Improved Quadric Surface Impostors for Large Bio-Molecular Visualization

Pranav D Bagur
Department of Information Technology
National Institute of Technology Karnataka, Surathkal
prnvbagur@gmail.com

Nithin Shivashankar*
Computer Science and Automation
Indian Institute of Science, Bangalore
nithin@csa.iisc.ernet.in

Vijay Natarajan
Computer Science and Automation
Supercomputer Education and Research Center
Indian Institute of Science, Bangalore
vijayn@csa.iisc.ernet.in

ABSTRACT

The shape of biomolecules, such as proteins, may be represented using different representations like space-fill, ball-stick, backbone, or secondary structures. The secondary structure of proteins, comprising of sheet-like, helix-like and loops structures, represent a higher level abstraction of its structure. With ever increasing sizes of protein structure data produced by high resolution x-ray crystallography and cryo-electron microscopy, biologists often rely on visualizations to better understand the overall structure of proteins. In this paper, we present a unified framework for accelerating the rendering of various representations of these structure using GPUs. The framework first produces "impostor primitives", which are simple linear element approximations of quadric objects, such as spheres, cylinders, and helices. Next, the rasterizations of the impostors are corrected to produce pixel-precise renderings of the quadric objects. We incorporate this framework into a bio-molecular visualization tool PROTEINVIS to demonstrate quantitative and qualitative performance gains over earlier approaches for rendering various representations of proteins.

Keywords

Bio-Molecular Visualization, GPU acceleration, Billboarding

1. INTRODUCTION

Biomolecules, such as proteins and nucleic acids (DNA and RNA), are involved in every aspect of cellular function. Often times, understanding their structure is key to understanding their function. In the past, crystallographers and biologists created detailed real-world models, called Corey-Pauling-Koltun models, using wooden or synthetic spheres to represent atoms and sticks to represent bonds [6, 10]. Today, these models of protein structures, referred to as space-filling and ball-stick models, have been adopted in computer graphics systems to create visual representations. Furthermore, biomolecules such as proteins, exhibit structural regularity by the formation of structures such as helices and sheets. This secondary structure is used to create abstracted representations which, at a coarser level, help in revealing the overall structure of the protein. The need for insight into biomolecular structure has led to the creation of a large number of free and commercial visualization applications. By the use of X-ray crystallography and cryo-electron microscopy, bio-molecular structural data is made available at an ever increasing rate through world-wide collaborative efforts such as the protein data bank (See *http://www.rcsb.com*) [3, 2]. With increase in size and detail of available data, it becomes crucial for visualization systems to produce high-quality visualizations at interactive frame-rates. Further, given the world-wide collaborative nature towards research in bio-molecular and cellular systems, the software should be deployable on commodity and off-the-shelf hardware for its wider adoption.

In the past decade, the demand for specialized hardware for computer games has catalysed the rapid development of Graphic Processing Units (GPUs). Traditional graphics systems were designed as multistage pipelines. GPUs, consisting of several processing cores, are employed to accelerate the graphics processing pipeline. Modern GPUs expose programmability of the vertex generation, geometry generation, and fragment coloring stages of the pipeline. With programmable GPUs, it is now possible to create high-quality visualizations at interactive frame-rates using commodity hardware. Many efforts in the bio-molecular visualization field also aim to leverage this technology. This paper presents a unified framework to accelerate the rendering of protein molecules.

1.1 Related Work

The use of GPUs to accelerate and create high quality renderings of biomolecules has been active for a few years. Traditional molecular visualization applications such as VMD ([8]) and PyMol ([14]), being rich in terms of functionality and features, now leverage GPUs to produce good quality renderings for the common representations. Other applications such as TexMol [1] and QuteMol [16] focus on di-

*Corresponding author

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICVGIP '12, December 16-19, 2012, Mumbai, India
Copyright.2012.ACM.978-1-4503-1660-6/12/12...$15.00.

rectly leveraging GPUs to generate high quality visualizations. Illuminated ellipsoids, by Gumhold [7], find application in many biomolecular visualization applications for rendering space-fill representations. Tarini et al. [16] describe a technique to enhance rendering of space fill models using ambient occlusion. Lampe et al. [13] describe a two level rendering hierarchy to exploit the structural duplication in proteins for efficient rendering using GPUs. Krone et al. [11] describe techniques to accelerate the rendering of the secondary structure of proteins. They also describe techniques for generating high-quality rendering of molecular surfaces [12]. Sigg et al. [15] describe accurate rendering of sphere ellipsoid and cylinder primitives, whose use is demonstrated for bio-molecular visualizations. In our work, we extend the methodology for fast accurate rendering of spherical elements, developed by Gumhold [7], to both cylindrical and helical quadric surface elements. We also generalize the cylindrical elements described by Sigg et al. [15] to produce smooth tube-like rendering of curves. In comparison to the methods developed by Gumhold [7] and Sigg et al. [15], our representations belong to a unified framework which extends to helical elements also, while achieving superior frame rates. In comparision to methods by Krone et al. [11], our representation of the α-helix secondary structure requires a single primitive for each helix, thereby requiring far lesser memory and overhead, which results in two to four times increase in frame rates for rendering the secondary structures of proteins.

1.2 Contributions

We present a framework using GPUs for the creation of high quality visualizations of the primary and secondary structure of proteins. Our framework adopts a two-step process for creating pixel precise renderings of quadric objects namely spheres, cylinders, and helices. The first stage produces an approximation of the object using simple linear elements such as triangles. The second stage corrects the rasterizations to produce precise renderings of the objects on every pixel. We develop the framework towards the rendering of the following quadric elements:

- **Spherical Elements:** We simplify the ellipsoid model from by Gumhold [7] to render spherical objects. Further spherical elements are handled within a unified framework that handles other quadric elements also.
- **Cylindrical Elements** We describe a procedure to produce pixel precise renderings of cylindrical objects. Since the primary application of this element is the rendering of tubular surfaces, the cylinders are designed to be cut by arbitrarily oriented planes. In addition, the normals on the tubular surface are computed to ensure a smooth appearance.
- **Helical Elements** We present a novel extension of the cylindrical element to render helical elements. This representation results in approximately two times on average and upto four times increase in the frame rates for rendering helical elements.

We implement the above framework in a tool called PROTEINVIS to represent primary and secondary protein structures. For the primary structure representations, the spherical element is applied to the space-fill representation, the spherical and cylindrical element is applied to the ball-stick representation. For secondary structure representations, the helical element is used to render α-helices and the cylindrical element is used to represent the free loops formed by the protein backbone.

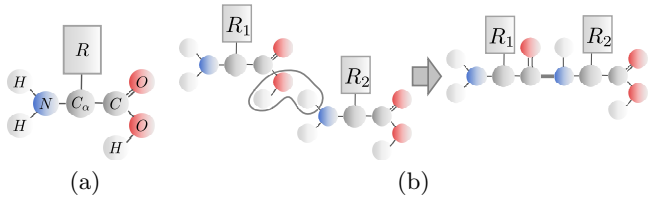


Figure 1: (a) Amino acids consist of an amino group (NH_2), and a carboxylate group (COOH), attaching to a carbon atom (C_α). Amino acids are differentiated by the residue (R) attaching to C_α. (b) A peptide-bond is formed between two amino acids, by the carbon of the carboxylate group in the first and the nitrogen of the amino group in the second.

Thus the framework enables a unified mechanism for representing various protein structures. We evaluate our implementation, in terms of performance and quality, against earlier related methods. We achieve upto two times increase in framerates compared to popular protein viewers while maintaining same quality of rendering. The improved performance will enable biologists to visualize larger biomolecules on desktop PCs or laptops without compromising the quality of the interaction or rendering.

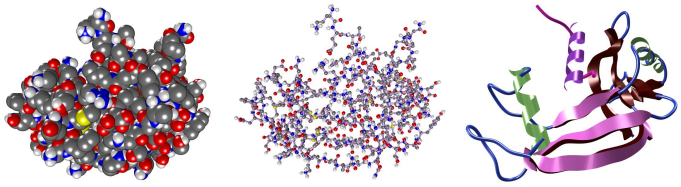
2. BACKGROUND

Biomolecules are complex assemblies of atoms that perform key biological functions at the cellular level such as storage of genetic information (nucleic acids such as DNA and RNA) and facilitating biological functions. It has long been known that the structure and spatial conformation of these molecules are key to their function.

Proteins.

Proteins form a sub-class of biomolecules that comprise of a linear sequence of amino acids linked by peptide bonds to form poly-peptide chains. There are twenty naturally occurring amino acids which bond together to yield a vast number of distinct proteins. Structurally, amino acids consist of an amino group (NH_2) linked to a carbon atom (C_α), which is in turn linked to the carbon atom of a carboxylate group (COOH) (see Figure 1a). Peptide bonds across the amino and the carboxyl groups of consecutive amino acids, which form by the loss of hydrogen and hydroxyl (OH) atoms (see Figure 1b). Amino acids are typically differentiated by a *residue* which attaches to the C_α atom. The sequence of amino acids is referred to as the *primary structure* of the protein and the sequence of nitrogen, C_α, carbon, and oxygen is referred to as the *backbone* of a protein.

The presence of bonds across amino acids in the polypeptide chains, such as weak hydrogen bonds between residues and disulphide bonds, facilitate spatial orientations of the backbone which typically conform to minimize the total energy of the system. This leads to formation of some regular structures, such as α-helices and β-sheets, and irregular coils of the backbone. These structures are referred to as the sec-



(a) (b) (c)

Figure 2: Various representations of the ribonuclease molecule 1D5H. (a) Space-fill. (b) Ball-stick. (c) Backbone spline, α-helices and β-Sheets.

ondary structure of the protein.

Visual Representations.

The *space-filling* model of bio-molecules a common representation of primary protein structure. The model comprises of three-dimensional volumetric spheres representing atoms with the radius corresponding to the *van der Waals* radius (see Figure 2a). These are essentially virtual representations of the classic CPK models [6, 10]. The atomic position data is available either experimentally or by simulation. Another common representation of the primary structure is the *ball-stick* model, where constant sized spheres are used for atoms and lines or cylinders are used to represent bonds between atoms (see Figure 2b). In the presence of large number of atoms, it is often necessary to visualize only the backbone of the protein.

Visualizations of secondary structure of proteins, otherwise known as *cartoon* renderings, aid in providing a higher level of abstraction. Of these, spline interpolations of the backbone atoms, α-helices and β-sheets are among the most common (see Figure 2c). The backbone spline representation is formed by interpolating a spline curve, such as B-Spline or Catmull-Rom [5] splines, through the backbone atom positions. The α-helix representations are formed by extruding the backbone spline segments corresponding to the helix along the axis of the helix. In proteins, β-sheets form because of weak hydrogen bonds across amino acids in different parts of the backbone. These sheets are often represented by extruding the backbone spline segments corresponding to the sheets in the direction of these bonds.

Programmable Graphics Pipeline.

Traditional OpenGL graphics systems [17] are designed as a multi-stage pipeline, used to transform primitives representing geometry in 3D space to raster based images displayed on the screen. The pipeline (see Figure 3) accepts linear elements, such as lines and triangles, as input. In the first stage, OpenGL commands are evaluated to fetch the parameters of vertices of input elements like position, normal, color etc. In the next stage, the per-vertex information is transformed from *model coordinate system* to a *world coordinate system* via the *model-view transformation*. Also, the vertex positions are projected along *view-rays* to a *view-plane* via the *projection transformation*. The projection transformation can be *orthographic*, *i.e*, view-rays are orthogonal to the view-plane, or *perspective*, *i.e*, view-rays originate from an *eye* behind the view-plane. In the next stage, the projected vertices are assembled into their prim-

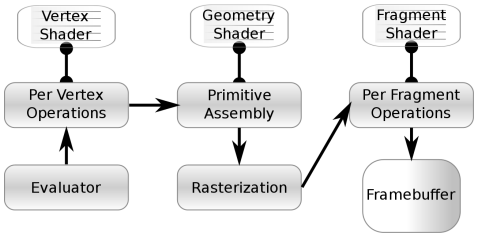


Figure 3: A typical OpenGL programmable graphics pipeline showing three programmable stages.

itives. Next, portions of the primitives which lie within the *viewport*, the rectangular portion of the view-plane corresponding to the desired output image, are rasterized into pixel fragments. The next stage assigns a color and depth along the view-ray for each fragment. The depth information is used by the *z-buffer* algorithm so that the intersections of 3D objects are rendered correctly *i.e.* when fragments of two or more elements are rendered on to the same image pixel, the color of the one that is closest to the eye is used for the image.

Modern GPUs allow programmablity of the per-vertex operations stage, primitive assembly stage, and the per-fragment operations stage. The programs for these stages are called vertex-shaders, geometry-shaders and fragment-shaders, respectively. The vertex shader modifies per-vertex information such as position, vertex color, vertex normal etc. The per-vertex output from the vertex shader for each primitive is fed to the geometry shader, which outputs zero or more primitives. The fragment shader is executed for each output fragment of primitives from the geometry shader, to update the fragment's color and depth. The pipeline is made more flexible by allowing shaders to communicate user-defined scalar and vector valued parameters between them. The geometry-shader specifies, or *binds*, values at the vertices of primitives which are made available to fragment-shader by linear interpolation along the vertex positions.

3. IMPOSTOR PRIMITIVES

We now proceed to discuss our framework for quadric elements using the programmable graphics pipeline. The procedure to render these elements proceeds in two stages. The first stage is carried out on the geometry shader and the second stage is carried out on the fragment shader. In the first stage, the quadric element is approximated by linear elements called "impostors", quads in the case of spheres and hexahedrons in the case of cylinders and helices. The geometry shaders generates impostors such that the raster images of the impostors would contain the raster images of the quadric elements. The geometry shader also binds the model coordinates of impostor vertices so that for each fragment an interpolated model coordinate is available. In the second stage, the impostors fragments are culled if they are not part of the quadric object's raster image. Otherwise, the correct depth and color of the fragment is updated. The above task is carried out by computing the intersection of the quadric object with the view-ray, from the eye position (in model coordinate) through the fragment's model coordinate. The eye position is obtained in model coordinates by multiplying the inverse of the model-view transformation to the eye position in world coordinates ($(0, 0, 0)$ in OpenGL). The depth

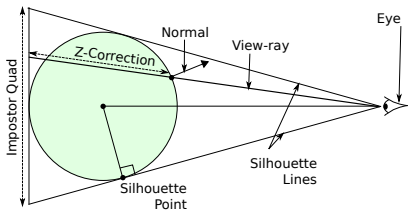


Figure 4: Schematic depicting the rendering of spheres in perspective projection

is updated as the distance from the eye to the intersection point (in world coordinates). In our implementations we consider only perspective projection, though the framework can be adapted for orthographic projection. Thus, in the second stage, we adopt the convention of referring to view-ray as the ray from the eye through the fragment's position in model coordinates. For high-quality images, the surface normal at the point of intersection is computed to be passed on for lighting algorithms using the Phong lighting [4] model.

In the following sub-sections we discuss in detail the procedure for rendering the spherical, cylindrical, and helical elements. In particular, we discuss the generation of impostor elements in the first stage, and the computation of view-ray-quadic-object intersections for each fragment to obtain accurate depth and surface normal in the second stage.

3.1 Spherical Impostor

The spherical impostor is a simpler version of the illuminated ellipsoids method, by Gumhold [7]. The input to the first stage is the position and radius of the sphere in model coordinates. The first stage begins by computing two mutually orthogonal unit vectors that are also orthogonal to the direction from the center to the eye (see Figure 4). Two *silhouette lines*, or lines from the eye tangential to the sphere, exist in the plane containing one of the vectors and the eye-center line. Each *silhouette point*, the point on the sphere and a silhouette line, is computed using the right triangle formed by the itself, the center, and the eye. The four silhouette points thus obtained define a quad. The vertices of the quad are projected along their respective silhouette lines away from the eye such that the quad is tangential to the sphere. This quad is generated by the geometry-shader. In the second stage, the solution to the quadratic equation representing sphere and view-ray intersection is computed. If there is no solution, the fragment is discarded. If there is only one solution, the fragment's true depth is calculated and if it is to be clipped (*i.e.* fragment is on the other side of the view plane) then it is discarded. In the case of two solutions, the solution closer to the eye along the view-ray is checked followed by the second solution. If both solutions are to be clipped, then the fragment is discarded. The technique by Tarini et al. [16], to avoid spheres popping in and out when they are positioned very close to the view plane, is used.

3.2 Cylindrical Impostor

The cylinder impostor is designed to provide tube-like renderings of poly-lines. Each segment of the poly-line is modeled as an infinite cylinder cut by two planes (see Figure 5a). The normal of the cutting plane at an end point is computed as the average of the direction of the two segments

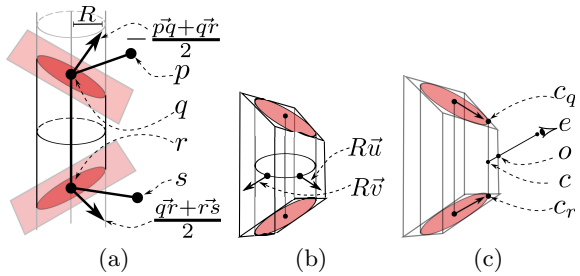


Figure 5: Schematic depicting the rendering of a cylindrical element, given four consecutive points on a poly-line p, q, r, s and a radius R. (a) The infinite cylinder with axis $\vec{qr}$ and radius R is cut by planes passing through q and r with normals $\vec{n_q} := -(\vec{pq}+\vec{qr})/2$ and $\vec{n_r} := (\vec{qr}+\vec{rs})/2$ respectively. $\vec{pq}$, $\vec{qr}$, and $\vec{rs}$ are unit vectors. (b) In the first stage, a hexahedron is generated in the geometry shader. Two faces of the hexahedron lie on the planes $(q, \vec{n_q})$ and $(r, \vec{n_q})$. The remaining four faces are defined by by lines in the direction of $\vec{qr}$ and passing through four points on the plane $(\frac{q+r}{2}, \vec{qr})$ defined by $\frac{q+r}{2} \pm R\vec{u} \pm R\vec{v}$. Here $\vec{u}$ and $\vec{v}$ are any pair of orthogonal unit vectors that are also orthogonal to $\vec{qr}$. (c) In the second stage, each rasterized fragment is corrected. First, the point c on the cylinder, and the line from eye position e passing through impostor's coordinate o, is computed. If no intersection exists or if c is outside either of the planes, $(q, \vec{n_q})$ and $(r, \vec{n_r})$, the fragment is discarded. The depth value of c is written to the depth buffer. The point c is projected onto the cut planes along $\vec{qr}$ to yield the points c_q and c_r. The normal at the point c is computed as $(1-w)q\vec{c}_q + wr\vec{c}_r$, where $w := \frac{\|c-c_q\|}{\|c_r-c_q\|}$.

incident upon the end point. In the first stage, the geometry shader computes the intersection of four infinite lines with the cutting planes (see Figure 5b). In the second stage, the fragment shader computes the intersection of the infinite cylinder and the view-ray (see Figure 5c). This resolves to a quadratic equation, which has at most two real solutions. If no real solutions exist, the fragment is discarded. Otherwise, only the intersection point closest to the eye is considered. The fragment is also discarded if this point is outside either plane. The correct depth from the eye position is written to the depth buffer. The surface normal at the intersection point is negative of the direction of its projection on to the cylinder's axis. This normal is discontinuous across cylindrical elements on either side of a cut plane. We produce smooth normal across cut planes by computing them as a convex combination of the projections of the normal to the cut planes along the cylindrical axis (see Figure 5c). This normal is then passed on to the Phong lighting calculations.

3.3 Helical Impostor

Points on a helix always lie on a cylinder. So, the first stage of the cylinder impostor described above is reused with a minor modification so that the cut planes are orthogonal to the cylindrical axis, *i.e.*, the cylinder is specified by the end points and radius only. Additionally, the helix requires the pitch along with a direction orthogonal to the axis, as input. The direction is used to determine a point on the cylinder through which the helix passes (see Figure 6). Also, the helix is extruded along the axis by a constant width.

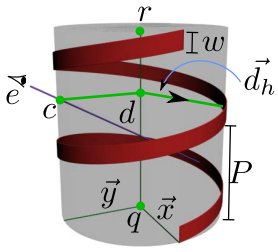


Figure 6: The helical element is created by modifying the second stage of the cylindrical impostor. P represents the pitch of the helix, w the width by which it is extruded. The directions $\vec{x}$ and $\vec{y}$ normal to $\vec{qr}$ and each other provide an orthonormal basis at q. The fragment shader computes the first intersection c of the view-ray and the cylinder. This point is projected to the point d on the axis. The direction along the corresponding helical point is computed by $\vec{d_h} := cos\theta\ \vec{x} + sin\theta\ \vec{y}$, where $\theta := 2\pi\frac{\|d-q\|}{P}$. The point c is retained only if $\langle\vec{d_h}, \vec{dc}\rangle < cos(2\pi\frac{w}{P})$, where $\vec{dc}$ is the unit direction along the line from d to c. If c is rejected, the second intersection of view ray and the cylinder is then evaluated similarly.

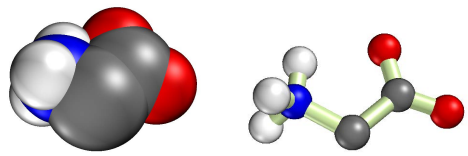
There exists a one-to-one correspondence between the points on the helix and its axis. The angle by which the helix turns, so that the corresponding points on the axis move by half the width, is computed. The cosine of this angle is made available to the fragment shader as a threshold parameter. The fragment shader begins by computing the first point on the cylinder along the view-ray (see Figure 6). This cylinder point c is projected to the axis. This axis point d yields a corresponding helix point. Unit vectors from the axis point to the cylinder point $\vec{dc}$ and helix point $\vec{d_h}$ are computed. If the inner product of the unit vectors is less than the threshold parameter, it is retained. The first unit vector $\vec{dc}$ is used as the normal of the helix surface. Since, the inner portions of the cylinder that lie on the helix may also be visible, the second intersection of the view-ray and the cylinder is also evaluated similarly, if the first intersection is discarded. The depth buffer is appropriately updated. If the second intersection point is not discarded, its normal is reversed. The appropriate normal is passed on to the lighting calculations.

4. BIO-MOLECULAR VISUALIZATION USING IMPOSTOR PRIMITIVES

This section discusses the application of the impostor primitives discussed in Section 3. The two stage framework is implemented in a protein visualization tool PROTEINVIS to accelerate rendering various representations of proteins. In particular, the space-fill and ball-stick primary structure representations and backbone loop cartoons and α-helix secondary structure representations leverage these primitives. In the following paragraphs, we discuss in detail, the application of the impostor primitives to the representations listed above.

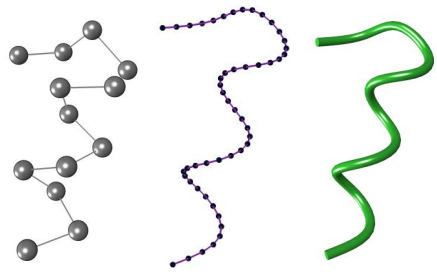
Space-fill and Ball-stick.

The spherical impostor is used for the visualization of the *space-fill* representation. It is used together with the cylinder impostor it is used for the visualization of the *ball-stick* model (see Figure 7). The application renders the spheres as points which are then processed by the spherical impostor framework. The sphere center and radius are available to the geometry shader as the vertex position and a user-defined vertex attribute, respectively. The application renders bonds as line segments between atoms which are then processed by the cylindrical impostor framework. The cylinder impostor model is simplified so that the cutting planes define an upright cylinder, *i.e.*, the plane normals are along the cylinder axis. Thus, only the end points of bonds are sent as input to the cylinder impostor framework. The bond radius is treated as a global constant.



(a) (b)

Figure 7: Glycine molecule rendered using impostor elements. (a) Space-fill model. (b) Ball-stick model.



(a) (b) (c)

Figure 8: Construction of the tube-like rendering of the backbone. (a) The C_α atoms of the backbone are used as input for the B-spline procedure. (b) The spline is sampled uniformly to generate a poly-line. (c) The cylinder impostor is used to generate tube-like renderings of each section of the poly-line.

Backbone loop.

The rendering of the backbone of a protein chain is accelerated using the cylinder impostor to create tube-like coils that trace the backbone C_α atoms. The C_α atoms in each residue of the polypeptide chain are used as control points of a uniform cubic B-spline similar to the approach by Krone et al. [11] (see Figure 8a). This spline is represented as a sequence of line segments. The number of samples is chosen to balance efficiency and quality. The cylinder impostor is used to render cylindrical elements for each line segment on the spline with cut planes determined by the preceding and succeeding line segments (see Figure 8c).

α-Helices.

Rendering the α-helix representation is accelerated by the

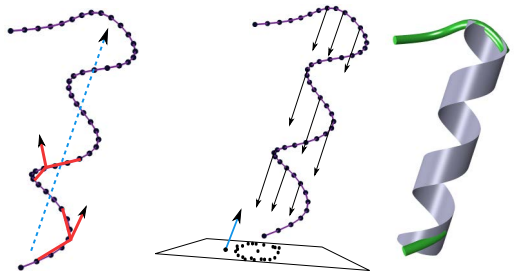


Figure 9: The helix impostor is placed and oriented using the section of the backbone spline corresponding to the helix. (a) A triplet of uniformly spaced points along the spline section is used to approximate the binormal direction at the middle point. The average of a pair of binormals from samples separated by 1.8 residues approximates the helix axis, since the helix executes a turn every 3.6 residues. The average of many such estimates is used as the helix direction. (b) The points on the spline section are projected to a plane with with normal as the helix direction to determine a point on the axis and the radius. (c) The spline section is replaced by the impostor.

use of the helical impostor (see Figure 9c). α-helices, in their most common form, have a pitch of 5.4 Å, and are thus closely approximated by an ideal helix. Also, they execute a complete turn every 3.6 residues. First, the direction of the helical axis is determined. Kahn [9] describes a popular algorithm to determine the axis. However, it requires at least four consecutive C_α atoms to be present on the helix. This condition does not hold in many cases. We use a simpler approximation, using the backbone spline section corresponding to the helix. We begin by first sampling sets of three equally spaced points on the spline section. Each triple provides a pair of directions oriented from the second to the first and third. The spacing of the samples along the spline section is controlled such that their inner product is lesser than a threshold. This ensures that the cross product does not approach the zero vector. The cross product coarsely approximates the binormal vector direction. This vector can be resolved into a component that aligns with the helix axis (ideally) and one that lies on the plane containing the helix axis as well as the binormal vector. The second component is canceled by a binormal vector from a second triple sample diametrically opposite on the helix. We obtain the second sample 1.8 residues away from the current sample. The axis direction is averaged over as many such pairs of sample triplets that fit on the spline section. A point on the axis is determined by first projecting all the points on the spline section to the plane with normal as the previously computed axis direction, and then computing the mean of the projections (see Figure 9b). The radius of the helix is computed as the average distance between the mean and all projections. The end points on the helical axis are obtained by projecting the end points of the backbone spline section to the helical axis. The helix is rotated such that it passes through the first sample point on the spline section (see Figure 9c).

Viewer	SF	BS	BB	α	SS
VMD 1.9.1	2.38	2.0	25	-	35
PyMol 1.4	45	10	-	-	21
PROTEINVIS	52	60	68	460	83

Table 1: Frame rates of our implementation for various representations of the GroEL molecule 1AON compared with other viewers. First column lists the viewer name. Columns 2-6 show the frame rates for space-fill representation (SF), ball-stick representation (BS), backbone representation using cylinders (BB), α-helix representation (α), and secondary structure representation (SS) consisting of α-helices, β-sheets, and loops.

5. RESULTS

The above discussed techniques were implemented in a bio-molecular visualization software PROTEINVIS. To validate our approaches quantitatively, we compare our algorithm with existing software.

We conducted our evaluations on an Intel Xeon workstation, with 16GB RAM, and an NVIDIA GeForce GTX 260 Graphics card with 896 MB memory. In our experiments, we use two lights and a viewport of 1024x768. For the backbone representation using the cylindrical impostor, the spline was sampled with six points for each section of the spline.

Comparison with Popular Protein Viewers.

We compute the frame rate and compare our implementation against popular molecular visualization programs, namely VMD [8] and PyMol [14]. Both VMD and PyMol implement a spherical impostor method for space-fill renderings. Table 1 shows the comparison of the frame rate for the above viewers for the GroEL molecule 1AON (approximately 59K atoms, see Figure 10). Our method performs considerably better for the space-fill and ball-stick representations. For the tube representation, to generate comparable visuals, VMD was configured to use ten sided polygonal approximations of circular elements, shaded using phong shading. We again note that the performance is considerably better. For the α-helix, we observe good performance due to the usage of a single element for each helix. Other viewers offer visualization of helices together with loops and β-sheets. Hence, we report frame rates for rendering all three structures in our implementation and compare it with the cartoon and new cartoon representations in PyMol and VMD respectively. Again, we observe better performance due to the usage of quadric primitives. Figure 11 shows a comparison of the rendering secondary structures of the ribonuclease 1D5H molecule with VMD. Artifacts due to the spline twisting, which are present in VMD's and PyMol's rendering, are not seen in PROTEINVIS's rendering.

Performance Statistics and Quality.

Table 2 shows the frame rates for molecules of various sizes. We note that the rendering of α-helices using helical impostors is significantly faster than other representations due to the usage of fewer primitives. Figure 9c shows the backbone spline along with the helical impostors for the 1D5H molecule. As can be seen, the helical impostors closely approximate the turns of the backbone spline.

We also compare the performance of our secondary structure rendering with Krone et al. [11] (see Table 3). In par-

Molecule	# atoms	SF	BS	BB	α	SS
1D5H	1122	556	900	2000	6000	1538
4RHV	6267	180	407	600	5000	500
1RCX	37,455	61	113	116	600	120
1AON	58,673	55	80	68	460	83
1HTQ	90,672	41	60	50	111	54

Table 2: Frame rates for various representations of molecules with varying sizes. First column lists the PDB ID ([2]) of the molecule. Second column lists the number of atoms in the PDB file. Columns 3-7 show the frame rates for space-fill representation (SF), ball-stick representation (BS), backbone representation using cylinders (BB), α-helix representation (α), and secondary structure representation (SS) consisting of α-helices, β-sheets, and loops.

Molecule	# atoms	SS	Hybrid [11]
1OGZ	943	870	550
1VIS	2481	556	200
1TII	5478	274	150
1AF6	10,049	160	100
1AON	58,673	34	10

Table 3: Frame rates for molecules with varying sizes rendered using the backbone tubes and helices. First column lists the PDB ID ([2]) of the molecule. Second column lists the number of atoms in the PDB file. Third column shows the frame rates of our implementation of secondary structures (SS). Fourth column shows the frame rates of Krone et al.'s [11] hybrid method for backbone, α-helices and β-sheets. Both renderings use the same graphics hardware; an NVIDIA GTX 8800 card with 768 MB RAM.

ticular, we compare with their hybrid implementation using six segments per spline and phong lighting using the same graphics hardware *i.e.* GTX 8800 with 768 MB of graphics memory. We find that our implementation performs better, primarily because of the usage of fewer primitives for the geometry of the backbone and helices. We note that even though our technique is computationally intensive in the fragment shader stage, it performs better overall.

6. CONCLUSIONS

We have presented a unified two stage framework to render quadric elements using GPUs. This framework is implementable using geometry and fragment shaders. We adopt this framework to define quadric "impostor" elements for spheres, cylinders cut by arbitrary planes, and helices. We implement these elements in protein visualization to accelerate the rendering of space-fill, ball-stick, backbone-tubes, and α-helix representations of proteins. We show quantitative and qualitative improvements over earlier methods for rendering similar representations.

In future it would be interesting to consider applying our secondary structure rendering techniques for nucleic acids. Here too, the presence of regular structures in the form of double helices might be exploitable on GPUs. Currently, impostored helices appear as thin strips. Development of "thick" helices, as is common in other viewers, using impostors is challenging. The implicit assumption of impostoring helices is that the axis for each helix roughly remains the same. This is not true for molecules such as haemoglobin. Development of adaptive impostoring techniques in such cases is also an interesting direction for future work.

7. REFERENCES

[1] C. Bajaj and P. Djeu. Texmol: Interactive visual exploration of large flexible multi-component molecular complexes. In *Proc. IEEE Conf. Visualization*, pages 243–250, 2004.

[2] H. M. Berman, J. Westbrook, Z. Feng, G. Gilliland, T. N. Bhat, H. Weissig, et al. The Protein Data Bank. *NAR*, 28:235–242, 2000.

[3] F. C. Bernstein, T. F. Koetzle, G. William, D. J. Meyer, M. D. Brice, J. R. Rodgers, et al. The protein databank: a computer-based archival file for macromolecular structures. *JMB*, 112:535–542, 1977.

[4] J. F. Blinn. Models of light reflection for computer synthesized pictures. *SIGGRAPH*, 11(2):192–198, July 1977.

[5] E. Catmull and R. Rom. A class of local interpolating splines. *Comp. aided geom. design.*, pages 317–326, 1974.

[6] R. B. Corey and L. Pauling. Molecular models of amino acids, peptides and proteins. *Rev. Sci. Instr.*, 24:621–627, 1953.

[7] S. Gumhold. Splatting illuminated ellipsoids with depth correction. In *Proc. VMV*, pages 245–252, 2003.

[8] W. Humphrey, A. Dalke, and K. Schulten. VMD – Visual Molecular Dynamics. *J. Mol. Graph*, 14:33–38, 1996.

[9] P. C. Kahn. Defining the axis of a helix. *Computers & Chemistry*, 13(3):185–189, 1989.

[10] W. L. Koltun. Precision space-filling atomic models. *Biopolymers*, 3:665–679, 1965.

[11] M. Krone, K. Bidmon, and T. Ertl. Gpu-based visualisation of protein secondary structure. *Proc. of TPCG 2008*, pages 115–122, 2008.

[12] M. Krone, J. Stone, T. Ertl, and K. Schulten. Fast visualization of gaussian density surfaces for molecular dynamics and particle system trajectories. pages 67–71.

[13] O. D. Lampe, I. Viola, N. Reuter, and H. Hauser. Two-level approach to efficient visualization of protein dynamics. *IEEE Trans. Vis. Comput. Graph.*, 13(6):1616–1623, 2007.

[14] L. Schrödinger. The PyMOL molecular graphics system, version 1.3r1. August 2010.

[15] C. Sigg, T. Weyrich, M. Botsch, and M. Gross. GPU-based ray-casting of quadratic surfaces. In *Eurographics Symp. Point-Based Graph.*, pages 59–65, 2006.

[16] M. Tarini, P. Cignoni, and C. Montani. Ambient occlusion and edge cueing for enhancing real time molecular visualization. *IEEE Trans. Vis. Comput. Graph.*, 12(5):1237–1244, 2006.

[17] M. Woo, J. Neider, T. Davis, and D. Shreiner. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 1999.

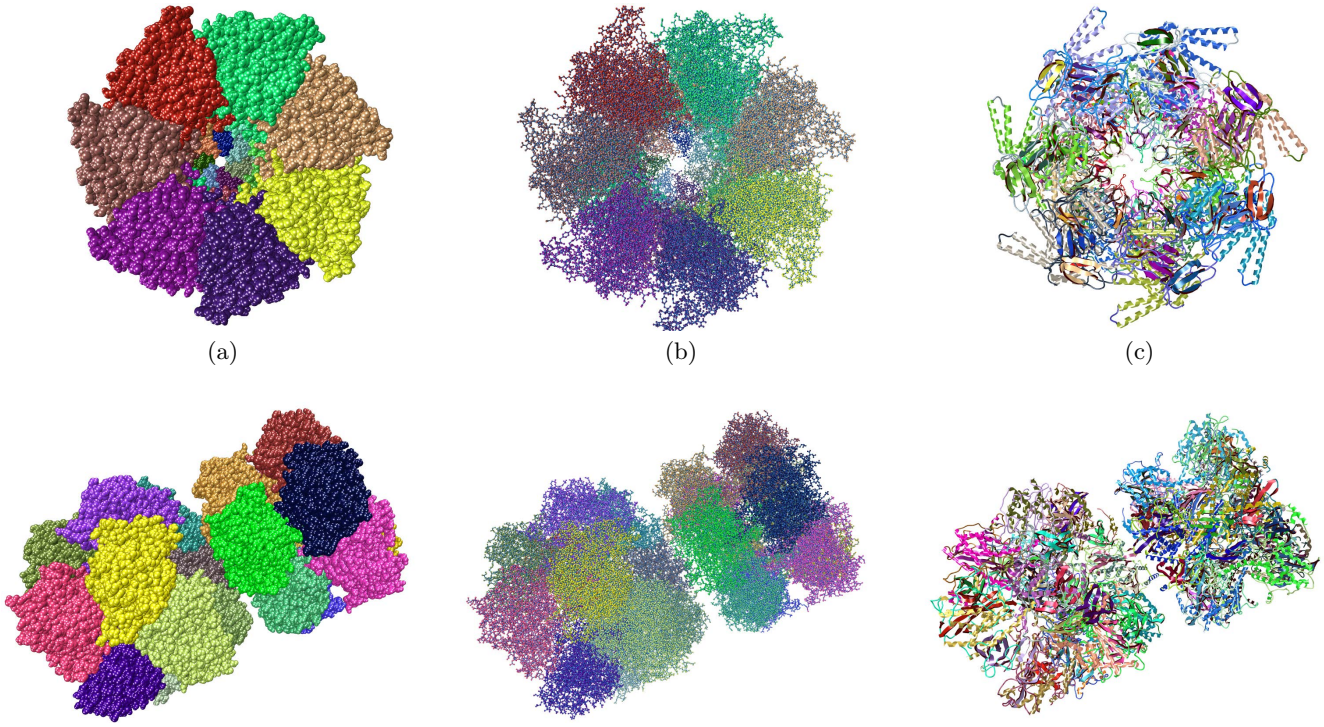


Figure 10: Visualizations of the primary and secondary structures of the GroEL 1AON and glutamine synthetase 1HTQ molecules, which contain approximately 59,000 atoms and 90,000 atoms respectively. (a) , (d) Space-fill representation. (b) , (e) Ball-stick representation. (c) , (f) Secondary structure shown with α-helices β-sheets and loops.

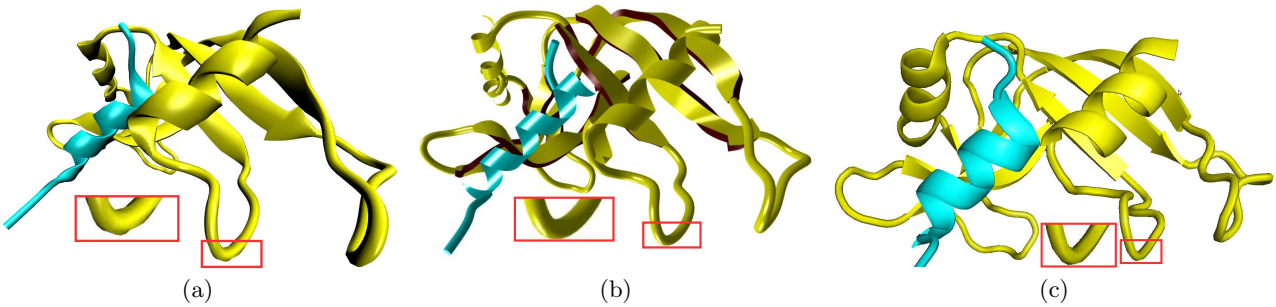


Figure 11: The secondary structures of ribonuclease molecule 1D5H rendered using (a) VMD, (b) PROTEINVIS, and (c) PyMol. Artifacts due to the spline twisting are present in VMD's and PyMol's rendering, which are not seen in PROTEINVIS's rendering. The number of segments per spline section was six for both PyMol and PROTEINVIS. VMD only allows configuring the number of segments used to approximate the circular cross section of a tube which was also set to six.