by using this method. However, as the number of atoms in a molecule increases the number of TransformGroup nodes increases since each atom has a unique position in the molecule. This makes interaction with the scene very inefficient because at each frame all the TransformGroup nodes need be processed to get the new position of each atom. This process involves a 4x4 matrix multiplication for each TransformGroup object.

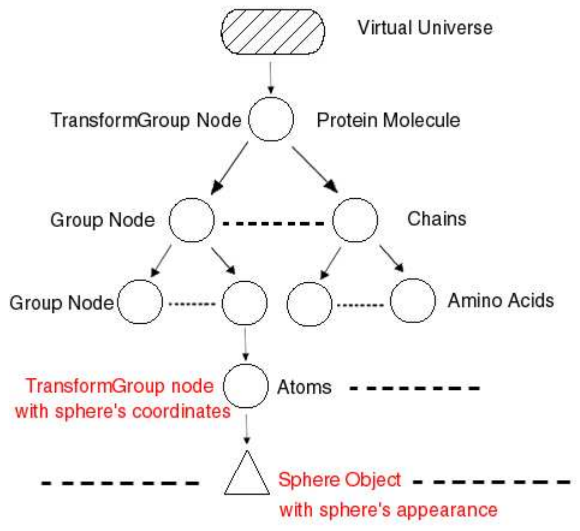


Figure 7: A fragment of an intuitive scene graph for VDW model.

To improve on the situation, one observation we made is that the protein molecule is static during interaction, i.e. individual atoms do not move freely. So, according to the interaction's nature **one** TransformGroup node is enough for representing protein molecule's rigid structure's position. However, by using Java 3D's Sphere nodes it is not possible to implement this solution, because the Sphere class does not allow creation of a sphere at an arbitrary position. Thus the only way to create a sphere at a specific position is to put a TransformGroup node above it.

But, there's a way to get around this restriction of Java3D. We have implemented our own Sphere class, which allows a sphere to be built at a specific location. By doing this, what we actually did was to propagate the transformation in the TransformGroup node to the geometry level, by creating geometry at a given static location. This puts a little overhead to the scene building process, i.e. by applying transformations during scene graph creation, but as we show in the next section this overhead is acceptable. The more important thing is that we have reduced the number of TransformGroup nodes in our scene graph to *one* (the one for the whole molecule) by getting rid of *all* the TransformGroup nodes representing individual atoms. As will be shown later, this modification improves the interactive rendering speed significantly. Figure 8 shows the scene graph after this improvement.

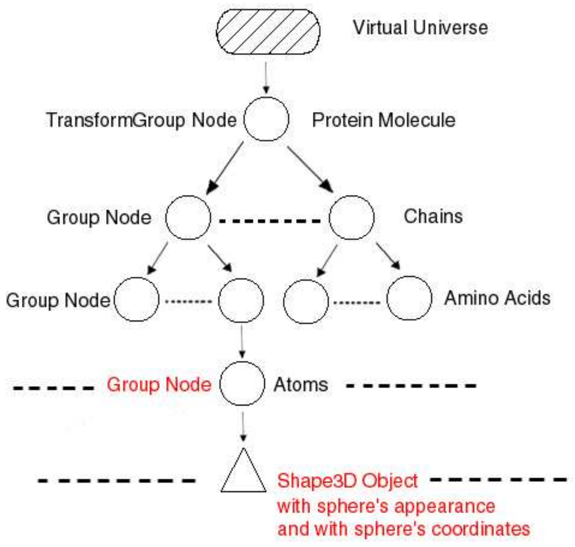


Figure 8: The scene graph after applying first technique.

As seen in Figure 8 each sphere is represented by a Shape3D object which encloses its *geometry* and *appearance*. The scene graph contains as many Shape3D objects as the number of atoms in the protein molecule. As the molecule size increases these increasing number of Shape3D nodes may cause memory problems. One way to overcome this is to put spheres with the same appearance under a single Shape3D node by combining their geometry information into a single geometry array. The number of Shape3D objects we need is equal to the number of different sphere appearances. For example, if we want to color each atom in a different color, we only need 6 Shape3D nodes, since the protein molecules consist of 6 different atoms (Carbon, Oxygen, Nitrogen, Hydrogen, Sulphur, and Phosphate). This way we can get rid of many Shape3D objects and free up memory space. This technique enables us load very large molecules, which contain as many as 4000 amino acids. Figure 9 shows the scene graph after application of this second technique.

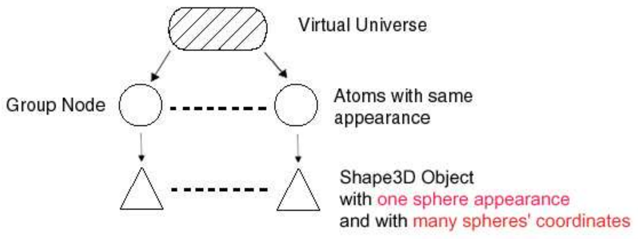

Virtual Universe
Group Node
Atoms with same appearance
Shape3D Object
with one sphere appearance
and with many spheres' coordinates

Figure 9: The scenegraph after applying the second technique.

What we have provided with these techniques is actually a hybrid method combining both retained mode and immediate mode graphics. The immediate mode is simulated by breaking the scene graph hierarchy and collapsing some nodes into a single node to save up memory space and to increase real-time interaction speed. In the next section we demonstrate the effectiveness of our methods by providing some test results.

6. PERFORMANCE TESTS AND RESULTS

We have compared our system (FPV) to two other molecular visualization tools according to their *scene building* and *real time interaction speed* performances. These tools chosen for the tests (JMV 0.85 and JMVS2) are among the few available molecular visualization tools based on Java 3D. We have chosen JMV and JMVS2 because they are closer to our system in terms of purpose and functionality.

The tests were performed using JAVA2 JRE 1.4.1 and JAVA 3D 1.2.1_04 (DirectX version) on a Microsoft Windows XP machine with Intel Pentium 4 Processor at 2.0GHz and 512MB of RAM. We have dedicated 256MB of this as the maximum size of memory allocation pool for Java Virtual Machine. The graphics accelerator card used for the tests was 64MB DDR NVIDIA GeForce4 MX Graphics Card. The data set comprised 22 protein structures in PDB format ranging in size from 29 amino acids (1BH0) to 8337 amino acids (1AON). Table 1 shows the protein molecules and their sizes respectively (both in terms of *number of amino acids* and *number of atoms*).

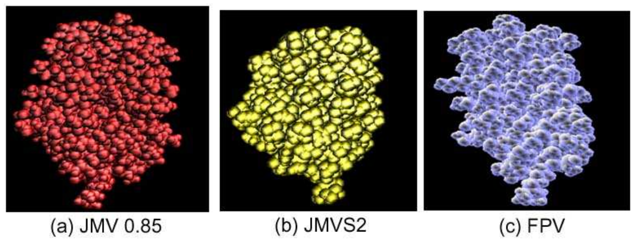


Figure 10: VDW (Spacefill) Model for protein molecule 2MHR

We have chosen three different types of visual representations to perform the tests: van der Waals (VDW or spacefill) model, bonds (wireframe) model, and ribbon model. The ribbon model type did not exist in JMVS2 so that part of test was performed on JMV and our system only. The *tube* model type of JMV, which was very close to our ribbon representation, was compared as the ribbon model. We tried to make the visual representations as close as possible by

Protein (PDB ID)	*Size (# of residues)*	*Size (# of atoms)*	*Protein (PDB ID)*	*Size (# of residues)*	*Size (# of atoms)*
1BH0	29	242	1A05	716	5386
1PTQ	50	402	1DUV	999	7648
1DF4	68	463	1A0S	1239	9606
1GCM	102	814	13PK	1660	12508
1K52	144	1122	1F8R	1992	15291
2AID	198	1516	1B25	2476	19144
1D9C	242	1993	1L1F	3030	23244
1A4F	287	2250	1DP0	4092	32500
3MDS	406	3282	1H6D	5196	35555
1SYN	528	4300	1GYT	6036	46152
1D3A	606	4602	1AON	8337	58688

Table 1: Sizes of Test Proteins

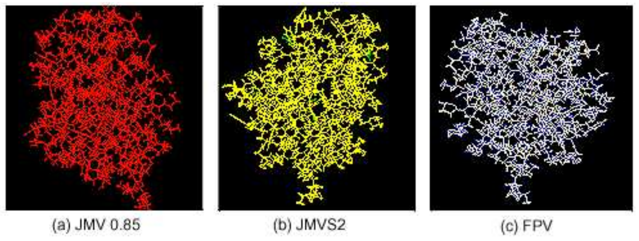


Figure 11: Bonds Model for protein molecule 2MHR

adjusting the display options of the compared systems, e.g. number of sphere divisions. Figures 10, 11, and 12 shows, for each system, the visual representations used for the tests.

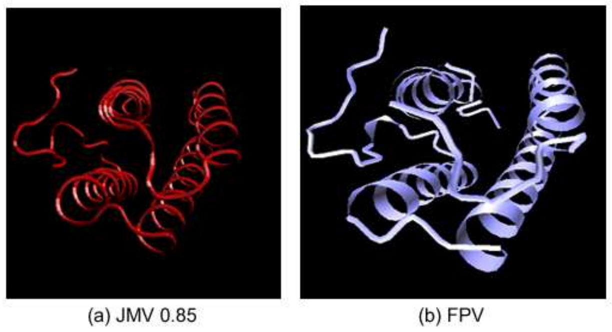


Figure 12: Ribbon Model for protein molecule 2MHR

The calculation of the timings and rendering speed measurements was possible because source codes of both tools were available. We've measured the scene building times and real-time interaction speed. The scene building times become important, when the user wants to switch between models during interaction. The latency between switching from one representation to another can be intolerable if it is more than a few seconds. One may consider building all the available models during start-up to decrease model switching time during interaction, but this requires much more memory compared to the memory required by a single model type. Therefore, the size of the largest loadable protein molecule decreases drastically. All the programs that we've compared use the suggested approach, which is building a specific model type on demand. That's why we've taken scene building times into consideration. The importance of the real-time interaction speed is obvious. It is one of the main quality metrics of interactive visualization tools.

Figures 13, 15, and 16 show results of the rendering speed tests. To measure rendering speed we've used a RotationInterpolator object to have the molecules rotate around y-axis

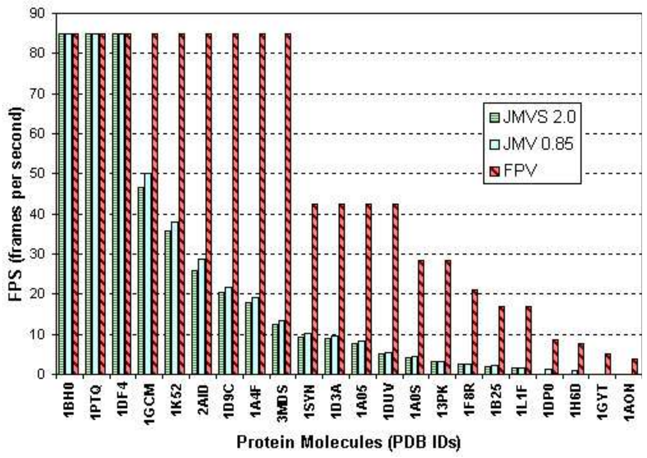


Figure 13: Rendering Speed for the VDW Model

at a constant speed. We then calculated rendering speed by looking at the difference in frame numbers at certain time intervals. Values of 25 and more are ideal in the graphs showing the results of rendering speed tests, because 25fps is the highest frequency the human eye can detect.

In the VDW Model rendering speed test, our system had better performance compared to the other programs, while they performed close to each other. That's because the new Sphere classes that we have implemented to get rid of the TransformGroup nodes and encapsulate many spheres under a single Shape3D node. Thus our system had up to eight times better rendering speed performance (at protein 1DUV) compared to the other programs. Furthermore, our system was able to load the largest molecule, which has 58688 atoms, while JMV and JMVS2 could at most load proteins that have 35555 and 23244 atoms respectively. Figure 14 shows the largest molecule of the test set displayed by our program, FPV. Furthermore, our program could render this molecule at 4 frames per second.

In the Bonds Model test, the performances of our system and JMV were close to each other, while JMVS2 had acceptable speeds for only small molecules. JMV performed better than FPV for large molecules, but it should be noted that even for those large molecules FPV could establish a rendering speed over 26 frames per second. So the difference between JMV and our program was not noticeable practically. Our scene graph structure for the bonds model consists of a single line segments array for all of the bonds of the protein molecule, thus resulting in a very simple scene graph structure. The JMV program uses a similar approach thus has similar performance results. However, the scene graph used by JMVS2 tries to put every bond in a separate

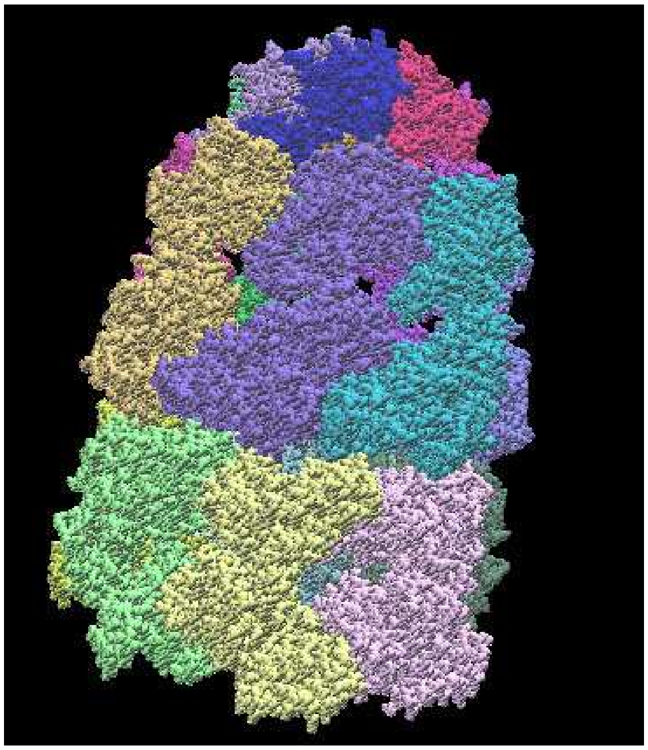


Figure 14: Spacefill model for protein 1AON

Shape3D object, thus resulting in a very poor performance.

The rendering speed comparison of the ribbon model was performed only with the JMV program. Our program performed better than JMV as seen in Figure 16. In ribbon model test, we could again load the largest molecule in our data set (1AON), which was 8337 amino acids long, while the largest molecule loaded by JMV had 3030 amino acids (1L1F). Furthermore, FPV achieved up to 20 times better rendering speed performance (at protein 1B25) compared to JMV.The main reason for this was again our method of combining related primitives under a single scene graph node.

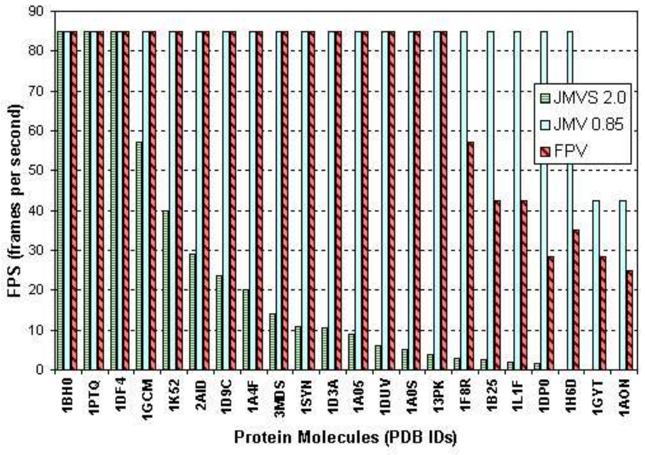


Figure 15: Rendering Speed for the Bonds Model

Figures 17, 18, and 19 show results of the Java 3D scene graph building tests. By presenting these results we show that the scene graph manipulation techniques we've proposed do not cause any overhead on molecular scene building.

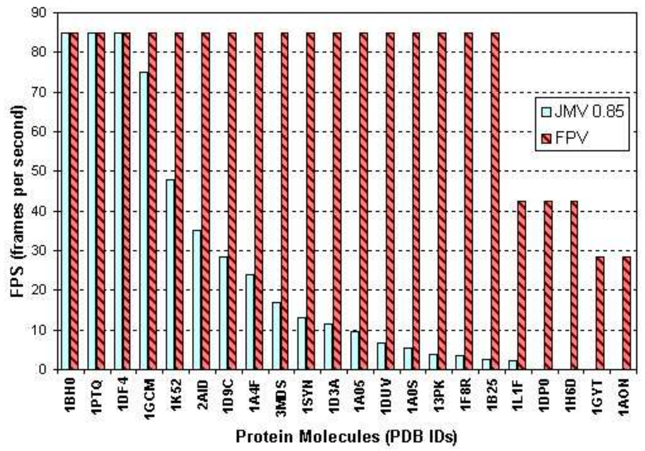


Figure 16: Rendering Speed for the Ribbon Model

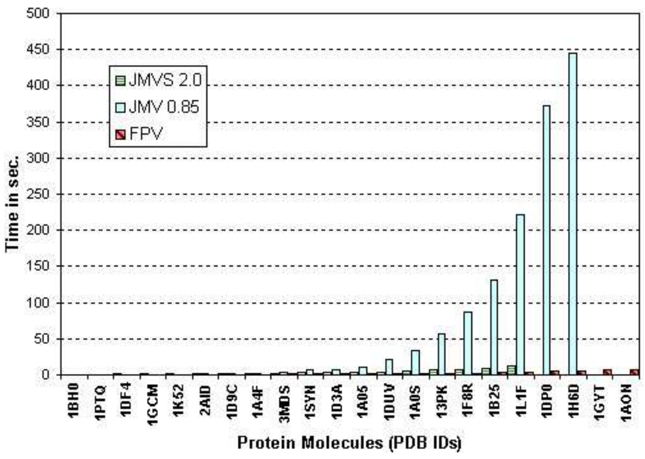


Figure 17: Scene Building Times for the VDW Model

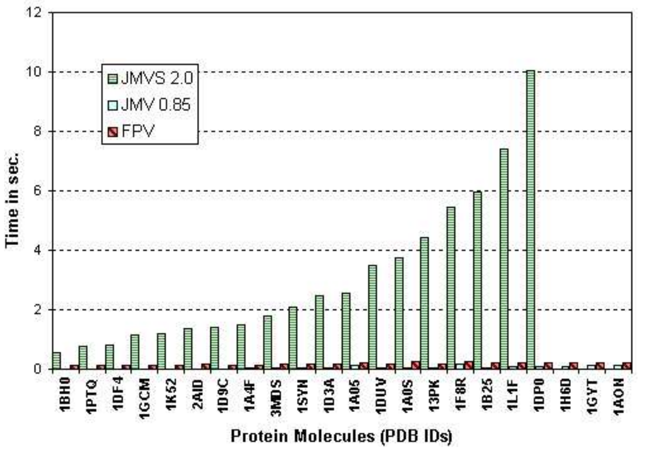


Figure 18: Scene Building Times for the Bonds Model

For the VDW model JMV program performed worst among all three programs compared. The time required to build the molecular scene grows very quickly with the size of the protein. For the large molecules the time required for JMV to build a molecular scene can grow up to hundreds of seconds, which is not acceptable. JMVS2 and our program had reasonable scene building times in the scene graph building tests for VDW model type.

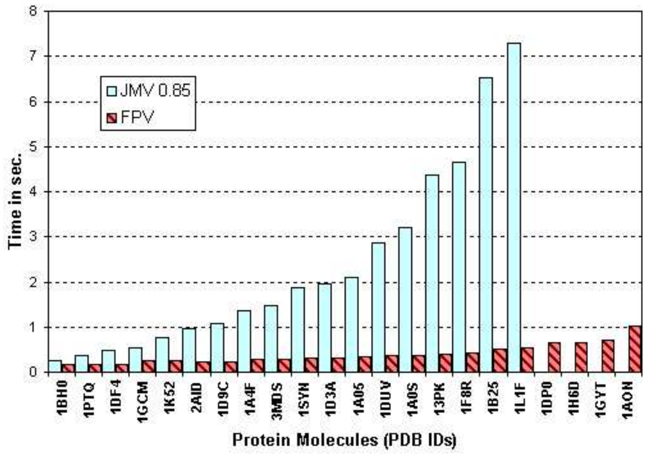


Figure 19: Scene Building Times for the Ribbon Model

For the bonds model, our program and JMV had similar results and the scene building time was negligible (much less than 1 sec). This time JMVS2 performed poorly compared to our system and JMV. In these results, it is seen that processing some primitives together under a single Shape3D note has benefits rather than an overhead.

Our program outperformed JMV on the scene graph building test for the ribbon model. As mentioned before the ribbon model tests were not performed for JMVS2 because it didn't have a ribbon type representation. It can be seen from Figure 19 that scene graph building times for FPV were less than 1 second for all the test proteins. This means FPV has a very low latency when switching between model types during interaction with the molecule. The results for this test again shows that processing related primitives together under a single group node has benefits instead of an overhead during scene building.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a high-performance protein visualization application called FPV. We've proposed implementation techniques to increase the usability of our application by improving the real-time rendering speed and increasing the range of protein data that can be examined. These improvements are accomplished by modifying the scene graph structure used by the Java 3D API. We have showed the effectiveness of our methods by comparing our system to two other molecular visualization tools based on Java 3D.

In order to make our tool more attractive to researchers, we are looking for ways to increase the functionality of our system. One way incorporating new functionality is providing new 3D representation types for protein molecules, such as electron density map and molecular surface representation. Since we've designed the visualization system as a toolkit, it is easy to add new functionalities depending on an application's needs, such as adding superpositioning functionality to the Graphics Module for comparison of protein structures. The design of our system also allows users to decouple and use components of the system, such as PDB Loader Module.

8. ACKNOWLEDGEMENTS

This work is supported in part by NSF grants IIS-9817432, IIS-9908441, and IIS-0101134.

9. REFERENCES

[1] H.M. Berman, J. Westbrook, Z. Feng, G. Gilliland, T.N. Bhat, H. Weissig, I.N. Shindyalov, and P.E. Bourne, "The Protein Data Bank", *Nucleic Acids Research*, 28 pp. 235-242, 2000.

[2] P.E. Bourne, M. Gribskov, G. Johnson, J. Moreland, and H. Weissig, "A Prototype Molecular Interactive Collaborative Environment (MICE)", *Pacific Symposium on Biocomputing*, 1998, pp. 118-129.

[3] S.K. Burley, S.C. Almo, J.B. Bonanno, M. Capel, M.R. Chance, T. Gaasterland, D.W. Lin, A. Sali, F.W. Studier, and S. Swaminathan, "Structural genomics: beyond the human genome project", *Nature Genetics*, vol. 23, pp .151:157, 1999.

[4] J.D. Foley, A. van Dam, S.K. Feiner, and J.F. Hughes, "Computer Graphics Principles and Practice", Second Edition Addison-Wesley, Reading, 1990.

[5] N. Guex and M. C. Peitsh, "SWISS-MODEL and Swiss-PdbViewer: an environment for comparative modeling." *Electrophoresis,* pages 2714-2723, 1997.

[6] W. F. Humphrey, A. Dalke, and K. Schulten, "VMD - Visual Molecular Dynamics", *Journal of Molecular Graphics*, 14:33-38, 1996.

[7] W. Kabsch and C. Sander, "Dictionary of Protein Secondary Structure: Pattern Recognition of Hydrogen-Bonded and Geometrical Features," *Biopolymers,* 22:2577, 1983.

[8] R. Koradi, M. Billeter, and K. Wüthrich, "MOLMOL: a program for display and analysis of macromolecular structures", *J Mol Graphics*, 14, 51-55, 1996.

[9] P. J. Kraulis, "MOLSCRIPT: A Program to Produce Both Detailed and Schematic Plots of Protein Structures", *Journal of Applied Crystallography*, vol. 24, pp. 946-950, 1991.

[10] Java 3D API Specification, Version 1.1.2, June 1999, http://java.sun.com/products/java-media/3D/ /forDevelopers/j3dguide/j3dTOC.doc.html

[11] S. Meloan, "Exploring The New Frontier: Java™ Technology Powers the "Post-Genomic" Era", *Feature Stories, java.sun.com*, September 28, 2001.

[12] R. A. Sayle and E. J. Milner-White, "RASMOL: biomolecular graphics for all", *Trends in Biochemical Sciences*, 20(9):374, Sep 1995.

[13] C. Shirky, "Seven Ways of Looking at a Protein", *FEED Magazine, After Darwin Column,* 23 Oct, 2000.

[14] D. Walther, "WebMol - a Java based PDB viewer", *Trends Biochem Sci*, 22: 274-275, 1997, http://www.embl-heidelberg.de/cgi/viewer.pl.