GPU-Accelerated Visualization of Protein Dynamics in Ribbon Mode

Manuel Wahle and Stefan Birmanns

University of Texas Health Science Center at Houston,
School of Biomedical Informatics,
7000 Fannin, Ste. 600, Houston, TX 77054

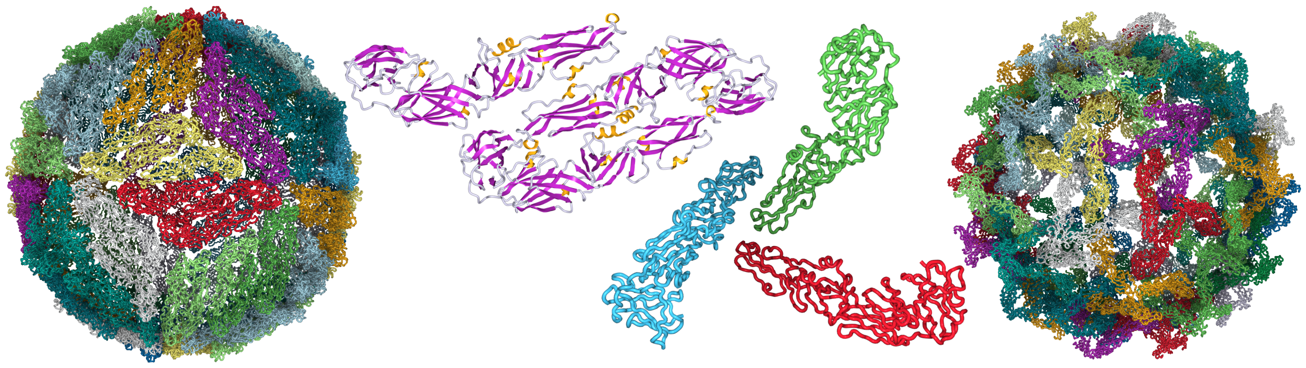
Figure 1. Left and right: Dengue Virus in different conformational states. In the middle, a cartoon and a tube depiction of a capsid face in the different states with different coloring schemes.

ABSTRACT

Proteins are biomolecules present in living organisms and essential for carrying out vital functions. Inherent to their functioning is folding into different spatial conformations, and to understand these processes, it is crucial to visually explore the structural changes. In recent years, significant advancements in experimental techniques and novel algorithms for post-processing of protein data have routinely revealed static and dynamic structures of increasing sizes. In turn, interactive visualization of the systems and their transitions became more challenging.

Therefore, much research for the efficient display of protein dynamics has been done, with the focus being space filling models, but for the important class of abstract ribbon or cartoon representations, there exist only few methods for an efficient rendering. Yet, these models are of high interest to scientists, as they provide a compact and concise description of the structure elements along the protein main chain.

In this work, a method was developed to speed up ribbon and cartoon visualizations. Separating two phases in the calculation of geometry allows to offload computational work from the CPU to the GPU. The first phase consists of computing a smooth curve along the protein's main chain on the CPU. In the second phase, conducted independently by the GPU, vertices along that curve are moved to set up the final geometrical representation of the molecule.

Keywords: molecular visualization, time-varying protein, protein structure, GPU acceleration

Further author information: (Send correspondence to S.B.)
M.W.: E-mail: manuel.wahle@uth.tmc.edu
S.B.: E-mail: stefan.birmanns@uth.tmc.edu

Visualization and Data Analysis 2011, edited by Pak Chung Wong, Jinah Park, Ming C. Hao, Chaomei Chen, Katy Börner, David L. Kao, Jonathan C. Roberts, Proc. of SPIE-IS&T Electronic Imaging, SPIE Vol. 7868, 786805 · © 2011 SPIE-IS&T · CCC code: 0277-786X/11/$18 · doi: 10.1117/12.872458

1. INTRODUCTION

Over the past decade, a steady improvement of experimental techniques has caused a major shift in the focus of structural biology. While the study of small biomolecules has been the main focus in the past, it is nowadays possible to investigate large macromolecular assemblies or even virus capsids with molecular masses in the range of megadaltons. Cryo-electron microscopy,[1] tomography,[2] and small-angle x-ray scattering[3] are routinely employed to image systems that do not crystallize, or to visualize the supra molecular organization of even larger cellular structures. Taking these considerations into account, it is obvious that the visualization of those time-varying biomolecular data sets becomes challenging.

The reason is that the construction of the geometry, done by the CPU, takes much more time than the projection of the resulting data onto the screen. For static models the construction needs to be conducted only once, and so the visualization is less challenging. But the CPU quickly becomes a limiting factor when dealing with dynamic systems, where the geometry has to be reconstructed for every frame.

In this paper a method is proposed that aims to speed up the process of generating the geometry and to deliver high optical quality. These two aims are reached by a combination of three ideas: outsourcing mathematical operations for the construction of the geometrical data to the GPU, introducing a method to compute local reference frames along the main chain of the protein, and computing a per pixel lighting. The visualization modes of interest are abstract ribbon or cartoon modes. Although they are one of the most important, if not the most important way of visualizing biomolecules, not much work regarding their rendering speed has been done.

Review of Protein Basics

Proteins consist of long chains of amino acids, used as building blocks. Amino acids are only distinct in their side chain (connected to the central carbon atom C_α), which also determines their chemical properties. Through peptide bond formation, amino acid residues are joined (Figure 2(a)).

In the chain of residues, an important feature to observe is that six atoms always lie in a plane. This peptide plane is determined through the atoms at its edges, which are the C_α and an oxygen atom of the n-th residue, and a hydrogen and the C_α of the $n+1$-th residue (see Figure 2(b)). Together with a nitrogen of the n-th and a carbon atom of the $n+1$-th residue they build what is called the backbone of the protein. The side chain and an additional hydrogen atom of the original amino acid is connected to each C_α.

Of interest for this work is the cartoon visualization, which illustrates the backbone with its secondary structure.* The existing secondary structure types can be sorted into three classes, each depicted by their own visual metaphor. The first are the helices, where the peptide planes are oriented around and along an imaginary cylinder. They are visualized by a ribbon with a rectangular cross section following a helical path. The second are the beta strands, represented by an arrow with the same cross section. The third are the turns, which do not have any specific structural arrangements. They connect the other types of secondary structures and are depicted by a smaller, round ribbon.

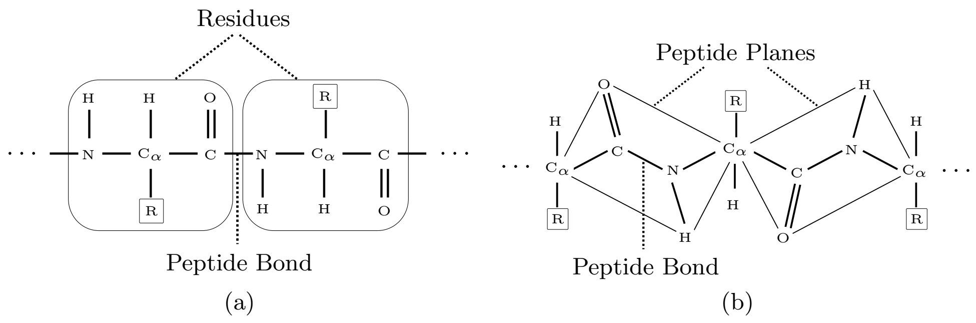

Figure 2. Protein basics. Residue, peptide plane.

*The secondary structure describes the folding of neighboring peptide planes along parts of the backbone.

2. RELATED WORK

In 1981, the ribbon drawing mode was popularized by Richardson,[4] becoming an important and often used style to depict proteins. As opposed to space-filling models like the van der Waals visualization (see Figure 3(a)), ribbon or cartoon renderings show a more abstract representation (Figure 3(b)). By omitting position, size, and bonding information of each individual atom, they give a representation that enables a researcher to grasp a much better overview of the protein. At the same time, they exhibit the secondary structure, giving a more informative view than a space-filling model. Cartoons are perhaps the most popular renderings among biologists. See[5] for a short review of visualization methods for biomolecules.

A related field of visualization is the one that is concerned with the visualization of streamlines or vector fields. The aim is to render huge numbers of lines or tubes that follow certain paths, illustrating, for example, currents in gas or liquid. Rendering techniques must, on the one hand, be efficient enough to handle the amounts of tubes to be rendered. On the other hand, they must add effects to visually enhance the renderings to enable useful perception of these structures.

To tackle the efficiency aspect, almost all approaches employ the idea of using rectangles representing segments of the tubular geometry. Oriented orthogonal to the viewing direction, projected onto the screen, and shaded appropriately, a rectangle looks exactly like an actual tube segment would. In[6] a rigorous mathematical formalism is presented to compute "correct" rectangles. Via a CPU-GPU hybrid method, renderings of almost perfect quality are achieved. The approach in[7] differs from[6] in that it trades off quality against rendering speed. Through simplified, approximative representations, rendering speed is greatly enhanced and a huge data set can be dealt with interactively.

When working with dynamic, biomolecular datasets, it is of interest to render an animation of how a molecule or a system of molecules folds from one spatial conformation to another. The challenge in rendering is that the data is dynamic, which means that from frame to frame, the representing geometry must be completely recomputed. The problem is not rendering the geometry, but constructing it for every frame.

Many different software suites are devoted to molecular visualization. They are routinely used as a means for exploring and understanding the structure and architecture of biomolecules, see for example[8] for a categorized overview. Rasmol[9] is a classic molecular visualization system aimed at the efficient generation of renderings for teaching and research purposes. Chimera[10] is a program for interactive visualization and analysis of molecular structures and related data. Visual Molecular Dynamics (VMD)[11] focuses on the results of molecular dynamics simulations.

Independently from these established software packages, a lot of work has been done on efficiently visualizing molecules. An important classification can be done according to the graphical primitives that are used. On the one hand, there are rendering modes depicting the individual atoms. They use independent geometry for each individual atom or bond, and map, therefore, extremely well to the rendering pipeline.

Hao et al.[12] achieve a speed-up for the van der Waals representation, using visibility based occlusion culling and a multi-resolution technique. Lampe et al.[13] employ the programmability of a modern GPU. The main idea

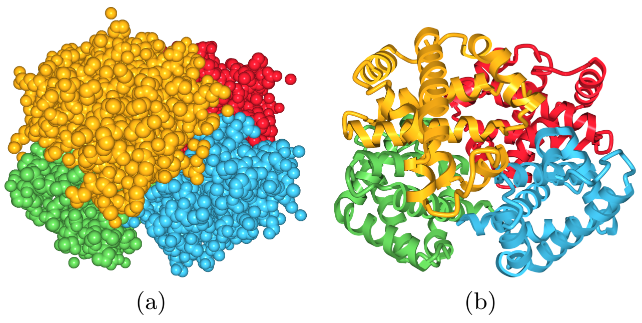

Figure 3. Protein Visualization Modes. The depicted protein is the human hemoglobin. In (a) it is drawn as a space-filling representation, illustrating the spatial extent of the protein. The abstract cartoon diagram in (b) shows structural properties of the backbone. The coloring scheme shows the two different chains that comprise this protein.

is to not transfer geometry for each atom in a residue, but to only send one coordinate along rotational and translational data. The final geometry is then generated on the graphics card. Qutemol[14] is a program which highly adapts the rendering pipeline for fast and high-quality rendering of van der Waals spheres.

On the other hand, there are renderings that show an abstracted model of the protein backbone. These are the ribbon visualization modes, of which the cartoon is the most popular. A ribbon following the backbone depicts structural characteristics by changing color and cross section. As opposed to the visualizations based on individual atoms, the ribbon models consist of geometry whose topology is more complicated to compute.

An approach for efficient cartoon rendering is presented by Halm et al.,[15] used in the visualization package BioBrowser.[16] The authors employ Combined BReps, which are polygonal boundary representations combined with Catmull/Clark subdivision surfaces. The level of detail can be adapted on the fly to keep the number of vertices minimal and achieve high rendering speed. A method based on the idea to conduct parts of the geometry generation on the GPU is presented in.[17] Control points are sent to the GPU, where geometry shaders compute B-splines to emit vertices building a smooth curve along the protein backbone. This reduces the amount of vertices the CPU has to deal with. However, the frame rate surprisingly drops far below the levels of their reference implementation where the CPU computes the spline.

Despite the importance of ribbon models for actual biological research, there is to the knowledge of the authors no method that explicitly employs programmable GPUs speeding up the generation of a ribbon representation for dynamic molecules. The work presented in this paper fills this gap.

3. METHOD

The aim of this work is to design a method that speeds up the ribbon visualization of a dynamic protein. It is key to minimize the time needed for geometry construction because with a dynamic scene, this has to be done for every frame. The means to achieve this is to involve the GPU actively in the geometry construction. As opposed to streamline visualization, we do not put the emphasis on rendering the geometry, but on creating it as efficient as possible. Since in the field of molecular modeling, many different data sets of different sizes reside in memory at the same time, no precomputations or additional memory usage should be introduced.

The proposed method is implemented into Sculptor,[18] a program for molecular multi-resolution modeling, which already has many different visualization modes. It also provides a mature testing environment to take measurements and evaluate the performance of the method.

3.1 Outline

For a static molecule, the time used to create a geometrical representation is not critical because once it is uploaded to the GPU, it can reside in the video card's memory. The GPU has enough computational resources to handle the user's viewing requests, so that an interactive experience is guaranteed. But when displaying a molecule that changes its shape from frame to frame, the geometry has to be recreated and uploaded to the video card for every frame. This is done by the CPU and takes considerable time. When displaying larger molecular systems, this easily impedes an interactive frame rate.

To speed up the geometry construction at each frame, the computation of the geometry is separated into two phases. In the first phase, a smooth path through the molecule's backbone is computed. At every vertex along this path, data for a local reference frame are generated. This phase is done on the CPU, because the computation of the spline path requires global information. The CPU stores the vertices and their associated data in an interleaved vertex array.

The computation of the spline path is done via a Kochanek-Bartels[19] spline. Like the popular Catmull-Rom spline, this is a cubic Hermite spline. It has the advantage of being controllable via continuity, tension and bias parameters. Those allow one to adapt the shape of the computed curve, which is beneficial at the round helix parts. Four points in space are input to the spline, and the spline computes new vertices between the two middle positions; the two outer points yield the tangent information that allow for an adaption of the shape of the resulting curve. The special cases at the beginning and at the end of the backbone are solved by duplicating the first and the last atom, respectively.

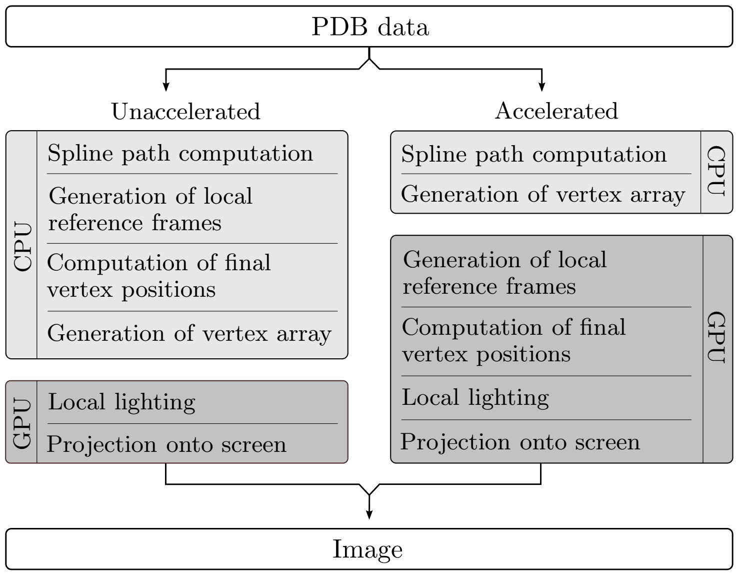
PDB data
Unaccelerated
Accelerated
CPU
Spline path computation
Generation of local reference frames
Computation of final vertex positions
Generation of vertex array
GPU
Local lighting
Projection onto screen
Spline path computation
Generation of vertex array
CPU
Generation of local reference frames
Computation of final vertex positions
Local lighting
Projection onto screen
GPU
Image

Figure 4. Schematic outline of the method, illustrating what parts of the computation are outsourced to the GPU from the unaccelerated to the accelerated version.

For the second phase, the vertex array is sent to the graphics card using a vertex buffer object. The GPU uses the data at each vertex to compute a local reference frame and move the vertex to its final position. A rectangular ribbon is comprised of a set of triangle strips. For each of these triangle strips, a different vertex shader program is called to move the vertices. For a ribbon with four sides, there is one program the moves the vertices to the upper side, and another one for each of the three other sides. After this is done for all sides of the ribbons, the protein's geometrical representation is set up.

The desired speed-up for the geometry generation is achieved by letting the GPU compute the local reference frame and using it to move the vertices to their final positions. This is a computationally expensive part since it requires a lot of operations on three dimensional vectors.

Figure 4 illustrates what parts of the computations are additionally conducted on the GPU in comparison to rendering in a conventional way.

In the following three subsections it is explained in detail how the geometry for the three different types of secondary structure along the protein's backbone is constructed.

3.2 Beta Strands

In Figure 5(a), the peptide planes of a beta strand are shown. The coloring of the two outer planes indicates where a strand starts and ends and that it is connected to a random coil part. The alternating light and dark shading of the planes illustrates the orientation of the peptide planes, showing that they are twisted by roughly 180 degrees to each other.

First, the vertex positions are computed as a spline path through the middle of the peptide planes. The local data at each vertex are in case of the strands, a binormal, the position of the next vertex, and color information.

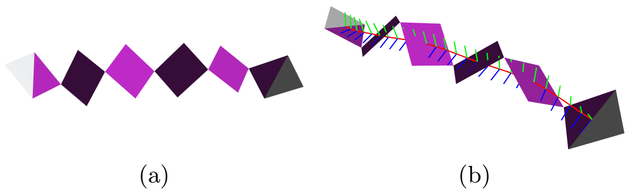

Figure 5. Beta strand peptide planes and local frames. (a) illustrates the different orientation of adjacent peptide planes, (b) shows the local reference frames along the spline path used to extrude the geometry.

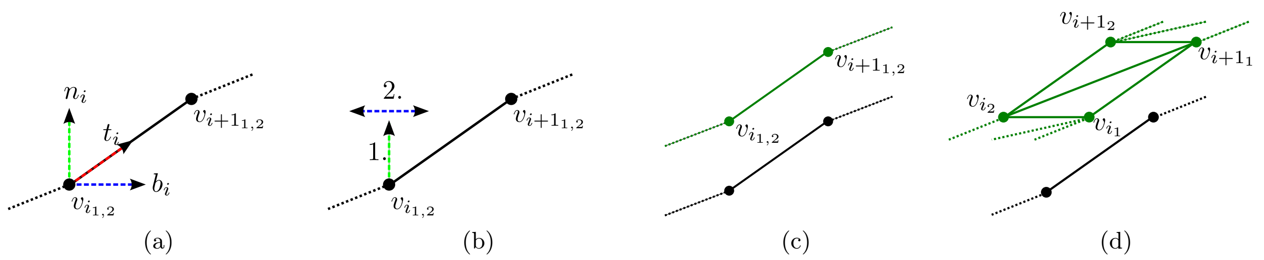

Figure 6. Movement of vertices by the GPU. (a) shows the local coordinate system at a vertex. (b) shows the movements for the upper triangle strip. First, a vertex pair is moved in the direction of the normal to the positions shown in (c). Then one of the vertices is moved into the direction of the binormal and the other one into the opposite direction. This is done for all vertex pairs along the spline path.

Binormals at the vertices are a weighted sum of orientation vectors of neighboring peptide planes. A peptide plane's orientation vector points from its O- to the H-atom in the peptide plane[†] (see Figure 2(b)). At every other peptide plane, this vector is multiplied by -1 to make all binormals point in roughly the same direction. The binormal b_{i_j} at the j-th vertex between peptide planes i and $i+1$ is computed as

$$b_{i_j} = j \cdot B_i + (n-j) \cdot B_{i+1},$$

where B_i and B_{i+1} are the orientation vectors of the corresponding peptide planes. $j \in \{0, \ldots, n-1\}$, where n is the number of vertices in the spline path between the two peptide planes.

Because they are of the same length, the orientation vectors don't need to be normalized, saving some computational work on the CPU. Each vertex is inserted twice into an interleaved vertex array. The only difference within this pair is that the first vertex has a value of 1 in the fourth component of its texture coordinate, whereas the second has a -1. This way, the vertex shader can differentiate between the two vertices. The binormals b_i are stored in each vertex's normal parameter, and the texture coordinate is used to store the coordinates of the next position v_{i+1} in the spline path.

After the generation of the vertex array, it is sent to the video card using a vertex buffer object (VBO). The advantage of a VBO is that it can reside in the video memory and be drawn multiple times, keeping transfer of vertex data from main to video card memory minimal.

In the case of rectangular strand ribbons, the vertex array is drawn four times, once for each of the four sides. For each side, a different shader program moves the vertices to the corresponding positions. In the following, this process is described in detail for the upper side of the ribbon (Figure 6). The other three sides are computed analogously.

On the GPU, the vertex shader uses the xyz-components of the vertex position v_i and that of the next vertex (stored as texture coordinate) v_{i+1} to compute the tangent t_i (in Figure 6(a), that would be the vector from v_i to v_{i+1}). The binormal b_i is given in the vertex normal. A cross product of tangent and binormal yields the normal, so

$$t_i = v_{i+1} - v_i, \quad \text{and} \quad n_i = t_i \times b_i.$$

Note that the frame is only nearly orthogonal, but indeed sufficient for the purpose of moving the vertices. After their computation, normal and binormal have to be normalized.

The next step is to update the vertex position. In the case of the upper side, vertices have to be moved into the direction of the normal (1. in Figure 6(b)) by half the height of the ribbon . Since the vertices for the lower side will be moved by the same amount into the opposite direction, the resulting positions will span the full height. So $v_i = v_i + (h \cdot 0.5) \cdot n_i$. The height h is passed to the shader program as a uniform parameter.

Vertices also have to be moved along the direction of the binormal (2. in Figure 6(b)), which determines the direction of the width of the ribbon. The width is stored in the fourth component v_{i_w} of the vertices. It is not given as a uniform parameter to the shader because it changes for the vertices at the arrowheads. To compute

[†]The position of the H-atom is approximated since it is often omitted.

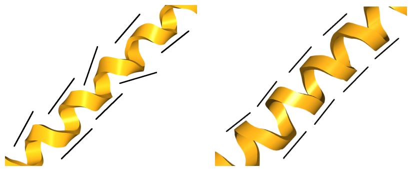

Figure 7. Helix binormal information. On the left image, the binormal direction follows the orientation of the peptide planes. Their irregular structure causes strong visual distortions. On the right image, a simple and fast heuristic is used to computed binormals that lead to a more appealing geometry.

the displacement of a vertex, the binormal is multiplied with v_{i_w} and t_{i_w}, so $v_i = v_i + b_i \cdot (v_{i_w} \cdot t_{i_w})$. Since for each vertex pair at a position in the spline path, t_{i_w} is 1 for one vertex and -1 for the other, the two vertices get displaced into opposite directions.

In summary, the formula for updating a vertex' position is

$$v_i = v_i + n_i \cdot h + b_i \cdot (v_{i_w} \cdot t_{i_w}),$$

which results in an efficient and compact vertex shader program.

3.3 Helices

The generation of helix geometry is done in a similar manner. In fact, movement of vertices on the GPU is conducted by the same vertex shader program. The only difference lies in the generation of binormal information computed on the CPU.

Most schemes[15, 20, 21] use the orientation of the peptide planes or the planes given by three consecutive C_α atoms to compute this information. This method is quite fast, but it can result in ribbons that are oriented like in Figure 7(a). To avoid this effect, a cylinder could be fitted analytically to the helix, but the required mathematical operations can be computationally intensive.[22] To avoid these time consuming computations, a simple and fast heuristic is employed that still delivers high optical quality.

The heuristic works as follows. The positions of a triple of consecutive C_α atoms are added $A = C_{\alpha_i} + C_{\alpha_{i+1}} + C_{\alpha_{i+2}}$. The same is done for the following triple of C_α atoms: $B = C_{\alpha_{i+3}} + C_{\alpha_{i+4}} + C_{\alpha_{i+5}}$. The binormal results from subtracting the two results, $\vec{b} = B - A$.[‡] The binormal vectors generated with this heuristic are illustrated in Figure 7(b). By completely avoiding divisions, this method can be executed rapidly on the CPU.

The spline path runs through the C_α atoms. The binormal information is the result of a weighted sum, similar to the strands. The only difference is that it has to be distributed to those vertices on the spline path that lie before the average position of the first triple of C_αs. Here, the binormal is not a weighted sum but simply the first binormal.

All other steps are exactly the same like with the beta strands. Each vertex is appended twice to an array, once with a 1 and once with a -1 in t_{i_w}. They are transferred to the video memory via a VBO and drawn four times with a vertex shader program for each side of the rectangular ribbon that will move the vertices to the proper positions.

3.4 Turns

The geometry for random coil parts is a round tube. To achieve a consistent look of the tube, it is important that two neighboring reference frames have a minimal twist to each other. The popular Frenet-Serret frames[23] do not have this property. Their orientation depends on curvature and torsion, so it is possible that a sudden change in orientation of adjacent frames occurs. This can result in geometry as depicted in Figure 8.

[‡]It is not necessary to actually average the three C_α coordinates because both A and B would be scaled equally, resulting in the same direction for $\vec{b}$.

While it is possible to check for this twist and to correct it, the result is not optimal. The better the desired quality of the reference frames, the more calculations are necessary. Because of this, we use parallel transport frames[24] instead of correcting the orientation of Frenet-Serret frames. Their computation is fast, and the resulting normals are more suitable for our purpose. They have the property of turning only minimally, just as much as is necessary to keep the normal orthogonal to the curve.

The parallel transport frames are computed as follows. For the first vertex in the spline path, an arbitrary normal n_i vector is generated. To create a normal for the next vertex, the previous normal is rotated around the vector that is the cross product of the current and the next tangent. The amount of rotation is the angle between these two tangents. In case the tangents are parallel, the angle is zero, and so the previous normal is reused. The process it repeated iteratively until the end of the spline path.

As with beta strands and helices, at each position along the spline path, two vertices are appended to the vertex array, again with the only difference being the 1 or -1 in t_{i_w}. Similar to strands and helices, the vertex normal parameter is used to store the normal, and the texture coordinate holds the coordinates of the next position along the spline path.

By default, the tube is generated with $n = 6$ sides, and each of them is represented by a triangle strip. The position and width of a triangle strip are determined by two parameters, angle α and span ϕ. The angle α determines the position, and the span ϕ the width (Figure 9). ϕ is computed as $(2 \cdot \pi)/n$, and α is set to $i \cdot \phi$ for the i-th strip (n is the number of strips, $i \in \{0, \ldots, n-1\}$). While ϕ is the same for all strips, α has to be updated for each triangle strip.

The normal n_i is already transmitted with the vertices, so the vertex shader has to compute the binormal. Similar to strands and helices,

$$t_i = v_{i+1} - v_i, \text{ and } b_i = n_i \times t_i.$$

The final position of a vertex is computed as

$$v_i = v_i + r \cdot b_i \cdot \cos\left(\alpha + t_{i_w} \cdot \frac{\phi}{2}\right) + r \cdot n_i \cdot \sin\left(\alpha + t_{i_w} \cdot \frac{\phi}{2}\right),$$

where r is a uniform parameter in the vertex shader representing the radius of the tube. Each one of two vertices at a position in the spline path is moved by half of the span, so that together they span the full triangle strip width.

Since the vertices are contained in a VBO, they can reside in the video card's memory. This means that the data transfer from main to video memory is independent of the number of sides with which the tube is drawn. As long as the GPU has free resources, the number of tube sides can be increased without impact on the frame rate, since the amount of work for the CPU stays constant. The more sides the tube has, the better it looks.

3.5 Lighting

The geometry is shaded with Phong lighting, which can be done on a per-vertex or per-pixel basis. In the case of vertex lighting, the standard fragment shader of the OpenGL pipeline interpolates the colors. With per-pixel lighting, the vertex shader emits with each vertex its object space position and normal. The raster operation units of the GPU interpolate this information for each pixel, and so the fragment shader can compute a smooth shading.

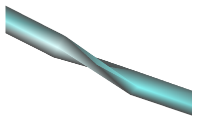

Figure 8. Effect of twisted local frames computed with the Frenet-Serret formulas. Parallel transport frames avoid this by guaranteeing minimal twisting between adjacent frames.

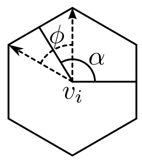

Figure 9. Illustration of movement of vertices at a position v_i for one of the triangle strips that comprise the tube geometry. The vertices' final positions are indicated by the arrows. The angle α determines the position for the triangle strip, the span ϕ determines how wide it is.

In addition to the high optical quality, the per pixel lighting can even speed up the rendering process. On older graphics hardware that doesn't implement the unified shader architecture, the number of hardware units that runs shader or fragment programs is constant. Usually the CPU restricts the frame rate, but when drawing tubes with a high number of sides, the GPU can become the bottleneck. In this case, the vertex shaders are too busy. Activating the per pixel lighting shifts some computational work from the vertex to the fragment shader, and the frame rate increases a bit.

All three types of OpenGL fog are supported to enhance depth perception. Furthermore, the method works fine in combination with a real-time ambient occlusion algorithm that is computed as a screen space post processing step.[25] The teaser image on the first page of this paper is rendered with fog and ambient occlusion.

4. RESULTS

To measure the impact of the outsourcing of the computations from the CPU to the GPU, two versions of the presented method were implemented: One as described above, and the other where the whole process of constructing the geometry is conducted by the CPU. The computations are analogous between the two versions, which ensures an adequate comparability of the two methods.

To assess the speed-up, a set of molecules with different sizes were rendered two times. Since it is of interest how the shader programs for strands/helices and for tubes speed up the geometry construction process, two different measurements are relevant. First, the molecules were rendered as a helix, and secondly as a tube. The algorithm was tested on three different machines: a 3.0GHz C2D/GeForce GTX295, a 2.4GHz C2D/GeForce 8800GTS, and a 2.2GHz Athlon64X2/GeForce 6600LE. All were running a recent Linux distribution and had 2GiB main memory.

Table 1 shows the speed-up of the algorithm for helices/strands, Table 2 for tubes. The chart in Figure 10 plots the speed-up factor against the number of residues for all three test systems. The number of residues is linearly correlated to the number of vertices. For visualization modes that give an abstract depiction of the backbone (like cartoon or tube), this is the appropriate dimension. Comparing the number of atoms against the frame time is misleading, since the atoms themselves are not rendered.

Table 1. Strand/Helix Frame Time Comparison (2.4GHz C2D/GeForce 8800GTS). Frame Time Given in Milliseconds.

PDB ID	#Residues	frame time (unaccelerated)	frame time (accelerated)	Speed-up factor
1VAK*	5910	24	9	2.7
1IJS*	16440	70	26	2.7
1AYM*	24450	97	38	2.6
2ZTN	28320	120	44	2.7
2VQ0	36360	144	56	2.6
3DPR	49200	195	76	2.6
1K4R	71100	295	110	2.7

Table 2. Tube Frame Rate Comparison (2.4GHz C2D/GeForce 8800GTS). Frame Time Given in Milliseconds.

PDB ID	#Residues	frame time (unaccelerated)	frame time (accelerated)	Speed-up factor
1VAK*	5910	38	13	2.9
1IJS*	16440	106	38	2.8
1AYM*	24450	152	56	2.7
2ZTN	28320	179	65	2.8
2VQ0	36360	230	85	2.7
3DPR	49200	307	114	2.7
1K4R	71100	457	159	2.9

(* half the virus capsid as downloaded from the ViperDB[26])

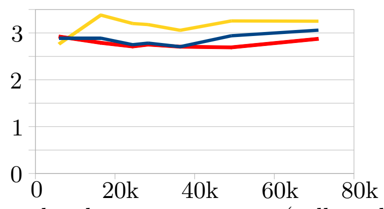

Figure 10. Comparing the tube speed-up on the three test systems (yellow slowest, red medium, blue fastest system). Speed-up is plotted against the number of residues of the resp. molecule.

5. DISCUSSION

The proposed scheme was applied successfully to achieve a fast generation of the geometry and to deliver an appealing look at the same time. The results show that by outsourcing part of the computations to the GPU, a significant speed-up factor can be achieved. For dynamic scenes, the time it takes to compute the geometry is crucial if an interactive rendering is desired. A speed-up of a factor of 2.7 or 2.8 moves a whole range of molecules into the category of those that can be rendered interactively while showing their dynamic properties.

The transfer of geometry from main to video memory also has an effect on the frame time. While in the unaccelerated version all vertices per spline path position are transferred, there are only two vertices in the accelerated version. That means the number of vertices is reduced by a factor of four for strands/helices and by a factor of six for tubes. On the other hand, the vertex size is twice as large for the accelerated version because more parameters need to be associated to each vertex. In terms of the amount of data, this still leads to a decrease by a factor of two for helices/strands and a factor of three for tubes.

For the transfer of the geometry data to the video memory, we used a compiled vertex array (CVA) for the unaccelerated version, because measurements have shown that a CVA is faster. For the accelerated version, we used a VBO, because it can reside in video memory and can be drawn arbitrarily often. This exposes an inherent advantage of our method because the same vertices are reused multiple times to create different triangle strips.

The speed-up factor is approximately the same on all different systems. The chart in Figure 10 shows that the slowest test system (yellow line) benefits slightly more than the faster ones. It also illustrates that our method scales well with increasing size of the displayed molecular system.

Additionally, since we have avoided the use of branching in the shader programs, our method runs on a wide range of hardware systems. It was successfully tested on ATI, Intel, and Nvidia graphics hardware, and on integrated chipsets as well as dedicated video cards. Without per-pixel lighting, the method can be used on a video card as early as a GeForce 4200 Ti and equivalents.

6. CONCLUSION

The presented method was applied successfully. It was shown that it scales well and a speed-up factor of about three is achieved. Additionally, it is operationable on a wide range of computers.

The limiting factor remains the computation time of the spline path on the CPU. Since the method is in principle amenable to exploit features of geometry shaders, their deployment may be worth wile. The results presented in[17] though hint to the fact that this is possibly not the case. Another disadvantage be that advanced requirements on shader profiles and thus in the hardware exclude the possibility of running our algorithm on older systems.

ACKNOWLEDGMENTS

This work was supported by startup funds of the University of Texas Health Science Center at Houston, by NIH grant R01GM62968, and by a grant from the Gillson Longenbaugh Foundation. Special thanks go to Mirabela Rusu for her support and helpful discussions.

REFERENCES

[1] Baumeister, W. and Steven, A. C., "Macromolecular electron microscopy in the era of structural genomics," *Trends Biochem. Sci.* **25**, 624–631 (2000).

[2] Lučić, V., Leis, A., and Baumeister, W., "Cryo-electron tomography of cells: connecting structure and function," *Histochem. and Cell Bio.* **130**, 185–196 (2008).

[3] Glatter, O. and Kratky, O., [*Small angle X-ray scattering*], Academic Press, London (1982).

[4] Richardson, J. S., [*Adv Protein Chem.*], vol. 34 of *Adv. Protein Chem.*, ch. The anatomy and taxonomy of protein structure, 167–339, Academic Press Inc. (1981).

[5] Goodsell, D. S., "Visual methods from atoms to cells," *Structure* **13**(3), 347–354 (2005).

[6] Stoll, C., Gumhold, S., and Seidel, H., "Visualization with stylized line primitives," in [*Visualization 2005*], 695–702, IEEE (2005).

[7] Melek, Z., Mayerich, D., Yuksel, C., and Keyser, J., "Visualization of fibrous and thread-like data," *IEEE Transactions on Visualization and Computer Graphics* **12**(5), 1165–1172 (2006).

[8] Goddard, T. D. and Ferrin, T. E., "Visualization software for molecular assemblies," *Curr. Opinion Struct. Biol.* **17**, 587–595 (2007).

[9] Sayle, R. A. and Milner-White, E. J., "Rasmol: biomolecular graphics for all," *Trends Biochem. Sci.* **20**(9), 374–376 (1995).

[10] Pettersen, E. F., Goddard, T. D., Huang, C. C., Couch, G. S., Greenblatt, D. M., Meng, E. C., and Ferrin, T. E., "UCSF Chimera – A visualization system for exploratory research and analysis," *J. Comp. Chem.* **25**(13), 1605–1612 (2004).

[11] Humphrey, W., Dalke, A., and Schulten, K., "VMD: Visual molecular dynamics," *J. Mol. Graphics* **14**, 33–38 (1996).

[12] Hao, X., Varshney, A., and Sukharev, S., "Real-time visualization of large time-varying molecules," in [*Proc. High Performance Computing Symposium '04*], Meyer, J., ed., 109–114, Simulation Councils, Inc., Arlington, VA (2004).

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

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

[15] Halm, A., Offen, L., and Fellner, D., "Visualization of complex molecular ribbon structures at interactive rates," *International Conference on Information Visualisation* **0**, 737–744 (2004).

[16] Halm, A., Offen, L., and Fellner, D., "Biobrowser: A framework for fast protein visualization," in [*Proc. EUROGRAPHICS – IEEE VGTC Symposium on Visualization*], 287–294, Eurographics (2005).

[17] Krone, M., Bidmon, K., and Ertl, T., "Gpu-based visualisation of protein secondary structure," in [*Proceedings of TPCG 2008*], 115–122 (2008).

[18] Birmanns, S. and Wriggers, W., “Interactive fitting augmented by force-feedback and virtual reality,” *J. Struct. Biol.* **144**, 123–131 (2003).

[19] Kochanek, D. H. U. and Bartels, R. H., “Interpolating splines with local tension, continuity, and bias control,” *SIGGRAPH Computer Graphics* **18**(3), 33–41 (1984).

[20] Bergman, L., Richardson, J., and Richardson, D., “An algorithm for smoothly tessellating β-sheet structures in proteins,” *J. Mol. Graphics* **13**(1), 36–45 (1995).

[21] Carson, M. and Bugg, C., “Algorithm for ribbon models of proteins,” *J. Mol. Graphics* **4**, 121–122 (1986).

[22] Christopher, J., Swanson, R., and Baldwin, T., “Algorithms for finding the axis of a helix: fast rotational and parametric least-squares methods,” *Comput. Chem.* **20**(3), 339–345 (1996).

[23] Gray, A., [*Modern Differential Geometry of Curves and Surfaces*], Studies in advanced mathematics, CRC Press, Boca Raton, FL (1993).

[24] Hanson, A. and Ma, H., “Parallel transport approach to curve framing,” Tech. Rep. 425, Indiana University Computer Science Department (1995).

[25] Wahle, M. and Birmanns, S., “Real-time ambient occlusion for visualization of multi-scale molecular models,” *submitted.* (2010).

[26] Carrillo-Tripp, M., Shepherd, C., Borelli, I., Venkataraman, S., Lander, G., Natarajan, P., Johnson, J., Brooks III, C., and Reddy, V., “Viperdb2: an enhanced and web api enabled relational database for structural virology,” *Nucl. Acids Res.* **37**, D436–D442 (2009).