Real-Time Cross-Sectioning of Dynamic Particle Systems

Pankaj Chaudhari*

Abstract

Particle Systems are extensively used to visualize a wide range of complex phenomena and volumes. Flow visualization is one such area where particle systems have capability to provide effective representations of various fields. However, the usefulness of any visualization system is strongly determined by its interactivity and by understandability of the data it represents. A system must provide means to explore the data, as exploration may reveal an insight that a set of fixed images cannot. Representations of complex fields and volumes using particles often lead to self-occluding structures. This impedes better exploration by hiding details of complex regions in the field. We provide a real-time interactive method for taking cross-sections of any field represented by particle systems without regenerating the particle dataset. This method enables a user to dynamically change the orientations of the cutting planes to observe details of the occluded regions from various view points. Usage of this method is not limited to vector fields and can be extended, rather then directly used, to explore any dynamic volumes generated using particles.

Keywords: particle system, cross-section, flow visualization, self-occlusion

1 Introduction

Particle methods are commonly used to simulate complex systems in various scientific domains. Most of the times, millions of particles are necessary to capture the behavior of a system accurately and this leads to very large and complex particle datasets. A visualization system should be able to communicate subtle changes in the three-dimensional field, organization of particles, and allow for data exploration [2004]. Unfortunately, not all visualization systems provide data exploration tools. In addition, rendering of millions of particles degrades performance of the system. Thus, it becomes difficult to provide any interactive methods for data exploration.

Uberflow [Kipfer et al. 2004] exploits various capabilities of the latest graphics hardware to improve the performance of particle systems. Kruger et al. [2005] further extend this work by providing additional visual cues and using visualization geometries like particle lines and stream ribbons. These systems could handle only a few thousands of particles. Ellsworth et al. [2004] handle a terascale particle system, but required a cluster with around fifty computational nodes and two data servers. Gribble et al. [2006] later brought particle systems to the commodity hardware such as desktops. Rendering of millions of point sprites degrades the performance and hence they employed a coherent hierarchical culling technique to restrict the number of sprites to be rendered. They also introduced data exploration methods like inter-particle reflection and ambient occlusion to enhance perception.

We describe a method that can be used for better exploration of dynamic particle systems by taking user-defined cross-sections of the field. We create a texture that defines block-based visibility for each particle and then use this value in the pixel shader to decide opacity of each particle. This GPU-based method does not work with the particle systems that use visibility culling techniques on the CPU before sending the particle data to graphics hardware. We eliminate this limitation by storing particle data in a way such that it facilitates cross-sectioning with block-level granularity on the CPU. Our method applies to any particle system that renders particles individually. Our method concentrates only upon the rendering of the particles and does not deal with the calculation of particle traces or generation of any particle dataset. This gives the ability to investigate a particular region of interest in a complex field without regenerating the particle traces. Our goal is to provide an interactive method to help using particle systems as a tool for better education and research.

The remainder of the paper is organized as follows. In Section 2 we first review the related work done in the field of particle systems used for flow visualization. In Section 3 we introduce a new method that provides a GPU-based cross-sectioning of the particle systems in real time. Section 4 provides an extension to our method to support visibility culling on the CPU. Performance and other results are described in Section 5.

2 Related Work

Particle Systems have been used for flow visualization for the last several years. Researchers have mainly focused on developing particle systems that provide real time performance with millions of particles, but "human factors" are rarely applied to them [Tory and Moller 2004]. Typically, such factors are task or domain dependent. However, many cognition and perception-based theories provide design guidelines that can be used to generate a minimal set of usability parameters. Particle systems have the potential to convey minute details of complex flow fields and incorporating them with techniques to explore the data they represent would be very valuable. Gribble et al. [2006] provided data exploration methods such as interactive viewing and use advanced lighting models. Through a formal user study they showed that their inter-particle reflections and ambient occlusion methods help to improve perception of the subtleties of complex particle systems. Ellsworth et al. [2004] provided various interactive modes by allowing filtering of particles by seedpoints. They also allow an interactive manipulation of the viewpoint and the current time step or allow time to move forward automatically, animating the particles. Coloring the particles by their age and the pressure in the field allows a user to distinguish between the particles that are injected at different time steps. All such features allow users to investigate and interrogate specific details of the field or the volume rendered. Visualization of particle traces using different shapes can significantly improve the perception of various properties of the field like pressure, flow direction, densities, etc. Kruger et al. [2005] used oriented ellipsoidal point sprites and oriented arrow sprites to effectively depict the instanta-

*e-mail: pankaj2@umbc.edu

neous direction in which a particle is moving. Oriented sprites help understand the direction of the flow even using a still image.

Lane [1994] describes particle tracing as an "embarrassingly" parallel application and hence uses parallel but expensive hardware such as Cray C90, Convex C3240 and SGI systems. Ellsworth et al. [2004] uses a PC cluster with fifty computational nodes and two high performance data servers to achieve parallelism and interactive performance for a terascale particle system. However, many users may not have such a large computational infrastructure. Therefore, researchers have been focusing on developing particle systems suitable for commodity hardware. Kipfer et al. [2004] introduced the very first GPU-based particle system, Uberflow, and demonstrated the use of commonly available hardware to achieve a real-time performance. Furthermore, Kruger et al. [2005] exploited the features of more recent graphics accelerators to calculate particle traces using an embedded Runge-Kutta integration scheme. Calculation of particle traces on GPU avoids data transfer from CPU memory to graphics memory. This approach allows for interactive streaming and rendering of millions of particles. Bruckschen et al. [2001] uses an inexpensive RAID storage system attached to a Linux machine and scales for growing data set sizes.

Gribble et al. [2006] focus on rendering large and time-varying data sets using graphics accelerators on desktop computers. They exploit the point sprite rendering capabilities of GPUs to efficiently render large number of high quality spherical glyphs using view aligned billboards as the base primitives. In addition, they achieve an interactive performance using Coherent Hierarchical Culling using CPU and GPU together for calculating the visibility of each particle. Also, for handling time varying data, CHC is extended to CHC-TV(Time Varying) that exploits temporal coherence between the frames. They also employ various advanced shading models for global illumination using multipass fragment processing. Compressed precomputed luminance textures are constructed on the graphics processor and mapped to particles during the playback.

Tarini et al. [2006] use ambient occlusion and edge cueing to enhance perception of a molecule structure rendered using a million particle system. They also allow for molecule cuts using a Z-clipping plane to study the inside details. However, their system does not allow the user to select multiple molecule cuts using cutting planes with different orientations.

According to Melanie et al. [2004], adoption of usability factors techniques by the visualization researchers is in its infancy. They suggest allowing domain-independent subtasks such as overview, zoom, filter, details-on-demand, relate, history, and extract. This work is just a single step towards achieving this goal. We aim to eliminate the need to regenerate particle traces by providing a method for data filtering to observe details about user-specified regions in a complex field or a volume.

3 GPU-based Cross-Sectioning

This section describes our method to generate a visibility texture for the blocks that can be used to decide the visibility of a particle. This method allows for user-defined cutting planes and we also provide a mechanism to undo a cutting operation.

3.1 Space partitioning and Cutting Planes

A scene to be rendered may consist of multiple particle systems. We define a bounding box for each particle system. This bounding box is then divided into smaller blocks to form a three-dimensional grid. These blocks decide the visibility of each particle. A user specifies

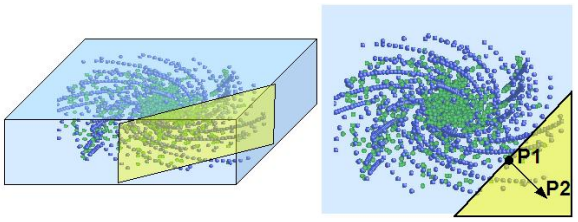

Figure 1: *Figure on the right shows how a cutting plane shown in the left figure can be selected. The user selects a point P1 in screen space and drags to point P2. P1 and P2 are then converted to object space as $P1_o$ and $P2_o$. $P1_o$ represents a point on the plane and the vector $(P2_o - P1_o)$ represents the normal to the plane.*

a cutting plane in the screen space. We allow the user to specify a cutting plane using a plane and its normal. These two points selected in the screen space are then unprojected to the world space to find the coordinates of the corresponding point on the plane and its normal. This procedure is depicted in figure 1. Screen space does not have any depth information. Thus, to obtain the depth information required to project a point into object space, we use the current depth value of at the selected screen coordinate. As the underlying hardware does not allow for a lockable depth stencil buffer, we calculate the depth information in pixel shader and store it in a texture by exploiting the capability of GPU to render simultaneous targets. Using multiple render targets decreases the performance, but makes it easy to retrieve the depth information. Thus, to convert point P1 to object space, it is first unprojected on the near plane, called $P1_n$, and then on the far plane, called $P1_f$. A ray is calculated as $P1_f - P1_n$, and then using the depth information from the current depth texture, exact coordinates in the object space are obtained as follows:

$$P1_o = P1_n + normalize(P1_f - P1_n)$$

Similar calculations are done for point P2 by obtaining $P2_o$. Thus required point on the plane is $P1_o$ and its normal is $P2_o - P1_o$.

Blocks in the three-dimensional grid for each particle system are then tested for their visibility against the user-defined plane and are marked as visible or clipped accordingly. This allows for block-level granularity for each particle system in the scene.

3.2 Building a Visibility Texture

Each particle system maintains its own visibility texture that can be used later in vertex shader to decide the visibility of each particle. This visibility texture defines the visibility of a particle system in its own object space.

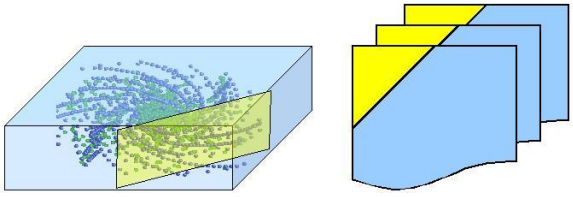

Figure 2: *Visibility information is stored as a texture. Each texel represents the visibility of a particular block in the grid. In this figure, yellow texels represent the clipped blocks. For better understandability, a texture is depicted as a collection of layers, however it is actually stored as a 2D texture.*

Figure 2 describes how a visibility texture is generated for a par-

ticle system. This texture is passed to a vertex shader program to correctly render the set of visible particles. Following code fragment shows how a clipped particle is eliminated in a vertex shader.

```
float4 VS(float3 InPos  : POSITION) : POSITION0
{
    float w=1.0;
    // visibility check in the object space
    if(notvisible(VisibilityTexture, InPos))
        w=0;
    // transform
    float3 Pos = mul(float4(InPos, 1), (float4x3)World;
    Pos = mul(float4(Pos, 1), (float4x3)View);
    float4 outPos= mul(float4(Pos, 1), Projection);
    // eliminate the particle
    if(clip && w==0)
        outPos.w=w;
    return outPos;
}
```

Elimination of these particles in the vertex shader stage ensures that they do not affect the Z-buffering that is required to correctly render the visible set of particles.

Besides its simplicity, the main advantage of using a visibility texture is that it can handle static as well as dynamic particle systems. Furthermore, visibility textures are updated only when a clipping plane is selected by the user. This avoids regeneration of visibility information for each frame. Unlike previous systems, it allows for specifying multiple clipping planes in different view spaces.

3.3 Compression

we currently use a 128-bit floating point visibility texture due to hardware limitations. However, in practice, we can reduce this storage requirement by using 8-bit or 4-bit integer textures. Furthermore, visibility textures can be compressed by storing a single bit information for each block. This requires extra computations in the vertex shader, but reduces the memory required to store visibility textures for large grids. However, storing just a single bit would not allow for an undo operation as described in the next section.

3.4 Allowing for an Undo operation

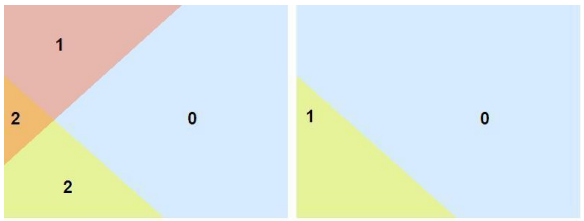

Figure 3: *Left texture shows the visibility information about two cutting planes. Right texture shows the texture after applying an undo operation on it to unselect the upper cutting plane.*

To support an undo operation to *unselect* previous clipping planes, extra information for each block needs to be stored. By storing n-bit information per block, we can support up to 2^n undo operations. We initialize the visibility structure with the value 0. Whenever a clipping plane is selected by the user, corresponding clipped blocks are found and each texel information is incremented by 1. Thus, to allow for an undo operation, we just need to decrement, by 1, each non-zero texel value in the visibility texture. Thus using n bits per block we can store up to the information about 2^n previous cutting planes specified by the user. Using just a single bit information does not allow for such a feature that is required for an accurate selection of a clipping plane. In the current implementation, we use 128 bits for each texel, as the underlying hardware does not support any other texture format for texture lookup in vertex shader. However, this selection is purely dependent on the availability of the resources or complexity of the scenes. Figure 3 depicts how a texture is manipulated during an undo operation.

4 Providing Additional Depth and Structure Cues

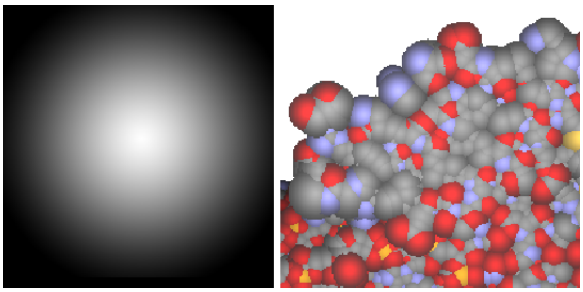

Figure 4: *Depth sprite used to render a particle as sphere. Each texel on this sprite stores a depth information which is used in a pixel shader to approximate current depth of the pixel by using an approximated radius of the sprite to be displayed. Thus, every point on a rendered sprite has a different depth information and this allows for indicating intersections of the spheres.*

As described earlier, particles rendered as imposters or glyphs often produce self occluding structures. These structures are hard to visualize because the depth information and the structure information cannot be provided. A dense molecule with thousands of atoms is a perfect example of such an occluding structure. Tarini et al. [Tarini et al. 2006] provides self-shadows, depth edges, structure edges, halo effect and artistic rendering techniques to visualize complex molecule structures. They use a multi-pass shader to achieve all these effects. However, we try to provide some of these effects by using just a single pass shader that further improves the performance. In addition, we make use of depth sprite to render each particle as a sphere and not just a shaded circle. This allows for storing depth information for each pixel of the sprite which is then used in a pixel shader to approximately render a sphere. Currently, sprites overlap each other and do not provide any information about the intersections of the particles. However, by using depth sprites we can easily visualize these intersections. Figure 4 shows how depth information is stored in a sprite. Artifacts produced due to approximation are clearly seen in this figure.

Usage of depth sprites also allows us to depict *structure edges* by using a single pass shader. Depth sprites are deliberately kept darker at the edge of the circle. This allows for highlighting a boundary of a sphere. A region where many spheres intersect can be considered as a structure. In a structure, edge information is overlapped because of the intersections. However, at the edge of the structure, where there are no intersections, this edge information gets automatically highlighted. This approach allows for highlighting outlines of various structures present in the molecule or any dense particle system. However this method would provide poor quality visualization for sparser particle systems as each particle would eventually get drawn as a sphere with an outline. Quality of our images is comparable to that of Tarini et al. [2006]. The next section describes our results in detail.

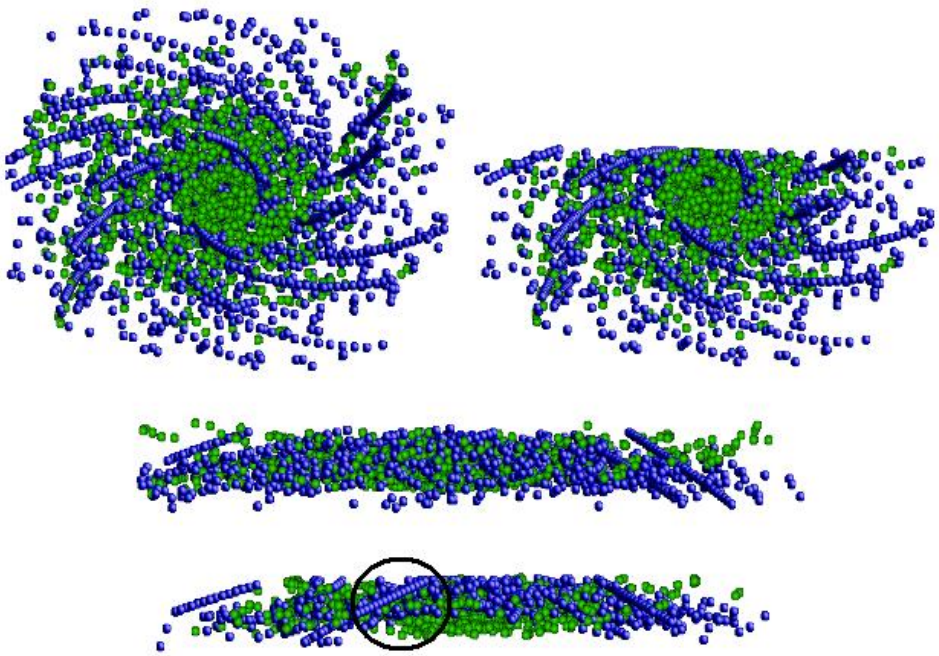

Figure 5: *In the first row, left figure shows an unclipped particle dataset and in the right figure, it is clipped by a simple horizontal plane. Figures below are their corresponding side views to indicate how clipping helps in understanding the internal details of the field. In this particular example, a streak of blue particles (circled) is revealed by clipping.*

5 Results

5.1 Simulation of Hurricane Bonnie : A Dynamic Particle System

We tested our method on the hurricane Bonnie simulation dataset containing around 6,000 particles. We rendered each particle using a point sprite. Figure 5 shows how our method can be used to select a clipping plane in a view space and then observe the details unveiled in another view space.

With this method we could achieve an interactive frame-rate ranging from 20 fps to 200 fps. This frame-rate is varying because the particle dataset is read from the disk per timestep and this I/O operation puts limitations on achieving a constant frame-rate. This particular particle system is not rendered using depth sprites.

5.2 Rendering of molecules : A Static Particle System

We applied our cross sectioning method to complex molecule structures. We obtained these molecule datasets from Protein Data Bank website, http://www.pdb.org. These structures contain from 10,000 atoms to 40,000 atoms in them. We rendered these molecules using depth sprites and achieved upto 20 frames per second and more for molecules with around 10,000 atoms and above 12 frames per second for molecules with around 40,000 atoms. Our performance is limited because of simultaneous render targets used for storing the depth information for selection of the cutting planes as described in the previous section.

Figure 6 shows a cross section obtained for a molecule with around 12,000 atoms. While figure 7 shows cross sections for a molecule with around 28,000 atoms.

6 Conclusion and Future Work

We presented a novel approach for taking cross sections of dynamic particle systems having thousands of particles. Our approach of cross sectioning allows for better exploration of dense particle systems. This is particularly useful in visualizing complex structures like hurricanes and molecules. This method can further be extended to work with any other volumes that can be rendered using particles. We also provided various structure and depth cues using a single pass shader. This method cannot achieve better performance because of the multiple render targets used to store depth information. However, any advanced graphics hardware that supports lockable depth stencil buffers would overcome this disadvantage of our system. In future, we plan to implement compression techniques for visibility structures.

References

BRUCKSCHEN, R., KUESTER, F., HAMANN, B., AND JOY, K. I. 2001. Real-time out-of-core visualization of particle traces. In *PVG '01: Proceedings of the Institute of Electrical and Electronics Engineers 2001 symposium on parallel and large-data visualization and graphics*, Institute of Electrical and Electronics Engineers Press, Piscataway, NJ, USA, 45–50.

CUNTZ, N., KOLB, A., LEIDL, M., REZK-SALAMA, C., AND BÖTTINGER, M. 2007. GPU-based dynamic flow visualization for climate research applications. In *Simulation und Visualisierung 2007 (SimVis 2007)*, 371–384.

ELLSWORTH, D., GREEN, B., AND MORAN, P. 2004. Interactive terascale particle visualization. In *VIS '04: Proceedings of the Institute of Electrical and Electronics Engineers conference on Visualization '04*, Institute of Electrical and Electronics Engineers Computer Society, Washington, DC, USA, 353–360.

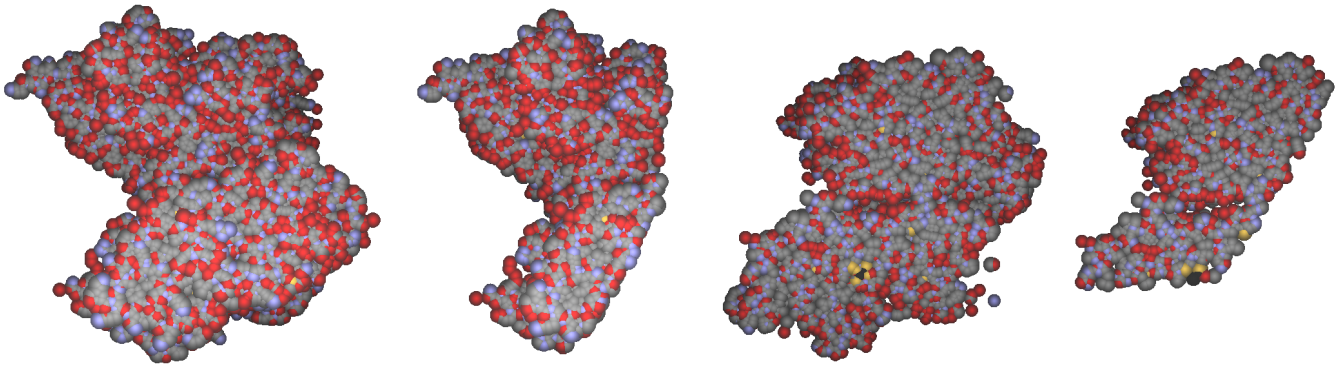

Figure 6: *This figure shows how out cross sectioning can be used on a molecule. Leftmost figure show the occluding structure of a molecule named 2IPY and adjacent figures show cuts taken. In the last to images two yellow molecules and a black molecule are revealed by the cut taken.*

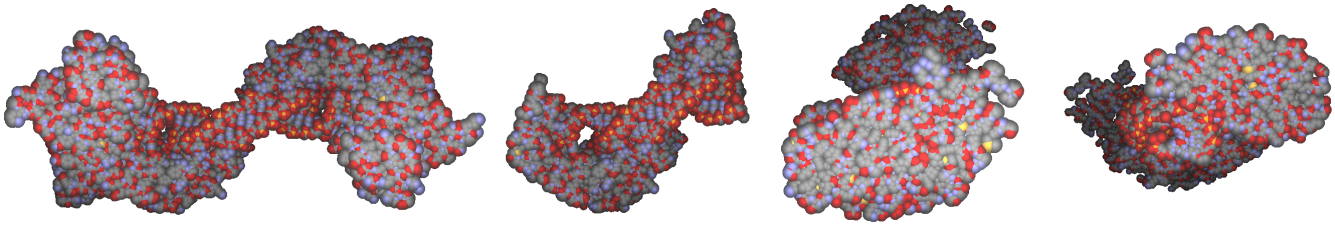

Figure 7: *Cross sections of 2B3Y molecule with 28,000 atoms in it.*

GRIBBLE, C., AND PARKER, S. 2006. Enhancing interactive particle visualization with advanced shading models. In *APGV '06: Proceedings of the 3rd symposium on Applied perception in graphics and visualization*, Association for Computing Machinery Press, New York, NY, USA, 111–118.

GRIBBLE, C. P., STEPHENS, A. J., GUILKEY, J. E., AND PARKER, S. G. 2006. Visualizing particle-based simulation datasets on the desktop. In *Proceedings of the British Human Computer Interaction 2006 Workshop on Combining Visualization and Interaction to Facilitate Scientific Exploration and Discovery*, 1–8.

KIPFER, P., SEGAL, M., AND WESTERMANN, R. 2004. UberFlow: a GPU-based particle engine. In *In Proceedings of the Association for Computing Machinery's Special Interest Group on Graphics /European Association for Computer Graphics conference on Graphics hardware*, Association for Computing Machinery Press, New York, NY, USA, 115–122.

KRUGER, J., KIPFER, P., KONDRATIEVA, P., AND WESTERMANN, R. 2005. A particle system for interactive visualization of 3D flows. *Institute of Electrical and Electronics Engineers Transactions on Visualization and Computer Graphics 11*, 6 (December), 744–756.

LANE, D. A. 1994. UFAT: A particle tracer for time-dependent flow fields. In *VIS '94: Proceedings of the Institute of Electrical and Electronics Engineers conference on Visualization '94*, Institute of Electrical and Electronics Engineers Computer Society Press, Los Alamitos, CA, USA, 257–264.

TARINI, M., CIGNONI, P., AND MONTANI, C. 2006. Ambient occlusion and edge cueing for enhancing real time molecular visualization. *IEEE Transactions on Visualization and Computer Graphics 12*, 5 (September), 1237–1244.

TORY, M., AND MOLLER, T. 2004. Human factors in visualization research. *Institute of Electrical and Electronics Engineers Transactions on Visualization and Computer Graphics 10*, 1 (January), 72–84.

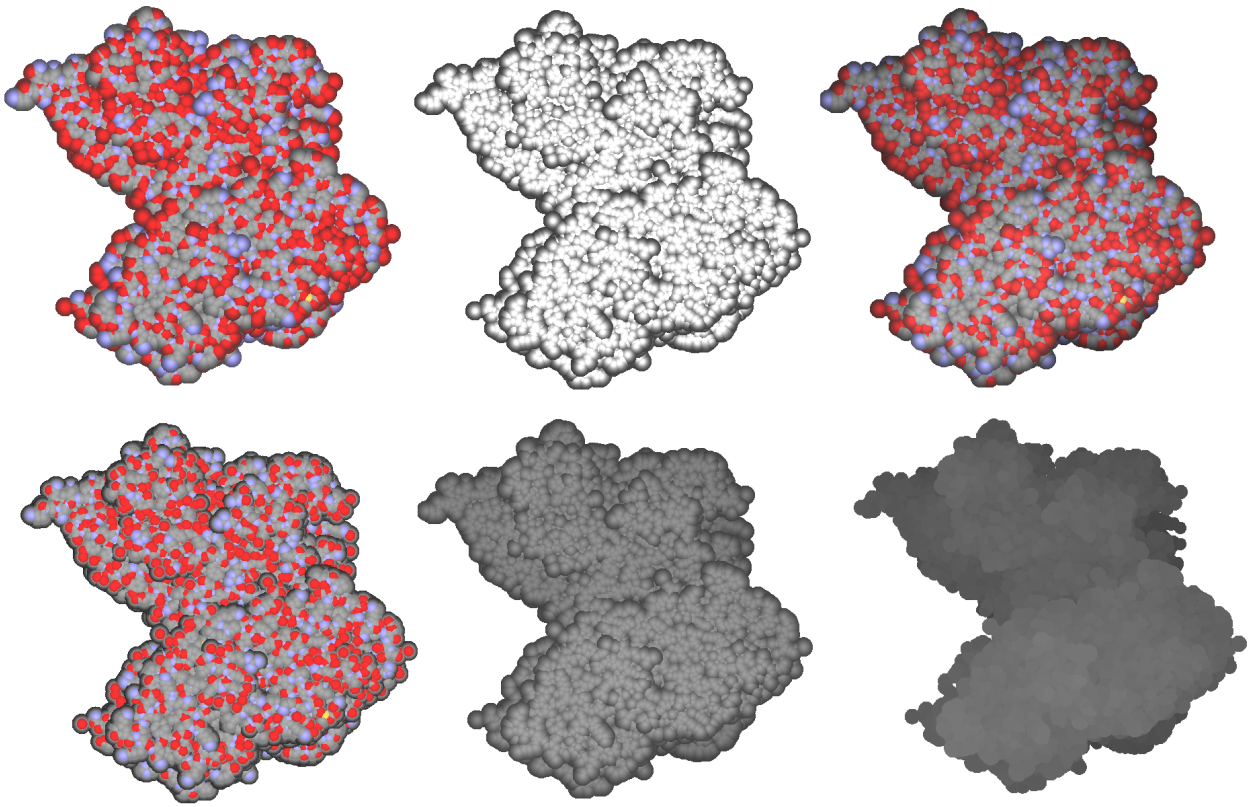

Figure 8: *Various depth and structure cues provided in a single pass shader. Top row, from left to right: original molecule, structure edges obtained due to depth sprites, molecule with depth information. Bottom row, left to right: molecule rendered along with the structure edges, molecule rendered with structure edges as well as depth information on the spheres, lastly the same molecule is rendered to indicate the depth of each sprite.*